Grammars and Parsing

Johan Jeuring
Doaitse Swierstra

Contents

Voorwoord

Het hiervolgende dictaat is gebaseerd op teksten uit vorige jaren, die onder andere geschreven zijn in het kader van het project *Kwaliteit en Studeerbaarheid*.

Het dictaat is de afgelopen jaren verbeterd, maar we houden ons van harte aanbevolen voor suggesties voor verdere verbetering, met name daar waar het het aangeven van verbanden met andere vakken betreft.

Veel mensen hebben een bijgedrage geleverd aan de totstandkoming van dit dictaat door een gedeelte te schrijven, of (een gedeelte van) het dictaat te becommentariëren. Speciale vermelding verdienen Jeroen Fokker, Rik van Geldrop, en Luc Duponcheel, die mee hebben geholpen door het schrijven van (een) hoofdstuk(ken) van het dictaat. Commentaar is onder andere geleverd door: Arthur Baars, Arnoud Berendsen, Gijsbert Bol, Breght Boschker, Martin Bravenboer, Pieter Eendebak, Rijk-Jan van Haaften, Graham Hutton, Daan Leijen, Andres Löh, Erik Meijer, en Vincent Oostindië.

Tenslotte willen we van de gelegenheid gebruik maken enige *studeeraanwijzingen* te geven:

- Het is onze eigen ervaring dat het uitleggen van de stof aan iemand anders vaak pas duidelijk maakt welke onderdelen je zelf nog niet goed beheerst. Als je dus van mening bent dat je een hoofdstuk goed begrijpt, probeer dan eens *in eigen woorden uiteen te zetten*.

- *Oefening baart kunst.* Naarmate er meer aandacht wordt besteed aan de presentatie van de stof, en naarmate er meer voorbeelden gegeven worden, is het verleidelijker om, na lezing van een hoofdstuk, de conclusie te trekken dat je een en ander daadwerkelijk beheerst. “Begrijpen is echter niet hetzelfde als “kennen”, “kennen” is iets anders dan “beheersen” en “beheersen” is weer iets anders dan “er iets mee kunnen”. Maak dus de opgaven die in het dictaat opgenomen zijn zelf, en doe dat niet door te kijken of je de oplossingen die anderen gevonden hebben, begrijpt. Probeer voor jezelf bij te houden welk stadium je bereikt hebt met betrekking tot alle genoemde leerdoelen. In het ideale geval zou je in staat moeten zijn een mooi tentamen in elkaar te zetten voor je mede-studenten!

- *Zorg dat je up-to-date bent.* In tegenstelling tot sommige andere vakken is het bij dit vak gemakkelijk de vaste grond onder je voeten kwijt te raken. Het is niet “elke week nieuwe kansen”. We hebben geprobeerd door de indeling van de stof hier wel iets aan te doen, maar de totale opbouw laat

hier niet heel veel vrijheid toe. Als je een week gemist hebt is het vrijwel onmogelijk de nieuwe stof van de week daarop te begrijpen. De tijd die je dan op college en werkcollege doorbrengt is dan weinig effectief, met als gevolg dat je vaak voor het tentamen heel veel tijd (die er dan niet is) kwijt bent om in je uppie alles te bestuderen.

- We maken gebruik van de taal Haskell om veel concepten en algoritmen te presenteren. Als je nog moeilijkheden hebt met de taal Haskell aarzel dan niet direct hier wat aan te doen, en zonodig hulp te vragen. Anders maak je jezelf het leven heel moeilijk. Goed gereedschap is het halve werk, en Haskell is hier ons gereedschap.

Veel sterkte, en hopelijk ook veel plezier,

Johan Jeuring en Doaitse Swierstra

Voorwoord
Open Universiteit Nederland

Het dictaat is bijgewerkt door Johan Jeuring en Manuela Witsiers ten behoeve van de cursus Talen en ontleders van de Open Universiteit Nederland.
Bijwerkingen hebben betrekking op de hoofdstukken 2, 3, 4, 9, 10 en 11.

