Parsing and Querying XML Documents in SML

Dissertation

zur Erlangung des akademischen Grades
des Doktors der Naturwissenschaften
am Fachbereich IV der Universität Trier

vorgelegt von

Diplom-Informatiker
Andreas Neumann

Dezember 1999

The forest walks are wide and spacious;
And many unfrequented plots there are…

The Moor Aaron, in[1]:
The Tragedy of Titus Andronicus
by William Shakespeare

Danksagung

An dieser Stelle möchte ich allen Personen danken, die mich bei der Erstellung meiner Arbeit direkt oder indirekt unterstützt haben.

An erster Stelle gilt mein Dank dem Betreuer meiner Arbeit, Helmut Seidl. In zahlreichen Diskussionen hatte er stets die Zeit und Geduld, sich mit meinen Vorstellungen von Baumautomaten auseinanderzusetzen. Seine umfassendes Hintergrundwissen und die konstruktive Kritik, die er ausübte, haben wesentlich zum Gelingen der Arbeit beigetragen.

Besonderer Dank gilt Steven Bashford und Craig Smith für das mühevolle und aufmerksame Korrekturlesen der Arbeit. Sie haben manche Unstimmigkeit aufgedeckt und nicht wenige all zu "deutsche" Formulierungen in meinem Englisch gefunden. Allen Mitarbeitern des Lehrstuhls für Programmiersprachen und Übersetzerbau an der Universität Trier sei hiermit auch für die gute Arbeitsatmosphäre ein Lob ausgesprochen, die gewiss nicht an jedem Lehrstuhl selbstverständlich ist.

Nicht zuletzt danke ich meiner Freundin Dagmar, die in den letzten Monaten viel Geduld und Verständnis für meine häufige – nicht nur körperliche – Abwesenheit aufbrachte. Ihre Unterstützung und Ermutigungen so wie ihr stets offenes Ohr waren von unschätzbarem Wert.

[1]Found with *fxgrep* using the pattern //`SPEECH`[_ (//`"forest"`) _] in `titus.xml` [Bos99]

Abstract

The Extensible Markup Language (XML) is a language for storing and exchanging structured data in a sequential format. Though originally designed for the use in document processing, it is by now employed for representation of data in virtually all areas of computing, especially on the Internet.

The basis of all XML processing software is an XML parser, which parses a document in XML syntax and exposes it as a document tree to the application. Document processing then basically reduces to tree manipulation. Modern functional programming languages like SML and HASKELL with their tree-structured data-types are therefore well-suited for implementing such applications. Nonetheless, the area of XML processing software is dominated by JAVA. Functional programming languages play a minor role, partly due to the lack of a complete implementation of an XML parser.

One of the most important tasks in document processing is querying, that is the extraction of parts of a document that match a structural condition and are in a specific context. Due to the tree-like view of documents, querying XML can be implemented with techniques from tree language and automata theory. These techniques must, however, be adapted to the needs of XML. One specific requirement is that even extremely large documents must be processed. It must therefore be possible to perform the querying algorithm in a single pass through the document, without the need of constructing a copy of the document tree in memory.

This work is divided into two parts: The first part presents *fxp*, an XML parser written completely in SML. We describe the implementation of *fxp* and discuss our experiences with SML. We analyze the run-time behavior of *fxp* and compare it to XML parsers written in imperative and object-oriented programming languages.

The second part presents an XML querying algorithm based on forest automata theory. The algorithm locates all matches of a query in at most two passes through the document. For an important subclass of queries even a single run suffices. Moreover, we discuss the implementation of the algorithm based on *fxp*. For each of the two parts a separate, more detailed introduction is given.

Zusammenfassung

XML (Extensible Markup Language) ist ein sequentielles Format zur Speicherung und Übermittlung strukturierter Daten. Obwohl es ursprünglich für die Dokumentenverarbeitung entwickelt wurde, findet XML heute Verwendung in nahezu allen Bereichen der Datenverarbeitung, insbesondere aber im Internet.

Jede XML-Dokumentenverarbeitungs-Software basiert auf einem XML-Parser. Der Parser liest ein Dokument in XML-Syntax ein und stellt es als Dokumentbaum der eigentlichen Anwendung zur Verfügung. Dokumentenverarbeitung ist dann im wesentlichen die Manipulation von Bäumen. Moderne funktionale Programmiersprachen wie SML und HASKELL unterstützen Bäume als Basis-Datentypen und sind daher besonders gut für die Implementierung von Dokumentenverarbeitungs-Systemen geeignet. Um so erstaunlicher ist es, dass dieser Bereich zum größten Teil von JAVA-Software dominiert wird. Dies ist nicht zuletzt darauf zurückzuführen, dass noch keine vollständige Implementierung der XML-Syntax als Parser in einer funktionalen Programmiersprache vorliegt.

Eine der wichtigsten Aufgaben in der Dokumentenverarbeitung ist Querying, d.h. die Lokalisierung von Teildokumenten, die eine angegebene Strukturbedingung erfüllen und in einem bestimmten Kontext stehen. Die baumartige Auffassung von Dokumenten in XML erlaubt die Realisierung des Querying mithilfe von Techniken aus der Theorie der Baumsprachen und Baumautomaten. Allerdings müssen diese Techniken an die speziellen Anforderungen von XML angepasst werden. Eine dieser Anforderungen ist, dass auch extrem große Dokumente verarbeitet werden müssen. Deshalb sollte der Querying-Algorithmus in einem einzigen Durchlauf durch das Dokument ausführbar sein, ohne den Dokumentbaum explizit im Speicher aufbauen zu müssen.

Diese Arbeit besteht aus zwei Teilen. Der erste Teil beschreibt den XML-Parser *fxp*, der vollständig in SML programmiert wurde. Insbesondere werden die Erfahrungen mit SML diskutiert, die während der Implementierung von *fxp* gewonnen wurden. Es folgt eine Analyse des Laufzeit-Verhaltens von *fxp* und ein Vergleich mit anderen XML-Parsern, die in imperativen oder objektorientierten Programmiersprachen entwickelt wurden.

Im zweiten Teil beschreiben wir einen Algorithmus zum Querying von XML-Dokumenten, der auf der Theorie der Waldautomaten fundiert ist. Er findet alle Treffer einer Anfrage in höchstens zwei Durchläufen durch das Dokument. Für eine wichtige Teilklasse von Anfragen kann das Querying sogar in einem einzelnen Durchlauf realisiert werden. Außerdem wird die Implementierung des Algorithmus in SML mit Hilfe von *fxp* dargestellt.

Contents

List of Figures

List of Tables

Part I

fxp – An XML Parser Written in SML

Introduction

Since its release in 1998 XML, the Extensible Markup Language, has become one the fastest evolving topics in information technology. As a markup language, XML is a language for storing and exchanging structured data in a sequential format. Though originally designed for the use in document processing, it is by now employed for representation of data in virtually all areas of computing. XML plays its most prominent role on the Internet, where it is expected to replace HTML as *the* World Wide Web markup language in the near future; it is already indispensable in fast developing areas such as eCommerce and Electronic Banking. XML is thus a technology of the future.

The basis of all XML processing software is an XML parser. This is a module or library which establishes the hierarchical structure of a document from its sequential XML representation. Thus it relieves the processing application from dealing with syntactical details: Document processing basically reduces to tree manipulation. The implementation language of an XML application should therefore have good support for processing tree-like data structures. Modern functional programming languages such as SML and HASKELL use trees as the basic data types. With their mechanisms of user-defined data types and pattern matching, they are a good candidate for an implementation language. Nonetheless, the area of XML processing is dominated by software written in JAVA. Until the release of *fxp*, there was no complete implementation of the XML syntax written in a functional programming language.

Functional programming languages are commonly considered as toy languages, designed for research and educational purposes only. Moreover, they are estimated to be inefficient and unsuited for implementing real-world applications. One of our goals was to refute this prejudice and prove the practical usability of functional programming.

As a basis for XML processing in SML, we implemented *fxp*. It is a fully functional XML parser implemented completely in SML, except for network file retrieval. Though written in functional programming style, it employs the imperative features of SML at few but important spots. On the basis of SML's parametrized modules, *fxp* provides a very elaborate and customizable programming interface. Moreover, comparing the execution times of *fxp* and other XML parsers written in imperative languages attests that SML can well compete with the currently most popular programming language in the XML area, namely JAVA. *fxp* is included in the XML software delivered with [Gol99].

This part is organized as follows: Chapter 1 gives an introduction to XML and an overview of XML processing software. The next chapter describes the implementation of *fxp* and points out the difficulties and conveniences we experienced with SML. Finally, Chapter 3 compares *fxp* to other XML parsers and discusses the implementation language.

Einführung

Seit der Einführung von XML (Extensible Markup Language) im Jahre 1998 vollzieht sich auf diesem Gebiet eine rasante Entwicklung. XML ist eine Auszeichnungssprache (Markup Sprache) für strukturierte Dokumente und dient der sequentiellen Speicherung und Übermittlung strukturierter Daten. Zwar wurde XML ursprünglich für den Einsatz in der klassischen Dokumentenverarbeitung entwickelt. Es wird jedoch heute in nahezu allen Bereichen der Datenverarbeitung verwendet. Das wichtigste Einsatzgebiet von XML ist das Internet: Es wird voraussichtlich schon in naher Zukunft HTML als *die* Markup-Sprache des World Wide Web ablösen. Schon heute ist XML aus solch zukunftsversprechenden Bereichen wie dem elektronischen Handel nicht mehr wegzudenken.

Die Grundlage jeder XML-Verarbeitungs-Software ist ein XML-Parser. Das ist ein Modul oder eine Bibliothek, welche die hierarchische Struktur eines Dokuments aus der sequentiellen XML-Darstellung wiederherstellt. Der Parser nimmt damit der eigentlichen Anwendung die Behandlung syntaktischer Details ab: Dokumentenverarbeitung ist dann im wesentlichen die Manipulation baumartiger Datenstrukturen. Die Implementierungssprache einer solchen Anwendung sollte daher die Verarbeitung baumartiger Daten gut unterstützen. Moderne funktionale Programmiersprachen wie SML und HASKELL verwenden Bäume als Basis-Datentypen und bieten Konzepte wie Benutzer-definierte Datentypen und Pattern-Matching. Sie sind daher besonders gut als Implementierungssprache geeignet. Um so erstaunlicher ist es, dass funktionale Sprachen im Bereich von XML kaum eine Rolle spielen: Vor dem Erscheinen von *fxp* gab es noch nicht einmal einen vollständigen XML-Parser in einer funktionalen Sprache. Statt dessen wird der XML-Markt von JAVA-Software beherrscht.

Eine weit verbreitete Ansicht ist, dass funktionale Programmiersprachen reine Experimentier-Sprachen seien, die nur für Zwecke der Forschung und der Lehre bestimmt sind. Auch herrscht die Meinung, sie seien ineffizient und ungeeignet für die Realisierung von Anwendungen unter reellen Anforderungen. Eines unsere Ziele war es mit diesem Vorurteil aufzuräumen und den Beweis zu erbringen, dass funktionale Programmierung tatsächlich gut in der Praxis anwendbar ist.

Als Grundlage für die XML-Verarbeitungs in SML entwickelten wir den Parser *fxp*, der bis auf die Netzwerk-Kommunikation vollständig in SML implementiert ist. Trotz eines funktionalen Programmierstils verwendeten wir allerdings auch die imperativen Bestandteile von SML, wenn auch an wenigen begrenzten Stellen. Auf der Basis von SML's parametrisierten Modulen bietet *fxp* eine sehr raffinierte und anpassbare Programmierschnittstelle. Der Geschwindigkeitsvergleich mit anderen XML-Parsern, die in imperativen Programmiersprachen geschrieben wurden, zeigt, dass SML den Vergleich mit der wohl populärsten Programmiersprache im XML-Bereich, nämlich JAVA, nicht scheuen muss. *fxp* wurde in die Reihe der Programme aufgenommen, die zusammen mit dem Buch [Gol99] ausgeliefert werden.

Chapter 1

Document Processing with XML

Documents usually have a hierarchical logical structure: A book, e.g., is divided into chapters which are themselves made up of sections that consist of subsections, and so on. Modern document processing systems therefore have a tree-like view of documents; document processing then basically reduces to tree manipulation.

In order to store and exchange structured documents, these must be brought into a sequential representation. This is achieved by inserting *markup* into the text, indicating the start and end of each logical component. XML, the Extensible Markup Language [W3C98b], is a standardized syntax for such markup developed and introduced for use on the Internet by the World Wide Web Consortium (W3C).

The basis of all XML processing software is an XML parser. This is a module or library which is aware of the XML syntax and can reproduce the tree structure of a sequentially represented document.

This chapter gives an introduction to the concepts of XML and discusses requirements and characteristics of XML processing software, particularly XML parsers. The next chapter will then present *fxp*, an XML parser written in the functional programming language SML.

1.1 Introduction to XML

As a markup language, XML provides a syntax for sequential representation of structured documents: Each logical component of a document, called *element*, is enclosed between a pair of specific markers, its *start-tag* and *end-tag*, which are easily distinguishable from the text. The *content* of an element is the sequence of elements and text enclosed between its tags. Additional properties of the element can be specified in its start-tag by a set of *attribute assignments*.

In this, XML is similar to HTML [W3C98e], the markup language of the World Wide Web. HTML, however, is restricted to a fixed set of *element types* such that, e.g., mathematical formulae can not be expressed in HTML. SGML [ISO86, Gol90], the predecessor of XML, addresses this deficiency by providing a mechanism for defining the markup vocabulary, i.e., the set of admissible element types, in the preamble of the document, which is called *document type* *declaration* (DTD). Each element type is declared along with a rule restricting the form and order of the contents of elements of that type. It is therefore not only a markup language but also a meta-language for defining markup languages such as HTML.

SGML includes, however, many historical features that make parsing of documents difficult and expensive. For this reason the World Wide Web Consortium introduced XML as a simplification of SGML. It provides the extensibility of SGML while retaining the syntactical simplicity of HTML, making it suitable for easy data exchange over the Internet. XML is therefore expected to replace HTML as *the* markup language of the World Wide Web within the next few years. Indeed, HTML is currently being reformulated as an XML document type [W3C99g].

This section introduces the basic concepts and the syntax of XML. A more comprehensive tutorial is [Bra98a, Gol99]; technical details are very well explained in [Bra98b].

1.1.1 Elements, Attributes, and the Document Type

In XML, each element of a document has an *element type*, e.g., `section` or `footnote`, a number of *attributes* and a *content* consisting of other elements or *character data*. An element of type `a` is enclosed between a start-tag `<a>` and an end-tag `</a>`. An empty element `<a></a>` can be abbreviated by a special form `<a/>`, an *empty-element tag*. If any attributes are present, they are specified in the start-tag. As an example, consider an entry of a bibliographic database:

XML Example 1

```
<bibentry id="XML:1998">
  <author org="Netscape">Tim Bray</author>
  <author org='Microsoft'>Jean Paoli</author>
  <author>C. M. Sperberg-McQueen</author>
  <title>Extensible Markup Language (XML) 1.0</Title>
  <publ month="feb" year="1998">W3C Recommendation</publ>
</bibentry>
```

This element has type `bibentry` and an attribute named `id` that identifies it among other `bibentry` elements and may serve as a reference to this element. It contains five elements: three of type `author`, one of type `title` and one of type `publ`, all of which contain only character data. The `publ` element has two attributes `month` and `year` indicating the date of appearance, and two of the `author` elements have an attribute `org` containing the author's affiliation. Be aware that, though these are the intended interpretations of the elements and attributes, XML associates no meaning with them: Its view of a document is solely syntactical.

In order to constrain the content of elements, XML assigns a *content model* to each element type by an *element-type declaration*. A content model is a regular expression over element types and the special word `#PCDATA` representing plain text, i.e, a possibly empty sequence of non-markup characters. For the `bibentry` example, the elements might be declared as follows (note that concatenation is denoted by the comma sign "`,`" in content models):

XML Example 2

```
<!ELEMENT bibentry (author+,title,publ)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT publ (#PCDATA)>
```

In addition to the content model, the attributes of an element type must be declared with an *attribute type* and a *default value*. For the bibentry example, the attribute declarations are as follows:

XML Example 3

```
<!ATTLIST bibentry  id    ID      #REQUIRED>
<!ATTLIST author    org   CDATA   "University of Illinois">
<!ATTLIST publ      year  NMTOKEN #REQUIRED
                    month (jan|feb|mar|apr|may|jun|
                           jul|aug|sep|oct|nov|dec) #IMPLIED>
```

A value for an attribute of type ID must be a name, and it must uniquely identify its element, i.e., the whole document must not contain another attribute of that type with the same value. Values for attributes of type CDATA may consist of arbitrary characters, whereas NMTOKEN attribute values must be name tokens, i.e., consist of alphanumeric characters only. Attribute month has an *enumeration type*: Its value must be one of the listed name tokens. For a complete listing of the possible XML attribute types see Table 2.1.

An attribute can be assigned a default value: This value is substituted by the parser whenever the attribute is omitted in a start-tag. Instead of a default value, the keyword #IMPLIED means that the attribute may be omitted but its value must then be generated by the application. #REQUIRED means that the attribute must always be specified. A fourth possibility, not appearing in the example, is a *fixed* default value: If the attribute is omitted, the default is used, but if specified the attribute value must be equal to the default value.

An XML document mainly consists of a *document type declaration* (DTD) followed by the *document instance* which is a single top-level element, also called the *root element* or *document element*. The DTD contains all declarations needed for the instance and it must have the same name as the type of the root element:

XML Example 4

```
<!DOCTYPE book [
  <!ELEMENT book ...
]>
<book>...</book>
```

Large document types are often stored in separate files. In this case the DTD specifies a *system identifier* indicating a file to be included additionally:

XML Example 5

```
<!DOCTYPE book SYSTEM "book.dtd" [
  ...
]>
```

The contents of this file are called the *external subset* of the DTD, whereas the declarations specified directly between "`[`" and "`]`" constitute its *internal subset*.

1.1.2 Entities

Entities are a mechanism for modularization of a document. In the DTD, entity names can be associated with a *replacement text*. Upon *reference* of an entity its replacement text is included into the document. Entities are classified into *general* and *parameter* entities as follows: General entities are dedicated to contain pieces of content. They can, with a few exceptions, only be used in the document instance. A general entity is declared as follows:

XML Example 6

```
<!ENTITY author "Andreas Neumann">
```

In the instance it can be referenced using the characters "`&`" and "`;`":

XML Example 7

```
This report was written by &author;.
```

On the other hand, parameter entities are dedicated for use in declarations; they are not available in the document instance. A parameter entity is declared and referenced using the character "`%`". E.g., in order to define several element types with the same content model, one could write:

XML Example 8

```
<!ENTITY % cont "(bold|emph|#PCDATA)*">
<!ELEMENT title %cont;>
<!ELEMENT caption %cont;>
<!ELEMENT footnote %cont;>
```

In addition to the classification into general and parameter entities, we further distinguish between *internal* and *external* entities: Internal entities have a sequence of characters as their replacement text. Similar to C preprocessor macros, they serve as abbreviations or definitions of frequently used text fragments. Unlike C macros, entities have no arguments: Their replacement text is constant. E.g., both entities `author` and `cont` from XML Examples 6 and 8 are internal entities.

As opposed to that, external entities facilitate modularization of documents: The replacement text of an external entity is the content of a file. It is declared with the `SYSTEM` keyword followed by a system identifier giving the location of the file. E.g., in order to modularize the DTD, one might put all declarations related to math formulae into a separate file `math.dtd`, and write in the DTD:

XML Example 9

```
<!ENTITY % math SYSTEM "math.dtd">
%math;
```

The declarations in this file can thus be shared between different DTDs. Note that the specification of an external subset at the beginning of the DTD (cf. XML

Example 5) can be viewed as an abbreviation for an external entity declaration followed by a reference to that entity. We can also use external identifiers for general entities:

XML Example 10

```
<!ENTITY resume SYSTEM "resume.xml">
Here is the author's resume:
&resume;
```

The physical distribution of a document over external entities must, however, match its logical structure. XML requires for each logical component, like attribute values, elements, declarations or comments, that its first and its last character are in the same entity (as a special case of an entity, the file containing the DTD and the root element is called the *document entity*).

Entities also allow inclusion of non-XML data into XML documents. Non-XML entities are called *unparsed* and declared with the `NDATA` keyword. Because an XML parser can not process non-XML data, unparsed entities must be associated with a *notation*. Each notation is assigned an application that can process its data by a *notation declaration* in the DTD:

XML Example 11

```
  <!NOTATION jpeg "/usr/local/bin/view">
  <!ENTITY portrait SYSTEM "portrait.jpg" NDATA jpeg>
```

Unparsed entities may not be referenced directly. They may only appear as the value of an attribute with type `ENTITY` or `ENTITIES`.

1.1.3 Public Identifiers and Catalogs

Because XML was designed for information exchange over the Internet, the system identifier in an external entity declaration need not point to a file on the local file system. It can also specify a document somewhere on the net. The system identifier is therefore interpreted as a *uniform resource identifier* (URI) [IET98a]. This allows for distributing a document over the net:

XML Example 12

```
<!ENTITY chapter1 SYSTEM "http://www.company.de/chap1.xml">
<!ENTITY chapter2 SYSTEM "ftp://ftp.company.fr/pub/chap2.xml">
...
<book>
&chapter1;
&chapter2;
</book>
```

The Internet, however, is a "moving target": Documents frequently change their locations. Moving one part of a document without changing the referring parts would therefore break the document's integrity. On the other hand, if one part of the distributed document is stable and not expected to be changed later, one can reduce net bandwidth by holding local copies of that part. Since such a copy is usually only temporary, one would like to avoid changing the reference in the document itself. Instead, XML provides the concept of public identifiers. A *public identifier* uniquely identifies an XML document or entity regardless of its physical location. Using public identifiers when declaring external entities therefore keeps the document independent of physical storage:

XML Example 13

```
<!ENTITY chapter1 PUBLIC
  "-//Company//Chapter 1//EN" "chapter1.xml">
```

The public identifier `"-//Company//Chapter 1//EN"` is now used to determine the location of the first chapter, and only if that fails the system identifier `"chapter1.xml"` is used. But how can a public identifier be resolved to a file name? A favored method is using an XML *catalog* [Cow99]. A catalog is a special XML document interpreted as a mapping from public identifiers to system identifiers. In our example, the catalog would contain the following element:

XML Example 14

```
<Map PubId = "-//Company//Chapter 1//EN"
      HRef = "http://www.company.de/chap1.xml"/>
```

XML catalogs are derived from a form of catalogs used with SGML [SO97]. Frequently, this syntax is used for XML as the SOCAT syntax of catalogs.

1.1.4 Miscellaneous Markup

Since some characters are reserved for markup, they may not appear literally at certain places in an XML document: E.g., the "<" character may not appear in the document instance unless it initiates a start-tag. If such a character is needed as character data, it must be entered as a decimal or hexadecimal *character reference*, in this case either "`&#60;`" or "`&#x3C;`". For larger parts of text containing reserved characters, a *CDATA section* can also be used. It is started with "`<![CDATA[`" and ended with "`]]>`"; no other markup is recognized within it.

XML Example 15

```
The greater-than sign can be entered as &#60; or &#x3C;.
A character reference has the form <![CDATA[ &#60; ]]>.
```

Comments are enclosed between "`<!--`" and "`-->`". Comments are not part of the documents content but contain useful information for human readers of the XML document.

XML Example 16

```
<!-- Must check spelling -->
Is this the road to Edinborough?
```

A special form of a comment is a *processing instruction*: It is not part of a document's content, but unlike comments it may contain valuable information for a processing application. In addition to its text, a processing instruction comprises a *target name* indicating the application it is aimed at.

XML Example 17

```
A misspelt word is
<?spell check off?>
Edinborough
<?spell check on?>
```

In this example, both processing instructions have the target `spell`.

1.1.5 UNICODE

XML documents are written in UNICODE [Uni96] which is a 21-bit character set containing most of the characters occurring in the written languages of the world. Since most operating systems support only 8-bit characters, UNICODE documents – and thus also XML documents – are usually encoded as 8-bit character streams. Various methods, e.g., ASCII, LATIN1, UTF-8, UTF-16 or UCS-4 [ISO98, IET92, IET98b, Uni96], are used for encoding XML documents. If an XML entity is not encoded in UTF-8, then it must start with an *XML* or *text declaration*. These are special processing instructions with target `xml` and incorporate an *encoding declaration*:

XML Example 18

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE thèse [ ...
```

This XML declaration indicates that the document is encoded in the LATIN1 character set using 8-bit characters, thus it is possible to use character "è" directly in a name.

1.1.6 Well-Formedness, Validity and Compatibility

XML distinguishes several degrees of how a document can conform to the XML specification: All documents must at least be *well-formed*. Well-formedness basically requires that the document is syntactically correct and all entities referenced in the document instance are declared in the DTD. In addition to being well-formed, a document should also be *valid*. Validity requires that the document instance obeys all declarations made in the DTD. For instance, the content of each element must match the content model declared for its element type, and all attribute values must be according to their declared types.

Validity also includes *compatibility* with XML's predecessor SGML. The intention is that each valid XML document should also be a valid SGML document. This leads to some obfuscating restrictions which go back to design decisions taken for SGML. When SGML appeared in the middle of the 80's, the circumstances were clearly different from today's:

- Most computers were equipped with only a small amount of main memory. Moreover, hard disks were small and networks were slow. Therefore it appeared essential for commercial acceptance to reduce the number of characters consumed by the markup to an absolute minimum.

- ✧ The markup was usually inserted manually into documents; there were no tools like syntax-directed or WYSIWYG (what-you-see-is-what-you-get) editors which can nowadays generate the markup automatically. Decreasing the amount of the markup therefore corresponded to reducing the authoring efforts.

As a consequence, SGML offers many possibilities for omitting or abbreviating the markup, wherever this does not obscure the logical document structure. Some of these features are the following:

- ✧ If an attribute has an enumeration type, then the attribute name may be omitted in the attribute specification. E.g., if the element `title` has an attribute `align` with type `(left|center|right)`, then its start-tag `<title align="center">` can be abbreviated to `<title center>`. Unfortunately, this abbreviation requires a restriction on the types of an element's attributes: The same value may not occur for different attributes (otherwise the attribute name can not be determined from the attribute value). The disadvantage is that an element can not have two attributes with type, e.g., `(yes|no)`.
- ✧ Content models can be implemented by deterministic finite automata(DFA). Constructing a DFA for a given content model generally requires a subset construction, eliminating non-determinism. But the subset construction possibly generates exponentially many states, thus consuming huge amounts of memory for the transition tables. Therefore SGML claims that content models must have a special form that avoids the subset construction: They must be *unambiguous*. Basically, unambiguity means that a symbol in the input can never match two different symbols in the regular expression without lookahead. Unambiguity is decidable in polynomial time (see, e.g., [BW92, Neu97]), but it is not at all intuitive to non-computer scientists. E.g., the content model `(a,(c,a)?)*` is unambiguous whereas `((a,c)?,a)*` is not.

XML abolished most of the historical features of SGML. In order to be compatible with SGML, however, the restrictions induced by these features were adopted for XML, some of them as *validity constraints* and others as *interoperability* recommendations only.

1.2 XML Software

Conceptually, processing of XML documents is divided into three stages: parsing of the input document, processing of that document, and generation of output. The processing application itself is not aware of the XML syntax and the physical representation of the documents. It has only a logical view of the document structure. In order to access XML documents, it needs an XML parser. This is a module or a library that can parse and possibly validate XML documents. Through a programming interface which is either tree-based or event-based, the parser makes the document available to the application. The application then processes the document and, in most cases, produces a different document as output. In order to store its output, the application makes

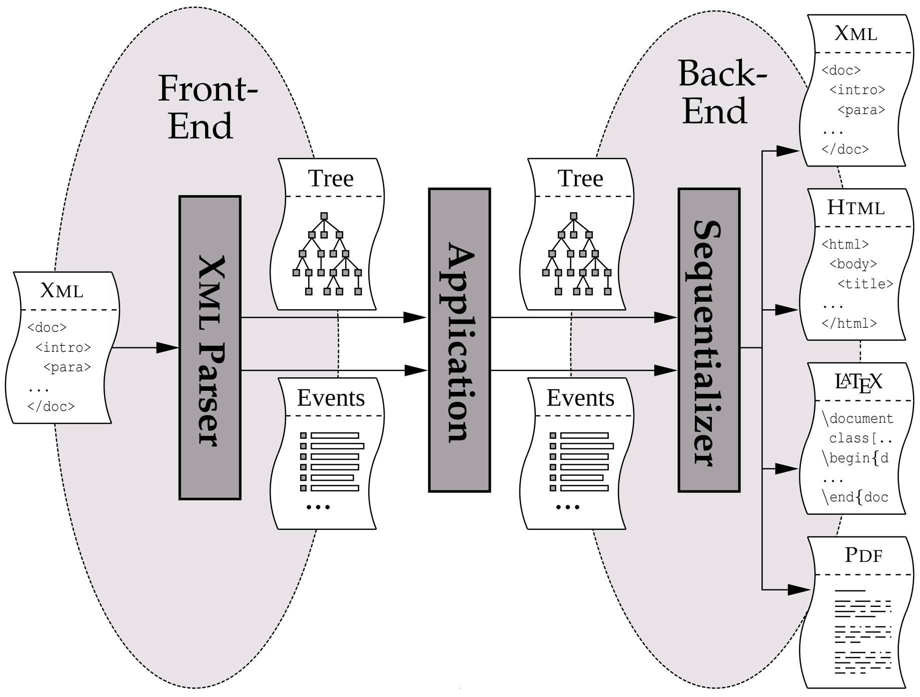


Figure 1.1: A model of XML processing.

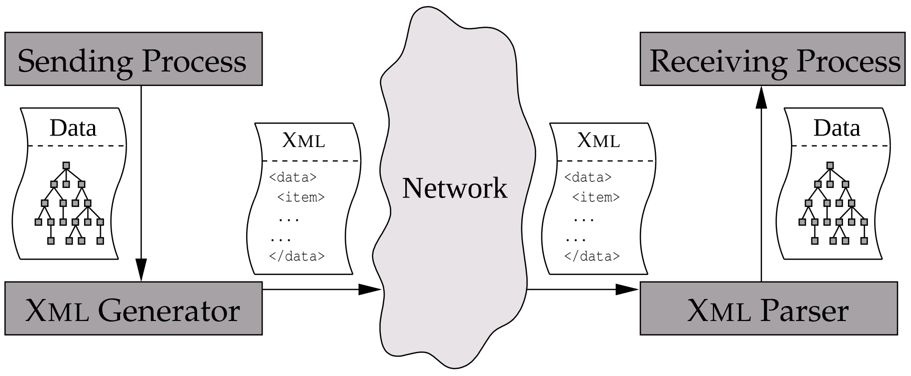


Figure 1.2: Using XML for network information transport.

use of a library generating a sequential representation of the output document. This need not necessarily be in XML, it might also be in a different format.

Figure 1.1 illustrates this model of XML processing. Of course, it does not cover all kinds of XML software. E.g., an application might produce no output document at all, but only check a property of its input such as validity. Other applications might involve multiple processing stages before returning an output, or obtain the input in a non-XML format.

Another aspect is that XML is designed for platform-independent exchange of information over networks. Since arbitrary structured data can be expressed in XML, it is also used for information transport by applications beyond the document processing area. Figure 1.2 illustrates this practice: In order to send a packet of data over a network, the data is packed into an XML document before transmission. The receiving agent unpacks the data from its XML representation with the help of an XML parser. Similarly, XML is a popular format for intermediate representation or permanent storage of data.

1.2.1 Areas of XML Processing

One of the simplest XML applications is a well-formedness checker. It generates no output but parses the document and reports violations of XML's well-formedness constraints. A *validator* additionally checks for validity.

An XML *transformer* is an application that takes one or more XML documents as input and generates one or more XML documents as output. In the simple case, the input is enriched by adding some information. Examples are the compilation of a table of contents or the generation of implied attributes such as figure numbers. While the output usually conforms to the same DTD as the input, more complex transformations might also restructure the document according to a different DTD.

A *converter* brings its input into a different format, e.g., SGML or LaTeX, or converts it from a non-XML format into XML. Similarly, a *recoder* reads an XML document and reproduces it in a different character encoding. Converting to a different format may require a transformation of the document if the output format is less or more expressive than the input format.

A *formatter* renders a document into a representation suitable for reading, printing or publishing. The formatting process typically starts with a transformation, annotating the document tree with additional information. The document is then formatted into a sequence of page descriptions or another suitable representation, which is finally displayed on a viewing medium or stored in a page description language. A more detailed discussion of the formatting process is given in, e.g., [ISO96, W3C99e].

Another significant class of XML applications are *querying tools*. These applications search and extract occurrences of a pattern in an XML document. Querying is a vital operation of many other applications, e.g., for selecting parts of the document to be transformed. In the context of document databases, querying plays a prominent role in information retrieval. Part II will describe a querying algorithm in detail.

There are many other XML applications in the classical document processing area. On the other hand, XML's capability of representing arbitrary – even binary – data highlights another application area of increasing importance: XML is widely used for data storage and exchange in virtually all areas of information technology. Let us only mention a few of them:

Electronic Commerce: The Bank Internet Payment System (BIPS) [FST98] uses XML for transmitting bank transactions over the Internet. Similarly, the cXML specification [Ari99] defines a set of DTDs for performing eCommerce transactions across the Internet.

Software Modeling: Diagrams of the Unified Modeling Language (UML) are described and exchanged using the UXF format [SY98], which is an XML DTD.

Computer Graphics: The Precision Graphics Markup Language (PGML) [AAC+98] is an XML-based format for describing scalable arbitrary precision graphics, capable of modeling POSTSCRIPT and PDF.

Distributed Computing: XML is used as an exchange format for JAVA Beans [Joh99], which are JAVA software components. The Object Model Group (OMG) uses XML for transmitting data in the context of CORBA [OMG99].

1.2.2 Commercial software

XML's application areas include Web publishing, document databases and document processing in general. This opens a large commercial market to XML software developers. Therefore the major part of XML software is commercial. In order to try out a commercial software, one can commonly obtain a free demo or evaluation version. This version, however is either restricted in functionality or limited to processing only small documents. Moreover, in most cases the documentation is not provided with the demo version, making it difficult to test advanced features of the software. Our knowledge about most commercial XML software is therefore limited and unsatisfactory.

Another aspect is that commercial developers compete with each other: A good algorithm provides the basis for outperforming the competitors. Technical know-how is therefore kept as a secret, and documentation never includes a description of the algorithms employed. Furthermore, program sources are not made available to the public: The software is only available in binary, i.e. executable, format and can frequently only be run with an operating system that is widely spread on the commercial market, namely Microsoft Windows.

In the area of XML parsers, the situation is slightly better than for other XML software. *sp* [Cla98], one of the first available XML parsers, was originally written as an SGML parser. It is a free, open-source software written in C++ and is the inofficial reference implementation for SGML. Since XML is – with a few exceptions – a subset of SGML, only slight changes were required for making *sp* a fully functional XML parser. Since *sp* is free software and has a general-purpose – yet unfriendly – programming interface, commercial competitors could not expect profits from releasing non-free parsers. Therefore, nearly all XML parsers are free and, in most cases, open-source software.

Of course, *fxp* is also open-source. Moreover it is implemented in a platform-independent programming language. Though it was developed under UNIX, porting it to other platforms should be easy – as far as there are SML compilers for these platforms, which is the case for most modern operating systems including Microsoft Windows.

1.2.3 XML Application Interfaces

XML parsers expose the parsed document to the application through a programming interface. Conceptually, one can distinguish two kinds of interfaces: *tree-based* and *event-based* ones. In the tree-based approach, the parser constructs the document tree during parsing and passes this entire data structure to the application when parsing is finished.

In contrast to that, in the event-based case the information is fed piecemeal to the application: Each syntactical component of the document, e.g. a start-tag or a piece of character data, triggers an event. The application can register a *handler* for each type of events, which is called by the parser when an event is triggered. The handler can then process the information signaled by the event and change the state of the application accordingly.

The main advantage of a tree-based interface is that the application can arbitrarily navigate through the document tree and copy or modify parts of it. This is especially useful in complex applications that have to access each part of the tree multiple times. The disadvantage is that a copy of the document must be completely constructed in memory before the application can access it. For large documents, this consumes a huge amount of memory, probably impairing the application's cache and memory behavior, thus slowing it down.

Therefore, an event-based interface is preferable for applications that do not depend on the physical presence of the document tree. The document can be processed "on the fly", and only the required parts are stored in memory by the application. E.g., compiling a table of contents for a book can be done by collecting the section titles while discarding all other information.

Through an event-based interface, the application has a serialized view of the document: It can only be inspected once in a predetermined order. The application can, however, use the events for generating its own copy of those parts that have to be traversed several times or in a different order. A tree-based interface can therefore always be provided on top of an event-based one.

For both kinds of interfaces, a standardization has been attempted in order to increase the interoperability of different XML parsers. For the tree-based case, the most commonly used abstract data type is the Document Object Model (DOM), standardized by a W3C Recommendation [W3C98a]. Though the DOM specification claims to be "designed to be used with any programming language", it is committed to the imperative object-oriented programming paradigm: It defines class interfaces for accessing XML document trees and (destructively) manipulating them. Similarly, there is a widespread – though not officially standardized – event-based interface called SAX [Meg$^+$98]. Like DOM, SAX defines class interfaces for registering event handlers in object-oriented programming languages.

A variant of the event-based approach is a *demand-driven* interface: Instead of registering handlers, the application repeatedly calls the parser in order to obtain the next event. This requires that the state of the parser is explicitly available – and not implicitly, e.g., by the current recursion stack. The advantage is that the control is on the application-side: It can, e.g., process several documents simultaneously, or stop processing a document after having retrieved the desired information. Though occasionally requested in discussion forums and mailing lists, to the best of our knowledge none of the available XML parsers provide a demand-driven interface.

1.2.4 The Information Set

XML parsers are also classified by the extent of the information they pass to the application. This is called the parser's *information set*. The XML recommendation requires that all parsers must provide at least a minimal information set. This includes all information that is necessary for generating a structurally equivalent copy of the document instance. Moreover, the parser must report all violations of the XML recommendation to the application; for each of these errors the parser must indicate whether it is a well-formedness, i.e., fatal or only a validity error.

The XML recommendation does not require reporting of comments, the encoding of entities, and the declarations in the DTD. It also disregards the physical distribution of the document among its entities, i.e., the starts and ends of included entity references need not be indicated. But an application might need this information: E.g., a spell checker produces an output document with all typing errors corrected. It would be annoying if the output had to be in a single file though the input was well modularized and spread among several files.

In order to to overcome this deficiency, the W3C is working on a recommendation defining XIS, a standard information set for XML [W3C99d]. It divides the information into information items, classified into required and optional ones. The required items are mainly those demanded by the XML recommendation; the optional items include comments, declarations and boundaries of included entity replacement text. XIS has no mechanism of reporting errors, nor does it provide information about the physical location, such as the document's URI. But this is valuable information: In order to be user-friendly, error messages produced by an application should report the position in a file where an error occurred. Similarly, if a querying tool extracts some information from a large document, the user might be interested in the location of the source of that information.

Moreover, XIS is not sufficient for generating a character-by-character identical copy of the input. The reason is that it abstracts from the syntactical representation of the markup. E.g., the information items for the two start-tags `<a x="1" y="2">` and `<a y='2'    x='1'>` would be identical because the specification order, the quote sign used for the value, and the white space between the attributes have no influence on the meaning of the start-tag.

For the readability of documents, however, these properties are important. E.g., if the character used to quote the attribute value occurs in the value itself, then this character can not be entered directly – a character reference must be used instead. Clearly, an attribute value `"O'Hara"` is preferable to `'O&apos;Hara'`. Therefore, an application might desire to preserve these properties of a document. Another example is the encoding of the document: If an application generates XML output the user probably wants the output to have the same encoding as the input because this might be the standard encoding used in his company. The application must therefore be aware of the input encoding.

1.2.5 Further Characteristics of XML Parsers

A *conforming* XML parser must be able to parse an XML document and check it for well-formedness. Through its application interface it must signal errors and report a minimal subset of the document's content to the application. It need not include external entities – processing the document entity alone is sufficient. A parser may, however supply additional features, some of which include:

Validation: A validating parser processes the complete document type declaration including external parameter entity references. It must check whether all declarations conform with XML and report all errors to the application. It parses the document instance, including external entities, and validates it against the DTD. This includes checking the attribute values for compliance with their declared types and checking whether the content of each element is according to the content model declared for it. Validation also includes checking for compatibility and, optionally, interoperability with SGML (see 1.1.6).

Reentrance: A reentrant parser allows several instances to be run simultaneously but independent of each other. E.g., XML catalogs used to resolve public identifiers are XML documents themselves (see 1.1.3). During parsing of the main document, a new instance of the parser must thus be started in order to read the catalog. Since the catalog has its own DTD, the data structure in which the DTD is stored may not be shared (e.g., as a global variable). Instead, each instance of the parser must have its own copy of this data structure. Similarly, the options for different parser instances may differ: E.g., catalog support should be disabled when parsing a catalog. Most of the available XML parsers are reentrant, though early versions of many parsers were not.

Catalogs: XML catalogs are, by now, only supported by very few parsers. Many parsers that were derived from existing SGML parsers, however, support the alternative SOCAT syntax for catalogs defined by the SGML Open Consortium for use with SGML (cf. 1.1.3).

In addition to catalogs, some parsers support various standards defined on top of XML, some of which are XML Namespaces [W3C99a], XLINK [W3C98d] and XPOINTER [W3C99f].

1.2.6 Existing XML Parsers

The most commonly used programming language in the XML area is JAVA [GJS96]. Among other reasons [Fuc99], this is due to the fact that XML is designed for data exchange on the Internet; and JAVA is designed for Internet programming. Moreover, JAVA programs are portable to arbitrary operating systems provided they have an implementation of the JAVA Virtual Machine (JVM). On the other hand, XML documents are written in UNICODE. Currently, the only programming language with full built-in UNICODE support is JAVA. Therefore JAVA is a natural choice for many developers. The most important XML parsers in JAVA include:

xp [Cla99b] is a non-validating parser written by the author of *sp*. Its goal is to be the fastest XML parser in JAVA;

xml4j [IBM99] is a validating parser from IBM. It includes support for catalogs and implements DOM and SAX. It is probably the most extensive implementation of XML available.

JAVA as an interpreted language, however, is known to be slow in comparison to compiled languages like C or C++. Aside from the already mentioned *sp*, several XML parsers have been implemented in C, e.g.:

expat [Cla99a] is written by the author of *xp* and *sp*. It can do only slightly more than checking for well-formedness, but it is extremely small and fast;

rxp [Tob99] is a validating, but still very fast parser. Catalogs are not supported.

On the other hand, scripting languages play an important role on the Internet. These are most commonly used for CGI-programming on Web servers. Scripting languages are usually interpreted languages. They are well-suited for fast prototyping and implementation of small programs. They are, however, rather inefficient and inappropriate for large applications. The most popular scripting languages are TCL/TK [Ous94], PERL [WCS96] and PYTHON [Lut96]. For all of these languages, foreign language interfaces to the *expat* parser have been implemented [WC99, PXS99, Bal99]. For PYTHON, there is also a native XML parser:

xmlproc [Gar99] is a validating parser written entirely in PYTHON. It provides a SAX and a DOM interface and supports catalogs in XML and SOCAT syntax. It lacks, however, full UNICODE support and omits some required validity checks. It is therefore not a conforming XML parser.

All parsers mentioned so far are implemented in imperative programming languages. In the functional programming area, only recently two projects emerged. Both were initially released in January 1999:

tony [Lin99] is a non-validating parser written in OCAML [LRV$^+$99]. It is an incomplete implementation of XML in that it lacks support for full UNICODE and several XML features like CDATA sections.

HaXml [WR99] is a parser written in HASKELL using combinator-style parsing. Like *tony*, it does not implement UNICODE and validation.

fxp, which had its first release in February 1999, has full UNICODE support and implements all requirements of the XML recommendation. It is therefore the first complete XML parser written in a functional programming language.

Chapter 2

Implementation of *fxp* in Standard ML

Due to XML's tree-like view of documents, the implementation language of an XML processing application should provide good support for defining and manipulating tree-structured data types. A good choice of implementation language are modern functional programming languages, like SML [MTH+97] or HASKELL [JHA+98], because they use trees as the basic supported type and have an easy and intuitive mechanism for user-defined tree data structures.

This chapter presents *fxp*, an XML parser written completely in the functional programming language SML. It reads and validates XML documents and provides an application programming interface for processing documents. It is thus a basis for implementing XML applications like formatters, converters or querying tools in SML. *fxp* supports catalogs in both syntaxes; it is reentrant, highly customizable, and provides an event-based, functionally flavored application interface.

We first motivate our choice of programming language and give an overview of *fxp*'s system architecture. We then describe the individual system components. Following that, we analyze the efficiency of *fxp* and compare it to XML parsers written in other programming languages.

2.1 The Implementation Language – SML

fxp is written in SML (Standard Meta Language) [MTH+97, Pau96], a strict non-pure functional programming language. Our goal was to provide an XML parser written completely in a functional language. *fxp* shows that the functional programming style is a good alternative to the object-oriented style widely adopted in the XML area. Beside this rather idealistic motivation, there are several other good reasons for choosing SML as implementation language:

Tree Types: Imperative programming languages use pointers for constructing complex data types and iteration for traversing them. In contrast to that, modern functional programming languages like SML or HASKELL have a tree-like view of data structures. Traversal of large data structures is implemented by recursion. Data-type constructors and *pattern matching* provide a convenient mechanism for manipulating tree-structured data while abstracting – from the programmer's point of view – from the physical representation. This makes programming more comfortable and less error-prone. Moreover, the tree-like view of data structures very closely matches XML's view of documents.

Polymorphism: SML supports polymorphic types and functions. A polymorphic type is a generic type (e.g., a list) parametrized by one or more types. It represents a data type containing data of the argument types, providing a single implementation for all instantiations of the argument types. Analogously, a polymorphic function is a function that works on a polymorphic type and does not depend on the argument types; as the type itself, the polymorphic function needs only be defined once. A popular example for a polymorphic function is list concatenation. Polymorphism thus gives high succinctness and reusability to the code.

Non-Pure Features: In pure functional languages like HASKELL the concept of explicit state does not exist. As a consequence, there is no destructive update on variable values. Furthermore, there is no purely functional concept of input and output because these depend on the state of the world in which a program is run. I/O is therefore provided through rather non-intuitive concepts such as streams or monads. This can render programs unintelligible and makes debugging of purely functional programs a difficult task.

SML is a non-pure functional language: It has a concept of imperative, i.e., side-effected input and output – a feature which turns out to be essential for an XML parser, which must access entities on the local file system and on the network. SML also supports mutable types such as references and arrays. Though not essentially necessary for an XML parser, mutable types ease the straight-forward and efficient implementation of frequently used data structures. A symbol table, for instance, is most naturally implemented as a hash table using an array with constant-time, destructive update.

fxp uses mutable types in few but frequently used functions, with the effect of significantly speeding up the program. If possible, the imperative nature of functions is hidden behind a module interface. The program source code has therefore a very functional appearance.

Strictness: SML is a strict functional language. Strictness means that arguments of a function are evaluated before the function is called. In contrary to that, in non-strict languages evaluation of an argument is delayed until its value is actually needed by the computation.

Strictness is a vital prerequisite for supporting non-pure features like mutable types and imperative I/O. Non-strictness, however, does not seem to be of any advantage for XML parsing: Because the processing order of XML documents is prescribed – at least during parsing – by its sequential representation, non-strict evaluation can only obfuscate the evaluation order. Moreover, non-strict evaluation requires additional efforts from the languages run-time system, slowing down program execution. On the other hand, SML's strictness allows for easy debugging.

Parametric Modules: SML has a very sophisticated module system. It features a first-order mechanism for parametrized modules: so-called *functors* (an unparametrized module is called *structure*, its interface *signature*). A functor is a structure parametrized with other structures. The definitions in its body depend on values defined in the parameter structures. The functor can be applied to actual argument structures, making these visible in the functors body in order to create a new structure, an *instance* of the functor. A good example for the use of functors are the dictionaries in Section 2.6.1. Functors have two main advantages: They make the code highly reusable and are a very elegant means of making it customizable.

Basis Library: With the definition of the SML Basis Library [Rep97], SML has a comprehensive, platform-independent interface to many common operating system functions. SML programs are therefore extremely portable.

fxp was developed using the SML of New Jersey (SML/NJ) compiler [Bel99]. It translates the source code into native code which, however, can only be run with the support of a run-time system. SML/NJ provides many useful features such as execution profiling and a sophisticated mechanism of separate compilation [Blu97].

There are several other implementations of SML which are, however, inappropriate for our purposes. They include Moscow ML [Ses99] and MLj [BKR99] both of which do not implement SML functors and are therefore not usable for *fxp*. The MLTON compiler [Wee99] translates SML to C. It does, however, not support separate compilation. Finally, there is a commercial product, MLWorks™ from Harlequin. Its development seems to be still in an early stage: Though it has a separate compilation mechanism, it can not figure out module dependencies. The programmer has to add annoying dependency annotations to the program sources which make them unusable with other compilers. Moreover, the compiler is annoyingly slow, while the generated code does not seem better than that of SML/NJ. We therefore chose SML/NJ for the development of *fxp*.

2.2 System Architecture

fxp's system architecture is shown in Figure 2.1. It has three main components:

Frontend: The frontend is divided into two separate modules: the UNICODE frontend and the entity manager.

The UNICODE frontend implements the physical access to the input document and all included external entities. Moreover, it performs the decoding into UNICODE.

The entity manager maintains the stack of open entities and provides the parser with its input. It decodes external entities with the help of the UNICODE frontend.

DTD manager: This module maintains in a bunch of tables the declarations of the document type. It contains as a separate module the implementation of the tables in a concrete data structure. The DTD manager reports all errors that occur to the application.

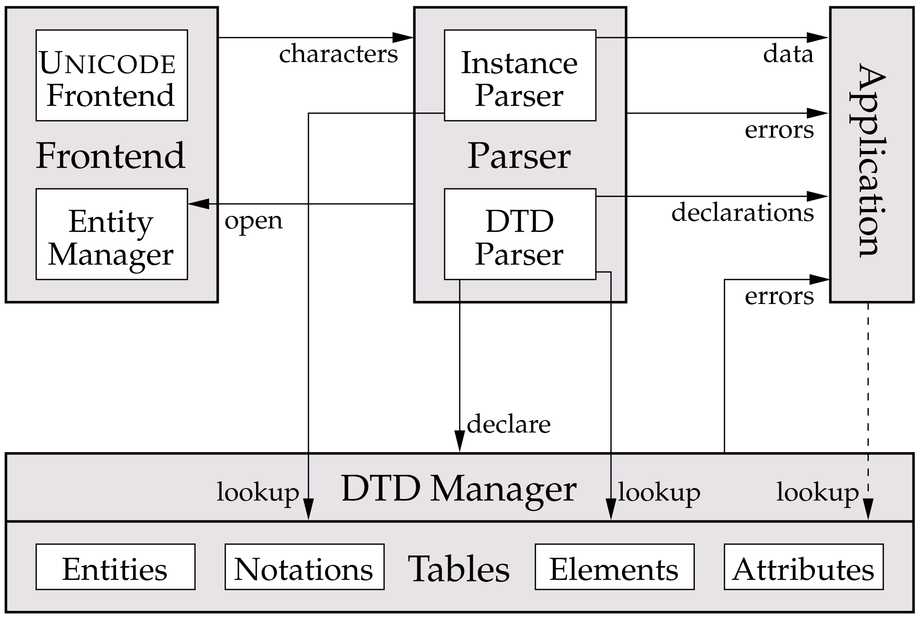


Figure 2.1: The system architecture of *fxp*.

Parser: The main parser module is divided into two parts: the DTD parser which processes the DTD with all its declarations and feeds them to the DTD manager, and the instance parser which processes the content of the document instance. The DTD parser fills the DTD tables with information, whereas the instance parser accesses these tables mostly for obtaining information. Both parser modules report individual parts of the document as well as errors to the application.

Figure 2.1 also shows a fourth component: the application. This is the document processing program that uses *fxp* as a frontend to the XML syntax. It is thus not part of *fxp*.

The following sections document the individual components of *fxp*. We illustrate the employed programming techniques by extracts from the source code which are simplified in order to ease description: Obfuscating details such as error handling are omitted in most cases. For simplicity, we first describe the parser without the programming interface. In Section 2.8 we then describe the programming interface and its implications on the other components.

2.3 The UNICODE Frontend

UNICODE is an international multi-byte character set capable of representing most written languages of the world. XML is based on UNICODE in order to make it language-independent. SML, however, supports only 8-bit characters and has no notion of UNICODE. Therefore, *fxp*'s UNICODE frontend provides types for UNICODE characters and strings, along with basic functions for manipulating them.

XML documents are encoded into byte-streams in order to maintain them

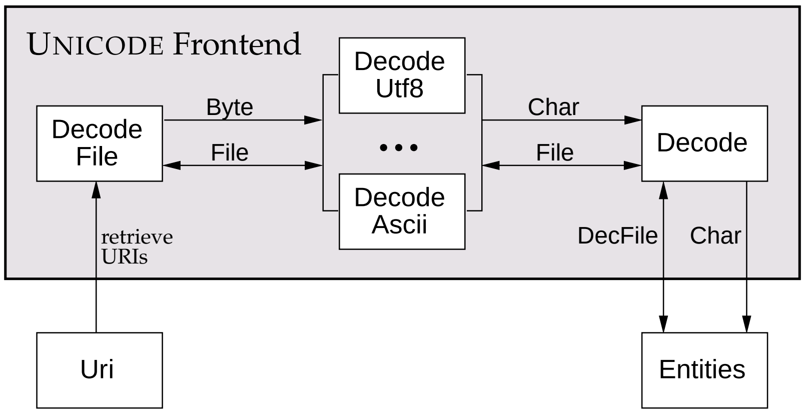


Figure 2.2: Overview of the UNICODE frontend.

in operating system files. A large number of character encodings is used with XML. *fxp*'s UNICODE frontend supports the most frequently used encodings. It provides a type for associating a file with a character encoding, together with functions for reading UNICODE characters from encoded files.

An overview of the UNICODE frontend is given in Figure 2.2. It has mainly three stages:

File operations: The module DecodeFile implements the access to the file system. It defines a type File, together with functions for opening and closing files and reading bytes from them. Access to files over the network happens through the module Uri, which retrieves a remote file and stores it on the local file system.

Main module: The main module of the UNICODE frontend is Decode. It defines a type DecFile which is a File tagged with its encoding. This type is visible to the entity manager, whereas the type File and all operations on the byte-level are hidden from the outside. In order to retrieve a character from a DecFile, the decoding function for the associated encoding is called.

Another task of the structure Decode, which is not captured by Figure 2.2, is the detection of a file's encoding upon opening of the file (cf. 1.1.5). Only after this is performed the file can be tagged with its encoding.

Decoding modules: For each of the supported character encodings there is a module implementing the decoding, e.g., DecodeUtf8 for the UTF-8 encoding. On request it reads a number of bytes through the module DecodeFile and decodes them to a single UNICODE character.

Figure 2.3 summarizes the interface of the UNICODE frontend by means of the signature Decode. The substructure Error provides a type for reporting decoding errors together with some auxiliary functions. Moreover, two exceptions are defined for indicating that either an end of file or an error occurred. Functions decUri, decName and decEncoding return the URI, the file name or the encoding of a DecFile.

```
signature Decode =
  sig
    structure Error : DecodeError

    type DecFile
    type Encoding

    exception DecEndOfFile of DecFile
    exception DecError of DecFile * Error.DecodeError

    val decUri      : DecFile → Uri.Uri
    val decName     : DecFile → string
    val decEncoding : DecFile → Encoding

    val decOpenXml : Uri.Uri option → DecFile
    val decClose    : DecFile → DecFile

    val decCommit   : DecFile → unit
    val decSwitch   : DecFile * string → DecFile

    val decGetChar  : DecFile → UniChar.Char * DecFile
    val decGetArray : DecFile → UniChar.Char array
                        → int * DecFile * Error.DecodeError option
  end
```

Figure 2.3: The Decode signature.

The functions decOpenXml and decClose are for opening a URI for reading and for closing a file. Upon opening of a file its encoding must be determined. This detected encoding, however, need not be identical with the one specified in the encoding declaration – it suffices if both are compatible. After the encoding declaration has been parsed, the parser must therefore either confirm the auto-detected encoding with decCommit, or switch to the declared encoding through function decSwitch. For a detailed description of this procedure, see 2.3.7.

The function decGetChar reads a UNICODE character from a DecFile and returns it together with the modified file. As an optimization, a function decGetArray is also provided. This function fills a whole array with UNICODE characters (see 2.3.6).

We will now describe the components of the UNICODE frontend in detail.

2.3.1 Basic Types

In its first version, UNICODE used 16 bits for describing a single character and was thus capable of representing 65536 characters. With version 2.0 [Uni96], the *surrogates*, an additional set of – yet unused – about one million characters, was added. In order to describe surrogates by means of 16-bit characters, two areas of 1024 characters were designated as *high-surrogates* and *low-surrogates*. These characters may only appear as a combination of a high-surrogate followed by a low-surrogate, together representing a surrogate character. In encodings capturing sufficiently many bits per character, a surrogate character may also appear directly.

The range of UNICODE characters is illustrated in Figure 2.4. The largest hexadecimal value a UNICODE character can have is thus 0x10FFFF, requir-

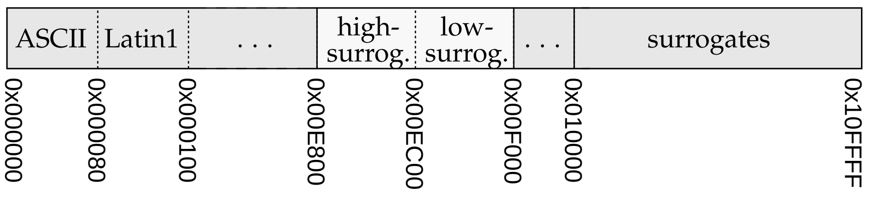


Figure 2.4: Distribution of UNICODE characters.

```
signature UniChar =
   sig
      structure Chars : WORD

      type Char = Chars.word
      type Data = Char list
      type Vector = Char vector

      val nullData      : Data
      val nullVector    : Vector

      val hashChar      : Char → word
      val hashData      : Data → word
      val hashVector    : Vector → word

      val compareChar   : Char ∗ Char → order
      val compareData   : Data ∗ Data → order
      val compareVector : Vector ∗ Vector → order

      val Char2Uni      : Char → string
      val Char2String   : Char → string
      val Data2String   : Data → string
      val Vector2String : Vector → string

      val Data2Vector   : Data → Vector
      val Vector2Data   : Vector → Data
end
```

Figure 2.5: The UniChar signature.

ing 21 significant bits to represent it. We therefore use the built-in SML type Word.word for implementing UNICODE characters. In all SML implementations we know of, this type represents words of at least 30 bits. The Word structure also qualifies because it provides the bit-manipulation and arithmetic operations required for decoding UNICODE.

The UniChar structure declares the types to be used for UNICODE and provides some basic functions on these types. Its signature is given in Figure 2.5: Substructure Chars must match the predefined WORD signature and provides the type for UNICODE characters together with bit-manipulation and arithmetic operations on this type. In addition to Char, two types are defined for representing UNICODE strings: Data and Vector. The latter is used for more efficient representation of constant strings. Constants nullData and nullVector represent the empty string; functions Data2Vector and Vector2Data convert between the two string representations.

For each of the three types defined, we also provide a function that hashes

```
signature UniClasses =
   sig
      val isName    : UniChar.Char → bool
      val isNms     : UniChar.Char → bool
      val isPubid   : UniChar.Char → bool
      val isS       : UniChar.Char → bool
      val isDec     : UniChar.Char → bool
      val isHex     : UniChar.Char → bool
      val isXml     : UniChar.Char → bool
      val isUnicode : UniChar.Char → bool

      val decValue  : UniChar.Char → UniChar.Char option
      val hexValue  : UniChar.Char → UniChar.Char option
   end
```

Figure 2.6: The UniClasses signature.

a value of this type to a word, a function that compares two values, and a function that returns a string representation of a value. For Char there is an additional function Char2Uni which returns the string "U+xxxx" where xxxx is the hexadecimal code of the character. These functions are useful for implementing symbol table, sorted lists and generation of error messages.

2.3.2 Character Classes

A character class is a set of characters that belong to a certain category, e.g., letters or digits. Structure UniClasses, whose signature is given in Figure 2.6, provides functions testing for membership in character classes. isHex and isNms, e.g., implement the classes *Hex* of hexadecimal digits and *Nms* of characters that can start a name. Class *Hex* is small and simple: It only contains the letters "A" to "F", "a" to "f" and the digits "0" to "9". It is implemented as follows:

```
fun isHex c =
   c⩾0wx30 andalso c⩽0wx39 orelse (* 0-9 *)
   c⩾0wx41 andalso c⩽0wx46 orelse (* A-F *)
   c⩾0wx61 andalso c⩽0wx66        (* a-f *)
```

Additionally we need a function that computes the numeric value of a hexadecimal character. This is a partial function defined only for hexadecimal digits; its return value is therefore of type Char option[1]:

```
fun hexValue (c:UniChar.Char) =
   if c⩾0wx30 andalso c⩽0wx39 then SOME(c−0wx30)      (* 0-9 *)
   else if c⩾0wx41 andalso c⩽0wx46 then SOME(c−0wx37) (* A-F *)
   else if c⩾0wx61 andalso c⩽0wx66 then SOME(c−0wx57) (* a-f *)
   else NONE
```

In contrast to that, class *Nms* is very large and complex to describe since it is spread in small portions over the whole UNICODE range: It consists of 13614 characters distributed among 206 intervals of UNICODE characters, plus two intervals of about 20000 and 11000 characters in the area of CJK idioms and Hangul syllables. We can not implement this class with a nested **if** expression

[1]The option type is predefined in SML as follows: **datatype** 'a option = NONE | SOME of 'a.

because that would be far too inefficient. On the other hand, a table containing all *Nms* characters would be very large. As a compromise, we implement only the 206 small intervals with a table:

```
fun isNms c =
    if c < 0wx4000 then inCharClass(c,nmsClass)
    else c ⩾ 0wx4E00 andalso c ⩽ 0wx9FA5 orelse (∗ CJK idioms      ∗)
         c ⩾ 0wxAC00 andalso c ⩽ 0wxD7A3         (∗ Hangul syllables ∗)
```

nmsClass is an array of 512 32-bit words characterizing each character between 0wx0000 and 0wx3FFF with a single bit. Function inCharClass checks whether the bit for a particular character is set:

```
fun inCharClass(c,class) =
    let val idx   = Chars.toInt(Chars.≫(c,0w5))
        val mask = Word32.≪(0wx1,Word.andb(Chars.toWord c,0wx1F))
    in Word32.andb(mask,Array.sub (class,idx)) <> 0wx0
    end
```

The character's five lowest bits are used for generating a one-bit mask; its other bits determine the position of a word to apply this mask to. Note that if the character is larger than 0x3FFF, this position is beyond the range of the array. However, this can never happen because nmsClass is visible only to function isNms.

The array nmsClass is a mutable data structure; this implementation is thus not purely functional. The reason is that nmsClass must be initialized in some way. This is most easily achieved by consecutively setting the bits corresponding to the characters in the class. In a vector, this would be impossible because it does not support destructive update[2]. The use of an array is therefore indispensable. However, updates on the array are performed only during initialization. This takes place at compile-time; the run-time behavior is thus purely functional.

2.3.3 Access to the File System and the Network

XML system identifiers are URIs which come in two variants: An *absolute* URI points to a fixed location, either on the local file system or on the Internet, whereas a *relative* URI specifies a location relative to that of the file referring to it. The URI syntax has many technical details; let us only briefly summarize it: An absolute URI starts with a *scheme* part describing the access method (e.g. `http:` or `file:`), followed by an optional host name and an absolute path to the location. An absolute path starts with a "/"; its components are separated with the same symbol. In a relative URI, the scheme and host parts are omitted, and the path needs not start with a "/". Here are some example URIs:

```
http://www.informatik.uni-trier.de/˜neumann/Fxp/
file:/usr/doc/w3c/xml-1.0.xml

/reports/1999.html
chapter1.xml
```

[2]In SML, vectors are an immutable variant of arrays.

```
signature Uri =
  sig
    eqtype Uri
    val emptyUri   : Uri

    val hashUri    : Uri → word
    val compareUri : Uri ∗ Uri → order

    val Data2Uri   : UniChar.Data → Uri
    val Vector2Uri : UniChar.Vector → Uri
    val String2Uri : string → Uri
    val Uri2String : Uri → string

    val uriJoin    : Uri ∗ Uri → Uri
    val retrieveUri : Uri → string ∗ string ∗ bool
  end
```

Figure 2.7: The Uri signature.

Though URIs are specified by arbitrary UNICODE strings in XML, they may only contain ASCII characters. Non-ASCII characters must be encoded in order to specify them as part of a URI; XML recommends to use UTF-8 for this encoding. The URI "`/home/müller/data.xml`", e.g., must be represented as "`/home/m%C3%BCller/data.xml`" because the UTF-8 encoding of character "ü" is the hexadecimal string "C3BC". Note that this is incompatible with the escaping mechanism widely used on the Internet that would encode the URI as "`/home/m%FCller/data.xml`".

URIs are implemented by the structure Uri, whose signature is given in Figure 2.7. It defines an equality type Uri and provides functions Data2Uri, Vector2Uri and String2Uri for converting from the different string formats to URIs while encoding non-ASCII characters. Uri2String decodes a URI into a string, but drops all characters requiring more than 8 bits. It is mainly used for reporting errors. Function uriJoin combines two URIs into a new one: The first one is treated as an (absolute) base URI, the second one as relative.

Function retrieveUri fetches a URI from the Internet if it is not on the local file system. It returns three values: a string representation of the URI, the name of a local file containing the URI's content, and a boolean indicating whether that is a temporary file into which the contents of the URI were downloaded. In this case, it must be deleted upon closing.

```
fun retrieveUri uri =
  case uriLocal uri
    of NONE      ⇒ retrieveRemote uri
     | SOME path ⇒ (Uri2String uri,path,false)
```

If the URI points to the local file system, i.e., has the form "`file:<path>`", then uriLocal returns that path, which is used as the file name. Otherwise the URI is downloaded by a call to retrieveRemote:

```
signature DecodeFile =
  sig
    structure Bytes : WORD

    type File
    type Byte = Bytes.word

    exception EndOfFile of File

    val Char2Byte  : UniChar.Char → Byte
    val Byte2Char  : Byte → UniChar.Char
    val Byte2Hex   : Byte → string

    val openFile   : Uri.Uri option → File
    val closeFile  : File → unit

    val getByte    : File → Byte ∗ File
    val ungetBytes : File ∗ Byte list → File

    val fileUri    : File → Uri.Uri
    val fileName   : File → string
  end
```

Figure 2.8: The DecodeFile signature.

```
fun retrieveRemote uri =
  let val tmp    = OS.FileSys.tmpName()
      val cmd    = substitute retrieveCommand (uri,tmp)
      val status = OS.Process.system cmd
      val _      = if status = OS.Process.success then ()
                   else let val _ = OS.FileSys.remove tmp
                        in raise NoSuchFile (uri, "command "^cmd^" failed")
                        end
  in (Uri2String uri,tmp,true)
  end
```

First, the Basis Library function OS.FileSys.tmpName supplies a name that can be used for a temporary file. A system command for retrieving the URI is generated from a template string retrieveCommand. It contains the strings "%1" and "%2" which are replaced with the URI and the temporary file by the function substitute. A sensible value for this template[3] is, e.g., "urlget -s -f -o %2 %1". Executing the system command stores the URI in the temporary file. If the command fails, that file is removed and an exception is raised, indicating that the URI could not be retrieved.

Note that *fxp* does no network communication itself: All access to the network is through system commands. This would not be possible in pure functional programming style, but it relieves us from the efforts of implementing Internet protocols as part of *fxp*.

2.3.4 Byte-Stream Operations

The UNICODE frontend reads documents encoded as byte-streams and decodes them to UNICODE. The access to these byte-streams is through the structure

[3]Since we use a system command here, the value of this template varies for different platforms. It is the only parameter that must be customized when installing *fxp*.

DecodeFile whose signature is given in Figure 2.8. It provides a type File that uniformly represents URIs, whether present permanently or only temporarily on the local file system. As a special case, a File can also represent the standard input; this makes it possible to use the output of another system command as input to the parser through an operating system pipe. The structure also provides a type Byte, together with a structure Bytes for bitwise and arithmetic operations on that type.

If openFile is called with (SOME uri) as argument, then it opens the given URI with the help of retrieveUri. If its argument is NONE, then openFile opens the standard input. A File is closed with closeFile; if it is a temporary file created by retrieveUri, this file is removed. fileUri and fileName return the URI described by the file and its string representation.

A single byte can be read from a file with getByte. This function closes the file and raises EndOfFile if no more bytes are available. On the contrary, function ungetBytes undoes reading of some bytes: They are inserted before the current position in the file such that subsequent calls to getByte will see these characters again. This is needed for implementing auto-detection of a URI's character encoding (cf. Section 2.3.7).

Note that though structure DecodeFile is implemented on top of the imperative TextIO structure from the SML Basis Library, its interface imitates functional behavior: Operations on files return the modified file as a component of their return value. This eases implementation of ungetBytes: Whereas reading from a file can be done imperatively with the TextIO functions, the reverse operation is not supported by that structure.

2.3.5 Decoding into UNICODE

fxp supports the most widely used character encodings for XML including ASCII, LATIN1, UTF-8, UTF-16 and UCS-4. With each encoding, a decoding function is associated which reads a single UNICODE character from a byte stream. For ASCII, this is the function getCharAscii:

```
val getCharAscii : DecodeFile.File → UniChar.Char * DecodeFile.File
```

ASCII is the 7-bit encoding used by most operating systems and Internet protocols. It is very simple: Only the first 128 UNICODE characters are representable, and a character is represented by its lowest byte. getCharAscii only needs to check whether the next byte is a valid ASCII character. If it is not, it raises the exception DecodeError:

```
fun getCharAscii f =
    let val (b,f1) = getByte f
    in if b<0wx80 then (Byte2Char b,f1)
       else raise DecodeError(f1, ERR_ILLEGAL_CHAR(b,"ASCII"))
    end
```

DecodeError has two arguments: the file where the error occurred and an error description. Note that the file argument is necessary because the interface of DecodeFile is non-imperative. In order to continue reading bytes after handling the exception, a File argument is required, describing the state of the file after the error occurred.

0-7 bits:	0 x_7 ... x_0			
8-11 bits:	110 x_{10} ... x_6	10 x_5 ... x_0		
12-16 bits:	1110 x_{15} ... x_{12}	10 x_{11} ... x_6	10 x_5 ... x_0	
17-21 bits:	11110 x_{20}...x_{18}	10 x_{17} ... x_{12}	10 x_{11} ... x_6	10 x_5 ... x_0
22-26 bits:	111110 x_{25} x_{24}	10 x_{23} ... x_{18}	· · ·	10 x_5 ... x_0
27-31 bits:	1111110 x_{30}	10 x_{29} ... x_{24}	· · ·	10 x_5 ... x_0

Figure 2.9: The UTF-8 character encoding.

The LATIN1 encoding, also known as ISO-8859-1, differs from ASCII only in that it supports the first 256 UNICODE characters; EBCDIC is a one-byte encoding with a different character order than LATIN1. UTF-16 and UCS-4 encode a character by splitting it into a multi-byte sequence of fixed length.

2.3.5.1 Decoding UTF-8

In order to illustrate how complex decoding can be, let us consider the UTF-8 encoding. In contrast to most other encodings, it represents a character with a variable number of at most six bytes. The number of bytes required for encoding an individual character depends on how many significant bits the character has (at most 31). Figure 2.9 illustrates how the bits are distributed among the individual bytes: Each byte consists of a *byte-mark* (the higher bits) and some data bits. The first byte-mark indicates how many bytes are used in total; all subsequent bytes have the byte-mark 10. We implement this with a vector byte1switch. At compile-time, this vector is initialized for each byte value with the number of UTF-8 bytes indicated by its byte-mark. Byte-mark 10 is invalid for the first byte of a UTF-8 sequence, indicated by a 0. This vector is used by the getOneUtf8 function:

```
fun getOneUtf8 f =
  let val (b,f1) = getByte f
  in case Array.sub(byte1switch,Word8.toInt b)
       of 0 => raise DecodeError(f1,ERR_ILLEGAL_UTF8 b)
        | 1 => (Byte2Char b,f1)
        | n => getBytes(b,f1,n)
  end
```

If the first byte is invalid, an appropriate exception is raised; if it indicates a single-byte character, it is converted to the Char type and returned. Otherwise, more bytes must be read with the help of function getBytes. Note that although 4 bytes are always sufficient for encoding the – at most 21 – bits of a UNICODE character, UTF-8 can encode up to 31 bits. We must therefore be able to handle sequences of more than 4 bytes, even if the result is not a valid UNICODE character.

Function getBytes accumulates the value of the result character in parameter w of its auxiliary function doit. It repeatedly shifts that value, reads another byte and adds it to the character value. The byte-mark is not cleared before adding a byte. Instead, the accumulated effect of all byte-marks is subtracted in a single operation from the final value. This effect depends only on the number of bytes read and can therefore be precomputed and stored in vector diffsByLen at compile-time.