Chapter 1

Goals

INTRODUCTION

Courses on *Grammars*, *Parsing* and *Compilation of programming languages* have always been some of the core components of a computer science curriculum. The reason for this is that from the very beginning of these curricula it has been one of the few areas where the development of formal methods and the application of formal techniques in actual program construction come together. For a long time the construction of compilers has been one of the few areas where we had a methodology available, where we had tools for generating parts of compilers out of formal descriptions of the tasks to be performed, and where such program generators were indeed generating programs which would have been impossible to create by hand. For many practicing computer scientists the course on compiler construction still is one of the highlights of their education.

One of the things which were not so clear however is where exactly this joy originated from: the techniques taught definitely had a certain elegance, we could construct programs someone else could not —thus giving us the feeling we had "the right stuff"—, and when completing the practical exercises, which invariably consisted of constructing a compiler for some toy language, we had the usual satisfied feeling. This feeling was augmented by the fact that we would not have had the foggiest idea how to complete such a product a few months before, and now we knew "how to do it".

This situation has remained so for years, and it is only in the last years that we have started to discover and make explicit the reasons why this area attracted so much interest. Many of the techniques which were taught on a "this is how you solve this kind of problems" basis, have been provided with a theoretical underpinning which explains why the techniques work. As a beneficial side-effect we also gradually learned to see where the discovered concept further played a rôle, thus linking the area with many other areas of computer science; and not only that, but also giving us a means to explain such links, stress their importance, show correspondences and transfer insights from one area of interest to the other.

GOALS

The goals of these lecture notes can be split into primary goals, which are associated with the specific subject studied, and secondary —but not less important— goals which have to do with developing skills which one would expect every educated computer scientist to have. The primary, somewhat more traditional, goals are:

- to understand how to describe structures (i.e. "formulas") using *grammars*;
- to know how to *parse*, i.e. how to recognise (build) such structures in (from) a sequence of symbols;
- to know how to *analyse grammars* to see whether or not specific properties hold;
- to understand the concept of *compositionality*;
- to be able to *apply these techniques* in the construction of all kinds of programs;
- to familiarise oneself with the concept of *computability*.

The secondary, more far reaching, goals are:

- to develop the capability *to abstract*;
- to understand the concepts of *abstract interpretation* and *partial evaluation*;
- to understand the concept of *domain specific languages*;
- to show how proper formalisations can be used as a starting point for the *construction of useful tools*;
- to improve the general *programming skills*;
- to show a wide variety of useful *programming techniques*;
- to show how to develop programs in a *calculational style*.

1.1 History

When at the end of the fifties the use of computers became more and more widespread, and their reliability had increased enough to justify applying them to a wide range of problems, it was no longer the actual hardware which posed most of the problems. Writing larger and larger programs by more and more people sparked the development of the first more or less machine-independent programming language FORTRAN (FORmula TRANslator), which was soon to be followed by ALGOL-60 and COBOL.

For the developers of the FORTRAN language, of which John Backus was the prime architect, the problem of how to describe the language was not a hot issue: much more important problems were to be solved, such as, what should be in the language and what not, how to construct a compiler for the language that would fit into the small memories which were available at that time (kilobytes instead of megabytes), and how to generate machine code that would not be ridiculed by programmers who had thus far written such code by hand. As a result the language was very much implicitly defined by what was accepted by

the compiler and what not.

Soon after the development of FORTRAN an international working group started to work on the design of a machine independent high-level programming language, to become known under the name ALGOL-60. As a remarkable side-effect of this undertaking, and probably caused by the need to exchange proposals in writing, not only a language standard was produced, but also a notation for describing programming languages was proposed by Naur and used to describe the language in the famous Algol-60 report. Ever since it was introduced, this notation, which soon became to be known as the Backus-Naur formalism (BNF), has been used as the primary tool for describing the basic structure of programming languages.

It was not for long that computer scientists, and especially people writing compilers, discovered that the formalism was not only useful to express what language should be accepted by their compilers, but could also be used as a guideline for structuring their compilers. Once this relationship between a piece of BNF and a compiler became well understood, programs emerged which take such a piece of language description as input, and produce a skeleton of the desired compiler. Such programs are now known under the name *parser generators*.

Besides these very mundane goals, i.e., the construction of compilers, the BNF-formalism also became soon a subject of study for the more theoretically oriented. It appeared that the BNF-formalism actually was a member of a hierarchy of *grammar classes* which had been formulated a number of years before by the linguist Noam Chomsky in an attempt to capture the concept of a "language". Questions arose about the *expressibility* of BNF, i.e., which classes of languages can be expressed by means of BNF and which not, and consequently how to express restrictions and properties of languages for which the BNF-formalism is not powerful enough. In the lectures we will see many examples of this.

1.2 Grammar analysis of context-free grammars

Nowadays the use of the word Backus-Naur is gradually diminishing, and, inspired by the Chomsky hierarchy, we most often speak of *context-free grammars*. For the construction of everyday compilers for everyday languages it appears that this class is still a bit too large. If we use the full power of the context-free languages we get compilers which in general are inefficient, and probably not so good in handling erroneous input. This latter fact may not be so important from a theoretical point of view, but it is from a pragmatical point of view. Most invocations of compilers still have as their primary goal to discover mistakes made when typing the program, and not so much generating actual code. This aspect is even stronger present in strongly typed languages, such as Java and Hugs, where the type checking performed by the compilers is one of the main contributions to the increase in efficiency in the programming process.

When constructing a recogniser for a language described by a context-free gram-

mar one often wants to check whether or not the grammar has specific desirable properties. Unfortunately, for a human being it is not always easy, and quite often practically impossible, to see whether or not a particular property holds. Furthermore, it may be very expensive to check whether or not such a property holds. This has led to a whole hierarchy of context-free grammars classes, some of which are more powerful, some are easy to check by machine, and some are easily checked by a simple human inspection. In this course we will see many examples of such classes. The general observation is that the more precise the answer to a specific question one wants to have, the more computational effort is needed and the sooner this question cannot be answered by a human being anymore.

1.3 Compositionality

As we will see the structure of many compilers follows directly from the grammar that describes the language to be compiled. Once this phenomenon was recognised it went under the name *syntax directed compilation*. Under closer scrutiny, and under the influence of the more functional oriented style of programming, it was recognised that actually compilers are a special form of homomorphisms, a concept thus far only familiar to mathematicians and more theoretically oriented computer scientist that study the description of the meaning of a programming language.

This should not come as a surprise since this recognition is a direct consequence of the tendency that ever greater parts of compilers are more or less automatically generated from a formal description of some aspect of a programming language; e.g. by making use of a description of their outer appearance or by making use of a description of the semantics (meaning) of a language. We will see many examples of such mappings. As a side effect you will acquire a special form of writing functional programs, which makes it often surprisingly simple to solve at first sight rather complicated programming assignments. We will see that the concept of *lazy evaluation* plays an important rôle in making these efficient and straightforward implementations possible.

1.4 Abstraction mechanisms

One of the main reasons for that what used to be an endeavour for a large team in the past can now easily be done by a couple of first year's students in a matter of days or weeks, is that over the last thirty years we have discovered the right kind of abstractions to be used, and an efficient way of partitioning a problem into smaller components. Unfortunately there is no simple way to teach the techniques which have led us thus far. The only way we see is to take a historians view and to compare the old and the new situations.

Fortunately however there have also been some developments in programming language design, of which we want to mention the developments in the area of functional programming in particular. We claim that the combination of a

modern, albeit quite elaborate, type system, combined with the concept of lazy evaluation, provides an ideal platform to develop and practice ones abstraction skills. There does not exist another readily executable formalism which may serve as an equally powerful tool. We hope that by presenting many algorithms, and fragments thereof, in a modern functional language, we can show the real power of abstraction, and even find some inspiration for further developments in language design: i.e. find clues about how to extend such languages to enable us to make common patterns, which thus far have only been demonstrated by giving examples, explicit.

Chapter 2

Context-Free Grammars

INTRODUCTION

We often want to recognise a particular structure hidden in a sequence of symbols. For example, when reading this sentence, you automatically structure it by means of your understanding of the English language. Of course, not any sequence of symbols is an English sentence. So how do we characterise English sentences? This is an old question, which was posed long before computers were widely used; in the area of natural language research the question has often been posed what actually constitutes a "language". The simplest definition one can come up with is to say that the English language equals the set of all grammatically correct English sentences, and that a sentence consists of a sequence of English words. This terminology has been carried over to computer science: the programming language Java can be seen as the set of all correct Java programs, whereas a Java program can be seen as a sequence of Java symbols, such as identifiers, reserved words, specific operators etc.