```
fun getBytes(b,f,n) =
   let fun doit (w,f,m) =
      if m ⩾ n then (w,f)
      else let val (b,f1) = getByte f handle exn as EndOfFile f
                                      ⇒ raise DecodeError(f,ERR_EOF_UTF8)
               val w1 = if Bytes.andb(b,0wxC0)=0wx80
                        then Chars.≪(w,0w6)+Byte2Char b
                        else raise DecodeError(f1, ERR_ILLEGAL_UTF8 b)
               in doit (w1,f1,m+1)
               end
      val (w,f1) = doit (Byte2Char b,f,1)
      val diff = Vector.sub(diffsByLen,n)
      val c = w−diff
   in if isUnicode c then (c,f1)
      else raise DecodeError (f1,ERR_NON_UNI_UTF8 c)
   end
```

If a byte can not be read because the end of the file is reached, or if its byte-mark is different from 10, an appropriate exception is raised. Because UTF-8 can represent non-UNICODE values, the computed character code must be checked for being valid UNICODE before returning it.

Having decoded one UTF-8 encoded character, the work is not yet done: This character might be part of a surrogate pair. Therefore we must check whether it is a high-surrogate. In case it is, the following low-surrogate is read and the surrogate pair is combined to the result UNICODE character:

```
fun getCharUtf8 f =
   let val (c1,f1) = getOneUtf8 f
   in if isHighSurrogate c1
      then let val (c2,f2) = getOneUtf8 f1 handle EndOfFile f
                                      ⇒ raise DecodeError(f,ERR_EOF_SURROGATE)
           in if isLowSurrogate c2
              then (combineSurrogates(c1,c2),f2)
              else raise DecodeError(f1,ERR_HIGH_SURROGATE c2)
           end
      else if isLowSurrogate c1
      then raise DecodeError(f1,ERR_LOW_SURROGATE c1)
      else (c1,f1)
   end
```

Note that handling of surrogates is not necessary for, e.g., the ASCII encoding because it can not represent the high- and low-surrogates.

2.3.6 Representing Encoded Files

In order to associate a file with a character encoding, we define an enumeration type for encoding names. It has two constructors for UTF-16: This encoding exists in big-endian and little-endian byte order; for UCS-4 it is similar.

```
datatype Encoding =
     NOENC | ASCII | LATIN1 | EBCDIC | ...
   | UTF8 | UTF16B | UTF16L | UCS4B | UCS4L
```

The value NOENC is used for unknown or unsupported encodings; it usually indicates an error or a file whose end is reached. Based on this type, we can define a type for encoded files along with a function for decoding a single character:

```
type DecFile = Encoding ∗ File

fun decGetChar (enc,f) =
   let val (c,f1) = case enc
                      of NOENC  ⇒ raise EndOfFile f
                       | ASCII  ⇒ getCharAscii f
                       | LATIN1 ⇒ getCharLatin1 f
                        ...
                       | UTF8   ⇒ getCharUtf8 f
   in (c,(enc,f1))
   end
handle EndOfFile f ⇒ raise DecEndOfFile(NOENC,f)
     | DecodeError(f,err) ⇒ raise DecError((enc,f),err)
```

The function handles EndOfFile and DecodeError exceptions raised by the decoding functions. It then raises a corresponding exception which carries the file and the character encoding. This is necessary for making the DecodeFile.File type and its exceptions invisible outside of the decoder.

But this function does not seem to be very efficient: For each character, the encoding-specific decoding function must be determined by a **case** distinction. An apparently evident optimization incorporates the decoding function into the DecFile type.

```
type GetChar = File → Char ∗ File
type DecFile = Encoding ∗ GetChar ∗ File

fun decGetChar (enc,get,f) =
   let val (c,f1) = get f
   in (c,(enc,get,f1))
   end
handle EndOfFile f ⇒ raise DecEndOfFile(NOENC,getCharEof,f)
     | DecodeError(f,err) ⇒ raise DecError((enc,get,f),err)
```

This variant has a slightly object-oriented flavor: Values of type DecFile carry the function for decoding a character, similar to the methods of an object. This approach, however, turned out to be absolutely impractical: At least in SML/NJ, this modification enormously increases the execution time of the parser. We did not investigate the reasons for this behavior, but it is certainly astonishing: Though functional programming languages encourage the use of data structures with functional components, current implementations apparently do not efficiently support that.

We abandoned this approach and pursued another one: We implemented a function that, instead of decoding a single character, tries to fill an entire buffer with UNICODE characters. The case expression for determining the decoding function is then required only once for the whole buffer. The buffer is filled with characters incrementally and must therefore be implemented as an array.

```
fun decGetArray (enc,f) arr =
  let fun loadArray getChar =
    let val len = Array.length arr
      exception Error of int * exn
      fun load (idx,f) = if idx=len then (len,(enc,f),NONE)
                         else let val (c,f1) = getChar f
                                     handle exn ⇒ raise Error (idx,exn)
                                   val _ = Array.update(arr,idx,c)
                               in load (idx+1,f1)
                               end
      in load (0,f) handle Error(idx,exn)
                  ⇒ case exn
                       of EndOfFile f ⇒ (idx,(NOENC,f),NONE)
                        | DecodeError (f,err) ⇒ (idx,(enc,f),SOME err)
                        | _ ⇒ raise exn
      end
    in case enc
         of NOENC ⇒ (0,(NOENC,f),NONE)
          | ASCII ⇒ loadArray getCharAscii
          | LATIN1 ⇒ loadArray getCharLatin1
            . . .
          | UTF8 ⇒ loadArray getCharUtf8
    end
```

Function loadArray decodes characters and writes them into the buffer until the array is full or an exception raised by the decoding function indicates the end of the file or a error!decode. It returns the number of characters actually decoded, the new encoded file, and an optional decoding error. Note that exceptions raised by the decoding function are handled and packed into the auxiliary exception Error. This exception additionally carries the current position in the array, required for reporting the number of decoded characters.

Using an array as a buffer is indeed imperative programming style, but the accomplished increase in execution speed justifies that (cf. Section 3.1.2). Moreover, the array is never updated outside of this function: All other functions only read from the buffer.

2.3.7 Auto-Detection of Character Encodings

XML entities must either be encoded in UTF-8 or have an encoding declaration at the beginning. This declaration, however, is in the same character encoding as the rest of the entity: The parser must therefore detect the encoding in order to parse the encoding declaration. The XML recommendation provides a heuristic for auto-detecting the encoding from the first four bytes of a file. E.g., the two bytes FE FF at the start of a file are interpreted as a *byte-order mark* and indicate big-endian UTF-16 encoding. On the other hand, 3C 3F 78 6D is the ASCII code sequence of the string "`<?xm`", initiating an XML or text declaration. Therefore this sequence suggests an encoding that is at least compatible with UTF-8 in the range of ASCII characters; this range is sufficient for parsing the encoding declaration. If the first four bytes are not recognized as part of the string "`<?xm`" in one of the supported encodings, the parser supposes that there is no encoding declaration and thus assumes UTF-8 encoding.

The encoding-detection heuristic is implemented by function decOpenXml.

It opens a file, auto-detects its encoding and returns the encoded file.

```
fun decOpenXml uri =
   let fun get4Bytes (n,f) = if n=4 then (nil,f)
                             else let val (b,f1) = getByte f
                                      val (bs,f2) = get4Bytes(n+1,f1)
                                  in (b::bs,f2)
                                  end
                                 handle EndOfFile f ⇒ (nil,f)
      fun detect bs = case bs
                        of 0wxFF::0wxFE::rest ⇒ (UTF16L,rest)
                         | 0wxFE::0wxFF::rest ⇒ (UTF16B,rest)
                           ...
                         | [0wx3C,0wx3F,0wx78,0wx6D] ⇒ (UTF8,bs)
                         | [0wx4C,0wx6F,0wxA7,0wx94] ⇒ (EBCDIC,bs)
                         | _ ⇒ (UTF8,bs)
      val f = openFile uri
      val (bs,f1) = get4Bytes(0,f)
      val (enc,unget) = detect bs
   in (enc,ungetBytes(f1,unget))
   end
```

The bytes read in order to detect the encoding are part of the file's content which must be completely available to the parser. Therefore we must unget these bytes from the file before returning it. The only exception is UTF-16: A byte-order mark (FE FF or FF FE) indicating this encoding is not treated as part of the file's data. The auxiliary function detect therefore returns the detected encoding together with the list of bytes to unget.

After opening a file with decOpenXml, the parser must try to read the encoding declaration and switch to the character encoding actually declared; if there is no encoding declaration, it commits to the auto-detected encoding. This is the purpose of the two functions decSwitch and decCommit. They raise an error if the declared encoding is incompatible with the auto-detected one, or if the auto-detected encoding requires an encoding declaration which was not present.

This concludes the description of the UNICODE frontend.

2.4 Errors and Options

The easiest way of reporting an error is to print a message to the standard error device. An application, however, might want to avoid error messages and instead let the user browse through the errors in a dedicated window on his screen. It might also be interested only in a certain class of errors and ignore all others. Therefore, an error is not represented by the message it generates but by a value of a dedicated data type Error, defined by structure Errors. An error is reported to the application as a value of that type together with its position in the document.

```
type Position = string * int * int
```

```
datatype Error =
    ERR_EMPTY of string
  | ERR_ENDED_BY_EE of string
  | ERR_EXPECTED of string * Data
    ...
  | ERR_ILLEGAL_CHAR of Char
  | ERR_DECODE_ERROR of DecodeError
```

A Position describes the physical location of an error by means of a URI, line and column number. Type Error enumerates the kinds of possible errors. E.g., the error ERR_ENDED_BY_EE str is reported if a component of the document is ended by an entity end; str denominates that component. Another common syntax error is that the parser finds one or more characters it doesn't expect. In this case the error ERR_EXPECTED(str,cs) is reported; str describes what the parser expected whereas cs are the characters found instead. A data type Warning similar to Error is defined for warnings.

Structure Errors also provides useful functions for generating error messages and for classifying errors:

```
val Position2String   : ErrorData.Position → string

val errorMessage      : ErrorData.Error → string
val warningMessage    : ErrorData.Warning → string

val isFatalError      : ErrorData.Error → bool
val isDecodeError     : ErrorData.Error → bool
val isSyntaxError     : ErrorData.Error → bool
val isValidityError   : ErrorData.Error → bool
val isWellFormedError : ErrorData.Error → bool
```

The application can then use these functions for handling errors and warnings. Section 2.8 will explain in detail how errors are reported to the application. For simplicity we will until then assume that this happens by a call to function reportError:

```
val reportError : Position * Error → unit
```

Note that the function result type is unit, and its only arguments are the position and the error. This means that the error must be reported by a side-effect. We will eliminate this non-functional behavior in Section 2.8.1.

Many of *fxp*'s features can be controlled by *options*: E.g., option O_VALIDATE indicates whether the parser should run in validating or non-validating mode. Other options control whether compatibility and interoperability with SGML should be checked (cf. Section 1.1.6), or they enable or disable certain kinds of warnings.

Options are normally of type bool. They are provided by a structure ParserOptions which holds all options as references. References are SML's variant of pointers. They are mutable and thus non-functional types. Their advantage is that they can easily be changed according to, e.g., command-line options. Since the parser itself only reads but never sets options, we need not have a bad conscience for using references here. Section 2.8.2 will show how to have each instance of a reentrant parser supplied with its own set of options.

```
datatype Special = DOC_ENTITY | EXT_SUBSET
datatype EntId = GENERAL of int | PARAMETER of int
datatype ExtType = SPECIAL of Special | NORMAL of EntId * State

and State = ENDED of EntId * State
          | CLOSED of DecFile * int * int * ExtType
          | INT of Vector * int * int * (EntId * State)
          | EXT1 of DecFile * int * int * bool * ExtType
          | EXT2 of Char array * int * int * int * int * bool
                 * (DecFile * DecodeError option * ExtType)
          | LOOKED of Data * State
```

Figure 2.10: The data type for the entity stack.

2.5 The Entity Manager

Through an entity reference, an XML document can include the replacement text of an entity into the document (cf. 1.1.2). Since inclusion of entities can be nested, the set of open entities must be maintained on a stack. A data type for this stack is provided by the *entity manager*, together with functions for manipulating it. Among others, these include functions for obtaining information about the entities on the stack and opening new entities. For each entity on the stack, the entity manager counts the input position in that entity. This position can be retrieved by functions from other modules, e.g., for error reporting.

Furthermore, the entity manager is the parser's interface to the UNICODE frontend. It provides a function getChar for reading a character from the entity on top of the stack, filtering out UNICODE characters forbidden in XML and normalizing line breaks into a platform-independent format.

Finally, the entity manager stores for each entity on the stack its *entity identifier*, which uniquely identifies the entity. This is due to a well-formedness constraint in XML requiring that the first and last character of each element must be in the same entity replacement text. E.g., if entity `start` has the replacement text `"<a>"` and entity `end` has replacement text `"</a>"`, then `&start;foo&end;` yields `<a>foo</a>`. This element is ill-formed because its last character is in a different entity replacement text than its first character. The entity manager therefore defines an equality type EntId for entity identifiers, which consists of the entity's index in the DTD tables (cf. 2.6.1 on page 45), together with the information whether it is a general or parameter entity. The parser can then obtain the EntId of the current entity, both when entering and leaving an element, and compare them in order to detect a violation of this well-formedness constraint.

The data type implementing the entity stack is given in Figure 2.10. In order to explain this type, we will consecutively show for each type of entities how it is represented and how the function getChar retrieves a character through it. Finally we will give the full signature of the Entities structure.

2.5.1 Internal Entities

Internal entities exist completely in memory: Their replacement text is a vector of UNICODE characters. An internal entity is represented by

```
INT(vec,s,i,(id,other))
```

where vec is the vector holding the entity's replacement text, and s and i are its size and the position of the next character to read. Component id is the entity identifier and other contains the underlying rest of the entity stack. Note that these two components are grouped into a pair because they change less frequently than the other components. Moreover, they are not inspected at all by the most important function of the entity manager, getChar.

Reading a character from an internal entity is simple: It need not be decoded, no lines must be counted and line breaks need not be normalized. Function getChar only looks up the next character and increments the position, if the end of the entity is not reached yet:

```
fun getChar (INT(vec,s,i,io)) =
  if i ⩾ s then (0wx0,ENDED io)
  else (Vector.sub(vec,i), INT(vec,s,i+1,io))
```

If all characters of an internal entity have been consumed, getChar returns a zero character indicating the entity end. The entity itself is not discarded but temporarily remains on top of the entity stack. The reason is that this entity end might cause an error. The error message should then refer to this entity as the source of the error.

In order to avoid reporting the entity end repeatedly, the entity is marked as *closed*, discarding the vector, its size and position, and using constructor ENDED instead of INT. When getChar encounters a closed entity, it proceeds to the underlying entity stack:

```
  | getChar (ENDED(_,other)) = getChar other
```

2.5.2 External Entities

External entities are more complicated to handle than internal ones: The replacement text of an external entity is given by the URI of the file that holds it. In order to read a character from an external entity we must decode the contents of this file through the UNICODE frontend. Furthermore, an external entity might contain characters that are illegal in XML; the entity manager must report an error for such a character.

Aside from that, the line breaks in external entities are not normalized. In XML, a line break is represented by a single line feed character (0xA). Different operating systems, however, have other representations: MS/DOS uses a carriage return (0xD) followed by a line feed; other systems use a single carriage return. XML requires that parsers normalize line breaks prior to processing the document, i.e., any of the three variants must be replaced by a single line feed character. This is achieved by replacing each carriage return with a line feed, and then skipping a possibly following line feed.

For reasons of encoding detection, we distinguish between two kinds of external entities, depending on whether their encoding is already known (cf. 2.3.7). The difference is that until the encoding is known, the entity must be decoded character by character because the parser might still switch to another encoding. As soon as the encoding is reliable, we can use a buffer for decoding a whole sequence of UNICODE characters at once. Let us start with the first case: An external entity with uncertain encoding is represented by

```
EXT1(dec,l,col,br,typ)
```

where dec is the file descriptor of type DecFile providing the entity's text through the Decode structure; l and col are the current line and column number in that file. br is a boolean indicating whether the last character was a carriage return and replaced with a line feed. In this case a succeeding line feed must be skipped.

Component typ is of type ExtType and describes the type of the external entity: SPECIAL indicates that it is either the document entity or the external subset. These entities are never removed from the entity stack because there is no enclosing entity. On the other hand, NORMAL(id,other) indicates that the entity was opened by a reference to an entity declared in the DTD: id is the entity identifier of that entity and other is the underlying stack.

Reading a character from an external entity is rather complicated. First a character must be decoded with decGetChar. Then we distinguish several cases:

- if this character is a tab character (0wx9) or an XML character other than carriage return or line feed (0wx20–0wxD7DFF, 0wxE000-0wxFFFD, 0wx10000-0wx10FFFF), it is returned together with the entity stack, the column incremented by one and the br flag set to false;
- if the character is a line feed, then there are two cases: If br is true then the last character was a carriage return and this line feed must be skipped; the next character is obtained by a recursive call to getChar. Otherwise the line feed is returned with the line number incremented by one and the column reset to zero;
- in case of a carriage return, we return a line feed instead, increment the line number and reset the column. We also set the br flag in the new entity stack, indicating that a succeeding line feed must be skipped;
- all characters not captured by these cases are illegal in XML documents. For such a character an error is reported and the next character is read by a recursive call to getChar.

The definition of getChar for this kind of external entities is as follows:

```
| getChar (EXT1(dec,l,col,br,typ)) =
  let val (c,dec1) = decGetChar dec
  in if c>=0wx0020 andalso c<=0wxD7FF orelse c=0wx9 orelse
        c>=0wxE000 andalso c<=0wxFFFD orelse c>=0wx10000
     then (c,EXT1(dec1,l,col+1,false,typ))
     else if c=0wxA (* line feed *)
          then if br then getChar (EXT1(dec1,l,col,false,typ))
               else (c,EXT1(dec1,l+1,0,false,typ))
     else if c=0wxD (* carriage return *)
          then (0wxA,EXT1(dec1,l+1,0,true,typ))
          else let val _ = reportError(getPos q,ERR_ILLEGAL_CHAR c)
               in getChar(EXT1(dec1,l,col+1,false,typ))
               end
  end
```

The decGetChar call, however, might raise an exception that must be caught by an exception handler. In case of a decoding error (DecError), an error is issued and the next character is read by a recursive call. The other possibility for an exception (DecEndOfFile) is that the end of the file is reached. Then a zero character indicating an entity end is returned. Similarly to internal entities, the entity is replaced by a closed external entity, discarding all information that is not needed any more:

```
      handle DecError(dec,err) ⇒
                 let val _ = reportError(getPos q,ERR_DECODE_ERROR err)
                 in getChar(EXT1(dec,col,l,br,typ))
                 end
           | DecEndOfFile dec ⇒ (0wx0,CLOSED(dec,l,col,typ))
```

When getChar encounters a closed external entity, it proceeds to the underlying stack. In case of a special external entity, however, this is an internal error because there is no underlying entity stack.

```
  | getChar (CLOSED (_,_,_,typ)) =
      case typ
        of SPECIAL _ ⇒ raise InternalError
         | NORMAL(_,other) ⇒ getChar other
```

For external entities whose character encoding has already been determined, we apply the optimization from 2.3.6: Instead of decoding single characters, we use an array buffer for decoding a whole sequence of UNICODE characters at once. The entry of such an external entity on the stack must additionally hold this buffer, its size s and its current input position i:

```
EXT2(buf,s,i,l,col,br,(dec,err,typ))
```

Furthermore, it has an additional component err of type DecError option. If this has the value SOME err, then err is a decoding error that occurred when trying to decode another character (cf. 2.3.6). This error must be reported before the buffer is reloaded.

2.5.3 Implementing Look-Ahead

Sometimes the parser must do a small look-ahead in the input. Actually this happens only at the very beginning of an external entity, when the parser tries to find out whether the entity starts with an XML or text declaration. For this purpose, instead of implementing a look-ahead, the entity manager provides a function ungetChars for placing a list of UNICODE characters back into the input. On the entity stack, this is represented by LOOKED(cs,other).

```
fun getChar ...
  | getChar (LOOKED(nil,q)) = getChar q
  | getChar (LOOKED(c::cs,q)) = (c,LOOKED(cs,q))

fun ungetChars (q,cs) = LOOKED(cs,q)
```

Note the difference between ungetChars and the function ungetBytes from 2.3.4: The latter operates on bytes, whereas ungetChars operates on already decoded UNICODE characters.

```
signature Entities =
   sig
      type State
      eqtype EntId
      datatype Special =DOC_ENTITY | EXT_SUBSET

      val getChar      : State → UniChar.Char ∗ State
      val ungetChars   : State ∗ UniChar.Data → State

      val pushIntern   : State ∗ int ∗ bool ∗ UniChar.Vector → State
      val pushExtern   : State ∗ int ∗ bool ∗ Uri.Uri → State ∗ Decode.Encoding
      val pushSpecial  : Special ∗ Uri.Uri option → State ∗ Decode.Encoding
      val closeAll     : State → unit

      val commitAuto   : State → State
      val changeAuto   : State ∗ string → State ∗ Decode.Encoding

      val getEntId     : State → EntId
      val getPos       : State → Errors.Position
      val getUri       : State → Uri.Uri

      val isOpen       : State ∗ int ∗ bool → bool
      val isSpecial    : State → bool
      val inDocEntity  : State → bool
   end
```

Figure 2.11: The Entities signature.

2.5.4 Other Important Entity Manager Functions

The signature of the Entities structure is given in Figure 2.11. Aside from those explained so far, it provides functions for opening and closing entities, and for extracting useful information other than input characters from the entity stack.

The entity manager provides different functions for opening internal and external entities. Both these functions compute the entity identifier from the entity's index in the DTD and a flag indicating whether it is a parameter entity:

```
fun makeEntId(idx,isParam) =
   if isParam then PARAMETER idx else GENERAL idx

fun pushIntern(q,id,isParam,vec) =
   INT(vec,Vector.length vec,0,(makeEntId(id,isParam),q))

fun pushExtern(q,id,isParam,uri) =
   let val dec = decOpenXml (SOME uri)
      val auto = decEncoding dec
      val q1 = EXT1(dec,1,0,false,NORMAL(makeEntId(id,isParam),q))
   in (q1,auto)
   end
```

pushIntern can directly use the character vector given as argument for constructing the new entity stack; pushExtern must open the entity through the UNICODE frontend. It then determines the auto-detected encoding of the entity and returns it together with the new entity stack.

For special entities, i.e., the document entity or the external subset, function pushSpecial is implemented similar to pushExtern, with the difference that it does not expect the current entity stack as argument. The aptly named function closeAll is intended for closing all files associated with external entities on the stack. This might involve deletion of temporary files for entities retrieved over the net; therefore the entity stack may not simply be discarded at a fatal error.

The auto-detected encoding of an external entity is not a definite decision. Instead, it is possible to change to a compatible encoding after parsing the entity's encoding declaration, or to commit to the auto-detected choice if there is no encoding declaration. The latter is the purpose of function commitAuto:

```
fun commitAuto q =
   case q
     of EXT1(dec,l,col,brk,typ) ⇒
         let val _ = decCommit dec handle DecError(_,err)
                         ⇒ reportError(getPos q,ERR_DECODE_ERROR err)
            val (arr,n,dec1,err) = initArray dec
         in EXT2(arr,n,0,l,col,brk,(dec1,err,typ))
         end
      | LOOKED(cs,q1) ⇒ LOOKED(cs,commitAuto q1))
      | _ ⇒ q
```

The main task of this function is to switch from the EXT1 representation to the more efficient EXT2. In addition it checks whether the auto-detected encoding requires an encoding declaration. In that case it reports an error. changeAuto is similar, only that it switches to a new encoding with decSwitch if that is compatible with the auto-detected one.

Other functions are getEntId for obtaining the entity identifier of the top-most entity on the stack and getPos which returns the position in the top-most external entity by means of its URI and the current line and column number, whereas getUri returns its URI only. isOpen tells whether an entity is already on the stack; this is for avoiding infinite recursion in entity references. isSpecial determines whether the top-most entity is a special external entity, and inDocEntity tells whether the top-most external entity is the document entity.

2.6 The DTD Manager

The DTD manager maintains the declarations of the DTD in a number of tables. In this section we first explain the data structure used for implementing these tables, namely dictionaries. Then we define the types for the information stored in the tables.

This information, however, has to be checked for well-formedness and validity before entering it into the tables. E.g., a valid document may not contain multiple declarations for the same element type; and each element type may have at most one attribute of type `ID`. After being stored in the tables, the DTD's declarations are used for processing other components of the document, e.g., normalization of attribute values depending on their declared types.

In contrast to the implementation of dictionaries, these operations can produce errors. They are packed into two separate structures DtdDeclare and DtdAttributes, which are explained at the end of this section.

```
signature Dict =
  sig
    type Key
    type 'a Dict

    exception NoSuchIndex

    val makeDict   : int ∗ 'a → 'a Dict
    val clearDict  : 'a Dict ∗ int option → unit

    val hasIndex   : 'a Dict ∗ Key → int option
    val getIndex   : 'a Dict ∗ Key → int
    val getKey     : 'a Dict ∗ int → Key

    val getByIndex : 'a Dict ∗ int → 'a
    val getByKey   : 'a Dict ∗ Key → 'a

    val setByIndex : 'a Dict ∗ int ∗ 'a → unit
    val setByKey   : 'a Dict ∗ Key ∗ 'a → unit
  end
```

Figure 2.12: The Dict signature.

2.6.1 Dynamic Dictionaries

Declarations in the DTD associate information with *keys* which are *names*, i.e., sequences of UNICODE characters. Comparison and other operations on names are expensive; it is therefore desired to represent each name by a unique index. This is achieved by a *symbol table*, commonly implemented with the help of a hash table.

Additionally, a name is associated with some information, e.g., an element type is associated with the content model and the attributes declared for it. We therefore use *dictionaries* instead of symbol tables for implementing the XML DTD: Aside from its index, a dictionary can also store a value for each name in its map.

Dictionaries must have unlimited capacity, because it is not known in advance how many declarations the DTD contains. Therefore, we use *dynamic* dictionaries: A dynamic dictionary automatically grows to the double of its size if it has no more space for further entries. Growing a dictionary happens rather rarely: Since its size it doubled each time, it increases exponentially by the number of times it is grown.

A dictionary is implemented by a structure according to signature Dict, which is given in Figure 2.12. For generality, it is not committed to using names as keys but provides a Key type itself, together with a polymorphic dictionary type 'a Dict for storing values of type 'a. A dictionary is created by makeDict(n,a): The result is a dictionary of size 2^n holding the default value a for all – yet unassigned – indices. It can be reset to its initial state with clearDict which takes an optional argument indicating a new size.

The mapping from keys to indices is by function getIndex. If the dictionary already holds an index for the given key, this index is returned; otherwise the key is associated with the next unused index in the dictionary. Function hasIndex restricts this functionality: It returns an already existing index but does not make a new entry otherwise. getKey realizes the reverse mapping: It returns, for an existing index, the key associated with it; otherwise it raises exception NoSuchIndex.

The value a dictionary holds for each key can be obtained with getByKey and getByIndex. Initially, all values are equal to the default value provided to makeDict when the dictionary was created. Functions setByKey and setByIndex change a particular value.

Note that the Dict signature imposes an imperative implementation. Operations affecting the state of a dictionary do not return the modified dictionary; instead the modifications must happen as non-pure side-effects. This is natural: A hash table as used for dictionaries is most intuitively implemented by an array, and that has destructive update.

The implementation of the Dict signature is by a functor Dict. It is parametrized with a structure Key providing the Key type together with essential functions for implementing a dictionary: a compare function and a hash function:

```
signature Key =
   sig
      type Key

      val null    : Key
      val hash    : Key → word
      val compare : Key ∗ Key → order
   end

functor Dict (structure Key : Key) : Dict =
   struct
      type Key = Key.Key
      ...
   end
```

Then it is easy to obtain a dictionary for XML names. We only need to apply the Dict functor to an appropriate Key structure:

```
structure KeyData : Key =
   struct
      type Key = UniChar.Data

      val null = UniChar.nullData
      val hash = UniChar.hashData
      val compare = UniChar.compareData
   end

structure DataDict = Dict (structure Key = KeyData)
```

The implementation of dictionaries is rather technical; we therefore omit the details here and refer the interested reader to the source code of *fxp* [Fxp99].

2.6.2 Data Types for Declarations

The values stored for declarations in the DTD tables are complex data structures: For an element type, e.g., the tables must hold its content model and all attributes that were declared for it. Additionally the type and default value must be stored for each of these attributes. Let us have a closer look at the types defined for this. To start with we need a type for content models:

```
datatype ContentModel =
    CM_ELEM of int
  | CM_OPT of ContentModel
  | CM_REP of ContentModel
  | CM_PLUS of ContentModel
  | CM_ALT of ContentModel list
  | CM_SEQ of ContentModel list
```

The int argument to constructor CM_ELEM is the index of an element in the DTD tables. Note that the ContentModel data-type does not allow character data. The reason is that XML distinguishes two forms of content: Content models specify *element content* which may not contain character data, whereas character data may occur in *mixed content*. However, only a restricted form of content models is allowed for element-types with mixed content: It may not specify the order in which elements appear. Such a content model is always of a form similar to `(#PCDATA|elem|...)*`. Having in mind that there are two other special cases, namely `ANY` for arbitrary content and `EMPTY` for empty content, we can define a type for content specifications:

```
datatype ContSpec =
    CT_ANY
  | CT_EMPTY
  | CT_MIXED of int list
  | CT_ELEMENT of ContentModel * Dfa
```

The CT_ELEMENT constructor has an additional argument of type Dfa: Content models are implemented by deterministic finite automata (DFA). Whenever a content model is declared, a DFA is constructed (see 2.7.6) and stored with it.

Aside from the content specification, we have to store for each element type the attributes declared for it. An attribute has a type and a default value; let us first have a look at attribute types. All XML attribute values are sequences of characters. Some of them, however, are additionally interpreted by the parser: The attribute type `ENTITIES`, e.g., requires the attribute value to be a space-separated list of names, all of which are declared as unparsed general entities. These names are mapped to indices by the DTD; a more succinct and informative representation for an attribute value of this type is therefore a list of integers. Table 2.1 lists the allowed types for XML attributes together with the SML types that implement them. Note that though values of the enumeration type $(\mathtt{n}_1|\mathtt{n}_2|\ldots)$ are uninterpreted name tokens, we map them to indices in order to compare them efficiently.

We can now define data types for attribute types and values:

```
datatype AttType =              datatype AttValue =
    AT_CDATA                        AV_CDATA of UniChar.Vector
  | AT_NMTOKEN                    | AV_NMTOKEN of UniChar.Data
  | AT_NMTOKENS                   | AV_NMTOKENS of UniChar.Data list
  | AT_ID                         | AV_ID of int
  | AT_IDREF                      | AV_IDREF of int
  | AT_IDREFS                     | AV_IDREFS of int list
  | AT_ENTITY                     | AV_ENTITY of int
  | AT_ENTITIES                   | AV_ENTITIES of int list
  | AT_GROUP of int list          | AV_GROUP of int list * int
  | AT_NOTATION of int list       | AV_NOTATION of int list * int
```

Attribute type	Description	SML type
`CDATA`	character data	Vector
`NMTOKEN`	a name token, consisting of alphanumeric characters only	Data
`NMTOKENS`	a list of name tokens	Data list
`ID`	a name, i.e., a name token starting with a letter. It may not occur as the value of another `ID` attribute	int
`IDREF`	a name that occurs as an `ID` attribute of some element	int
`IDREFS`	a list of `IDREF` names	int list
`ENTITY`	the name of an unparsed general entity	int
`ENTITIES`	a list of `ENTITY` names	int list
`(`n_1`|`n_2`|...)`	one of the specified name tokens	int
`NOTATION(`n_1`|...)`	one of the specified names; these must be declared as notations	int

Table 2.1: XML attribute types

Note that attribute values carry all information about their type. In non-validating mode, however, attribute values are not interpreted but only normalized. In this case the attribute value is represented by a Vector. Since we want to use the same type in both validating and non-validating mode, an attribute value is always of type Vector ∗ AttValue option, for instance in the following two types: AttDefault describes an attribute's default value: Either it is required, implied, fixed or equipped with a default. Similarly, we describe the attribute values in an element's start-tag with the type AttPresent[4]: It may have been specified in the start-tag, but it may also be omitted. In that case it either has a default value, it is implied or it is missing though required:

```
datatype AttDefault =
    AD_IMPLIED
  | AD_REQUIRED
  | AD_FIXED of UniChar.Vector * AttValue option
  | AD_DEFAULT of UniChar.Vector * AttValue option

datatype AttPresent =
    AP_IMPLIED
  | AP_MISSING
  | AP_DEFAULT of UniChar.Vector * AttValue option
  | AP_PRESENT of UniChar.Vector * AttValue option
```

An attribute definition consists of the attribute's index, an attribute type and a default value. An attribute specification[4] takes the attribute's index and the present value:

```
type AttDef = int * AttType * AttDefault
type AttSpec = int * AttPresent
```

[4]Though values of type AttPresent or AttSpec are not stored in the DTD tables, we declare these two types here because they are closely related to the other types.

Having defined all necessary types, we can now define the type for the information that is stored in the DTD for each element type:

```
type ElemInfo = {decl : ContSpec option,
                 atts : AttDef list }
```

It consists of an optional content specification and a list of attribute definitions. Note that the type really used in the implementation holds additional information, e.g., a boolean field indicating whether the element declaration was in an external parameter entity. This information is needed for determining the *standalone status* of the document, i.e., whether a non-validating parser can ignore external entities without misbehaving. This status can be declared as true in the XML declaration. A validating parser must then report an error if, e.g., an externally declared element with element content contains white-space characters (for details, see Section 2.9 of [W3C98b]). For brevity, we omit such details here and simplify the types.

Analogously, we define types representing external identifiers, general and parameter entities, notations and ID attribute values.

```
type ExtId = string option
           * (Uri * Uri) option

datatype GenEntity =
      GE_NULL
    | GE_INTERN of Vector
    | GE_EXTERN of ExtId
    | GE_UNPARSED of ExtId * int

datatype ParEntity =
      PE_NULL
    | PE_INTERN of Vector
    | PE_EXTERN of ExtId

type NotationInfo = ExtId option
type IdInfo = bool * Position list
```

An external identifier consists of an optional public identifier, represented as a string, and an optional system identifier, given by a URI and the base URI of the entity in which it is declared.

A parameter entity is either undeclared, internal or external; for general entities there is the additional possibility of an unparsed entity associated with the index of a notation. A notation itself is mapped to an external identifier if declared.

The IdInfo type reports for a name whether it occurred as an ID attribute and records the positions of all IDREF attributes referring to it.

2.6.3 The DTD structure

Having defined all types necessary for describing the information stored in the DTD, we can now give the signature of the Dtd structure in figure 2.13. It provides five name spaces for the DTD: for elements, for attributes and notations, for ID names, for general entities and for parameter entities. The attribute name space is used for attribute names and the name tokens occurring in enumerated attribute types. Because these name tokens include notation names, attributes and notations share a name space.

For each of the name spaces, the Dtd structure defines functions for mapping a name to its index and vice versa, and for obtaining and setting the information associated with it. Note that updates on the DTD are performed as side-effects: The modified DTD is not part of the functions' return values. Function initializeDtd is provided for initializing the tables.

```
signature Dtd =
  sig
    type Dtd

    val initializeDtd   : unit → Dtd

    val Element2Index : Dtd → UniChar.Data → int
    val AttNot2Index  : Dtd → UniChar.Data → int
    val Id2Index      : Dtd → UniChar.Data → int
    val GenEnt2Index  : Dtd → UniChar.Data → int
    val ParEnt2Index  : Dtd → UniChar.Data → int

    val Index2Element : Dtd → int → UniChar.Data
    val Index2AttNot  : Dtd → int → UniChar.Data
    val Index2Id      : Dtd → int → UniChar.Data
    val Index2GenEnt  : Dtd → int → UniChar.Data
    val Index2ParEnt  : Dtd → int → UniChar.Data

    val getElement    : Dtd → int → Base.ElemInfo
    val getNotation   : Dtd → int → Base.NotationInfo
    val getId         : Dtd → int → Base.IdInfo
    val getGenEnt     : Dtd → int → Base.GenEntity
    val getParEnt     : Dtd → int → Base.ParEntity

    val setElement    : Dtd → int ∗ Base.ElemInfo → unit
    val setNotation   : Dtd → int ∗ Base.ExtId → unit
    val setId         : Dtd → int ∗ Base.IdInfo → unit
    val setGenEnt     : Dtd → int ∗ Base.GenEntity → unit
    val setParEnt     : Dtd → int ∗ Base.ParEntity → unit
  end
```

Figure 2.13: The Dtd signature.

Type Dtd is implemented as a record of dictionaries, one for for each name space:

```
type Dtd = {elDict   : ElemInfo DataDict.Dict,
            idDict   : IdInfo DataDict.Dict,
            genDict  : GenEntity DataDict.Dict,
            parDict  : ParEntity DataDict.Dict,
            notDict  : NotationInfo DataDict.Dict }
```

Again, we disregard some technical details of the real implementation: There, the DTD has additional information about the document's standalone status and similar issues. For the purpose of this description, these details would only be confusing.

2.6.4 Adding Declarations

Whenever the parser adds a declaration to the DTD tables, it has to be checked for well-formedness and validity before actually entering it. This happens through the functions in structure DtdDeclare. Its signature is given in Figure 2.14. Because these functions can produce errors, they expect as an additional argument the state of the entity stack, in order to report the current position together with an error.

```
signature DtdDeclare =
   sig
      val addAttribute : Dtd.Dtd → Entities.State → int ∗ Base.AttDef → unit
      val addElement : Dtd.Dtd → Entities.State → int ∗ Base.ContSpec → unit
      val addGenEnt  : Dtd.Dtd → Entities.State → int ∗ Base.GenEntity → unit
      val addParEnt   : Dtd.Dtd → Entities.State → int ∗ Base.ParEntity → unit
      val addNotation : Dtd.Dtd → Entities.State → int ∗ Base.ExtId → unit

      val checkDefinedIds : Dtd.Dtd → Entities.State → unit
      val checkMultEnum  : Dtd.Dtd → Entities.State → unit
   end
```

Figure 2.14: The DtdDeclare signature.

The first five functions in the signature are for adding attributes, element types, entities or notations to the DTD tables. For example, let us have a look at function addElement:

```
fun addElement dtd q (idx,cont) =
   let val {decl,atts} = getElement dtd idx
   in case decl
        of NONE ⇒ setElement dtd (idx,{decl=SOME cont,atts=atts})
         | SOME _ ⇒ if not (!O_VALIDATE) then ()
                    else reportError(getPos q,ERR_REDEC_ELEM
                                              (Index2Element dtd idx))
   end
```

If the element type was declared previously, then the optional decl component of the ElemInfo has some value. In this case, the declaration is ignored and in validating mode (if option O_VALIDATE is true) an error is reported. Otherwise the element type's ElemInfo is updated with the declared content model.

Adding an entity or notation declaration is analogous; for attribute definitions, more has to be done: It must be checked whether the element already has an attribute with that name; if yes, the definition is ignored. Otherwise, if the attribute has type `ID`, it must be checked that an attribute of that type has not yet been declared for the element type. Only then is the attribute definition added to its ElemInfo.

The two other functions, checkDefinedIds and checkMultEnum, have a conceptually different task: Instead of adding pieces of information, they check properties of the whole DTD or document: checkMultEnum validates for each element type that no name token occurs more than once in its enumerated attribute types. Because there may be several attribute list declarations for a single element, this can not be checked before the whole DTD is parsed.

Similarly, checkDefinedIds verifies that all names occurring in attributes values of type `IDREF` or `IDREFS` do also occur as the value of an `ID` attribute value. Since an `IDREF` can point to an `ID` later in the document, this check can not be done when the `IDREF` attribute is encountered. It must be delayed until the end of the document instance.

```
signature DtdAttributes =
   sig
      exception AttValue

      val makeAttValue     : Dtd.Dtd → Entities.State
                              → int ∗ Base.AttType ∗ UniChar.Data
                              → UniChar.Vector ∗ Base.AttValue option
      val checkAttValue    : Dtd.Dtd → Entities.State
                              → Base.AttDef ∗ UniChar.Data → Base.AttPresent

      val genMissingAtts   : Dtd.Dtd → Entities.State
                              → Base.AttDef list → Base.AttSpec list
      val handleUndeclAtt : Dtd.Dtd → Entities.State → int ∗ int → unit
   end
```

Figure 2.15: The DtdAttributes signature.

```
fun checkDefinedIds dtd q =
   if not (!O_VALIDATE) then ()
   else let fun doit i =
                 let val (decl,refs) = getId dtd i
                     val _ = if decl orelse null refs then ()
                             else reportError(getPos q,ERR_UNDECL_ID
                                                       (Index2Id dtd i,refs))
                 in doit(i+1)
                 end
        in doit 0 handle NoSuchIndex ⇒ ()
        end
```

Function doit consecutively checks, starting from 0, for each index whether the ID name with that index is defined. Eventually, it reaches an index not associated with an ID name, making getId raise NoSuchIndex. This exception is handled in the body of the let and terminates the function.

2.6.5 Processing Attribute Values

Within the document instance, the most intensive use of the DTD tables is for attribute processing: In validating mode, the parser must check for each attribute in a start-tag whether it was declared for the element. In this case the attribute value is normalized depending on its declared type. It must also fulfill the constraints associated with that type. On the other hand, if one of the element's declared attributes is not specified in the start-tag, the parser must generate an attribute value from its default value. The functions for attending these tasks are defined in structure DtdAttributes whose interface is given in Figure 2.15.

Function makeAttValue processes the literal value of an attribute given as a list of characters, converting it into a character vector and – in validating mode – a value of type AttValue. For CDATA attributes, this requires only conversion of a list to a vector; for the other types, the following is performed:

- ✧ The attribute value is normalized by replacing consecutive space characters with a single space and removing leading and trailing spaces.

- ✧ In non-validating mode, the normalized attribute value is returned as a character vector, without the optional AttValue component.
- ✧ In validating mode, the attribute value is split into a list of tokens at space characters. The syntactical constraints imposed by the attribute type are verified. E.g., for type `ID` the list must consist of a single token which is a name. If these constraints are not met, an error is reported and exception AttValue is raised.
- ✧ The semantic constraints imposed by the attribute type are validated using and possibly updating the DTD tables. E.g., the single name constituting an `ID` attribute must not have occurred as an `ID` name before; this token is marked as being a used `ID` name in the DTD tables. Similarly, the position of an `IDREF` attribute is added to the name's IdInfo. Again, an error is reported and exception AttValue is raised if a constraint is not fulfilled.
- ✧ A value of type AttValue is constructed from the list of tokens. E.g., for an attribute of type `ENTITIES`, each token is interpreted as the name of a general entity and mapped to its index.
- ✧ The normalized attribute value is returned as a character vector together with the AttValue.

makeAttValue is a very complex function; documenting its source code is beyond the scope of this description. In addition to validating the specified attribute value against its type, however, the parser must also check whether it complies with its default value: If it has a fixed default value, then the normalized attribute value must be identical to that value. Function checkAttValue performs this additional check and constructs an attribute value of type AttPresent:

```
fun checkAttValue dtd q ((att,attType,defVal),cs) =
   let val (vec,av) = makeAttValue dtd q (att,attType,cs)
      val _ = case defVal
                of AD_FIXED(fix,_) ⇒
                   if not (!O_VALIDATE) orelse vec=fix then ()
                   else let val _ = reportError
                                  (getPos q, ERR_FIXED_VALUE(att,vec,fix))
                        in raise AttValue
                        end
                 | _ ⇒ ()
   in AP_PRESENT(vec,av)
   end
```

The two other functions in the DtdAttributes signature deal with attributes not declared or specified: handleUndeclAtt is called for an attribute specified but not declared for an element type. Basically, it reports an error if in validating mode. Function genMissingAtts takes as argument a list of attribute definitions whose attributes values have not been specified in a start-tag. It generates a list of attribute specifications from the default values declared for the attributes. The default values must in some cases be checked for validity: For example, the value for an attribute of type `ENTITY` must be the name of an unparsed general entity. That entity need not be declared when the default value is declared; only when the default is actually used, its declaredness must be verified. For those attributes that do not have a default value, the function generates AP_IMPLIED or – if the attribute is required – AP_MISSING after reporting an error.

2.7 The Parser Modules

This section describes the modules and functions that actually parse the XML document. They are written in a recursive-descent style: Roughly speaking, there is one parsing function for each syntactical component of an XML document. If one component is made up from other components, its parsing function makes use of the functions associated with these components. For an introduction into recursive-descent parsing, see, e.g., [WM95, ASU86].

Most parsing functions are rather technical and long-winded: They must check many syntactical and validity conditions. Therefore we illustrate the programming style of the parsing functions only with a few, simplified examples.

2.7.1 No Tokenization

In classical parser construction one usually divides the *lexical* analysis from the *syntactical* analysis. Lexical analysis is often referred to as *tokenization*: The input stream is converted into a stream of basic lexical components, the tokens, before it is fed to the parser. Examples for tokens are keywords, numbers or special symbols like operators or parentheses. The parser then need not bother with the syntactical representation of tokens; a token is the smallest unit from its point of view.

In *fxp* no tokenization is performed for several reasons:

- A common task of lexical analysis is filtering out the white space and skipping comments. In XML both white space and comments may be significant to the application and can therefore not be ignored.

- Tokenization of XML documents is context-dependent: According to the state of the parser, different tokens must be recognized. E.g., character "%" is recognized as the start of a parameter entity reference in the DTD, whereas it is treated as character data in content.

- Context-dependent tokenization can be realized by a set of different *recognition modes*. The parser must then switch between these modes when the context changes.

  Recognition modes were used for defining the syntax of XML's predecessor SGML [ISO86]. Following the SGML standard, parsing requires 10 different modes. When implementing an SGML parser, however, one recognizes that these are not sufficient: There are a bunch of exceptions when a special token may not be recognized in a particular mode. E.g., character "%" may not be recognized as the start of a parameter entity reference if it precedes the entity name in an entity declaration. Consequently, this situation requires its own recognition mode. The real number of recognition modes required for parsing SGML is therefore nearly 30.

SGML's syntax could not be given other than on a token basis, because SGML allows to change the concrete syntax, i.e., the lexical representation of tokens. In XML, the syntax change feature was dropped, and its syntax is given on a character basis.

◇ Using different recognition modes also complicates error recovery: After a syntax error, the context becomes unclear. The syntax error might have been caused by a missing token that would otherwise have initiated switching to another recognition mode. Trying to recover from the error might therefore fail, because the current recognition mode might ignore a token that could help the parser regain its orientation.

◇ Tokenizers are usually generated with the help of specialized tools such as ML-LEX [AMT94]. These tools, however, can not process UNICODE as input. In order to use a tokenizer for XML, all recognition modes have to be written by hand, or we have to develop a generator that can handle UNICODE input.

◇ Most XML tokens consist of only a single character; these can be recognized easily by the parser. For the few multi-character tokens in XML, the effort of implementing a tokenizer appears to be wasted.

The parser is thus written on character basis. This has, however, one major disadvantage: The code becomes difficult to read because a UNICODE character is represented by its hexadecimal value in the SML sources. E.g., the Char representation for character "%" is 0wx25. In order to make the code intelligible, careful use of comments is therefore indispensable.

2.7.2 Syntax Errors and Error Recovery

In recursive-descent parsing, each syntactical component is associated with a function that parses it. If a component is composed of other components, its parsing function calls the associated functions for parsing those parts. If a syntax error occurs, i.e., an input character is encountered that was not expected, the parsing function must either recover from that error or report it to the calling function. Reporting an error is done by raising an exception, either SyntaxError or NotFound. Both carry as arguments the erroneous character and the state of the entity stack:

```
exception NotFound of UniChar.Char * State
exception SyntaxError of UniChar.Char * State
```

NotFound is raised by a parsing function if the first character it reads does not start the component it expects to find. A NotFound exception can be handled easily in many cases. E.g., if syntactically required white space is not present, the parser reports an error and continues as if the white space had been there. A different use of this exception is for parsing of optional components. E.g., in a start-tag, the element type may be followed by a list of attribute specifications. After having read the element-type, function parseStartTag calls function parseName. If that returns a name, an attribute specification follows. Otherwise parseName raises NotFound indicating that the end of the start-tag should follow. parseStartTag handles this exception and continues by reading the ">" or "/>" terminating the start-tag.

As opposed to that, a SyntaxError is raised if a function can read an initial segment of a component but fails to finish parsing it. Handling a SyntaxError is more difficult: In most cases the parser runs into an undefined state. One way of regaining orientation is to switch into *panic mode*: In this mode the parser skips all characters until it finds one that would terminate the component in which the error occurred; we call such a character a *last-character*. For a start-tag, e.g., the last-character is ">". The parser then pretends that it has successfully parsed this component, and continues parsing with the succeeding character.

However, the behavior of panic mode is not sufficient for recovering from errors in many situations: It fails if, e.g., the error was caused by the absence of the last-character itself. Panic mode would then skip characters until the input is exhausted, or until it finds another occurrence of a last-character, which might belong to a completely different component though. For instance, if the closing bracket "]" of the DTD's internal subset is missing, panic mode would skip characters until the end of the document, or at least until the end of the first CDATA section.

Another deficiency of panic mode is that it blindly takes the first occurrence of a last-character as the end of the component. But this character might as well be part of a subcomponent and should therefore be skipped. A start-tag, e.g., can contain attribute values enclosed between quote characters. If character ">" occurs within an attribute value, it should not be interpreted as the end of the start-tag. A character that might start such a subcomponent is called a *start-character*.

In order to improve on the behavior of panic mode, [WM95] proposes in Section 8.3.6 to switch between two different modes: Error recovery starts in *error mode*. As soon as a start-character for a subcomponent is encountered, *parser mode* is entered: It attempts to entirely parse the subcomponent before it switches back into error mode. Moreover, error mode is more intelligent than panic mode: It also terminates if it encounters a last-character for an enclosing component, or if it finds a *follow-character*, i.e., a start-character for a component that might follow the current component.

For *fxp*, we implemented a slightly weaker variant of this approach: Error recovery starts in *skip mode* for the current component. This mode does not attempt to parse a subcomponent – it only skips all characters that might be part of it. Skip mode is aware of start-characters and last-characters; if it finds a start-character for a subcomponent, it recursively calls itself for that subcomponent. More precisely, skip mode ignores characters until one of the following situations comes up:

Local continuation on last-characters: A last-character of the current component is found. Skip mode finishes for the current component, and depending on the situation, either parsing is continued or skip mode is resumed for the enclosing component.

Non-local continuation on last-characters: A last-character for an enclosing component is found, which can not be part of the current component. Similarly to a local continuation, skip mode is terminated, with the difference that the character is not consumed. For instance, during skip mode for a markup-declaration, a "`]`" is a last-character for the enclosing declaration subset.

Continuation on follow-characters: A first-character of a component is encountered which might follow the current component. The behavior of the parser is the same as for non-local continuation on last-characters. An example for this situation is a "`&`" character during skip mode for a start-tag. This character can not occur in the start-tag; presumably it starts a reference following the tag.

Recursive call on start-characters: A start-character for a syntactical subcomponent is found. In this case skip mode calls itself recursively for that subcomponent. After termination of the recursive call, skip mode continues for the current component. An example for such a subcomponent is an attribute value in a start-tag. Note that since all characters are allowed in attribute values, the recursive skip mode for the attribute value will only terminate upon the closing quote character or an entity end.

Our error recovery strategy is intelligent enough to cope with single errors in most cases. This strategy, however, can still fail, especially if multiple errors coincide: Consider the erroneous start-tag `<a x=1 y="2>`. The first error is that the attribute value for `x` is not enclosed between quotes. The parser switches into skip mode and, when it encounters the quote sign, recursively calls skip mode for an attribute value literal. But now another error follows: The closing quote character is missing. Therefore the "`>`" character is not recognized as a last-character for the start-tag.

2.7.3 Parsing Simple Components

This and the following two subsections will try to give an impression of the programming style of the parsing functions. However, the more complex the syntactical components, the more validity and well-formedness constraints must be verified, and the more special cases must be distinguished by the parsing functions. We will see in 2.7.4 that the complexity of these functions grows beyond what is describable in this writing.

To start with let us consider a very simple parsing function: parseName reads a name and returns it as a list of characters, together with the immediately following character and the obtained state of the entity stack.

```
fun parseName' (c,q) =
   if isName c then let val (cs,cq1) = parseName'(getChar q)
                    in (c::cs,cq1)
                    end
   else (nil,(c,q))
```

```
fun parseName (c,q) =
    if isNms c then let val (cs,cq1) = parseName'(getChar q)
                    in (c::cs,cq1)
                    end
    else raise NotFound(c,q)
```

If the first character is not a name-start character, exception NotFound is raised. Otherwise all characters until the next non-name character are consumed by function parseName'. The character immediately following the name must be inspected in order to recognize the end of the name. That character is therefore part of the function's return value. This is typical for most of *fxp*'s parsing functions: Their return value includes the next character and the obtained state of the entity stack.

Function parseName' is not tail-recursive. Tail-recursion is the functional variant of iteration: If the recursive call of a function constitutes its return value, the function's stack-frame can be reused for the recursive call. A tail-recursive function therefore needs only a constant amount of space on the stack. In contrast to that, a non-tail-recursive function requires space proportional to the recursion depth.

In the case of function parseName the non-tail-recursive behavior is affordable because names are usually short; the recursion stack does not grow very deep. This is different when parsing a comment or processing instruction: These can become very long. Especially comments, which are frequently used for marking alternative version of large document parts, often have a size of thousands of characters. Let us have a look at function parseComment. It reads a comment starting after the initial "`<!--`" and returns its text as a vector together with the following character and the obtained entity stack:

```
fun parseComment q =
    let fun check_end yet (c,q) =
            if c=0wx2D (* "-" *)
            then let val (c1,q1) = getChar q
                 in if c1=0wx3E (* ">" *) then (yet,getChar q1)
                    else doit (0wx2D::0wx2D::yet) (c1,q1)
                 end
            else doit (0wx2D::yet) (c,q)

        and doit yet (c,q) =
            if c=0wx2D (* "-" *) then check_end yet (getChar q)
            else if c<>0wx0 then doit (c::yet) (getChar q)
            else let val _ = reportError(getPos q,
                                          ERR_ENDED_BY_EE "comment")
                 in (yet,(c,q))
                 end

        val (cs,cq1) = doit nil (getChar q)
        val vec = Data2Vector (rev cs)
    in (vec,cq1)
    end
```

The auxiliary function doit reads characters until it finds the end of the comment. The comment is ended by the string "`-->`", which is tested for by check_end, after a single "`-`" has been encountered[5]. An entity end (0wx0) occur-

[5]The real implementation of checkEnd is more extensive: If must report an error if a "`--`" is not

ring in the comment constitutes an error. The function recovers from the error: It ends the comment as if the "-->" had been present, concealing the error from the calling function.

The text of the comment is accumulated by doit in parameter yet as a list of characters. When the end of the comment is reached, this parameter holds the whole comment text in *reverse* order. This is characteristic for accumulating parameters and exploited, e.g., by a popular implementation of the list reversing function rev. For *fxp*, it has the disadvantage that after parsing the comment, its text must be traversed a second time for reestablishing the original order. In any case, this overhead is small compared to the cost of a non-tail-recursive function, which would need a deep recursion stack for parsing long comments.

2.7.4 Entity References

Let us now consider some more complex parsing functions. Both for general and parameter entity references the function that parses the reference also checks whether the entity is declared and whether a reference to the entity is allowed. The reference is illegal if the entity is declared as unparsed, or if it is already open: The latter would lead to an infinite recursion of references. Similarly, the function parsing a character reference checks whether the character is a legal XML character. An error is indicated by raising an exception after issuing an error message:

```
exception NoSuchChar of State
exception NoSuchEntity of State
```

These exceptions are easily handled by the calling function: The erroneous reference is ignored.

Function parseGenRef parses a general entity reference, starting after the initial "&" character. It first parses the entity name with parseName; it reports an error and raises SyntaxError if no name is found. Then it checks whether the next character is a semicolon. The entity name is mapped to its index and the replacement text declared for it is obtained.

```
fun parseGenRef dtd cq =
   let val (name,(c1,q1)) = parseName cq handle NotFound (c,q)
          ⇒ let val _ = reportError
                   (getPos q,ERR_EXPECTED("an entity name",[c]))
            in raise SyntaxError(c,q)
            end
      val _ = if c1=0wx3B (* ";" *) then ()
              else let val _ = reportError
                          (getPos q1,ERR_EXPECTED("';'",[c1]))
                   in raise SyntaxError(c1,q1)
                   end
      val idx = GenEnt2Index dtd name
      val (rep,ext) = getGenEnt dtd idx
```

The function could now return the entity's index and replacement text to the calling function. Before doing this it checks whether a reference to this entity is legal, reporting an error and raising NoSuchEntity if it is not:

followed by a ">".

```
      val _ = case rep
                of GE_NULL ⇒
                   let val _ = reportError
                          (getPos q,ERR_UNDEC_ENTITY name)
                   in raise NoSuchEntity q1
                   end
                 | GE_UNPARSED _ ⇒
                   let val _ = reportError
                          (getPos q,ERR_UNPARSED name)
                   in raise NoSuchEntity q1
                   end
                 | _ ⇒ if not (isOpen(q1,idx,false)) then ()
                        else let val _ = reportError
                                    (getPos q,ERR_RECURSIVE name)
                             in raise NoSuchEntity q1
                             end
   in ((idx,rep),q1)
   end
```

Unlike most of the other parser functions, parseGenRef does not return the character following the reference: If the entity is included, the first character of its replacement text has to be processed next.

One of the functions that use parseGenRef is parseAttValue. This function is given a quote character starting an attribute value literal. It reads characters up to the next occurrence of that quote character. It replaces character references with the character they name and includes the replacement text of general entity references. It also normalizes the text it reads by substituting a space character for each white-space character it encounters. The main work is done by its subfunction doit which accumulates the text of the literal and counts the nesting level of included entities in its parameters text and level.

```
fun parseAttValue dtd (quote,q) =
   let fun doit (level,text) (c,q) =
         case c
           of 0wx00 ⇒
                 if level > 0 then doit (level−1,text) (getChar q)
                 else let val err = ERR_ENDED_BY_EE "attribute value"
                          val _ = reportError(getPos q,err)
                      in (text,(c,q))
                      end
```

If the next character is an entity end then there are two cases: If it is the end of an included entity (level > 0) then it is skipped and the level is decremented. Otherwise this is an error, because the final quote character of a literal must be in the same entity as the initial one. This error is handled by assuming that the final quote was forgotten: doit returns the text read up to the entity end.

The case is more complicated if the next character is a "&", initiating a character or general entity reference. A new level and accumulated literal text are determined depending on the next character: A "#" indicates a character reference. It is parsed and its replacement character is added to the literal text. Otherwise a general entity reference is parsed by parseGenRef. This function can only return an internal or an external entity; the latter is forbidden in attribute values. If it is an internal entity, it is pushed onto the entity stack and the level is incremented.

```
| 0wx26 (* "&" *) ⇒
      let val (c1,q1) = getChar q
          val ((level1,text1),cq2) =
              (if c1=0wx23 (* "#" *)
               then let val (ch,q2) = parseCharRef q1
                        in ((level,ch::text),getChar q2)
                        end
               else let val ((idx,rep),q2) = parseGenRef dtd (c1,q1))
                        in case rep
                               of GE_INTERN(_,vec) ⇒
                                  let val q3 = pushIntern(q2,idx,false,vec)
                                  in ((level+1,text),getChar q3)
                                  end
                                | GE_EXTERN _ ⇒
                                  let val _ = reportError
                                      (getPos q1,ERR_EXTERNAL
                                                 (Index2GenEnt dtd idx))
                                  in ((level,text),getChar q2)
                                  end
                                | _ ⇒ raise InternalError
                        end)
               handle SyntaxError cq ⇒ ((level,text),cq)
                      | NoSuchEntity q ⇒ ((level,text),getChar q)
                      | NoSuchChar q ⇒ ((level,text),getChar q)
      in doit (level1,text1) cq2
      end
```

If either parseCharRef or parseGenRef raises an exception, then it is caught by the exception handler before function doit calls itself recursively. The exceptions are handled by continuing with the old level and literal text. Note that if the handler were after the recursive call of doit, this would destroy the tail-recursive nature of the function.

If the first character is neither an entity end nor a "&", then doit checks whether it is a white-space character. In this case it adds a space character to the literal text and continues. The character might also be the quote character that started the literal: If it is in an entity replacement text included during parsing of the literal (level<>0), then it is not recognized as the end of the literal. It is added to the literal text. Otherwise doit terminates.

```
| _ ⇒ if isS c
      then doit (level,0wx20::text) (getChar q)
      else if c=quote andalso level=0
           then (text,getChar q)
           else doit (level,c::text) (getChar q)
```

The body of the main function parseAttValue is simple: It first checks whether the supplied character is really a quote character and raises exception NotFound if it is not. Then the literal text is parsed by a call to doit. As in function parseComment, the text must be reversed before returning it.

```
         val _ = if quote=0wx22 (* ' *) orelse quote=0wx27 (* " *)
                    then () else raise NotFound (quote,q)
         val (text,cq1) = doit (0,nil) (getChar q)
      in (rev text,cq1)
      end
```

This function demonstrates how complex and technical the implementation of the parser functions is. For declarations and the content of elements, even more validity constraints have to be checked. Therefore, documenting these parsing functions goes beyond the scope of this description. The examples given so far, however, have pointed out the style in which the parsing functions are programmed. For more technical details, the reader is referred to the source code of *fxp* [Fxp99]. Here, we will content ourselves with describing one other important function.

2.7.5 Character Data in Content

A large part of a typical XML document is plain text, more precisely character data occurring in mixed content. A portion of character data containing no markup is parsed by the function do_chardata, similar to parseComment: It accumulates characters until it encounters a markup character or an entity end. Markup occurring in content always starts with a "&" (a reference) or a "<" which initiates a tag, comment, processing instruction or CDATA section. Instead of passing the parsed text to the calling function as part of its return value, function do_chardata reports the text directly to the application. For the time being, let us assume that this happens by a call to the function reportData, which expects a character vector as argument and returns nothing. Section 2.8 will present a better method for doing so.

```
   fun do_chardata (c,q) =
      let fun doit (yet,q) = let val (c1,q1) = getChar q
                             in case c1
                                   of 0wx00 ⇒ (yet,(c1,q1))
                                    | 0wx26 (* "&" *) ⇒ (yet,(c1,q1))
                                    | 0wx3C (* "<" *) ⇒ (yet,(c1,q1))
                                    | _ ⇒ doit (c1::yet,q1)
                             end
         val (cs,cq1) = doit ([c],q)
         val vec = Data2Vector(rev cs)
         val _ = reportData vec
      in cq1
      end
```

As in function parseComment, the accumulated text must be reversed before it is reported to the application. In addition to this extra effort, do_chardata contains another source of inefficiency: If a continuous piece of character data is very long, so is the accumulated text in parameter yet. Moreover, the list representation consumes much more memory than an array or a vector.

In order to optimize the space efficiency of this function, we implemented a variant do_chardata_array that uses an array buffer of fixed size for collecting the text. When the array is full, its content is reported to the application, and the array is reused for the next characters. As a consequence, the text is reported to the application in portions of a reasonable size. Moreover, we need only constant space for reading arbitrarily long character data sequences, and we avoid reversing of the list. These benefits outweigh the additional efforts for maintaining the array.

```
val dataBuffer = Array.array(DATA_BUFSIZE,0w0)

fun do_chardata_array (c,q) =
   let val _ = Array.update(dataBuffer,0,c)

      fun report i = reportData (Array.extract(dataBuffer,0,SOME i))

      fun takeOne (c,i,q) =
         if i<DATA_BUFSIZE
         then i+1 before Array.update(dataBuffer,i,c)
         else let val _ = report i
                  val _ = Array.update (dataBuffer,0,c)
              in 1
              end

      fun doit (i,q) =
         let val (c1,q1) = getChar q
         in case c1
              of 0wx00 ⇒ (c1,q1) before report i
               | 0wx26 (* "&" *) ⇒ (c1,q1) before report i
               | 0wx3C (* "<" *) ⇒ (c1,q1) before report i
               | _ ⇒ doit (takeOne(c1,i),q1)
         end
   in doit (1,q)
   end
```

In order to avoid allocating a new buffer each time do_chardata_array is called, the buffer is declared global to that function. The function starts by entering the first character into the buffer. Subfunction doit then counts the position in the buffer in its parameter i. Characters are added to the buffer by takeOne which flushes the buffer and reports the data it holds to the application if its size is exhausted.

The use of an array in this function is imperative style. Similar to the argument in Section 2.3.6, it is justified by the gain of efficiency for large documents with a low markup quota (cf. Section 3.1.2). Still this mutable data structure is visible only in a single function, parseMixedContent, and hidden from the rest of the parser.

This and previous examples demonstrate that most of the parsing functions are very complex and tedious to explain. We therefore cease documenting more parsing functions here. Those explained so far have given an impression of the style in which the parser functions are implemented. The interested reader is encouraged to read the source code of *fxp* in order to find out more details.

2.7.6 Validation of Element Content

While parsing the content of an element, the parser must validate its conformance with the content model declared for the element type. In order to do so, the parser constructs for each content model, which is a regular expression, a deterministic finite automaton (DFA), using the *Berry-Sethi construction* [BS86] (cf. Section 4.1.1) which is also known as *Glushkov construction* [Glu61]. This construction produces a non-deterministic automaton (NFA) for any regular expression. XML content models have the additional property of unambiguity. For unambiguous regular expressions, the Glushkov automaton is unambiguous [BW92, Brü93], i.e., we can obtain a DFA by adding an error state q_e and completing the transition relation to a function by filling it with transitions to that state. We can thus directly use this automaton for validating the content of elements.

Parsing of an element's content starts with the initial state q_0 of the DFA. For each start-tag encountered, the transition function of the DFA is applied to the current state q in order to obtain a new state q_1. If that state is the error state q_e, the element's content is invalid and an error message is issued. Now the following transitions will always end up in that error state again. In order to avoid repeated error messages for the content of this element, we therefore add a second error state q_f to the DFA. A transition from one of the error states then always leads to q_f, whereas erroneous transitions from non-error states lead to q_e. By reporting an error only in state q_e we produce just a single error message for each element with invalid content.

An invalid start-tag is thus handled by reporting an error and skipping the validation for the rest of the current element; the content of the element started by that start-tag is validated in any case. It is also easy to deal with an invalid end-tag: This is an end-tag for the current element although its content model requires further content. This error is handled by reporting an error message and finishing the current element.

The case is more difficult if the parser encounters an end-tag for an element type other than that of the current element. Though this constitutes a well-formedness error, i.e., a fatal error, we would still like to recover from this error without losing the context. We therefore distinguish three cases:

(1) If there is an open element with the same element type as the end-tag, we assume that the end-tag for the current element was forgotten. We therefore finish the current element without consuming the end-tag.

(2) Otherwise, if the current element requires no further content according to its content model, we assume that this is the end-tag for the current element but its element type was misspelt: The current element is finished and the end-tag is consumed.

(3) Otherwise, we assume that the author of the document accidentally inserted the end-tag. We therefore ignore it and continue parsing the content of the current element.

As usual in error recovery, these assumptions can also fail. For instance, in case (3) the end-tag might indeed be the misspelt end-tag for the current element. But that would mean that two errors coincide: The misspelling of the element type and the omission of some required content. Even in this rather unlikely case, the wrong decision will be corrected at the end-tag of the enclosing element, because the parser will then still be in the current element. Hence case (2) will apply and both the current and the enclosing element will be finished. After that, the parser has regained the context. Thus a wrong end-tag usually confuses the parser only for a small part of the document.

Observe the difference between this error recovery strategy and the one presented in 2.7.2: The latter handles syntax errors, i.e., well-formedness errors and is performed regardless of validation. In contrast to that, the handling of invalid tags described in this section happens only in validating mode. Moreover, though an end-tag that does not match the current element type is a well-formedness error, recovery from this error employs information declared in the DTD and thus depends on whether the parser is in validating mode – case (3) can never occur in non-validating mode.

2.8 The Programming Interface

In this section we describe *fxp*'s application interface. It is a functional variant of an event-based interface that vitally relies on SML's parametrized modules for customization. Its information set includes all required and most optional information items of the XML information set [W3C99d].

After describing the programming interface, we will point out the necessary modifications of the parser described so far in order to support this interface. Then we give some simple examples of using *fxp*'s programming interface for building XML applications.

2.8.1 Hooks - A Functional Variant of Event Handlers

fxp's programming interface pursues a functional variant of the event-based approach discussed in Section 1.2.3. In the classical imperative implementation of that approach the application's state is visible to event handlers through global variables; it can only be modified by destructive update.

The counterpart of global state in functional programming style is an accumulating parameter which is passed around all functions depending on that state. In order to illustrate this technique, let us consider a small C function:

```
int sum_up (int n) {
    int i,sum=0;
    for (i=1; i⩽n; i++)
        sum=sum+i;
    return sum;
}
```

This function computes the sum of all integers between 1 and its argument by consecutively adding them to variable sum. We could implement an analogous SML function, using a reference type for sum in order to enable destructive updates on it:

```
fun sum_up n =
    let val sum = ref 0
        fun loop i = if i>n then ()
                     else (sum := !sum+i; loop (i+1))
        val _ = loop 1
    in !sum
    end
```

Since SML has no concept of for-loops, function loop is used for accomplishing the iteration. Variable sum can be viewed as the global state of the computation.

Hook	XML Event
hookError	error
hookWarning	warning
hookStartTag	start-tag
hookEndTag	end-tag
hookData	segment of character data
hookCData	CDATA section
hookCharRef	character reference
hookProcInst	processing instruction
hookComment	comment
hookDecl	element, attribute, notation or entity declaration
hookDocType	start of the DTD
hookSubset	start of the internal subset
hookExtSubset	start of the external subset
hookEndDtd	end of the DTD
hookGenRef	general entity reference
hookParRef	parameter entity reference
hookEntEnd	end of an included entity
hookXml	start of the document entity
hookFinish	end of the document

Table 2.2: The hooks in *fxp* and their purposes.

Nevertheless, using a mutable reference type is very bad SML programming style. It would better be realized as an additional parameter of function loop:

```
fun sum_up n =
   let fun loop sum i = if i>n then sum
                        else loop (sum+i) (i+1)
   in loop 0 1
   end
```

Function loop *accumulates* the result in parameter sum. This implementation avoids the use of reference types and destructive updates. On the other hand, function loop now has one additional parameter and its return value comprises an additional component, because parameter sum must be passed around the computation.

In a similar way, *fxp*'s programming interface avoids destructive updates on the application's state in event handlers. The application defines a data type AppData representing the part of its state affected by the event handlers. We call this information the *application data*. During parsing, a value of this type is maintained by the parser. Upon triggering an event, the event handler receives the application data as an additional parameter and returns it – possibly modified – to the parser. The modified application data is then an argument to the next event, and so forth. In order to distinguish this kind of event handlers from the imperative variant we call them *hooks*. The effects of all hooks are thus accumulated into one value of type AppData.

In order to implement hooks in *fxp*, the parser must be provided with the type of the application data and with the functions implementing the hooks. The easiest way of doing so is to pack them into a structure Hooks, fulfilling the

```
signature Hooks =
  sig
    type AppData
    type AppFinal

    val hookError      : AppData ∗ HookData.ErrorInfo       → AppData
    val hookWarning    : AppData ∗ HookData.WarningInfo     → AppData

    val hookStartTag   : AppData ∗ HookData.StartTagInfo    → AppData
    val hookEndTag     : AppData ∗ HookData.EndTagInfo      → AppData

    val hookData       : AppData ∗ HookData.DataInfo        → AppData
    val hookCData      : AppData ∗ HookData.CDataInfo       → AppData
    val hookCharRef    : AppData ∗ HookData.CharRefInfo     → AppData

    val hookProcInst   : AppData ∗ HookData.ProcInstInfo    → AppData
    val hookComment    : AppData ∗ HookData.CommentInfo     → AppData
    val hookDecl       : AppData ∗ HookData.DeclInfo        → AppData

    val hookDocType    : AppData ∗ HookData.DtdInfo         → AppData
    val hookSubset     : AppData ∗ HookData.SubsetInfo      → AppData
    val hookExtSubset  : AppData ∗ HookData.ExtSubsetInfo   → AppData
    val hookEndDtd     : AppData ∗ HookData.EndDtdInfo      → AppData

    val hookGenRef     : AppData ∗ HookData.GenRefInfo      → AppData
    val hookParRef     : AppData ∗ HookData.ParRefInfo      → AppData
    val hookEntEnd     : AppData ∗ HookData.EntEndInfo      → AppData

    val hookXml        : AppData ∗ HookData.XmlInfo         → AppData
    val hookFinish     : AppData                            → AppFinal
  end
```

Figure 2.16: The Hooks signature.

signature in Figure 2.16. The purposes of the single hooks are listed in Table 2.2. A hook expects as argument the current application data and – except for hookFinish – the information belonging to the signaled event, and returns the modified application data.

The data types providing the event-specific information passed to a hook are defined by structure HookData. Each information item contains at least the position in the document where the event occurred; for some events even two positions are specified: a start position and an end position. E.g., type ErrorInfo describes an error by means of its position and an error description. A start-tag is described by the positions of its first and last character, the index of the element, the list of attribute specifications and a boolean flag, indicating whether it is an empty-element tag:

```
type ErrorInfo = Position ∗ Error
type StartTagInfo = Position ∗ Position ∗ int ∗ AttSpec list ∗ bool
```

The information set provided through *fxp*'s hooks is sufficient for producing a character-by-character identical copy of the document instance. Except for white space and parameter entity references within declarations, this is also possible for the DTD.

In addition to type AppData, a Hooks structure must define a type AppFinal. This type is the result type of the parser: At the end of the document, the parser

```
funsig Parse ( structure Dtd     : Dtd
               structure Hooks   : Hooks
               structure Resolve : Resolve
               structure Options : Options ) =
   sig
      val parseDocument : Uri.Uri option → Dtd.Dtd option
                          → Hooks.AppData → Hooks.AppFinal
   end
```

Figure 2.17: The Parser functor.

calls hookFinish in order to *finalize* the accumulated application data. This is useful because values of type AppData often contain auxiliary information: E.g., if an application collects information from the document entity, it must ignore all events occurring within entity replacement texts. For this reason, the hooks might maintain a counter indicating the nesting depth of included entity references. hookFinish then removes this counter because it is of no interest to the application. Another example for the benefits of finalization is the following: An application might collect section titles in order to compile a table of contents. Since the application data is an accumulating parameter, these titles are collected in reverse order (cf. 2.7.3). AppFinal can be used for reestablishing the original order. For a similar example, see 2.8.5.

2.8.2 Functorizing the Parser

Since *fxp* is reentrant it must support running several independent instances of the parser. More precisely, it must be possible to have each instance of the parser supplied with its own set of hooks. For instance, resolving a public identifier may involve parsing of an XML catalog. The hooks for processing the catalog are presumably others than those processing the main document.

The parser is therefore implemented as an SML *functor* (cf. page 23), expecting the Hooks structure as argument. The application can then generate several instances of the parser for different purposes. In addition to the hooks, the parser functor is parametrized with three other structures:

Options: This structure supplies the parser with its options as described in Section 2.4. It is useful to have each instance of the parser run with its own set of options: E.g., XML catalogs frequently have no DTD. Therefore a catalog is parsed in non-validating mode, even if the main document is validated.

Resolve: This structure provides a single function for resolving an external identifier to a URI:

```
val resolveExtId : Base.ExternalId → Uri.Uri
```

In the simplest case this function returns the system identifier that is part of the external identifier. If the parser supports XML catalogs, however, this is the function that searches the catalog.

Dtd: The implementation of the DTD tables can be provided by the application. In most cases this is the Dtd structure from Section 2.6.3, but the application can also provide a more efficient implementation, or enhance the functionality of the DTD tables. For instance, the operations on the DTD tables might be wrapped into functions producing debugging or statistical information. On the other hand, the application can *hard-code* element types or attributes to fixed indices. E.g., in order to collect `href` attributes in an XHTML document, one might use the following implementation of the Dtd argument structure:

```
structure HrefDtd =
   struct
      open Dtd

      val hrefData = UniChar.String2Data "href"

      fun initializeDtd () =
         let val dtd = Dtd.initializeDtd()
             val _ = AttNot2Index dtd hrefData
         in dtd
         end

      val hrefIdx = AttNot2Index (initializeDtd()) hrefData
   end
```

A `href` attribute will then always have index hrefIdx; searching and comparing can be done with this constant rather than the list of characters [0wx68,0wx72,0wx65, 0wx66]. This is reasonable also because element types are passed to the hooks by means of their indices in the DTD, not by their names.

The signature of the parser functor is given in Figure 2.17. It defines a single function parseDocument which, given an optional URI of a document and an optional DTD, parses that document – if no URI is given, the document is read from the standard input (cf. 2.3.4). It returns the finalized value of the application data received in its third argument, modified by the hooks during parsing.

If the optional Dtd argument is given as NONE, the parser initializes the DTD tables with the initializeDtd function. In this case hooks have no access to the DTD because it is not provided as an argument to them. For many applications this is not necessary indeed. If the hooks need to access the DTD, the application must initialize the tables itself, incorporate them into the application data and pass them to the parser in its Dtd argument.

2.8.3 Implications on the Implementation of the Parser

In order to report events to the application via hooks, the application data must be passed through all functions that might involve a call to a hook. The implementation presented so far must therefore be altered to make each function have an additional argument of type AppData and return a modified value of that type. Because hooks are also used for signaling errors, this affects all functions that might produce an error. E.g., function getChar in structure Entities must now have type

```
val getChar : AppData * State → UniChar.Char * AppData * State
```

Furthermore, exceptions such as SyntaxError must comprise the application data; otherwise the function handling the exception had no application data to continue with:

```
exception NotFound of UniChar.Char * AppData * State
exception SyntaxError of UniChar.Char * AppData * State

exception NoSuchChar of AppData * State
exception NoSuchEntity of AppData * State
```

In order to illustrate how the parsing functions must be modified for supporting hooks, let us consider function parseComment from Section 2.7.3. It is modified to the following:

```
fun parseComment (a,q) =
   let fun check_end yet (c,a,q)
         if c=0wx2D (* "-" *)
         then let val (c1,a1,q1) = getChar(a,q)
                in if c1=0wx3E (* ">" *) then (yet,getChar(a1,q1))
                   else doit (0wx2D::0wx2D::yet) (c1,a1,q1)
                end
         else doit (0wx2D::yet) (c,a,q)

      and doit yet (c,a,q) =
         if c=0wx2D (* "-" *) then check_end yet (getChar(a,q))
         else if c<>0wx00 then doit (c::yet) (getChar(a,q))
         else let val a1 = hookError (a,(getPos q,ERR_ENDED_BY_EE "comment"))
               in (yet,(c,a1,q))
               end

      val (cs,(c1,a1,q1)) = doit nil (getChar aq)
      val vec = Data2Vector (rev cs)
      val a2 = hookComment (a1,(getPos q,getPos q1,vec))
   in (c1,a2,q1)
   end
```

In addition to the state q of the entity stack, the auxiliary functions doit and check_end now maintain the application data a. Errors are reported through hookError instead of function reportError. Furthermore, parseComment no longer returns the text of the comment to the calling function. Instead, it reports the comment to the application through hookComment, together with its start and end position.

Similar modifications must be made to all functions in the parser modules, the entity manager and the DTD manager. The DTD tables and the UNICODE frontend are not affected because they report errors by raising exceptions. This allows for employment of the UNICODE frontend for other applications, independently of *fxp*.

2.8.4 Functor Dependencies

Parse is not the only module of the parser which is implemented as a functor: All modules that make use of hooks must be functorized either. E.g., the Entities structure is now a functor:

```
signature Entities =
  sig
    type State
    type AppData

    val getChar : AppData ∗ State → UniChar.Char ∗ AppData ∗ State

    . . .
  end

functor Entities (structure Hooks : Hooks) : Entities =
  struct
    open Hooks
    . . .
  end
```

The types of the functions defined by Entities depend on the type of the application data. In order to specify them in the signature, type AppData must also be included. Similarly, types AppData and State must appear in the signature of functor ParseBase which, among others, defines the exceptions NotFound and SyntaxError.