This chapter introduces the most important notions of this course: the concept of a *language* and a *grammar*. A language is a, possibly infinite, set of sentences and sentences are sequences of symbols taken from a finite set (e.g. sequences of characters, which are referred to as strings). Just as we say that the fact whether or not a sentence belongs to the English language is determined by the English grammar (remember that before we have used the phrase "grammatically correct"), we have a grammatical formalism for describing artificial languages.

A difference with the grammars for natural languages is that this grammatical formalism is a completely formal one. This property may enable us to mathematically prove that a sentence belongs to some language, and often such proofs can be constructed automatically by a computer in a process called *parsing*. Notice that this is quite different from the grammars for natural languages, where one may easily disagree about whether something is correct English or not. This completely formal approach however also comes with a disadvantage; the expressiveness of the class of grammars we are going to describe in this chapter is rather limited, and there are many languages one might want to describe but which cannot be described, given the limitations of the formalism

GOALS

The main goal of this chapter is to introduce and show the relation between the main concepts for describing the parsing problem: languages and sentences, and grammars.

In particular, after you have studied this chapter you will:

- know the concepts of *language* and *sentence*;
- know how to describe languages by means of *context-free grammars*;
- know the difference between a *terminal* symbol and a *nonterminal* symbol;
- be able to read and interpret the *BNF* notation;
- understand the *derivation* process used in describing languages;
- understand the rôle of *parse trees*;
- understand the relation between context-free grammars and datatypes;
- understand the *EBNF* formalism;
- understand the concepts of *concrete* and *abstract syntax*;
- be able to convert a grammar from EBNF-notation into BNF-notation by hand;
- be able to construct a simple context-free grammar in EBNF notation;
- be able to verify whether or not a simple grammar is *ambiguous*;
- be able to *transform* a grammar, for example for removing left recursion.

2.1 Languages

The goal of this section is to introduce the concepts of language and sentence.

In conventional texts about mathematics it is not uncommon to encounter a definition of sequences that looks as follows:

Definition 1: Sequence

sequence

Let X be a set. The set of sequences over X, called X^*, is defined as follows:

- ϵ is a sequence, called the empty sequence, and
- if `xs` is a sequence and `x` is an element of X, then `x:xs` is also a sequence, and
- nothing else is a sequence over X.

□

There are two important things to note about this definition of the set X^*.

1. It is an instance of a very common definition pattern: it is *defined by induction*, i.e. the definition of the concept refers to the concept itself.

induction

2. It corresponds almost exactly to the definition of the type `[x]` of *lists* of elements of a type `x` in `Haskell`; except for the final clause — nothing else is a sequence of X-elements (in Haskell you can define infinite lists, sequences are always finite). Since this pattern of definition is so common when defining a recursive datatype, the last part of the definition is always implicitly understood: if it cannot be generated it does not exist.

We will use Haskell datatypes and functions for the implementation of X^* in the sequel.

Functions on X^* such as reverse, length and concatenation (`++` in Haskell) are inductively defined and these definitions often follow a recursion pattern which is similar to the definition of X^* itself. (Recall the `foldr`'s that you have used in the course on Functional Programming.)

One final remark on sequences is on notation: In Haskell one writes

`[x1,x2, ... ,xn]`	for	`x1 : x2 : ... xn : ` ϵ
`"abba"`	for	`['a','b','b','a']`

When discussing languages and grammars traditionally one uses

`abba`	for	`a : b : b : a : []`
`xy`	for	`x ++ y`

So letters from the beginning of the alphabet represent single symbols, and letters from the end of the alphabet represent sequences of symbols.

Note that the distinction between single elements (like `a`) and sequences (like `aa`) is not explicit in this notation; it is traditional however to let characters from the beginning of the alphabet stand for single symbols (`a, b, c,..`) and symbols from the end of the alphabet for sequences of symbols (`x, y, z`). As a consequence `ax` should be interpreted as a sequence which starts with a single symbol called `a` and has a tail called `x`.