```
signature ParseBase =
  sig
    type State
    type AppData

    exception NotFound of UniChar.Char ∗ AppData ∗ State
    exception SyntaxError of UniChar.Char ∗ AppData ∗ State
    . . .
  end
```

Functor ParseNames is now parametrized by instances of these two functors, Entities and ParseBase, both defining types AppData and State. In order to make ParseNames be correctly typed, the programmer must ensure that the definitions of these types are identical in both structures. Therefore the parameter list of functor ParseNames contains *equality constraints* for them:

```
functor ParseNames
          (structure Entities   : Entities
           structure ParseBase : ParseBase
           sharing Entities.State = ParseBase.State
           and  Entities.AppData  = ParseBase.AppData ) : ParseNames =
  struct
    . . .
  end
```

Functor ParseNames is a simple case of a parser module: It only depends on ParseBase and Entities. Most other modules additionally depend on the Hooks, Dtd and Options structures, plus the parser modules whose functions they use. Figure 2.18 shows the dependencies among the parser modules. The main functor Parse, e.g., depends on nine other parser modules. This leads to numerous equality constraints in the functor headers: In an early version of *fxp*, which realized this implementation of the functor hierarchy, 261 lines of code were required only for the headers of functors. The disadvantage is that these technical details obscure the source code, deviating the attention from the essentials. Moreover, structural changes to the program, which happen rather

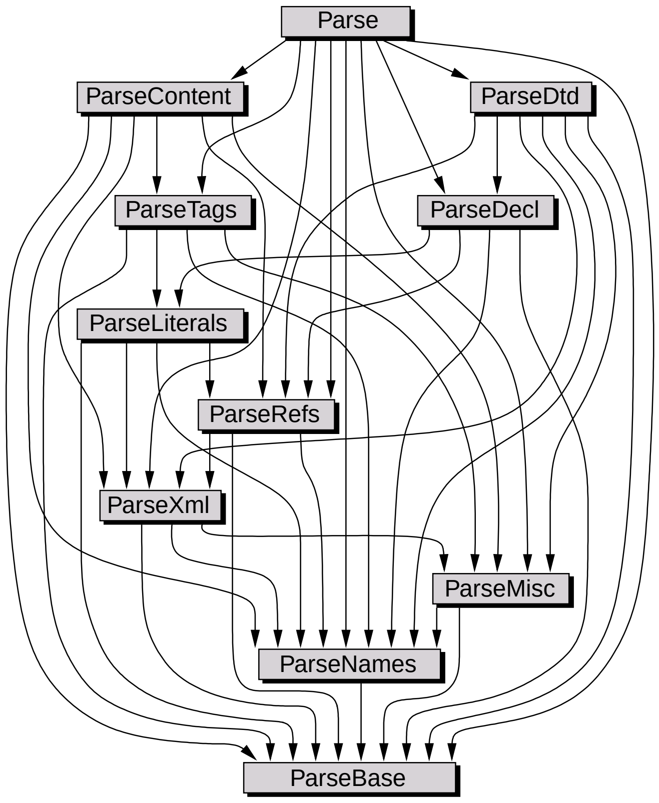


Figure 2.18: Dependencies between the parser modules.

frequently especially during program development, require a large effort in adapting the equality constraints.

In order to find a more elegant and concise realization, we refrained from using equality constraints by avoiding repeated declarations of the same type in different signatures. To start with, we incorporated all parameters of the parser functor, i.e., Hooks, Dtd, Options and Resolve, together with the entity manager, the DTD manager, and the ParseBase module, into a single signature:

```
signature ParseBase =
   sig
      include Dtd Hooks Resolve Options Entities DtdDeclare DtdAttributes

      exception NotFound of UniChar.Char ∗ AppData ∗ State
      exception SyntaxError of UniChar.Char ∗ AppData ∗ State
      ...
   end
```

Implementing a functor ParseBase providing all functions specified in this signature is possible without equality constraints. The hierarchy of parser modules is now based on this functor: From the module dependencies we can obtain a topological ordering on the modules, with ParseBase as the least and Parse as the greatest element. Figure 2.19 shows this sequential ordering. Each functor in this sequence is now defined to include the signature of the preceding functor in its output signature. E.g., ParseMisc is now defined as follows:

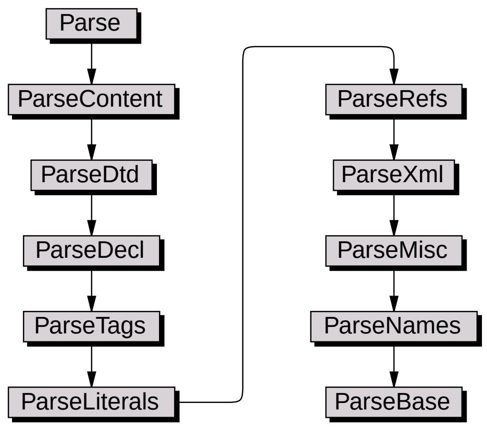


Figure 2.19: The topological order obtained from the functor hierarchy.

```
signature ParseMisc =
   sig
      include ParseNames
      . . .
   end

functor ParseMisc ( structure ParseBase : ParseBase ) : ParseMisc =
   struct
      structure ParseNames = ParseNames
                              (structure ParseBase = ParseBase)
      open ParseNames
      . . .
   end
```

Each functor thus includes all functions defined by the modules that come earlier in the sequential ordering. The effect is that, e.g., the definitions from ParseTags are visible in ParseDtd though this module doesn't depend on them. On the other hand, the definitions from ParseNames are now available to each module through the signature of its predecessor. Each functor depends only on a single parameter structure ParseBase, except for the main parser module Parse. It expects the four argument structures coming from the application, feeds them to the ParseBase functor and supplies the result structure to its predecessor ParseContent:

```
functor Parse ( structure Dtd : Dtd
                structure Hooks : Hooks
                structure Resolve : Resolve
                structure ParserOptions : ParserOptions) :
   sig
      val parseDocument : Uri.Uri option → Dtd.Dtd option
                          → Hooks.AppData → Hooks.AppFinal
   end
=  struct
      structure ParseBase = ParseBase ( structure Dtd = Dtd
                                        structure Hooks = Hooks
                                        structure Resolve = Resolve
                                        structure Options = Options)
```

```
        structure ParseContent = ParseContent
                                  (structure ParseBase = ParseBase)
        . . .
    end
```

This realization of the module hierarchy is much more concise than the previous variant: It only requires two lines per functor, one for the **include** statement in its signature, and one for generating an instance of the immediately preceding functor. Together with the four lines for generating the ParseBase instance, this is a total of only 24 lines. Moreover, each functor is applied exactly once in order to obtain an instance of the parser. Interestingly enough, the SML/NJ compiler produces a slightly smaller heap image for this implementation, though the signatures of the functors are larger; in execution time we could not measure a difference between the two variants.

In order to find out whether dependent functors introduce a run-time overhead at all, we packed all parser modules into a single functor. This functor, having a size of more than 300 KB of source code, certainly violates the paradigm of modularized, well-structured programs. Despite that, its execution speed is not at all higher than that of the hierarchical functors. Going even one step further, we defunctorized this large parser module, making it explicitly dependent on a fixed set of Hooks, Dtd, Resolve and Options structures. The resulting program is exactly as fast as the implementation with the functor hierarchy. We can therefore attest that SML functors are a very elegant means of program modularization without sacrificing efficiency.

2.8.5 Building Applications with *fxp*

fxp provides a rich information set through its hooks interface. Many applications, however, are only interested in a subset of that information: A formatter is probably not interested in comments, and a querying tool will only search the document instance and ignore the DTD. Therefore we provide a set of hooks that simply return the application data unchanged:

```
structure IgnoreHooks =
    struct
        fun hookError(a,_) = a
        fun hookWarning(a,_) = a
        fun hookStartTag(a,_) = a
        ...
        fun hookXml(a,_) = a
        fun hookFinish a = a
    end
```

These functions are polymorphic: They do not depend on the type of the application data and can thus be used with arbitrary types. The only exception is hookFinish which requires types AppData and AppFinal to be equal. An application must now only define the hooks that have a different behavior. E.g., the hooks for a validator are implemented in a few lines, because it only prints errors and warnings but ignores all other events:

```
structure CheckHooks =
    struct
        open TextIO Errors IgnoreHooks
```

```
      type AppData = OS.Process.status

      fun message(pos,msg) = output(stdErr,Position2String pos^": "^msg)

      fun hookError (_,(pos,err)) =
         OS.Process.failure before message(pos,errorMessage err)
      fun hookWarning (status,(pos,warn)) =
         status before message(pos,warningMessage warn)
   end
```

Except for hookError and hookWarning, all hooks are taken over from structure IgnoreHooks. Another example is the application already mentioned on page 69: It collects all attributes named `href`. For this purpose we defined a structure HrefDtd, hard-coding this attribute name to a constant index hrefIdx. This is used in the definition of the hooks:

```
structure HrefHooks =
   struct
      open Base HrefDtd IgnoreHooks

      type AppData = UniChar.Vector list
      type AppFinal = AppData

      fun findHref nil = NONE
        | findHref ((idx,attPres)::rest) =
            if idx <> hrefIdx then findHref rest
            else case attPres
                    of AP_PRESENT(vec,_) => SOME vec
                     | AP_DEFAULT(vec,_) => SOME vec
                     | _ => findHref rest

      fun hookStartTag (a,(_,_,_,attSpecs,_)) =
         case findHref attSpecs
           of NONE => a
            | SOME x => x::a

      val hookFinish = rev
   end
```

In order to identify a `href` attribute, function findHref need only compare its index with hrefIdx. Note that due to the accumulative nature of the AppData argument, the attribute values are collected in reverse order. Therefore function hookFinish is defined to be the list reversing function and reestablishes the original order.

2.8.6 Implementing a Tree-Based Interface

The hooks interface of *fxp* its event-based; nevertheless a tree-based interface can easily be implemented on top of the hooks. Figure 2.20 shows an implementation of such an interface. For brevity, it has only a restricted information set: It ignores the DTD, comments, processing instructions and the entity structure of the document. It is, however, easy to extend the implementation to supply all this information.

The Tree data type is simple: A tree is either a piece of text or an element consisting of a start-tag and a list of trees as content. The application data represents the partial document tree constructed so far. It contains in its stack

```
structure TreeData =
   struct
      exception IllFormed

      type Tag = int * Base.AttSpec list
      datatype Tree = TEXT of UniChar.Vector
                    | ELEM of Tag * Content
      withtype Content = Tree list
   end


structure TreeHooks =
   struct
      open IgnoreHooks TreeData UniChar

      type AppData = Content * (Tag * Content) list
      type AppFinal = Tree

      val appStart = (nil,nil)

      fun hookStartTag ((content,stack), (_,_,elem,atts,empty)) =
         if empty then (ELEM ((elem,atts),nil)::content,stack)
         else (nil,((elem,atts),content)::stack)

      fun hookEndTag ((_,nil),_) = raise IllFormed
        | hookEndTag ((content,(tag,content')::stack),_) =
         (ELEM (tag,rev content)::content',stack)

      fun hookData ((content,stack),(_,_,vec,_)) =
          (TEXT vec::content,stack)
      fun hookCData ((content,stack),(_,_,vec)) =
          (TEXT vec::content,stack)
      fun hookCharRef ((content,stack),(_,_,c)) =
          (TEXT(Data2Vector [c])::content,stack)

      fun hookFinish ([elem],nil) = elem
        | hookFinish _ = raise IllFormed
   end


functor ParseTree ( structure Dtd      : Dtd
                    structure Options  : Options
                    structure Resolve  : Resolve) :
   sig
      val parseTree : Uri.Uri option → Dtd.Dtd option → TreeData.Tree
   end
= struct
      structure Parser = Parse ( structure Dtd      = Dtd
                                 structure Hooks    = TreeHooks
                                 structure Options  = Options
                                 structure Resolve  = Resolve)
      open Parser TreeHooks

      fun parseTree uri dtd = parseDocument uri dtd appStart
   end
```

Figure 2.20: A simple tree-based interface on top of hooks.

component for each ancestor element of the current position, its start-tag together with the list of its – already complete – left siblings; component content holds the children of the current element that are known so far. At the beginning of the parse both components are empty. After the whole document has been parsed, the stack must be empty and a single element tree must have been constructed. In this case function hookFinish returns that element; otherwise it raises an exception.

The three functions hookData, hookCData and hookCharRef add the piece of text reported to them to the content of the current element. The hook for a start-tag pushes that tag together with the content of the current element onto the stack. The element started by that tag now becomes the current element. Function hookEndTag reverses the content of the current element in order to compensate for the reversing effect of accumulation. Its tag is popped from the stack and combined with its content. The constructed tree is then prepended to the content of the parent element which now becomes the current element.

2.8.7 Catalog Support

The parser functor expects as one of its parameters a structure Resolve. This structure defines a single function resolveExtId which generates a system identifier, i.e., a URI, from an external identifier. It raises exception NoSuchFile if that is not possible. The simplest implementation of this function is to combine the system identifier and the base URI that are part of the external identifier (cf. 2.6.2):

```
structure ResolveNull : Resolve =
  struct
    open Errors Uri

    fun resolveExtId (pub,sys) =
      case sys
        of NONE ⇒ raise NoSuchFile "No system identifier available"
         | SOME (base,file) ⇒ uriJoin(base,file)
  end
```

On the other hand, structure Resolve can also be used to implement catalog support. In this case function resolveExtId searches a catalog for an entry matching the external identifier. The implementation of a suitable data structure for catalogs and a search routine is rather technical and therefore left out here. Note only that function resolveExtId must access the catalog through a global variable because the external identifier is its only argument. Parsing of the catalog and initialization of the catalog data structure must be performed either by side-effects in resolveExtId, or by an initialization procedure before the start of the parser.

Parsing a catalog is itself an XML application similar to the `href` attribute collector in 2.8.5: An XML catalog is according to a DTD that is at least a superset of the following:

XML Example 19

```
<!ELEMENT Map ANY>
<!ATTLIST Map PublicId CDATA #REQUIRED
              HRef     CDATA #REQUIRED>

<!ELEMENT Remap ANY>
<!ATTLIST Remap SystemId CDATA #REQUIRED
                HRef     CDATA #REQUIRED>

<!ELEMENT Delegate ANY>
<!ATTLIST Delegate PublicId CDATA #REQUIRED
                   HRef     CDATA #REQUIRED>

<!ELEMENT Extend ANY>
<!ATTLIST Extend HRef CDATA #REQUIRED>

<!ELEMENT Base ANY>
<!ATTLIST Base HRef CDATA #REQUIRED>
```

Everything except for the elements and attributes in these declarations is considered a comment and ignored when parsing a catalog. The hooks need only find these elements and extract the required attributes in order to assemble the catalog entries.

Chapter 3

Analysis, Comparison and Discussion

In this chapter, we analyze the run-time behavior of *fxp* and compare it to that of XML parsers written in other programming languages. We conclude by critically discussing the choice of SML as the implementation language.

3.1 Run-Time Analysis and Comparison

In this section, we analyze the execution time of *fxp* with respect to three areas of interest: First, we investigate how much the single modules contribute to the overall run-time. We then examine the effect of two optimizations on the execution time. Finally we compare *fxp*'s execution speed to other publicly available XML parsers.

3.1.1 Profiling the Parser

In order to optimize *fxp*, we are interested in those parts of the program that contribute the largest share to the execution time of the parser. We can expect to obtain the largest effect from optimizations in these parts. In order to detect these spots, we profiled the execution of the validator from 2.8.5. Since it adds no functionality to the parser, we called this program *fxp*; we used it for all statistics in this section. Using the profiler of the SML/NJ system we tracked the execution of *fxp* on four different input documents:

`rec` is the XML recommendation coded in XML [W3C98b]. Its size is about 185 KB, 30 of which constitute the DTD; slightly more than 100 KB are character data in content, i.e., the markup quota is around 45%. The encoding is LATIN1.

`data-64` is derived from `rec` by repeating each piece of character data in its content 64 times; it is approximately 6.7 MB large; due to the large amount of character data, its markup quota is very low.

`cjk` was created by concatenating the document instances of the 19 parts of the Web CJK-English Character Dictionary [Mul99]. With only 5 KB its DTD is very small in relation to the document's size of nearly 3 MB. This

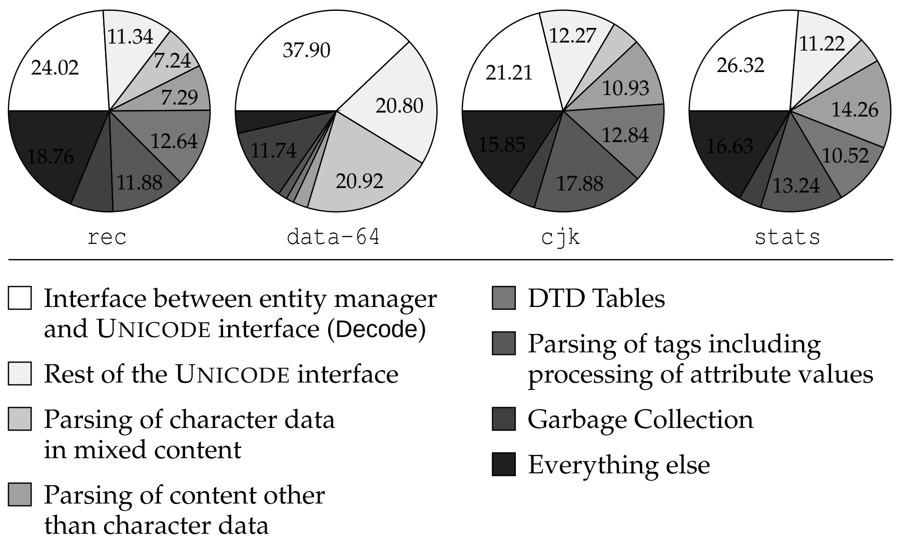


Figure 3.1: The distribution of *fxp*'s execution time among specific program parts (in %, values less than 7% not numbered).

document is a collection of dictionary entries with rather short pieces of character data; the markup quota exceeds 60%. Because it contains many East-Asian characters requiring 16 significant bits, this document is encoded in UTF-16.

`stats` is an XML version of 1998 baseball statistics [Har99]. This document consists mainly of markup; only 75 KB of character data with a total document size of nearly 620 KB make a markup quota of about 88%. The document is encoded in UTF-8 but contains only ASCII characters.

The profiling results are shown in Figure 3.1. Most notably, *fxp* spends a large amount of its execution time in the `Decode` structure, which is the interface between the UNICODE frontend and the entity manager. This structure contributes about one quarter of the total execution time for the `rec`, `cjk` and `stats` documents. Due to the high markup quota of these documents, the parser also spends a considerable amount of time on parsing of tags and processing of attribute values. The rest of the run-time is distributed relatively uniformly among the other parts of the parser. Interestingly enough, garbage collection consumes only a small amount of execution time.

The `data-64` document is a more extreme case: Due to its very few markup and long sequences of plain character data, it spends more than three quarters of its run-time in the UNICODE frontend and for parsing character data; the other program parts are nearly insignificant.

3.1.2 Imperative Optimizations

The profiling results showed that a major part of *fxp*'s execution time is spent in the interface between the decoder and the entity manager; for documents with few markup, parsing of character data contributes another very large share. In 2.3.6 we mentioned an optimization that uses an array for decoding a whole

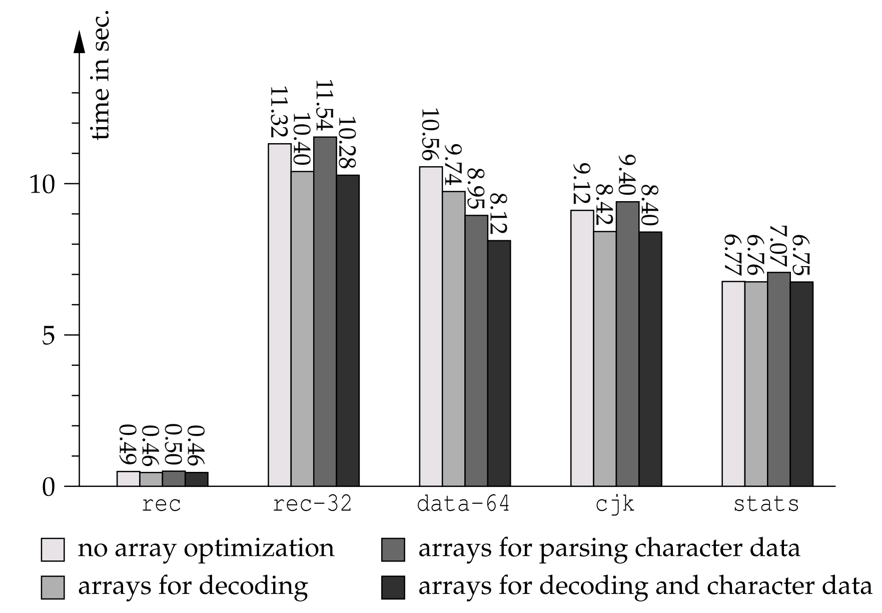


Figure 3.2: Effects of two imperative optimizations on execution times.

sequence of UNICODE characters at a time; a similar optimization was introduced for parsing of character data in 2.7.5.

While both of these optimizations affect only a few lines of source code, their impact on the execution time is significant. In order to measure the increase in speed we built four versions of *fxp*: Incorporating none of these optimizations, either of them or both of them. The obtained programs were run on the example documents from 3.1.1. Due to the relatively small size of the `rec` and `stats` documents, the difference was hardly measurable. We therefore used two other example documents:

`rec-32` is the `rec` document with the document instance repeated 32 times, encoded in LATIN1. Its size is about 5 MB with a markup quota of 33%.

`stats-4` is the `stats` example with the document instance repeated 4 times, encoded in UTF-8. Its is about 2.5 MB large and has a markup quota of about 88%.

Figure 3.2 summarizes the measured execution times. The `rec` example is too small for drawing conclusions. For most of the other examples, the optimization of the UNICODE frontend alone (2nd column) speeds up the parser by about 8%; only in case of the `stats` document, the difference is hardly perceivable. On the other hand, the optimization for character data (3rd column) does not speed up the parser in most cases: Only for the `data-64` example with its long chunks of character data a significant enhancement is achieved; for the other documents it even seems to slow down the execution. The combination of both optimizations, however, achieves the largest speed-up in all cases, ranging from less than 1% for `stats-4` to nearly 24% for `data-64` with an average of about 10%. This certainly justifies the use of imperative features for these optimizations.

3.1.3 Comparison to XML Parsers Written in Other Languages

As pointed out in Section 1.2.6, the XML area is dominated by software written in imperative and object-oriented programming languages such as C, C++ or JAVA. One of the goals in the development of *fxp* was to show that functional programming languages are also well-suited for XML processing. We therefore compared *fxp* with several other publicly available parsers.

3.1.3.1 Other Functional Programming Languages

Besides *fxp*, two other XML parsers are implemented in functional programming languages: *tony* written in OCAML and *HaXml* written in HASKELL (cf. 1.2.6). Both lack support for many of XML's features and must therefore be seen as experimental software. Nevertheless, at least the compiler used for *tony* is supposed to produce faster programs than the SML/NJ compiler: OCAML is famous for executables twice as fast as those generated by SML/NJ.

Both *tony* and *HaXml* provide only a tree-based interface and come with a pretty-printer as the simplest example application. In order to compare them, we implemented a similar pretty-printer using *fxp*. Because *tony* and *HaXml* implement only a subset of XML, most of our example documents can not be processed by them; we therefore measured the execution times for pretty-printing the following document:

`hamlet` is Shakespeare's play coded in XML [Bos99]. It has a simple DTD and a very flat structure – it is basically a sequence of verses. Its size is 273 KB with a markup quota of about 36%.

Comparing execution times revealed that *fxp* is about 12 times faster than *tony*; *HaXml* even runs 30-40 times longer than *fxp*, requiring nearly double the memory used by *fxp*. This strengthens our proposition that pure functional programming languages such as HASKELL are unsuited for efficient implementation of I/O-intensive applications.

We conclude that, besides being the only complete XML implementation in a functional language, *fxp* is also the only functional XML parser with a time efficiency that allows for processing of large real-world documents.

3.1.3.2 Imperative and Object-Oriented Programming Languages

There is a vast number of publicly available XML parsers written in imperative or object-oriented programming languages. The majority of these parsers is non-validating. Validation is often considered an easy task as compared to parsing and checking for well-formedness. Indeed, *fxp* is only slightly slower in validating mode than in non-validating mode.

Nevertheless, the capability of validation has a significant impact on the design of the software. In particular, data types representing the declarations in the DTD must be far more complex if validation is performed. Moreover, validation noticeably increases the size of the program. This can have a substantial influence on its run-time behavior: E.g., a non-validating parser might be small enough to fit into the processor cache, making its execution extremely fast, whereas a larger, validating parser might suffer from many cache misses slowing down its execution. Comparing a non-validating parser to a validating parser running in non-validating mode is therefore problematic.

	Size of		Number of					
	DTD	Instance	Decl.	Elem.	Data Char.	Encoding	Structure	Data Quota
`rec`	30 KB	155 KB	280	2306	104040	LATIN1	rich	54%
`xsl`	43 KB	358 KB	341	7328	224814	UTF-8	rich	54%
`hamlet`	1 KB	273 KB	22	6636	179656	UTF-8	flat	64%
`stats`	2 KB	600 KB	45	24698	75173	UTF-8	medium	12%
`cjk-9`	1.2 MB	5 KB	81	24031	126932	UTF-16	medium	21%

Table 3.1: Details of the example documents used for comparison.

For this reason, we compared *fxp*'s execution speed only with that of validating parsers; still we could not take into account all of them. We chose to compare *fxp* with the following parsers (cf. 1.2.6):

✧ the C-program *rxp*,

✧ *sp*, written in C++,

✧ *xml4j*, the JAVA parser from IBM, and

✧ *xmlproc*, implemented in PYTHON.

All of these parsers come with a validator as an example application. We can therefore well compare them to *fxp*. Besides `rec`, `hamlet` and `stats` from the previous sections, we used two additional documents:

`xsl` is an XML version of an early working draft of XSL [W3C99e, W3C98c], the stylesheet language commonly used with XML. It is 358 KB large with an additional DTD size of 43 KB; in structure being similar to `rec`, its markup quota is about 46%. The character encoding is UTF-8 though actually no non-ASCII characters occur in the document;

`cjk-9` is part 9 of the Web CJK-English Character Dictionary [Mul99] used for creating the `cjk` document. Encoded in UTF-16, it is about 1.2 MB large and has a markup quota of nearly 80%. Because *xmlproc* does not support multi-byte encoded characters, it can not parse this document.

All of these documents are real-world examples; their details are summarized in Table 3.1. We measured the execution times of all programs on these documents, except for *xmlproc* which is not capable of processing `cjk-9` due to its encoding. The times were taken on a two-processor 400 MHz Pentium-II running LINUX. Each program was run once for establishing it in the computer's hard-disk cache. Then it was executed 10 times on each document, and the measured times were averaged, disregarding the largest and the smallest value.

The results are illustrated in Figure 3.3. As expected, the C and C++ parsers are by far the fastest. It is also no surprise that the PYTHON program is very slow: PYTHON is a scripting language and was not designed for developing large applications.

The most interesting comparison is between *fxp* and the JAVA parser: For the smaller documents *fxp* is faster, but it is outperformed by *xml4j* for the rather large `cjk-9`. The reason is that the employed JAVA Virtual Machine (JVM)

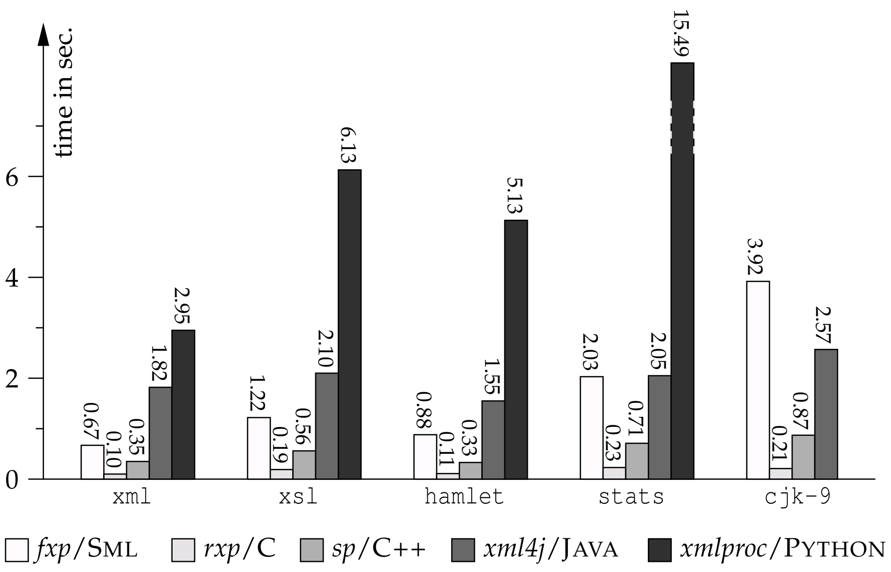


Figure 3.3: Execution time comparison with parsers written in imperative and object-oriented languages.

uses a *just-in-time* compiler (JIT) for translating the program to native code, which can be executed much more efficiently than interpreting the JVM byte-code. It has therefore a rather long start-up phase of more than one second; the larger the document, the less significant is this start-up time, and the faster is the parser in comparison to *fxp*.

Even if we ignore the start-up time, the JAVA parser is still less than twice as fast as *fxp*. If we use a JVM without a JIT, then *fxp* is faster than *xml4j* by a factor between 1.5 and 2. With a compiler producing more efficient code than SML/NJ, *fxp* could also compete with JIT-compiled JAVA parsers. The compiler for the OCAML language [LRV+99], e.g., which is very similar to SML, generates native code that is about twice as fast as a comparable program compiled by SML/NJ.

We believe that a performance comparable to that of the OCAML compiler can also be achieved for SML. Some candidates for good future compilers are Moscow ML, MLWorks™ and MLTON (see also 2.1). Particularly for comparison to JAVA, the most challenging approach is MLj [BKR99], a compiler that translates SML to the JVM byte-code. SML programs can then run on virtually all platforms. Moreover, they can profit from the ongoing rapid development and improvement of JVM implementations. It would be very interesting to compile *fxp* with MLj and compare it to a JAVA parser. At the moment, however, *fxp* can not be compiled with MLj since this compiler does not support SML functors; an extension of MLj to deal with functors is planned.

We can thus expect to have better SML compilers available in the near future. Having this in mind, *fxp* can compete with parsers written in JAVA concerning execution time. The benefits of functional programming such as higher order functions, polymorphic types and referential transparency thus do not force us to sacrifice reasonable execution speed.

On the other hand, SML programs are more concise than JAVA programs: Without comments, *fxp* consists of 10000 lines of source code (365 KB), whereas *xml4j* requires nearly 22000 lines (888 KB) for implementing a comparable functionality.

3.2 Discussion of the Implementation Language

In 2.1 we gave some arguments for the choice of SML as implementation language. Most of these arguments are reinforced by the experiences we made during the development of *fxp*.

SML's module system supports well the development of large and complex software. Its parametric modules and the polymorphic types give a high reusability to the source code without impairing efficiency. SML has a very comfortable type system with a type inference mechanism that relieves the programmer from annotating the program with type information. Moreover, with SML's user-defined data-types the error-prone construction of data structures with pointers becomes obsolete. Another common source of errors are misspellings or inadequate use of functions, e.g., with the wrong order of arguments. Usually, the type system is strong enough to find most of these errors at compile-time. Indeed, the first public release of *fxp* had only few bugs in it – most of these did not arise from programming errors but were due to ambiguities and obscurities in the XML recommendation.

The SML/NJ system is a comfortable development environment. Its interpreter is an excellent environment for interactive modular testing of programs. Its separate compilation manager allows for fast recompilation after changing a detail in the source code; thus it enables quick test iterations. The SML Basis Library is a comfortable way of accessing system-level functions like I/O or operating system processes in a platform-independent way.

On the other hand, we also missed a few features in SML. First of all, we would wish to see UNICODE support built-in to SML or integrated into the Basis Library. For *fxp*, we had to define our own types for UNICODE characters and strings together with functions for manipulating them. Many of these functions can be implemented more efficiently by a system library. Moreover, *fxp*'s source code is obscured by the use of hexadecimal notation for a UNICODE character, like `0wx61`, which is due to implementing it as a `word`. It would be far more convenient to specify characters literally, e.g., `#"a"`, similar to conventional ASCII characters.

Moreover, virtually all areas of computing are by now subject to internationalization efforts. UNICODE support will therefore become substantial not only in the area of XML and document processing. In order for world-wide acceptance, a programming language will have to provide a standardized means of processing UNICODE data. For instance, JAVA's popularity for network programming is partially based on its predefined UNICODE types and utilities.

Another deficiency of SML applies to functional programming in general: In Section 2.7.3, we encountered a situation where a list of characters, accumulated by a tail-recursive function, has reverse order. In order to reestablish the original order before further processing, the list must be traversed a second time by the `rev` function. Avoiding a non-tail-recursive function thus requires two tail-recursive functions. This is a very common situation in functional programming.

How could the additional reversing traversal of the list be avoided? In this special case, the result of the rev call is immediately converted into the more compact vector representation. Function Data2Vector achieves this by a call to the Basis Library function Vector.fromList. This function has to traverse the list two times: once for determining the size of the vector to be allocated, and once for filling that vector with the elements of the list. A modified version fromRevList could incorporate the reversing of the list without loss of efficiency, by simply changing the direction in which the vector is filled. Since accumulating list parameters are very common in functional programming, such a function would certainly be a sensible extension to the SML Basis Library. Other, more comprehensive variants of generating vectors are surveyed in [Wad86]

The more general case is when the rev call is not followed by a conversion to a vector. For this case we might let us inspire by the logic programming language PROLOG: Here the problem would be solved with the help of *difference lists* (see also, e.g., Section 1.5 of [O'K90]). This programming style exploits the unification of logic programming. The end of a difference list is not the empty list nil but a hole in form of an uninstantiated variable. This hole can later be filled by instantiating the variable with a list, possibly having another hole at the end. Finally, when the list is complete, the hole at the end is instantiated to nil.

In SML, there are no logical variables and no unification. Implementing difference lists, however, does not require full unification. Instead of logic variables instantiated by unification, place-holders that can be assigned a value would be sufficient. SML's references are such place-holders, but they belong to the imperative features of the language. Moreover, implementing a list with references prevents application of built-in functions like Vector.fromList. Therefore, the integration of an appropriate concept into SML would be desirable. The write-once variables in [PE88] and the futures of MULTILISP [Hal85] and OZ [Smo98] provide such extensions to functional languages. These, however, are realized with the help of concurrency, introducing an administrative runtime overhead that is probably larger than that of a non-tail-recursive function. The implications of such an extension on other aspects of SML, like the type system, the strict evaluation or exception handling, lie beyond the scope of this writing.

Summarizing our experiences, we conclude that SML is well-suited for implementing large software. Its sophisticated module system allows for good modularization of the program and writing of highly reusable code. We found that the non-pure features of SML were essential for achieving a reasonable efficiency; their employment can, however, be limited to very few and small parts of the program. Nonetheless, we found that SML needs improvement: Most of all, we demand for UNICODE support and better compilers. Other extensions to the language might ease programming but are not vital.

Part II

Forest Automata for Querying Structured Documents

Introduction

A vital task in document processing is *querying*, i.e., extraction of parts from a document that match a specified pattern. The query consists of two conditions: the form of the requested parts (what to search) and the context in which they must be (where to find it). For instance, if the document is the technical documentation of an aircraft, the query might request a section whose title contains "safety regulations". Since an aircraft is a complex vehicle, one might not be interested in the safety regulations for the whole aircraft, but only in those for the landing gear. Then one would specify as the context that the requested section must occur within a chapter whose title contains "landing gear".

After the advent of modern markup languages such as SGML and XML, documents are *hierarchically structured*. The task of querying thus reduces to the *location* of subtrees in a document tree, which fulfill a *structural condition* and are in a specified *context*. Techniques from tree language and tree automata theory can therefore be applied in order to perform the querying.

Structural conditions can be given as *regular tree languages*, which are a well-studied area of language theory. A common way of specifying regular tree languages is by tree grammars; they are recognized by bottom-up *tree automata*. The theory of regular tree languages has many applications, for instance in pattern recognition and code generation. Most applications, however, consider *ranked* trees in which the number of the children of a node is determined by the symbol at that node. This is different for documents: For instance, the number of sections in a chapter is not fixed. Document trees are therefore *non-ranked* trees: The children of a node are a sequence of arbitrarily many trees, that is, a *forest*. In order to implement querying based on tree automata theory, the concepts of tree grammars and tree automata must be transferred to the non-ranked case.

Moreover, a formalism for specifying contextual conditions must be developed. Because the users of a querying tool are not necessarily computer scientists, the specification language should be intuitive and easily understandable. If possible, it should use the same formalism for specifying both the structural and the contextual condition. Nonetheless, it should be expressive enough to denote regular tree languages for the structural part; for the contextual condition, a comparable expressiveness is desired.

Of course, a querying algorithm must be efficient for being practical. In document processing, this does not only refer to the time efficiency, i.e., the execution speed of the program. Due to the possibly immense size of a document (for instance an encyclopedia), it might not fit into the physical memory of the computer. For such large documents it is desirable to perform the querying without loading the whole document into memory. Instead it should happen "*on the fly*" by a single pass through the document tree. This traversal must be in the same order as the sequential representation of the document, that is in depth-first, left-to-right order. Due to the expressiveness of the query language, we can not expect that this is possible for all queries but at least for a subclass of sufficiently simple queries.

In this work we present a querying algorithm fulfilling the above desiderata. From the concept of tree grammars we derive the formalism of *forest grammars* by allowing regular expressions on right-hand sides of productions. The regular expressions account for the arbitrary number of children allowed for a node. We use forest grammars as a uniform method for specifying both structural and contextual conditions. We then extend forest grammars with *conjunctions* and *negations*. This increases the succinctness but not the expressiveness of the grammars: They describe the *regular forest languages*.

In order to implement forest grammars, we introduce the class of bottom-up *forest automata*. We show that these forest automata recognize exactly the regular forest languages. Then we adapt the traversing order of an XML parser and enhance the forest automata with a pushdown. The resulting class of *pushdown forest automata* has the interesting property that an automaton can be made *deterministic* in spite of its pushdown. Deterministic pushdown automata are significantly more succinct than their bottom-up counterparts.

On the basis of pushdown forest automata, we present an algorithm which implements a query by two consecutive runs of pushdown automata; the first one traverses in left-to-right order and the second one in right-to-left order, or vice versa. We also identify an important subclass of queries for which a single run suffices, namely *right-ignoring grammars*. These grammars are such that only the left part of the context must be verified; the right part can be safely skipped because it is always fulfilled.

Given this querying algorithm, we adapt it to the practical requirements of XML processing: We add support for matching text and handling of XML attributes, as well as some other XML-specific features. The resulting specification formalism are *query grammars*. Since grammars are unintelligible for non-computer scientists, we develop an alternative *pattern* syntax, which can express most queries more intuitively. The simplicity of this pattern language, however, is at the cost of expressiveness: It can not specify all regular queries.

We implemented our querying algorithm in the functional programming language SML, based on the XML parser *fxp* described in Part I. The program is reasonably efficient and proves another time that SML is well-suited for implementing practical applications.

We proceed as follows: In Chapter 4 we introduce the basic concepts: regular expressions and non-ranked trees and forests. Chapter 5 defines the regular forest languages by forest grammars. Chapter 6 introduces forest automata and pushdown forest automata and shows how structural conditions are implemented with these automata. Chapter 7 establishes the concept of context and presents the querying algorithm. It also explains the extensions to the grammar formalism and the one-pass algorithm. The following chapter deals with the particularities of XML documents and introduces the alternative pattern syntax. Finally, Chapter 9 describes the implementation in SML and draws a conclusion.

Einführung

Eine der wichtigsten Aufgaben in der Dokumentenverarbeitung ist das Suchen (*Querying*) in Dokumenten. Dabei werden Teildokumente extrahiert, die ein angegebenes Muster erfüllen. Die Anfrage stellt zweierlei Bedingungen, zum einen an die Form der Teildokumente (*was* wird gesucht?), und zum anderen an den Kontext (*wo* wird gesucht?), in dem sie stehen. Ist das Dokument z.B. die technische Dokumentation eines Flugzeugs, könnte die Anfrage nach Abschnitten suchen, deren Titel das Wort "Sicherheitsbestimmungen" enthält. Nun ist ein Flugzeug ein sehr komplexer Apparat, und man könnte nicht an den Bestimmungen für das gesamte Flugzeug, sondern nur an denen für das Fahrwerk interessiert sein. Sinnvollerweise würde man dann als Kontext angeben, dass der Abschnitt innerhalb eines Kapitels vorkommt, dessen Titel das Wort "Fahrwerk" enthält.

Aus der Sicht moderner Dokumenten-Auszeichnungssprachen wie SGML und XML sind Dokumente hierarchisch strukturiert. Den Vorgang des Querying kann man deshalb auch als die Lokalisierung von Teilbäumen in einem Dokument-Baum auffassen, die eine *strukturelle* und eine *kontextuelle* Bedingung erfüllen. Zur Lösung dieser Aufgabe können Techniken aus der Theorie der Baumsprachen und Baumautomaten herangezogen werden.

Eine strukturelle Bedingung kann als *reguläre Baumsprache* angegeben werden; eine häufig verwendete Spezifikationsmethode für reguläre Baumsprachen sind z.B. Baumgrammatiken. Die Klasse der regulären Baumsprachen wird von Baumautomaten akzeptiert. Diese Sprachen und Automaten sind ein gut erforschtes Gebiet in der Theorie der formalen Sprachen: Sie wurden z.B. im Rahmen der Code-Erzeugung und der Mustererkennung ausführlich untersucht. In den meisten Anwendungen werden allerdings Bäume *fester Stelligkeit* untersucht, in denen die Anzahl der Söhne eines Knotens durch das Symbol an diesem Knoten festgelegt ist. In Dokument-Bäumen ist das nicht so: Beispielsweise variiert die Anzahl der Abschnitte von Kapitel zu Kapitel. Dokument-Bäume haben daher beliebige Stelligkeit: Die Söhne eines Knotens sind eine Folge von *beliebig* vielen Bäumen, also ein *Wald*. Deshalb müssen die Konzepte der Baumgrammatiken und der Baumautomaten auf den Fall beliebiger Stelligkeit übertragen werden.

Des weiteren ist ein Formalismus für die Spezifikation der kontextuellen Bedingung nötig. Da die Benutzer einer Querying-Software im allgemeinen keine Informatiker sind, sollte der Formalismus intuitiv und leicht verständlich sein. Wenn möglich, sollte sowohl für die strukturelle als auch für die kontextuelle Bedingung ein und der selbe Spezifikations-Formalismus verwendet werden. Trotz allem sollte dieser Formalismus ausdrucksstark genug sein, um reguläre Baumsprachen für die strukturelle Bedingung anzugeben; für die kontextuelle Bedingung ist eine vergleichbare Ausdrucksstärke gefordert.

Ein Querying-Algorithmus muss natürlich effizient sein, um in der Praxis Anwendung zu finden. Damit ist nicht nur die Geschwindigkeit des Programms gemeint: Da ein Dokument (z.B. ein Lexikon) immens groß sein kann, ist es möglich, dass es nicht in den physikalischen Speicher des Rechners passt. Für solch große Dokumente ist es erforderlich, die Suche auszuführen, ohne das gesamte Dokument vorher in den Speicher zu laden. Statt dessen muss der Algorithmus *"on the fly"* ablaufen, d.h. in einem einzigen Durchlauf durch das Dokument. Die Reihenfolge dieses Durchlaufs muss mit der sequentiellen Repräsentation des Dokuments übereinstimmen, die einem Links-Rechts-Tiefendurchlauf entspricht. Wegen der Ausdrucksstärke der Anfragesprache ist dies natürlich nicht in allen Fällen, zumindest aber für eine Klasse von hinreichend einfachen Anfragen möglich.

Diese Arbeit stellt einen Querying-Algorithmus vor, der die aufgeführten Anforderungen erfüllt. Aus dem Konzept der Baumgrammatiken entwickeln wir den Formalismus der *Waldgrammatiken*, indem wir reguläre Ausdrücke auf den rechten Seiten von Produktionen zulassen. Diese regulären Ausdrücke sind notwendig, um über die beliebig vielen Söhne eines Knotens zu argumentieren. Wir verwenden Waldgrammatiken als einheitliche Spezifikationsmethode sowohl für strukturelle als auch für kontextuelle Bedingungen. Außerdem erweitern wir die Grammatiken um *Konjunktion* und *Negation*. Diese Erweiterung erhöht zwar die Präzision der Grammatiken, d.h. die Kürze der Beschreibung, nicht aber ihre Ausdrucksstärke: Sie beschreiben *reguläre Waldsprachen*.

Für die Implementierung von Waldgrammatiken führen wir die Klasse der Waldautomaten ein. Wir zeigen, dass diese Automaten genau die regulären Waldsprachen akzeptieren. Durch das Hinzufügen eines Kellers und das Angleichen der Durchlaufreihenfolge an die eines XML-Parsers entwickeln wir die Automaten weiter und erhalten die Klasse der *Keller-Waldautomaten*. Eine interessante Eigenschaft dieser Automaten ist, dass sie trotz ihres Kellers deterministisch gemacht werden können. Des weiteren sind deterministische Keller-Waldautomaten bedeutend – bis zu einem exponentiellen Faktor – kompakter als Waldautomaten ohne Keller.

Auf der Grundlage von Keller-Waldautomaten entwickeln wir einen Querying-Algorithmus, der eine Anfrage mithilfe von zwei aufeinander folgenden Läufen je eines Automaten implementiert. Der erste dieser Automaten läuft von rechts nach links, der zweite von links nach rechts durch das Dokument. Für eine wichtige Teilklasse von Anfragen, die *rechts-ignorierenden* Grammatiken, genügt sogar ein einzelner Lauf. Für diese Grammatiken muss nur der linke Teil des Kontexts überprüft werden. Der rechte Teil kann ignoriert werden, denn er trifft immer zu.

Um diesen Querying-Algorithmus auf XML-Dokumente anzuwenden, passen wir ihn den praktischen Anforderungen von XML an. Insbesondere integrieren wir Unterstützung für Text und Processing Instructions sowie einige andere XML-spezifische Gesichtspunkte. Der resultierende Spezifikations-Formalismus sind die *Query-Grammatiken*. Da Grammatiken für Nicht-Informatiker unverständlich sind, entwickeln wir eine alternative *Muster*-Syntax, die die meisten Anfragen intuitiver ausdrücken kann als Grammatiken. Die Einfachheit dieser Mustersprache hat allerdings ihren Preis: Nicht alle regulären Anfragen können formuliert werden.

Schließlich beschreiben wir die Implementierung des Algorithmus in der funktionalen Programmiersprache SML und auf der Basis des XML-Parsers *fxp*, der in Teil I beschrieben wurde. Das Programm ist durchaus effizient und liefert einen weiteren Beweis für die gute Eignung von SML für die Implementierung praktischer Anwendungen.

Chapter 4

Preliminaries

In this chapter we define regular expressions and regular word languages, needed for defining forest grammars and text patterns later on. Then we introduce the notion of non-ranked trees and forests and relate them to conventional ranked trees.

4.1 Regular Expressions and Word Languages

An *alphabet* Σ is a finite set of symbols. A *regular expression* defines a set of words from Σ^*. The set $\mathcal{R}_\Sigma$ of regular expressions r over Σ is given as follows:

$$r \ ::= \ \emptyset \ \mid \ \epsilon \ \mid \ a \in \Sigma \ \mid \ (r?) \ \mid \ (r^*) \ \mid \ (r_1 r_2) \ \mid \ (r_1 \,|\, r_2)$$

where r_1 and r_2 are regular expressions. For brevity we often omit the parentheses, with the convention that * and ? have precedence over concatenation which itself binds stronger than $|$. As an abbreviation, we also use r^+ instead of rr^*. The number of occurrences of symbols from Σ in r is $|r|_\Sigma$; the *size* $|r|$ of a regular expression is $|r|_\Sigma$ plus the number of occurrences of $\emptyset$, ϵ, *, ?, $|$ and $\cdot$ (concatenation) in r. The *language* $[\![r]\!]_\mathcal{R} \subseteq \Sigma^*$ of a regular expression r is defined inductively on the structure of r:

$$\begin{array}{llll}
[\![\emptyset]\!]_\mathcal{R} & = \emptyset & [\![r^*]\!]_\mathcal{R} & = \{w_1 \dots w_n \mid w_i \in [\![r]\!]_\mathcal{R},\ n \geqslant 0\} \\
[\![\epsilon]\!]_\mathcal{R} & = \{\epsilon\} & [\![r_1 r_2]\!]_\mathcal{R} & = \{w_1 w_2 \mid w_1 \in [\![r_1]\!]_\mathcal{R},\ w_2 \in [\![r_2]\!]_\mathcal{R}\} \\
[\![a]\!]_\mathcal{R} & = \{a\},\ a \in \Sigma & [\![r_1 \,|\, r_2]\!]_\mathcal{R} & = [\![r_1]\!]_\mathcal{R} \cup [\![r_2]\!]_\mathcal{R} \\
[\![r?]\!]_\mathcal{R} & = \{\epsilon\} \cup [\![r]\!]_\mathcal{R} & &
\end{array}$$

Note that regular expressions of the form $r?$ are equivalent to $(r \,|\, \epsilon)$. It is easy to see that each regular expression r can be rewritten with the help of ? to an equivalent regular expression r', which is at most as large as r, and which is either ϵ or does not contain ϵ at all. With a similar transformation we can ensure that a regular expression r is either $\emptyset$ or $\emptyset$ does not occur in r at all. W.l.o.g., we will therefore assume for the remainder that all regular expressions obey these two restrictions.

A set $L \subseteq \Sigma^*$ is called *regular* if it is the language of a regular expression. It is a long-known fact [Kle56] that the regular languages are exactly those accepted by finite automata. A *non-deterministic finite automaton* (NFA) $A = (Q, q_0, F, \delta)$ consists of a set of states Q, an *initial* state $q_0 \in Q$, a set of *final* states $F \subseteq Q$

and a *transition relation* $\delta \subseteq Q \times \Sigma \times Q$. A word $a_1 \dots a_n \in \Sigma^*$ is accepted by A if there are $q_1, \dots, q_{n+1} \in Q$ such that $q_1 = q_0$, $(q_i, a_i, q_{i+1}) \in \delta$ for $1 \leqslant i \leqslant n$ and $q_{n+1} \in F$. The language $\mathcal{L}_A$ of an NFA A is the set of words it accepts.

Theorem 4.1: A language is regular iff it is accepted by an NFA, more precisely:

(1) For each regular expression r, there is an NFA A with $\mathcal{L}_A = [\![r]\!]_{\mathcal{R}}$.

(2) For each NFA A, there is a regular expression r with $[\![r]\!]_{\mathcal{R}} = \mathcal{L}_A$.

Proof: A construction for (2) is given in Section 3.3 of [Woo87] together with a correctness proof. (1) is a direct consequence of the next proposition (4.1). □

4.1.1 The Berry-Sethi Construction

A well-studied method for obtaining an NFA for a given regular expression is the *Berry-Sethi construction* [BS86]. It is based on an algorithm for direct construction of a *deterministic* automaton introduced by [NY60, Glu61]. Several variants of the algorithm are discussed and related to each other in, e.g., [Wat93, CZ96].

For a regular expression r, the Berry-Sethi construction yields an NFA $Berry(r) = (Q, q_0, F, \delta)$. The construction is as follows: If $r = \epsilon$ then choose some arbitrary q_0 and return $(\{q_0\}, q_0, \{q_0\}, \emptyset)$; for $r = \emptyset$, return $(\{q_0\}, q_0, \emptyset, \emptyset)$. Otherwise r does not contain ϵ or $\emptyset$. Then perform the following steps:

1. Choose a suitable set P of *positions* p with $|P| = |r|_{\Sigma}$.

2. Assign a unique position to each occurrence of a symbol from Σ in r. Construct a new regular expression $\overline{r}$ over P by replacing each symbol a in r with its assigned position p; a is then the *underlying symbol* $\chi(p)$ of p. Note that all subexpressions of $\overline{r}$ are pairwise different.

3. Compute for each subexpression r' of $\overline{r}$ a flag $Empty(r')$, indicating whether $\epsilon \in [\![r']\!]_{\mathcal{R}}$, as follows:

$$\begin{aligned}
Empty(p) &= \mathit{false} \\
Empty(r_1?) &= \mathit{true} \\
Empty(r_1{}^*) &= \mathit{true} \\
Empty(r_1 r_2) &= Empty(r_1) \wedge Empty(r_2) \\
Empty(r_1 \,|\, r_2) &= Empty(r_1) \vee Empty(r_2)
\end{aligned}$$

4. Compute for each subexpression r' of $\overline{r}$ the set $First(r')$ of positions that can start a word in $[\![r']\!]_{\mathcal{R}}$:

$$\begin{aligned}
First(p) &= \{p\} \\
First(r_1?) &= First(r_1) \\
First(r_1{}^*) &= First(r_1) \\
First(r_1 \,|\, r_2) &= First(r_1) \cup First(r_2) \\
First(r_1 r_2) &= First(r_1) \cup \begin{cases} First(r_2) & \text{if } Empty(r_1) \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

5. Compute for each subexpression r' of $\overline{r}$ the set $Follow(r')$ of positions that can follow a word $w' \in [\![r']\!]_{\mathcal{R}}$ within a word in $[\![\overline{r}]\!]_{\mathcal{R}}$. Moreover, if $Follow(r')$ contains the auxiliary symbol $\# \notin P$ then w' can finish a word in $[\![\overline{r}]\!]_{\mathcal{R}}$. In the original construction by [NY60], the positions that can end a word are computed as an explicit set *Last*. Incorporating their computation into *Follow* is due to [BS86]. *Follow* is computed in a top-down traversal of the expression:

$$\begin{array}{rl} r' = \overline{r}: & Follow(r') = \{\#\} \\ r' = r_1?: & Follow(r_1) = Follow(r') \\ r' = r_1{}^*: & Follow(r_1) = Follow(r') \cup First(r_1) \\ r' = r_1 \,|\, r_2: & Follow(r_1) = Follow(r_2) = Follow(r') \\ r' = r_1 r_2: & Follow(r_2) = Follow(r') \\ & Follow(r_1) = First(r_2) \cup \begin{cases} Follow(r') & \text{if } Empty(r_2) \\ \emptyset & \text{otherwise} \end{cases} \end{array}$$

6. Choose a $q_0 \notin P$, let $Q = \{q_0\} \cup P$ and $F = \{p \in P \mid \# \in Follow(p)\} \cup F_0$, where $F_0 = \{q_0\}$ if $Empty(\overline{r})$ is true and $F_0 = \emptyset$ otherwise. The transition relation δ is given by

$$\begin{array}{l} \delta = \{(q_0, \chi(p), p) \mid p \in First(\overline{r})\} \\ \quad \cup \{(p, \chi(p_1), p_1) \mid p, p_1 \in P, p_1 \in Follow(p)\} \end{array}$$

Proposition 4.1: For a regular expression r, let $A = (Q, q_0, F, \delta) = Berry(r)$. Then $\mathcal{L}_A = [\![r]\!]_{\mathcal{R}}$. The proof is given in [BS86]. □

The Berry-Sethi construction produces an NFA traversing the input word from left to right. Analogously, the *reverse Berry-Sethi construction* yields an NFA $Berry^{\leftarrow}(r)$ which consumes it input from right to left. For this algorithm, we must modify the Berry-Sethi construction for the case of concatenation: The roles of the two subexpressions in the computation of *First* and *Follow* are switched:

$$First(r_1 r_2) = First(r_2) \cup \begin{cases} First(r_1) & \text{if } Empty(r_2) \\ \emptyset & \text{otherwise} \end{cases}$$

and for $r' = r_1 r_2$:

$$\begin{array}{l} Follow(r_1) = Follow(r') \\ Follow(r_2) = First(r_1) \cup \begin{cases} Follow(r') & \text{if } Empty(r_1) \\ \emptyset & \text{otherwise} \end{cases} \end{array}$$

Proposition 4.2: For a regular expression r, let $A = Berry^{\leftarrow}(r)$. Then $w \in \mathcal{L}_A$ iff $w^{\leftarrow} \in [\![r]\!]_{\mathcal{R}}$ for all $w \in \Sigma^*$, where $(a_1 \dots a_n)^{\leftarrow} = a_n \dots a_1$. □

Example 4.1: Let $r = (a^*c \,|\, aa?b)$. It is easy to see that $[\![r]\!]_{\mathcal{R}} = \{ab, aab, c, ac, aac, aaac, \dots\}$. Let us perform the Berry-Sethi construction for r:

1. Since r has five occurrences of symbols, choose $P = \{1, \dots, 5\}$ for the positions.

2. Construct $\overline{r} = (1^*2 \,|\, 34?5)$. Thus $\chi(1) = \chi(3) = \chi(4) = a$, $\chi(2) = c$ and $\chi(5) = b$.

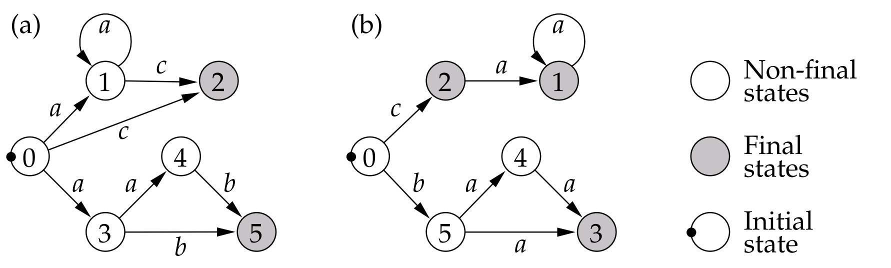


Figure 4.1: The Berry-Sethi construction (a) and the reverse Berry-Sethi construction (b) for $r = (a^*c \mid aa?b)$.

3. $Empty(1) = \ldots = Empty(5) = false$;
   $Empty(1^*) = Empty(4?) = true$;
   $Empty(1^*2) = Empty(34?) = Empty(34?5) = Empty(\overline{r}) = false$.

4. $First(p) = \{p\}, 1 \leqslant p \leqslant 5$;
   $First(1^*) = \{1\}; First(4?) = \{4\}$;
   $First(1^*2) = \{1,2\}; First(34?) = First(34?5) = \{3\}$;
   $First(\overline{r}) = \{1,2,3\}$.

5. $Follow(\overline{r}) = Follow(1^*2) = Follow(34?5) = \{\#\}$;
   $Follow(2) = \{\#\}; Follow(1^*) = \{2\}; Follow(1) = \{1,2\}$;
   $Follow(5) = \{\#\}; Follow(34?) = \{5\}$;
   $Follow(4) = Follow(4?) = \{5\}; Follow(3) = \{4,5\}$

6. With $q_0 = 0$, we obtain $Q = \{0,\ldots,5\}$, $F = \{2,5\}$, and $\delta = \{(0,a,1), (0,c,2), (0,a,3), (1,a,1), (1,c,2), (3,a,4), (3,b,5), (4,b,5)\}$. *Berry*$(r)$ is illustrated in Figure 4.1, case (a).

The reverse Berry-Sethi construction for r yields the NFA in Figure 4.1(b). □

Both the Berry-Sethi construction and the reverse Berry-Sethi construction have an interesting and important property: All transitions leading to a state p are labeled with the same symbol, namely $\chi(p)$:

Proposition 4.3: For a regular expression r, let (Q, q_0, F, δ) be the NFA produced by *Berry*(r) or $Berry^{\leftarrow}(r)$, and let $(q_1, a_1, q), (q_2, a_2, q) \in \delta$. Then $a_1 = a_2$.

4.2 Trees and Forests

This section introduces the basic notions of trees and forests and relates them to conventional, ranked trees.

Let Σ be an alphabet. The sets $\mathcal{T}_\Sigma$ of *trees* t and $\mathcal{F}_\Sigma$ of *forests* f over Σ are given as follows:

$$t \ ::= \ a\langle f\rangle,\ a \in \Sigma \qquad f \ ::= \ t_1 \ldots t_n,\ n \geqslant 0$$

For brevity, we often omit the alphabet Σ in names, writing simply $\mathcal{T}$ and $\mathcal{F}$ instead of $\mathcal{T}_\Sigma$ and $\mathcal{F}_\Sigma$. Throughout the remainder of this writing, we use the following naming conventions: $a, a_1, \ldots$ denote elements from Σ; we use $t, t_1, t', \ldots$ for trees and $f, f_1, f', \ldots$ for forests.

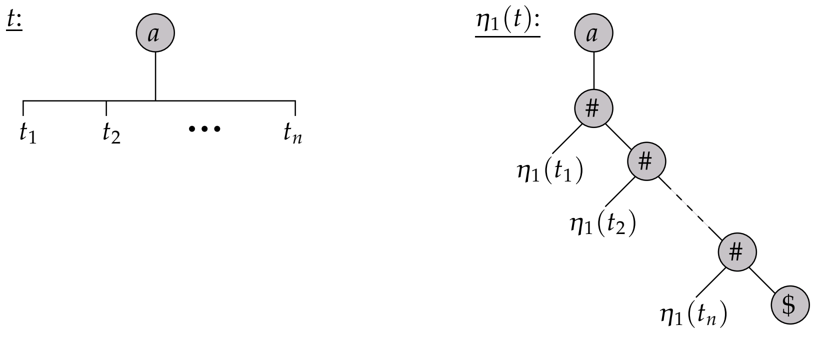


Figure 4.2: Unique representation of trees as ranked trees.

Note that, in contrast to graph theory, a forest is an ordered sequence of trees rather than an unordered set. The empty forest, i.e., the empty sequence of trees, is usually written as ϵ. The occurrences of symbols from Σ in a tree t are also called *nodes*. For a tree $t = a\langle f\rangle$, this occurrence of a is the *root node* of t and $sym(t) = a$ is the *symbol* or *label* of that node.

For $t = a\langle t_1 \dots t_n\rangle$, the trees t_i are called the *successors* or *children* of t, whereas t is the *parent* of t_i. A *leaf* is a tree $a\langle\rangle$ without children and is often abbreviated to a. A *descendant* t' of a tree t is either a child of t or a child of a descendant of t; t is then an *ancestor* of t'. For a forest $t_1 \dots t_n$ and $i \neq j$, the tree t_i is called a *sibling* of t_j; if $i < j$ then it is a *left sibling*, otherwise a *right sibling*. When no ambiguities can arise, we often identify trees with their root nodes, speaking of children, parents, ancestors, etc. of nodes. Similarly, we sometimes denominate nodes by their symbols.

4.2.1 Relation to Ranked Trees

The elements of Σ are not ranked: A node in the tree can have arbitrarily many children. This is different from conventional tree theory, where each symbol a is assigned a fixed *rank* $\rho(a)$; each tree labeled a must then have exactly that many children. For clarity, we will always call this kind of trees *ranked trees*.

Allowing arbitrarily many children for a node does not enhance expressiveness of trees: Each tree or forest can be uniquely represented as a ranked tree. One possible way of doing so is to extend Σ by two additional symbols:

$$\Sigma^\# = \Sigma \cup \{\#, \$\} \quad \text{with} \quad \rho(a) = \begin{cases} 1, & \text{for } a \in \Sigma \\ 2, & \text{for } a = \# \\ 0, & \text{for } a = \$ \end{cases}$$

We can now map trees and forests to ranked trees with a function $\eta_1 : \mathcal{T}_\Sigma \cup \mathcal{F}_\Sigma \to \mathcal{T}_{\Sigma^\#}$ as illustrated by Figure 4.2:

$$\begin{aligned} \eta_1(\epsilon) &= \$ \\ \eta_1(tf) &= \#\langle \eta_1(t)\eta_1(f)\rangle \\ \eta_1(a\langle f\rangle) &= a\langle \eta_1(f)\rangle \end{aligned}$$

η_1 expands the sequence of children of a tree t into a spine of #-nodes, with \$ at the tip of the spine; the children of t sit on the ribs of the spine. Note that this is exactly how lists are represented in declarative programming languages. η_1

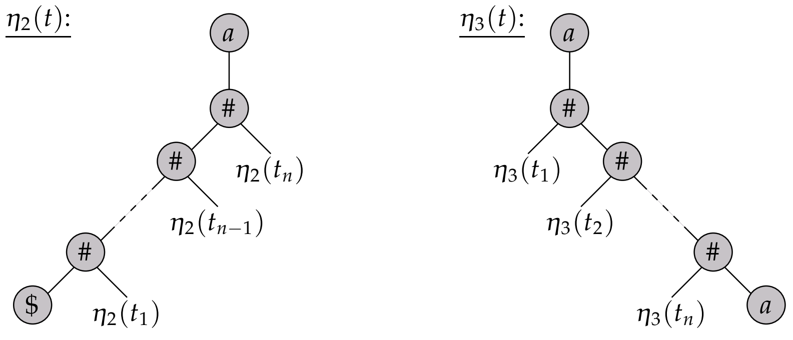


Figure 4.3: Alternative ways of mapping trees to ranked trees.

is an injective mapping: Each tree or forest is uniquely represented and can be reconstructed from its ranked-tree image.

Note that η_1 is only one possible mapping to ranked trees; many others are possible and might be as sensible as η_1. Figure 4.3 shows two of them: η_2 is very similar to η_1; the main difference is that a bottom-up traversal encounters a node's left-most child first in $\eta_2(t)$, whereas in $\eta_1(t)$ the right-most child is seen first. The representation by η_3 has the advantage that both a bottom-up and a top-down traversal see the symbol a before visiting the node's children. However, the resulting trees are not strictly ranked: Each symbol from Σ occurs with ranks 0 and 1.

Documents with their arbitrary number of children could thus be modeled by ranked trees; yet the non-ranked approach corresponds more closely to the document processing view. Moreover, the ranked representation prescribes the order in which a bottom-up automaton visits the children of a node. Conceptually, such an automaton proceeds to a node from all of its children simultaneously in a single step. We will see later that it is sensible to abandon this illusion: In an implementation, the children of a node must be processed in some order, be it left-to-right or right-to-left. The representation of non-ranked trees allows for both of these orders.

4.2.2 Structural Induction

The most important proof method for statements about trees and forests is *structural induction*. In order to formally introduce this concept, we need the notion of *subtrees* and *subforests*:

- ✧ each tree is a subtree of itself;
- ✧ if $a\langle t_1 \ldots t_n\rangle$ is a subtree of t then so is t_i for $1 \leqslant i \leqslant n$;
- ✧ for $f = \ldots t_i \ldots$, the subtrees of t_i are also subtrees of f.
- ✧ if $f = t_1 \ldots t_n$, then $t_1 \ldots t_i$ is a *left subforest* and $t_i \ldots t_n$ is a *right subforest* of f for $1 \leqslant i \leqslant n$;
- ✧ if $t_1 = a\langle f_1\rangle$ is a subtree of t or f and f_2 is a left or right subforest of f_1 then it is also a left or right subforest of t or f.

Corollary 4.1: (Principle of Structural Induction) Suppose we want to prove a property $\mathcal{A}_\mathcal{F}\, f$ for all forests $f \in \mathcal{F}$. Then it suffices to find a suitable property $\mathcal{A}_\mathcal{T}$ and show the following:

(E) $\mathcal{A}_\mathcal{F}\, \epsilon$ holds;

(F) If $\mathcal{A}_\mathcal{T}\, t_i$ holds for $1 \leqslant i \leqslant n$, then so does $\mathcal{A}_\mathcal{F}\, t_1 \ldots t_n$;

(T) $\mathcal{A}_\mathcal{F}\, f$ implies $\mathcal{A}_\mathcal{T}\, a\langle f\rangle$ for all $a \in \Sigma$.

$\mathcal{A}_\mathcal{T}$ and $\mathcal{A}_\mathcal{F}$ are called the *induction invariants*. Step (F) concludes from all trees of a forest simultaneously. Sometimes it is desirable to proceed through these trees in a certain order. This is reflected by replacing step (F) with either of the following:

(R) If $\mathcal{A}_\mathcal{T}\, t$ and $\mathcal{A}_\mathcal{F}\, f$ are fulfilled, then so is $\mathcal{A}_\mathcal{F}\, tf$;

(L) If $\mathcal{A}_\mathcal{T}\, t$ and $\mathcal{A}_\mathcal{F}\, f$ are fulfilled, then so is $\mathcal{A}_\mathcal{F}\, ft$.

(R) proceeds through the trees of a forest from right to left: It concludes from a tree t and a right subforest f to the larger subforest tf; analogously, (L) proceeds from left to right. Note that (E) is just a special case of (F) with $n = 0$. Therefore, if we show (F) with $n \leqslant 0$ then we do not need to verify (E).

4.2.3 Path Induction

The principle of structural induction gives us a means of arguing about a whole tree or forest by reasoning about its subtrees and subforests. However, if we want to prove a property for all subtrees of a forest f, we need a different technique: *path induction*. In order to introduce this proof method, we first need a way of uniquely specifying a subtree located somewhere in a forest. In particular, if the same subtree occurs multiple times we must be able to distinguish the different occurrences because they might have different properties. A straightforward way of identifying a subtree is by the path through all of its ancestors: Let f be a forest. Then $\Pi(f) \subseteq \mathbb{N}^*$ is the set of all paths π in f and is defined as follows:

$$\begin{aligned} \Pi(\epsilon) &= \emptyset \\ \Pi(t_1 \ldots t_n) &= \{i\pi \mid \pi \in \Pi(t_i), 1 \leqslant i \leqslant n\} \\ \Pi(a\langle f\rangle) &= \{\epsilon\} \cup \Pi(f) \end{aligned}$$

A path in f identifies one of f's subtrees: For $\pi \in \Pi(f)$, $f[\pi]$ is called the subtree *located at* π and is defined as follows:

$$\begin{aligned} t[\epsilon] &= t \text{ for all } t \in \mathcal{T}_\Sigma \\ a\langle f\rangle[\pi] &= f[\pi] \text{ for } \pi \neq \epsilon \\ t_1 \ldots t_n[i\pi] &= t_i[\pi] \end{aligned}$$

For a path π, we define $last_f(\pi)$ as the number of the right-most child of the node located at π. Precisely, $last_f(\pi) = max\{n \mid \pi n \in \Pi(f)\}$. Note that $last_f(\pi) = 0$ if π identifies a leaf.

Corollary 4.2: (Principle of Path Induction) For a given forest f_0, suppose we want to prove properties $\mathcal{A}_\mathcal{F}\, \pi$ for all paths $\pi \in \Pi(f_0)$ and and $\mathcal{A}_\mathcal{T}\, \pi$ for all paths $\pi \in \Pi(f_0) \setminus \{\epsilon\}$. Then it suffices to show the following:

(s) $\mathcal{A}_\mathcal{F}\ \epsilon$ holds;

(f) $\mathcal{A}_\mathcal{T}\ \pi$ implies $\mathcal{A}_\mathcal{F}\ \pi$, for $\pi \neq \epsilon$.

(t) If $\mathcal{A}_\mathcal{F}\ \pi$ holds then so do $\mathcal{A}_\mathcal{T}\ \pi i$ for $1 \leqslant i \leqslant \mathit{last}_{f_0}(\pi)$.

If we are only interested in $\mathcal{A}_\mathcal{T}\ \pi$ for all $\pi \neq \epsilon$, then it suffices to show:

(o) $\mathcal{A}_\mathcal{T}\ i$ holds for $1 \leqslant i \leqslant \mathit{last}_{f_0}(\epsilon)$;

(d) $\mathcal{A}_\mathcal{T}\ \pi$ implies $\mathcal{A}_\mathcal{T}\ \pi i$ for $1 \leqslant i \leqslant \mathit{last}_{f_0}(\pi)$.

Having defined the preliminary concepts, we can now proceed to regular sets of forests in the next chapter.

Chapter 5

Regular Forest Languages

Regular languages of ranked trees are a well-studied area of language theory. Methods for specifying regular ranked-tree languages include projections of local tree sets, algebraic approaches and tree grammars. For our purposes, the grammar approach is most convenient, though we have to modify it in order to deal with non-ranked alphabets.

5.1 Forest Grammars

A *forest grammar* over Σ is a tuple $G = (X, r_0, R)$ where X is a set of *variables*, $r_0 \in \mathcal{R}_X$ is the *start expression* and R is a finite set of *rules*, also called *productions*, of the form $x \rightarrow a\langle r\rangle$ with $x \in X$, $a \in \Sigma$ and r a regular expression over X. For brevity, if the context is clear, we often write simply $x \rightarrow a\langle r\rangle$ instead of $x \rightarrow a\langle r\rangle \in R$.

In order to define the meaning of a forest grammar, we might use a generative approach: A grammar produces a forest by starting with the start expression and consecutively replacing a variable with the right-hand side of one of its productions, until no more variables occur. In this very popular approach taken by, e.g., [CDG+99] for ranked trees, the trees produced by a grammar are constructed top-down, i.e., starting at the root and proceeding to the leafs. A different approach is to construct them bottom-up, i.e., make up new trees from already constructed ones, according to the productions of the grammar. We follow this approach.

The *meaning* $[\![G]\!] : X \rightarrow 2^{\mathcal{T}_\Sigma}$ of a forest grammar $G = (X, r_0, R)$ assigns sets of trees to the variables in X and is defined inductively on the structure of trees:

$$t = a\langle t_1 \dots t_n\rangle \in [\![G]\!]\, x \text{ iff } \begin{array}{l}\text{there is an } r \in \mathcal{R}_X \text{ and a word } x_1 \dots x_n \in [\![r]\!]_{\mathcal{R}} \\ \text{with } x \rightarrow a\langle r\rangle \text{ and } t_i \in [\![G]\!]\, x_i \text{ for } 1 \leqslant i \leqslant n\end{array}$$

We can easily extend $[\![G]\!]$ to map regular expressions to forests:

$$[\![G]\!]\, r = \{t_1 \dots t_n \mid \text{there is a word } x_1 \dots x_n \in [\![r]\!]_{\mathcal{R}} \text{with } t_i \in [\![G]\!]\, x_i\}$$

The *language* of a forest grammar $G = (X, r_0, R)$ is then $\mathcal{L}_G = [\![G]\!]\, r_0$. A set of forests is *regular* if it is the language of some forest grammar.

Example 5.1: For $\Sigma = \{a, b\}$, let G_1 be the forest grammar $(\{x\}, x^+, R)$ with $R = \{x \rightarrow a\langle x^+\rangle, x \rightarrow b\langle \epsilon\rangle\}$. The language of G_1 is the set of all non-empty forests where all leaves are labeled b and all other nodes have symbol a. □

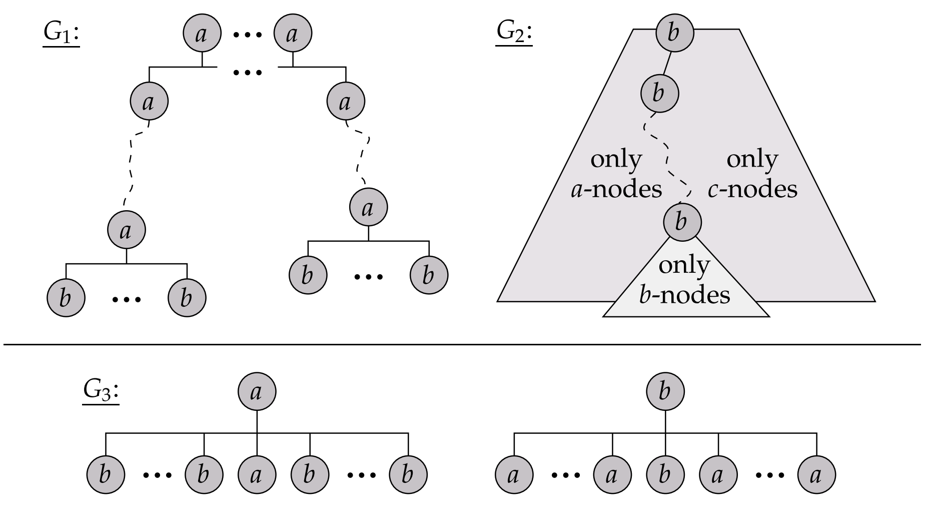


Figure 5.1: Forests in the languages $\mathcal{L}_{G_1}$ – $\mathcal{L}_{G_3}$ from Examples 5.1 – 5.3.

Example 5.2: For $\Sigma = \{a, b, c\}$, let G_2 be the tree grammar $(\{x_a, x_b, x_c, x_1\}, r_0, R)$ with $r_0 = x_a{}^*(x_b \mid x_1)x_c{}^*$ and the following rules:

$$\begin{array}{ll} x_a \longrightarrow a\langle x_a{}^* \rangle & x_c \longrightarrow c\langle x_c{}^* \rangle \\ x_b \longrightarrow b\langle x_b{}^* \rangle & x_1 \longrightarrow b\langle x_a{}^*(x_b \mid x_1)x_c{}^* \rangle \end{array}$$

The language of G_2 is the set of all forests such that

- ✧ there is exactly one path from the root to some node on which all nodes are labeled b;
- ✧ all descendants of that node are labeled b;
- ✧ all nodes to the left of this path are labeled a;
- ✧ all nodes to the right of this path are labeled c.

This is illustrated by Figure 5.1. □

Example 5.3: For $\Sigma = \{a, b\}$, consider the grammar $G_3 = (X, x_1, R)$ with $X = \{x_1, x_a, x_b\}$ and the following rules:

$$\begin{array}{ll} x_1 \longrightarrow a\langle x_b{}^* x_a x_b{}^* \rangle & x_a \longrightarrow a\langle\rangle \\ x_1 \longrightarrow b\langle x_a{}^* x_b x_a{}^* \rangle & x_b \longrightarrow b\langle\rangle \end{array}$$

$\mathcal{L}_{G_3}$ is the set of all trees of depth 2, whose root symbol occurs exactly once at the second level, i.e., all forests of the form $a\langle b \ldots bab \ldots b \rangle$ or $b\langle a \ldots aba \ldots a \rangle$. □

The next lemma basically states that the meaning of a grammar G is a fix-point of $[\![G]\!]$. It is very helpful in structural induction proofs.

Lemma 5.1: Let G be a forest grammar. Then $a\langle f \rangle \in [\![G]\!]\, x$ iff $x \longrightarrow a\langle r \rangle$ for some r, and $f \in [\![G]\!]\, r$. The proof is by definition of $[\![G]\!]$. □

Forest grammars are a very natural and intuitive way of specifying regular forest languages. The next section relates our notion of grammars to the grammars of conventional, ranked-tree theory in order to obtain some closure results.

5.2 Closure Properties of Regular Forest Languages

In Section 4.2 we related our notion of non-ranked trees and forests to conventional, ranked trees and showed that each tree or forest can be uniquely represented as a ranked tree. A regular language of ranked trees can be specified by a forest grammar with the restriction that right-hand sides of productions have the form $a\langle x_1 \ldots x_n\rangle$ with $n = \rho(a)$, and the start expression is a variable x_0. This form of a grammar, which we call *ranked-tree grammar*, is exactly the normal form of regular tree grammars in [GS97]. There it is also shown that the class of regular ranked-tree languages is closed under union, intersection and complement. This result can be transferred to regular forest languages by establishing a one-to-one relation between forest languages and ranked-tree languages.

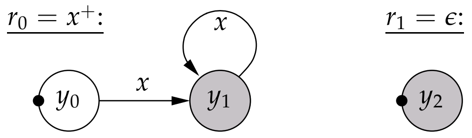
Proposition 5.1: For a regular forest language L, its image $\eta_1(L)$ is a regular ranked-tree language. In other words: Let $G = (X, r_0, R)$ be a forest grammar over Σ. Then a ranked-tree grammar $G^\#$ over $\Sigma^\#$ can be constructed with $\mathcal{L}_{G^\#} = \eta_1(\mathcal{L}_G)$, where η_1 is as in Section 4.2.1.

The construction is as follows: Let $G = (X, r_0, R)$ and $\{r_1, \ldots, r_l\}$ be the set of regular expressions different from r_0 occurring on right-hand sides of rules in R. For each $j = 0, \ldots, l$, let $(Y_j, y_{0,j}, F_j, \delta_j) = \mathit{Berry}(r_j)$. By a renaming of states, we can easily ensure that $Y_i \cap Y_j = \emptyset$ for $i \neq j$. Then $G^\# = (X \cup Y_0 \cup \ldots \cup Y_l, y_{0,0}, R^\#)$ with

$$\begin{aligned} R^\# = &\{x \to a\langle y_{0,j}\rangle \mid x \to a\langle r_j\rangle \in R\}\\ &\cup \{y \to \$ \mid y \in F_j \text{ for some } j\}\\ &\cup \{y \to \#\langle x y_1\rangle \mid (y, x, y_1) \in \delta_j \text{ for some } j\}\end{aligned}$$

It remains to show that $\eta_1(t) \in \mathcal{L}_{G^\#}$ iff $t \in \mathcal{L}_G$. This is easily done by structural induction, with the help of Lemma 5.1 and Proposition 4.1. □

Example 5.4: Consider the grammar G_1 from Example 5.1. It contains the two regular expressions $r_0 = x^+$ and $r_1 = \epsilon$. The Berry-Sethi construction yields the following NFAs:



Thus $G_1^\# = (\{x, y_0, y_1, y_2\}, y_0, R^\#)$ with the following rules:

$$\begin{array}{lll} x \to a\langle y_0\rangle & y_0 \to \#\langle x y_1\rangle & y_1 \to \$ \\ x \to b\langle y_2\rangle & y_1 \to \#\langle x y_1\rangle & y_2 \to \$ \end{array}$$ □

The converse case is more difficult: A ranked-tree grammar $G^\#$ over $\Sigma^\#$ can produce trees that are not in $\eta_1(\mathcal{F}_\Sigma)$ and thus have no corresponding forest in $\mathcal{F}_\Sigma$. Even if $\mathcal{L}_{G^\#} \subseteq \eta_1(\mathcal{F}_\Sigma)$, a single variable might have two rules $x \to a\langle y_1\rangle$ and $x \to \#\langle y_2\rangle$: There is no clear distinction between variables representing trees and variables representing forests. Therefore we say that such a grammar $G^\#$ is in η_1*-normal form* iff $G^\# = (X \cup Y, y_0, R^\#)$ with $y_0 \in Y$, $X \cap Y = \emptyset$ and:

- ✧ $z \in X$ and $z_1 \in Y$ for all $z \to a\langle z_1 \rangle \in R^{\#}$;
- ✧ $z \in Y$ for all $z \to \$ \in R$;
- ✧ $z, z_2 \in Y$ and $z_1 \in X$ for all $z \to \#\langle z_1 z_2 \rangle \in R$.

Proposition 5.2: For each regular ranked-tree language $L \subseteq \eta_1(\mathcal{F}_\Sigma)$ there is a ranked-tree grammar $G^{\#}$ such that $G^{\#}$ is in η_1-normal form and $\mathcal{L}_{G^{\#}} = L$.

We only sketch the proof: Because L is regular, there is a ranked-tree grammar $G^{\#} = (Z, z_0, R)$ with $\mathcal{L}_{G^{\#}} = L$. We can bring $G^{\#}$ into *reduced* form by eliminating all *non-productive* and *unreachable* variables. (cf. Section 2.1 of [CDG⁺99]). The reduced grammar G^r must be in η_1-normal form because otherwise $\mathcal{L}_{G^r} \not\subseteq \eta_1(\mathcal{F}_\Sigma)$, due to the following argument: Suppose that there is a variable z which has two productions $z \to a\langle z_1 \rangle$ and $z \to \#\langle z_2 z_3 \rangle$. Because all variables are productive and reachable, there must be a tree $t \in \mathcal{L}_{G^r}$ with a subtree $t_1 = a\langle \dots \rangle \in [\![G^r]\!]\, z$. Because $t \in \eta_1(\mathcal{F}_\Sigma)$, t_1 must be the left child of a #-node. On the other hand, t_1 can be replaced in t with an arbitrary $t_2 \in [\![G^r]\!]\, z$, such that the resulting tree t' is also in $\mathcal{L}_{G^r}$. Because all variables are productive, there is such a tree t_2 of the form $\#\langle \dots \rangle$. The resulting tree t' has a #-node as the left child of another #-node and can therefore not be in $\eta_1(\mathcal{F}_\Sigma)$. This is a contradiction, and thus no variable in the reduced grammar can have two such productions. The argument is similar if the second production has the form $z \to \$$ instead of $z \to \#\langle z_2 z_3 \rangle$. □

Proposition 5.3: If $L \subseteq \eta_1(\mathcal{F}_\Sigma)$ is regular then $\eta_1^{-1}(L) \subseteq \mathcal{F}_\Sigma$ is regular. In other words: Let $G^{\#} = (X \cup Y, y_0, R^{\#})$ be a ranked-tree grammar over $\Sigma^{\#}$ in η_1-normal form. Then a forest grammar G over Σ can be constructed with $\mathcal{L}_{G^{\#}} = \eta_1(\mathcal{L}_G)$.

Here is the construction: For each $y \in Y$ with $y = y_0$ or $x \to a\langle y \rangle \in R^{\#}$, we define an NFA (Y, y, F, δ) over X with $F = \{y_1 \mid y_1 \to \$ \in R^{\#}\}$ and $\delta = \{(y_1, x, y_2) \mid y_1 \to \#\langle x y_2 \rangle \in R^{\#}\}$. The language accepted by this NFA is regular and can be denoted as a regular expression r_y. Now $G = (X, r_{y_0}, R)$ with

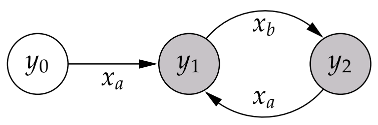
$$R = \{x \to a\langle r_y \rangle \mid x \to a\langle y \rangle \in R^{\#}, a \in \Sigma\}$$

Structural induction shows that $\eta_1(t) \in \mathcal{L}_{G^{\#}}$ iff $t \in \mathcal{L}_G$. □

Example 5.5: Consider the ranked-tree grammar $G^{\#} = (X \cup Y, y_0, R^{\#})$ with $X = \{x_a, x_b\}$, $Y = \{y_0, y_1, y_2\}$, and the following rules:

$x_a \to a\langle y_1 \rangle$	$y_0 \to \#\langle x_a y_1 \rangle$	$y_1 \to \$$
$x_b \to b\langle y_2 \rangle$	$y_1 \to \#\langle x_b y_2 \rangle$	$y_2 \to \$$
	$y_2 \to \#\langle x_a y_1 \rangle$	

The variables in Y and their rules can be interpreted as the following NFA:



Depending on whether the initial state is y_0, y_1 or y_2, the language accepted by the NFA is given by $r_0 = x_a(x_b x_a)^* x_b?$, $r_1 = (x_b x_a)^* x_b?$, or $r_2 = (x_a x_b)^* x_a?$. Thus we obtain the forest grammar $(X, r_0, \{x_a \to a\langle r_1\rangle, x_b \to b\langle r_2\rangle\})$. □

Proposition 5.4: $\eta_1\ (\mathcal{F}_\Sigma)$ is a regular ranked-tree language.

Proof: Let $G_1^\# = (\{x, y\}, y, R_1)$ be the ranked-tree grammar with $R_1 = \{y \to \$,$ $y \to \#\langle xy\rangle\} \cup \{x \to a\langle y\rangle \mid a \in \Sigma\}$. It is easy to see that $\mathcal{L}_{G_1^\#} = \eta_1(\mathcal{F}_\Sigma)$. □

This enables us to transfer the closure results from ranked trees to forests:

Theorem 5.1: Let L_1, L_2 be regular forest languages. Then $L_1 \cup L_2$, $L_1 \cap L_2$, and $L_1^c = \mathcal{T}_\Sigma \setminus L_1$ are also regular forest languages.

Proof: Since L_1 and L_2 are regular, so are $\eta_1(L_1)$ and $\eta_1(L_2)$. Because regular ranked-tree languages are closed under set union, $\eta_1(L_1) \cup \eta_1(L_2)$ is regular. Now $\eta_1(L_1) \cup \eta_1(L_2) = \eta_1(L_1 \cup L_2)$ and thus $L_1 \cup L_2$ is regular. The proof for "$\cap$" is analogous.

For the complement, note that injectivity of η_1 implies that $\eta_1(L^c) = (\eta_1(L))^c \cap \eta_1(\mathcal{F}_\Sigma)$. Now if $L \subseteq \mathcal{F}_\Sigma$ is regular, then so is $\eta_1(L)$. Because regular ranked-tree languages are closed under intersection and complement, $(\eta_1(L))^c \cap \eta_1(\mathcal{F}_\Sigma) = \eta_1(L^c)$ is also regular. Then, by Proposition 5.3, L^c is regular. □

5.3 Bibliographic Notes

Regular tree languages have been considered in the literature since the early sixties. Most authors, however, restrict themselves to the case of ranked trees. Inspired by [Cho60] who deals with the derivation trees of context-free grammars, the first systematic treatment of regular tree languages appears to be [Tha67]: The author characterizes regular languages of non-ranked trees (under the name of recognizable sets of *pseudoterms*) as projections of the derivation trees of extended context-free grammars. He also shows that the regular tree languages are closed under union, intersection and complement.

In [PQ68], non-ranked trees and forests appear under the (French) names *arborescence* and *ramification*. The authors introduce a form of forest grammars capable of describing *local* forest languages; the regular forest languages (called *bilangages réguliers* there) are then homomorphic images of local forest languages.

Restricting himself to the ranked case, [Bra69] introduces so-called *expansive systems* for generating regular tree languages; such a system closely corresponds to our notion of ranked-tree grammars. A characterization of regular tree languages by *monadic second order logic*(MSO) is given by [TW68, Don70].

[Tak75] characterizes regular tree and forest languages by *finite congruences*. She also shows that a string representation of a regular forest language, where each node labeled a is enclosed between parentheses $(_a$ and $)_a$, is a *nest language*, which is a special class of context-free languages.

In our previous work [NS98a], we proposed *μ-formulae* for specification of regular forest languages. Though μ-formulae are more succinct than grammars, they have the disadvantage of being absolutely incomprehensible to non-computer scientists. In a revised version of this paper [NS98b] we used *constraint systems* for specification of regular forest languages. This formalism is similar to grammars, but has an explicit set of forest variables describing the regular expressions over variables, similar to the η_1-normal form of ranked-tree grammars. We abandoned this approach in favor of the more intelligible forest grammars.

Other ways of specifying regular tree languages include algebraic characterizations [GS97] and regular tree expressions [Mur95, GS97]. A good survey of characterizations of regular tree languages is given in [GS97]; an overview of how the results from ranked-tree theory carry over to non-ranked trees and forests is given in [BW98].

Chapter 6

Forest Automata

This chapter introduces a class of bottom-up forest automata accepting the class of regular forest languages. We then enhance these automata with a push-down, significantly increasing succinctness but not expressiveness. Finally we show how to implement forest grammars with these pushdown automata.

6.1 Bottom-Up Forest Automata

In this section we introduce a class of bottom-up automata that accept regular forest languages. For ranked trees such an automaton has a set of states Q and a transition relation Δ with transitions $(q_1 \ldots q_n, a, q)$ with $n = \rho(a)$ (cf., e.g., [Bra69, Mur96]). In order to deal with non-ranked trees, [Tha67, Mur95, BW98] drop the restriction that $n = \rho(a)$ and require instead that the set $\{w \mid (w, a, q) \in \Delta\}$ must be regular for all a, q. An automaton can thus have infinitely many transitions. In order to implement such an automaton, however, the transition relation must be finitely represented, which is most naturally done by constructing a finite automaton for each regular set of words w.

We make this finite automaton explicit in the forest automata, similarly to the construction of [BMW91] in the context of tree pattern matching: There an n-ary transition relation $Q^n \times \Sigma \to Q$ is represented for each $a \in \Sigma$ by a finite automaton on Q, i.e., by a unary transition function $Q_a \times Q \to Q_a$. Accordingly we distinguish two classes of states in a forest automaton: tree states corresponding to the states of conventional automata, and forest states simulating finite automata on words of tree states.

A *left-to-right forest automaton* (LFA) $A = (P, Q, I, F, Up, Side)$ consists of a set of *tree states* P, a set of *forest states* Q, a set of *initial states* I, a set of *final states* F, an *up-relation* $Up \subseteq Q \times \Sigma \times P$ and a *side-relation* $Side \subseteq Q \times P \times Q$. The *size* of an LFA A is the number of states plus the number of transitions in A, i.e. $|A| = |Q| + |P| + |Up| + |Side|$.

Based upon I, Up and $Side$, we define transition relations $\delta_{\mathcal{F}}^A \subseteq \mathcal{F}_\Sigma \times Q$ and $\delta_{\mathcal{T}}^A \subseteq \mathcal{T}_\Sigma \times P$ describing the behavior of A on an input forest:

$$\begin{array}{rl} (\epsilon, q) \in \delta_{\mathcal{F}}^A & \text{for all } q \in I \\ (ft, q_1) \in \delta_{\mathcal{F}}^A & \text{iff } (f, q) \in \delta_{\mathcal{F}}^A,\ (t, p) \in \delta_{\mathcal{T}}^A \text{ and} \\ & (q, p, q_1) \in Side \text{ for some } q \in Q, p \in P \\ (a\langle f \rangle, p) \in \delta_{\mathcal{T}}^A & \text{iff } (f, q) \in \delta_{\mathcal{F}}^A \text{ for some } q \text{ and } (q, a, p) \in Up \end{array}$$

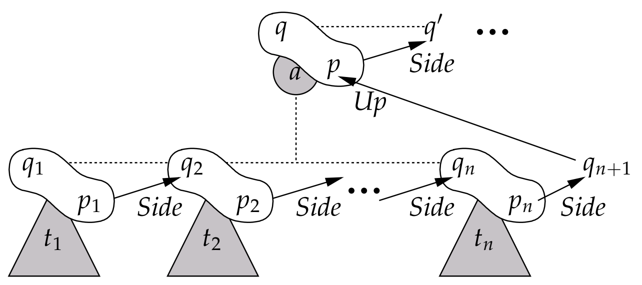


Figure 6.1: The computation model of a left-to-right forest automaton.

If the context is clear, we often omit the superscript A and write simply $\delta_{\mathcal{T}}$ and $\delta_{\mathcal{F}}$. The *language* of an LFA A is $\mathcal{L}_A = \{f \mid (f,q) \in \delta_{\mathcal{F}} \text{ for some } q \in F\}$. The processing model of an LFA is illustrated in Figure 6.1. It can be viewed as a bottom-up procedure: In order to assign a state to a tree $t = a\langle t_1 \ldots t_n\rangle$, a tree state p_i is first assigned to each t_i. Then an initial state q_1 is chosen from I. Traversing the word $p_1 \ldots p_n$ from left to right and performing a side-transition at each step, a forest state q_{n+1} is obtained from q_1. An up-transition for q_{n+1} and a yields a tree state p for t.

6.1.1 Regularity

An LFA processes the word of tree states assigned to the individual trees of a forest by starting with an initial state and proceeding from left to right, applying the side-transition at each tree. This can be simulated by a ranked-tree automaton on the image of the forest under η_2 (cf. Figure 4.3): The states from I are assigned to \$, and transitions at #-nodes simulate the side-transitions. The languages accepted by ranked-tree automata are known to be exactly the regular ranked-tree languages [GS97, CDG$^+$99]. Therefore it is no surprise that the languages accepted by LFAs are the regular forest languages.

Theorem 6.1: A forest language is regular iff it is the language of some forest automaton. More precisely:

(1) For each forest grammar G, there is an LFA A with $\mathcal{L}_A = \mathcal{L}_G$.

(2) For each LFA A, there is a forest grammar G with $\mathcal{L}_G = \mathcal{L}_A$.

Proof of (1): The construction is analogous to that for Proposition 5.1. For $G = (X, r_0, R)$, let $\{r_1, \ldots, r_l\}$ be the set of regular expressions different from r_0 occurring on the right-hand sides of rules, and for each $j = 0, \ldots, l$, let $(Y_j, y_{0,j}, F_j, \delta_j) = \mathit{Berry}(r_j)$ such that $Y_i \cap Y_j = \emptyset$ for $i \neq j$. Let $Y = Y_0 \cup \ldots \cup Y_l$. Then $A = (X, Y, I, F_0, \mathit{Up}, \mathit{Side})$ with

$$\begin{aligned} I &= \{y_{0,j} \mid 0 \leqslant j \leqslant l\} \\ \mathit{Up} &= \{(y, a, x) \mid y \in F_j \text{ and } x \to a\langle r_j\rangle\} \\ \mathit{Side} &= \delta_0 \cup \ldots \cup \delta_l \end{aligned}$$

In order to show that $\mathcal{L}_A = \mathcal{L}_G$, we first prove the following lemma by induction over the length n of a forest:

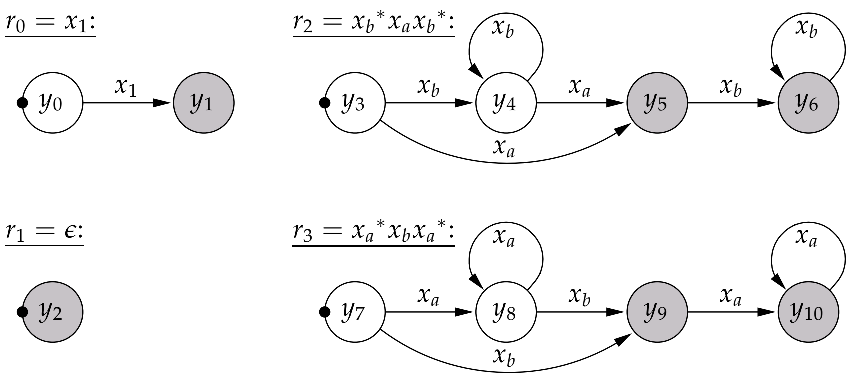


Figure 6.2: The Berry-Sethi construction for the regular expressions in grammar G_3 from Example 5.3.

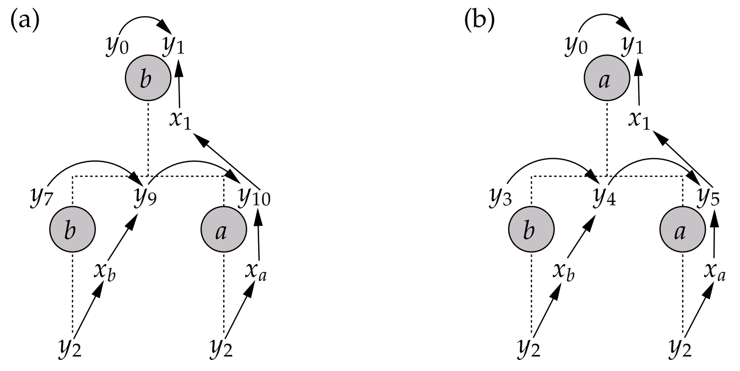


Figure 6.3: Two runs of the LFA A_3 from Example 6.1 on the forests (a) $b\langle ba\rangle$ and (b) $a\langle ba\rangle$.

Lemma 6.1: Let $A = (P, Q, I, F, Up, Side)$ be an LFA, and $f = t_1 \dots t_n$. Then $(f, q) \in \delta_{\mathcal{F}}$ iff there are $p_1, \dots, p_n \in P$ and $q_1, \dots, q_{n+1} \in Q$ with $q_1 \in I$, $q_{n+1} = q$, and for $1 \leqslant i \leqslant n$, $(t_i, p_i) \in \delta_{\mathcal{T}}$ and $(q_i, p_i, q_{i+1}) \in Side$.

Using this lemma, the proof of $\mathcal{L}_A = \mathcal{L}_G$ is a simple structural induction. It is given in Appendix A.1.1. □

Example 6.1: Let us construct an LFA A_3 for grammar G_3 from Example 5.3. Figure 6.2 shows the automata produced by the Berry-Sethi construction for the regular expressions occurring in G_3. Thus $Y = \{y_0, \dots, y_{10}\}$ and $A_3 = (X, Y, I, F, Up, Side)$ with $I = \{y_0, y_2, y_3, y_7\}$, $F = \{y_1\}$ and the following transitions:

$$\begin{aligned} Side = \{&(y_0, x_1, y_1), (y_3, x_b, y_4), (y_3, x_a, y_5), (y_4, x_b, y_4), (y_4, x_a, y_5), \\ &(y_5, x_b, y_6), (y_6, x_b, y_6), (y_7, x_a, y_8), (y_7, x_b, y_9), (y_8, x_a, y_8), \\ &(y_8, x_b, y_9), (y_9, x_a, y_{10}), (y_{10}, x_a, y_{10})\} \end{aligned}$$

$$Up = \{(y_2, a, x_a), (y_2, b, x_b), (y_5, a, x_1), (y_6, a, x_1), (y_9, b, x_1), (y_{10}, b, x_1)\}$$

Figure 6.3 shows two example runs of A_3. □

For the proof of Theorem 6.1 (2), let $A = (P, Q, I, F, Up, Side)$. Find a $q_0 \notin Q$, let $Q_0 = Q \cup \{q_0\}$ and

$$\delta = Side \cup \{(q_0, p, q_1) \mid (q, p, q_1) \in Side \text{ for some } q \in I\}$$

Then define for each $q \in Q$ an NFA $N_q = (Q_0, \{q_0\}, F_q, \delta)$ with $F_q = \{q, q_0\}$ if $q \in I$ and $F_q = \{q\}$ otherwise. $[\![N_q]\!]_{\mathcal{R}}$ is a regular language and can therefore be denoted by a regular expression r_q. Now define $G = (P, r_0, R)$ with

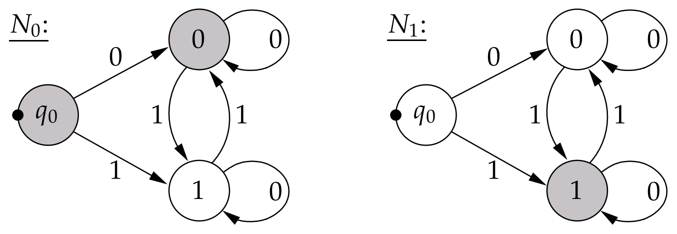
$$\begin{aligned} r_0 &= r_{q_1} \mid \ldots \mid r_{q_k}, \text{ with } F = \{q_1, \ldots, q_k\} \\ R &= \{p \to a\langle r_q \rangle \mid (q, a, p) \in Up\} \end{aligned}$$

For the proof that $\mathcal{L}_G = \mathcal{L}_A$, we first show by structural induction that for $q \in Q$, $(f, q) \in \delta_{\mathcal{F}}$ iff $f \in [\![G]\!]\, r_q$. Given this part of the proof, which is in Appendix A.1.2, it is easy to see that $[\![G]\!]\, r_0 = \bigcup_{q \in F} [\![G]\!]\, r_q = \mathcal{L}_A$. □

Example 6.2: Let us perform the construction for the LFA $A = (\{0,1\}, \{0,1\}, \{0\}, \{1\}, Up, Side)$ with

$$\begin{aligned} Up &= \{(0,a,1), (0,b,0), (1,a,0), (1,b,1)\} \\ Side &= \{(0,0,0), (0,1,1), (1,0,1), (1,1,0)\} \end{aligned}$$

First, we add a state q_0 and construct the automata N_0 and N_1. Note that q_0 is a final state in N_0 because 0 is an initial state in A, whereas q_0 in not final in N_1:



A closer look shows that $r_0 = 0^*(10^*10^*)^*$ and $r_1 = 0^*10^*(10^*10^*)^*$. Thus $G = (\{0,1\}, 0^*10^*(10^*10^*)^*, R)$ with the following rules:

$$\begin{array}{ll} 0 \to a\langle 0^*10^*(10^*10^*)^* \rangle & 1 \to a\langle 0^*(10^*10^*)^* \rangle \\ 0 \to b\langle 0^*(10^*10^*)^* \rangle & 1 \to b\langle 0^*10^*(10^*10^*)^* \rangle \end{array}$$

Observe that $[\![r_0]\!]_{\mathcal{R}}$ is the set of words containing an even number of 1's, whereas r_1 describes words with an odd number of 1's. The language of G and A is therefore the set of forests with an odd number of nodes labeled a.

□

6.1.2 Deterministic and Right-to-Left Forest Automata

An LFA $A = (P, Q, I, F, Up, Side)$ is *deterministic* (DLFA) if $I = \{q_0\}$ is a singleton and $Up : Q \times \Sigma \to P$ and $Side : Q \times P \to Q$ are functions. In other words, there is only one initial state, and during a run of A, there is never a choice between two different transitions. In this case, we write $p = Up_a\, q$ instead of $(q, a, p) \in Up$ and $q_1 = Side(q, p)$ instead of $(q, p, q_1) \in Side$. Note that for a DLFA, $\delta_{\mathcal{T}}$ and $\delta_{\mathcal{F}}$ are also functions: We write $p = \delta_{\mathcal{T}}(t)$ instead of $(t, p) \in \delta_{\mathcal{T}}$ and $q = \delta_{\mathcal{F}}(f)$ for $(f, q) \in \delta_{\mathcal{F}}$.

Similarly to ranked-tree automata, each forest automaton can be made deterministic:

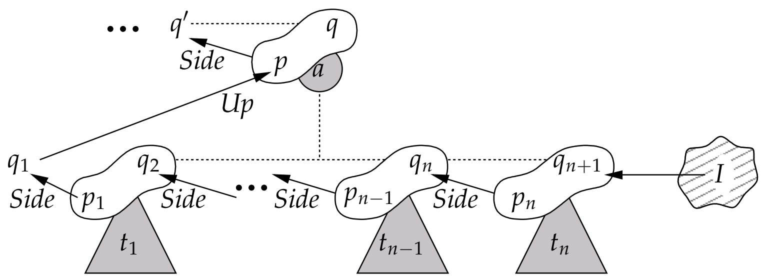


Figure 6.4: The computation model of a right-to-left forest automaton.

Theorem 6.2: For each LFA A, there is a DLFA D with $\mathcal{L}_D = \mathcal{L}_A$.

The proof is by subset construction: Let $A = (P, Q, I, F, Up, Side)$. Then $D = (2^P, 2^Q, \{I\}, F', Up', Side')$ with

$$\begin{aligned} F' &= \{q' \subseteq Q \mid q' \cap F \neq \emptyset\} \\ Up'_a\, q' &= \{p \mid (q, a, p) \in Up \text{ for some } q \in q'\} \\ Side'(q', p') &= \{q_1 \mid (q, p, q_1) \in Side \text{ for some } q \in q', p \in p'\} \end{aligned}$$

Obviously the size of D is exponential in the size of A. The proof that $\mathcal{L}_D = \mathcal{L}_A$ is given in Appendix A.2. □

An LFA traverses the children of a node, more precisely the word of tree states assigned to its children, from left to right. In 4.2.1 we argued that one advantage of the non-ranked tree representation is its suitability for both traversing directions. A *right-to-left forest automaton* (RFA) proceeds through the children of a node from right to left. It is defined in the same way as an LFA with the exception that the second case of the definition of $\delta^A_{\mathcal{F}}$ must be changed as follows:

$$\begin{aligned} (tf, q) \in \delta^A_{\mathcal{F}} \text{ iff } & (f, q) \in \delta^A_{\mathcal{F}},\ (t, p) \in \delta^A_{\mathcal{T}} \text{ and} \\ & (q, p, q_1) \in Side \text{ for some } q \in Q, p \in P \end{aligned}$$

The computation model of an RFA is illustrated in Figure 6.4. Furthermore, we define DRFAs analogously to DLFAs. It can easily be shown that RFAs accept exactly the regular forest languages, and that each RFA can be made deterministic. Hence Theorems 6.1 and 6.2 hold also for RFAs.

6.2 Pushdown Forest Automata

In this section we enhance forest automata with a pushdown. On the one hand, the computation model of pushdown forest automata corresponds more closely to the processing order of an XML parser (and most other parsers for structured documents). On the other hand, the pushdown significantly increases the succinctness of deterministic forest automata. Moreover, pushdown forest automata are useful not only for implementing forest grammars, but also for locating subdocuments in a specified context, as we will see in 7.2.

Conceptually, the behavior of an LFA is a bottom-up procedure: $\delta_{\mathcal{F}}$ can be implemented by first computing a tree state for each tree of a forest, and

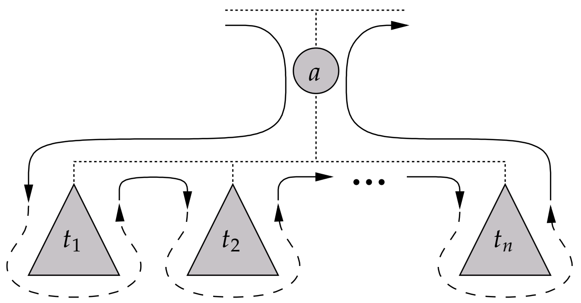


Figure 6.5: The visiting order of a recursive function implementing an LFA.

then traversing this word of tree states while performing side-transitions on forest states. An implementation of this procedure, however, typically uses a pushdown, either implicitly or explicitly. In an SML-like notation, e.g., it could be implemented as follows (for simplicity, we consider the deterministic case here):