Now we move from individual sequences to finite or infinite sets of sequences. We start with some terminology:

Definition 2: Alphabet, Language, Sentence

- An *alphabet* is a finite set of *symbols*. — alphabet
- A *language* is a subset of T^*, for some alphabet T. — language
- A *sentence* is an element of a language. — sentence

□

Some examples of alphabets are:

- the conventional Roman alphabet: $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{z}\}$;
- the binary alphabet $\{\mathtt{0}, \mathtt{1}\}$;
- sets of reserved words $\{\mathtt{if}, \mathtt{then}, \mathtt{else}\}$;
- the set of characters $l = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{i}, \mathtt{k}, \mathtt{l}, \mathtt{m}, \mathtt{n}, \mathtt{o}, \mathtt{p}, \mathtt{r}, \mathtt{s}, \mathtt{t}, \mathtt{u}, \mathtt{w}, \mathtt{x}\}$;
- the set of English words $\{\mathtt{course}, \mathtt{practical}, \mathtt{exercise}, \mathtt{exam}\}$.

Examples of languages are:

- T^*, $\emptyset$ (the empty set), $\{\epsilon\}$ and T are languages over alphabet T;
- the set $\{\mathtt{course}, \mathtt{practical}, \mathtt{exercise}, \mathtt{exam}\}$ is a language over the alphabet l of characters and `exam` is a sentence in it.

The question that now arises is how to *specify* a language. Since a language is a set we immediately see three different approaches:

- enumerate all the elements of the set explicitly;

- characterise the elements of the set by means of a predicate;
- define which elements belong to the set by means of induction.

We have just seen some examples of the first (the Roman alphabet) and third (the set of sequences over an alphabet) approach. Examples of the second approach are:

- the even natural numbers $\{ n \mid n \in \{0,1,..,9\}^*, n \,\mathtt{mod}\, 2 = 0\}$;
- PAL, the palindromes, sequences which read the same forward as backward, over the alphabet $\{\mathtt{a},\mathtt{b},\mathtt{c}\}$: $\{ s \mid s \in \{\mathtt{a},\mathtt{b},\mathtt{c}\}^*, s = s^R \}$, where s^R denotes the reverse of sequence s.

One of the fundamental differences between the predicative and the inductive approach to defining a language is that the latter approach is *constructive*, i.e., it provides us with a way to enumerate all elements of a language. If we define a language by means of a predicate we only have a means to decide whether or not an element belongs to a language. A famous example of a language which is easily defined in a predicative way, but for which the membership test is very hard, is the set of prime numbers.

Quite often we want to prove that a language L, which is defined by means of an inductive definition, has a specific property P. If this property is of the form $P(L) = \forall x \in L : P(x)$, then we want to prove that $L \subseteq P$.

Since languages are sets the usual set operators such as union, intersection and difference can be used to construct new languages from existing ones. The complement of a language L over alphabet T is defined by $\overline{L} = \{x | x \in T^*, x \notin L\}$.

In addition to these set operators, there are more specific operators, which apply only to sets of sequences. We will use these operators mainly in the chapter on regular languages, Chapter 5. Note that $\cup$ denotes set union, so $\{1,2\} \cup \{1,3\} = \{1,2,3\}$.

Definition 3: Language operations

Let L and M be languages over the same alphabet T, then

$\overline{L}$	$=$	$T^* - L$	complement of L
L^R	$=$	$\{ s^R \mid s \in L \}$	reverse of L
LM	$=$	$\{ st \mid s \in L,\, t \in M \}$	concatenation of L and M
L^0	$=$	$\{\epsilon\}$	$0^{th}-$power of L
L^n	$=$	$LL \ldots L$ (n times)	$n^{th}-$power of L
L^*	$=$	$L^0 \cup L^1 \cup L^2 \cup \ldots$	star $-$ closure of L
L^+	$=$	$L^1 \cup L^2 \cup \ldots$	positive closure of L

□

The following equations follow immediately from the above definitions.