```
fun $\delta_{\mathcal{T}}\ a\langle f\rangle = Up_a\ (\delta_{\mathcal{F}}\ f)$
and $\delta_{\mathcal{F}}\ f =$ let val w = map $\delta_{\mathcal{T}}\ f$
              fun doit $q\ \epsilon = q$
                | doit $q\ pw =$ doit $(Side(q,p))\ w$
            in doit $q_0\ w$
            end
```

This implementation implicitly uses a pushdown, namely the function recursion stack. An implementation without recursive functions, i.e., with loops, must even maintain the stack of entered subtrees explicitly. Note that in this implementation, two list traversals are performed in order to compute function $\delta_{\mathcal{F}}$ [1]: one by map and one by doit. A more intelligent variant is the following:

```
fun $\delta_{\mathcal{F}}\ f =$ let fun doit $q\ \epsilon = q$
                 | doit $q\ tf =$ doit $(Side(q,\delta_{\mathcal{T}}\ t))\ f$
            in doit $q_0\ f$
            end
```

Now the first list traversal is spared by incorporating the calls of $\delta_{\mathcal{T}}$ into function doit[2]. Observe that when $\delta_{\mathcal{T}}$ is called for a tree t, all subtrees to its left have already been processed. The forest state q with which the tree state p produced by $\delta_{\mathcal{T}}$ is combined, is already available before the call of $\delta_{\mathcal{T}}$. It is therefore possible to supply q to $\delta_{\mathcal{T}}$ as an additional argument; $\delta_{\mathcal{T}}$ can then use it for optimizing the computation of p.

The visiting order of a recursive function implementing $\delta_{\mathcal{F}}$ is illustrated in Figure 6.5. Note that it coincides with the order in which an XML parser encounters the elements of a document tree: When the parser arrives at an element, the start-tags of all enclosing elements have already been processed, and all elements to the left have been entirely traversed.

We adopt this visiting order and make the pushdown explicit in order to increase the succinctness of forest automata, at least for the deterministic case.

[1]An experienced SML programmer would rewrite the definition of $\delta_{\mathcal{F}}$ more concisely to **fun** $\delta_{\mathcal{F}}\ f =$ foldl (**fn** $(p,q) \Rightarrow Side(q,p)$) q_0 (map $\delta_{\mathcal{T}}\ f$).

[2]Again, this can be rewritten using foldl: **fun** $\delta_{\mathcal{F}}\ f =$ foldl (**fn** $(t,q) \Rightarrow Side(q,\delta_{\mathcal{T}}(t))$) $q_0\ f$.

The idea is as follows: In order to compute $\delta_{\mathcal{F}}(t_1 \dots t_n)$, an LFA must choose an q_1 from I. By a sequence of side-transitions it then obtains a forest state q_{n+1}, which yields together with a a tree state p through an up-transition. But if q_1 is inaptly chosen this can lead to the situation that there is no up-transition (q_{n+1}, a, p). Even if there is an up-transition, p must still be combined with a state q in a succeeding side-transition; it is possible that there is no such transition for q and p.

An example for such a situation is illustrated in Figure 6.3. It shows the runs of automaton A_3 for the two forests $b\langle ba\rangle$ and $a\langle ba\rangle$. This LFA can not distinguish the b subtrees in the two forests. However, we have to choose different initial states for a successful run. If we had chosen y_3 instead of y_7 in case (a), then we would obtain y_5 instead of y_{10} after traversing ba. But for y_5 there is no up-transition under b.

Given an LFA A, it can only be implemented efficiently if it is first made deterministic. An equivalent DLFA D, however, can not choose between initial states: It must simultaneously track all possibilities in A. The subset construction in the proof of Theorem 6.2 achieves this by using sets of states in A as the states of D. The consequence is that the deterministic automaton may need exponentially many states. One source of this blow-up is the following: Each side-transition for a set of states q in D yields a new set of states q', where all of the states in q' are obtained from a state in q through a side-transition A. Obviously, the size of q directly influences the size of q'.

$\delta^D_{\mathcal{F}}$, however, starts with the set of all initial states in A, which is often rather large. As we saw above, many of these states do not lead to a successful up- or side-transition. Constraining the initial states to those that can lead to such a transition keeps the states of D involved in $\delta_{\mathcal{F}}$ smaller. The smaller these sets, the fewer of them can actually occur during a run of D: The number of reachable states in D is often reduced. At the end of this section we will see that this can indeed save an exponential number of states.

In order to select the sensible initial states, the forest state q with which the next side-transition will be performed, must be available to $\delta_{\mathcal{T}}$. At the beginning of this section we argued that, in practice, this does not induce an implementation overhead. We therefore enhance forest automata with a new transition relation *Down*. For the automaton A_3 in Figure 6.3, e.g., *Down* should contain (q_0, a, q_3) and (q_0, b, q_7).

A *left-to-right pushdown forest automaton* (LPA) $A = (P, Q, I, F, \mathit{Down}, \mathit{Up}, \mathit{Side})$ consists, in addition to the components of an LFA, of a *down-relation* $\mathit{Down} \subseteq Q \times \Sigma \times Q$. The elements of I are called *start states* in order to distinguish them from the initial states of an LFA. The *size* of an LPA A is the number of states plus the number of transitions in A, i.e. $|A| = |Q| + |P| + |\mathit{Down}| + |\mathit{Up}| + |\mathit{Side}|$.

Based on *Down*, *Up* and *Side*, the behavior of A is described by relations $\delta^A_{\mathcal{F}} \subseteq Q \times \mathcal{F}_\Sigma \times Q$ and $\delta^A_{\mathcal{T}} \subseteq Q \times \mathcal{T}_\Sigma \times P$ as follows:

$$\begin{array}{rl}
(q, \epsilon, q) \in \delta^A_{\mathcal{F}} & \text{for all } q \in Q \\
(q_1, ft, q_2) \in \delta^A_{\mathcal{F}} & \text{iff } (q_1, f, q) \in \delta^A_{\mathcal{F}},\ (q, t, p) \in \delta^A_{\mathcal{T}} \text{ and} \\
& (q_1, p, q_2) \in \mathit{Side} \text{ for some } q \in Q, p \in P \\
(q, a\langle f\rangle, p) \in \delta^A_{\mathcal{T}} & \text{iff } (q, a, q_1) \in \mathit{Down},\ (q_1, f, q_2) \in \delta^A_{\mathcal{F}} \\
& \text{and } (q_2, a, p) \in \mathit{Up} \text{ for some } q_1, q_2 \in Q
\end{array}$$

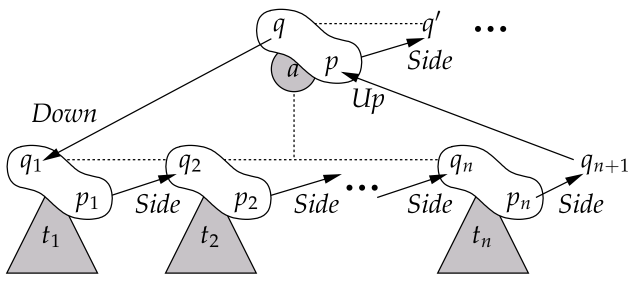


Figure 6.6: The computation model of a left-to-right pushdown automaton.

Again we omit superscript A if the context is clear. The *language* of an LPA is $\mathcal{L}_A = \{f \mid (q_1, f, q_2) \in \delta^A_{\mathcal{F}}$ for some $q_1 \in I, q_2 \in F\}$. The processing model of an LPA is illustrated in Figure 6.6. Note that in contrast to an LFA, an LPA can not be realized without a pushdown: When entering a subtree, the current forest state must be saved on the pushdown; it is popped off the pushdown by the side-transition after that subtree has been completely traversed.

Note that moves on the pushdown are determined by the structure of the input tree: A state is pushed if and only if the automaton descends to the children of a node. Similarly a state is popped from the pushdown exactly after the automaton returns the from children of a node. If we had used a ranked representation for trees of arbitrary arity (cf. 4.2.1), then this relationship were less precise: Moves on the pushdown would have to happen at symbols from Σ, whereas at auxiliary nodes (# and $) the pushdown had to remain untouched. Effectively, this leads to a different class of pushdown automata whose expressiveness probably exceeds that of our pushdown automata, because they are less restrictive concerning the use of the pushdown.

Another possibility would be to use the pushdown at auxiliary nodes as well, by pushing and popping auxiliary states. But then the pushdown would become fairly large: The depth of the ranked-tree representation is the depth of the non-ranked tree plus a number that is influenced by the width of the non-ranked tree, which is in practice significantly larger than its depth.

Moreover, in the ranked-tree image of a non-ranked tree, the direct child relationship is lost: The original children of a node are represented by non-direct descendants in the ranked tree. The up- and down-relations become much more difficult to express, because intermediate transitions through #-nodes are necessary.

6.2.1 Regularity

Though a pushdown forest automaton has a more powerful model of computation, it can always be simulated by a bottom-up automaton. The class of languages accepted by LPAs is therefore the class of regular forest languages.

Theorem 6.3: A forest language is regular iff it is the language of some pushdown forest automaton, more precisely:

(1) For each LFA A, there is an LPA B such that $\mathcal{L}_B = \mathcal{L}_A$;

(2) For each LPA B, there is an LFA A such that $\mathcal{L}_A = \mathcal{L}_B$;

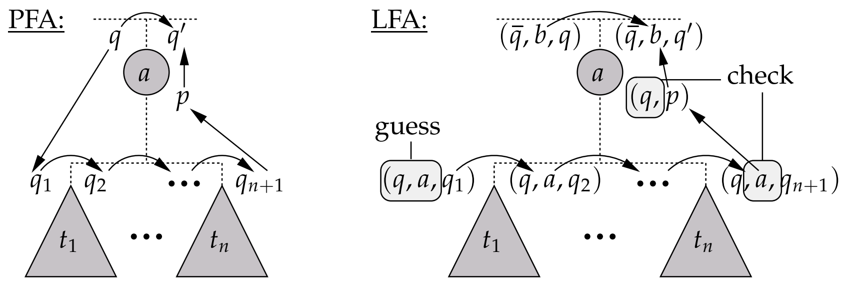


Figure 6.7: Simulation of a pushdown forest automaton by a bottom-up forest automaton.

(1) is trivial: An LFA is an LPA with $Down = \{(q, a, q_1) \mid q \in Q, a \in \Sigma, q_1 \in I\}$, i.e., all states from I may be used as initial states at each point.

For the proof of (2), let $B = (P, Q, I, F, Down, Up, Side)$. The idea is illustrated by Figure 6.7: A simulates $B's$ down-relation by augmenting its states with two components q and a. When A selects an initial state, it guesses a forest state q and a symbol a and chooses a down-transition for them in order to obtain a triple (q, a, q_1). Components q and a are preserved through the succeeding side-transitions. The up-transition is then only defined if a was correctly guessed. Similarly, the next side-transition is only possible if the proper q was guessed. Since a start state of B does not result from a down-transition, a forest state and a symbol need not be guessed for it. At the top-most level of the input forest, the forest states of B are therefore also used also by A. Thus, $A = (Q \times P, Q \cup (Q \times \Sigma \times Q), I \cup Down, F, Up', Side')$ with:

$$\begin{aligned}
Up' &= \{((q, a, q_1), a, (q, p)) \mid (q_1, a, p) \in Up\} \\
Side' &= \{((q, a, q_1), (q_1, p), (q, a, q_2)) \mid (q_1, p, q_2) \in Side\} \\
&\cup \{(q_1, (q_1, p), q_2) \mid (q_1, p, q_2) \in Side\}
\end{aligned}$$

The size of A is quadratic in the size of B: $|A| = \mathcal{O}(|\Sigma| \cdot |B|^2)$.

The proof that $\mathcal{L}_A = \mathcal{L}_B$ is by structural induction with the following induction invariants:

$$\begin{aligned}
\mathcal{A}_{\mathcal{T}}\, t &\equiv (t, (q, p)) \in \delta^A_{\mathcal{T}} \text{ iff } (q, t, p) \in \delta^B_{\mathcal{T}}, \text{ for all } q, p; \\
\mathcal{A}_{\mathcal{F}}\, f &\equiv (f, (q_1, a, q_2)) \in \delta^A_{\mathcal{F}} \text{ iff } (q, f, q_2) \in \delta^B_{\mathcal{F}}, \text{ for } (q, a, q_1) \in Down, \\
&\quad\ \text{and } (f, q_2) \in \delta^A_{\mathcal{F}} \text{ iff } (q_1, f, q_2) \in \delta^B_{\mathcal{F}}, \text{ for all } q_1 \in I, q_2 \in Q.
\end{aligned}$$

We omit the proofs of (E), (L) and (T) because they use only the definitions and are rather technical. □

6.2.2 Deterministic Pushdown Forest Automata

Similarly to LFAs, an LPA $A = (P, Q, I, F, Down, Up, Side)$ is *deterministic* (a DLPA) iff $I = \{q_0\}$ is a singleton and $Down : Q \times \Sigma \to Q$ is a function as well as *Up* and *Side*. In this case, we also write $q_1 = Down_a\, q$ instead of $(q, a, q_1) \in Down$. Furthermore, $\delta_{\mathcal{T}} : Q \times \mathcal{T}_\Sigma \to P$ and $\delta_{\mathcal{F}} : Q \times \mathcal{F}_\Sigma \to Q$ are also functions, and we use the notations $p = \delta_{\mathcal{T}}(q, t)$ and $q_1 = \delta_{\mathcal{F}}(q, f)$.

Interestingly enough, each LPA can be made deterministic. The reason for this is that all runs of an LPA are synchronized, i.e., moves on the pushdown

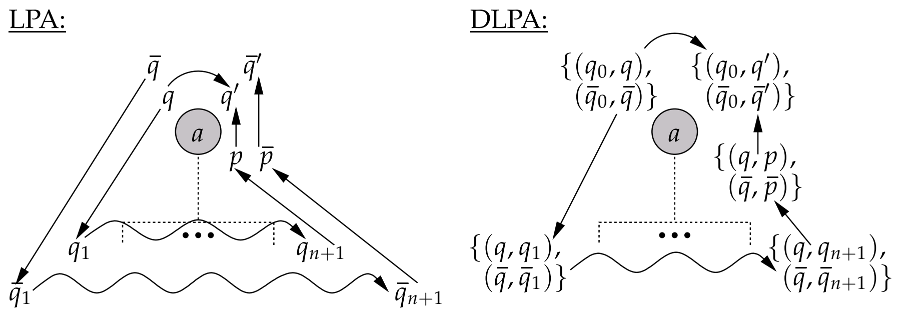

LPA:
$\bar{q}$
$\bar{q}'$
q
q'
a
p
$\bar{p}$
q_1
q_{n+1}
$\bar{q}_1$
$\bar{q}_{n+1}$
DLPA:
$\{(q_0,q),(\bar{q}_0,\bar{q})\}$
$\{(q_0,q'),(\bar{q}_0,\bar{q}')\}$
$\{(q,p),(\bar{q},\bar{p})\}$
$\{(q,q_1),(\bar{q},\bar{q}_1)\}$
$\{(q,q_{n+1}),(\bar{q},\bar{q}_{n+1})\}$

Figure 6.8: The subset construction for pushdown forest automata.

are determined by the structure of the input forest, independent of the state of the automaton.

Theorem 6.4: For each LPA A, there is a DLPA D with $\mathcal{L}_D = \mathcal{L}_A$.

The proof is by subset construction. The conventional approach is to use as states the subsets of P and Q. But this does not suffice: Consider the two partial runs of an LPA in Figure 6.8. A down-transition for q and a yields a forest state q_1 which is transformed into q_{n+1} by $\delta_{\mathcal{F}}$. The up-transition yields a tree state p which is combined with q by the succeeding side-transition. A similar run is obtained for $\bar{q}$, $\bar{q}_1$, $\bar{q}_{n+1}$ and $\bar{p}$. If we perform a conventional subset construction, then the side-transition of the deterministic automaton must combine the two sets $\{q,\bar{q}\}$ and $\{p,\bar{p}\}$: It applies *Side* to all pairs built from these two sets. But then q is combined with $\bar{p}$ which is probably illegal: This constellation might be impossible in the non-deterministic automaton. More precisely, we must ensure that a tree state p is only combined with a forest state q if p was obtained as a consequence of performing a down-transition with q. This relationship is not determinable with sets of tree or forest states. We capture it with sets of pairs (q,p) as tree states and sets of pairs (q,q_1) as forest states, indicating that state p or q_1 was obtained as a consequence of a down-transition with q. This is illustrated in Figure 6.8. Since the start states of A do not result from a down-transition, they are instead paired with themselves.

Formally, define $D = (2^{Q\times P}, 2^{Q\times Q}, \{q_0'\}, F', Down', Up', Side')$ with:

$$\begin{array}{rcl}
q_0' & = & \{(q_1,q_1) \mid q_1 \in I\} \\
F' & = & \{q' \mid (q_1,q_2) \in q' \text{ for some } q_1 \in I, q_2 \in F\} \\
Down_a' \, q' & = & \{(q,q_1) \mid (q_0,q) \in q' \text{ for some } q_0 \text{ and } (q,a,q_1) \in Down\} \\
Up_a' \, q' & = & \{(q,p) \mid (q,q_1) \in q' \text{ and } (q_1,a,p) \in Up \text{ for some } q_1\} \\
Side'(q',p') & = & \{(q,q_2) \mid (q,q_1) \in q',\ (q_1,p) \in p' \text{ and } (q_1,p,q_2) \in Side\}
\end{array}$$

The size of D is exponential in the square of the size of A. The proof that $\mathcal{L}_A = \mathcal{L}_D$ is by structural induction and given in Appendix A.3. □

Theorem 6.3 shows that the pushdown does not increase the expressiveness of forest automata: LPAs accept exactly the same languages as LFAs. In the deterministic case, however, pushdown automata are more succinct than bottom-up automata: For some languages, a DLPA can do with significantly less states than a DLFA.

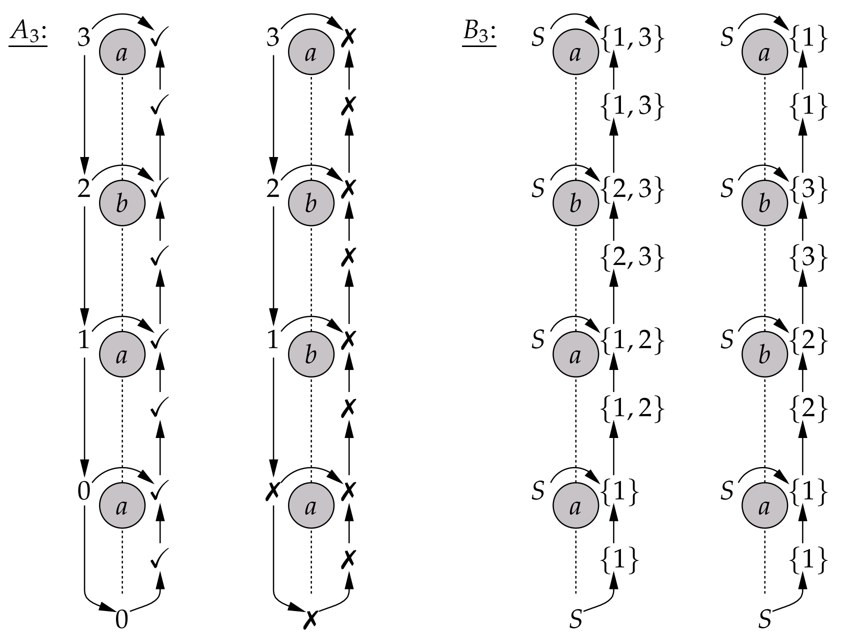


Figure 6.9: Example runs of the DLPA A_3 and the DLFA B_3 in the proof of Theorem 6.5 for input trees $a\langle b\langle a\langle a\rangle\rangle\rangle \in L_3$ and $a\langle b\langle b\langle a\rangle\rangle\rangle \notin L_3$

Theorem 6.5: There is a class of languages $L_1, L_2, \ldots$ such that for all $n > 0$, there is a DLPA A_n accepting L_n with $\mathcal{O}(n)$ states, whereas each DLFA B_n accepting L_n has at least $\boldsymbol{\Omega}(2^n)$ states.

Proof: Let $\Sigma = \{a, b\}$ and L_n be the set of all unary trees that have symbol a at the node at depth n. I.e.,

$$L_1 = \{a\langle t\rangle \mid t \text{ is unary}\} \quad \text{and} \quad L_{i+1} = \{x\langle t\rangle \mid x \in \{a,b\} \text{ and } t \in L_i\}.$$

For $n > 0$, L_n is accepted by the DLPA $A_n = (\{\checkmark, \text{✗}\}, \{0, \ldots, n, \checkmark, \text{✗}\}, \{n\}, \{\checkmark\}, \mathit{Down}, \mathit{Up}, \mathit{Side})$ with:

$$\begin{array}{ll}
\mathit{Down}_x\, i \;= i{-}1, \text{ for } i > 1 & \mathit{Up}_x\, i \;= \text{✗}, \text{ for } i > 0 \\
\mathit{Down}_a\, 1 \;= 0 & \mathit{Up}_x\, 0 \;= \checkmark \\
\mathit{Down}_b\, 1 \;= \text{✗} & \mathit{Up}_x\, \checkmark \;= \checkmark \\
\mathit{Down}_x\, 0 \;= 0 & \mathit{Up}_x\, \text{✗} \;= \text{✗} \\
\mathit{Down}_x\, \checkmark \;= \text{✗} & \\
\mathit{Down}_x\, \text{✗} \;= \text{✗} & \mathit{Side}(i, \checkmark) = \checkmark, \text{ if } i \in \{0, \ldots, n\} \\
 & \mathit{Side}(q, p) \;= \text{✗}, \text{ if } q \notin \{0, \ldots, n\}
\end{array}$$

The idea is that ✓ indicates that the symbol at depth n is an a, whereas ✗ indicates that it is a b. For $i > 0$, entering a subtree with forest state i means that the a must be at depth i in this subtree; state 0 indicates that the current depth is larger than n and an a was found at depth n. Now it is easy to see that $\mathcal{L}_{A_n} = L_n$. Figure 6.9 shows two example runs for $n = 3$. A_n has 2 tree states and $n + 3$ forest states which make a total of $\mathcal{O}(n)$.

On the other hand, a *DLFA* that accepts L_n is $B_n = (\{S, \text{✗}\} \cup 2^{\{1,\ldots,n\}}, \{\text{✗}\} \cup 2^{\{1,\ldots,n\}}, \{S\}, F, \mathit{Up}, \mathit{Side})$ with

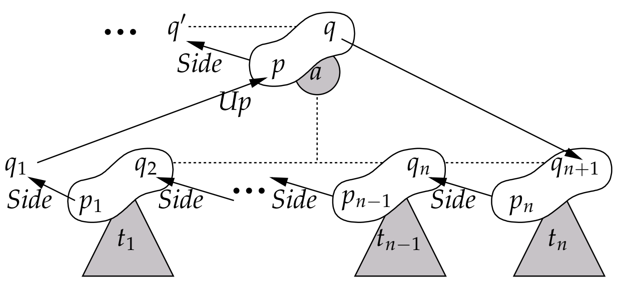


Figure 6.10: The computation model of a right-to-left pushdown automaton.

$$F = \{q \subseteq \{1,\dots,n\} \mid n \in q\} \qquad \begin{aligned} Side(S,p) &= p, \text{ for } p \subseteq \{1,\dots,n\} \\ Side(q,p) &= \text{✗}, \text{ for } q \neq S \text{ or } p = \text{✗} \end{aligned}$$

$$\begin{array}{lll} Up_a\, S = \{1\} & & \\ Up_b\, S = \emptyset & Up_a\, q = \{1\} \cup \{i \leqslant n \mid i-1 \in q\}, & \text{for } q \in \{1,\dots,n\} \\ Up_x\, \text{✗} = \text{✗} & Up_b\, q = \{i \leqslant n \mid i-1 \in q\}, & \text{for } q \in \{1,\dots,n\} \end{array}$$

A state $p \subseteq \{1,\dots,n\}$ indicates that a tree has symbol a at depth i for all $i \in p$, and similarly for forest states. Again, ✗ is an error state and S is the initial state of the automaton. Two example runs of B_3 are shown in Figure 6.9. Because the *DLFA* runs bottom-up, the current depth in the tree is not known when performing a transition. Therefore, a state must always contain the information whether a is at depth i for all $i \in \{1,\dots,n\}$. Therefore B_n has exponentially many states. A formal proof that each DLFA accepting L_n must have $\Omega(2^n)$ states is given in Appendix A.4. □

6.2.3 Right-to-Left Pushdown Forest Automata

LPAs traverse a forest from left to right. Similarly to the bottom-up case, there is also a variant of pushdown automata that proceed in the other direction: A *right-to-left pushdown forest automaton* (RPA) consists of the same components as an LPA. Relations $\delta_{\mathcal{F}}$ and $\delta_{\mathcal{T}}$ are defined in the same way, except for the second case of $\delta_{\mathcal{F}}$, which is replaced as follows:

$$\begin{array}{ll} (q_1, tf, q_2) \in \delta^A_{\mathcal{F}} & \text{iff } (q_1, f, q) \in \delta^A_{\mathcal{F}},\ (q, t, p) \in \delta^A_{\mathcal{T}} \text{ and} \\ & (q_1, p, q_2) \in Side \text{ for some } q \in Q, p \in P \end{array}$$

The traversing order of an RPA is illustrated in Figure 6.10. We also define deterministic right-to-left pushdown automata (DRPA) in analogy to DLPAs. It can be shown that RPAs accept exactly the same languages as LPAs, and that each RPA has an equivalent deterministic automaton. Moreover, each RPA can be simulated by an RFA. Thus Theorems 6.3, 6.4 and 6.5 carry over to right-to-left automata.

As long as we use forest automata for accepting regular forest languages, both left-to-right and right-to-left automata are equally well suited. However, in a document processing system one would prefer the left-to-right automata because their traversing order corresponds to the order in which a document processor naturally traverses a document.

In 7.2 we will use pushdown forest automata for implementing contextual conditions. This requires two consecutive runs of a DLPA and a DRPA, or vice versa. We will then see that the traversing order of automata is of high importance.

6.3 Decision Problems

In this section we treat some decision problems for forest automata and relate them to the corresponding results for ranked-tree automata. The problems we consider are:

Emptiness: For a given forest automaton A, decide whether its language is empty, i.e., $\mathcal{L}_A = \emptyset$.

Inclusion: For two forest automata A_1 and A_2, decide whether the language of the first is included in the language of the second, i.e., $\mathcal{L}_{A_1} \subseteq \mathcal{L}_{A_2}$.

Equivalence: Decide whether two forest automata A_1 and A_2, are equivalent, i.e., $\mathcal{L}_{A_1} = \mathcal{L}_{A_2}$.

We will first solve these problems for bottom-up forest automata; we will then modify the described algorithms in order to give the solutions for pushdown automata.

6.3.1 Bottom-Up Forest Automata

We solve the emptiness problem for LFAs with the help of boolean systems of inequations. The two other problems are based upon this result.

Emptiness

The question whether the language of an LFA $A = (P, Q, I, F, Up, Side)$ is empty can be decided in linear time. The algorithm is as follows: First, we construct a system S of boolean inequations. A variable of S is either $\langle p \rangle$ with $p \in P$ or $\langle q \rangle$ with $q \in Q$:

$$\begin{array}{lll} \langle p \rangle & \leftarrow \langle q \rangle, & \text{if } (q, a, p) \in Up \text{ for some } a \\ \langle q \rangle & \leftarrow true, & \text{for } q \in I \\ \langle q_2 \rangle & \leftarrow \langle q_1 \rangle \wedge \langle p \rangle & \text{if } (q_1, p, q_2) \in Side \end{array}$$

Note that the alphabet Σ has no influence on the size of S. The size of S is the number of transitions plus the number of initial states in A, thus $|S| \leqslant |A|$. Because S is a boolean system, its least solution σ can be computed in linear time (An algorithm is given, e.g., in [WM95] in the context of grammar flow analysis).

It is easy to see that a variable describes the *productivity* of the corresponding state:

$$\begin{array}{ll} \sigma\langle p \rangle = true & \text{iff there is a tree } t \text{ with } (t, p) \in \delta_{\mathcal{T}} \\ \sigma\langle q \rangle = true & \text{iff there is a forest } f \text{ with } (f, q) \in \delta_{\mathcal{F}} \end{array}$$

The language of A is then empty iff $\sigma\langle q \rangle = false$ for all $q \in F$. Emptiness of $\mathcal{L}_A$ can thus be decided in time $\mathcal{O}(|A|)$.

Inclusion

Let $A_1 = (P_1, Q_1, I_1, F_1, Up_1, Side_1)$ and $A_2 = (P_2, Q_2, I_2, F_2, Up_2, Side_2)$. In order to decide whether $\mathcal{L}_{A_1} \subseteq \mathcal{L}_{A_2}$, we check whether $\mathcal{L}_{A_1} \setminus \mathcal{L}_{A_2} = \emptyset$. For this purpose, we perform the following steps:

1. First we use the subset construction (see the proof of Theorem 6.2) for obtaining a DLFA $B = (P_B, Q_B, I_B, F_B, Up_B, Side_B)$ with $\mathcal{L}_B = \mathcal{L}_{A_2}$. The states of B are the subsets of P_2 and Q_2; the size of B is thus exponential in the size of A_2, i.e., $|B| = \mathcal{O}(2^{|A_2|})$.

2. Then we define the *complement automaton* B^c with $\mathcal{L}_{B^c} = \mathcal{L}^c_{A_2} = \mathcal{F}_\Sigma \setminus \mathcal{L}_{A_2}$. Because B is deterministic, we can obtain B^c by inverting the set of final states of B: $B^c = (P_B, Q_B, I_B, F^c_B, Up_B, Side_B)$, where $F^c_B = Q_B \setminus F_B$. It is easy to see that $\mathcal{L}_{B^c} = \mathcal{L}^c_B$. The size of B^c is equal to the size of B, i.e., $|B^c| = \mathcal{O}(2^{|A_2|})$.

3. Now we construct an LFA C with $\mathcal{L}_C = \mathcal{L}_{A_1} \cap \mathcal{L}_{B^c} = \mathcal{L}_{A_1} \setminus \mathcal{L}_{A_2}$. C is the *product automaton* of A_1 and B^c, defined as $C = (P_1 \times P_B, Q_1 \times Q_B, I_1 \times I_B, F_1 \times F^c_B, Up_C, Side_C)$, where

$$\begin{aligned} Up_C &= \{((q_1, q'), a, (p_1, p')) \mid (q_1, a, p_1) \in Up_1,\ (q', a, p') \in Up_B\} \\ Side_C &= \{((q_1, q'_1), (p, p'), (q_2, q'_2)) \mid (q_1, p, q_2) \in Side_1, \\ &\qquad\qquad (q'_1, p', q'_2) \in Side_B\} \end{aligned}$$

   It is easy to see that $f \in \mathcal{L}_C$ iff $f \in \mathcal{L}_{A_1}$ and $f \in \mathcal{L}_{B^c}$. Thus $\mathcal{L}_C = \mathcal{L}_{A_1} \setminus \mathcal{L}_{A_2}$.

4. Emptiness of $\mathcal{L}_C$ can now be decided in time linear to the size of C. The size of C is bounded by the product of the sizes of A_1 and B^c, thus emptiness of $\mathcal{L}_C$ can be decided in *DEXPTIME*.

If A_2 is a deterministic automaton, then we can skip step 1, sparing the expensive subset construction. In this case the size of C is bounded by the product of the sizes of A_1 and A_2: The inclusion problem can then be decided in quadratic time.

Equivalence

The question whether $\mathcal{L}_{A_1} = \mathcal{L}_{A_2}$ can be reduced to the two inclusion problems $\mathcal{L}_{A_1} \subseteq \mathcal{L}_{A_2}$ and $\mathcal{L}_{A_2} \subseteq \mathcal{L}_{A_1}$. Both of these can be solved in *DEXPTIME*, thus the equivalence problem for LFAs is also decidable in *DEXPTIME*.

It is easy to see that each conventional ranked-tree automaton can be simulated be an LFA of linear size. The equivalence problem for ranked-tree automata was shown to be *DEXPTIME*-complete by [Sei90]; thus it is also *DEXPTIME*-complete for LFAs.

For deterministic automata A_1 and A_2, however, the inclusion problem can be solved in quadratic time, and so can the equivalence problem.

6.3.2 Pushdown Forest Automata

The decision problems for pushdown forest automata are solved in a similar way as those for bottom-up automata. However, things are more complicated due to the down-transitions of pushdown automata.

Emptiness

The question whether the language of an LPA $A = (P, Q, I, F, Down, Up, Side)$ is empty can be decided in quadratic time. The algorithm is as follows: First, we establish a system S of boolean inequations. A variable of S is either $<q, p>$ or or $<q, q_1>$ with $p \in P$ and $q, q_1 \in Q$:

$$\begin{array}{lll} <q,p> & \leftarrow <q_1,q_2>, & \text{for } (q,a,q_1) \in Down \text{ and } (q_2,a,p) \in Up \\ <q,q> & \leftarrow true, & \text{for all } q \\ <q,q_2> & \leftarrow <q,q_1> \wedge <q_1,p> & \text{for } q \in Q \text{ and } (q_1,p,q_2) \in Side \end{array}$$

The size of S is quadratic in the size of A: There are at most $|Down| \cdot |Up|$ inequations of the first form, $|Q|$ inequations of the second form, and $|Q| \cdot |Side|$ inequations of the third form. Because S is a boolean system, its least solution σ can be computed in linear time.

A variable of S describes the *one-level reachability* of a state p or q_1 from a state q:

$$\begin{array}{ll} \sigma<q,p> = true & \text{iff there is a tree } t \text{ with } (q,t,p) \in \delta_{\mathcal{T}} \\ \sigma<q,q_1> = true & \text{iff there is a forest } f \text{ with } (q,f,q_1) \in \delta_{\mathcal{F}} \end{array}$$

The language of A is then empty iff $\sigma<q_1, q_2> = false$ for all $q_1 \in I$, $q_2 \in F$. Emptiness of $\mathcal{L}_A$ can thus be decided in time $\mathcal{O}(|A|^2)$.

To a certain degree this result is astounding: Emptiness for an LPA is easier to decide than for a pushdown automaton on words, which requires cubic time w.r.t. the size of the automaton.

Inclusion

The decision procedure is analogous to that for bottom-up automata: Let $A_1 = (P_1, Q_1, I_1, F_1, Down_1, Up_1, Side_1)$ and $A_2 = (P_2, Q_2, I_2, F_2, Down_2, Up_2, Side_2)$. In order to decide whether $\mathcal{L}_{A_1} \subseteq \mathcal{L}_{A_2}$, we check whether $\mathcal{L}_{A_1} \setminus \mathcal{L}_{A_2} = \emptyset$. We perform the following steps:

1. First we use the subset construction (see the proof of Theorem 6.4) for obtaining a DLPA $B = (P_B, Q_B, I_B, F_B, Down_B, Up_B, Side_B)$ with $\mathcal{L}_B = \mathcal{L}_{A_2}$. The states of B are the subsets of $Q_2 \times P_2$ and $Q_2 \times Q_2$; the size of B is thus exponential in the square of the size of A_2, i.e., $|B| = \mathcal{O}(2^{|A_2|^2})$.

2. Next we define the complement automaton B^c with $\mathcal{L}_{B^c} = \mathcal{L}^c_{A_2} = \mathcal{F}_\Sigma \setminus \mathcal{L}_{A_2}$. Because B is deterministic, we obtain B^c by inverting the set of final states of B. The size of B^c is equal to the size of B, i.e., $|B^c| = \mathcal{O}(2^{|A_2|^2})$.

3. Now we construct the product automatonC of A_1 and B^c with $\mathcal{L}_C = \mathcal{L}_{A_1} \cap \mathcal{L}_{B^c} = \mathcal{L}_{A_1} \setminus \mathcal{L}_{A_2}$. It is defined analogously to the bottom-up case: $C = (P_1 \times P_B, Q_1 \times Q_B, I_1 \times I_B, F_1 \times F^c_B, Down_C, Up_C, Side_C)$, where

$$\begin{aligned} Down_C &= \{((q_1,q_1'),a,(q_2,q_2')) \mid (q_1,a,q_2) \in Down_1, \\ &\qquad\qquad (q_1',a,q_2') \in Down_B\} \\ Up_C &= \{((q_1,q'),a,(p_1,p')) \mid (q_1,a,p_1) \in Up_1,\ (q',a,p') \in Up_B\} \\ Side_C &= \{((q_1,q_1'),(p,p'),(q_2,q_2')) \mid (q_1,p,q_2) \in Side_1, \\ &\qquad\qquad (q_1',p',q_2') \in Side_B\} \end{aligned}$$

It is easy to see that $f \in \mathcal{L}_C$ iff $f \in \mathcal{L}_{A_1}$ and $f \in \mathcal{L}_{B^c}$. Thus $\mathcal{L}_C = \mathcal{L}_{A_1} \setminus \mathcal{L}_{A_2}$.

4. Emptiness of $\mathcal{L}_C$ can now be decided in time quadratic to the size of C. The size of C is bounded by the product of the sizes of A_1 and B^c, thus emptiness of $\mathcal{L}_C$ can be decided in *DEXPTIME*.

If A_2 is deterministic, then we can skip the subset construction in step 1. In this case the size of C is bounded by the product of the sizes of A_1 and A_2: The inclusion problem can then be decided in polynomial time.

Equivalence

The question whether $\mathcal{L}_{A_1} = \mathcal{L}_{A_2}$ can be reduced to the two inclusion problems $\mathcal{L}_{A_1} \subseteq \mathcal{L}_{A_2}$ and $\mathcal{L}_{A_2} \subseteq \mathcal{L}_{A_1}$. Both of these can be solved in *DEXPTIME*, thus the equivalence problem for LPAs is also decidable in *DEXPTIME*.

Because LFAs are a special case of LPAs, and the equivalence problem for LFAs is *DEXPTIME*-complete, so is the equivalence problem for LPAs.

However, if A_1 and A_2 are deterministic, then the inclusion problem can be decided in polynomial time, and so can the equivalence problem.

6.4 Matching Structural Conditions

An important task in document processing is verifying a structural property of a document tree. For most purposes, this structural property can be given as a regular forest language. For instance, the set of element-type declarations of an XML DTD establish a special form of a forest grammar and therefore describe a regular forest language [3]. Validating the structure of an XML document against a DTD is therefore a test for membership in a regular language. A good algorithm implementing forest grammars is therefore of high importance in document processing.

6.4.1 Matching Structure with Bottom-Up Automata

One possibility of implementing a given forest grammar G is to construct an LFA according to the construction in the proof of Theorem 6.1. This automaton is non-deterministic and must be made deterministic in order to run it efficiently on a computer. Applying the subset construction in the proof of Theorem 6.2 yields a DLFA B_G with $\mathcal{L}_{B_G} = \mathcal{L}_G$. Combining both steps into one, we obtain the following construction:

Let $G = (X, r_0, R)$ and $\{r_1, \ldots, r_l\}$ be the set of regular expressions different from r_0 occurring on the right-hand sides of rules in R. For each $j \in \{0, \ldots, l\}$, let $(Y_j, q_{0,j}, F_j, \delta_j) = \mathit{Berry}(r_j)$, such that $Y_i \cap Y_j = \emptyset$ for $i \neq j$. Let $Y = Y_0 \cup \ldots \cup Y_l$ and $\delta = \delta_0 \cup \ldots \cup \delta_l$. Then $B_G = (2^X, 2^Y, \{q_0\}, F, \mathit{Up}, \mathit{Side})$ with

$$\begin{aligned} q_0 &= \{y_{0,j} \mid 0 \leqslant j \leqslant l\} \\ F &= \{q \mid q \cap F_0 \neq \emptyset\} \\ \mathit{Up}_a\, q &= \{x \mid x \to a\langle r_j\rangle \text{ and } q \cap \mathcal{F}_j \neq \emptyset\} \\ \mathit{Side}(q, p) &= \{y_1 \mid y \in q, x \in p \text{ and } (y, x, y_1) \in \delta\} \end{aligned}$$

Example 6.3: Consider the grammar G_3 from Example 5.3. B_{G_3} is obtained by

[3] More precisely, a DTD is a *local* forest language. For a discussion of local ranked-tree languages see, e.g. [GS97]. A generalization of DTDs to describe regular forest languages are XML Schemata [W3C99h].

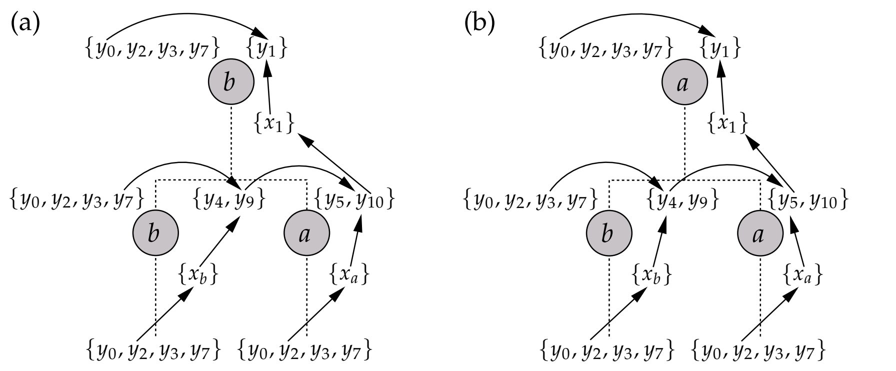


Figure 6.11: Two runs of the DLFA B_{G_3} from Example 6.4 on the forests (a) $b\langle ba\rangle$ and (b) $a\langle ba\rangle$.

applying the subset construction to the LFA A_3 from Example 6.1. Figure 6.11 shows two example runs of B_{G_3}. □

Theoretically, B_G has exponentially many states: Its tree states are sets of variables, and its forest states are sets of NFA states. Many of these states can never occur during a run of the automaton. A clever implementation will therefore compute the *reachable states* and store *Up*- and *Side*-transitions only for these states.

Figure 6.12 shows an algorithm[4] for computing the reachable states of a DLFA. It maintains in the work-list W a set of transitions for reachable states which have not been processed yet. An entry in the work-list can have two forms: UP(q) represents all up-transitions for a forest state q, whereas SIDE(q, p) is for the side-transition under forest state q and tree state p. Initially, the only state known to be reachable is the start state q_0; hence the work-list is initialized with UP(q_0).

The algorithm then repeatedly extracts an entry from the work-list and performs the transitions indicated by that entry. If such a transition yields a state that was not yet known to be reachable, new entries are added to the work-list for this state. In case of a tree state, these are side-transitions with all reachable forest states; for a new forest state, an UP-entry and SIDE-entries for all reachable forest states are added. As soon as the work-list is empty, all reachable states have been found and processed.

Example 6.4: Consider again the DLFA B_{G_3} from Example 6.3 obtained by subset construction from the LFA A_3. A_3 has a set Y of 11 forest states and a set X of 4 tree states. The forest states of B_{G_3} are the subsets of Y, and its tree states are the subsets of X. Thus there are $2^{11} = 2048$ forest states and $2^4 = 16$ tree states. But most of them are unreachable: Running algorithm *ReachDlfa* yields that only the following 4 tree states and 13 forest states are reachable in B_{G_3}:

[4]We do not use an SML-like notation for this algorithm because an imperative notation is clearer for this purpose. However, we do use SML's concept of pattern matching.

```
Algorithm ReachDlfa
Input:   An alphabet $\Sigma$
         A DLFA $A = (P, Q, \{q_0\}, F, Up, Side)$
Output:  The set $\mathcal{R}_P \subseteq P$ of tree states reachable in $A$
         The set $\mathcal{R}_Q \subseteq Q$ of forest states reachable in $A$
Algorithm:

  $\mathcal{R}_Q := \{q_0\};\ \mathcal{R}_P := \emptyset;\ W := \{\mathsf{UP}(q_0)\};$
  while $W \neq \emptyset$ do
     item := select($W$); $W := W \setminus \{$item$\}$;
     case item
        of UP($q$) $\Rightarrow$ foreach $a \in \Sigma$ do
                           $p := Up_a\, q;$
                           if $p \notin \mathcal{R}_P$ then
                              $\mathcal{R}_P := \mathcal{R}_P \cup \{p\};$
                              foreach $q_1 \in \mathcal{R}_Q$ do
                                 $W := W \cup \{\mathsf{SIDE}(q_1, p)\};$
        | SIDE($q, p$) $\Rightarrow$ $q_1 := Side(q, p);$
                             if $q_1 \notin \mathcal{R}_Q$ then
                                $\mathcal{R}_Q := \mathcal{R}_Q \cup \{q_1\};$
                                $W := W \cup \{\mathsf{UP}(q_1)\};$
                                foreach $p \in \mathcal{R}_P$ do
                                   $W := W \cup \{\mathsf{SIDE}(q_1, p)\};$
  return $\mathcal{R}_P$, $\mathcal{R}_Q$;
```

Figure 6.12: An algorithm for computing the reachable states of a DLFA.

Tree States:	Forest States:		
$p_0 = \{x_a\}$	$q_0 = \{y_0, y_2, y_3, y_7\}$	$q_4 = \{y_8\}$	$q_8 = \{y_1\}$
$p_1 = \{x_b\}$	$q_1 = \{y_5, y_8\}$	$q_5 = \{y_4\}$	$q_9 = \{y_6\}$
$p_2 = \{x_1\}$	$q_2 = \{y_4, y_9\}$	$q_6 = \{y_5, y_{10}\}$	$q_{10} = \{y_{10}\}$
$p_3 = \emptyset$	$q_3 = \{y_6, y_9\}$	$q_7 = \emptyset$	$q_{11} = \{y_9\}$
			$q_{12} = \{y_5\}$

This is also the order in which the algorithm finds these states. □

This example demonstrates that in spite of the theoretically immense size of B_G even for small grammars G, the reachable states of the automaton are usually few. It can therefore be implemented efficiently. In some cases, however, the DLFA must have exponentially many reachable states, as Theorem 6.5 shows. The example language L_n in the proof of this theorem is easily expressed as a forest grammar $G_n = (\{x_0, \ldots, x_n\}, x_n, R)$ with the following rules:

$$\begin{array}{ll} x_0 \to a\langle x_0?\rangle & \left.\begin{array}{l} x_i \to a\langle x_{i-1}\rangle \\ x_i \to b\langle x_{i-1}\rangle \end{array}\right\} \text{ for } 1 < i \leqslant n \\ x_0 \to b\langle x_0?\rangle & \\ x_1 \to a\langle x_0?\rangle & \end{array}$$

We can now construct B_{G_n} in order to obtain a DLFA accepting L_n. This automaton is identical (up to a renaming of states) to the DLFA B_n in the proof of Theorem 6.5. Running algorithm *ReachDlfa* results in (2^n+1) reachable tree states and (2^n+2) reachable forest states.

The reason for this exponential blow-up is that a DLFA is unaware of the ancestors and thus also of the depth of the current node when it makes a transition: It must therefore keep track of a's at all depths that do not exceed n. On the contrary, the pushdown automaton can count the current depth with the help of its down-transitions. We will pursue this idea in the next section and modify the construction of B_G to yield a pushdown automaton.

6.4.2 Matching Structure with Pushdown Automata

In order to motivate the definition of a down-relation when implementing structural conditions, let us look again at the two runs of B_{G_3} in Figure 6.11. Though the two input forests are different, the automaton uses exactly the same states in both runs. The forest states at the second level are of particular interest: In case (a), the root of the forest is labeled b; thus its children must match the regular expression $x_a{}^*x_bx_a{}^*$. This regular expression is implemented by forest states $y_7,\dots,y_{10}$. In order to verify this particular regular expression, it would therefore suffice to start with the singleton set $\{y_7\}$ instead of $q_0=\{y_0,y_2,y_3,y_7\}$. Similarly, in case (b), we are only interested in the regular expression $x_b{}^*x_ax_b{}^*$, which is represented by forest states $y_3,\dots,y_6$. In this case, it would be sensible to start with the singleton $\{y_3\}$. A DLFA can not decide which initial states are interesting at a certain point; therefore it must always start with all of them.

As opposed to that, a pushdown automaton can select the interesting initial states by its down-transition. Starting with a smaller set at the left-most tree of a forest, we can assume that fewer states are reachable from that set on this level. We will now modify our construction of B_G to yield a pushdown automaton. The down-relation of this automaton selects only those $y_{0,j}$ which might lead to a tree variable contributing to a side-transition.

For a forest grammar $G=(X,r_0,R)$, let $\{r_1,\dots,r_l\}=\{r\neq r_0\mid x\to a\langle r\rangle\in R\}$. Moreover, for $0\leqslant j\leqslant l$, let $(Y_j,y_{0,j},F_j,\delta_j)=\mathit{Berry}(r_j)$ such that $Y_i\cap Y_j=\emptyset$ for $i\neq j$. Let $Y=Y_0\cup\ldots\cup Y_l$ and $\delta=\delta_0\cup\ldots\cup\delta_l$. Then the DLPA $A_{\vec{G}}$ is defined as $(2^X,2^Y,\{q_0\},F,\mathit{Down},\mathit{Up},\mathit{Side})$ with

$$\begin{aligned}
q_0 &= \{y_{0,0}\}\\
F &= \{q\mid q\cap F_0\neq\emptyset\}\\
\mathit{Down}_a\,q &= \{y_{0,j}\mid y\in q,\ (y,x,y_1)\in\delta,\ x\to a\langle r_j\rangle \text{ for some } x,y_1\}\\
\mathit{Up}_a\,q &= \{x\mid x\to a\langle r_j\rangle \text{ and } q\cap\mathcal{F}_j\neq\emptyset\}\\
\mathit{Side}(q,p) &= \{y_1\mid y\in q, x\in p \text{ and } (y,x,y_1)\in\delta\}
\end{aligned}$$

Theorem 6.6: For each forest grammar G, $\mathcal{L}_{A_{\vec{G}}}=\mathcal{L}_G$

The proof is by structural induction and given in Appendix A.5 □

Example 6.5: Consider again grammar G_3 from Example 6.4. Figure 6.13 shows the runs of $A_{\vec{G}_3}$ for the two input forests from Figure 6.11. Note that the initial states for traversing the second level are different for both input forests: In the first case, $\mathit{Down}_b\,\{y_0\}=\{y_7\}$, because there is only one side-transition for y_0; this uses variable x_1, which has only a single rule for symbol b. The initial state for the regular expression $x_a{}^*x_bx_a{}^*$ in this rule is y_7. Similarly, in the second

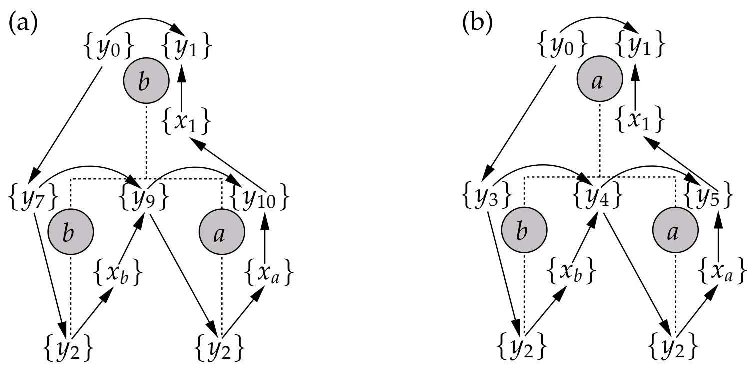


Figure 6.13: Two runs of the DLPA $A_{\vec{G}_3}$ from Example 6.5 on the forests (a) $b\langle ba\rangle$ and (b) $a\langle ba\rangle$.

run, $Down_a\,\{y_0\} = \{y_3\}$, because $x_b{}^*x_ax_b{}^*$ this is the only interesting regular expression if the top-most symbol is a. □

6.4.2.1 Reachable States of Pushdown Automata

$A_{\vec{G}}$, like B_G, has exponentially many states, though a large number of them are not reachable indeed. Similarly to DLFAs, we can also compute the set of reachable states of a DLPA. An algorithm for accomplishing this is given in Figure 6.14.

It is fairly more complex than for DLFAs: There, each reachable forest state can be combined with each reachable tree state in a side-transition. In a pushdown automaton, it is not evident for a reachable forest state q and a reachable tree state p that these two are ever input to a side-transition: That can only happen if p is reachable from q, i.e., if there is a tree t such that $p = \delta_{\mathcal{T}}(q,t)$. Similarly, for a reachable forest state q and symbol a, an up-transition under q and a can only happen if q is *one-level-reachable* from a state q_0 that is the result of a down-transition for a, i.e., if there is a q' and a forest f such that $q = \delta_{\mathcal{F}}(Down_a\,q', f)$. Note that, unlike up-transitions, a down-transition can occur for all symbols a and reachable forest states q.

The consequence is that, in addition to the sets $\mathcal{R}_Q$ and $\mathcal{R}_P$ of reachable forest and tree states, the algorithm also computes the sets $\mathcal{R}_{Up}$ and $\mathcal{R}_{Side}$ of reachable up- and side-transitions. Moreover, the algorithm must maintain for each forest state q the set $r_P[q]$ of tree states reachable from q, and the set $r_Q[q]$ of forest states one-level-reachable from q.

The algorithm initializes the set $\mathcal{R}_Q$ of reachable forest states with the singleton set containing only q_0 and calls procedure init for that state. The task of this procedure is to find all forest states q_1 reachable by applying only down-transitions, and to initialize $r_Q[q_1]$ and $r_P[q_1]$ for these states. In each iteration of the main loop, the sets $r_Q[q]$ and $r_P[p]$ are recomputed for each reachable state q found yet. If this involves a forest state q_2 that was not known to be reachable yet, q_2 is initialized with procedure init. The main loop terminates if none of the $r_Q[]$ and $r_P[]$ sets was grown by the recomputation.

A more general approach of computing the reachable states and transitions is to construct a system of equations describing $r_Q[q]$ and $r_P[q]$ for each q:

Algorithm *ReachDlpa*

Input: An alphabet Σ
A DLPA $A = (P, Q, \{q_0\}, F, Down, Up, Side)$

Output: The set $\mathcal{R}_P \subseteq P$ of tree states reachable in A
The set $\mathcal{R}_Q \subseteq Q$ of forest states reachable in A
The set $\mathcal{R}_{Up} \subseteq Up$ of up-transitions reachable in A
The set $\mathcal{R}_{Side} \subseteq Side$ of side-transitions reachable in A

Algorithm:

```
proc new_rp($q$)
    foreach $a \in \Sigma$ do
        foreach $q_1 \in r_Q[Down_a\, q]$ do
            $p := Up_a\, q_1$; $\mathcal{R}_P := \mathcal{R}_P \cup \{p\}$; $\mathcal{R}_{Up} := \mathcal{R}_{Up} \cup \{(q_1, a, p)\}$
            if $p \notin r_P[q]$ then
                done := false; $r_P[q] := r_P[q] \cup \{p\}$;

proc new_rq($q$)
    foreach $q_1 \in r_Q[q]$ do
        foreach $p \in r_P[q_1]$ do
            $q_2 := Side(q_1, p)$; $\mathcal{R}_{Side} := \mathcal{R}_{Side} \cup \{(q_1, p, q_2)\}$
            if $q_2 \notin r_Q[q]$ then
                done := false; $r_Q[q] := r_Q[q] \cup \{q_2\}$; init($q_2$);

proc init(q)
    if $q \notin \mathcal{R}_Q$ then
        $\mathcal{R}_Q := \mathcal{R}_Q \cup \{q\}$; $r_Q[q] := \{q\}$; $r_P[q] := \emptyset$;
        foreach $a \in \Sigma$ do
            init($Down_a\, q$);

$\mathcal{R}_Q := \emptyset$; $\mathcal{R}_P := \emptyset$; $\mathcal{R}_{Up} := \emptyset$; $\mathcal{R}_{Side} := \emptyset$; init($q_0$);
repeat
    done := true;
    foreach $q \in \mathcal{R}_Q$ do
        new_rp($q$); new_rq($q$);
until done = true;
return $\mathcal{R}_P, \mathcal{R}_Q, \mathcal{R}_{Up}, \mathcal{R}_{Side}$;
```

Figure 6.14: An algorithm for computing the reachable states and transitions of a DLPA.

$$r_P[q] = \{Up_a\, q_1 \mid q_1 \in r_Q[Down_a\, q], a \in \Sigma\}$$
$$r_Q[q] = \{q\} \cup \{Side(q_1, p) \mid q_1 \in r_Q[q] \text{ and } p \in r_P[q_1]\}$$

Note the similarity to the boolean system in 6.3.2. The least solution of this system of equations yields exactly the states one-level reachable from each forest state q. Computing only a partial solution for those $r_P[q]$ and $r_Q[q]$ needed for computation of the value for $r_Q[q_0]$ involves exactly the reachable states and transitions of the automaton. [LH92] propose *local solvers* for computing such a solution. Indeed, our algorithm implements such a local solver. Because the only operation required for solving the system is set union, *differential local solvers* as in [FS98] can also be applied. The interested reader may consult that paper for efficient algorithms.

Example 6.6: Consider again the DLPA $A_{\vec{G}_3}$ constructed for grammar G_3 in Example 6.5. Running algorithm *ReachDlpa* yields the following reachable states:

$$\begin{array}{llll}
\text{Tree States:} & \text{Forest States:} & & \\
p_0 = \emptyset & q_0 = \{y0\} & q_4 = \{y7\} & q_8 \;= \{y8\} \\
p_1 = \{x_a\} & q_1 = \{y3\} & q_5 = \{y5\} & q_9 \;= \{y9\} \\
p_2 = \{x_b\} & q_2 = \{y2\} & q_6 = \{y4\} & q_{10} = \{y6\} \\
p_3 = \{x_1\} & q_3 = \emptyset & q_7 = \{y1\} & q_{11} = \{y10\}
\end{array}$$

Moreover, though there are theoretically 48 side-transitions for these 12 forest states and 4 tree states, only 25 of them can actually occur during a run of the automaton; similarly, only 12 up-transitions are reachable. Together with the 24 down-transitions this makes a total of 61 transitions. Compared to that, the reachable transitions in the DLFA B_{G_3} are 26 down-transitions, as many up-transitions and 52 side-transitions, which make a total of 104.

Note that all of the reachable states are at most singletons. Though this is not necessarily true for all grammars G, it is however characteristic for $A_{\vec{G}}$ that its states have only few elements (see also 9.2). □

Example 6.7: Consider again the language L_n from the proof of Theorem 6.5 and the grammar G_n mentioned above which describes L_n. Constructing $A_{\vec{G}_n}$ and running *ReachDlpa* yields that, after a renaming of states, it has the following reachable states and transitions:

$$\begin{array}{ll}
\text{Tree States:} & \mathcal{R}_P = \{\checkmark_0, \dots, \checkmark_n, ✗\} \\
\text{Forest States:} & \mathcal{R}_Q = \{0, \dots, n, \checkmark_0, \dots, \checkmark_n, ✗\}
\end{array}$$

$$\begin{array}{ll@{\qquad}ll}
\mathit{Down}_x\, i & = i{-}1,\ i{=}2,\dots,n & \mathit{Up}_a\, 0 & = \checkmark_1 \\
\mathit{Down}_a\, 1 & = 0 & \mathit{Up}_b\, 0 & = \checkmark_0 \\
\mathit{Down}_b\, 1 & = ✗ & \mathit{Up}_x\, i & = ✗,\ i{=}1,\dots,n \\
\mathit{Down}_x\, 0 & = 0 & \mathit{Up}_a\, \checkmark_0 & = \checkmark_1 \\
\mathit{Down}_x\, \checkmark_i & = ✗,\ i{=}0,\dots,n & \mathit{Up}_b\, \checkmark_0 & = \checkmark_0 \\
\mathit{Down}_x\, ✗ & = ✗ & \mathit{Up}_x\, \checkmark_i & = \checkmark_{i+1},\ i{=}1,\dots,n{-}1 \\
 & & \mathit{Up}_x\, ✗ & = ✗ \\
\mathit{Side}(i, \checkmark_i) & = \checkmark_i,\ i{=}0,\dots,n & & \\
\mathit{Side}(i, ✗) & = ✗,\ i{=}0,\dots,n & & \\
\mathit{Side}(0, \checkmark_1) & = ✗ & &
\end{array}$$

where $x \in \Sigma$. This automaton is very similar to A_n in the proof of Theorem 6.5. The difference is that $A_{\vec{G}_n}$ has a state $\checkmark_i$ for each $i \leqslant n$, whereas in A_n all of these states are unified into a single state $\checkmark$. □

6.4.2.2 Matching Structure with Right-to-Left Pushdown Automata

Analogously to $A_{\vec{G}}$ we can also define a right-to-left pushdown automaton for a given grammar: For $G = (X, r_0, R)$, let $\{r_1, \dots, r_l\} = \{r \neq r_0 \mid x \to a\langle r\rangle\}$. In contrast to the construction of $A_{\vec{G}}$, we use the reverse Berry-Sethi construction: Let $(Y_j, y_{0,j}, F_j, \delta_j) = \mathit{Berry}^{\leftarrow}(r_j)$ such that $Y_i \cap Y_j = \emptyset$ for $i \neq j$. With $Y = Y_0 \cup \ldots \cup Y_l$ and $\delta = \delta_0 \cup \ldots \cup \delta_l$, the DRPA $A_{\overleftarrow{G}}$ is defined as $(2^X, 2^Y, \{q_0\}, F, \mathit{Down}, \mathit{Up}, \mathit{Side})$ with

$$\begin{aligned}
q_0 &= \{y_{0,0}\}\\
F &= \{q \mid q \cap F_0 \neq \emptyset\}\\
\mathit{Down}_a\, q &= \{y_{0,j} \mid y \in q,\ (y,x,y_1) \in \delta,\ x \to a\langle r_j\rangle \text{ for some } x, y_1\}\\
\mathit{Up}_a\, q &= \{x \mid x \to a\langle r_j\rangle \text{ and } q \cap \mathcal{F}_j \neq \emptyset\}\\
\mathit{Side}(q,p) &= \{y_1 \mid y \in q, x \in p \text{ and } (y,x,y_1) \in \delta\}
\end{aligned}$$

Theorem 6.7: For each forest grammar G, $\mathcal{L}_{A_G^{\leftarrow}} = \mathcal{L}_G$

The proof is analogous to that of Theorem 6.6. □

6.5 Bibliographic Notes

Tree automata were introduced [Tha67] under the name *pseudo-automata* as a special class of finite algebras. The author shows that they accept the class of regular tree languages, and that each tree automaton has an equivalent deterministic automaton. For his slightly different definition of tree automata on ranked trees, [Bra69] obtains the same results. Tree automata were also used for implementing monadic second order logic by [TW68, Don70].

The automata of these early approaches have no explicit side-transitions; there is only a single set of states Q. The transition relation α determines a state for a node a in a single step from the word w of states assigned to its children. For non-ranked trees, α can thus have infinitely many transitions, with a regularity restriction for the set of words w yielding a state q under a symbol a. The idea of explicitly representing these regular sets by a separate state space with its own transition relation originally goes back to [Kro75], who used decision trees for efficient representation of large, n-ary transition relations. It was refined by [BMW91] who explicitly introduced a *horizontal automaton* for the side-transitions.

Pushdown tree automata have been studied in several variants. The first approach of implementing regular tree languages by pushdown automata is probably by [Tak75]: The author uses so called *P-tracers*, a special class of pushdown automata on words, for accepting the string representation of regular tree or forest languages. Her construction, however, is non-deterministic and she does not provide a method for making a P-tracer deterministic.

In [AU71] pushdown automata are used in the context of syntax-directed translation. These automata traverse derivation trees of a context-free grammars while writing to an output tape. Inspired by this work, [KS81] introduced two-way dag-walking automata, a special case of which are *two-way tree-walking automata*. Such an automaton can arbitrarily move up and down in the input tree while maintaining a pushdown. Changes to the pushdown are synchronized by the tree structure: The pushdown grows by one when moving to a child, and it shrinks by one when moving to the parent. The automaton has no explicit side-transitions: It can only move to the parent or the ith child of the current node. Side-transitions are simulated by moving to the parent first, and to the next child from there; the pushdown is used for remembering the number of the last child visited. The authors show that two-way tree-walking automata accept exactly the regular tree languages and that they can be made deterministic: Their construction generates a deterministic automaton whose traversing order corresponds exactly to that of our LPAs. However, this automaton is conceptually a bottom-up automaton: The down-transitions never generate information. A restriction of two-way tree-walking automata are the *pebble automata* of [EH99]. The size of the pushdown of these automata is finitely bounded; they can thus only store a finite number of *pebbles* on the pushdown. The authors show that the class of languages accepted by pebble automata is a subclass of the regular tree languages; they conjecture but can not prove that this is a proper inclusion.

A conceptually different kind of pushdown tree automata is introduced by [Gue83] and [SG85], in a top-down or a bottom-up variant, respectively. These automata traverse the tree from the root to the leaves or vice versa, visiting each node exactly once. Changes to the pushdown are not synchronized with moves in the tree. By contrast to the pushdown automata of [KS81], a transition always concerns all children of a node. Both variants of these automata are shown to accept the *context-free tree languages*. [Mor94] extends these automata by allowing them to change the moving direction and thus to revisit nodes. His two-way automata accept an even larger class of languages. Removing the pushdown from these automata leads to the class of two-way tree automata as defined by [BW98], which accept the regular tree languages.

A method similar to our LPAs but without using a pushdown is presented in [BKR96] for implementing monadic second order logic: In order to minimize the size of a bottom-up automaton, its run is preceded by a run of a top-down automaton, the so-called *guide*. The guide annotates the tree with some finite information which aids the bottom-up automaton in choosing transitions. This is similar to the behavior of LPAs: Each node is visited exactly two times, first when descending and a second time when ascending.

[MS98] use the string representation of ranked trees for the purpose of code generation. They use LR-parsing and exploit the property that even in the non-deterministic case moves on the stack are directed by the tree structure. Thus alternative runs can be tracked simultaneously. LR-parsing is mainly a bottom-up strategy: The pushdown is used for determining possible points for reductions. In our case these points are determined by the tree structure without the need to look at the pushdown. Instead, our algorithm uses the pushdown for gathering information about the part of the forest visited so far. It uses this information for selecting sensible expansions when descending to a node's children and is therefore more closely related to recursive-descent parsing.

Chapter 7

Locating Matches in Context

In this chapter we introduce contextual conditions and context grammars. We investigate how to implement context with forest automata and give an algorithm for locating matches in context by two consecutive runs of forest automata. Then we enhance the grammar formalism with conjunctions and negations and modify the matching algorithm to deal with these two concepts. Finally, we identify a class of context grammars which can be implemented by a single run of a forest automaton.

7.1 Contextual Conditions

In document processing, we are not always satisfied with verifying a structural condition for an input forest. In fact we often want to *locate* subdocuments in a specific *context*, that additionally have a structural property. For instance, in order to compile a table of contents for a book, it does not suffice to verify that the document contains section titles. In fact, we must collect all titles that are a child of a section. In this case the structural condition is being a title, and the context is that the parent node is a section. For this purpose we introduce context grammars in this section.

The contextual condition not only refers to the ancestors of a subtree: It may also impose conditions on the siblings of the subtree or of its ancestors. We can view a context as a forest from which a whole subtree is cut out and replaced by a hole "$\circ$". The situation is illustrated in Figure 7.1. We call the subtree we are interested in the *target*. The path from the target to the root[1] of the forest is its *upper context*. All nodes to the left of this path belong to the target's *left context*, whereas everything to its right is part of the *right context*. All three together make up the context of the target. The left context together with the upper context is the *left upper context*, and similarly the *right upper context* is the combination of the right and upper context.

A contextual condition is a regular set of contexts, i.e., of forests containing a hole. The hole could be represented as an additional symbol in the alphabet: In order to describe a contextual condition with a forest grammar G, we dedicate a variable $x_\circ$ to the hole and specify a single rule $x_\circ \to \circ$ for that variable. If we ensure that $\circ$ does not occur in any other rule of the grammar, then the

[1]Strictly speaking, a forest has no root. More precisely, we should therefore say: the root of the largest subtree containing the target.

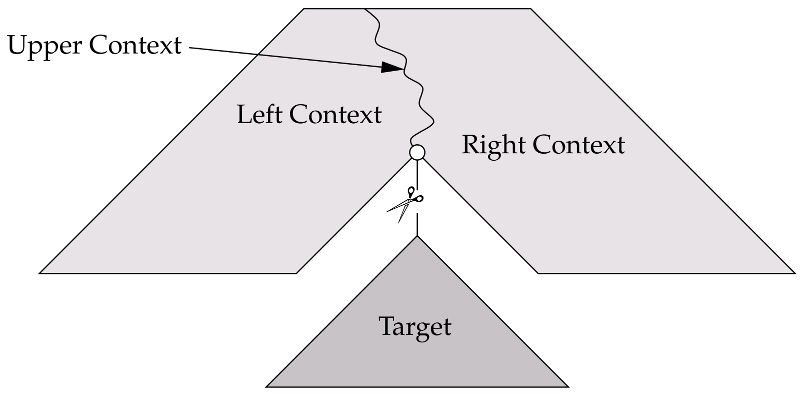


Figure 7.1: Context in pattern matching.

hole can only show up as a leaf in a forest in $\mathcal{L}_G$. The contextual condition is then fulfilled for all nodes labeled with a $\circ$. These are exactly the nodes for which the rule $x_\circ \to \circ$ must be applied in the definition of $[\![G]\!]$.

But our intention goes further: In addition to the context, we want to verify a structural condition for the target. We can describe this structural condition by replacing the rule $x_\circ \to \circ$ with one or more rules that do not use $\circ$. Still, the contextual condition is fulfilled for all subtrees that involve a rule for $x_\circ$, but now they must additionally be in $[\![G]\!]\ x_\circ$. Since the grammar has no more occurrences of symbol $\circ$, it is an ordinary forest grammar. The only additional component is the variable $x_\circ$, which describes the structure of the target; we therefore call it a *target variable.* Because we might want to specify different structural conditions for different contexts, we allow a set of target variables instead of only a single one.

A *context grammar* $C = (G, X_\circ)$ consists of a forest grammar $G = (X, r_0, R)$ and a set of *target variables* $X_\circ \subseteq X$. Let $f_0 \in \mathcal{L}_G$. For a regular expression r or a variable x, a path π is an *r-context in* f_0 (w.r.t. G) or an *x-context in* f_0, for short $f_0 \rightsquigarrow_\pi r$ and $f_0 \rightsquigarrow_\pi x$, if one of the following holds:

$f_0 \rightsquigarrow_\pi r$ iff $n = last_{f_0}(\pi),\ f_0[\pi 1] \ldots f_0[\pi n] \in [\![G]\!]\ r$ and either
- $\pi = \epsilon$ and $r = r_0$, or
- $\pi \neq \epsilon,\ f_0[\pi] = a\langle f\rangle,\ x \to a\langle r\rangle$, and $f_0 \rightsquigarrow_\pi x$ for some x.

$f_0 \rightsquigarrow_{\pi i} x$ iff $f_0 \rightsquigarrow_\pi r$ for some r, $n = last_{f_0}(\pi)$, there is a word $x_1 \ldots x_n \in [\![r]\!]_{\mathcal{R}}$ with $f_0[\pi j] \in [\![G]\!]\ x_j$ for $j = 1, \ldots, n$ and $x = x_i$ for some i.

If $f_0[\pi] = t$ and $f_0 \rightsquigarrow_\pi x$, then we also say that t matches x in f_0. The meaning of the context grammar C is defined as

$$[\![C]\!]\ f_0 = \{\pi \mid f_0 \rightsquigarrow_\pi x_\circ \text{ for some } x_\circ \in X_\circ\}$$

For all paths $\pi \in [\![C]\!]\ f_0$, we also say that π is a match of C in f_0.

Example 7.1: Consider again the grammar G_3 from Example 5.3. It describes all trees of depth 2 whose root symbol occurs exactly once at a leaf. In order to locate these leafs, we slightly reformulate G_3, introducing two target

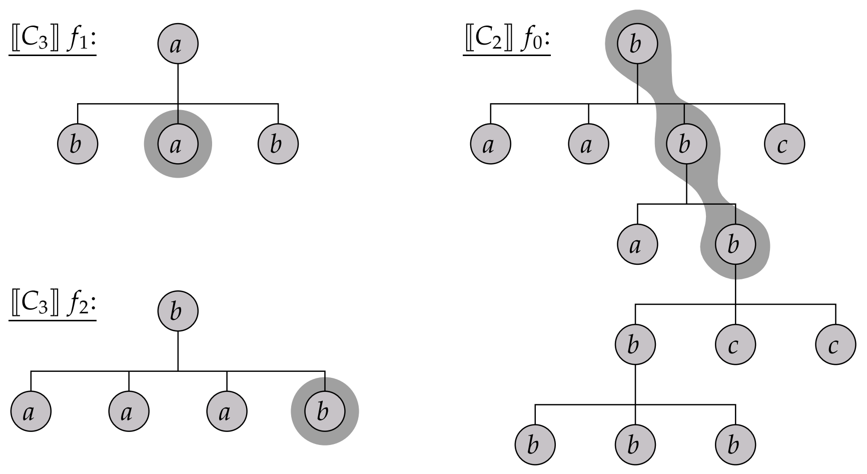


Figure 7.2: Matches of context grammars C_3 and C_2 from Examples 7.1 and 7.2.

variables x_2 and x_3, and obtain a context grammar $C_3 = (G_3', \{x_2, x_3\})$ where $G_3' = (\{x_a, x_b, x_1, x_2, x_3\}, x_1, R)$ with the following rules:

$$\begin{array}{lll} x_1 \longrightarrow a\langle x_b{}^* x_2 x_b{}^* \rangle & x_a \longrightarrow a\langle\rangle & x_2 \longrightarrow a\langle\rangle \\ x_1 \longrightarrow b\langle x_a{}^* x_3 x_a{}^* \rangle & x_b \longrightarrow b\langle\rangle & x_3 \longrightarrow b\langle\rangle \end{array}$$

Note that the rules for x_a and x_b are identical to those for x_2 and x_3; the latter, however, are target variables. Now consider the forest $f_1 = a\langle bab\rangle \in \mathcal{L}_{G_3'}$. To start with, $f_1 \rightsquigarrow_\epsilon x_1$ for the start expression x_1. Because $x_1 \in [\![x_1]\!]_\mathcal{R}$ and $a\langle bab\rangle \in [\![G_3']\!]\, x_1$, we also have $f_1 \rightsquigarrow_1 x_1$ for the variable x_1. Next, $x_1 \longrightarrow a\langle r_1\rangle$ with $r_1 = x_b{}^* x_2 x_b{}^*$, and because $bab \in [\![G_3']\!]\, r_1$, also $f_1 \rightsquigarrow_1 r_1$. Now $f_1[11] = b \in [\![G_3']\!]\, x_b$, $f_1[12] = a \in [\![G_3']\!]\, x_2$ and $f_1[13] = b \in [\![G_3']\!]\, x_b$. With $x_b x_2 x_b \in [\![r_1]\!]_\mathcal{R}$, we get that $f_1 \rightsquigarrow_{12} x_2$ and thus $12 \in [\![C_3]\!]\, f_1$. A closer look shows that this is the only match of C_3 in f_1.

For $f_2 = b\langle aaab\rangle$, we obtain in a similar way that $[\![C_3]\!]\, f_2 = \{14\}$. Note that for $f_3 = b\langle bab\rangle$, $[\![C_3]\!]\, f_3 = \emptyset$ because $f_3 \notin \mathcal{L}_{G_3'}$. The situation is illustrated for f_1 and f_2 in Figure 7.2. □

Example 7.2: Consider grammar G_2 from Example 5.2, which describes all forests that a have path π from the root to a node such that all nodes on π and all descendants of that node are labeled b, all nodes to the left of the path are labeled a and all nodes to its right are labeled c: $G_2 = (\{x_a, x_b, x_c, x_1\}, x_a{}^*(x_b \mid x_1)x_c{}^*, R)$ with:

$$\begin{array}{ll} x_a \longrightarrow a\langle x_a{}^*\rangle & x_c \longrightarrow c\langle x_c{}^*\rangle \\ x_b \longrightarrow b\langle x_b{}^*\rangle & x_1 \longrightarrow b\langle x_a{}^*(x_b \mid x_1)x_c{}^*\rangle \end{array}$$

Let $C_2 = (G_2, \{x_1\})$. Then the matches of C_2 are exactly the proper prefixes of that path π. For instance, for the forest $f_0 = b\langle aab\langle ab\langle b\langle bbb\rangle cc\rangle\rangle c\rangle$, $[\![C_2]\!]\, f_0 = \{1, 13, 132\}$. Figure 7.2 shows the situation. □

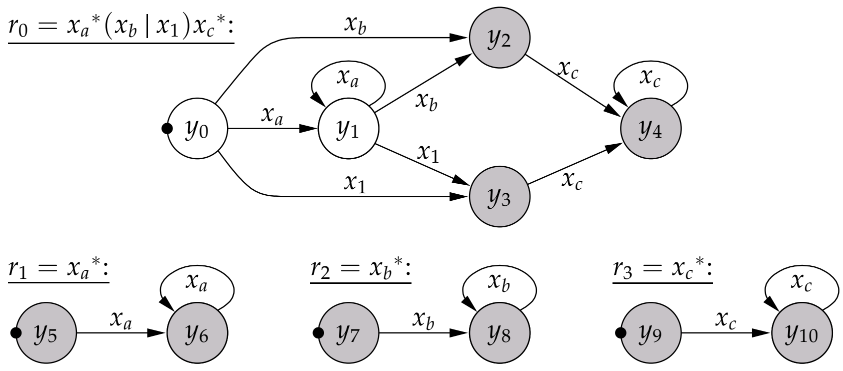


Figure 7.3: The Berry-Sethi construction for the regular expressions in grammar G_2 from Example 5.2.

7.2 Locating Matches of Context Grammars

A characteristic property of a left-to-right pushdown forest automaton is that, when entering a subtree, it has already visited the entire left upper context. Moreover, it can store some information about the left upper context on its pushdown and in its state. Pushdown automata are therefore a good candidate for implementing context grammars.

Indeed, if we construct for a context grammar $(G, X_\circ)$ the DLPA $A_{\vec{G}}$, this automaton can already identify all candidates for a match of the grammar. They can be recognized by means of the state with which $A_{\vec{G}}$ leaves each particular subtree. This is most easily explained with an example:

Consider the context grammar $C_2 = (G_2, \{x_1\})$ from Example 7.2. The NFAs obtained by the Berry-Sethi construction for the regular expressions of G_2 are shown in Figure 7.3. Note that y_3 is the only NFA state with an incoming x_1-transition, and that all transitions leading to y_3 are labeled x_1 (due to Proposition 4.3). Therefore, a forest state of $A_{\vec{G}_2}$ containing y_3 can be the result of a transition $Side(q, p)$ only if p contains x_1.

Let us have a look at the run of $A_{\vec{G}_2}$ on the forest f_1 in Figure 7.4 (a). The matches of C_2 are the paths 1 and 12, i.e., the two top-most b-nodes. Observe that the tree states computed for these subtrees contain x_1 which is the target variable of the grammar. Moreover, the forest states produced by the side-transitions at these subtrees contain y_3. This gives rise to the assumption that for all paths matching C_2, y_3 is in the forest state produced by the side-transition at that node. These nodes are underlaid in grey in the figure.

But not all of these paths are necessarily matches of the grammar: Since the automaton is only aware of the left upper context of each subtree, the right context remains to be verified. This is illustrated by case (b): In f_2, the only match of C_2 is the path 1. Even so, the forest state at path 111 contains y_3 and indicates a candidate for a match. But for this subtree the right context is not fulfilled: It has a right sibling labeled b. Filtering out those paths for which the right context does not match requires a run of a second automaton which proceeds from right to left.

Let us formalize this idea: To start with, it does not suffice to consider only the output state of an automaton. In fact, we have to regard the states involved

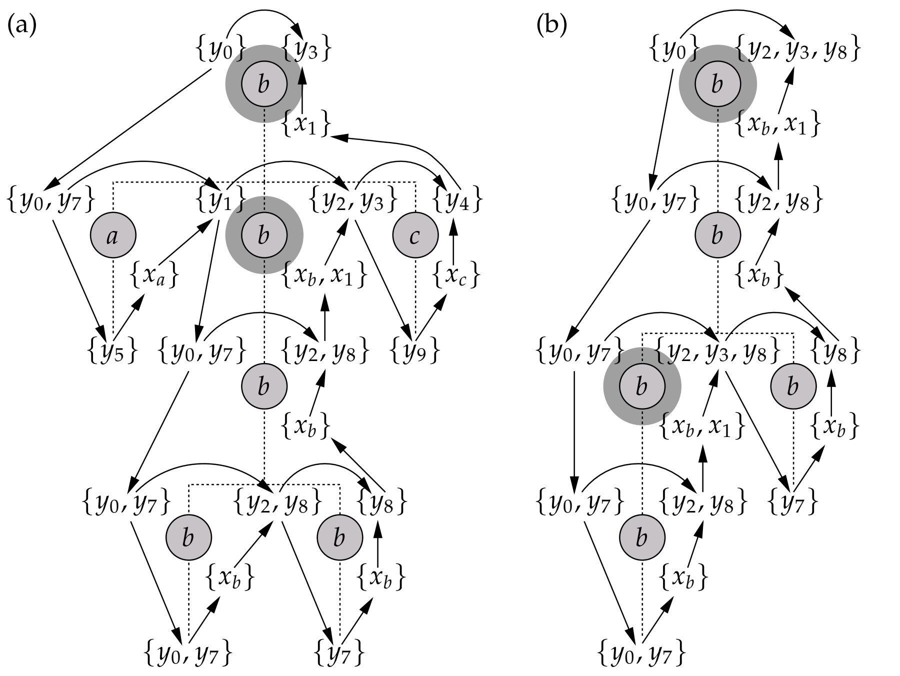


Figure 7.4: The runs of $A_{\vec{G}_2}$ on (a) the forest $f_1 = b\langle ab\langle b\langle bb\rangle\rangle c\rangle$ and on (b) the forest $f_2 = b\langle b\langle b\langle b\rangle b\rangle\rangle$.

in transitions at the individual nodes of the input forest f. Because the same subtree t can occur multiply in f, we must be able to distinguish the states of the automaton at these occurrences.

Let $A = (P, Q, \{q_0\}, F, \mathit{Down}, \mathit{Up}, \mathit{Side})$ be a DLPA. The *labeling* of a forest f by A is a mapping $\lambda : \Pi(f) \to Q \times P \times Q$, assigning each node in f the triple of states involved in the transitions at that node during a run of A. More precisely, λ fulfills the following:

- $\lambda(1) = (q, p, q')$ with $q = q_0$;
- If $f[\pi] = t$ and $\lambda(\pi) = (q, p, q')$ then $p = \delta_{\mathcal{T}}(q, t)$ and $q' = \mathit{Side}(q, p)$;
- If $f[\pi] = a\langle t_1 \dots t_n\rangle$, $n > 0$, $\lambda(\pi) = (q, p, q')$, $\lambda(\pi 1) = (q_1, p_1, q_1')$ and $\lambda(\pi n) = (q_n, p_n, q_n')$, then $q_1 = \mathit{Down}_a\, q$ and $q' = \mathit{Up}_a\, q_n'$;
- If $\lambda(\pi i) = (q_1, p_1, q_1')$ and $\lambda(\pi(i{+}1)) = (q_2, p_2, q_2')$, then $q_1' = q_2$.

Similarly, if A is a DRPA, then λ has the following properties:

- If $f = t_1 \dots t_n$, then $\lambda(n) = (q, p, q')$ with $q' = q_0$;
- If $f[\pi] = t$ and $\lambda(\pi) = (q, p, q')$ then $p = \delta_{\mathcal{T}}(q', t)$ and $q = \mathit{Side}(q', p)$;
- If $f[\pi] = a\langle t_1 \dots t_n\rangle$, $n > 0$, $\lambda(\pi) = (q, p, q')$, $\lambda(\pi 1) = (q_1, p_1, q_1')$ and $\lambda(\pi n) = (q_n, p_n, q_n')$, then $q_n' = \mathit{Down}_a\, q'$ and $q = \mathit{Up}_a\, q_1$;
- If $\lambda(\pi i) = (q_1, p_1, q_1')$ and $\lambda(\pi(i{+}1)) = (q_2, p_2, q_2')$, then $q_1' = q_2$.

Since A is deterministic in both cases, λ is uniquely determined for each f. We can easily derive the following observation:

- ✧ For $j \geqslant i$ and $\{\pi i, \pi j\} \subseteq \Pi(f)$, let $\lambda(\pi i) = (q_1, p_1, q_1')$, $\lambda(\pi j) = (q_2, p_2, q_2')$ and $f_{i,j} = f[\pi i] \dots f[\pi j]$. If A is a DLPA, then $q_2' = \delta_{\mathcal{F}}(q_1, f_{i,j})$, and if A is a DRPA, then $q_1 = \delta_{\mathcal{F}}(q_2', f_{i,j})$.

Now let $C = (G, X_\circ)$ be a context grammar, $A_{\vec{G}}$ be as in Section 6.4.2, $f_0 \in \mathcal{L}_G$, and $\vec{\lambda}$ be the labeling of f_0 by $A_{\vec{G}}$. The $A_{\vec{G}}$-*annotation* of f_0 is the forest $\vec{f_0}$ over $\Sigma \times P \times Q$ with each node of f_0 enhanced with the tree state produced by $\delta_{\mathcal{T}}$ and the forest state obtained by the side-transition at that node. Formally, $\Pi(\vec{f_0}) = \Pi(f_0)$ and if $\vec{\lambda}(\pi) = (q, p, q')$ and $sym(\pi[f_0]) = a$, then $sym(\pi[\vec{f_0}]) = (a, p, q')$. The DRPA $B_{\overleftarrow{G}}$ over $\Sigma \times P \times Q$ is now defined as $(P, Q, F_0, \emptyset, Down^{\leftarrow}, Up^{\leftarrow}, Side^{\leftarrow})$ where P, Q and F_0 are as in the definition of $A_{\vec{G}}$ and:

$$\begin{array}{ll}
Down^{\leftarrow}_{(a,p,\vec{q})}\, q & = \{y_2 \mid y \in q \cap \vec{q},\ (y_1, x, y) \in \delta,\ x \to a\langle r_j \rangle, \text{ and } y_2 \in F_j\} \\
Up^{\leftarrow}_{(a,p,\vec{q})}\, q & = p \\
Side^{\leftarrow}(q, p) & = \{y_1 \mid (y_1, x, y) \in \delta, y \in q, x \in p\}
\end{array}$$

$B_{\overleftarrow{G}}$ operates on the same sets of states as $A_{\vec{G}}$. However, when descending to the children of a node, it selects final NFA states instead of initial ones and performs the NFA transitions in the reverse direction while traversing the children. Moreover, when determining the regular expressions whose final NFA states are selected, $Down^{\leftarrow}$ does not consider all applicable NFA transitions (y_1, x, y). It takes into account only those y that were reached by $A_{\vec{G}}$ as well: Then $A_{\vec{G}}$ must also have performed a transition with x and the current subtree must be in $[\![G]\!]\, x$. Moreover, by performing the transitions of $A_{\vec{G}}$ in reverse order, $B_{\overleftarrow{G}}$ will certainly reach the initial state of y's NFA. In other words, the left part of the context has been verified for this level by $A_{\vec{G}}$. This assures that the current subtree matches x in f_0.

Note that $B_{\overleftarrow{G}}$ never actually computes an up-transition; it always uses the tree states assigned by $B_{\vec{G}}$. We can therefore view $B_{\overleftarrow{G}}$ as a *top-down forest automaton*. In the sequel, when arguing about $A_{\vec{G}}$ and $B_{\overleftarrow{G}}$ we will use the naming convention $\vec{q}$ for forest states of $A_{\vec{G}}$ in order to clearly distinguish them from the states of $B_{\overleftarrow{G}}$.

Theorem 7.1: For a context grammar $C = (G, X_\circ)$, let $A_{\vec{G}}$ and $B_{\overleftarrow{G}}$ be as above, $f_0 \in \mathcal{L}_G$, $\vec{\lambda}$ the labeling of f_0 by $A_{\vec{G}}$, $\vec{f_0}$ the $A_{\vec{G}}$-annotation of f_0, and $\overleftarrow{\lambda}$ be the labeling of $\vec{f_0}$ by $B_{\overleftarrow{G}}$. Then

$$\begin{array}{ll}
\pi \in [\![C]\!]\, f_0 \text{ iff } & \vec{\lambda}(\pi) = (\vec{q}_1, p, \vec{q}),\ \overleftarrow{\lambda}(\pi) = (q_1, p, q),\ y \in q \cap \vec{q} \\
& \text{and } (y_1, x_\circ, y) \in \delta \text{ for some } y, y_1 \in Y \text{ and } x_\circ \in X_\circ.
\end{array}$$

Informally, a subtree t matches a target variable $x_\circ$, if both the forest state with which $A_{\vec{G}}$ leaves t and the forest state with which $B_{\overleftarrow{G}}$ arrives at t contain the same NFA state y which has an incoming $x_\circ$-transition.

The proof is given in Appendix A.6. □

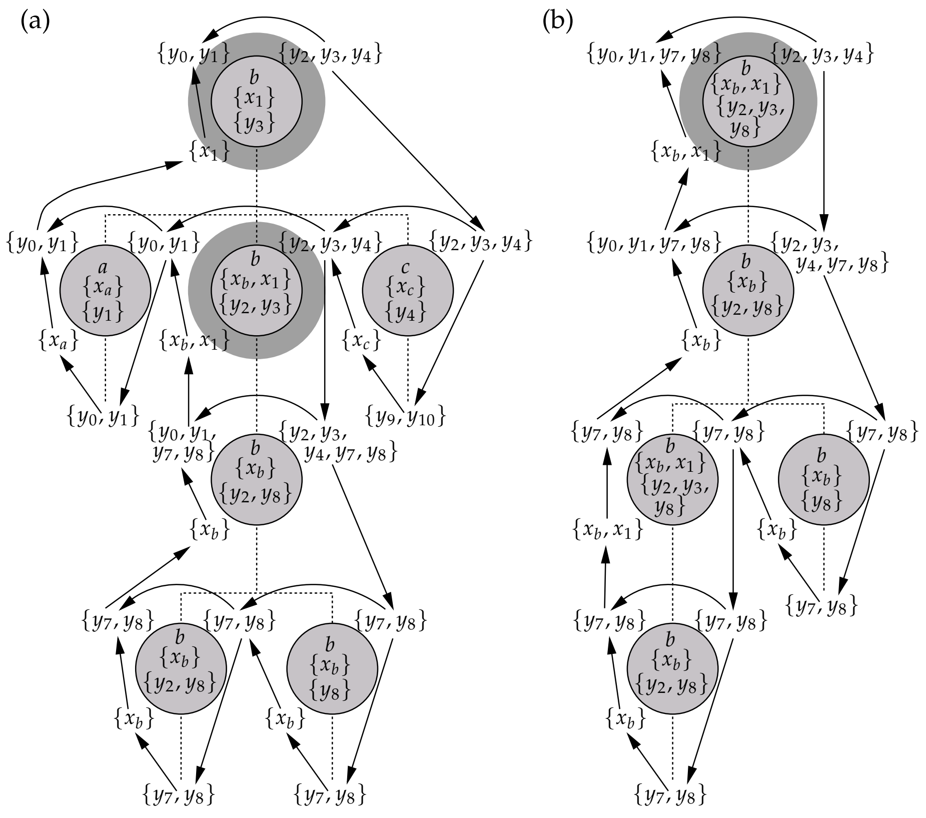


Figure 7.5: The runs of $B_{\overleftarrow{G}_2}$ on the forests $\vec{f}_1$ and $\vec{f}_2$, obtained from the runs of $A_{\vec{G}_2}$ in Figure 7.4.

Example 7.3: Let us illustrate this result for the context grammar C_2 from above. Figure 7.5 shows the runs of $B_{\overleftarrow{G}_2}$ on the annotated forests $\vec{f}_1$ and $\vec{f}_2$, according to Figure 7.4. For $\vec{f}_1$, $B_{\overleftarrow{G}_2}$ finds exactly the nodes that were already marked as candidates for a match by $A_{\vec{G}_2}$. On the other hand, $A_{\vec{G}_2}$ found path 111 as a candidate for a match in f_2. In $\vec{f}_2$, however, $B_{\overleftarrow{G}_2}$ arrives with a forest state a that node which does not contain y_3. 111 is therefore no match of C_2 in f_2. □

7.3 Extending the Grammar Formalism

Forest grammars are a formalism for specifying regular forest languages. The class of these languages is closed under set union, intersection and complement as we showed in Section 5.2. The latter two operations have no counterpart in the syntax of forest grammars: Only set union can be expressed by giving multiple rules for a variable.

In document processing, it is often desired to express intersection and complement: On the one hand, we might want to specify that a subtree may not fulfill a structural condition. For instance, it is sensible to disallow floating figures within other figures; or we might want to limit the nesting depth of lists to a fixed number – e.g., in LaTeX this limit is 4. In order to denote the complement of regular forest languages we must therefore extend the grammar formalism with *negations*.

On the other hand, one might want to express that a forest must fulfill several structural conditions at a time. This can only be specified intuitively if we extend forest grammars by *conjunctions*. Therefore we will now redefine the syntax and meaning of grammars in order to offer these operations.

7.3.1 Extended Forest Grammars

A *forest expression* e over X has the form $\sigma_1 r_1 \sqcap \ldots \sqcap \sigma_n r_n$, where $\sigma_i \in \{+, \neg\}$, and r_i is a regular expression over X for all i. σ_i is called the *sign* of r_i in e. If $\sigma_i = +$ then we say that r_j occurs *positively* in e; we often omit the $+$ for brevity. If $\sigma_i = \neg$ then r_j is said to occur *negated* in e.

An *extended forest grammar* (EFG) over Σ is a tuple $G = (X, E_0, R)$ where X is a set of *variables*, E_0 is a set of forest expressions over X called *start expressions*, and R is a finite set of rules of the form $x \rightarrow a\langle e\rangle$ with $x \in X$, $a \in \Sigma$ and e a forest expression over X. In the sequel we will often omit the adjective extended when speaking about EFGs; for clarity, we will use the term *simple* forest grammar when meaning the non-extended form.

Note that unlike simple forest grammars, an EFG has a set of start expressions: While, an alternative of regular expressions can be denoted as a single regular expression using $|$, this is not possible for forest expressions. We therefore allow specification of multiple start expressions.

The *meaning* $[\![G]\!]$ of an extended forest grammar $G = (X, E_0, R)$ assigns sets of trees to the variables in X and sets of forests to regular expressions and forest expressions:

$$\begin{array}{rcl} f \in [\![G]\!]\, e & \text{iff} & f \in [\![G]\!]\, r \text{ for all } r \text{ with } e = \ldots \sqcap +r \sqcap \ldots, \\ & & \text{and } f \notin [\![G]\!]\, r \text{ for all } r \text{ with } e = \ldots \sqcap \neg r \sqcap \ldots; \\ t_1 \ldots t_n \in [\![G]\!]\, r & \text{iff} & \text{there is } x_1 \ldots x_n \in [\![r]\!]_{\mathcal{R}}, \text{ with } t_i \in [\![G]\!]\, x_i \text{ for all } i; \\ a\langle f\rangle \in [\![G]\!]\, x & \text{iff} & f \in [\![G]\!]\, e \text{ for some } e \text{ with } x \rightarrow a\langle e\rangle. \end{array}$$

The *language* of G is $\mathcal{L}_G = \bigcup_{e_0 \in E_0} [\![G]\!]\, e_0$.

Example 7.4: For $\Sigma = \{a, b, c\}$, suppose we want to describe the language of all forests containing an a *and* a b. This is straight-forward with the extended forest grammar $G_4 = (\{x_\top, x_a, x_b\}, \{e_0\}, R)$ with $e_0 = \_\, x_a \,\_ \;\sqcap\; \_\, x_b \,\_$ and the following rules (we use $\_$ as an abbreviation for $x_\top{}^*$):

$$\begin{array}{lll} x_\top \rightarrow a\langle \_ \rangle & x_a \rightarrow a\langle \_ \rangle & x_b \rightarrow a\langle \_\, x_b \,\_ \rangle \\ x_\top \rightarrow b\langle \_ \rangle & x_a \rightarrow b\langle \_\, x_a \,\_ \rangle & x_b \rightarrow b\langle \_ \rangle \\ x_\top \rightarrow c\langle \_ \rangle & x_a \rightarrow c\langle \_\, x_a \,\_ \rangle & x_b \rightarrow c\langle \_\, x_b \,\_ \rangle \end{array}$$

Note that $x_\top$ describes arbitrary trees from $\mathcal{T}_\Sigma$. In the sequel, we will assume that each grammar implicitly has this variable with rules $x_\top \rightarrow a\langle \_ \rangle$ for all $a \in \Sigma$, and we abbreviate $x_\top{}^*$ to $\_$.

Variables x_a and x_b describe trees that contain at least one a or b respectively. The $\sqcap$-notation in e_0 expresses concisely and intuitively our intention.

Describing this language without conjunctions is fairly more complicated: If a forest $f = t_1 \ldots t_n$ contains an a and b, then a must occur in t_i for some i and b must occur in t_j for some j. Now there are three possibilities: $i{<}j$, $i{>}j$, or $i{=}j$. Therefore the start expression must have a form similar to $\_ x_a \_ x_b \_ \ \mid \ \_ x_b \_ x_a \_ \ \mid \ \_ x_{ab} \_$, where x_a and x_b are as above and x_{ab} describes all trees containing a and b. This phenomenon is scalable: For expressing that a forest contains n different symbols, there are exponentially many cases in the above case distinction. The grammar then becomes large and unintelligible. □

In document processing, such situations occur frequently: E.g., a user of a document database might search for all documents whose abstract contains the words "document", "automata" and "pattern". Conjunctions are therefore a very sensible extension to the grammar formalism.

Example 7.5: For $\Sigma = \{a, b, c\}$, consider the extended grammar $G_5 = (\{x_\top, x_a, x_b, x_c\}, \{\_ x_a \_\}, R)$ with the following rules (we numbered the rules because that will be needed for the next example):

$$\begin{array}{ll}
R_1 = x_\top \to a\langle \_ \rangle & R_4 = x_a \to a\langle \_ x_a \_ \rangle \\
R_2 = x_\top \to b\langle \_ \rangle & R_5 = x_a \to a\langle \_ x_b \_ \sqcap \neg \ \_ x_c \_ \rangle \\
R_3 = x_\top \to c\langle \_ \rangle & R_6 = x_b \to b \\
 & R_7 = x_c \to c\langle x_b \rangle
\end{array}$$

Variable x_c describes the tree $c\langle b \rangle$. x_a represents all trees that have a subtree labeled a, which has a child b but does not have a child $c\langle b \rangle$; all ancestors of the a-node must also be labeled a. $\mathcal{L}_{G_5}$ is the set of all forests one of whose trees is according to x_a. For instance, $f_1 = a\langle a\langle bc \rangle\rangle \in \mathcal{L}_{G_5}$, but $f_2 = a\langle bc\langle b \rangle\rangle \notin \mathcal{L}_{G_5}$. □

7.3.2 Implementing Extended Forest Grammars

We will now adapt the definition of $A_{\vec{G}}$ to extended forest grammars. Note that simple forest grammars are a special case of EFGs, so the new construction will also work for them. However, the new construction is not an extension of the old one: It uses a different set of tree states. This is not strictly necessary for the purpose of matching structure, but we intend to use the same construction for matching context. For that purpose, the tree states must not only indicate whether a subtree matches a variable: It must also denominate the individual rules fulfilled for that variable (cf. Example 7.8). Therefore the tree states of the new $A_{\vec{G}}$ are sets of rule numbers rather than tree variables.

Let $G = (X, E_0, R)$ be a an EFG with $R = \{R_1, \ldots, R_h\}$, such that each rule R_k has the form $x^k \to a_k\langle e_k \rangle$ for $k = 1, \ldots, h$. Furthermore, let $H = \{1, \ldots, h\}$ and $\{r_1, \ldots, r_l\}$ be the set of regular expressions occurring in G. For each $j = 1, \ldots, l$, let $(Y_j, y_{0,j}, F_j, \delta_j) = Berry(r_j)$ such that $Y_i \cap Y_j = \emptyset$ for $i \neq j$, and define $Y = Y_1 \cup \ldots \cup Y_l$ and $\delta = \delta_1 \cup \ldots \cup \delta_l$. Then $A_{\vec{G}}$ is defined as $(2^H, 2^Y, q_0, F, Down, Up, Side)$ with:

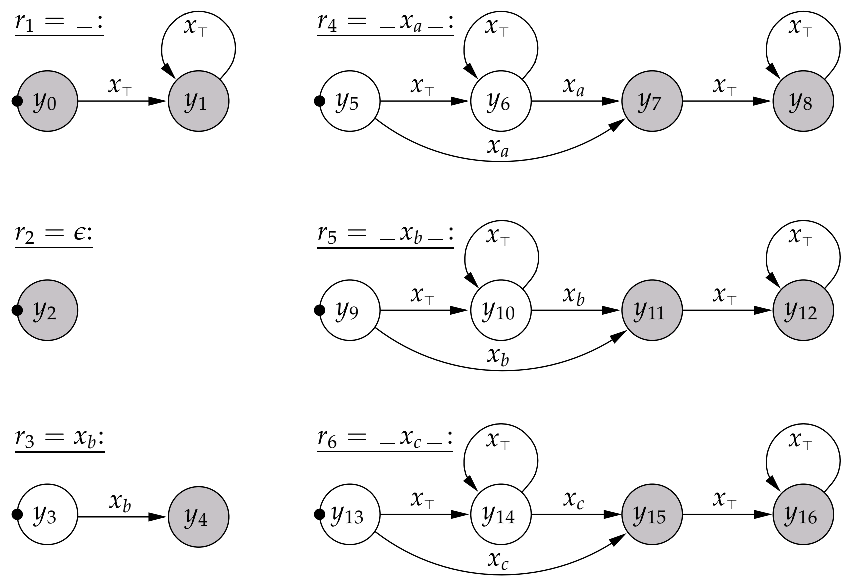


Figure 7.6: The Berry-Sethi construction for the regular expressions in grammar G_5 from Example 7.5.

$$
\begin{aligned}
q_0 &= \{y_{0,j} \mid \ldots \sqcap \sigma r_j \sqcap \ldots \in E_0\} \\
F &= \{q \mid \text{there is an } e_0 \in E_0 \text{ such that} \\
&\qquad q \cap F_j \neq \emptyset \text{ for all } j \text{ with } e_0 = \ldots \sqcap +r_j \sqcap \ldots, \\
&\qquad \text{and } q \cap F_j = \emptyset \text{ for all } j \text{ with } e_0 = \ldots \sqcap \neg r_j \sqcap \ldots\} \\
\mathit{Down}_a\, q &= \{y_{0,j} \mid y \in q,\ (y, x, y_1) \in \delta,\ x \to a\langle \ldots \sqcap \sigma r_j \sqcap \ldots \rangle\} \\
\mathit{Up}_a\, q &= \{k \mid a = a_k,\ q \cap F_j \neq \emptyset \text{ for all } j \text{ with } e_k = \ldots \sqcap +r_j \sqcap \ldots, \\
&\qquad \text{and } q \cap F_j = \emptyset \text{ for all } j \text{ with } e_k = \ldots \sqcap \neg r_j \sqcap \ldots\} \\
\mathit{Side}(q, p) &= \{y_1 \mid y \in q,\ k \in p, \text{ and } (y, x^k, y_1) \in \delta\}
\end{aligned}
$$

Theorem 7.2: For each extended forest grammar G, $\mathcal{L}_{A_{\vec{G}}} = \mathcal{L}_G$

The proof is by structural induction and given in Appendix A.7. An immediate consequence is that the language of an EFG is regular. □

Example 7.6: Let us consider again grammar G_5 from Example 7.5. Figure 7.6 shows the NFAs produced by the Berry-Sethi construction. The rules (and thus the forest expressions on their right-hand sides) are already numbered in Example 7.5: e.g., $e_4 = \_ x_a \_$. Figure 7.7 shows the runs of $A_{\vec{G}_5}$ on the forests $f_1 = a\langle a\langle bc\rangle\rangle \in \mathcal{L}_{G_5}$ and $f_2 = a\langle bc\langle b\rangle\rangle \notin \mathcal{L}_{G_5}$. $A_{\vec{G}_5}$ accepts f_1, but not f_2. The reason is that the top-most tree state $\{1\}$ in the run on f_2 does not contain the number of a rule for x_a. This tree state is obtained through an up-transition for a and forest state $\{y_1, y_6, y_{10}, y_{12}, y_{14}, y_{15}\}$. This set contains a final state both for r_5 and r_6. In order to fulfill rule R_5, however, it must not contain any final state for r_6. □

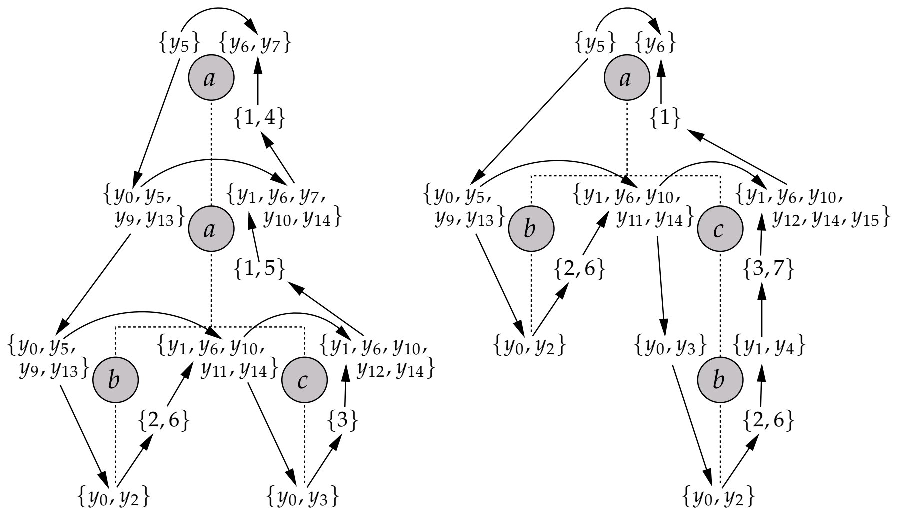


Figure 7.7: Two runs of $A_{\vec{G}_5}$ from Example 7.6 on the forests $f_1 = a\langle a\langle bc\rangle\rangle$ and $f_2 = a\langle bc\langle b\rangle\rangle$.

7.3.3 Extended Context Grammars

We can also use extended forest grammars for describing contextual conditions: An extended context grammar (ECG) $C = (G, X_0)$ consists of an EFG $G = (X, E_0, R)$ and a set of *target variables* $X_\circ \subseteq X$. Let us now upgrade the notion of $\rightsquigarrow_\pi$ to extended context grammars. For a path π and a forest expression e, a regular expression r, or a variable x, we define:

$f_0 \rightsquigarrow_\pi e$ iff $n = last_{f_0}(\pi)$, $f_0[\pi 1]\ldots f_0[\pi n] \in [\![G]\!]\, e$ and either

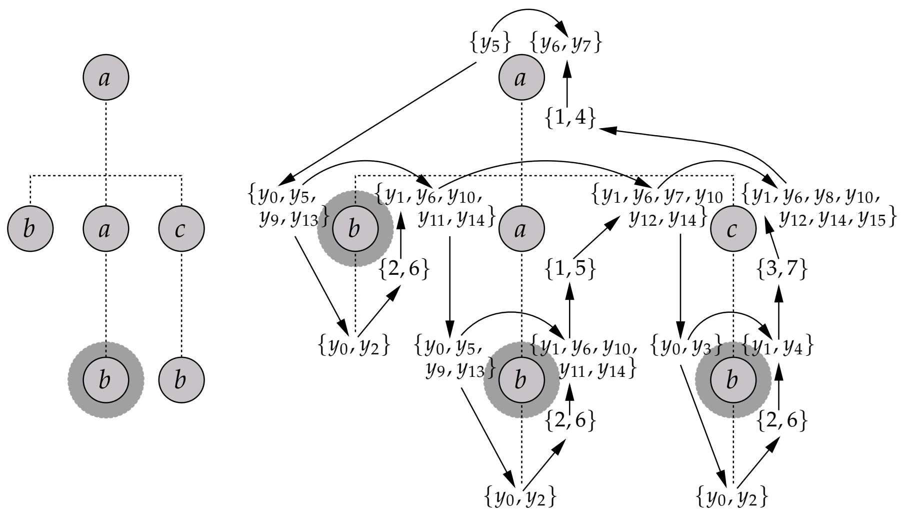
- ✧ $\pi = \epsilon$ and $e \in E_0$, or
- ✧ $\pi \neq \epsilon$, $f_0[\pi] = a\langle f\rangle$, $x \to a\langle e\rangle$, and $f_0 \rightsquigarrow_\pi x$ for some x.

$f_0 \rightsquigarrow_{\pi i} x$ iff $f_0 \rightsquigarrow_\pi r$ for some r and there is $x_1 \ldots x_n \in [\![r]\!]_\mathcal{R}$ with $n = last_{f_0}(\pi)$, $x = x_i$ and $f_0[\pi j] \in [\![G]\!]\, x_j$ for $j = 1, \ldots, n$.

$f_0 \rightsquigarrow_\pi r$ iff $f_0 \rightsquigarrow_\pi e$ for some $e = \ldots \sqcap + r \sqcap \ldots$.

Example 7.7: Consider again grammar G_5 from Example 7.5. It describes the forests containing an a that has a child b but no child $c\langle b\rangle$. In order to locate these b children, we define $C_5 = (G_5, \{x_b\})$. Figure 7.8 shows the single match of C_5 in $f_3 = a\langle ba\langle b\rangle c\langle b\rangle\rangle$. Note that only the second occurrence of b is indeed a match of C_5: The first b-node is not a match because it has a sibling $c\langle b\rangle$, and the third b-node does not match because it is a child of a c-node, and the forest expression $\_\, x_c\, \_$ occurs negated in rule R_5.

Figure 7.8 also shows the run of $A_{\vec{G}_5}$ on f_3. Similar to Section 7.2, a subtree is a candidate for a match of C_5 if the automaton leaves that subtree with a forest state containing an NFA state with an incoming x_b-transition. According to Figure 7.6, these NFA states are y_4 and y_{11}. Thus $A_{\vec{G}_5}$ identifies the paths 11, 121 and 131 as candidates for a match. □



a
b
a
c
b
b
$\{y_5\}$
$\{y_6, y_7\}$
$\{1,4\}$
$\{y_0, y_5, y_9, y_{13}\}$
$\{y_1, y_6, y_{10}, y_{11}, y_{14}\}$
$\{y_1, y_6, y_7, y_{10}, y_{12}, y_{14}\}$
$\{y_1, y_6, y_8, y_{10}, y_{12}, y_{14}, y_{15}\}$
$\{2,6\}$
$\{1,5\}$
$\{3,7\}$
$\{y_0, y_2\}$
$\{y_0, y_3\}$
$\{y_1, y_4\}$

Figure 7.8: The match of C_5 from Example 7.7 in $f_3 = a\langle ba\langle b\rangle c\langle b\rangle\rangle$, and the run of $A_{\vec{G}_5}$ on f_3.

7.3.4 Locating Matches of Extended Forest Grammars

In order to locate all matches of an extended context grammar in a forest f_0, we adapt the definition of $B_{\overleftarrow{G}}$: Let $A_{\vec{G}}$ be as above and $\vec{\lambda}$ be the labeling of f_0 by $A_{\vec{G}}$. Moreover, let $\vec{f_0}$ be the $A_{\vec{G}}$-annotation of f_0, and q_F the output state [2] of $A_{\vec{G}}$ for f_0. The DRPA $B_{\overleftarrow{G}}$ over $\Sigma \times P \times Q$ is now defined as $(P, Q, \{\overleftarrow{q}_0\}, \emptyset, Down^{\leftarrow}, Up^{\leftarrow}, Side^{\leftarrow})$ with:

$$\begin{aligned}
\overleftarrow{q}_0 &= \{y \mid y \in F_j \text{ for some } e \in E_0 \text{ with } e = \ldots \sqcap +r_j \sqcap \ldots, \text{ and} \\
&\qquad \text{for all } i \text{ with } e = \ldots \sqcap \sigma r_i \sqcap \ldots, F_i \cap q_F \neq \emptyset \text{ iff } \sigma{=}{+}\} \\
Down^{\leftarrow}_{(a,p,\vec{q})}\, q &= \{y_2 \mid y \in q \cap \vec{q},\ k \in p,\ (y_1, x, y) \in \delta \text{ for some } y_1, \\
&\qquad R_k = x \to a\langle \ldots \sqcap +r_j \sqcap \ldots \rangle \text{ and } y_2 \in F_j\} \\
Up^{\leftarrow}_{(a,p,\vec{q})}\, q &= p \\
Side^{\leftarrow}(q, p) &= \{y_1 \mid (y_1, x^k, y) \in \delta, y \in q, k \in p\}
\end{aligned}$$

In contrast to the definition of $B_{\overleftarrow{G}}$ for simple context grammars, $Down^{\leftarrow}$ selects only the final states for those regular expressions which occur positively in a forest expression e. Additionally, the tree state p produced by $A_{\vec{G}}$ must indicate that e was fulfilled by the children of the current node. This is the reason why tree states must distinguish the individual rules fulfilled for a variable.

Note that $\overleftarrow{q}_0$ is defined dependent on the output state q_F of $A_{\vec{G}}$. Strictly speaking, $B_{\overleftarrow{G}}$ can thus be different for all input forests f_0. This prevents analysis of $B_{\overleftarrow{G}}$ in a preprocessing stage: The reachable states of $B_{\overleftarrow{G}}$ can not be determined until after the run of $A_{\vec{G}}$. Due to the immensely large alphabet of $B_{\overleftarrow{G}}$, this is virtually impossible anyway. It is therefore sensible to compute the states and transitions on demand, during the run of $B_{\overleftarrow{G}}$ (see Chapter 9 for details of such an implementation).

Theorem 7.1 now carries over to extended context grammars:

[2]The output state of $A_{\vec{G}}$ for f_0 is $\delta_{\mathcal{F}}(q_0, f_0)$.

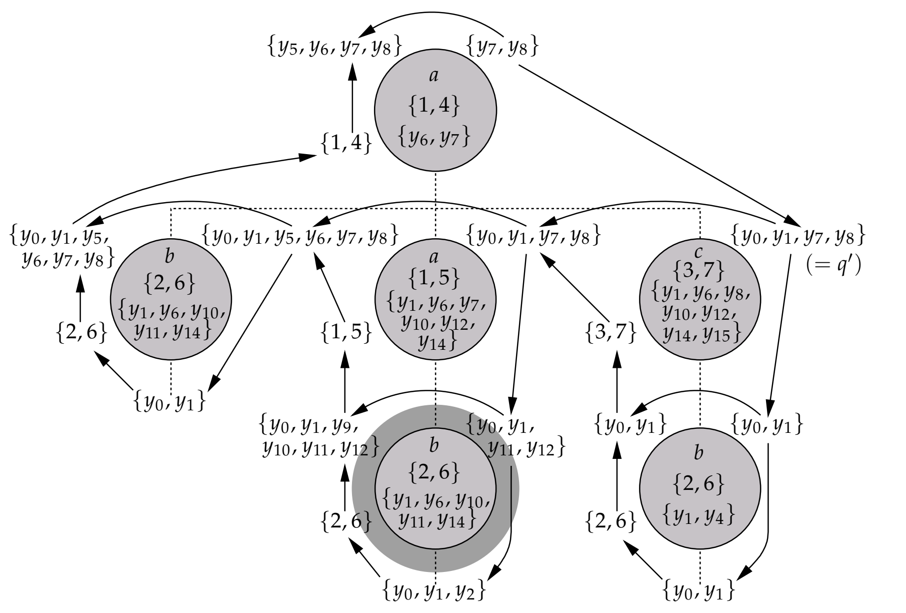


Figure 7.9: The run of $B_{\overleftarrow{G_5}}$ on the forest $\vec{f_3}$, obtained from the run of $A_{\vec{G_5}}$ on f_3 in Figure 7.8.

Theorem 7.3: For an extended context grammar $C = (G, X_\circ)$, let $A_{\vec{G}}$ and $B_{\overleftarrow{G}}$ be as above, $\vec{\lambda}$ the labeling of f_0 by $A_{\vec{G}}$, $\vec{f_0}$ the $A_{\vec{G}}$-annotation of f_0, and $\overleftarrow{\lambda}$ be the labeling of $\vec{f_0}$ by $B_{\overleftarrow{G}}$. Then

$$\pi \in [\![C]\!]\, f_0 \text{ iff } \vec{\lambda}(\pi) = (\vec{q}_1, p, \vec{q}),\ \overleftarrow{\lambda}(\pi) = (q_1, p, q),\ y \in q \cap \vec{q}$$
$$\text{and } (y_1, x_\circ, y) \in \delta \text{ for some } y, y_1 \in Y \text{ and } x_\circ \in X_\circ.$$

The proof is given in Appendix A.8. □

Example 7.8: Consider again the context grammar C_5 and the run of $A_{\vec{G_5}}$ on f_3 in Example 7.7. The run of $B_{\overleftarrow{G_5}}$ on $\vec{f_3}$ is illustrated in Figure 7.9. The matches of C_5 are the paths in $\vec{f_3}$ at which the automaton arrives with a forest state containing an NFA state with an x_b-transition, that is also in the (annotated) forest state with which $A_{\vec{G_5}}$ left that node. Path 121 is thus correctly indicated as the only match of C_5 in f_3, because y_{11} is in both forest states there.

Note that, on the one hand, the b node at path 131 is not recognized as a match though identified as a candidate by $A_{\vec{G_5}}$: It is annotated with a state containing y_4 which has an incoming x_b-transition. However, in order to have y_4 in the $B_{\overleftarrow{G_5}}$ state at that node, it is necessary by definition of *Down*$^{\leftarrow}$ that $\overleftarrow{\lambda}(13) = (q, p, q')$ with $y_{15} \in q'$ or $y_{16} \in q'$. This is not the case, because $q' = Down^{\leftarrow}_{(a,\{1,4\},\{y_6,y_7\})}\ \{y_7, y_8\}$. This *Down*$^{\leftarrow}$-transition ignores NFA states y_{15} and y_{16} because they are the final states of r_6 which occurs negated in rule R_5.

On the other hand, the first b in f_3, located at path 11, is not identified as a match of C_5 either. In order to be a match, $B_{\overleftarrow{G_5}}$ has to arrive at that node with a state containing y_{11}. This is impossible because the forest state q' with which $B_{\overleftarrow{G_5}}$ enters this level does not contain a state from Y_5; thus y_{11} can not be obtained through side-transitions from q'. The reason why q' does not contain any states from Y_5 is that 5 is not in the tree state at the root of the tree. Therefore rule R_5 is ignored in the first $\mathit{Down}^{\leftarrow}$-transition, and the final states for r_5 are not taken into account. The $\mathit{Down}^{\leftarrow}$-transition only considers those rules for a variable which are structurally matched by the forest to be entered. If we had used, as in the construction in Section 7.2, sets of variables as tree states, then we could not make this distinction. □

7.3.5 Locating Matches in Document Order

As shown in the previous section, the matches of an extended context grammar can be located with runs of two forest automata, namely $A_{\vec{G}}$ and $B_{\overleftarrow{G}}$. The latter is conceptually a top-down automaton: It traverses the forest in right-to-left, depth-first order. A match can be recognized by the forest state with which $B_{\overleftarrow{G}}$ arrives at the concerning subtree, i.e., before descending into that subtree. All matches can thus be reported in the same order in which the automaton traverses the forest.

In document processing, however, we are usually interested in a different order: *document order*, which is depth-first left-to-right. In order to locate the matches in this order, we have to switch the directions of the two automata: The first traversal of the forest must be from right to left, and the second run from left to right.

We therefore define automata $A_{\overleftarrow{G}}$ and $B_{\vec{G}}$ analogously to $A_{\vec{G}}$ and $B_{\overleftarrow{G}}$, with the difference that the reverse Berry-Sethi construction is employed for computing the NFAs. Theorems 7.2 and 7.3 can be reformulated for $A_{\overleftarrow{G}}$ and $B_{\vec{G}}$; the proofs are completely analogous.

7.4 Matching Context in a Single Pass

With the automata $A_{\vec{G}}$ and $B_{\overleftarrow{G}}$ from the previous section, we can locate all matches of a context grammar in a forest in two traversals. The first traversal already identifies all candidates for matches of the grammar; the task of the second traversal is to discard those candidates that do not match. In many cases, however, all the candidates found by $A_{\vec{G}}$ are indeed matches of the grammar; the second traversal is then superfluous.

We will now define a class of grammars for which the second traversal can be safely omitted. Then the matching can be done in a single pass which is a valuable advantage in document processing, because it allows to search a document without constructing a physical copy of it in memory (cf. 1.2.3).

7.4.1 Right-Ignoring Regular Expressions

In order to define the requirements a grammar must fulfill for being implemented by a single pass, let us have another look at the definition of $f_0 \rightsquigarrow_{\pi} x$:

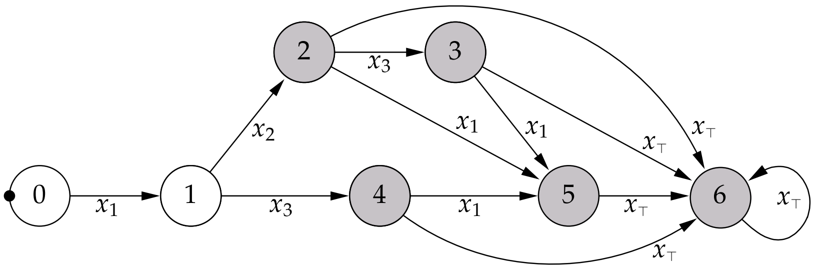
$$\begin{array}{ll} f_0 \rightsquigarrow_{\pi i} x \ \text{ iff } & f_0 \rightsquigarrow_{\pi} r \text{ for some } r \text{ and there is } x_1 \ldots x_n \in [\![r]\!]_{\mathcal{R}} \text{ with} \\ & n = \mathit{last}_{f_0}(\pi),\ x = x_i \text{ and } f_0[\pi j] \in [\![G]\!]\, x_j \text{ for } j = 1, \ldots, n. \end{array}$$

For $j \leqslant i$, the left-to-right automaton $A_{\vec{G}}$ can verify that $f_0[\pi j] \in [\![G]\!]\, x_j$; but the subtrees at πj for $j > i$ must be checked by $B_{\overleftarrow{G}}$. In order to skip the run of

$B_G^{\leftarrow}$, it must be sure that $f_0[\pi j] \in [\![G]\!]\, x_j$ for $j > i$. This is guaranteed if $x_j = x_\top$ for $j > i$ (recall from 7.3.1 that $x_\top$ describes arbitrary trees).

Now let r be a regular expression, $(Q, q_0, F, \delta) = \mathit{Berry}(r)$, and $Q_x = \{q \in Q \mid (q_1, x, q) \in \delta \text{ for some } q_1\}$. Then r is *right-ignoring* w.r.t. x, if there is a $Q_\top \subseteq F$ such that $Q_x \subseteq Q_\top$ and for all $q \in Q_\top$, there is a $q_1 \in Q_\top$ with $(q, x_\top, q_1) \in \delta$. In other words, if we can reach a state in Q through an x-transition, then this state is final, and we can end up in a final state after an arbitrary number of $x_\top$-transitions.

Example 7.9: The regular expression $\_\,x\,\_$ is right-ignoring w.r.t. x. As a more complex example, consider the regular expression $r = x_1(x_2x_3? \mid x_3)x_1?\_$. Here is the Berry-Sethi automaton for r:



$$\begin{tikzcd}
 & & 2 \arrow[r, "x_3"] \arrow[drr, "x_1"] \arrow[drrr, bend left, "x_\top"] & 3 \arrow[dr, "x_1"] \arrow[drr, "x_\top"] & & \\
0 \arrow[r, "x_1"] & 1 \arrow[ur, "x_2"] \arrow[rr, "x_3"] & & 4 \arrow[r, "x_1"] \arrow[rr, bend right, "x_\top"] & 5 \arrow[r, "x_\top"] & 6 \arrow[loop right, "x_\top"]
\end{tikzcd}$$

Now $Q_{x_2} = \{2\}$, and with $Q_\top = \{2, 6\}$, r is right-ignoring w.r.t. x_2. It is also right-ignoring w.r.t. x_3, because $Q_{x_3} = \{3, 4\}$, and we can choose $Q_\top = \{3, 4, 6\}$. However, r is not right-ignoring w.r.t. x_1, because $Q_{x_1} = \{1, 5\}$ and 1 is not a final state. $\square$

The intention of the term right-ignoring is that if we can accept a prefix vx of some word $w \in [\![r]\!]_\mathcal{R}$, then $vxu \in [\![r]\!]_\mathcal{R}$ for all $u \in \{x_\top\}^*$. However, this is not precisely expressed by our definition: It requires that all involved states must be final. Consequently, the regular expression $r' = xx_\top?(x_\top x_\top)^*$ is not right-ignoring w.r.t. x, although $xu \in [\![r']\!]_\mathcal{R}$ for all $u \in \{x_\top\}^*$. But r' is a rather cryptic way of denoting the language $[\![xx_\top{}^*]\!]_\mathcal{R}$. We can therefore assume that regular expressions of this kind do not occur in practice.

It can be efficiently determined whether a regular expression r is right-ignoring w.r.t. to a variable x. Figure 7.10 gives a corresponding algorithm *IsRiRegExp*. The algorithm collects in set seen the final states that are reachable from q through $x_\top$-transitions and final states only. In each iteration of the **while**-loop, the set new contains the states found in the last iteration; the set temp of final states reachable by an $x_\top$-transition from a state in new is computed. As soon as temp contains a state q_1 that is already in seen, the loop terminates: Then there must be an $x_\top$-cycle through q_1 and q_1 is reachable from q using only $x_\top$-transitions. Conversely, if temp is empty, then there is no such cycle and r is not right-ignoring w.r.t. x. Note that the **while**-loop computes the set seen *incrementally*: The transitions for each state in $Q_\top$ are examined at most once.

The complexity of algorithm *IsRiRegExp* is quadratic in the size of A, provided that the set operations can be done in constant time. More precisely, it is $\mathcal{O}(|F| \cdot |\delta|)$: In order to determine Q_x, at most one transition must be examined for each state in Q (due to Proposition 4.3 all transitions for q are labeled with the same symbol). The **foreach** loop is executed at most $|F|$ times. Be-