$$\begin{aligned} L^* &= \{\epsilon\} \cup LL^* \\ L^+ &= LL^* \end{aligned}$$

Exercise 2.1 ▷ Let $L = \{\mathtt{ab}, \mathtt{aa}, \mathtt{baa}\}$, where $\mathtt{a}$, and $\mathtt{b}$ are the terminals. Which of the

following strings are in L^*:
abaabaaabaa, aaaabaaaa, baaaaabaaaab, baaaaabaa? ◁

Exercise 2.2 ▷ What are the elements of $\emptyset^*$? ◁

Exercise 2.3 ▷ For any language, prove

1. $\emptyset\, L \;=\; L\, \emptyset \;=\; \emptyset$
2. $\{\epsilon\}\, L \;=\; L\, \{\epsilon\} \;=\; L$

◁

Exercise 2.4 ▷ Can you motivate our choice for $L^0 \;=\; \{\epsilon\}$?
Hint: Establish an inductive definition for the powers of a language. ◁

Exercise 2.5 ▷ In this section we defined two "star" operators: one for arbitrary sets (Definition 1) and one for languages (Definition 3). Is there a difference between these operators? ◁

2.2 Grammars

The goal of this section is to introduce the concept of context-free grammars.

Working with sets might be fun, but it is complicated to manipulate sets, and to prove properties of sets. For these purposes we introduce syntactical definitions, called grammars, of sets. This section will only discuss so-called *context-free grammars*, a kind of grammars that are convenient for automatical processing, and that can describe a large class of languages. But the class of languages that can be described by context-free grammars is limited.

In the previous section we defined PAL, the language of palindromes, by means of a predicate. Although this definition defines the language we want, it is hard to use in proofs and programs. An important observation is the fact that the set of palindromes can be defined inductively as follows.

Definition 4: Palindromes by induction

- The empty string, ϵ, is a palindrome;
- the strings consisting of just one character, `a`, `b`, and `c`, are palindromes;
- if P is a palindrome, then the strings obtained by prepending and appending the same character, `a`, `b`, and `c`, to it are also palindromes, that is, the strings

  `a`P`a`

  `b`P`b`

  `c`P`c`

  are palindromes.

□

The first two parts of the definition cover the basic cases. The last part of the definition covers the inductive cases. All strings which belong to the language PAL inductively defined using the above definition read the same forwards and backwards. Therefore this definition is said to be *sound* (every string in PAL is a palindrome). Conversely, if a string consisting of `a`'s, `b`'s, and `c`'s reads the same forwards and backwards then it belongs to the language PAL. Therefore this definition is said to be *complete* (every palindrome is in PAL).

sound

complete

Finding an inductive definition for a language which is described by a predicate (like the one for palindromes) is often a nontrivial task. Very often it is relatively easy to find a definition that is sound, but you also have to convince yourself that the definition is complete. A typical method for proving soundness and completeness of an inductive definition is *mathematical induction*.

Now that we have an inductive definition for palindromes, we can proceed by giving a formal representation of this inductive definition.

Inductive definitions like the one above can be represented formally by making use of *deduction rules* which look like:

$$a_1, a_2, \cdots, a_n \vdash a \quad \text{or} \quad \vdash a$$

The first kind of deduction rule has to be read as follows:

> if a_1, a_2, ... and a_n are true,
> then a is true

The second kind of deduction rule, called an axiom, has to be read as follows:

> a is true

Using these deduction rules we can now write down the inductive definition for PAL as follows:

$$\begin{aligned}
&\vdash && \epsilon \in \text{PAL'} \\
&\vdash && \texttt{a} \in \text{PAL'} \\
&\vdash && \texttt{b} \in \text{PAL'} \\
&\vdash && \texttt{c} \in \text{PAL'} \\
P \in \text{PAL'} &\vdash && \texttt{a}P\texttt{a} \in \text{PAL'} \\
P \in \text{PAL'} &\vdash && \texttt{b}P\texttt{b} \in \text{PAL'} \\
P \in \text{PAL'} &\vdash && \texttt{c}P\texttt{c} \in \text{PAL'}
\end{aligned}$$