```
Algorithm IsRiRegExp
Input:   An NFA $A = (Q, q_0, F, \delta)$
         A variable $x$
Output:  true  if $A$ is right-ignoring w.r.t. $x$
         false otherwise
Algorithm:

  $Q_x := \{q \mid (q_1, x, q) \in \delta$ for some $q_1\}$;
  if $Q_x \not\subseteq F$ then return false
  else foreach $q \in Q_x$ do
      seen := $\{q\}$; new := $\{q\}$;
      while new $\neq \emptyset$ do
          temp := $\{q_1 \in F \mid q \in$ new, $(q, x_\top, q_1) \in \delta\}$;
          if temp $= \emptyset$ then return false;
          else if temp $\cap$ seen $\neq \emptyset$ then new := $\emptyset$;
          else new := temp $\setminus$ seen;
               seen := seen $\cup$ temp;
  return true;
```

Figure 7.10: An algorithm for determining whether a regular expression is right-ignoring w.r.t. a variable

cause the set seen is computed incrementally, the whole **while**-loop and thus each iteration of the **foreach**-loop has a complexity of $\mathcal{O}(|\delta|)$.

7.4.2 Match-Relevance and Right-Ignoring Context Grammars

Let us now have a look at the definition of $f_0 \leadsto_\pi r$:

$$f_0 \leadsto_\pi r \text{ iff } f_0 \leadsto_\pi e \text{ for some } e = \ldots \sqcap +r \sqcap \ldots.$$

In order for π to be an r-context, it must also be an e-context, i.e., each regular expressions occurring in e must be fulfilled if its sign is $+$, and may not be fulfilled if its sign is $\neg$. Thus, even if r is right-ignoring w.r.t. x, we must inspect all right siblings of a subtree matching x in order to verify these side-conditions. If we want to skip the second pass, it is therefore necessary, that no forest expression involved in a *Down*-transition uses conjunction or negation. More precisely, each forest expression that is used for matching the context, must have the form $+r$ where r is right-ignoring w.r.t. the variable that is used for matching the context. Let us formalize this:

Let $C = (G, X_\circ)$ be an extended context grammar with $G = (X, E_0, R)$. A forest expression or variable is classified as *match-relevant* as follows:

1. All $x \in X_\circ$ are match-relevant.
2. If x is match-relevant, x occurs in r and $e = \ldots \sqcap \sigma r \sqcap \ldots$, then e is match-relevant.
3. If e is match-relevant and $x \to a\langle e\rangle$, then x is match-relevant.

As a direct consequence, if E_0 contains no match-relevant forest expressions, then C can never have a match. C is called *right-ignoring* if all match-relevant forest expressions in G have the form $e = +r$ such that r is right-ignoring w.r.t. all match-relevant variables.

Example 7.10: None of the example grammars presented so far is right-ignoring. The context grammar $C_6 = (G_6, \{x_c\})$ is right-ignoring, where $G_6 = (\{x, x_1, x_b, x_c\}, \{\_\, x\, \_\}, R)$ with the following rules:

$$\begin{array}{lll} x \to a\langle \_\, x\, \_\rangle & x_b \to b\langle \neg\, \_\, x_1\, \_\rangle & x_1 \to c\langle \_\rangle \\ x \to b\langle x_b x_c\, \_\rangle & x_c \to c\langle \_\rangle & \end{array}$$

The match-relevant variables are x_c and x; the match-relevant forest expressions are $\_\, x\, \_$ and $x_b x_c\, \_$. Obviously, both of these forest expressions consist of a single positive regular expression which is right-ignoring w.r.t. x and x_c.

Note that variables x_c and x_1 are structurally equivalent. However, if we remove x_1 and replace it by x_c, then the forest expression $\neg\, \_\, x_1\, \_$ becomes match-relevant; because it consists of a negated regular expression, this would prevent C_6 from being right-ignoring. □

In order to decide whether a context grammar C is right-ignoring, we first have to determine the match-relevant variables and forest expressions in C. For this purpose we construct a system of boolean inequations, which has a variable $<x>$ or $<e>$ for each variable x or forest expression e:

$$\begin{array}{lll} <x> \leftarrow \mathit{true} & \text{if } x \in X_\circ \\ <x> \leftarrow <e> & \text{if } x \to a\langle e\rangle \text{ for some } a \\ <e> \leftarrow <x> & \text{if } e = \ldots \sqcap \sigma r \sqcap \ldots \text{ and } x \text{ occurs in } r \end{array}$$

The system of inequations can be constructed in time linear to the size of C; its least solution σ can be computed in linear time because it is a boolean system. A variable or forest expression is match-relevant if its value under σ is *true*.

We must now check for each match-relevant regular expression r whether it is right-ignoring w.r.t. each match-relevant variable x. In the worst case, all variables and regular expressions are match-relevant; thus at most $\mathcal{O}(|X| \cdot l)$ such checks are necessary, where l is the number of regular expressions in C. It can thus be decided in polynomial time whether a grammar is right-ignoring.

The main result of this section is that if C is right-ignoring, then all candidates for a match identified by $A_{\vec{G}}$ do indeed match; C can thus be implemented by a single pass:

Theorem 7.4: For a right-ignoring context grammar $C = (G, X_\circ)$, let λ be the $A_{\vec{G}}$-labeling of f_0. Then $\pi \in [\![C]\!]\, f_0$ iff $\lambda(\pi) = (q, p, q')$ and $y' \in q'$ with $(y, x, y') \in \delta$ for some $x \in X_\circ$.

The proof is given in Appendix A.9. □

Note that, though $A_{\vec{G}}$ proceeds from left to right, the matches of C are not found in document order: It can only be decided whether a subtree t matches after t has been completely traversed. If any subtrees of t match as well, then they are found earlier. Section 9.1.4 will explain how a one-pass matcher can be implemented in spite of this.

7.5 Bibliographic Notes

In contrast to regular tree languages, which are well studied and understood, there has only been few work concerning the location of subtrees in context.

[Mur96] uses pointed trees as introduced by [Pod92, NP93] for specifying the contextual conditions: A *pointed tree* is a tree with exactly one occurrence of a special symbol $\circ$ as a leaf. The concatenation of two pointed trees t_1 and t_2 is done by replacing the $\circ$ in t_1 with t_2. A context can then be described by the concatenation of a sequence of pointed trees; a contextual condition is a regular language of such sequences. It can be given as a regular expression over a finite set of *pointed base-tree representations*. A pointed base-tree representation is, for binary trees, of the form $a\langle S\circ\rangle$ or $a\langle\circ S\rangle$, where S is a regular tree language. Though this might be practicable in the binary case, it is inconvenient for larger ranks: For a symbol with rank n, there are n possibilities for the position of $\circ$, and $n-1$ regular tree languages must be given for the other children of a. For non-ranked trees there are even infinitely many possibilities for the position of $\circ$. A possible way of transferring this specification method to non-ranked trees and forests is to use base representations of the form $a\langle L\circ R\rangle$, where L and R specify regular forest languages. Indeed, the context qualifiers of the pattern language which will be introduced in Section 8.2 are inspired by this approach.

[Mur96] implements a contextual condition by running a bottom-up tree automaton on the input tree and annotating each node with its state in that run. A top-down automaton on the annotated tree then identifies the matches of the pattern. Using the ranked-tree representation η_2 from Section 4.2.1 for forests, this bottom-up automaton corresponds to an LFA in our framework; the top-down automaton is the counterpart of our RPA $B_G^{\leftarrow}$. However, since the first traversal is by a bottom-up automaton, Murata's algorithm always requires the runs of both automata; he can not give a special class of contextual conditions, such as our right-ignoring grammars, where a single pass suffices.

A different approach is taken by [NS99]: Here a query incorporating both the structural and the contextual condition is expressed as a formula of monadic second order logic (MSO) with one free first-order variable. It is a long-known fact that MSO without free variables expresses exactly the regular tree languages [TW68, Don70]. A query in [NS99] is implemented by a *query automaton* which is a deterministic two-way automaton as defined by [Mor94], without a pushdown but enhanced with a *selection function*. The selection function determines for each node whether it matches the query, depending on the state of the automaton and the symbol at that node. The basic construction is for ranked trees; in order to implement MSO formulae over non-ranked trees, the authors extend query automata with *stay transitions*.

Query automata are a rather theoretical solution to the problem of locating matches: The automaton constructed for a query must visit each node in the input tree a number of times which is bounded only by the size, more precisely the depth of the tree; for large trees this is unacceptable in practice. Moreover, the size of the automaton is iterated exponential in the size of the MSO formula, because it incorporates a one-way bottom-up automaton for checking closed MSO formulae. Due to [TW68], the construction of a such an automaton requires k constructions of a complement automaton in an intermediate step, where k is the alternation depth of $\forall$ and $\exists$ quantors in the formula. Each of these constructions can lead to an exponential growth of the state space. However, though this iterated exponential size is not practicable, we should not conceal that MSO formulae are an extremely succinct formalism.

[BHW98] uses *caterpillars* for specifying contextual conditions. A caterpillar is a sequence of node tests and movements. A node test can either check whether a node is labeled with some symbol $a \in \Sigma$, or whether it is the root (*isRoot*) or a leaf (*isLeaf*) of the input forest, or the left-most (*isFirst*) or right-most (*isLast*) child of another node. A movement directs the next node test to one of the neighbors of the current node: its parent (*up*), immediate left or right sibling (*left* or *right*), or its left-most (*first*) or right-most (*last*) child. A node matches the caterpillar w if, starting at this node, the sequence of movements and node-tests can be successfully performed. A contextual condition is then a caterpillar language, i.e., a regular word language of caterpillars. Note that for caterpillars it does not matter whether the input forest is ranked.

Though caterpillars are a very intuitive formalism, their implementation is rather inefficient; the authors give an algorithm for locating all matches of a caterpillar language C in $\mathcal{O}(n \cdot m)$ time, where n is the size of the input tree, and m is the number of states of an NFA A accepting C. This algorithm does not work on the tree itself: It solves a system of inequations, assigning each node in the tree the set of states in A from which a final state is reachable by successful movements and node tests starting at that node. The order in which an algorithm solving this system of inequations visits the nodes of the input forest is rather unpredictable. Apparently this algorithm can not be performed by a fixed number of traversals through the input tree.

A sensible restriction of caterpillars is to constrain movements to a specific traversing strategy. For instance, in order to make a caterpillar perform a left-to-right depth-first traversal, one would disallow *left* and *last* movements; moreover a *down* movement would be illegal after an *up* move. Since this traversing order corresponds to that of LPAs, it is not hard to see that such a caterpillar language could be implemented efficiently with a pushdown forest automaton.

It is an intriguing open problem whether caterpillars can express all regular contexts: This would include the ability of specifying that, e.g., the left sibling of the target fulfills a regular structural condition. With caterpillars, we can express structural conditions by claiming that the root of a subtree is in a caterpillar language. Such structural conditions are called *caterpillar-regular* languages. The authors of [BHW98] show that each caterpillar-regular tree language is a regular tree language, and that each local tree language is caterpillar-regular. It remains open whether each regular tree language is also caterpillar-regular. In the framework of pebble automata for ranked trees, [EH99] suggests a candidate for a regular tree language whose implementation requires at least one pebble, but can not prove this. Since caterpillars closely correspond to Engelfriet's automata without pebbles, this proof would imply that caterpillars can not express all regular tree languages.

In our previous papers [NS98b, NS98a], we used μ-formulae and, alternatively, constraint systems over pointed trees for specifying contextual conditions (cf. 5.3). However, we restricted ourselves to contextual conditions that correspond to right-ignoring grammars. For these queries we gave an algorithm which is very similar to the one-pass algorithm in 7.4.

Chapter 8

Querying XML Documents

In this chapter we adapt our grammar formalism to the requirements of XML: We add handling of plain text, attributes and processing instructions. Then we introduce a pattern language for more succinct formulation of queries.

8.1 Particularities of XML Documents

We have presented an algorithm for locating matches of context grammars in a forest by one or two runs of a pushdown forest automaton. In document processing this procedure is called *querying*. In order to query XML documents, we have to extend our grammar formalism to deal with some XML-specific aspects not covered by forest language theory.

8.1.1 External Predicates for Matching Character Data

When querying an XML document, the alphabet Σ is chosen as the set of element types occurring in the document. But this does not cover all nodes in an XML document tree: In addition to other elements, an element can also contain character data, i.e., text. In order to deal with character data, we could extend the alphabet with the set of characters and treat each single character as a leaf on its own. But this has two major disadvantages:

- Since XML documents are written in UNICODE, the alphabet must contain the full range $\mathcal{U}$ of UNICODE characters, which are theoretically more than a million. An alphabet of this size is hard to implement efficiently.
- When searching for text in a document, one is usually not only interested in a single character but in a whole sentence or paragraph. For instance, we might want to query for all section titles containing the word "automata".

It is therefore sensible to treat each segment of character data as a single node. This prevents location of single characters, but that is hardly ever required in practice. Thus, the set of trees is now given by

$$t \ ::= \ a\langle f \rangle, \ a \in \Sigma \ \mid \ "s", \ s \in \mathcal{U}^*$$

In order to match such a text node, we allow to specify structural conditions on text by a special mechanism, namely regular expressions over the UNICODE alphabet. In order to distinguish them from regular expressions over tree variables, we call these regular expressions *text patterns*. Now we allow text patterns as an additional form of right-hand sides of grammar rules (recall from 4.1 that $\mathcal{R}_X$ is the set of regular expressions over X):

$$x \to "\tau", \quad \tau \in \mathcal{R}_\mathcal{U}$$

We extend the meaning function for grammars accordingly:

$$"s" \in [\![G]\!]\, x \text{ iff there is a } \tau \in \mathcal{R}_\mathcal{U} \text{ with } x \to "\tau" \text{ and } s \in [\![\tau]\!]_\mathcal{R}.$$

In order to implement grammars with text patterns, we also have to enhance our notion of a forest automaton. Since a forest automaton can not check whether a text matches a text pattern, it treats these patterns as a set of *external predicates* $T = \{\tau_1, \ldots, \tau_k\}$: In order to perform a transition at a text node, each text pattern is surveyed by a call to an external procedure, which in this case[1] performs a run of a deterministic finite automaton on the UNICODE text. The tree state for the text node is determined from the set of fulfilled predicates by a special up-relation $Up_{txt} : 2^T \to P$. In a bottom-up automaton it is always necessary to examine all text patterns. By contrast, a pushdown automaton can reduce the number of text patterns to be checked to those which might contribute to a succeeding side-transition, similarly to the down-relation of $A_{\vec{G}}$ which selects a set of sensible NFA states. This selection happens with a special down-transition $Down_{txt} : Q \to 2^T$. With these concepts we can extend $\delta_\mathcal{T}$ for a DLPA to deal with text nodes:

$$\delta_\mathcal{T}(q, "s") = Up_{txt}\{\tau \mid \tau \in Down_{txt}\, q, s \in [\![\tau]\!]_\mathcal{R}\}$$

In the construction of $A_{\vec{G}}$, we must now add support for text patterns, i.e., define the two functions Up_{txt} and $Down_{txt}$:

$$\begin{aligned} Down_{txt}\, q &= \{\tau_j \mid y \in q,\ (y, x, y_1) \in \delta,\ x \to "\tau_j" \text{ for some } x, y_1\} \\ Up_{txt}\, M &= \{k \mid R_k = x \to "\tau_j" \text{ and } \tau_j \in M\} \end{aligned}$$

The definition of these two functions for $B_{\overleftarrow{G}}$ is trivial since this (top-down) automaton uses the tree states produced by $A_{\vec{G}}$. It is only a technical issue to integrate handling of text patterns into the proofs of Theorems 6.6, 7.1 and 7.4.

The use of external predicates for text patterns makes precomputing the reachable states of an automaton virtually impossible: If the grammar contains n text patterns, then each text node in the document can match one of the 2^n subsets of these patterns. In the case of pushdown automata the situation is slightly less serious: The maximal number m of text patterns resulting from a $Down_{txt}$ transition might be smaller than n. Still there are 2^m possible transitions at a single text node. It is therefore sensible to delay the computation of a transition until it is actually needed during the run of the automaton (see 9.1.2 for more details).

8.1.2 Text Patterns

A text pattern is a regular expression over UNICODE characters. The syntax for regular expressions presented in 4.1, however, is insufficient for the needs

[1]Note that the concept of external predicates allows for tests of arbitrary, even non-regular conditions on text. E.g., an external predicate might check whether the text contains a prime number of words, or whether it is correctly spelled. In this framework we are satisfied with regular text patterns.

in document processing. We therefore make the following modifications to the syntax of text patterns, motivated by the syntax used for the UNIX tool *grep* [FSF99]:

Character Ranges: When searching for text it is often desired to specify a *range* of characters instead of just a single character. For instance, a number is described by the text pattern "`(0|1|2|3|4|5|6|7|8|9)`$^+$". This is rather uncomfortable because all digits between 0 and 9 must be enumerated; for larger ranges of characters the case is even worse. Therefore we introduce a shorter way of specifying a character range using square brackets:

$$\begin{array}{lcl} cr & ::= & [ci\ldots ci] \mid [\hat{}\,ci\ldots ci] \mid . \mid \sim \mid c \in \mathcal{U} \\ ci & ::= & c \in \mathcal{U} \mid c_1{-}c_2 \mid \sim \end{array}$$

Let us informally describe character ranges and their meaning: A *character interval ci* is either a single character, describing itself, the special character ~ which stands for white-space characters, or of the form $c_1{-}c_2$. In the last case it represents all characters whose UNICODE value lies between those of c_1 and c_2. A character range *cr* is composed of a sequence of intervals and describes all characters belonging to the single intervals. If the first character within the brackets is "^", then the meaning of the character range is negated: It describes all characters not in any of the intervals. The character "." is a short-hand for the range of all UNICODE characters. The character range [~] can be abbreviated to the single character "~", and for each character $c \in \mathcal{U}$, c abbreviates $[c]$.

For example, "`[a−zA−Z0−9]`" is the range of alphanumeric ASCII characters, whereas "`[^ ~,.:;?!]`" describes the range of all characters except for white-space and punctuation characters. A text pattern is now a regular expression over character ranges: For instance, we can describe numbers by "`[0−9]`$^+$".

White Space: Text pattern "`to be or not to be`" matches exactly the string "`to be or not to be`". However, in an XML document there might be a line break between two of the words, and additionally there might be white space for purposes of indentation. Therefore, we interpret the space character as a short-hand for "~$^+$". If the intention is to match really a single space character, it must be escaped with a "\" in the text pattern.

For instance, "`a b`" matches, among others, the strings "`a     b`", "`a   b`" and "`a b`", whereas "`a\ b`" matches only "`a b`".

Leading and Trailing Characters: In order to match a text node containing a certain word, e.g., "`forest`", it would be convenient to use that word as a text pattern. Therefore, we interpret an entire text pattern as implicitly preceded and followed by ".*", thus requiring the text to contain a match of the text pattern, instead of exactly matching it. However, if an exact match is desired, we can precede the pattern with "^" for disallowing leading characters; terminating a text pattern with "$" requires that there are no trailing characters.

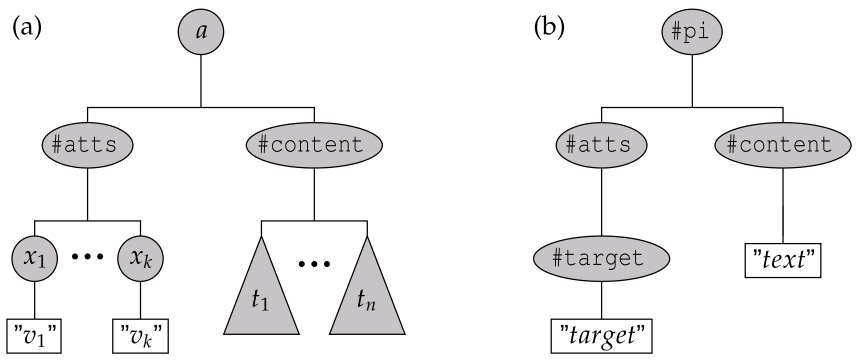


Figure 8.1: Representation of XML documents as forests: (a) an element with start-tag <a x_1="v_1" ... x_k="v_k"> and content $t_1 \ldots t_n$, and (b) a processing instruction <?*target text*?>.

8.1.3 Representation of XML Elements and Attributes

In 8.1.1 we chose the set of element types as the alphabet Σ. In XML, however, a node of the document tree is not only labeled with its element type, but additionally with a set of attribute assignments (cf. 1.1.1). One possibility of dealing with attributes is by external predicates[2]. For uniformity, however, we follow a different approach similar to that of, e.g., DSSSL [ISO96] and XPATH [W3C99c]. There the attributes of an element are represented in the document tree as an additional subtree of the element. Our representation is illustrated by Figure 8.1 (a): Each element has exactly two children, labeled with auxiliary symbols #atts and #content . The #atts subtree contains for each attribute assignment x="s" one subtree labeled x having a single child: a text node giving the attribute value s. Note that attribute assignments are unordered; therefore they need not appear in the same order as in the XML start-tag. The children of the #content node are the content of the element. This representation requires extension of the alphabet Σ with the two auxiliary symbols and the attributes names.

We can now specify conditions on the attributes of XML elements: For instance, in order to match an a element that has an attribute u whose value contains the word "forest" but which has no attribute named v, and whose content matches a forest expression e, we specify the following rules:

$$\begin{array}{llll}
x & \rightarrow \texttt{a}\langle x_a x_c\rangle & x_u & \rightarrow \texttt{u}\langle x_t\rangle \\
x_a & \rightarrow \texttt{\#atts}\langle \_\, x_u \,\_ \sqcap \neg \,\_\, x_v \,\_\rangle & x_v & \rightarrow \texttt{v}\langle x_\top\rangle \\
x_c & \rightarrow \texttt{\#content}\langle e\rangle & x_t & \rightarrow \texttt{"forest"}
\end{array}$$

Because this description of the intended structural condition is rather long-winded, we introduce an abbreviated syntax in which we can specify the above as follows:

$$x \rightarrow \texttt{<a u="forest"} \neg\texttt{v>}\ e$$

We call grammars in this abbreviated syntax *query grammars*. In a rule of a query grammar, a right-hand side for an element has the form <*a aps*> *e*, where

[2]It is straight-forward to extend the mechanism of external predicates to non-text nodes.

aps is a sequence of possibly negated *attribute patterns*. Similarly to the regular expressions in forest expressions, each attribute pattern has a sign $+$ or $\neg$; the $+$ is usually omitted. Each attribute pattern *ap* has the form u or u="τ", where u is an attribute name and τ is a text pattern.

$$\begin{array}{lcl} \mathit{rhs} & ::= & "\tau" \mid \texttt{<}a\ \mathit{aps}\texttt{>}\ e \\ \mathit{aps} & ::= & \epsilon \mid \mathit{ap}\,\mathit{aps} \mid \neg\mathit{ap}\,\mathit{aps} \\ \mathit{ap} & ::= & u \mid u="\tau" \end{array}$$

An attribute pattern u specifies that the element must have an attribute named u; for u="τ" its value must additionally match the text pattern τ. A negated attribute pattern $\neg u$ means that no attribute named u may be present; for $\neg u$="τ", there may be such an attribute, but its value must not match τ.

Query grammars are translated to extended context grammars by a function γ_{rhs}. It replaces each abbreviated right-hand side $\texttt{<}a\ \sigma_1 ap_1 \ldots \sigma_k ap_k\texttt{>}\ e$ with a set of right-hand sides in conventional syntax, introducing new variables $x_a, x_c, x_1, \ldots x_k$ not occurring elsewhere:

$$\begin{array}{l} \gamma_{rhs}(x \rightarrow \texttt{<}a\ \sigma_1 ap_1 \ldots \sigma_k ap_k\texttt{>}\ e) = \\ \qquad R_1 \cup \ldots \cup R_k \cup \{x \rightarrow a\langle x_a x_c \_\rangle,\ x_c \rightarrow \texttt{\#content}\langle e\rangle, \\ \qquad\qquad x_a \rightarrow \texttt{\#atts}\langle \sigma_1\ \_x_1\_\ \sqcap \ldots \sqcap\ \sigma_k\ \_x_k\_\rangle\} \end{array}$$

where $R_i = \gamma_{ap}(x_i, ap_i)$ for $i = 1, \ldots, k$, with:

$$\begin{array}{ll} \gamma_{ap}(x, u) & = \{x \rightarrow u\langle x_\top\rangle\} \\ \gamma_{ap}(x, u="\tau") & = \{x \rightarrow u\langle x_\tau\rangle,\ x_\tau \rightarrow "\tau"\},\ x_\tau \text{ is a new variable} \end{array}$$

Note that in the definition of γ_{rhs}, a $\_$ follows the x_c in the right-hand side for variable x, though the representation of documents ensures that the `#content` child of a has no right sibling. However, if we omit the $\_$ then the resulting grammar can never be right-ignoring, even if the query grammar is.

8.1.4 Element-Type Patterns

In a rule of the form $x \rightarrow \texttt{<}a\ \mathit{aps}\texttt{>}\ e$ a single element type a can be specified. It would be more convenient to subsume a set of element types into a single rule. For instance, in an HTML or XHTML document, if we are interested in the text of headers (h1,...,h6) containing the word "forest", the following rules are required:

$$\begin{array}{lll} x \rightarrow \texttt{<h1>}\ x_t & x \rightarrow \texttt{<h3>}\ x_t & x \rightarrow \texttt{<h5>}\ x_t \\ x \rightarrow \texttt{<h2>}\ x_t & x \rightarrow \texttt{<h4>}\ x_t & x \rightarrow \texttt{<h6>}\ x_t \end{array}$$

$$x_t \rightarrow \texttt{"forest"}$$

It would be much more concise to write a single rule comprising all six header element types. We therefore allow specification of an alternative of one or more element types with the help of an *element-type pattern*:

$$x \rightarrow \texttt{<h1|h2|h3|h4|h5|h6>}\ x_t$$

Another useful feature is negation in element-type patterns: Suppose we want to locate all a elements that are not a subtree of another a element, i.e., that have no ancestor labeled a. This grammar must have the following rules:

$x \longrightarrow$ `<a>` e
$x \longrightarrow$ `<`b`>` e, for $b \in \Sigma \setminus \{\texttt{a}\}$

The second line is a template for as many rules as the size of Σ minus one. For large alphabets, which are quite common as XML DTDs, this is very inconvenient. Moreover, if the DTD of a document is not known in advance, then neither is Σ: The elements different from *a* can not be enumerated, and a grammar can not be specified at all. We therefore introduce negation to element-type patterns, enabling the following:

$x \longrightarrow$ `<¬a>` e

Summarizing, right-hand sides in query grammars may have the following form:

$$\begin{array}{lll} rhs & ::= & \text{"}\tau\text{"} \quad | \quad \texttt{<}ep\ aps\texttt{>}\ e \\ ep & ::= & a_1|\dots|a_k \quad | \quad \neg\, a_1|\dots|a_k \quad | \quad * \end{array}$$

The first form of an element-type pattern *ep* is equivalent to enumerating the rule for each a_i; the second form corresponds to enumerating the rule for each $b \in \Sigma$ which is different from all a_i. A $*$ denotes an arbitrary element type; it is equivalent to a negation of zero element types. Section 9.1.2 will explain how negated element-type patterns can be implemented without explicitly enumerating all the rules.

8.1.5 XML Processing Instructions

In addition to other elements and character data, an XML element can contain comments and processing instructions (cf. 1.1.4). While comments may be ignored, processing instructions form an integral part of the document. Since they may contain essential information for an application, it is desirable to specify processing instructions in grammars. A processing instruction has the form `<?`*target text*`?>`, where *target* and *text* are character data. In order to deal with such a processing instruction, we represent it as an element with the auxiliary element type `#pi`. Figure 8.1 illustrates this representation: The target is specified as the value of an auxiliary attribute `#target`, and the text of the processing instruction is the content of the element. In order to specify processing instructions in grammars, we allow a new form of right-hand sides in query grammars:

$x \longrightarrow$ `<?`*target*`?>` e

This is defined to be equivalent to $x \longrightarrow$ `<#pi #target=`"*target*"`>` e, and can thus be translated to context grammar syntax by γ_{rhs}.

8.1.6 White Space

XML documents are frequently formatted according to the element structure by indenting, i.e., insertion of white-space characters. White space is not part of an element's content, unless that element has declared mixed content (cf. 1.1.1). An XML parser must, however, always report the white space to the application, regardless of whether it is significant (cf. Section 2.10 of [W3C98b]). The white space is thus present in the document tree. In grammars, however, we do not want to bother with white space. For instance, we want the regular expression $x_a x_b$ to match two consecutive elements, regardless of any white space in between them. Similarly, processing instructions can occur anywhere in the content of an element, but do not represent data. We therefore want to ignore processing instruction unless they are explicitly mentioned in the grammar.

In order to ease the specification of grammars, we therefore assume that, similarly to the variable $x_\top$, a variable x_w is always implicitly declared with the following rules (recall that ~ and . denote a single white-space character and an arbitrary character in the text pattern syntax, respectively):

$$x_w \longrightarrow \text{"}{\sim}^*\text{"}$$
$$x_w \longrightarrow \texttt{<?}\ .^*\ \texttt{?>}\ \_$$

That is, x_w structurally matches an arbitrary processing instruction or text node consisting of white-space characters only. Now we modify the translation function γ_{rhs} to transform regular expressions on right-hand sides of rules: It replaces subexpressions of the form $r_1 r_2$ with $r_1 x_w{}^* r_2$, and similarly r^* with $(r x_w{}^*)^*$ and r^+ with $(r x_w{}^*)^+$. If this transformation is not desired, concatenation can be specified with the "," operator, and repetition with ** and $^{++}$. Moreover, an entire regular expression r on the right-hand side of a rule is replaced by $x_w{}^* r x_w{}^*$. In analogy to text patterns, this can be disabled by preceding the regular expression with a ^ and appending a \$.

8.2 A More Convenient Pattern Language

Grammars are a precise method of specifying regular forest languages and contextual conditions. However, each rule of a grammar can speak only about one single node in the input forest. For specifying a nesting of element types, at least as many rules as the depth of this nesting are required. For instance, in order to describe the constant tree $\texttt{a}\langle\texttt{b}\langle\texttt{c}\rangle\rangle$, we need the three rules

$$x_a \longrightarrow \texttt{a}\langle x_b\rangle$$
$$x_b \longrightarrow \texttt{b}\langle x_c\rangle$$
$$x_c \longrightarrow \texttt{c}$$

In many circumstances, such as querying from the command line, it is desirable to specify the query in a single line, without the need of introducing auxiliary variables for subtrees. XPATH [W3C99c], the querying language of XSLT [W3C99b], offers this possibility by specifying the query as a *location path*. A location path describes the context by means of the path from the target to the root of the document tree, i.e., by its ancestors. However, in XPATH it is not possible to specify regular conditions on the siblings of a node. Instead, XPATH provides an expression language for filtering the nodes selected by a location path. The expression language has the capability of navigating through the document tree in arbitrary directions and can thus also relate to siblings, children or ancestors of a node. This contradicts our intention of a fixed traversing order. However, we adopt the concept of path-oriented context specification.

8.2.1 An Informal Description of the Pattern Language

We will now present a querying language similar to XPATH that additionally allows specification of regular conditions on the siblings of nodes. Let us first introduce this language informally, by means of examples.

Node Tests and Node Patterns

A *node test nt* describes the allowable element types for a subtree:

$$\begin{array}{lcl} nt & ::= & * \;\mid\; a \;\mid\; <\!ep\!> \\ ep & ::= & a_1|\ldots|a_k \;\mid\; \neg\, a_1|\ldots|a_k \;\mid\; * \end{array}$$

Note that element-type patterns *ep* are as for query grammars. The node test $*$ is fulfilled by all element types, whereas only elements of type a satisfy a. The node test $<\!ep\!>$ is true for an element with type a if $ep = a_1|\ldots|a_k$ and $a = a_i$ with $1 \leqslant i \leqslant k$; if $ep = \neg\, a_1|\ldots|a_k$, then a must be different from all a_i.

A *node pattern np* describes a structural property of a subtree by means of its element type and its attributes:

$$\begin{array}{lcl} np & ::= & "\tau" \;\mid\; <?\tau?> \;\mid\; nt\; aqs \;\mid\; .\; aqs \\ aqs & ::= & \epsilon \;\mid\; [\,@ap\,]\; aqs \;\mid\; [\,\neg@ap\,]\; aqs \\ ap & ::= & u \;\mid\; u="\tau" \end{array}$$

where u is an attribute name and τ is a text pattern. Note that attribute patterns *ap* are the same as for query grammars. A text node fulfills node pattern "τ" if its text is in $[\![\tau]\!]_{\mathcal{R}}$, whereas the node pattern $<?\tau?>$ is fulfilled by a processing instruction whose target is in $[\![\tau]\!]_{\mathcal{R}}$. For elements satisfying a node test *nt*, a list of attribute qualifiers *aqs* can be specified, constraining the element's attributes. Each qualifier must be fulfilled unless it is negated by $\neg$; in that case it must not be fulfilled. A node pattern of the form $.\; aqs$ is similar, but it additionally matches text nodes and processing instructions. Note, however, that these kinds of nodes have no attributes; it therefore makes no sense to specify attribute qualifiers if text nodes or processing instructions shall be matched.

For instance, the node pattern `<a|b>`[@`x` = "`1`"][¬@`y`] describes all elements of type `a` or `b` that have an attribute `x` whose value contains "1" and do not have an attribute `y`. Similarly, `<?^fxp-?>` describes all processing instructions whose target begins with `fxp-`.

Note that though the @ appears to be superfluous, it is necessary for distinguishing attribute qualifiers from other forms of qualifiers introduced later on.

Path Patterns and Tree Patterns

Path patterns and *tree patterns* are composed from node patterns using operators /, // and ‖:

$$\begin{array}{lcl} pp & ::= & np \;\mid\; pp_1 \,\|\, pp_2 \;\mid\; pp\; tp \;\mid\; (pp) \\ tp & ::= & /pp \;\mid\; /\!/pp \end{array}$$

A path pattern *pp* or a tree pattern *tp* identifies subtrees of a tree *t* by describing the paths from the root of *t* to these subtrees. Intuitively, operators / and //

express the child and descendant relations, respectively, whereas ‖ specifies an alternative. If a path pattern *pp* identifies a subtree t' of a tree t, then we also say that t' matches *pp* in t, or *pp* locates t' in t, and similarly for a tree pattern *tp* [3].

The simplest form of a path pattern is a node pattern *np*. It locates t itself if that satisfies *np*. A path pattern $pp_1 \| pp_2$ identifies all subtrees in t matching either pp_1 or pp_2. If a subtree t' matches *pp* in t, then *pp tp* identifies all subtrees t'' that match *tp* in one of the children of t'.

A tree pattern *tp* of the form /*pp* locates all subtrees in t that match *pp* in t, whereas //*pp* identifies all subtrees matching *pp* in a subtree of t. Let us illustrate this with some examples:

- The tree pattern /. always matches t itself for all t, whereas //`a[@x="1"]` matches all subtrees of t that are labeled `a` and have an attribute `x` whose value contains "`1`".
- The tree pattern /`a`/`b`//`c` matches all `c` elements that are descendants of a `b` element which is itself a child of t, provided that t has element type `a`.
- /(`a`‖`b`)/`c` matches all children of t with element type `c`, if t has element type `a` or `b`; otherwise no subtree matches in t.
- If t is a processing instruction whose target starts with `fxp-`, then the tree pattern `/<?^fxp-?>/"forest"` locates its text, provided the text contains the word `forest`.

For brevity, if a tree pattern *tp* does not occur as part of a path pattern *pp tp*, we allow omission of the leading /; but a leading // may never be dropped.

Observe that // is the only means of iteration, i.e., of specifying paths exceeding a fixed length. Since there is no means of constraining nodes on that path, it is impossible to specify, e.g., a path of arbitrary length on which all elements have type `a`. If this expressiveness is required, the query-grammar syntax must be employed.

Structure Qualifiers

The pattern language presented so far can only proceed to a single child of a node; we can not argue about the entirety of a node's children. We therefore add a new kind of qualifiers now: *structure qualifiers* for specifying the content of an element. These qualifiers can be specified as part of a node pattern, in addition to the attribute qualifiers presented above. We therefore extend the syntax of node patterns:

$$\begin{array}{lcl} np & ::= & "\tau" \;\mid\; \texttt{<?}\tau\texttt{?>}\; sqs \;\mid\; nt\ aqs\ sqs \;\mid\; .\ aqs\ sqs \\ sqs & ::= & \epsilon \;\mid\; [\,fp\,]\ sqs \;\mid\; [\,\neg fp\,]\ sqs \end{array}$$

where a *forest pattern fp* is a regular expression over tree patterns. A forest pattern is a structural condition and specifies a forest language. In order to define its meaning, let us first define a structural match of a tree pattern: A

[3]More precisely, because a subtree t' might occur twice within t, we should rather speak of occurrences of t' in t, or of paths in t which uniquely identify subtrees. For this informal description, however, it is more straight-forward to speak about subtrees.

tree pattern *tp structurally matches* a tree *t* if *tp* locates some subtree of *t*. Now a forest pattern *fp* matches a forest $t_1 \ldots t_n$ iff there is some $tp_1 \ldots tp_n \in [\![fp]\!]_{\mathcal{R}}$ such that t_i structurally matches tp_i for all *i*.

Specifying a structure qualifier [*fp*] for a node pattern *np* means that the children of a tree fulfilling *np* must match the forest pattern *fp*; if the qualifier is negated, then they must not match *fp*. Specifying two or more structure qualifiers means that all of them must be fulfilled: This is a way of expressing conjunction. For instance:

- The node pattern `a[b[c]]` is fulfilled only by an `a` element that has a single child `b` which itself has a `c` element as its only child.
- The node pattern `a[_b[_c_]_]` is fulfilled by an `a` element that has a child `b` which itself has child with element type `c`. Within patterns we use `_` as an abbreviation for .*, whereas within grammars it stands for $x_{\top}{}^*$.
- The node pattern `a[_(b|(//c))_][¬_c_]` is fulfilled by an element with type `a` which has either a child labeled `b` or a descendant labeled `c`, but it must not have a child of type `c`, which is specified by the second, negated qualifier.
- The node pattern `<?^fxp-?>["forest"]` is fulfilled by a processing instruction whose target begins with `fxp-` and whose text contains the word `forest`. In a structure qualifier for a processing instruction it makes no sense to specify something other than a text pattern because processing instructions contain only text.

In order to ease the handling of white space in forest patterns, we interpret them analogously to 8.1.2: Concatenation and repetition operators implicitly allow white space between elements. This can be disabled by using operators ",", ** and $^{++}$ instead.

Structure qualifiers can appear at any node pattern occurring in a path or tree pattern. They specify structural conditions on the nodes that lie on the path from the document root to the target. Therefore they introduce conjunction into patterns. For instance, the tree pattern $a[\_b\_]/c$ matches an element with type *c* that is a child of an *a* node only if that node also has a child of type *b*.

Note the difference between the two tree patterns $tp_1 = a/b[\_c\_]$ and $tp_2 = a/b/c$. Within a structure qualifier, these two patterns are equivalent, because they structurally match the same trees *t*, namely *a* elements with a *b* child that has a child *c*. However, they locate different subtrees of *t*: tp_1 identifies the *b* whereas tp_2 selects the *c* node.

Context Qualifiers

Structure qualifiers impose regular conditions on the children of a node matching a node pattern. But they can not be used for constraining the left or right siblings of the node matching the next node pattern in the path pattern. Consider, e.g., the path pattern a/b, and suppose we want to specify that *b* must be the first child of *a*. This is not possible with a structure qualifier. Even if we ensure that the first child of *a* is a *b* by specifying $a[b\_]/b$, the pattern still locates all *b* children of *a*.

In order to overcome this deficiency, we add another kind of qualifiers: *context qualifiers*. For a node pattern that can match an element, one optional context qualifier *cq* may be given. Note that it would not make sense to specify a context qualifier for text nodes or processing instructions: A text node has no children, and a processing instruction can only have a single child.

$$\begin{array}{lcl} np & ::= & \text{"}\tau\text{"} \;\;|\;\; \texttt{<?}\tau\texttt{?>}\; sqs \;\;|\;\; nt\; aqs\; sqs\; cq \;\;|\;\; .\; aqs\; sqs\; cq \\ cq & ::= & \epsilon \;\;|\;\; [\, fp_1 \# fp_2 \,] \end{array}$$

Specifying a context qualifier for a node pattern *np* attaches a contextual constraint, consisting of two forest patterns *l* and *r*, to a tree *t* satisfying *np*. If *np* is followed by a tree pattern *tp* in the path pattern, then the child of *t* in which the matches of *tp* are located must be such that its left siblings structurally match *l* and its right siblings structurally match *r*. This is most easily explained by some examples:

- In an element of type `a`, the tree pattern `a[#_]/b` identifies the first child of `a` if that child has element type `b`. This is the solution to the introductory example above.
- Consider the tree pattern *tp* = `.[a*#c*]/b`. In an arbitrary tree *t*, *tp* locates a `b` child of *t* all of whose left and right siblings have element types `a` and `c`, respectively.
- `a[#]//b` matches in a tree with element type `a` all descendants of `a` with element type `b`, provided that the `a` has only a single child. Note that this child may have arbitrarily many children: The context qualifier does not extend to the descendants of `a`, regardless of operator //.

Since a context qualifier can occur at any node pattern, it can also be part of a tree pattern in a structure qualifier, e.g., in `a[_(b[c#c]/b)_]`. Note however, that such a context qualifier can be replaced by a structure qualifier, in this case yielding `a[_(b[cbc])_]`.

Context Patterns and Patterns

A tree pattern identifies subtrees of an XML document tree. But the input to the pattern matcher is not a single tree: It is the forest consisting of the document element and all preceding or following processing instructions. We therefore allow specification of structure and context qualifiers also for this top-level forest. These qualifiers together with a tree pattern form a *context pattern*. A *pattern* is then a disjunction of context patterns.

$$\begin{array}{lcl} p & ::= & cp_1 \,\|\, \ldots \,\|\, cp_k \\ cp & ::= & sqs\; cq\; tp \end{array}$$

Let us describe the meaning of patterns by examples:

- The pattern *p* = `[_a_]//b` ‖ `[_c_]//d` locates all `b` elements in an XML document whose root element has type `a`, whereas it locates all `d` elements if the root element type is `c`.
- `[_#_*_]/<??>` selects all processing instructions that come before the document element, whereas `[_*_#_]/<??>` identifies those which

$$
\begin{array}{lcl}
p & ::= & cp_1 \,\|\, \ldots \,\|\, cp_k \\
cp & ::= & tp' \;\mid\; sqs\; cq\; tp \\[4pt]
sqs & ::= & \epsilon \;\mid\; [\, fp\,]\; sqs \;\mid\; [\,\neg fp\,]\; sqs \\
cq & ::= & \epsilon \;\mid\; [\, fp_1 \# fp_2 \,] \\[4pt]
fp & ::= & \epsilon \;\mid\; \wedge \;\mid\; \$ \;\mid\; \_ \;\mid\; tp'' \;\mid\; (fp) \\
 & \mid & fp_1 \,|\, fp_2 \;\mid\; fp_1 fp_2 \;\mid\; fp_1 , fp_2 \\
 & \mid & fp^{*} \;\mid\; fp^{**} \;\mid\; fp^{+} \;\mid\; fp^{++} \;\mid\; fp? \\[4pt]
tp' & ::= & pp \;\mid\; tp \\
tp'' & ::= & np \;\mid\; (tp') \\
tp & ::= & /pp \;\mid\; /\!/pp \\
pp & ::= & np \;\mid\; pp_1 \,\|\, pp_2 \;\mid\; pp\; tp \;\mid\; (pp) \\[4pt]
np & ::= & \text{"}\tau\text{"} \;\mid\; \texttt{<?}\tau\texttt{?>}\; sqs \;\mid\; nt\; aqs\; sqs\; cq \;\mid\; .\; aqs\; sqs\; cq \\
nt & ::= & * \;\mid\; a \;\mid\; \texttt{<}ep\texttt{>} \\
ep & ::= & a_1 | \ldots | a_k \;\mid\; \neg\, a_1 | \ldots | a_k \;\mid\; * \\
aqs & ::= & \epsilon \;\mid\; [\,@ap\,]\; aqs \;\mid\; [\,\neg @ap\,]\; aqs \\
ap & ::= & u \;\mid\; u{=}\text{"}\tau\text{"} \\[4pt]
a & ::= & \text{(an element type)} \\
u & ::= & \text{(an attribute name)} \\
\tau & ::= & \text{(a text pattern)}
\end{array}
$$

Figure 8.2: Summary of the pattern syntax.

come after the document element (recall that the node pattern $*$ matches only elements but not processing instructions).

Summary of the Patterns Syntax

Figure 8.2 summarizes the syntax of patterns. Observe the following:

- Nonterminal tp' explicitly accounts for omission of the leading / operator of tree patterns that are part of a path pattern.
- In order to avoid syntactical ambiguities, a tree pattern tp'' occurring in a forest pattern must be enclosed in parentheses unless it is just a node pattern. Otherwise the forest pattern $/a\ /b$ could be interpreted as a sequence of the tree patterns $/a$ and $/b$, or as the single tree pattern $/a/b$.
- We explicitly account for the alternative concatenation and repetition operators ",", ** and $^{++}$ in the syntax of forest patterns. Note also that the special symbols ^ and \$ may occur anywhere in a forest pattern, though their employment is hardly sensible other than at the start or end of a forest pattern.

8.2.2 Examples from XML Practice

Let us illustrate the use of patterns in querying XML by some more sophisticated examples which are motivated by real-world applications.

XML Example 20

```
<prod id='NT-document'>
  <lhs>document</lhs>
  <rhs>
    <nt def='NT-prolog'>prolog</nt>
    <nt def='NT-element'>element</nt>
    <nt def='NT-Misc'>Misc</nt>*
  </rhs>
</prod>
```

Figure 8.3: Representation of EBNF productions in the XML version of the XML recommendation.

Querying the XML recommendation

The first example queries the XML recommendation [W3C98b] itself. This document defines the syntax of XML by means of an EBNF grammar whose rules are scattered over the whole document. A developer of an XML application who wants to use this specification as a reference, has to search through the document in order to find a rule. In the XML version of the recommendation, each rule is given by a `prod` element, which has an `ID` attribute named `id` for referral and contains an `lhs` element and one or more `rhs` elements. The `lhs` element specifies the nonterminal defined by this rule; the `rhs` elements give the right-hand sides for this nonterminal. Each nonterminal occurring on such a right-hand side is represented by an `nt` element which carries an `IDREF` attribute named `def` identifying the `prod` element which defines this nonterminal. For instance, the production for nonterminal `document`, which is formatted as

[1] document ::= prolog element Misc*

is given by the element shown in Figure 8.3.

Note that the numbering of productions is done by the formatting process; the XML element therefore does not specify the production's number [1]. Let us now formulate some queries on this document:

- The pattern //`prod`[@`id` = "`^NT-Char$`"] matches the production whose `id` attribute is exactly `NT-Char`, whereas //`prod`[@`id`="`Char`"] matches all `prod` elements for which this attribute contains the word `Char`, thus also allowing for `CharData` and `NameChar`.
- If we don't want to use the `id` attribute for selecting the productions – perhaps we are not sure whether each `prod` element really has an `id` attribute – we can also specify a structure qualifier for the `lhs` child: //`prod`[(`lhs`/"`Char`")_] selects all productions whose `lhs` child contains a text node with the word `Char`.
- //`prod`[(`lhs`["`^Char$`"])_]/`rhs` selects all `rhs` children of productions whose left-hand side is exactly `Char`.
- The pattern //`prod`[#_(`rhs`/`nt`[@`def`="`Char`"])_]/`lhs`/"" matches the text of the left-hand sides of productions whose right-hand sides use a nonterminal whose name contains `Char`. This can be expressed more shortly – but less precisely – by //`prod`[_(//`nt`/"`Char`")_]/`lhs`/""

XML Example 21

```
<SCENE>
  <TITLE>SCENE I.  A desert place.</TITLE>
  <STAGEDIR>Thunder and lightning. Enter three Witches</STAGEDIR>

  <SPEECH>
    <SPEAKER>First Witch</SPEAKER>
    <LINE>When shall we three meet again</LINE>
    <LINE>In thunder, lightning, or in rain?</LINE>
  </SPEECH>

  <SPEECH>
    <SPEAKER>Second Witch</SPEAKER>
    <LINE>When the hurlyburly's done,</LINE>
    <LINE>When the battle's lost and won.</LINE>
  </SPEECH>

  <SPEECH>
    <SPEAKER>Third Witch</SPEAKER>
    <LINE>That will be ere the set of sun.</LINE>
  </SPEECH>
  ...

  <STAGEDIR>Exeunt</STAGEDIR>
</SCENE>
```

Figure 8.4: A scene of Shakespeare's "Macbeth" in XML.

- In order to find out the number of the production for `Char`, we can count the matches of //.[_#_(//`prod`[@`id`="^`NT-element`$"])_]//`prod` and add one. The pattern identifies all `prod` nodes that come before the production with identifier `NT-element` in document order. Note that due to the use of operator // for both `prod` subpatterns, the `prod` elements need not be siblings. They need only have ancestors which are siblings.

Querying Shakespeare

The second example deals with Shakespeare's play "Macbeth". Encoded in XML, a scene basically has a title and contains a sequence of speeches, each represented by a `SPEECH` element, interspersed with stage directions. Each speech contains a `SPEAKER` element and a sequence of lines containing plain text. Let us formulate some queries for this document:

- The pattern //`SPEECH`[_(`LINE`/"`thunder`")_] selects all speeches containing a line with the word `thunder`.
- The pattern //`SPEECH`[_(//`LINE`/"`hurlyburly`")_]/`SPEAKER`/. selects the speaker of a line containing the word `hurlyburly`, that is, the second witch.
- The same result is produced by the following pattern, using a context qualifier: //`SPEECH`[_#_(`LINE`/"`hurlyburly`")_]/`SPEAKER`/. This pattern is much more precise than the previous one: It requires that the `SPEAKER`

precedes the `LINE` elements within a `SPEECH`, and that the `LINE` elements are direct descendants of the `SPEECH` node.

- ✧ The pattern `//SPEECH[ _(SPEAKER/"Second Witch")_#_ ]/LINE/""` locates all text nodes in lines spoken by the second witch.
- ✧ The pattern `//SPEECH[ _(LINE/"hurlyburly")#_ ]/LINE` matches the line immediately following after the line containing `hurlyburly`, whereas `//*[ _(SPEECH//"hurlyburly")#_ ]/SPEECH/SPEAKER` selects the speaker who responds to that speech, that is, the third witch.
- ✧ `//*[<¬ACT>*#_ ]/ACT[<¬SCENE>*#_ ]/SCENE/TITLE/""` selects the title text of the first scene in the first act, namely: `SCENE I. A desert place.`
- ✧ `//SCENE[ _(//SPEAKER/"Witch")_ ][ _(//SPEAKER/"MACBETH")_ ]/TITLE` matches the titles of scenes in which both Macbeth and a witch speak.
- ✧ `//SCENE[ _(TITLE/"desert")_ ]/*[¬ _(SPEAKER/"Witch")_ ]/LINE` locates the lines in a scene whose title contains the word `desert`, which are not spoken by a witch.

8.2.3 Translation from Patterns to Grammars

We did not give a formal semantics to patterns. Instead we will now define a translation scheme from patterns to query grammars. Basically, the resulting grammar has one variable for each occurring tree pattern. Of course, an intelligent implementation will generate only a single variable for multiple occurrences of the same tree pattern; we disregard such optimizations here for simplicity.

The translation scheme is given by a function γ_p which yields for a pattern p a query grammar C_p. A path π then matches the pattern p in f_0 iff it matches C_p. C_p is obtained from the variables, start expressions, rules and target variables generated for the single context patterns of p:

$$\begin{aligned}\gamma_p(cp_1 \parallel \ldots \parallel cp_k) = {} & (X_1 \cup \ldots \cup X_k, \{e_1, \ldots, e_k\}, \\ & \quad R_1 \cup \ldots \cup R_k, X_{\circ,1} \cup \ldots \cup X_{\circ,k}) \\ & \text{with } (X_i, e_i, R_i, X_{\circ,i}) = \gamma_{cp}\; cp_i \text{ for } i = 1, \ldots, k\end{aligned}$$

Function γ_{cp} yields for a context pattern $cp = sqs\; cq\; tp$ a set of variables, a start expression, a set of rules and a set of target variables as follows: First a forest expression describing the structure qualifiers *sqs*, then two regular expressions r_l and r_r for the right and left context given by *cq* are generated. The tree pattern *tp* yields an alternative r_t of variables[4] and the set of target variables. Arguments $(true, \perp)$ to the γ_{tp} call indicate that *tp* contributes to the target variables and that *tp* is not followed by some path pattern *pp* within an enclosing path pattern (see the comments on functions γ_{tp} and γ_{pp} below). The three regular expressions are then combined into a single one and added to the forest expression generated for *sqs*.

[4]By an alternative of variables $x_1, \ldots, x_k$ we mean the regular expression $x_1 \mid \ldots \mid x_k$.

$$\begin{aligned}\gamma_{cp}(sqs\ cq\ tp) = (X_1 \cup X_2 \cup X_3, e \sqcap r_l r_t r_r, R_1 \cup R_2 \cup R_3, X_\circ)\\ \text{with} \quad (X_1, R_1, e) &= \gamma_{sqs}\ sqs\\ (X_2, R_2, r_l, r_r) &= \gamma_{cq}\ cq\\ (X_3, R_3, X_\circ, r_t) &= \gamma_{tp}\ (true, \perp)\ tp\end{aligned}$$

For a sequence *sqs* of structure qualifiers, the function γ_{sqs} yields a set of variables, a set of rules for these variables and a forest expression corresponding to *sqs*: For each structure qualifier σfp, a regular expression is generated and added to the forest expression, signed with σ:

$$\begin{aligned}\gamma_{sqs}([\,\sigma_1 fp_1\,]\ldots[\,\sigma_k fp_k\,]) = (X_1 \cup \ldots \cup X_k, R_1 \cup \ldots \cup R_k, \sigma_1 r_1 \sqcap \ldots \sqcap \sigma_k r_k)\\ \text{with } (X_i, R_i, r_i) = \gamma_{fp}\ fp_i \text{ for } i = 1, \ldots, k\end{aligned}$$

Note that γ_{sqs} generates no target variables since a structure qualifier does not contribute to the possible matches of a pattern.

Similarly to γ_{sqs}, function γ_{cq} generates for a context qualifier $[\,fp_l \# fp_r\,]$, two regular expressions corresponding to the two forest patterns. If the context qualifier is empty, _ is used as default; in this case no variables and rules are generated:

$$\begin{aligned}\gamma_{cq}\ \epsilon &= (\emptyset, \emptyset, \_, \_)\\ \gamma_{cq}\ [\,fp_l \# fp_r\,] &= (X_1 \cup X_2, R_1 \cup R_2, r_l, r_r)\\ &\quad \text{with } (X_1, R_1, r_l) = \gamma_{fp}\ fp_l\\ &\quad\phantom{\text{with }} (X_2, R_2, r_r) = \gamma_{fp}\ fp_r\end{aligned}$$

A forest pattern *fp* is translated to a regular expression by substituting each tree pattern *tp* occurring in *fp* by the alternative of variables returned for *tp* by γ_{tp}. A forest pattern can not generate target variables; it therefore calls γ_{tp} with $(false, \perp)$ as first argument; the set of target variables generated by γ_{tp} is then empty and can safely be ignored (see below).

$$\begin{aligned}\gamma_{fp}\ fp = (X_1 \cup \ldots \cup X_k, R_1 \cup \ldots \cup R_k, fp_{[tp_1/r_1, \ldots, tp_k/r_k]})\\ \text{where } \{tp_1, \ldots, tp_k\} \text{ is the set of tree patterns in } fp \text{ and}\\ (X_i, R_i, \emptyset, r_i) = \gamma_{tp}\ (false, \perp)\ tp_i\end{aligned}$$

For a tree pattern */pp* or *//pp*, first the path pattern *pp* is translated, yielding an alternative of variables and a set of target variables. If *tp* starts with /, then this is already the result for *tp*. For //, a new variable *x* must be added, representing a tree that has a subtree matching *pp*, i.e., either of the variables in the alternative; the new variable is not a target variable.

$$\begin{aligned}\gamma_{tp}\ (isCxt, cont)\ /pp &= \gamma_{pp}\ (isCxt, cont)\ pp\\ \gamma_{tp}\ (isCxt, cont)\ /\!/pp &= (X \cup \{x\}, R \cup \{x \to \texttt{<*>}\ \_\,(x \mid r)\,\_\}, X_\circ, x \mid r)\\ &\quad \text{where } (X, R, X_\circ, r) = \gamma_{pp}\ (isCxt, cont)\ pp\\ &\quad \text{and } x \text{ is a new variable.}\end{aligned}$$

γ_{tp} has an additional argument: a pair $(isCxt, cont)$. If *isCxt* is false, then this tree pattern occurs within a forest pattern. In this case it is part of a structure or context qualifier and may not generate target variables. The parameter *cont* is either $\perp$ or a regular expression, more precisely an alternative of variables, which represents the *continuation* of an enclosing path pattern. This becomes clearer by explaining function γ_{pp}, which also has this additional argument and translates a path pattern *pp*. If *pp* is a node pattern *np*, a new variable x is reserved for *np*, and a set of rules, possibly together with auxiliary variables, is generated for x by a call to function γ_{np}. If *isCxt* is *true* and there is no continuation, then x is a target variable:

$$\gamma_{pp}\ (\mathit{isCxt}, \mathit{cont})\ np = (X \cup \{x\}, R, X_\circ, x)$$

$$\text{where } x \text{ is a new variable,}\quad (X, R) = \gamma_{np}\ (x, \mathit{cont})\ np, \text{ and}\quad X_\circ = \begin{cases} \{x\} & \text{if } \mathit{isCxt} = \mathit{true} \text{ and } \mathit{cont} = \bot \\ \emptyset & \text{otherwise} \end{cases}$$

For a disjunction of two path patterns, both are translated separately and the results are united:

$$\gamma_{pp}\ (\mathit{isCxt}, \mathit{cont})\ (pp_1 \,\|\, pp_2) = (X_1 \cup X_2, R_1 \cup R_2, X_{\circ,1} \cup X_{\circ,2}, r_1 \,|\, r_2)$$

$$\text{with } (X_1, R_1, X_{\circ,1}, r_1) = \gamma_{pp}\ (\mathit{isCxt}, \mathit{cont})\ pp_1 \quad (X_2, R_2, X_{\circ,2}, r_2) = \gamma_{pp}\ (\mathit{isCxt}, \mathit{cont})\ pp_2$$

For a path pattern *pp tp*, first *tp* is translated by γ_{tp}. The resulting alternative of variables r_1 is the continuation for the translation of *pp*. Because *pp* is nested into a path pattern, it does not generate target variables. Instead it must be continued with a rule that demands for a child fulfilling r_1 (see the comments on γ_{np} below).

$$\gamma_{pp}\ (\mathit{isCxt}, \mathit{cont})\ (pp\ tp) = (X_1 \cup X_2, R_1 \cup R_2, X_\circ, r_2)$$

$$\text{with } (X_1, R_1, X_\circ, r_1) = \gamma_{tp}\ (\mathit{isCxt}, \mathit{cont})\ tp \quad (X_2, R_2, \emptyset, r_2) = \gamma_{pp}\ (\mathit{isCxt}, r_1)\ pp$$

The most important function – the one which generates the major part of the variables in the output grammar – is γ_{np}. It generates for a variable x and a node pattern *np* a set of rules for x implementing *np*. It also generates a (possibly empty) set of auxiliary variables and appropriate rules.

The first case is that *np* is a text pattern "τ": If there is no continuation, then x has a single text rule. Otherwise x has no rules, because a text node can never have children; thus no tree can fulfill both *np* and the continuation. This case should not occur in practice because a pattern of this form, such as `"forest"/a`, makes no sense; the syntax of patterns, however, allows it.

$$\gamma_{np}\ (x, \bot)\ "\tau" = (\emptyset, \{x \to "\tau"\})$$
$$\gamma_{np}\ (x, r\ )\ "\tau" = (\emptyset, \emptyset)$$

The next case is that *np* has the form `<?`τ`?>` *sqs*: First the structure qualifiers *sqs* are translated into a forest expression e_1. If there is a continuation r then this forest expression is extended by a regular expression $\_\,r\,\_$, and a processing instruction rule is generated:

$$\gamma_{np}\ (x, \mathit{cont})\ \texttt{<?}\tau\texttt{?>}\ sqs = (X, R \cup \{x \to \texttt{<?}\tau\texttt{?>}\ e_2\})$$

$$\text{where } (X, R, e_1) = \gamma_{sqs}\ sqs, \text{ and}\quad e_2 = \begin{cases} \_ & \text{if } \mathit{cont} = \bot \text{ and } e_1 = \epsilon \\ e_1 & \text{if } \mathit{cont} = \bot \text{ and } e_1 \neq \epsilon \\ e_1 \sqcap \_\,r\,\_ & \text{if } \mathit{cont} = r \end{cases}$$

A similar case is when *np* has the form *nt aqs sqs cq*. Here, additionally the node test *nt* is translated to the grammar syntax by a call to γ_{nt}, and the attribute qualifiers are incorporated into the generated rule for x. Moreover, the continuation might be constrained by a context qualifier which is translated into a pair of regular expressions.

$$\begin{aligned}
&\gamma_{np}\ (x{,}cont)\ nt\ aqs\ sqs\ cq = \\
&\qquad (X_1 \cup X_2, R_1 \cup R_2 \cup \{x \to \texttt{<}\gamma_{nt}\ nt\ \sigma_1 ap_1 \ldots \sigma_k ap_k\texttt{>}\ e\}) \\
&\qquad \text{where } aqs = [\,\sigma_1@ap_1\,]\ldots[\,\sigma_k@ap_k\,], \\
&\qquad\qquad (X_1, R_1, e_1) = \gamma_{sqs}\ sqs, \\
&\qquad\qquad (X_2, R_2, r_l, r_r) = \gamma_{cq}\ cq \\
&\qquad\qquad e = \begin{cases} \_ & \text{if } e_1 = \epsilon, cont = \bot \text{ and } r_l = r_r = \_ \\ e_1 & \text{if } e_1 \neq \epsilon, cont = \bot \text{ and } r_l = r_r = \_ \\ e_1 \sqcap r_l x_\top r_r & \text{if } cont = \bot \text{ and } r_l \neq \_ \text{ or } r_r \neq \_ \\ e_1 \sqcap r_l r r_r & \text{if } cont = r \neq \bot \end{cases}
\end{aligned}$$

$$\begin{aligned}
&\gamma_{nt}\ * &&= * \\
&\gamma_{nt}\ a &&= a \\
&\gamma_{nt}\ \texttt{<}a_1 \,|\, \ldots \,|\, a_k\texttt{>} &&= a_1 \,|\, \ldots \,|\, a_k \\
&\gamma_{nt}\ \texttt{<}\neg a_1 \,|\, \ldots \,|\, a_k\texttt{>} &&= \neg a_1 \,|\, \ldots \,|\, a_k
\end{aligned}$$

The last case is when *np* has the form . *aqs sqs cq*. This node pattern comprises all of the three other cases: It matches an arbitrary text node, processing instruction or element. However, a processing instruction can only match if no attribute qualifiers are present; for a text node there must be no structure or context qualifier and no continuation either.

$$\begin{aligned}
&\gamma_{np}\ (x, cont)\ .\ aqs\ sqs\ cq = (X_1 \cup X_2, R_1 \cup R_2 \cup R_3 \cup R_4 \cup R_5) \\
&\qquad \text{where } aqs = [\,\sigma_1@ap_1\,]\ldots[\,\sigma_k@ap_k\,], \\
&\qquad\qquad (X_1, R_1, e_1) = \gamma_{sqs}\ sqs, \\
&\qquad\qquad (X_2, R_2, r_l, r_r) = \gamma_{cq}\ cq \\
&\qquad\qquad e = \begin{cases} \_ & \text{if } e_1 = \epsilon, cont = \bot \text{ and } r_l = r_r = \_ \\ e_1 & \text{if } e_1 \neq \epsilon, cont = \bot \text{ and } r_l = r_r = \_ \\ e_1 \sqcap r_l x_\top r_r & \text{if } cont = \bot \text{ and } r_l \neq \_ \text{ or } r_r \neq \_ \\ e_1 \sqcap r_l r r_r & \text{if } cont = r \neq \bot \end{cases} \\
&\qquad\qquad R_3 = \{x \to \texttt{<}*\ \sigma_1 ap_1 \ldots \sigma_k ap_k\texttt{>}\ e\}) \\
&\qquad\qquad R_4 = \begin{cases} \{x \to \texttt{<??>}\ e\} & \text{if } aqs = \epsilon \\ \emptyset & \text{otherwise} \end{cases} \\
&\qquad\qquad R_5 = \begin{cases} \{x \to \text{".*"}\} & \text{if } aqs = sqs = cq = \epsilon \text{ and } cont = \bot \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

Example 8.1: Consider pattern //`prod[#_(rhs/nt[@def="Char"])_]/lhs/""` from 8.2.2. This pattern is translated by γ_p to the query grammar $((\{x_\top, x_1, \ldots, x_6\}, \_\,(x_5 \,|\, x_6)\,\_\,, R), \{x_1\})$ with the following rules:

$$\begin{array}{lll}
x_\top \to \text{""} & x_1 \to \text{""} & x_4 \to \texttt{<rhs>}\ \_\,x_3\,\_ \\
x_\top \to \texttt{<??>}\ \_ & x_2 \to \texttt{<lhs>}\ \_\,x_1\,\_ & x_5 \to \texttt{<prod>}\ x_2\,\_\,x_4\,\_ \\
x_\top \to \texttt{<*>}\ \_ & x_3 \to \texttt{<nt def="Char">}\ \_ & x_6 \to \texttt{<*>}\ \_\,(x_5 \,|\, x_6)\,\_
\end{array}$$

We can further translate this grammar to an extended context grammar; instead of listing that grammar here, let us only mention that it has 22 variables with 26 rules. The growth in size is caused by the generation of two auxiliary variables for each right-hand side corresponding to an element: one describing its attributes (`#atts`) one describing its content (`#content`). □

Example 8.2: Let us consider another example pattern from 8.2.2: The pattern $/\!/{*}[\,\_\,($`SPEECH`$/\!/$"`hurlyburly`"$)\#\_\,]/$`SPEECH`$/$`SPEAKER` is translated by γ_p to the grammar $((\{x_\top, x_1, \ldots, x_7\},\ \_\,(x_6 \mid x_7)\,\_\,, R), x_1)$. It has the following rules (the rules for $x_\top$ are omitted):

$x_1 \longrightarrow$ `<SPEAKER>` $\_$	$x_5 \longrightarrow$ `<SPEECH>` $\_\,(x_3 \mid x_4)\,\_$
$x_2 \longrightarrow$ `<SPEECH>` $\_\,x_1\,\_$	$x_6 \longrightarrow$ `<*>` $\_\,x_5 x_2\,\_$
$x_3 \longrightarrow$ "`hurlyburly`"	$x_7 \longrightarrow$ `<*>` $\_\,(x_6 \mid x_7)\,\_$
$x_4 \longrightarrow$ `<*>` $\_\,(x_3 \mid x_4)\,\_$	

In contrast to the previous example, this grammar is right-ignoring and can be implemented by a single run of a DLFA. □

8.2.4 Comparison with Other Querying Languages

In this section we compare our pattern language to other query languages for tree-structured data. We concentrate on four topics: The pattern matching language of TRAFOLA, the querying language of *sgrep*, the W3C query language XPATH and a number of database querying languages.

TRAFOLA

Long before the rise of XML, a query language on tree-like data structures was employed in the non-deterministic functional programming language TRAFOLA [HS93], which supports trees as one of its basic data structures. Designed, among others, for complex tree transformations in the context of program optimization in compilers, it has – for a general purpose programming language – a sophisticated querying mechanism. The pattern matching process in TRAFOLA decomposes a tree t by deleting the subtree t' that matches the pattern and replacing it with a hole `@`. Moreover it binds the variables occurring in the pattern to subtrees of t'. Pattern matching happens in the head of function clauses; the body of the clause can then manipulate the variables bound by the pattern. For instance, consider the following example from [Fec94]:

```
dec simp = { a ^ ('add(0,e)|'add(e,0)) => a ^ e
           # a ^ ('mul(1,e)|'mul(e,1)) => a ^ e }
```

The function `simp` finds occurrences of `add` nodes one of whose children is labeled `0` using the pattern `('add(0,e)|'add(e,0))`. E.g., for the tree `mul(3,add(1,0))`, the pattern binds `a` to `mul(3,@)` and `e` to `1`. In the body, the operator `^` composes the new tree `mul(3,1)` from `a` and `e`.

The TRAFOLA pattern language offers conjunction, alternatives, negation and even non-linear patterns. However, the only operator that can select subtrees at a non-constant depth is `^`. This operator identifies arbitrary subtrees of t, similarly to our $/\!/$ operator. Though TRAFOLA trees are non-ranked, the patterns are committed to a fixed rank. There is no means of imposing a regular condition on the siblings of a node. TRAFOLA thus can not express regular conditions with its pattern language.

sgrep

A popular tool for querying structured text is *sgrep* [JK96a, JK96b]. It is a text-based pattern-matcher and unaware of the tree structure of a document. Instead it implements an algebra of *region sets*. A region is a fragment of the input text; sets of regions can be manipulated by operations of the algebra.

A region set is generated from the input document by matching a constant string; the result is the set of all occurrences of the string in the document. Using operator "`..`" (or its slightly different variant "`__`") two regions can be combined into a single region incorporating all text enclosed between the two regions. Other operations on region sets include union, intersection and difference as well as filtering by, e.g., containment conditions. As an example, our pattern //`SPEECH`[_ (//`LINE`/"`hurlyburly`") _]/`SPEAKER`/. is expressed in *sgrep* as:

```
("<SPEAKER>"__"</SPEAKER>") in
  ("<SPEECH>".."</SPEECH>" containing
    ("<LINE>".."</LINE>" containing "hurlyburly"))
```

Compared to our pattern, this *sgrep* pattern is still less precise: It would also match if the `SPEAKER` element were not a direct descendant of the `SPEECH` element. Though the above *sgrep* pattern can be refined to express the pattern precisely, it would become very complex.

Another weak point of *sgrep* is that only exact occurrences of a string can be matched in order to define a basic region set. Therefore the query can not abstract from XML syntax details: It must account for all possible ways of writing, e.g., a start-tag. In order to ease handling of the XML syntax, *sgrep* provides a macro mechanism which allows for more intelligible specification of queries.

sgrep is very fast. However, it can not express regular conditions, neither on the ancestors nor on the siblings of a node. Moreover, it is completely unaware of the document's tree structure; the result of querying is always a string. Though well-suited for fast extraction of plain text from a document, *sgrep* is therefore absolutely unsuited for selecting subdocuments for further processing.

Database Querying Languages

Querying languages are a well-studied subject in the area of database systems. A database typically stores large amounts of flat, i.e., non-hierarchically structured data and provides a query language for extracting data. Only recently concepts for the representation of structured documents in databases have been developed; documents are then modeled as *semi-structured data*semi-structured data [Abi97]. Database query languages for semi-structured data include *Lorel* [AQM$^+$97, GMW99] and YAT$_L$ [CDS$^+$98]. Specifically for XML, among others, XQL [Rob99] and XML-QL [DFF$^+$99] have been developed. A practical comparison of these languages is given in [FSW99].

Usually, the functionality of database query languages exceeds by far that of pure pattern matching: They provide features like sorting, filtering and restructuring. Moreover, database querying languages for semi-structured data often allow construction of new data from the data extracted by the query, or transformation of the obtained data. An overview of the requirements on XML query languages from a database point of view is given by [Mai98].

In contrast to these languages, our pattern language is solely designed for *location* of subtrees; yet filtering can still be easily incorporated through external predicates. Sorting, restructuring and construction of new data, however, can not be integrated. These tasks have to be completed by querying application.

XPATH

The most prominent query language for XML is XPATH [W3C99c], which is used both by XSLT [W3C99b] for selecting subtrees to be transformed or formatted, and by XPOINTER [W3C99f] for identifying subdocuments referenced by links in a hypertext language such as HTML.

Since the syntax of our pattern language is inspired by XPATH, both are very similar. However, the operational model of XPATH is completely different from ours. While our approach is to locate all matches in one or two traversals of the document tree, XPATH might need as many as the size of the pattern. Let us illustrate this by an example. The XPATH pattern `a[@x="1"]//b` is evaluated as follows:

1. All elements with type `a` are collected in a *node-set* S_1.
2. The qualifier `[@x="1"]` selects a node-set $S_2 \subseteq S_1$ containing all nodes from S_1 that fulfill the expression `@x="1"`. Note that, instead of interpreting this expression as a structural condition, it is evaluated as an expression on the abstract data type of nodes.
3. Operator // generates the node-set S_3 of all descendants of nodes in S_2.
4. The set S_4 of all elements with type `b` is filtered from S_3.

A qualifier or a node test can thus be viewed as a filter predicate. XPATH allows predicates that exceed the capabilities of our pattern language. For instance, the following features are offered:

Arithmetic Expressions: All nodes in a node set are numbered in document order. The predicate `[position() mod 2=0]` selects every other node in a node-set. A predicate may evaluate an arbitrary arithmetic expression on this position. It is questionable whether this generality is of any use in practice.

Navigation: A predicate may navigate arbitrarily through the document tree. For instance, the predicate `[ancestor::a]` selects all nodes from a node-set that have an ancestor of element type `a`. This concept of arbitrary navigation requires that the document tree is explicitly present; locating of matches during parsing is not possible.

On the one hand, these features are not offered by our pattern language, because they can only be implemented if we drop the goal of matching in at most two passes.

On the other hand, our pattern language has structure and context qualifiers. While XPATH has no context qualifiers at all, it provides a weak version of structure qualifiers, which relates only to a single child of the concerned node. Expressed in the syntax of our structure qualifiers, which are arbitrary regular expressions over tree patterns, XPATH only allows the form $[\ \_\, tp\, \_\ ]$.

Both XPATH and our pattern language lack an iteration operator that can express a regular condition on a path; they only provide operator // which selects arbitrary descendants. However, in contrast to XPATH we offer query grammars as an alternative syntax with full regular expressiveness.

A valuable advantage of our pattern language is its clearly defined semantics, which is given precisely by a translation scheme to query grammars. The specification of XPATH does not give a formal semantics of patterns: Their meaning is described informally, partly by means of examples only. However, a formal semantics for a subset of an early draft version of XPATH is given in [Wad99].

Chapter 9

Implementation in SML

In this chapter we present *fxgrep* [Fxg99], an XML querying tool based on the algorithm from the previous chapters. *fxgrep* reads a grammar or pattern and locates in an XML document all matches of that grammar or pattern. *fxgrep* can be used through a command-line interface similar to the line-oriented text search tool *grep*.

fxgrep is implemented in SML, based upon the XML parser *fxp*. We do not present all implementation details here. Instead we concentrate on two important issues, both of which deal with efficient representation of possibly huge transition tables. On the one hand, we describe the implementation of text patterns, which must be able to deal with the full range of more than one million UNICODE characters. On the other hand, we present a demand-driven way of computing the transition tables of the forest automata employed for the matching procedure.

9.1 System Architecture and Implementation

Figure 9.1 shows the system architecture of *fxgrep*. It consists basically of three stages: the frontend, the preprocessing stage and the matcher. The collector is the querying application to which all matches are reported; this is a function which accumulates all matches in some data structure. Though the collector is hard-coded in *fxgrep*, it can be replaced by virtually any function.

fxgrep consists of about 8000 lines of SML source code, of which 2000 are comments and 3000 are generated. 4000 lines, i.e., half of the source code is used by the frontend. The analyzing and preprocessing stage has about 500 lines, and 2000 lines constitute the matcher itself. The rest of the code is for the command-line interface, debugging output and error reporting.

9.1.1 The Frontend

fxgrep supports two different forms of specifying a query: either as a query grammar or as pattern. In the first case it is parsed by the grammar parser and then translated into an extended context grammar by the grammar translator. If the input is a pattern, then it is parsed by the pattern parser and translated to a query grammar by the pattern translator before feeding it to the grammar translator.

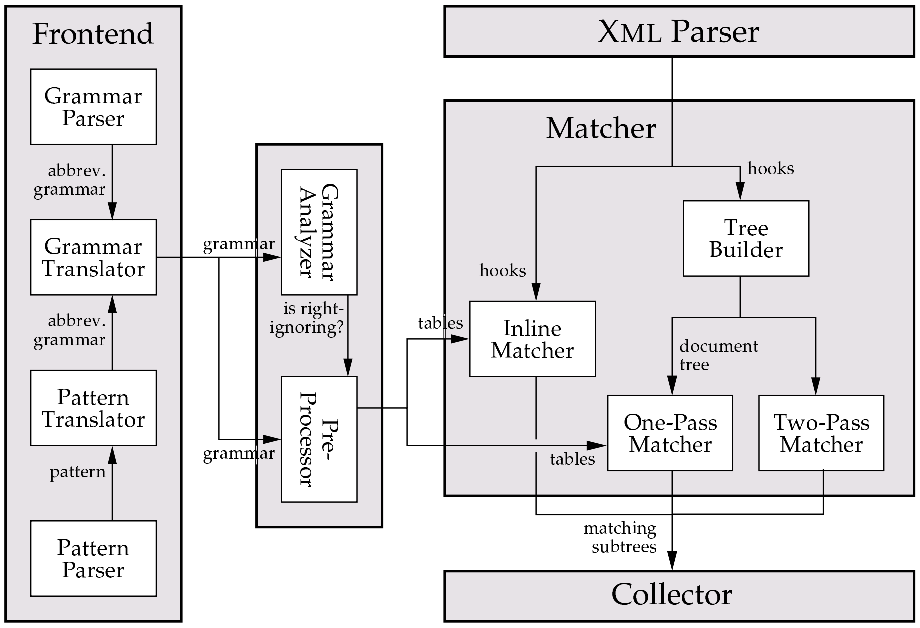


Figure 9.1: The system architecture of *fxgrep*

The Grammar Parser

It is a common practice to specify all input to an XML processor as an XML document. Examples include XSL style sheets [W3C99e] and XML catalogs [Cow99]. *fxgrep* adopts this convention by allowing specification of grammars as an XML document instance. The advantage is that the XML syntax can be used for specifying, e.g., non-printable UNICODE characters, in a platform-independent way. An XML parser can be used as a generic frontend for parsing the specification.

The grammar parser reads an XML document and interprets it as a query grammar. Though this document is by default parsed in non-validating mode, it should be according to the following DTD:

XML Example 22

```
<!ELEMENT grammar (rule|start|targets)*>
<!ELEMENT rule (#PCDATA)>
<!ELEMENT start (#PCDATA)>
<!ELEMENT targets (#PCDATA)>
<!ATTLIST rule var NMTOKEN #REQUIRED>
```

The content of a `rule` element, e.g., specifies a right-hand side for the variable given as its `var` attribute. Similarly, a `targets` element contains a list of target variables, and a `start` element specifies a start expression. A simple example for a grammar is the following:

XML Example 23

```
<grammar>
  <rule var=  "x"  ><![CDATA[  <*> _ x _  ]]></rule>
  <rule var=  "x"  ><![CDATA[  <a> _      ]]></rule>
  <targets>                    x            </targets>
  <start>                      _ x _        </start>
</grammar>
```

In order to parse a grammar, a set of hooks is defined that collects the character data contained in the three relevant element types and ignores all other information. In order to parse, e.g., a right-hand side given as the character data of a `rule` element, this data is first tokenized by a lexer. The lexer is hand-coded because ML-LEX [AMT94], the lexer generator for SML, has no UNICODE support (cf. 2.7.1). The stream of tokens is fed into an ML-YACC-generated parser [AT94] which produces an SML data structure representing the query grammar.

Since the grammar syntax employs the characters "<" and ">", specifying a grammar as an XML document usually requires the use of CDATA sections (cf. 1.1.4). The above example shows that this can make the grammar hardly intelligible. We therefore allow as an alternative specification of the grammar as a UNICODE file that is not parsed as an XML document. However, the encoding auto-detection facilities of the XML parser (cf. 2.3.7) can not be used for such input. It must therefore be in one of the standard encodings UTF-16 or UTF-8 which can be distinguished without the need for an encoding declaration. The above example can thus be specified as follows:

XML Example 24

```
TARGETS
  x
START
  _ x _
RULES
  x -> <*> _ x _
  x -> <a> _
```

This alternative syntax is tokenized by a hand-written lexer, and analyzed by a parser generated with ML-YACC. The specification has 30 nonterminals with 70 rules; the generated parser has 107 states and uses about 35 KB of SML source code.

For text patterns occurring in the grammar, a different tokenization is required. The lexer therefore returns text patterns, which are always enclosed between quotes or "<?" and "?>", as a whole, without tokenizing. In order to parse a text pattern, it is tokenized and analyzed by a separate lexer and parser. The ML-YACC specification for text patterns has 9 nonterminals and 53 rules; the generated parser has 72 states and is 28 KB large.

The Pattern Parser

The second possibility of formulating a query is in pattern syntax. Since a pattern does not involve multiple rules, it can be specified by a single UNICODE string, usually given as a command-line argument. This string is tokenized and fed into an ML-YACC-generated parser. The specification for this parser has 29 nonterminals and 86 rules. The generated parser has 155 states and consists of 49 KB of SML code. Note that it is a non-trivial task to specify the pattern grammar in Figure 8.2 as an LALR-1 grammar, as required by ML-YACC.

The Pattern Translator

The pattern translator takes a pattern produced by the pattern parser and generates a query grammar. It is basically an implementation of the γ_p function defined in 8.2.3. However, the implementation avoids repeated generation of variables for multiply occurring tree patterns. Interestingly enough, this module consists of less than 200 lines of SML code, which are extremely few for this fairly complex function. This demonstrates the high suitability of SML for the manipulation of tree-structured data – in this case the syntax trees of patterns.

The Grammar Translator

The grammar translator performs the translation of a query grammar to an extended context grammar as described in 8.1, except that it preserves element-type patterns. Expansion of element-type patterns is expensive because it can generate a large number of rules for a single element-type pattern. Moreover, this expansion is usually impossible because the DTD of the input document is not known in advance; if the document is parsed in non-validating mode, it need not even have a DTD.

While processing the grammar, the translator additionally performs the Berry-Sethi construction for each regular expression occurring on a right-hand side of a rule. If possible, it identifies multiple occurrences of the same regular expression in order to avoid redundant generation of Berry-Sethi automata, and to keep the number of NFA states small.

The grammar translator consists of 150 lines of source code, plus an additional number of 120 lines for the Berry-Sethi construction.

9.1.2 The Preprocessing Stage

The preprocessing stage has two tasks: It analyzes the grammar for being right-ignoring and computes a number of tables supporting the matcher in computing transitions.

The Grammar Analyzer

The grammar analyzer determines whether a grammar is right-ignoring, i.e., whether it can be matched by a single pass through the document. In order to do so it employs an algorithm derived from the one given in Section 7.4. It is implemented in about 150 lines of SML code.

The Preprocessor

Matching of a pattern is performed by one or two runs of a deterministic push-down forest automaton on the input document. However, the automata might be very large: Although only a small number of states do actually occur during a run of the automaton, computation of all reachable states is generally expensive. Moreover, in two-pass matching the start state of the second automaton depends on the output state of the first pass; since computation of reachable states is initiated with that start state, it would be dependent on the input document, which is certainly not our intention.

Therefore the transition tables of the automata are not constructed in advance. Instead all transitions are computed *on demand* as soon as they are required during the run of the automaton. In order to keep this computation as cheap as possible, the preprocessor stores in a bunch of tables all information which can be derived directly from the grammar and which is shared between the computations of transitions. For instance, the down-transition of $A_G^{\vec{}}$ is defined as follows:

$$\mathit{Down}_a\, q = \{y_{0,j} \mid y \in q,\ (y,x,y_1) \in \delta,\ x \to a\langle \ldots \sqcap \sigma r_j \sqcap \ldots \rangle\}$$

In order to compute $\mathit{Down}_a\, q$, the matcher must determine the set $y_0\mathit{s\_for\_y}\ y = \{y_{0,j} \mid (y,x,y_1) \in \delta,\ x \to a\langle \ldots \sqcap \sigma r_j \sqcap \ldots \rangle\}$ for each $y \in q$. Since y probably occurs in more than one forest state, this information will be needed several times for the computation of a down-transition. It is therefore sensible to precompute this set for all $y \in Y$. This is the task of the preprocessor. For down-transitions in $A_G^{\vec{}}$, it computes the following information:

$$\begin{array}{l} y_0\mathit{s\_for\_y}\ y = \{y_{0,j} \mid (y,x,y_1) \in \delta,\ x \to a\langle \ldots \sqcap \sigma r_j \sqcap \ldots \rangle\} \text{ for all } y \in Y \\ y_0\mathit{s\_for\_a}\ a = \{y_{0,j} \mid x \to a\langle \ldots \sqcap \sigma r_j \sqcap \ldots \rangle\} \text{ for all } a \text{ occurring in } G \\ \mathit{other\_y_0s} = \{y_{0,j} \mid x \to a\langle \ldots \sqcap \sigma r_j \sqcap \ldots \rangle,\ a \text{ does not occur in } G\} \end{array}$$

This also illustrates how element-type patterns are handled: For each element type a that occurs in G, the information about all rules concerning a is stored directly in a table. *fxgrep*'s frontend reserves indices for these element types in the DTD when the grammar or pattern is parsed. Because this happens prior to parsing the input document, it ensures that these element types contiguously occupy the lowest indices in the DTD tables. The relevant information can therefore be stored in a small table.

If an element type does not occur in the query, there can nonetheless be concerning rules, namely all rules which have a negated element-type pattern. Information about these rules is stored in single value; it must always be considered if the element type in question is beyond the range of the table for the known element types.

With the help of the precomputed information, a down-transition for a state q of size n can now be computed by exactly n set operations, namely one intersection and $(n-1)$ unions:

$$\begin{array}{ll} \mathit{Down}_a\, q = y_0\mathit{s\_for\_a}\ a \cap \bigcup_{y \in q} y_0\mathit{s\_for\_y}\ y, & \text{if } a \text{ occurs in } G \\ \mathit{Down}_a\, q = \mathit{other\_y_0s} \cap \bigcup_{y \in q} y_0\mathit{s\_for\_y}\ y, & \text{if } a \text{ does not occur in } G \end{array}$$

For the other transitions the preprocessor computes similar information; in case of two-pass matching, this information must facilitate transitions both in $A_G^{\overleftarrow{}}$ and $B_G^{\vec{}}$.

9.1.3 The Collector

The collector is a module to which all matches of the query are reported by a call to its function reportMatch. In *fxgrep*, this module either prints the matching subtree onto the screen or simply counts the matches. Nonetheless, the matcher is implemented in a way that allows for replacement of the collector with an arbitrary structure fulfilling the following signature:

```
signature MatchReport =
  sig
    type Report

    val null   : Report
    val report : DocDtd.Dtd → MatchData.Match ∗ Report → Report
  end
```

This is similar to the principle of hooks in *fxp*: The collector accumulates all matches in a value of type Report. It must define a start value null of that type and a function report for incorporating one match. All matches of the query are then reported through this function in document order.

9.1.4 The Matcher

The matcher is the part of *fxgrep* which actually locates matches of the query by one or two runs of a forest automaton. First the input is parsed by the tree builder and the document tree is constructed in memory. Then the matches of the query are located either by the two-pass matcher or, if the grammar is right-ignoring, by the one-pass matcher. In this case construction of the document tree can be avoided by using the inline matcher which incorporates the transitions of $A_{\vec{G}}$ into the hooks.

The Tree Builder

If the matching requires two passes, then the input document is parsed and converted to a tree representation prior to matching. In order to parse the document, an instance of the *fxp* parser similar to the tree-constructing parser in 2.8.6 is used.

The Two-Pass Matcher

The two-pass matcher locates all matches of a grammar G in a run of $A_{\overleftarrow{G}}$ followed by a run of $B_{\vec{G}}$. During the run of $A_{\overleftarrow{G}}$, the labeling produced by this automaton is constructed as a tree data structure. The run of $B_{\vec{G}}$ then traverses the input document and the labeling simultaneously. It detects a match of the grammar at a subtree as soon as it enters this subtree. Since the tree is completely available as a data structure, the matching subtree can be reported immediately to the collector.

The forest states and tree states of these automata are sets of variables and NFA states. Each time a transition is performed, it must be computed with the help of the information compiled by the preprocessor. However, the structure of XML documents is usually constrained by a DTD. As a consequence, there are many subdocuments with a similar logical structure. A run of $A_{\overleftarrow{G}}$ or

$B_{\vec{G}}$ typically involves the same states and transitions during traversal of these subtrees. The same transition is thus performed multiple times during matching.

In order to avoid repeated computation of a single transition, the matcher stores all computed transitions in a dictionary. For efficient management of the dictionary, the forest states and tree states are hashed to integers, and so are the occurring element types and attribute names. In order to find out whether, e.g., a transition $Down_a\ q$ was already computed, the lookup in the dictionary requires only comparison of pairs of integers. Only if the transition was not computed yet, the set of NFA states in q must be considered.

Moreover, tabulation is also performed for intermediate results in the computation of transitions. The reason is that, for a forest state q, down-transitions with different symbols a and b may occur. Now, according to 9.1.2,

$$\begin{aligned} Down_a\ q &= y_0s\_for\_a\ a \cap \bigcup_{y \in q} y_0s\_for\_y\ y, \text{ and} \\ Down_b\ q &= y_0s\_for\_a\ b \cap \bigcup_{y \in q} y_0s\_for\_y\ y \end{aligned}$$

Thus, the subexpression $\bigcup_{y \in q} y_0s\_for\_y\ y$ is needed by all down-transitions for q. It is therefore sensible to store its value once it is computed; subsequent computations of $Down$ transitions for q then only require a single set intersection operation.

Summarizing, demand driven computation of transitions has the following advantages:

- ◇ The possibly exponentially large transition tables need not be computed in advance;
- ◇ Transitions are only computed on demand; transitions which are not actually required for traversal of the input document are ignored.
- ◇ The number of transitions that are actually computed is at most linear in the size of the document. For small documents only few transitions are necessary; due to the logical structure of XML documents, this holds even for the major part of large documents.

The One-Pass Matcher

The one-pass matcher locates all matches of a right-ignoring grammar in a single run of $A_{\vec{G}}$ through the input document. In contrast to the two-pass algorithm, a match of the query can not be detected when arriving at that subtree t: In order to verify the structural condition, t must first be traversed; only when the automaton exits t it can decide whether it is a match. If a subtree of t also matches the query, then this match is detected earlier than the match of t: The matches of the query are not found in document order. Therefore, a match may only be immediately reported if none of its ancestors can match, i.e., if the left upper context was fulfilled for none of its ancestors. Otherwise, reporting must be delayed until that ancestor is completely traversed.

In order to implement this delay mechanism, the one-pass matcher determines whether the left upper context is fulfilled before it descends to the children of a node. This is the case whenever the current forest state contains a y

for which a transition with a target variable is possible, i.e., if $y \in \mathit{could\_match}$ with

$$\mathit{could\_match} = \{y \mid (y, x, y_1) \in \delta \text{ for some } x \in X_\circ\}$$

This set is precomputed by the preprocessor such that a single set intersection with the current forest state suffices for determining whether the left upper context is fulfilled. In this case, the matcher collects all matches within the node's children instead of reporting them; this introduces an overhead into the matching procedure. Nonetheless, one-pass matching is still more efficient than performing two traversals.

The Inline Matcher

Since the traversing order of $A_{\vec{G}}$ is the same as that of the XML parser, matching of a right-ignoring grammar can even be performed during parsing. This is done by the inline matcher, which incorporates the transitions of $A_{\vec{G}}$ into the hooks used for parsing. In addition to delaying the reporting of matches, the inline matcher must also perform construction of the document tree. This, however, is only necessary for the matching subtrees because only these are reported to the collector. The decision whether to construct a subtree must be taken when entering that subtree; at this time it is not yet known, whether the structural condition is met. Construction of the document tree thus takes place for all subtrees for which the upper left context is fulfilled.

Interestingly enough, inline matching is slower than one-pass matching. This is probably due to the administrative overhead of demand-driven construction of the document tree (cf. 9.2). Therefore the inline matcher is disabled by default and must be explicitly enabled by a command-line option. This is sensible if the input document is very large: Though the inline matcher is slower, it requires only a constant amount of memory. For large inputs, this advantage out-rules the speed loss.

9.1.5 Implementation of Text Patterns

A text pattern is a regular expression over the alphabet of UNICODE characters. The theory of string matching is a well-studied subject; efficient programs, like the UNIX tool *grep* and its variants, have been implemented and established. These tools implement regular expressions over the ASCII or LATIN1 alphabet by deterministic finite automata (DFA).

Since the input alphabet of such an automaton has at most 256 characters, its transition table can be implemented efficiently. Table compaction algorithms can additionally reduce the space required for representation. A possible problem arises due to the subset construction employed for obtaining the automaton: It can produce exponentially many states. [ASU86] therefore proposes in Section 3.7 to perform the subset construction only on demand. This technique is implemented, e.g., in *grep*.

When matching UNICODE strings, a number of additional difficulties emerge:

- ✧ Many characters, such as accented Latin characters have multiple representations in UNICODE: either as a single character in the LATIN1 character range, or as a sequence of combining characters. A UNICODE string matcher should be aware of equivalent representations for a single character.

- ⋄ There are several degrees of how precise a text matches a pattern. For instance, it is not intuitively clear whether the pattern “these” should be matched by the words “thèse”, “These” or “Thèse”. The user should be able to specify how precise a match must be.
- ⋄ Since the UNICODE alphabet has more than one million characters, the transition table of the DFA can become very large. It must therefore be intelligently represented.

An overview of these and similar problems and possible solutions is given in [Wer99]. For the implementation of text patterns in *fxgrep*, we only addressed the efficient representation of transition tables.

9.1.5.1 Representation of Transitions

In order to motivate our representation, let us consider some examples: The text pattern "`(购物中心)`" matches the Chinese word for “shopping centre”. The UNICODE codes of the four Chinese letters are 8D2D, 7269, 4E2D and 5FC3, i.e., they are spread over a range of 16000 characters.

If the text pattern additionally contains characters from the low ASCII range, as in "`(shopping centre|购物中心)`", then we even have to consider a range of about 36000 characters. Observe that the four Chinese characters are sparsely distributed in the highest 16000 characters, whereas all other characters are in the small ASCII range, and an area of about 20000 characters is not used at all.

A text pattern can also contain characters ranges, as in "`[一–龥]`", which matches a text that contains an arbitrary East Asian CJK character. In this case, all 20000 characters in the range share the same transitions.

The transitions for a single state of the DFA should be stored in a way that is space-efficient on the one hand and allows for fast access on the other hand. A straight-forward implementation is by a vector large enough to hold all transitions, except for a default transition that is used for symbols beyond the range of the vector. For instance, if the transitions for state q are $\{(q,5,q_1),(q,8,q_2),(q,13,q_3),(q,14,q_2)\} \cup \{(q,c,q_0) \mid c \notin \{5,8,13,14\}\}$, they can be represented as follows:

default: q_0	q_1	q_0	q_0	q_2	q_0	q_0	q_0	q_0	q_3	q_2
offset: 5	0			3			6			9

The vector is exactly large enough to hold all non-default transitions. A transition can be performed efficiently: An offset is added to the input symbol and the result is used as the index in the vector; if this is not in the vector’s range, the default is taken. However, this representation is unsuited for large ranges of characters because the size of the vector can be immense, more precisely up to a million for UNICODE. Even if we can employ table compaction algorithms such as described in [WM95], Section 7.4.3, we obtain at least one vector of that large size in the worst case.

Before we improve the representation, note the following two points:

- ✧ Though the transitions for each state are spread among a huge range of characters, nearly all of them are usually equal to the default transition.
- ✧ In some cases, however, there might be large contiguous blocks of characters for which all transitions lead to the same non-default state. This is the case, e.g., for text pattern ".*[一–龥].*", where all CJK characters have the same transition, but this is not the default transition. We therefore need an efficient way of representing such blocks.

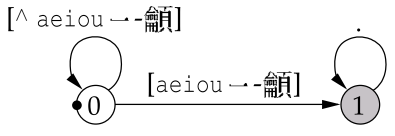
Our solution is the following SML data type:

```
datatype Segment = FIXED of int | TRANS of int array
type Row = (Char * Char * Segment) list * int * bool
type Dfa = Row vector
```

A state of the DFA is represented as an integer. Each row of the transition table holds a list of segments associated with an interval of UNICODE characters, a default transition and a boolean indicating whether the state represented by this row is a final state. Each segment (off,len,seg) describes the transitions for the character interval {off,...,off+len}. If seg has the form FIXED q, then the transitions for all of these characters lead to the same state q. Otherwise seg is TRANS arr, where arr is an array of size len+1, holding the resulting state for each character in the interval. The segments are in ascending order according to the character intervals they describe.

In order to perform a transition for a character *c*, the list of segments must be searched for a the segment containing *c*; because the list is sorted the search can be aborted as soon as a segment is encountered which is higher than *c*. A possible optimization is to use a vector instead of a list in order to enable binary search on the segments. We did, however, not implement this optimization because the lists are short in practice.

Example 9.1: Consider the text pattern ".*[aeiou一–龥].*" which matches a text that contains either a vowel or a CJK character. This is a rather artificial example but well-suited for illustrating the representation of transitions. This text pattern can be implemented by the following DFA:



From initial state 0 we reach state 1 by one of the desired characters; for all other characters, the automaton remains in state 0. State 1 is final; it makes a transition to itself on arbitrary characters. The the row for state 0 is[1]:

```
([(0wx61,0wx14,TRANS [|1,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1|]),
  (0wx4E00,0wx51A5,FIXED 1)],
 0,false)
```

The default transition leads to state 0. The first of the two segments holds all transitions for characters a...u (0wx61...0wx75 in UNICODE); only the entries for vowels are set to 1, all other fields of the array hold the default 0. The

[1]The SML notation for arrays is [|...|].

	Size in KB	XML document: number of Elem.	Attr. Spec.	Data Segm.	Elem.-types	Attr. names	*fxgrep* tree: Elem. nodes	Text nodes
`xml`	159.4	2306	1147	4762	74	20	8069	5101
`macbeth`	163,1	3975	0	7906	16	0	11925	7904

Table 9.1: The two XML documents used for compiling the statistics.

second segment represents the range of CJK characters (0wx4E00...0wx9FA5 = 0wx4E00+0wx51A5 in UNICODE). All characters in this segment have the same transition, leading to state 1.

The row for state 1 is very simple: It is ([],1,true). The same transition is performed for all characters, thus there is only a default transition for this row, but there are no segments. □

Summarizing, our representation has the following advantages:

- Contiguous blocks of states with the same transition are represented in constant space.
- Blocks of characters with the default transition consume no space at all.
- Small ranges of characters with diverse transitions are represented by an array with constant access time.
- If the text pattern involves only a small range of characters then all transitions can be represented by a single segment of type TRANS, i.e., a transition can be performed in constant time. This is particularly useful if only characters from the ASCII or LATIN1 range occur.

9.2 Statistics and Analysis

We conclude this chapter by giving some statistics: For the patterns and input documents from 8.2.2, we measured the time required for matching and compared it to the time needed for parsing the input. Moreover, we counted the size of the tables generated by the preprocessor and the number of states and transitions that occur during matching of the query.

Documents and Patterns Used for the Statistics

For generating the statistics we used the documents and example patterns from 8.2.2. The two documents are the XML recommendation and Shakespeare's play "Macbeth" [Bos99]; they are summarized in Table 9.1. The XML recommendation has a rather rich structure: It specifies many attributes and the number of occurring element-types is large. Compared to that, the Macbeth document has a poor structure: Only few different element types occur, and no attributes are specified at all.

Note that in the document tree constructed by *fxgrep* each element is represented by three nodes, according to Figure 8.1. Thus the number of element nodes in the document tree is at least three times the number of elements in the document. For the macbeth document, this is the precise number, whereas in case of the XML recommendation there is an additional element node for each attribute specified in the document (the remaining four element nodes are due to a processing instruction). Each text node in the document tree is either the concatenation of a number of adjacent character data segments, or it is an attribute value specified in the XML document.

The patterns we used are the example patterns from 8.2.2. For the xml document, these are the following:

p_1 = //prod[@id = "^NT-Char$"]
p_2 = //prod[@id="Char"]
p_3 = //prod[(lhs/"Char")_]
p_4 = //prod[(lhs["^Char$"])_]/rhs
p_5 = //prod[#_(rhs/nt[@def="Char"])_]/lhs/""
p_6 = //prod[_(//nt/"Char")_]/lhs/""
p_7 = //.[_#_(//prod[@id="^NT-element$"])_]//prod

For the macbeth document, we used the following patterns:

p_8 = //SPEECH[_(LINE/"thunder")_]
p_9 = //SPEECH[_(//LINE/"hurlyburly")_]/SPEAKER/.
p_{10} = //SPEECH[_#_(LINE/"hurlyburly")_]/SPEAKER/.
p_{11} = //SPEECH[_(SPEAKER/"Second Witch")_#_]/LINE/""
p_{12} = //SPEECH[_(LINE/"hurlyburly")#_]/LINE
p_{13} = //*[_(SPEECH//"hurlyburly")#_]/SPEECH/SPEAKER
p_{14} = //*[<¬ACT>*#_]/ACT[<¬SCENE>*#_]/SCENE/TITLE/""
p_{15} = //SCENE[_(//SPEAKER/"Witch")_][_(//SPEAKER/"MACBETH")_]/TITLE
p_{16} = //SCENE[_(TITLE/"desert")_]/*[¬_(SPEAKER/"Witch")_]/LINE

The statistics gathered for the individual patterns are presented in Table 9.2. In particular, we measured for each pattern the size of the generated context grammar, the number and size of the states which occurred during the runs of the automata, the number of different transitions that had to be computed, and the execution times of *fxgrep* and *fxp*.

Size of Context Grammars

The second column in Table 9.2 summarizes for each pattern the size of the generated extended context grammar. It lists the number of variables and rules in the grammar and the total number of NFA states generated by the Berry-Sethi construction for the regular expressions occurring in the grammar.

Observe that the generated context grammars are relatively large in comparison to the patterns: Even for the simple pattern p_1, 15 tree variables, 19 rules and 45 NFA states are generated. There are several reasons for this:

- ✧ Representation of the predefined variables $x_\top$ and x_w alone requires 10 variables, 13 rules and 28 NFA states.

- ✧ Each pattern is translated to a query grammar which has roughly one variable for each occurring tree pattern. According to the translation scheme presented in 8.1, each variable in a query grammar can be the

	Context Grammar			Tree States			Forest States			Transitions			Execution Time in sec.		
Pattern	Variables	Rules	NFA states	Occurred	Maximal Size	Average Size	Occurred	Maximal Size	Average Size	*Down*	*Up*	*Side*	Two Passes	One Pass	Inline
Document: `xml`													Parsing Time: 0.49		
p_1	15	19	45	15	3	2.0	32	4	2.3	122	111	31	–	0.70	0.78
p_2	15	19	45	15	3	2.0	34	4	2.4	127	112	36	–	0.68	0.80
p_3	16	20	52	17	3	2.2	40	4	2.7	131	114	45	–	0.69	0.79
p_4	17	21	56	18	3	2.3	70	9	4.2	296	132	88	0.74	–	–
p_5	22	26	73	24	3	2.3	99	8	4.1	356	148	124	0.75	–	–
p_6	21	25	75	23	3	2.4	101	12	4.6	354	148	127	0.76	–	–
p_7	26	31	87	23	5	2.6	98	11	5.4	442	147	136	0.81	–	–
Document: `macbeth`													Parsing Time: 0.55		
p_8	16	20	53	12	2	2.3	31	5	3.0	34	27	36	–	0.80	0.94
p_9	24	30	85	15	4	3.1	74	12	4.8	88	33	76	0.93	–	–
p_{10}	22	28	73	15	4	3.1	68	9	4.3	84	33	68	0.94	–	–
p_{11}	19	23	63	15	3	2.4	39	6	3.3	36	29	41	–	0.80	0.83
p_{12}	17	21	54	13	4	2.5	31	4	3.0	33	27	35	–	0.81	0.85
p_{13}	21	25	67	16	3	2.5	40	6	4.0	40	29	50	–	0.91	0.97
p_{14}	22	26	73	15	5	2.9	37	6	3.7	38	26	34	–	0.80	0.85
p_{15}	24	28	91	20	3	2.6	78	18	6.0	131	39	129	0.95	–	–
p_{16}	22	26	79	18	3	2.6	55	10	5.1	96	36	94	0.94	–	–

Table 9.2: Querying statistics.

source of a number of auxiliary variables in the generated context grammar. For instance, an element rule generates at least two auxiliary variables, namely for the `#atts` and `#content` subtrees, plus up to two auxiliary variables for each attribute pattern.

◇ Regular expressions are implicitly transformed, allowing optional white space between tree patterns and at the start and end of the whole expression (cf. 8.1.6). This procedure typically doubles the size of the regular expression.

Execution Times

For each pattern, we ran *fxgrep* ten times and averaged the execution times; the results are given in the last column of Table 9.2. Parsing of the document was performed in non-validating mode in order to spare the time needed for parsing the DTD which is not considered by the matcher in any case. For comparison, parsing was also performed with the stand-alone parser *fxp* which does no processing of the document at all. Comparing only the execution times, our first observation is that matching the query always consumes less time than parsing the document (the execution times of *fxgrep* include the time for parsing).

For right-ignoring patterns which are matched in a single pass, we measured the execution times of both the one-pass matcher and the inline matcher. Comparing these times it is striking that the inline matcher is always slower than the one-pass matcher, although it does not construct the document tree prior to matching. This can be explained as follows:

- The tables generated by the preprocessor must be available to the matcher. Matching on the constructed document tree, as in the one-pass matcher, is implemented by a few recursive functions. The tables are visible to these functions as global variables, namely through the scope of the enclosing function definition. This is different when matching in hooks: The tables must be incorporated explicitly into the application data, i.e., the arguments of the hooks. Apparently SML/NJ can not compile this as efficiently as a global variable.

- The inline matcher must construct the document trees for all subtrees that fulfill the left upper context. This requires the extra administrative task of maintaining an optional partial document tree, together with the information whether one of the ancestors fulfills the left upper context. This argument is strengthened by the times measured for patterns p_8 and p_{11}: For p_8 the difference of the two execution times is clearly larger than for p_{11}. The reason is that for p_8 the left upper context (*//*) is fulfilled by all nodes; the whole document tree must therefore be constructed in memory, though most subtrees will be discarded later due to violating the structural condition. p_{11} is much more restrictive: Its left upper context is only fulfilled by very few nodes.

The second point gives rise to a possible optimization which is, however, not incorporated into *fxgrep*: Although the node test `SPEECH` conceptually belongs to the structural condition of p_8, it can be checked before descending to the children of an element. This would avoid construction of the document tree at least for all nodes of a different type than `SPEECH`. Thinking one step further, the decision whether to construct the subtree can even be delayed until the attributes of an element are processed. This would spare the construction of even more subtrees.

It is worth mentioning that the time needed for matching a pattern is most strongly influenced by the number of passes required rather than by the pattern itself. This is reflected by the measured execution times: Matching is approximately equally fast for all two-pass patterns on a single document. The only exception is p_7 which is noticeably slower than the other patterns for the `xml` document. The reason is that the average size of the occurring states is larger than for the other patterns. The case is similar for one-pass matching: Pattern p_{13} is matched clearly slower than the other one-pass patterns for the `macbeth` document. In this case, however, the reason it not so obvious. We can only guess that the larger states are involved more frequently in transitions than the smaller ones, but this is not reflected by the statistics.

States and Transitions Computed During Matching

The third, fourth and fifth columns of Table 9.2 list the number of tree states, forest states and transitions that were computed during matching. First observe that extremely few states do actually occur, compared to the number of theoretically possible states.

- ◇ In many cases the number of tree states that occurred is smaller than the number of variables in the grammar, though the tree states are sets of these variables.
- ◇ The case is similar for forest states. However, two-pass matching requires more states than one-pass matching. This indicates that the reachable states of $B_G^{\rightarrow}$ do not coincide with those of $A_G^{\leftarrow}$.
- ◇ Generally the occurring tree states are by far fewer than the forest states. The reason is that there are also fewer variables than NFA states.

Observe also that the size of states is usually small: Tree states have an average size of about 2 – 3; the maximal size only rarely exceeds 4. For forests states, these numbers are larger: They have an average size of less than 6, which is still small in comparison to the large number of NFA states.

Note that three of the four patterns with the largest state sizes have a common characteristic: They have a structure qualifier. In contrast to a context qualifier, a structure qualifier introduces conjunction into the generated query grammar (cf. 8.2.3). This becomes clear when comparing patterns p_9 and p_{10}. Both express basically the same pattern, but p_{10} uses a context qualifier where p_9 has a structure qualifier. The effect is that the maximum and average size of forest states is noticeably larger for p_9. It is not surprising that the largest forest states occur for p_{15}: This pattern has even two structure qualifiers in a single node pattern.

The number of transitions that have to be computed is apparently related to the number of occurring states. Note however, that at most a few hundred different transitions occur. This is a small number compared to the number of nodes in the document tree.

Summary of Statistics

The statistics attest that demand-driven computation of the transitions is very practicable. Though the automata are theoretically of immense size, only a small fraction of the states and transitions are actually required during the matching procedure, even for large input documents. Moreover, the states, i.e., the sets that have to be processed in order to compute transitions, are typically small. The demand-driven computation of the transition tables is therefore reasonably efficient.

Conclusion

We have presented an algorithm for locating all matches of a query in an XML document by two consecutive runs of pushdown forest automata. Here is a summary of the addressed topics:

Representation of Documents: We represented documents as non-ranked trees. Though representation as ranked trees is also possible, the non-ranked representation has the advantage of allowing for both left-to-right and right-to-left traversal by automata, independently of whether the automaton is top-down or bottom-up. Moreover, the ranked-tree representation would significantly complicate the definition of pushdown forest automata: There would be no one-to-one correspondence between the tree structure and the moves on the pushdown.

In order to represent XML documents as forests, we added text nodes and introduced auxiliary nodes in order to distinguish between the attributes and the content of an XML element.

Forest Grammars and Query Grammars: We used forest grammars for specification of both the contextual and the structural condition. Forest grammars extend the notion of conventional tree grammars by allowing regular expressions on the right-hand sides of productions, accounting for the arbitrary number of children a node may have.

Forest grammars express regular forest languages. We enhanced the grammar formalism by adding conjunction and negation. While increasing the succinctness of the formalism, this adds nothing to the expressiveness of the grammars. In order to meet the requirements of XML we derived from forest grammars the formalism of query grammars. In detail, we added support for matching text nodes by external predicates and an abbreviated syntax for specifying conditions on the attributes of an element and for easy handling of white space in XML documents.

Pattern Language: We provided a pattern syntax as an alternative specification method for queries. This syntax is similar to the pattern language of XPATH. It adds, however, the capability of constraining the forest of children or siblings of a node by structure qualifiers or context qualifiers, while refraining from the arbitrary predicates allowed in XPATH qualifiers.

The pattern language is less expressive than query grammars: It can not express all regular conditions. In return it is much more concise and intuitive than query grammars. It is therefore suited for specification of queries even by non-computer scientists.

Regular Forest Languages: The class of languages accepted by forest grammars is the class of regular forest grammars. We established a one-to-one connection to regular languages of ranked trees, and showed that regular forest languages are closed under union, intersection and complement.

Forest Automata: We introduced the class of forest automata which accept exactly the regular forest languages. Forest automata are obtained from conventional bottom-up tree automata by enhancing them with an explicit side-transition. The side-transition is used for recognizing regular languages of words of states assigned to the children of a node. There are two variants of forest automata, depending on whether the side-transitions proceed from left to right or vice versa. We showed that forest automata can be made deterministic and can thus be implemented efficiently.

Pushdown Forest Automata: We adapted the traversing order of an XML parser by enhancing the bottom-up forest automata with a pushdown and a down-transition. The resulting class of pushdown forest automata is as expressive as the bottom-up automata: They accept exactly the regular forest languages. In contrast to other models of pushdown automata, pushdown forest automata can be made deterministic because moves on the pushdown are restricted. In the deterministic case, pushdown forest automata are significantly more succinct than bottom-up automata. Depending on the direction of side-transitions, there are two variants of pushdown forest automata.

Querying Algorithm: We presented a querying algorithm that employs two consecutive runs of pushdown forest automata for locating all matches of a query. The first automaton annotates the input forest with its states, and indicates with these states candidates for possible matches of the query. The second automaton runs on the annotated forest produced by the first automaton and traverses in the opposite direction. A match of a subtree is determined by the state of the second automaton when it arrives at that subtree. This automaton can thus report all matches during traversal of the forest. Choosing the traversing order of these two automata such that the second one proceeds from left to right ensures that the matches are reported in document order.

The automata employed for this algorithm can be exponentially large in the size of the query. Moreover, implementation of text patterns by external predicates additionally increases the size of the transition tables. It is therefore sensible to compute the transitions required for a run on the actual input forest on demand. Our implementation shows that only few states and transitions are actually required during each individual run.

Single-Pass Matching: We identified the subclass of right-ignoring grammars for which a single run of a forest automaton suffices. For these queries the matching can be performed during parsing of the document. The document tree then need only be constructed for those subtrees that match the upper left context. Because this requires an additional administrative effort, this strategy is slightly slower than performing the match on the readily constructed document tree. For large documents, however, it spares huge amounts of consumed memory and is therefore the preferable method.

It is efficiently decidable whether a grammar is right-ignoring. The user who specifies the query therefore need not be aware of this property. This is especially important because XML querying is often done by non-computer scientists.

Implementation: We implemented the querying algorithm in SML on top of the XML parser *fxp*. The results are extremely satisfactory: On the one hand, querying an XML document is faster than parsing it, even for complex queries. On the other hand, we found that the number of states and transitions that occur during a run of a forest automaton is usually very small, even for large documents. This justifies the approach of demand driven computation of the transition tables.

Moreover, our implementation proves the suitability of SML for processing structured documents. We were able to use ML-YACC for generation of the frontend for the syntax of grammars and patterns. Implementation of the automata constructions and the tree traversing functions was easy and straight-forward with the help of SML's user-defined data-types and recursive functions. The employment of polymorphic and higher-order functions as well as SML's parametric modules makes the code highly reusable and customizable. The parser library of *fxp* turned out to be a comfortable platform for development of XML processing applications.

Bibliography

[AAC$^+$98] Nabeel Al-Shamma, Robert Ayers, Richard Cohn, Jon Ferraiolo, et alias, editors. *Precision Graphics Markup Language (PGML)* . W3C Note, World Wide Web Consortium, April 1998. Available online at `http://www.w3.org/TR/1998/NOTE-PGML-19980410`.

[Abi97] Serge Abiteboul. Querying Semi-Structured Data. In F.N. Afrati and P.G. Kolaitis, editors, *Proceedings of the International Conference on Database Theory, Delphi, Greece*, volume 1186 of *Lecture Notes in Computer Science*, pages 1–18. Springer, Heidelberg, 1997.

[AMT94] Andrew W. Appel, James S. Mattson, and David R. Tarditi. *A lexical analyzer generator for Standard ML, Version 1.6.0.* Software Documentation, Princeton University, October 1994. Available online at `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ML-Lex/manual.html`.

[AQM$^+$97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, December 1997.

[Ari99] Ariba, Inc. *cXML/1.0*. Business Standard, August 1999. Available online at `http://www.cxml.org/home/`.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullmann. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[AT94] Andrew W. Appel and David R. Tarditi. *ML-Yacc User's Manual, Version 2.3.* Software Documentation, Princeton University, October 1994. Available online at `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ML-Yacc/manual.html`.

[AU71] Alfred V. Aho and Jeffrey D. Ullman. Translations on a Context-Free Grammar. *Information and Control*, 19(5):439–475, December 1971.

[Bal99] Steve Ball. *TclXml 1.2.* Software Documentation, Zveno Pty. Ltd., May 1999. Available online at `http://www.zveno.com/zm.cgi/in-tclxml/`.

[Bel99] *Standard ML of New Jersey*. Home Page, 1989-1999. Available online at `http://cm.bell-labs.com/cm/cs/what/smlnj/`.

[BHW98] Anne Brüggemann-Klein, Stefan Hermann, and Derick Wood. Context and Caterpillars and Structured Documents. In Ethan V. Munson, Charles Nicolas, and Derick Wood, editors, *Principles of Digital Document Processing (PODDP'98)*, volume 1481 of *Lecture Notes in Computer Science*, pages 1–9. Springer, Heidelberg, March 1998.

[BKR96] Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *First International Workshop on Implementing Automata (WIA'96)*, volume 1260 of *Lecture Notes in Computer Science*. Springer, Heidelberg, 1996.

[BKR99] Nick Benton, Andrew Kennedy, and George Russell. *The MLj Compiler*. Software and documentation, Persimmon IT, Inc., Cambridge, U.K., 1999. Available online at `http://www.dcs.ed.ac.uk/~mlj/`.

[Blu97] Matthias Blume. *CM – A Compilation Manager for SML/NJ*. User Manual, Princeton University, 1997. Available online at `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/CM/index.html`.

[BMW91] Jürgen Börstler, Ulrich Möncke, and Reinhard Wilhelm. Table Compression for Tree Automata. *ACM Transactions on Programming Languages and Systems*, 13(3):295–314, 1991.

[Bos99] Jon Bosak, editor. *The Complete Plays of Shakespeare, Marked up in XML*, 1999. Available online at `http://metalab.unc.edu/xml/examples/shakespeare`.

[Brü93] Anne Brüggemann-Klein. Regular Expressions into Finite Automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[Bra69] Walter S. Brainerd. Tree Generating Regular Systems. *Information and Control*, 14:217–231, 1969.

[Bra98a] Neil Bradley. *The XML Companion*. Addison Wesley Longman, Harlow, Essex, 1998.

[Bra98b] Tim Bray. *The Annotated XML Specification*. Tutorial, 1998. Available online at `http://www.xml.com/axml/axml.html`.

[BS86] Gerard Berry and Ravi Sethi. From Regular Expressions to Deterministic Automata. *Theoretical Computer Science*, 48:117–126, 1986.

[BW92] Anne Brüggemann-Klein and Derick Wood. Deterministic Regular Languages. In A. Finkel and M. Jansen, editors, *STACS 92*, volume 577 of *Lecture Notes in Computer Science*, pages 173–184. Springer, Heidelberg, 1992.

[BW98] Anne Brüggemann-Klein and Derick Wood. *Regular Tree Languages over Non-Ranked Alphabets*. Unpublished draft, April 1998. Available online at `http://www.oasis-open.org/cover/regTreeLanguages-ps.gz`.

[CDG+99] Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*, April 1999. Available online at `http://www.grappa.univ-lille3.fr/tata/`.

[CDS+98] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your Mediators Need Data Conversion! In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 177–188. ACM Press, 1998.

[Cho60] Noam Chomsky. On the Notion "Rule of Grammar". In Roman Jakobson, editor, *Proceedings of the 12th Symposium in Applied Mathematics*, pages 6–24. American Mathematical Society, Providence, Rhode Island, April 1960.

[Cla98] James Clark. *sp 1.3*. Software Documentation, March 1998. Available online at `http://www.jclark.com/sp/index.htm`.

[Cla99a] James Clark. *Expat 1.1*. Software Documentation, May 1999. Available online at `http://www.jclark.com/xml/expat.html`.

[Cla99b] James Clark. *xp 0.5*. Software Documentation, January 1999. Available online at `http://www.jclark.com/xml/xp/index.htm`.

[Cow99] John Cowan. *XML Catalog Proposal, Draft 0.4*, April 1999. Available online at `http://www.ccil.org/~cowan/XML/XCatalog.html`.

[CZ96] Pascal Caron and Djelloull Ziadi. *Characterization of Glushkov Automata*. Technical Report LIR-96.06, Laboratoire d'Informatique, Université de Rouen, France, 1996.

[DFF+99] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. In *International World Wide Web Conference*, 1999.

[Don70] John Doner. Tree Acceptors and Some of Their Applications. *Journal of Computer and System Sciences*, 4:406–451, 1970.

[EH99] Joost Engelfriet and Hendrik Jan Hoogeboom. Tree-Walking Pebble Automata. In J. Karhumäki, H. Maurer, G. Paun, and G.Rozenberg, editors, *Jewels are Forever, Contributions to Theoretical Computer Science in Honor of Arto Salomaa*, pages 72–83. Springer, Heidelberg, 1999.

[Fec94] Christian Fecht. *A Guide to TrafoLa*. Software documentation, Universität des Saarlandes, 1994.

[FS98] Christian Fecht and Helmut Seidl. Propagating Differences: An Efficient New Fixpoint Algorithm for Distributive Constraint Systems. In *Programming Languages and Systems, 7th European Symposium on Programming (ESOP '98), Lisbon, Portugal*, volume 1381 of *Lecture Notes in Computer Science*, pages 90–104. Springer, Heidelberg, 1998.

[FSF99] Free Software Foundation. *GNU grep*. Software and Documentation, 1999. Available online at `ftp://ftp.gnu.org/gnu/grep/`.

[FST98] Financial Services Technology Consortium. *Bank Internet Payment System, Specification, Version 1.0*. Public Review Draft, August 1998. Available online at `http://www.fstc.org/projects/bips/`.

[FSW99] Mary Fernandez, Jerome Simeon, and Philip Wadler, editors. *XML Query Languages: Experiences and Exemplars*. Draft manuscript, September 1999. Available online at `http://www-db.research.bell-labs.com/user/simeon/xquery.html`.

[Fuc99] Matthew Fuchs. Why XML Is Meant for Java – Exploring the XML/Java Connection. *Web Techniques*, June 1999. Available online at `http://www.webtechniques.com/archives/1999/06/fuchs/`.

[Fxg99] Andreas Neumann. *fxgrep 1.2*. Source Code, 1999. Available online at `http://www.informatik.uni-trier.de/~neumann/Fxp/fxgrep/`.

[Fxp99] Andreas Neumann. *fxp 1.2*. Source Code, 1999. Available online at `http://www.informatik.uni-trier.de/~neumann/Fxp/`.

[Gar99] Lars Marius Garshol. *xmlproc: A Python XML parser, Version 0.61*. Software Documentation, April 1999. Available online at `http://www.stud.ifi.uio.no/~larsga/download/python/xml/xmlproc.html`.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification, Version 1.0*. Addison Wesley, August 1996.

[Glu61] V. M. Glushkov. The Abstract Theory of Automata. *Russian Mathematical Surveys*, 16:1–53, 1961.

[GMW99] Roy Goldman, Jason McHugh, and Jennifer Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In Sophie Cluet and Tova Milo, editors, *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99), Philadelphia, Pennsylvania*, pages 25–30. INRIA, June 1999.

[Gol90] Charles F. Goldfarb. *The SGML Handbook*. Clarendon Press, Oxford, 1990.

[Gol99] Charles Goldfarb. *The XML Handbook*. Prentice Hall, New Jersey, 2nd edition, November 1999.

[GS97] Ferenc Gécseg and Magnus Steinby. Tree Languages. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 1, pages 1–68. Springer, Heidelberg, 1997.

[Gue83] Irène Guessarian. Pushdown Tree Automata. *Mathematical Systems Theory*, 16(4):237–263, 1983.

[Hal85] Robert H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[Har99] Eliotte R. Harold, editor. *1998 Baseball Statistics – XML Sample Files*, 1999. Available online at `http://metalab.unc.edu/xml/examples/1998validstats.xml`.

[HS93] Reinhold Heckmann and Georg Sander. TrafoLa-H Reference Manual. In B. Hoffmann and B. Krieg-Brückner, editors, *Program Development by Specification and Transformation*, volume 680 of *Lecture Notes in Computer Science*, part II, chapter 8, pages 275–313. Springer, Heidelberg, 1993.

[IBM99] IBM AlphaWorks. *XML Parser for Java*. Software Documentation, August 1999. Available online at `http://www.alphaworks.ibm.com/formula/xml/`.

[IET92] K. Simonsen, editor. *Character Mnemonics & Character Sets*. Internet RFC 1345, IETF (Internet Engineering Task Force), June 1992. Available online at `http://www.ietf.org/rfc/rfc1345.txt`.

[IET98a] T. Berners-Lee, R. Fielding, and L. Masinter, editors. *Uniform Resource Identifiers (URI): Generic Syntax*. Internet RFC 2396, IETF (Internet Engineering Task Force), August 1998. Available online at `http://www.ietf.org/rfc/rfc2396.txt`.

[IET98b] F. Yergeau, editor. *UTF-8, a transformation format of ISO 10646*. Internet RFC 2279, IETF (Internet Engineering Task Force), January 1998. Available online at `http://www.ietf.org/rfc/rfc2279.txt`.

[ISO86] International Organization for Standardization. *Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*. Ref. No. ISO 8879:1986 (E). Geneva/New York, 1986.

[ISO96] International Organization for Standardization. *Information technology – Processing Languages – Document Style Semantics and Specification Language (DSSSL)*. Ref. No. ISO/IEC 10179:1996(E), 1996.

[ISO98] International Organization for Standardization. *Information technology – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1*. International Standard ISO/IEC 8859-1:1998, 1998.

[JHA$^+$98] Simon Peyton Jones, John Hughes, Lennart Augustson, et alias. *Haskell 98: A Non-strict, Purely Functional Language*, February 1998. Available online at `http://haskell.systemsz.cs.yale.edu/definition/`.

[JK96a] Jani Jaakkola and Pekka Kilpeläinen. *Sgrep 0.99*. Software and Documentation, Document Management Group, Computer Science Department, University of Helsinki, 1996. Available online at `http://www.cs.helsinki.fi/~jjaakkol/sgrep.html`.

[JK96b] Jani Jaakkola and Pekka Kilpeläinen. *Using Sgrep for Querying Structured Text Files*. Technical Report C-1999-83, Department of Computer Science, University of Helsinki, November 1996.

[Joh99] Mark Johnson. XML JavaBeans, Part 1 – Make JavaBeans mobile and interoperable with XML. *Java World*, February 1999. Available online at `http://www.javaworld.com/`. Parts 2 and 3 appeared in March and July.

[Kle56] Stephen C. Kleene. Representation of Events in Nerve Sets and Finite Automata. In C.E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, Princeton, New Jersey, 1956.

[Kro75] H. Kron. *Tree Templates and Subtree Transformational Grammars*. PhD thesis, University of California, Santa Cruz, 1975.

[KS81] Tsutomu Kamimura and Giora Slutzki. Parallel and Two-Way Automata on Directed Ordered Acyclic Graphs. *Information and Control*, 49(1):10–51, April 1981.

[LH92] Baudouin LeCharlier and Pascal Van Hentenryck. *A Universal Top-Down Fixpoint Algorithm*. Technical Report CS-92-25, Brown University, Providence, 1992.

[Lin99] Christian Lindig. *Tony - a XML Parser and Pretty Printer*. Software Documentation, 1999. Available online at `http://www.cs.tu-bs.de/softech/people/lindig/software/tony.html`.

[LRV$^+$99] Xavier Leroy, Didier Rémy, Jérôme Vouillon, and Damien Doligez. *The Objective Caml System,Documentation and User's Guide*. Online documentation, I.N.R.I.A., France, 1999. Available online at `http://pauillac.inria.fr/caml/ocaml/htmlman/`.

[Lut96] Mark Lutz. *Programming Python*. O'Reilly & Associates, October 1996.

[Mai98] David Maier. Database Desiderata for an XML Query Language. In *The W3C Query Languages Workshop (QL'98), Boston, Massachussets*. World Wide Web Consortium, November 1998. Available online at `http://www.w3.org/TandS/QL/QL98/pp/maier.html`.

[Meg$^+$98] David Megginson et alias, editors. *SAX 1.0: The Simple API for XML*. Online Documentation, Megginson Technologies, May 1998. Available online at `http://www.megginson.com/SAX/index.html`.

[Mor94] Etsuro Moriya. On two-way tree automata. *Information Processing Letters*, 50:117–121, 1994.

[MS98] Maya Madhavan and Priti Shankar. Optimal Regular Tree Pattern Matching Using Pushdown Automata. In V. Arvind and R. Ramamujan, editors, *Foundations of Software Technology and Theoretical Computer Science, (18th FST&TCS)*, volume 1530 of *Lecture Notes in Computer Science*, pages 122–133. Springer, Heidelberg, 1998.

[MTH$^+$97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Mul99] Charles Muller, editor. *Dictionary of East Asian Literary CJK Terms – XML Sample Files*, 1999. Available online at `http://www.human.toyogakuen-u.ac.jp/~acmuller/dicts/xmlcjkdict/data/ind%ex.html`.

[Mur95] Makoto Murata. *Forest Regular Languages and Tree Regular Languages*. Unpublished manuscript, 1995. Available online at `http://www.geocities.com/ResearchTriangle/Lab/6259/podp.pdf`.

[Mur96] Makoto Murata. Transformations of Trees and Schemas by Patterns and Contextual Conditions. In Charles Nicolas and Derick Wood, editors, *Principles of Document Processing (PODP'96)*, volume 1293 of *Lecture Notes in Computer Science*, pages 153–169. Springer, Heidelberg, 1996.

[Neu97] Andreas Neumann. Unambiguity of SGML Content Models – Pushdown Automata Revisited. In Symeon Bozapalidis, editor, *Proceedings of the 3rd International Conference Developments in Language Theory (DLT'97)*, pages 507–518. Aristotle University of Thessaloniki, 1997.

[NP93] Maurice Nivat and Andreas Podelski. Another Variation on the Common Subexpression Problem. *Discrete Mathematics*, 114:379–401, 1993.

[NS98a] Andreas Neumann and Helmut Seidl. *Locating Matches of Tree Patterns in Forests*. Technical Report 98-08, Mathematik/Informatik, Universität Trier, 1998.

[NS98b] Andreas Neumann and Helmut Seidl. Locating Matches of Tree Patterns in Forests. In V. Arvind and R. Ramamujan, editors, *Foundations of Software Technology and Theoretical Computer Science, (18th FST&TCS)*, volume 1530 of *Lecture Notes in Computer Science*, pages 134–145. Springer, Heidelberg, 1998.

[NS99] Frank Neven and Thomas Schwentick. Query Automata. In *Proceedings of the Eighteenth Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*, pages 205–214. ACM Press, 1999.

[NY60] R. Naughton and H. Yamada. Regular Expressions and State Graphs for Automata. *IRE Transactions on Electronic Computers*, EC-9(1):39–47, 1960.

[O'K90] Richard R. O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, Massachusetts, 1990.

[OMG99] Object Model Group. *XML Metadata Interchange (XMI). Proposal to the OMG OA&DTF RFP 3: Stream-based Model Interchange Format (SMIF)*. OMG Document ad/98-07-01, July 1999. Available online at `ftp://ftp.omg.org/pub/docs/ad/98-07-01.ps`.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, May 1994.

[Pau96] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, July 1996.

[PE88] Keshav Pingali and Kattamuri Ekanadham. Accumulators: New Logic Variable Abstractions for Functional Languages. In Kesav V. Nori and Sanjeev Kumar, editors, *Foundations of Software Technology and Theoretical Computer Science, (8th FST&TCS), Pune, India*, volume 338 of *Lecture Notes in Computer Science*, pages 377–399. Springer, 1988.

[Pod92] Andreas Podelski. A Monoid Approach to Tree Automata. In Maurice Nivat and Andreas Podelski, editors, *Tree Automata and Languages*, pages 41–56. North Holland, 1992.

[PQ68] C. Pair and A. Quere. Définition et Etude des Bilangages Réguliers. *Information and Control*, 13:565–593, 1968.

[PXS99] Python XML Special Interest Group. *xml 0.5.1*. Software Documentation, April 1999. Available online at `http://www.python.org/sigs/xml-sig/`.

[Rep97] John Reppy, editor. *The Standard ML Basis Library*. Draft Report, Bell Labs, Lucent Technologies, 1997. Available online at `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/basis/index.html`.

[Rob99] Jonathan Robie, editor. *XQL (XML Query Language)*. Proposal, August 1999. Available online at `http://metalab.unc.edu/xql/xql-proposal.html`.

[Sei90] Helmut Seidl. Deciding Equivalence of Finite Tree Automata. *SIAM Journal on Computing*, 19(3):424–437, June 1990.

[Ses99] Peter Sestoft. *Moscow ML*. Software and documentation, Royal Veterinary and Agricultural University, Copenhagen, Denmark, 1999. Available online at `http://www.dina.kvl.dk/~sestoft/mosml.html`.

[SG85] Karl M. Schimpf and Jean H. Gallier. Tree Pushdown Automata. *Journal of Computer and System Sciences*, 30(1):25–40, 1985.

[Smo98] Gerd Smolka. *Concurrent Constraint Programming Based on Functional Programming*. Talk given at the European Joint Conferences on Theory and Practice of Software (ETAPS) Lisbon, Portugal, 1998. Available online at `http://www.ps.uni-sb.de/~smolka/drafts/etaps98.ps`.

[SO97] Paul Grosso, editor. *Entity Management (Amendment 2 to TR 9401)*. Technical Resolution 9401:1997, SGML Open, September 1997. Available online at `http://www.oasis-open.org/html/a401.htm`.

[SY98] Junichi Suzuki and Yoshikazu Yamamoto. Making UML models exchangeable over the Internet with XML: UXF approach. In *≪UML≫'98 - Beyond the Notation, Mulhouse, France*, June 1998.

[Tak75] M. Takahashi. Generalizations of Regular Sets and their Application to a Study of Context-Free Languages. *Information and Control*, 27:1–36, 1975.

[Tha67] J. W. Thatcher. Characterizing Derivation Trees of Context-Free Grammars through a Generalization of Finite Automata Theory. *Journal of Computer and System Sciences*, 1:317–322, 1967.

[Tob99] Richard Tobin. *RXP 1.1*. Software Documentation, Language Technology Group, Edinburgh, July 1999. Available online at `http://www.cogsci.ed.ac.uk/~richard/rxp.html`.

[TW68] J. W. Thatcher and J. B. Wright. Generalization of Finite Automata Theory with an Application to a Decision Problem of Second Order Logic. *Mathematical Systems Theory*, 2(1):57–81, 1968.

[Uni96] The Unicode Consortium. *The Unicode Standard, Version 2.0*. Addison Wesley Developers Press, Reading, Massachusetts, 1996.

[W3C98a] Vidur Apparao, Steve Byrne, Mike Champion, et alias, editors. *Document Object Model (DOM) Level 1 Specification, Version 1.0*. W3C Recommendation, World Wide Web Consortium, October 1998. Available online at `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001`.

[W3C98b] Tim Bray, Jean Paoli, and C.M. Sperberg-McQueen, editors. *Extensible Markup Language (XML) 1.0*. W3C Recommendation, World Wide Web Consortium, February 1998. Available online at `http://www.w3.org/TR/1998/REC-xml-19980210`. XML version available at `http://www.w3.org/TR/1998/REC-xml-19980210.xml`.

[W3C98c] James Clark and Stephen Deach, editors. *Extensible Style Language (XSL) Version 1.0*. W3C Working Draft, World Wide Web Consortium, December 1998. XML Version. Available online at `http://www.w3.org/TR/1998/WD-xsl-19981216.xml`.

[W3C98d] Eve Maler and Steve DeRose, editors. *XML Linking Language (XLink)*. W3C Working Draft, World Wide Web Consortium, March 1998. Available online at `http://www.w3.org/TR/WD-xlink`.

[W3C98e] Dave Raggett, Arnaud Le Hors, and Ian Jacobs, editors. *HTML 4.0 Specification*. W3C Recommendation, World Wide Web Consortium, April 1998. Available online at `http://www.w3.org/TR/REC-html40/`.

[W3C99a] Tim Bray, Dave Hollander, and Andrew Layman, editors. *Namespaces in XML*. W3C Recommendation, World Wide Web Consortium, January 1999. Available online at `http://www.w3.org/TR/1999/REC-xml-names-19990114`.

[W3C99b] James Clark, editor. *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation, World Wide Web Consortium, November 1999. Available online at `http://www.w3.org/TR/xslt`.

[W3C99c] James Clark and Steve DeRose, editors. *XML Path Language (XPath) Version 1.0*. W3C Recommendation, World Wide Web Consortium, November 1999. Available online at `http://www.w3.org/TR/xpath`.

[W3C99d] John Cowan and David Megginson, editors. *XML Information Set*. W3C Working Draft, World Wide Web Consortium, May 1999. Available online at `http://www.w3.org/TR/1998/WD-xml-infoset-19990517`.

[W3C99e] Stephen Deach, editor. *Extensible Style Language (XSL) Specification*. W3C Working Draft, World Wide Web Consortium, April 1999. Available online at `http://www.w3.org/TR/1999/WD-xsl-19990421`.

[W3C99f] Steve DeRose and Ron Daniel Jr., editors. *XML Pointer Language (XPointer)*. W3C Working Draft, World Wide Web Consortium, July 1999. Available online at `http://www.w3.org/TR/WD-xptr`.

[W3C99g] S. Pemberton et alias, editors. *XHTML 1.0: The Extensible HyperText Markup Language. A Reformulation of HTML 4.0 in XML 1.0*. W3C Working Draft, World Wide Web Consortium, March 1999. Available online at `http://www.w3.org/TR/WD-html-in-xml/`.

[W3C99h] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn, editors. *XML Schema Part 1: Structures*. W3C Working Draft, World Wide Web Consortium, November 1999. Available online at `http://www.w3.org/TR/1999/WD-xmlschema-1-19991105/`.

[Wad86] Philip Wadler. A New Array Operation. In Joseph H. Fasel and Robert M. Keller, editors, *Proceedings of the Graph Reduction Workshop, Santa Fé, New Mexico*, volume 279 of *Lecture Notes in Computer Science*, pages 328–335. Springer, 1986.

[Wad99] Philip Wadler. *A formal semantics of patterns in XSLT*. *Markup Technologies, Philadelphia*, to appear, December 1999. Available online at `http://cm.bell-labs.com/cm/cs/who/wadler/papers/xsl-semantics/xsl-seman%tics.ps`.

[Wat93] Bruce W. Watson. *A Taxonomy of Finite Automata Construction Algorithms*. Computing Science Report 93/43, Faculty of Mathematics and Computing Science, Eindhoven University of Technology, Netherlands, 1993.

[WC99] Larry Wall and Clark Cooper. *XML::Parser 2.26*. Software Documentation, July 1999. Available online at `http://www.cpan.org/modules/by-module/XML/COOPERCL`.

[WCS96] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, 2nd edition, September 1996.

[Wee99] Stephen Weeks. *MLton User's Guide*. Software and documentation, NEC Software Systems Research, 1999. Available online at `http://www.neci.nj.nec.com/PLS/MLton/`.

[Wer99] Laura L. Werner. *Efficient Text Searching in Java: Finding the Right String in any Language* . Talk given at the *14th International Unicode Conference*, March 1999. Available online at `http://www-4.ibm.com/software/developer/library/text-searching.html`.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, Reading, Massachusetts, 1995.

[Woo87] Derick Wood. *Theory of Computation*. Wiley, New York, 1987.

[WR99] Malcolm Wallace and Colin Runciman. *Haskell and XML: Generic Document Processing Combinators vs. Type-Based Translation*. Draft Paper, University of York, March 1999. Available online at `http://www.cs.york.ac.uk/fp/HaXml/paper.html`.

Appendix A

Proofs

A.1 Proof of Theorem 6.1

A.1.1 Proof of (1)

In order to complete the proof, we have to show that $\mathcal{L}_A = \mathcal{L}_G$. Let us first show that $\mathcal{L}_A \subseteq \mathcal{L}_G$: By structural induction (Corollary 4.1), it suffices to show (F) and (T) for the following invariants $\mathcal{A}_\mathcal{T}$ and $\mathcal{A}_\mathcal{F}$:

$$\begin{array}{rcl} \mathcal{A}_\mathcal{T}\, t & \equiv & \text{if } (t,x) \in \delta_\mathcal{T} \text{ then } t \in [\![G]\!]\, x, \text{ for all } x \in X; \\ \mathcal{A}_\mathcal{F}\, f & \equiv & \text{if } (f,y) \in \delta_\mathcal{F} \text{ and } y \in F_j, \text{ then } f \in [\![G]\!]\, r_j, \text{ for all } j. \end{array}$$

Let us start with (F): Let $f = t_1 \ldots t_n$ and assume that $\mathcal{A}_\mathcal{T}\, t_i$ holds for all i. If $(f,y) \in \delta_\mathcal{F}$ then, by Lemma 6.1, there are $x_1, \ldots, x_n$ and $y_1, \ldots, y_{n+1}$ such that $y_1 \in I$, $y_{n+1} = y$, and for $1 \leqslant i \leqslant n$, $(t_i, x_i) \in \delta_\mathcal{T}$ and $(y_i, x_i, y_{i+1}) \in \mathit{Side}$. Because $y \in F_j \subseteq Y_j$ and all Y_j are disjoint, $(y_i, x_i, y_{i+1}) \in \delta_j$ and $y_i \in Y_j$ for all i. Because $y_1 \in I$, we have that $y_1 = y_{0,j}$, and with $y_{n+1} = y \in F_j$, we get by Proposition 4.1 that $x_1 \ldots x_n \in [\![r_j]\!]_\mathcal{R}$. Moreover, $t_i \in [\![G]\!]\, x_i$ because of $\mathcal{A}_\mathcal{T}\, t_i$ for all i, and thus, by definition, $f \in [\![G]\!]\, r_j$, and (F) holds.

It remains to show (T): Let $t = a\langle f \rangle$, $(t,x) \in \delta_\mathcal{T}$, and assume that $\mathcal{A}_\mathcal{F}\, f$ holds. By definition there is a y such that $(f,y) \in \delta_\mathcal{F}$ and $(y,a,x) \in \mathit{Up}$. By, construction, $x \to a\langle r_j \rangle$ for some j, and $y \in F_j$. Now $\mathcal{A}_\mathcal{F}\, f$ yields that $f \in [\![G]\!]\, r_j$. Thus, by definition, $t \in [\![G]\!]\, x$, (T) holds and $\mathcal{L}_A \subseteq \mathcal{L}_G$.

We still have to show that $\mathcal{L}_G \subseteq \mathcal{L}_A$. Again, we define induction invariants $\mathcal{A}_\mathcal{T}$ and $\mathcal{A}_\mathcal{F}$:

$$\begin{array}{rcl} \mathcal{A}_\mathcal{T}\, t & \equiv & \text{if } t \in [\![G]\!]\, x \text{ then } (t,x) \in \delta_\mathcal{T}, \text{ for all } x \in X; \\ \mathcal{A}_\mathcal{F}\, f & \equiv & \text{if } f \in [\![G]\!]\, r_j \text{ then there is a } y \in F_j \text{ with } (f,y) \in \delta_\mathcal{F}. \end{array}$$

Let us show (F): Let $f = t_1 \ldots t_n \in [\![G]\!]\, r_j$. Then there is a word $x_1 \ldots x_n \in [\![r_j]\!]_\mathcal{R}$ such that $t_i \in [\![G]\!]\, x_i$, and thus $(t_i, x_i) \in \delta_\mathcal{T}$ by $\mathcal{A}_\mathcal{T}\, t_i$, for $i = 1, \ldots, n$. By Proposition 4.1, there are $y_1, \ldots, y_{n+1}$ with $y_1 = y_{0,j}$, $y_{n+1} \in F_j$ and $(y_i, x_i, y_{i+1}) \in \delta_j$. Because $y_{0,j} \in I$ and $\delta_j \subseteq \mathit{Side}$, we get with Lemma 6.1 that $(f, y_{n+1}) \in \delta_\mathcal{F}$, and (F) holds.

Finally, let us show (T): If $t = a\langle f \rangle \in [\![G]\!]\, x$, then $x \to a\langle r_j \rangle$ for some j and $f \in [\![G]\!]\, r_j$. By, $\mathcal{A}_\mathcal{F}\, f$, there is a $y \in F_j$ with $(f,y) \in \delta_\mathcal{F}$, and by construction $(y,a,x) \in \mathit{Up}$. Thus $(t,x) \in \delta_\mathcal{T}$. This completes the proof of (1). □

A.1.2 Proof of (2)

We have to show that for $q \in Q$, $(f,q) \in \delta_{\mathcal{F}}$ iff $f \in [\![G]\!]\, r_q$. Let us first define invariants $\mathcal{A}_{\mathcal{T}}$ and $\mathcal{A}_{\mathcal{F}}$ for the structural induction:

$$\begin{array}{rcl} \mathcal{A}_{\mathcal{T}}\, t & \equiv & (t,p) \in \delta_{\mathcal{T}} \text{ iff } t \in [\![G]\!]\, p, \text{ for all } p \in P. \\ \mathcal{A}_{\mathcal{F}}\, f & \equiv & (f,q) \in \delta_{\mathcal{F}} \text{ iff } f \in [\![G]\!]\, r_q, \text{ for all } q \in Q. \end{array}$$

We have to show (F) and (T); let us start with (F): Let $f = t_1 \ldots t_n$ and $(f,q) \in \delta_{\mathcal{F}}$. By Lemma 6.1, there are $p_1, \ldots, p_n$ and $q_1, \ldots, q_{n+1}$ such that $q_1 \in I$, $q_{n+1} = q$, and for $1 \leqslant i \leqslant n$, $(t_i, p_i) \in \delta_{\mathcal{T}}$ and $(q_i, p_i, q_{i+1}) \in \mathit{Side}$. If $n = 0$, then $q = q_1 \in I$, thus $q_0 \in F_q$ and $f = \epsilon \in [\![r_q]\!]_{\mathcal{R}}$. Otherwise, by construction, $(q_i, p_i, q_{i+1}) \in \delta$ for all i and $(q_0, p_1, q_2) \in \delta$; hence $p_1 \ldots p_n \in \mathcal{L}_{N_q} = [\![r_q]\!]_{\mathcal{R}}$. Now $t_i \in [\![G]\!]\, p_i$ because of $\mathcal{A}_{\mathcal{T}}\, t_i$ for $1 \leqslant i \leqslant n$, and thus $f \in [\![G]\!]\, r_q$.

On the other hand, suppose that $f \in [\![G]\!]\, r_q$. Then there is a word $p_1 \ldots p_n \in [\![r_q]\!]_{\mathcal{R}}$ such that $t_i \in [\![G]\!]\, p_i$ for $1 \leqslant i \leqslant n$. Because N_q accepts $[\![r_q]\!]_{\mathcal{R}}$, there are $q_1, \ldots, q_{n+1} \in Q_0$ with $q_1 = q_0$, $q_{n+1} \in F_q$ and $(q_i, p_i, q_{i+1}) \in \delta$ for all i. If $n = 0$, then $q_0 \in F_q$ and by construction $q \in I$; hence $(\epsilon, q) \in \delta_{\mathcal{F}}$. Otherwise, by definition of δ, there is a $q_I \in I$ with $(q_I, p_1, q_2) \in \mathit{Side}$. Moreover, because no transition in δ leads to q_0, $q_i \neq q_0$ and thus $(q_i, p_i, q_{i+1}) \in \mathit{Side}$ for $i > 1$. By $\mathcal{A}_{\mathcal{T}}\, t_i$, $(t_i, p_i) \in \delta_{\mathcal{T}}$ for all i, and by Lemma 6.1 $(f,q) \in \delta_{\mathcal{F}}$. Thus (F) holds.

It remains to show (T): On the one hand, suppose that $t = a\langle f \rangle$ and $(t,p) \in \delta_{\mathcal{T}}$. Then there is a q such that $(f,q) \in \delta_{\mathcal{F}}$ and $(q,a,p) \in \mathit{Up}$. By $\mathcal{A}_{\mathcal{F}}\, f$, $f \in [\![G]\!]\, r_q$, and by construction $p \to a\langle r_q \rangle \in R$. Thus $t \in [\![G]\!]\, p$. On the other hand, if $t \in [\![G]\!]\, p$, then there is a q such that $p \to a\langle r_q \rangle \in R$ and $f \in [\![G]\!]\, r_q$. By $\mathcal{A}_{\mathcal{F}}\, f$, $(f,q) \in \delta_{\mathcal{F}}$ and by construction, $(q,a,p) \in \mathit{Up}$. Hence $(t,p) \in \delta_{\mathcal{T}}$ and (T) holds. $\square$

A.2 Proof of Theorem 6.2

In order to prove $\mathcal{L}_D = \mathcal{L}_A$, we show by structural induction that for all t, f:

$$\begin{array}{rcl} \mathcal{A}_{\mathcal{T}}\, t & \equiv & (t,p) \in \delta^A_{\mathcal{T}} \text{ iff } p \in \delta^D_{\mathcal{T}}(t); \\ \mathcal{A}_{\mathcal{F}}\, f & \equiv & (t,q) \in \delta^A_{\mathcal{F}} \text{ iff } q \in \delta^D_{\mathcal{F}}(f); \end{array}$$

We have to show (E), (L) and (T). $\delta^D_{\mathcal{F}}(\epsilon) = I = \{q \mid q \in \delta^A_{\mathcal{F}}(\epsilon)\}$ and thus (E) holds. For (L), suppose that $\mathcal{A}_{\mathcal{T}}\, t$ and $\mathcal{A}_{\mathcal{F}}\, f$ hold. By definition, $(ft, q_1) \in \delta^A_{\mathcal{F}}$ iff there are p, q such that $(f,q) \in \delta^A_{\mathcal{F}}$, $(t,p) \in \delta^A_{\mathcal{T}}$ and $(q,p,q_1) \in \mathit{Side}$. By $\mathcal{A}_{\mathcal{T}}\, t$, $\mathcal{A}_{\mathcal{F}}\, f$ and definition of Side', this is true iff $q \in q' = \delta^D_{\mathcal{F}}(f)$, $p \in p' = \delta^D_{\mathcal{T}}(t)$ and $q_1 \in \mathit{Side}'(q', p')$. But this is equivalent to $q_1 \in \delta^D_{\mathcal{F}}(ft)$, and thus (L) holds. It remains to prove (T): Suppose that $\mathcal{A}_{\mathcal{F}}\, f$ holds and $t = a\langle f \rangle$. By definition, $(t,p) \in \delta^A_{\mathcal{T}}$ iff there is a q such that $(f,q) \in \delta^A_{\mathcal{F}}$ and $(q,a,p) \in \mathit{Up}$. By $\mathcal{A}_{\mathcal{F}}\, f$ and definition of Up', this is equivalent to $q \in q' = \delta^D_{\mathcal{F}}(f)$ and $p \in \mathit{Up}'_a\, q'$. This is true iff $p \in \delta^D_{\mathcal{T}}(t)$ by definition; thus (R) holds and $\mathcal{L}_D = \mathcal{L}_A$. $\square$

A.3 Proof of Theorem 6.4

The proof that $\mathcal{L}_A = \mathcal{L}_D$ is divided into two parts: Let us first show that $\mathcal{L}_A \subseteq \mathcal{L}_D$. We show by structural induction that (E), (L) and (T) hold with the following induction invariants:

$$\begin{array}{rcl} \mathcal{A}_{\mathcal{T}}\, t & \equiv & \text{If } (q_1, t, p) \in \delta^A_{\mathcal{T}} \text{ and } (q, q_1) \in q', \text{ then } (q_1, p) \in \delta^D_{\mathcal{T}}(q', t); \\ \mathcal{A}_{\mathcal{F}}\, f & \equiv & \text{If } (q_1, f, q_2) \in \delta^A_{\mathcal{F}} \text{ and } (q, q_1) \in q', \text{ then } (q, q_2) \in \delta^D_{\mathcal{F}}(q', f) \end{array}$$

Having shown (E)-(T), we know that $\mathcal{A}_{\mathcal{F}}$ holds for all $f \in \mathcal{F}_\Sigma$. Especially, for all $q_1 \in I, q_2 \in F$, if $(q_1, f, q_2) \in \delta^A_{\mathcal{F}}$ then $(q_1, q_2) \in \delta^D_{\mathcal{F}}(q'_0, f)$ and thus $\mathcal{L}_A \subseteq \mathcal{L}_D$. It remains to show that $\mathcal{L}_D \subseteq \mathcal{L}_A$. For this purpose we define new invariants $\mathcal{A}_{\mathcal{T}}$ and $\mathcal{A}_{\mathcal{F}}$:

$$\begin{array}{rcl} \mathcal{A}_{\mathcal{T}}\, t & \equiv & \text{If } (q_1, p) \in \delta^D_{\mathcal{T}}(q', t) \\ & & \text{then } (q, q_1) \in q' \text{ for some } q, \text{ and } (q_1, t, p) \in \delta^A_{\mathcal{T}}; \\ \mathcal{A}_{\mathcal{F}}\, f & \equiv & \text{If } (q, q_2) \in \delta^D_{\mathcal{F}}(q', f) \\ & & \text{then } (q, q_1) \in q' \text{ and } (q_1, f, q_2) \in \delta^A_{\mathcal{F}} \text{ for some } q_1 \end{array}$$

Having shown that $\mathcal{A}_{\mathcal{F}}$ holds for all forests, we know especially that for all $q_2 \in F$, if $(q_1, q_2) \in \delta^D_{\mathcal{F}}(q'_0, f)$ for some q_1 then $(q_1, q_1) \in q'_0$ and $(q_1, f, q_2) \in \delta^A_{\mathcal{F}}$. Thus $\mathcal{L}_D \subseteq \mathcal{L}_A$. □

A.4 Proof of Theorem 6.5

Let us show that each DLFA $B = (P, Q, \{q_0\}, I, Up, Side)$ accepting L_n must have at least 2^n states. For each $S \subseteq \{1, \ldots, n\}$ let f_S be the unary forest $x_1\langle x_2\langle \ldots \langle x_n\rangle\rangle\rangle$ with $x_i = a$ if $i \in S$ and $x_i = b$ otherwise. In other words, $f_S \in L_i$ iff $i \in S$. Let $\Diamond_b(f)$ be the unary forest $b\langle f\rangle$ for all f. Clearly, $f \in L_i$ iff $\Diamond^j_b(f) \in L_{i+j}$. Thus,

(i) $\Diamond^{n-i}_b(f_S) \in L_n$ iff $i \in S$.

Now, for a forest state q, define $\delta_b(q) = Side(q_0, Up_b\, q)$. Obviously, if $q = \delta_{\mathcal{F}}(f)$, then $\delta^i_b(q) = \delta_{\mathcal{F}}(\Diamond^i_b(f))$. Thus by (i):

(ii) $\delta^{n-i}_b(\delta_{\mathcal{F}}(f_S)) \in F$ iff $i \in S$.

Now suppose that there are $S, R \subseteq \{1, \ldots, n\}$ with $S \neq R$. W.l.o.g., there is a $i \in S$ with $i \notin R$. Let $q_S = \delta_{\mathcal{F}}(f_S)$ and $q_R = \delta_{\mathcal{F}}(f_R)$. By (ii), $\delta^{n-i}_b(q_S) \in F$ and $\delta^{n-i}_b(q_R) \notin F$, and hence $q_S \neq q_R$. Thus B must have at least as many forest states as there are subsets of $\{1, \ldots, n\}$, i.e., $\boldsymbol{\Omega}(2^n)$. □

A.5 Proofs of Theorems 6.6 and 6.7

In order to prove Theorem 6.6, we have to show that (a) $\mathcal{L}_{A_{\vec{G}}} \subseteq \mathcal{L}_G$ and (b) $\mathcal{L}_G \subseteq \mathcal{L}_{A_{\vec{G}}}$. In order to show (a), we define invariants $\mathcal{A}_{\mathcal{T}}$ and $\mathcal{A}_{\mathcal{F}}$ for structural induction:

$$\begin{array}{rcl} \mathcal{A}_{\mathcal{T}}\, t & \equiv & \text{If } x \in \delta_{\mathcal{T}}(q, t) \text{ then } t \in [\![G]\!]\, x; \\ \mathcal{A}_{\mathcal{F}}\, f & \equiv & \text{If } q \cap Y_j \subseteq \{y_{0,j}\},\ y \in \delta_{\mathcal{F}}(q, f) \text{ and } y \in F_j \text{ then } f \in [\![G]\!]\, r_j. \end{array}$$

According to Corollary 4.1, if we show (T) and (F), then $\mathcal{A}_{\mathcal{F}}$ holds especially for $j = 0$. Thus, by definition of $A_{\vec{G}}$, $\delta_{\mathcal{F}}(q_0, f) \in F$ iff $f \in [\![G]\!]\, r_0 = \mathcal{L}_G$. Let us start with (T): Let $t = a\langle f\rangle$ and $p = \delta_{\mathcal{T}}(q, t)$. Then there are q_1, q_2 such that $q_1 = Down_a\, q$, $q_2 = \delta_{\mathcal{F}}(q_1, f)$ and $p = Up_a\, q_2$. By definition of $A_{\vec{G}}$, $q_1 \cap Y_j \subseteq \{y_{0,j}\}$ and there is a $y \in F_j \cap q_2$ with $x \to a\langle r_j\rangle$ for some j. Thus we can apply $\mathcal{A}_\mathcal{F}\, f$ and get that $f \in [\![G]\!]\, r_j$. Moreover, $t \in [\![G]\!]\, x$ because $x \to a\langle r_j\rangle$, and (T) holds.

In order to show (F), let $f = t_1 \ldots t_n$ and suppose that $q \cap Y_j \subseteq \{y_{0,j}\}$, $q' = \delta_\mathcal{F}(q, f)$, and $y \in q'$ for some $y \in F_j$. Let $q_1 = q$ and for $i = 1, \ldots, n$, $p_i = \delta_\mathcal{T}(q_i, t_i)$ and $q_{i+1} = \mathit{Side}(q_i, p_i)$. Then $q_{n+1} = q'$. Using the definition of *Side*, one can easily see that there must be $x_1, \ldots, x_n$ and $y_1, \ldots, y_{n+1}$ such that $y_{n+1} = y$ and $y_i \in q_i$, $x_i \in p_i$ and $(y_i, x_i, y_{i+1}) \in \delta$. Because all Y_j are disjoint, $y_{n+1} \in Y_j$ implies that $y_i \in Y_j$ and $(y_i, x_i, y_{i+1}) \in \delta_j$ for all i. Because $q_1 \cap Y_j \subseteq \{y_{0,j}\}$, $y_1 = y_{0,j}$, and by Proposition 4.1, $x_1 \ldots x_n \in [\![r_j]\!]_\mathcal{R}$. Furthermore, $t_i \in [\![G]\!]\, x_i$ because $\mathcal{A}_\mathcal{T}\, t_i$ holds for all i. Thus, by definition, $f \in [\![G]\!]\, r_j$, (F) holds and $\mathcal{L}_{A_{\vec{G}}} \subseteq \mathcal{L}_G$.

It remains to show (b): $\mathcal{L}_G \subseteq \mathcal{L}_{A_{\vec{G}}}$. Again we define invariants $\mathcal{A}_\mathcal{F}$ and $\mathcal{A}_\mathcal{T}$:

$$\begin{array}{lcl} \mathcal{A}_\mathcal{T}\, t & \equiv & \text{If } t \in [\![G]\!]\, x,\ y \in q \text{ and } (y, x, y_1) \in \delta \text{ for some } y_1, \\ & & \text{then } x \in \delta_\mathcal{T}(q, t); \\ \mathcal{A}_\mathcal{F}\, f & \equiv & \text{If } f \in [\![G]\!]\, r_j \text{ and } y_{0,j} \in q, \text{ then } \delta_\mathcal{F}(q, f) \cap F_j \neq \emptyset. \end{array}$$

If we prove (T) and (F), then $\mathcal{A}_\mathcal{F}\, f$ holds for all f, particularly for $j = 0$. Thus, if $f \in [\![G]\!]\, r_0$, then $\delta_\mathcal{F}(q_0, f) \cap F_0 \neq \emptyset$ and thus $f \in \mathcal{L}_{A_{\vec{G}}}$. In order to prove (T), suppose that $t = a\langle f\rangle \in [\![G]\!]\, x$, $y \in q$ and $(y, x, y_1) \in \delta$. By definition of $[\![G]\!]$, there is a j such that $x \to a\langle r_j\rangle$ and $f \in [\![G]\!]\, r_j$. Now let $q_1 = \mathit{Down}_a\, q$ and $q_2 = \delta_\mathcal{F}(q_1, f)$. By construction, $y_{0,j} \in q_1$, we can apply $\mathcal{A}_\mathcal{F}\, f$ and obtain that $q_2 \cap \mathcal{F}_j \neq \emptyset$. Now, by definition of *Up*, $x \in \mathit{Up}_a\, q_2 = \delta_\mathcal{T}(q, t)$, and thus (T) holds.

It remains to show (F): Let $f = t_1 \ldots t_n \in [\![G]\!]\, r_j$ and $y_{0,j} \in q$. By definition of $[\![G]\!]$, there is a word $x_1 \ldots x_n \in [\![r_j]\!]_\mathcal{R}$ and $t_i \in [\![G]\!]\, x_i$ for all i. Thus, by Proposition 4.1, there are $y_1, \ldots, y_{n+1}$ with $y_1 = y_{0,j}$, $y_{n+1} \in F_j$ and $(y_i, x_i, y_{i+1}) \in \delta_j$ for all i. Let $q_1 = q$ and for $i = 1, \ldots, n$, define $p_i = \delta_\mathcal{T}(q_i, t_i)$ and $q_{i+1} = \mathit{Side}(q_i, p_i)$. Now, for all i, given that $y_i \in q_i$, it follows by $\mathcal{A}_\mathcal{T}\, t_i$ that $x_i \in p_i$, and by definition of *Side*, $y_{i+1} \in q_{i+1}$. With $y_1 \in q_1$, it follows by induction on i that $y_{n+1} \in q_{n+1} = \delta_\mathcal{F}(q, f)$. Thus (F) holds and $\mathcal{L}_G \subseteq \mathcal{L}_{A_{\vec{G}}}$. □

The proof of Theorem 6.7 is completely analogous to that of Theorem 6.6. The difference is that $A_{\overleftarrow{G}}$ proceeds from right to left; this is compensated by using the reverse Berry-Sethi construction in its definition.

A.6 Proof of Theorem 7.1

The proof is divided into three parts: We start by showing that $A_{\vec{G}}$ marks at least all matches of C as candidates; and that the left siblings of all candidates identified by $A_{\vec{G}}$ are such that the left context is fulfilled (Lemma A.1). Then we prove that $B_{\overleftarrow{G}}$ finds all matches of C (Lemma A.2). Finally we formulate as Lemma A.3, that $B_{\overleftarrow{G}}$ only indicates correct matches.

First of all note that $B_{\overleftarrow{G}}$ assign the same tree states as $A_{\vec{G}}$:

$$\text{if } \vec{\lambda}(\pi) = (\vec{q}_1, \vec{p}, \vec{q}) \text{ and } \overleftarrow{\lambda}(\pi) = (q_1, p, q), \text{ then } \vec{p} = p \text{ for all } \pi \in \Pi(f_0).$$

Let us now perform the first part of the proof by proving the following lemma:

Lemma A.1: For a path π with $last_{f_0}(\pi) = n$, let $\vec{\lambda}(\pi i) = (q_i, p_i, q_i')$ and $f_0[\pi i] = t_i$ for $i = 1, \dots, n$. Furthermore, if $\pi \neq \epsilon$, let $\vec{\lambda}(\pi) = (q, p, q')$ and $f_0[\pi] = t$ with $t = a\langle t_1 \dots t_n\rangle$. Then the following hold:

(1) If $x \in p$ then $t \in [\![G]\!]\, x$.

(2) If $y_{i+1} \in q_i' \cap Y_j$ then there are $x_1, \dots, x_i$ and $y_1, \dots, y_i$ with $y_1 = y_{0,j}$ such that $t_m \in [\![G]\!]\, x_m$ and $(y_m, x_m, y_{m+1}) \in \delta_j$ for $1 \leqslant m \leqslant i$.

(3) If $f_0 \rightsquigarrow_\pi x$ then $y \in q$ and $y' \in q'$ for some $(y, x, y') \in \delta$.

(4) If $f_0 \rightsquigarrow_\pi r_j$, $x_1 \dots x_n \in [\![r_j]\!]_{\mathcal{R}}$ and there are $y_1, \dots, y_{n+1}$ such that $y_1 = y_{0,j}$, and for $i = 1, \dots, n$, $(y_i, x_i, y_{i+1}) \in \delta_j$ and $f_0[\pi i] \in [\![G]\!]\, x_i$, then $x_i \in p_i$, $y_i \in q_i$ and $y_{i+1} \in q_i'$ for all i.

(1) and (2) are immediate consequences of $\mathcal{A}_\mathcal{T}$ and $\mathcal{A}_\mathcal{F}$ in the proof of Theorem 6.6 (a). The other two statements are proven by path induction with $\mathcal{A}_\mathcal{T} \equiv (3)$ and $\mathcal{A}_\mathcal{F} \equiv (4)$: We have to show (s)–(t).

Let us start with (s): If $f_0 \rightsquigarrow_\epsilon r_j$, then $j = 0$. Suppose that $x_1 \dots x_n \in [\![r_0]\!]_{\mathcal{R}}$, $y_1 = y_{0,0}$ and for $i = 1, \dots, n$, $(y_i, x_i, y_{i+1}) \in \delta_0$ and $f_0[\pi i] \in [\![G]\!]\, x_i$. Invariant $\mathcal{A}_\mathcal{T}$ from the proof of Theorem 6.6 (b) holds for all t_i. Thus, if y_i in q_i then $x_i \in p_i = \delta_\mathcal{T}(q_i, t_i)$ and also $y_{i+1} \in q_i' = q_{i+1}$. Because $y_{0,0} \in q_1 = q_0$, we can easily show by induction on i that $x_i \in p_i$, $y_i \in q_i$ and $y_{i+1} \in q_i'$ for all i, and thus (s) holds.

Let us show (f): Suppose that $\mathcal{A}_\mathcal{T}\ \pi$ holds, $x_1 \dots x_n \in [\![r_j]\!]_{\mathcal{R}}$ and there are $y_1, \dots, y_{n+1}$ such that $y_1 = y_{0,j}$, and for $i = 1, \dots, n$, $(y_i, x_i, y_{i+1}) \in \delta_j$ and $t_i \in [\![G]\!]\, x_i$. If $f_0 \rightsquigarrow_\pi r_j$, then there is an x with $x \to a\langle r_j\rangle$ and $f_0 \rightsquigarrow_\pi x$. Because of $\mathcal{A}_\mathcal{T}\ \pi$, there is a $y \in q$ with $(y, x, y_1) \in \delta$ for some y_1. By definition of *Down*, $y_{0,j} \in q_1$. Similarly to (s) we can show by induction on i, using invariant $\mathcal{A}_\mathcal{T}$ from the proof of Theorem 6.6 (b), that $x_i \in p_i$, $y_i \in q_i$ and $y_{i+1} \in q_i'$ for all i; hence (f) holds.

In order to show (t), suppose that $f_0 \rightsquigarrow_{\pi m} x$ with $1 \leqslant m \leqslant n$. Then there must be a j such that $f_0 \rightsquigarrow_\pi r_j$ and for some word $x_1 \dots x_n \in [\![r_j]\!]_{\mathcal{R}}$ with $x_m = x$, $t_i \in [\![G]\!]\, x_i$ for $i = 1, \dots, n$. By Proposition 4.1, there are $y_1, \dots, y_{n+1}$ with $y_1 = y_{0,j}$ and $(y_i, x_i, y_{i+1}) \in \delta_j$ for all i. Applying $\mathcal{A}_\mathcal{F}\ \pi$ we obtain that $x_i \in p_i$, $y_i \in q_i$ and $y_{i+1} \in q_i'$ for all i, and particularly for $i = m$. Thus (t) holds. □

We can now prove the first part of Theorem 7.1:

Lemma A.2: For a path π with $last_{f_0}(\pi) = n$, let $f_0[\pi i] = t_i$, $\vec{\lambda}(\pi i) = (\vec{q}_i, p_i, \vec{q}_i')$ and $\overleftarrow{\lambda}(\pi i) = (\overleftarrow{q}_i, p_i, \overleftarrow{q}_i')$ for $i = 1, \dots, n$. Moreover, if $\pi \neq \epsilon$, let $\vec{\lambda}(\pi) = (\vec{q}, p, \vec{q}')$, $\overleftarrow{\lambda}(\pi) = (\overleftarrow{q}, p, \overleftarrow{q}')$ and $f_0[\pi] = t$ with $t = a\langle t_1 \dots t_n\rangle$. Then the following hold:

(5) If $f_0 \rightsquigarrow_\pi x$ then $y' \in \vec{q}' \cap \overleftarrow{q}'$ for some $(y, x, y') \in \delta$.

(6) If $f_0 \rightsquigarrow_\pi r_j$ and $n > 0$ then $F_j \subseteq \overleftarrow{q}_1'$.

We prove this lemma by path induction with using $\mathcal{A}_\mathcal{T} \equiv (5)$ and $\mathcal{A}_\mathcal{F} \equiv (6)$. We start by showing (s): For $\pi = \epsilon$, j must be 0. Now $F_0 \subseteq \overleftarrow{q}_n'$ because $\overleftarrow{q}_n'$ is the start-state $\overleftarrow{q}_0 = F_0$ of $B_{\overleftarrow{G}}$.

For (f), suppose that $\mathcal{A}_\mathcal{T}\ \pi$ holds and $f_0 \rightsquigarrow_\pi r_j$. Then there is an x such that $x \to a\langle r_j\rangle$ and $f_0 \rightsquigarrow_\pi x$. By $\mathcal{A}_\mathcal{T}\ \pi$, there is a $y' \in \vec{q}' \cap \overleftarrow{q}'$ such that $(y, x, y') \in \delta$ for some y. Then by construction of $B_{\overleftarrow{G}}$, $F_j \subseteq \overleftarrow{q}_n' = \overleftarrow{Down}_{(a,p,\vec{q}')}\ \overleftarrow{q}$ and (f) holds.

Finally, let us prove (t): suppose that $\mathcal{A}_{\mathcal{F}}\ \pi$ holds and $f_0 \rightsquigarrow_{\pi m} x$. Then there is a j such that $f_0 \rightsquigarrow_{\pi} r_j$, and for some word $x_1 \ldots x_n \in \llbracket r_j \rrbracket_{\mathcal{R}}$ with $x_m = x$, $t_i \in \llbracket G \rrbracket\ x_i$ for $i = 1, \ldots, n$. By Proposition 4.1, there are $y_1, \ldots, y_{n+1}$ such that $y_1 = y_{0,j}$, $y_{n+1} \in F_j$ and for all i, $(y_i, x_i, y_{i+1}) \in \delta$. Thus, by (4), $x_i \in p_i$ and $y_{i+1} \in \vec{q}'_i$ for all i, and particularly $y_{m+1} \in \vec{q}'_m$. On the other hand, by construction of $B_G^{\overleftarrow{}}$, if $y_{i+1} \in \overleftarrow{q}'_i$, then $y_i \in \overleftarrow{q}_i$. By $\mathcal{A}_{\mathcal{F}}\ \pi$, $y_{n+1} \in \overleftarrow{q}_n{}'$, and we can show by induction on i that $y_{i+1} \in \overleftarrow{q}'_i$ for $i = 1, \ldots, n$, and especially for $i = m$, $y_{m+1} \in \overleftarrow{q}'_m$. Moreover, $(y_i, x_m, y_{i+1}) \in \delta$, and because $x = x_m$, (t) holds, and we are done with the proof of Lemma A.2. $\square$

Let us now complete the proof of Theorem 7.1. It follows from (5) in the previous lemma and (7) in the following lemma:

Lemma A.3: For a path π with $last_{f_0}(\pi) = n$, let $f_0[\pi i] = t_i$, $\vec{\lambda}(\pi i) = (\vec{q}_i, p_i, \vec{q}'_i)$ and $\overleftarrow{\lambda}(\pi i) = (\overleftarrow{q}_i, p_i, \overleftarrow{q}'_i)$ for $i = 1, \ldots, n$. Moreover, if $\pi \neq \epsilon$, let $\vec{\lambda}(\pi) = (\vec{q}, p, \vec{q}')$, $\overleftarrow{\lambda}(\pi) = (\overleftarrow{q}, p, \overleftarrow{q}')$ and $f_0[\pi] = t$ with $t = a\langle t_1 \ldots t_n \rangle$. Then the following hold:

(7) If $y' \in \vec{q}' \cap \overleftarrow{q}'$ and $(y, x, y') \in \delta$ for some y, then $f_0 \rightsquigarrow_{\pi} x$.

(8) If $n > 0$ and $\overleftarrow{q}'_n \cap \vec{q}'_n \cap F_j \neq \emptyset$ then $f_0 \rightsquigarrow_{\pi} r_j$.

We prove this lemma by path induction using $\mathcal{A}_{\mathcal{T}} \equiv (7)$ and $\mathcal{A}_{\mathcal{F}} \equiv (8)$. We start with (s): For $\pi = \epsilon$, $\overleftarrow{q}'_n = F_0$; thus j must be 0, and with $f_0 \in \mathcal{L}_G$, $f_0 \rightsquigarrow_{\epsilon} r_0$ by definition.

In order to show (f), suppose that $y_{n+1} \in \overleftarrow{q}'_n \cap \vec{q}'_n \cap F_j$. Then, by definition of $Down^{\leftarrow}$, there must be an x and a $y' \in \overleftarrow{q}' \cap \vec{q}'$ such that $(y, x, y') \in \delta$ for some y, and $x \to a\langle r_j \rangle$. By $\mathcal{A}_{\mathcal{T}}\ \pi$, $f_0 \rightsquigarrow_{\pi} x$. Moreover, by (2) there are $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ such that $y_1 = y_{0,j}$ and for $i = 1, \ldots, n$, $(y_i, x_i, y_{i+1}) \in \delta_j$ and $t_i \in \llbracket G \rrbracket\ x_i$. Hence $x_1 \ldots x_n \in \llbracket r_j \rrbracket_{\mathcal{R}}$ and $t_1 \ldots t_n \in \llbracket G \rrbracket\ r_j$. Thus $f_0 \rightsquigarrow_{\pi} r_j$.

It remains to show (t): Suppose that $y_{m+1} \in \vec{q}'_m \cap \overleftarrow{q}'_m$, and $(y, x, y') \in \delta$ for some y. By (2), there are $x_1, \ldots, x_m$ and $y_1, \ldots, y_m$ such that $y_1 = y_{0,j}$ and for $i = 1, \ldots, m$, $(y_i, x_i, y_{i+1}) \in \delta_j$ and $t_i \in \llbracket G \rrbracket\ x_i$. Moreover, for $m < i \leqslant n$, if $y_i \in \overleftarrow{q}_i$, then by construction of $B_G^{\overleftarrow{}}$, there are $y_{i+1} \in \overleftarrow{q}'_i$ and $x_i \in p_i$ with $(y_i, x_i, y_{i+1}) \in \delta_j$, and because $y_{m+1} \in \overleftarrow{q}'_m = \overleftarrow{q}_{m+1}$, we can show by induction that there are such x_i and y_{i+1} for all $i = m+1, \ldots, n$. Now this implies that also $y_{i+1} \in \vec{q}'_i$ by definition of $A_G^{\vec{}}$. Because $\overleftarrow{q}'_n \cap Y_j \subseteq F_j$, $y_{n+1} \in F_j$. Thus $\vec{q}'_n \cap \overleftarrow{q}'_n \cap F_j \neq \emptyset$ and by $\mathcal{A}_{\mathcal{F}}\ \pi$, $f_0 \rightsquigarrow_{\pi} r_j$. Moreover, $x_1 \ldots x_n \in \llbracket r_j \rrbracket_{\mathcal{R}}$, and by definition $f_0 \rightsquigarrow_{\pi} x_m$. By Proposition 4.3, $x = x_m$ and thus $f_0 \rightsquigarrow_{\pi} x$. $\square$

This completes the proof of Theorem 7.1.

A.7 Proof of Theorem 7.2

For this proof, we first show by structural induction the following invariants:

$$\begin{array}{lcl} \mathcal{A}_{\mathcal{T}}\ t & \equiv & \text{For } y \in q \text{ and } (y, x, y_1) \in \delta, \\ & & \text{there is a } k \in \delta_{\mathcal{T}}(q, t) \text{ with } x = x^k \text{ iff } t \in \llbracket G \rrbracket\ x; \\ \mathcal{A}_{\mathcal{F}}\ f & \equiv & \text{For } q \cap Y_j = \{y_{0,j}\},\ \delta_{\mathcal{F}}(q, f) \cap F_j \neq \emptyset \text{ iff } f \in \llbracket G \rrbracket\ r_j, \end{array}$$

By Corollary 4.1 it suffices to show (T) and (F); we start with (T): Suppose that $\mathcal{A}_{\mathcal{F}}\ f$ holds, and let $t = a\langle f \rangle$, $y \in q$, $(y, x, y_1) \in \delta$ and $p = \delta_{\mathcal{T}}(q, t)$. Then

$p = Up_a\, q_2$, where $q_2 = \delta_{\mathcal{F}}(q_1, f)$ and $q_1 = Down_a\, q$. Let us prove direction "$\Rightarrow$" of $\mathcal{A}_{\mathcal{T}}\, t$: Suppose that $k \in p$. By definition of *Up*, $R_k = x \to a\langle e_k \rangle$ and for all j with $e_k = \ldots \sqcap \sigma_j r_j \sqcap \ldots$, $q_2 \cap F_j \neq \emptyset$ iff $\sigma_j = +$. By definition of *Down*, $q_1 \cap Y_j = \{y_{0,j}\}$ for all these j. We can thus apply $\mathcal{A}_{\mathcal{F}}\, f$ and obtain that $f \in [\![G]\!]\, r_j$ iff $q_2 \cap F_j \neq \emptyset$. Thus by definition $t \in [\![G]\!]\, x$.

In order to show the other direction ("$\Leftarrow$") of $\mathcal{A}_{\mathcal{T}}\, t$, suppose that $t \in [\![G]\!]\, x$. Then there is a k with $R_k = x \to a\langle e_k \rangle$, and for all j with $e_k = \ldots \sqcap \sigma_j r_j \sqcap \ldots$, $f \in [\![G]\!]\, r_j$ iff $\sigma_j = +$. By definition of *Down*, $q_1 \cap Y_j = \{y_{0,j}\}$ for all these j, and $\mathcal{A}_{\mathcal{F}}\, f$ yields that $q_2 \cap F_j \neq \emptyset$ iff $\sigma_j = +$. Hence $k \in p$ by definition of *Up*, and (T) holds.

It remains to show (F): Let $f = t_1 \ldots t_n$, assume that $\mathcal{A}_{\mathcal{T}}\, t_i$ holds for $i = 1, \ldots, n$, and let $q \cap Y_j = \{y_{0,j}\}$. With $q_1 = q$, let $p_i = \delta_{\mathcal{T}}(q_i, t_i)$ and $q_{i+1} = Side(q_i, p_i)$ for $i = 1, \ldots, n$. Then $q_{n+1} = \delta_{\mathcal{F}}(q, f)$. Let us now show direction "$\Rightarrow$" of $\mathcal{A}_{\mathcal{F}}\, f$: Suppose that there is a $y_{n+1} \in q_{n+1} \cap F_j$. By definition of *Side*, if $y_{i+1} \in q_{i+1} \cap Y_j$, then there must be $k_i \in p_i$ and $y_i \in q_i \cap Y_j$ such that $(y_i, x^{k_i}, y_{i+1}) \in \delta_j$. Thus, by induction on i, there must be such k_i, y_i for all $i = 1, \ldots, n$. Moreover, $y_1 = y_{0,j}$ because $q_1 \cap Y_j = \{y_{0,j}\}$, and thus $x^{k_1} \ldots x^{k_n} \in [\![r_j]\!]_{\mathcal{R}}$ by Proposition 4.1. With $\mathcal{A}_{\mathcal{T}}\, t_i$ we get that $t_i \in [\![G]\!]\, x^{k_i}$ for all i and thus $f \in [\![G]\!]\, r_j$.

For the other direction ("$\Leftarrow$"), let $f \in [\![G]\!]\, r_j$. Then there is a word $x_1 \ldots x_n \in [\![r_j]\!]_{\mathcal{R}}$ such $t_i \in [\![G]\!]\, x_i$ for $i = 1, \ldots, n$. By Proposition 4.1, there are $y_1, \ldots, y_{n+1}$ such that $y_1 = y_{0,j}$, $y_{n+1} \in F_j$ and $(y_i, x_i, y_{i+1}) \in \delta_j$ for all i. Now, if $y_i \in q_i$, then by $\mathcal{A}_{\mathcal{T}}\, t_i$ there is a k_i with $x_i = x^{k_i}$ and $k_i \in p_i$, and thus by definition of *Side*, $y_{i+1} \in q_{i+1}$. Since $y_1 \in q_1$, we get by induction on i that $y_{n+1} \in q_{n+1}$ and thus $\delta_{\mathcal{F}}(q, f) \cap F_j \neq \emptyset$.

Having shown that $\mathcal{A}_{\mathcal{F}}$ holds for all f, we can now prove Theorem 7.2. For a forest f, let $q_1 = \delta_{\mathcal{F}}(q_0, f)$ and $e \in E_0$. By construction $q_0 \cap F_j = \{y_{0,j}\}$ for all j with $e = \ldots \sqcap \sigma_j r_j \sqcap \ldots$. Similarly to (T), we can show with the help of $\mathcal{A}_{\mathcal{F}}\, f$ that $f \in [\![G]\!]\, e$ iff $q_1 \cap F_j \neq \emptyset$ for $\sigma_j = +$ and $q_1 \cap F_j = \emptyset$ for $\sigma_j = \neg$. Hence $q_1 \in F$ iff $f \in [\![G]\!]\, e$ for some $e \in E_0$. This completes the proof of Theorem 7.2.

□

A.8 Proof of Theorem 7.3

The structure of the proof is to that of the proof for Theorem 7.1: First we show that $A_G^{\vec{}}$ identifies all matches of C as candidates, and that it generates enough information for $B_G^{\overleftarrow{}}$ to select the correct matches (Lemma A.4). Then we show with Lemma A.5 that $B_G^{\overleftarrow{}}$ indeed finds all matches, and that all matches identified by $B_G^{\overleftarrow{}}$ are correct (Lemma A.6).

Lemma A.4: For a path π with $last_{f_0}(\pi) = n$, let $\vec{\lambda}(\pi i) = (q_i, p_i, q_i')$ and $f_0[\pi i] = t_i$ for $i = 1, \ldots, n$. Furthermore, if $\pi \neq \epsilon$, let $\vec{\lambda}(\pi) = (q, p, q')$ and $f_0[\pi] = t$ with $t = a\langle t_1 \ldots t_n \rangle$. Then the following hold:

(1) If $k \in p$ then $t \in [\![G]\!]\, x^k$, and $t_1 \ldots t_n \in [\![G]\!]\, e_k$.

(2) If $y_{m+1} \in q_m' \cap Y_j$ then there are $k_1, \ldots, k_m$ and $y_1, \ldots, y_m$ with $y_1 = y_{0,j}$ such that $t_i \in [\![G]\!]\, x^{k_i}$ and $(y_i, x^{k_i}, y_{i+1}) \in \delta_j$ for $1 \leqslant i \leqslant m$.

(3) If $f_0 \rightsquigarrow_\pi x$ then $k \in p$, $y \in q$ and $y' \in q'$ for some $(y, x, y') \in \delta$ and some k with $x^k = x$.

(4) If $f_0 \rightsquigarrow_\pi r_j$, $x_1 \dots x_n \in [\![r_j]\!]_\mathcal{R}$ and there are $y_1, \dots, y_{n+1}$ such that $y_1 = y_{0,j}$, and for $i = 1, \dots, n$, $(y_i, x_i, y_{i+1}) \in \delta_j$ and $f_0[\pi i] \in [\![G]\!]\, x_i$, then there are $k_1, \dots, k_n$ with $x_i = x^{k_i}$ and $k_i \in p_i$, $y_i \in q_i$ and $y_{i+1} \in q_i'$ for all i.

(1) follows immediately from $\mathcal{A}_\mathcal{T}$ and $\mathcal{A}_\mathcal{F}$ in the proof of Theorem 7.2: If $k \in p$ then, by definition of *Up*, for all j with $e_k = \dots \sqcap \sigma r_j \sqcap \dots$, $q_n' \cap F_j \neq \emptyset$ iff $\sigma = +$. With $\mathcal{A}_\mathcal{F}\, f$ we get that $f \in [\![G]\!]\, r_j$ iff $\sigma = +$, and thus $f \in [\![G]\!]\, e_k$.

For (2), suppose that $y_{m+1} \in q_m' \cap Y_j$. By definition of *Side*, if $y_{i+1} \in q_i' \cap Y_j$, then there must be $k \in p_i$, $y_i \in q_i \cap F_j$ such that $(y_i, x^k, y_{i+1}) \in \delta_j$. Starting with $y_{m+1} \in q_m'$, we can show by induction on i that there must such k_i, y_i for all $i = 1, \dots, m$. Moreover, by definition of *Down* $y_1 = y_{0,j}$ and $\mathcal{A}_\mathcal{T}\, t$ yields that $t_i \in [\![G]\!]\, x^{k_i}$ for all i. Thus (2) holds.

The other two statements are proven by path induction with $\mathcal{A}_\mathcal{T} \equiv (3)$ and $\mathcal{A}_\mathcal{F} \equiv (4)$: We have to show (s)–(t).

Let us start with (s): Suppose that $x_1 \dots x_n \in [\![r_j]\!]_\mathcal{R}$, $y_1 = y_{0,j}$, and for $i = 1, \dots, n$, $(y_i, x_i, y_{i+1}) \in \delta_j$ and $f_0[\pi i] \in [\![G]\!]\, x_i$. If $f_0 \rightsquigarrow_\epsilon r_j$, then there is an $e \in E_0$ with $e = \dots \sqcap +r_j \sqcap \dots$. Because $\mathcal{A}_\mathcal{T}\, t_i$ from the proof of Theorem 7.2 holds for all i, if y_i in q_i then there is a k_i with $x_i = x^{k_i}$ and $k_i \in p_i = \delta_\mathcal{T}(q_i, t_i)$; hence $y_{i+1} \in q_i' = q_{i+1}$ by definition of *Side*. Because $y_{0,0} \in q_1 = q_0$, we can easily show by induction on i that $k_i \in p_i$, $y_i \in q_i$ and $y_{i+1} \in q_i'$ for all i, and thus (s) holds.

Let us show (f): Suppose that $\mathcal{A}_\mathcal{T}\, \pi$ holds, $x_1 \dots x_n \in [\![r_j]\!]_\mathcal{R}$ and there are $y_1, \dots, y_{n+1}$ such that $y_1 = y_{0,j}$, and for $i = 1, \dots, n$, $(y_i, x_i, y_{i+1}) \in \delta_j$ and $t_i \in [\![G]\!]\, x_i$. If $f_0 \rightsquigarrow_\pi r_j$, then there is an x with $x \to a\langle \dots \sqcap +r_j \sqcap \dots \rangle$ and $f_0 \rightsquigarrow_\pi x$. Because of $\mathcal{A}_\mathcal{T}\, \pi$, there is a $y \in q$ with $(y, x, y') \in \delta$ for some y', and $y_{0,j} \in q_1$. By invariant $\mathcal{A}_\mathcal{T}\, t_i$ from the proof of Theorem 7.2, if $y_i \in q_i$ then there is a $k_i \in p_i$ with $x_i = x^{k_i}$, and $y_{i+1} \in q_i' = q_{i+1}$ by definition of *Side*. Induction on i shows that $k_i \in p_i$, $y_i \in q_i$ and $y_{i+1} \in q_i'$ for all $i = 1, \dots, n$, and (f) holds.

In order to show (t), suppose that $n = last_{f_0}(\pi)$ and $f_0 \rightsquigarrow_{\pi m} x$ with $1 \leqslant m \leqslant n$. Then $f_0 \rightsquigarrow_\pi r_j$ for some j, and there is a word $x_1 \dots x_n \in [\![r_j]\!]_\mathcal{R}$ such that $x_m = x$ and $t_i \in [\![G]\!]\, x_i$ for $1 = 1, \dots, n$. By Proposition 4.1, there are $y_1, \dots, y_{n+1}$ with $y_1 = y_{0,j}$ and $(y_i, x_i, y_{i+1}) \in \delta_j$ for all i. Thus, by $\mathcal{A}_\mathcal{F}\, \pi$, there are $k_1, \dots, k_n$ such that $x_i = x^{k_i}$, $k_i \in p_i$, $y_i \in q_i$ and $y_{i+1} \in q_i'$ for all i. Particularly for $i = m$, $y_m \in q_m$, $y_{m+1} \in q_m'$, $k_m \in p$, $x^{k_m} = x$ and $(y_m, x_i, y_{m+1}) \in \delta_j$; thus (t) holds. □

We can now prove the first part of Theorem 7.3:

Lemma A.5: For a path π with $last_{f_0}(\pi) = n$, let $f_0[\pi i] = t_i$, $\vec{\lambda}(\pi i) = (\vec{q}_i, p_i, \vec{q}_i')$ and $\overleftarrow{\lambda}(\pi i) = (\overleftarrow{q}_i, p_i, \overleftarrow{q}_i')$ for $i = 1, \dots, n$. Moreover, if $\pi \neq \epsilon$, let $\vec{\lambda}(\pi) = (\vec{q}, p, \vec{q}')$, $\overleftarrow{\lambda}(\pi) = (\overleftarrow{q}, p, \overleftarrow{q}')$ and $f_0[\pi] = t$ with $t = a\langle t_1 \dots t_n \rangle$. Then the following hold:

(5) If $f_0 \rightsquigarrow_\pi x$ then $y \in \vec{q} \cap \overleftarrow{q}$ and $y' \in \vec{q}' \cap \overleftarrow{q}'$ for some $(y, x, y') \in \delta$.

(6) If $f_0 \rightsquigarrow_\pi r_j$ and $n > 0$ then $F_j \subseteq \overleftarrow{q}_n'$.

We prove this lemma by path induction with $\mathcal{A}_\mathcal{T} \equiv (5)$ and $\mathcal{A}_\mathcal{F} \equiv (6)$. We start by showing (s): If $f_0 \rightsquigarrow_\epsilon r_j$ then there is an $e \in E_0$ with $e = \dots \sqcap +r_j \sqcap \dots$, and $f_0 \in [\![G]\!]\, e$. Thus for all i with $e = \dots \sqcap \sigma r_i \sqcap \dots$, $f_0 \in [\![G]\!]\, r_i$ iff $\sigma = +$. By invariant $\mathcal{A}_{\mathcal{F}}$ from the proof of Theorem 7.2, $q_F \cap F_i \neq \emptyset$ iff $\sigma = +$. Hence, by construction of $B_{\overleftarrow{G}}$, $F_j \subseteq \overleftarrow{q}_0 = \overleftarrow{q}'_n$.

For (f), suppose that $\mathcal{A}_{\mathcal{T}}\ \pi$ holds and $f_0 \rightsquigarrow_{\pi} r_j$. Then there is a k such that $R_k = x^k \to a\langle e_k \rangle$ with $e_k = \ldots \sqcap +r_j \sqcap \ldots$, $t_1 \ldots t_n \in [\![G]\!]\, e_k$, and $f_0 \rightsquigarrow_{\pi} x^k$. Thus for all i with $e_k = \ldots \sqcap \sigma r_i \sqcap \ldots$, $t_1 \ldots t_n \in [\![G]\!]\, r_i$ iff $\sigma = +$. By $\mathcal{A}_{\mathcal{T}}\ \pi$, there are $y \in \vec{q} \cap \overleftarrow{q}$ and $y' \in \vec{q}' \cap \overleftarrow{q}'$ such that $(y, x^k, y') \in \delta$. Hence $\vec{q}_1 \cap \mathcal{F}_i = \{y_{0,i}\}$ by definition of *Down*, and with invariant $\mathcal{A}_{\mathcal{F}}$ from the proof of Theorem 7.2, $\vec{q}'_n \cap F_i \neq \emptyset$ iff $\sigma = +$, and by definition of *Up*, $k \in p$. Thus, by definition of $\mathit{Down}^{\leftharpoonup}$, $F_j \subseteq \overleftarrow{q}'_n = \mathit{Down}^{\leftharpoonup}_{(a,p,\vec{q}')}\ \overleftarrow{q}$, and (f) holds.

Finally, let us prove (t): suppose that $\mathcal{A}_{\mathcal{F}}\ \pi$ holds and $f_0 \rightsquigarrow_{\pi m} x$. Then there is a j such that $f_0 \rightsquigarrow_{\pi} r_j$, and for some word $x_1 \ldots x_n \in [\![r_j]\!]_{\mathcal{R}}$ with $x_m = x$, $t_i \in [\![G]\!]\, x_i$ for $1 = 1, \ldots, n$. By Proposition 4.1, there are $y_1, \ldots, y_{n+1}$ such that $y_1 = y_{0,j}$, $y_{n+1} \in F_j$ and for all i, $(y_i, x_i, y_{i+1}) \in \delta$. Thus, by (4), there is a $k_i \in p_i$ with $x^{k_i} = x_i$, $y_i \in \vec{q}_i$ and $y_{i+1} \in \vec{q}'_i$ for all i, and particularly $y_m \in \vec{q}_m$ and $y_{m+1} \in \vec{q}'_m$. We also have to show that $y_m \in \overleftarrow{q}_m$ and $y_{m+1} \in \overleftarrow{q}'_m$: By definition of $\mathit{Side}^{\leftharpoonup}$, if $y_{i+1} \in \overleftarrow{q}'_i$ and $k_i \in p_i$, then $y_i \in \overleftarrow{q}_i = \overleftarrow{q}'_{i-1}$. By $\mathcal{A}_{\mathcal{F}}\ \pi$, $y_{n+1} \in \overleftarrow{q}_n{}'$, and we can show by induction on i that $y_i \in \overleftarrow{q}_i$ and $y_{i+1} \in \overleftarrow{q}'_i$ for $i = 1, \ldots, n$, and especially for $i = m$, $y_m \in \overleftarrow{q}_m$ and $y_{m+1} \in \overleftarrow{q}'_m$. Moreover, $(y_m, x_m, y_{m+1}) \in \delta$, and because $x = x_m$, (t) holds, and we are done with the proof of Lemma A.5.

□

Let us now complete the proof of Theorem 7.3. It follows from (5) in the previous lemma and (7) in the following lemma:

Lemma A.6: For a path π with $\mathit{last}_{f_0}(\pi) = n$, let $f_0[\pi i] = t_i$, $\vec{\lambda}(\pi i) = (\vec{q}_i, p_i, \vec{q}'_i)$ and $\overleftarrow{\lambda}(\pi i) = (\overleftarrow{q}_i, p_i, \overleftarrow{q}'_i)$ for $i = 1, \ldots, n$. Moreover, if $\pi \neq \epsilon$, let $\vec{\lambda}(\pi) = (\vec{q}, p, \vec{q}')$, $\overleftarrow{\lambda}(\pi) = (\overleftarrow{q}, p, \overleftarrow{q}')$ and $f_0[\pi] = t$ with $t = a\langle t_1 \ldots t_n \rangle$. Then the following hold:

(7) If $y' \in \vec{q}' \cap \overleftarrow{q}'$ and $(y, x, y') \in \delta$ for some y, then $f_0 \rightsquigarrow_{\pi} x$.

(8) If $n > 0$ and $\overleftarrow{q}'_n \cap \vec{q}'_n \cap F_j \neq \emptyset$ then $f_0 \rightsquigarrow_{\pi} r_j$.

We prove this lemma by path induction using $\mathcal{A}_{\mathcal{T}} \equiv (7)$ and $\mathcal{A}_{\mathcal{F}} \equiv (8)$. We start with (s): For $\pi = \epsilon$, $\lambda(\pi n) = \lambda(n) = \overleftarrow{q}_0$. By construction there must be an $e \in E_0$ with $e = \ldots \sqcap +r_j \sqcap \ldots$, and for all i with $e = \ldots \sqcap +r_i \sqcap \ldots$, $F_i \cap q_F \neq \emptyset$ iff $\sigma = +$. With invariant $\mathcal{A}_{\mathcal{F}}$ from the proof of Theorem 7.2, this implies that $f_0 \in [\![G]\!]\, r_i$ iff $\sigma = +$. Thus $f_0 \in [\![G]\!]\, e$, $f_0 \rightsquigarrow_{\epsilon} e$ and also $f_0 \rightsquigarrow_{\epsilon} r_j$.

In order to show (f), suppose that $y_{n+1} \in \overleftarrow{q}'_n \cap \vec{q}'_n \cap F_j$. Then, by definition of $\mathit{Down}^{\leftharpoonup}$, there must be a $k \in p$ and a $y' \in \overleftarrow{q}' \cap \vec{q}'$ such that $(y, x^k, y') \in \delta$ for some y, and $R_k = x \to a\langle e_k \rangle$ with $e_k = \ldots \sqcap +r_j \sqcap \ldots$. By $\mathcal{A}_{\mathcal{T}}\ \pi$, $f_0 \rightsquigarrow_{\pi} x$, and with (1) we obtain that $t_1 \ldots t_n \in [\![G]\!]\, e_k$. Thus $f_0 \rightsquigarrow_{\pi} e_k$, and because r_j occurs positively in e_k, also $f_0 \rightsquigarrow_{\pi} r_j$.

It remains to show (t): Suppose that $y_{m+1} \in \vec{q}'_m \cap \overleftarrow{q}'_m$, and $(y, x, y_{m+1}) \in \delta$ for some y. By (2), there are $x_1, \ldots, x_m$ and $y_1, \ldots, y_m$ such that $y_1 = y_{0,j}$ and for $i = 1, \ldots, m$, $(y_i, x_i, y_{i+1}) \in \delta_j$ and $t_i \in [\![G]\!]\, x_i$. Moreover, for $m < i \leqslant n$, if $y_i \in \overleftarrow{q}_i$, then by definition of $\mathit{Side}^{\leftharpoonup}$, there are $y_{i+1} \in \overleftarrow{q}'_i$ and $k_i \in p_i$ with $(y_i, x_i, y_{i+1}) \in \delta_j$ and $x_i = x^{k_i}$. Starting with $y_{m+1} \in \overleftarrow{q}'_m = \overleftarrow{q}_{m+1}$, we can show by induction that there are such k_i, x_i and y_{i+1} for all $i = m+1, \ldots, n$. Now this implies that also $y_{i+1} \in \vec{q}'_i$ by definition of $A_{\vec{G}}$'s side-relation. Because $\overleftarrow{q}'_n \cap Y_j \subseteq F_j$, $y_{n+1} \in F_j$. Thus $\vec{q}'_n \cap \overleftarrow{q}'_n \cap F_j \neq \emptyset$ and by $\mathcal{A}_{\mathcal{F}}\ \pi$, $f_0 \rightsquigarrow_{\pi} r_j$. Moreover,

$x_1 \ldots x_n \in [\![r_j]\!]_{\mathcal{R}}$, and by definition $f_0 \rightsquigarrow_\pi x_m$. By Proposition 4.3, $x = x_m$ and thus $f_0 \rightsquigarrow_\pi x$.

This completes the proof of Theorem 7.3. □

A.9 Proof of Theorem 7.4

Before we prove the theorem, we show the following lemma, which states that for a right-ignoring regular expression r, the right siblings of a node matching x need not be inspected:

Lemma A.7: Let G be a forest grammar, $A_{\vec{G}}$ be defined as above, and r_j be right-ignoring w.r.t. x. Moreover, for some $f = t_1 \ldots t_n$ and $1 \leqslant m \leqslant n$, suppose that there are $x_1, \ldots, x_m$ and $y_1, \ldots, y_{m+1}$ such that $x = x_m$, $y_1 = y_{0,j}$, and for $1 \leqslant i \leqslant m$, $t_i \in [\![G]\!]\, x_i$ and $(y_i, x_i, y_{i+1}) \in \delta$. Then there are $y_{m+2}, \ldots, y_{n+1}$ such that $(y_i, x_\top, y_{i+1}) \in \delta$ for $m < i \leqslant n$, $y_{n+1} \in F_j$, and $f \in [\![G]\!]\, re_j$.

Proof: Because r_j is right-ignoring w.r.t. x, there is a $Y_\top \subseteq F_j$ such that $y_{m+1} \in Y_\top$, and for each $y \in Y_\top$ there is a y' such that $(y, x_\top, y') \in \delta_j$. Thus there must be $(y_{m+2}, \ldots, y_{n+1})$ with $(y_i, x_\top, y_{i+1}) \in \delta$. For $i = m+1, \ldots, n$ define $x_i = x_\top$. Because $[\![G]\!]\, x_\top = \mathcal{T}_\Sigma$, $t_i \in [\![G]\!]\, x_i$ for $i = m+1, \ldots, n$, and by Proposition 4.1, $x_1 \ldots x_n \in [\![r_j]\!]_{\mathcal{R}}$. Thus $f \in [\![G]\!]\, r_j$. □

Theorem 7.4 is now proven by path induction with the following invariant $\mathcal{A}_{\mathcal{T}}$. Let λ be the $A_{\vec{G}}$-labeling of f_0, $\pi \in \Pi(f_0)$ and $\lambda(\pi) = (q, p, q')$. Then:

$$\mathcal{A}_{\mathcal{T}}\ \pi \quad \equiv \quad \text{If } y' \in q',\ (y, x, y') \in \delta \text{ for some } y \text{ and } x \text{ is match-relevant, then } f_0 \rightsquigarrow_\pi x.$$

Because all $x_\circ \in X_\circ$ are match-relevant, the theorem follows immediately once $\mathcal{A}_{\mathcal{T}}$ is proven to hold for all paths. By Corollary 4.2, it suffices to show (o) $\mathcal{A}_{\mathcal{T}}\ m$ holds for $1 \leqslant m \leqslant \mathit{last}_{f_0}(\epsilon)$, and (d) if $\mathcal{A}_{\mathcal{T}}\ \pi$ holds, then so does $\mathcal{A}_{\mathcal{T}}\ \pi m$ for $1 \leqslant m \leqslant \mathit{last}_{f_0}(\pi)$.

Let us start with (o): Let $f_0 = t_1 \ldots t_n$, $\pi = m$ with $1 \leqslant m \leqslant n$, and for $i = 1, \ldots, n$, $\lambda(i) = (q_i, p_i, q'_i)$. Now suppose that $y_{m+1} \in q'_m$ and $(y, x, y_{m+1}) \in \delta_j$. By Lemma A.4 (2), there are $k_1, \ldots, k_m$ and $y_1 \in q_1, \ldots, y_m \in q_m$ such that $(y_i, x^{k_i}, y_{i+1}) \in \delta_j$ and $t_i \in [\![G]\!]\, x^{k_i}$ for all i, and $y_1 = y_{0,j}$. Because $q_1 = q_0$, there is an $e \in E_0$ with $e = \ldots \sqcap \sigma r_j \sqcap \ldots$ by construction of $A_{\vec{G}}$. By Proposition 4.3, $x^{k_m} = x$ and because x is match-relevant, so is e_0. Thus, because C is right-ignoring, $e_0 = r_j$ and r_j is right-ignoring w.r.t. x. By Lemma A.7, there are $y_{m+2}, \ldots, y_{n+1}$ such that $(y_i, x_\top, y_{i+1}) \in \delta$ for $m < i \leqslant n$, $y_{n+1} \in F_j$, and $f \in [\![G]\!]\, r_j$. Thus $f_0 \rightsquigarrow_\epsilon e_0$ and also $f_0 \rightsquigarrow_m x$, and we are done with (o).

It remains to show (d): Suppose that $\mathcal{A}_{\mathcal{T}}$ holds for π. Let $n = \mathit{last}_{f_0}(\pi)$ and for $i = 1, \ldots, n$ let $t_i = f_0[\pi i]$ and $(q_i, p_i, q'_i) = \lambda(\pi i)$. Moreover, let $t = a\langle f \rangle$ with $f = t_1 \ldots t_n$ and $(q, p, q') = \lambda(\pi)$. Now suppose that $y_{m+1} \in q'_m$ and $(y, x, y_{m+1}) \in \delta_j$. By Lemma A.4 (2), there are $k_1, \ldots, k_m$ and $y_1 \in q_1, \ldots, y_m \in q_m$ such that $(y_i, x^{k_i}, y_{i+1}) \in \delta_j$ and $t_i \in [\![G]\!]\, x^{k_i}$ for all i, and $y_1 = y_{0,j}$. By definition of *Down*, t here must be a $y \in q$ such that $(y, x', y') \in \delta$, $x' \to a\langle e \rangle$ and $e = \ldots \sqcap \sigma r_j \sqcap \ldots$. Because x is match-relevant, so are e and x', and because C is right-ignoring, $e = r_j$ and r_j is right-ignoring. By $\mathcal{A}_{\mathcal{T}}\ \pi$ we have that $f_0 \rightsquigarrow_\pi x'$.Moreover, by Lemma A.7, there are $y_{m+2}, \ldots, y_{n+1}$ such that

$(y_i, x_\top, y_{i+1}) \in \delta$ for $m < i \leqslant n$, $y_{n+1} \in F_j$, and $f \in [\![G]\!]\, r_j$. Thus also $f_0 \rightsquigarrow_{\pi m} x^{k_m}$, and by Proposition 4.3 $x^{k_m} = x$, and the proof of Theorem 7.4 is complete. □

Index