Although the definition PAL' is completely formal, it is still laborious to write. Since in computer science we use many definitions which follow this pattern, we introduce a shorthand for it, called a *grammar*. A grammar consists of production rules for constructing palindromes. The rule with which the empty string is constructed is:

grammar

$$P \quad \rightarrow \quad \epsilon$$

This rule corresponds to the axiom that states that the empty string ϵ is a palindrome. A rule of the form $s \to \alpha$, where s is symbol and α is a sequence of symbols, is called a *production rule*, or *production* for short. A production rule can be considered as a possible way to rewrite the symbol s. The symbol P to the left of the arrow is a symbol which denotes palindromes. Such a symbol is an example of a *nonterminal symbol*, or *nonterminal* for short. Nonterminal symbols are also called auxiliary symbols: their only purpose is to denote structure, they are not part of the alphabet of the language. Three other basic production rules are the rules for constructing palindromes consisting of just one character. Each of the one element strings `a`, `b`, and `c` is a palindrome, and gives rise to a production:

production rule

nonterminal

$$
\begin{aligned}
P &\to \texttt{a} \\
P &\to \texttt{b} \\
P &\to \texttt{c}
\end{aligned}
$$

These production rules correspond to the axioms that state that the one element strings `a`, `b`, and `c` are palindromes. If a string α is a palindrome, then we obtain a new palindrome by prepending and appending an `a`, `b`, or `c` to it, that is, `a`α`a`, `b`α`b`, and `c`α`c` are also palindromes. To obtain these palindromes we use the following recursive productions:

$$
\begin{aligned}
P &\to \texttt{a}P\texttt{a} \\
P &\to \texttt{b}P\texttt{b} \\
P &\to \texttt{c}P\texttt{c}
\end{aligned}
$$

These production rules correspond to the deduction rules that state that, if P is a palindrome, then one can deduce that `a`P`a`, `b`P`b` and `c`P`c` are also palindromes. The grammar we have presented so far consists of three components:

- the set of *terminals* $\{\texttt{a}, \texttt{b}, \texttt{c}\}$;
- the set of nonterminals $\{P\}$;
- and the set of productions (the seven productions that we have introduced so far).

terminal

Note that the intersection of the set of terminals and the set of nonterminals is empty. We complete the description of the grammar by adding a fourth component: the nonterminal *start-symbol* P. In this case we have only one choice for a start symbol, but a grammar may have many nonterminal symbols.

start-symbol

This leads to the following grammar for PAL

$$
\begin{aligned}
P &\to \epsilon \\
P &\to \texttt{a} \\
P &\to \texttt{b} \\
P &\to \texttt{c} \\
P &\to \texttt{a}P\texttt{a}
\end{aligned}
$$

$$P \rightarrow \mathtt{b}P\mathtt{b}$$
$$P \rightarrow \mathtt{c}P\mathtt{c}$$

The definition of the set of terminals, $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, and the set of nonterminals, $\{P\}$, is often implicit. Also the start-symbol is implicitly defined since there is only one nonterminal.

We conclude this example with the formal definition of a context-free grammar.

Definition 5: Context-Free Grammar

context-free grammar

A context-free grammar G is a four tuple (T, N, R, S) where

- T is a finite set of terminal symbols;
- N is a finite set of nonterminal symbols (T and N are disjunct);
- R is a finite set of production rules. Each production has the form $A \rightarrow \alpha$, where A is a nonterminal and α is a sequence of terminals and nonterminals;
- S is the start-symbol, $S \in N$.

□

The adjective "context-free" in the above definition comes from the specific production rules that are considered: exactly one nonterminal in the left hand side. Not every language can be described via a context-free grammar. The standard example here is $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \in \mathbb{N}\}$. We will encounter this example again later in these lecture notes.

2.2.1 Notational conventions

In the definition of the grammar for PAL we have written every production on a single line. Since this takes up a lot of space, and since the production rules form the heart of every grammar, we introduce the following shorthand. Instead of writing

$$S \rightarrow \alpha$$
$$S \rightarrow \beta$$

we combine the two productions for S in one line as using the symbol |.

$$S \rightarrow \alpha \mid \beta$$

We may rewrite any number of rewrite rules for one nonterminal in this fashion, so the grammar for PAL may also be written as follows:

$$P \rightarrow \epsilon \mid \mathtt{a} \mid \mathtt{b} \mid \mathtt{c} \mid \mathtt{a}P\mathtt{a} \mid \mathtt{b}P\mathtt{b} \mid \mathtt{c}P\mathtt{c}$$

BNF

The notation we use for grammars is known as *BNF* - Backus Naur Form -, after Backus and Naur, who first used this notation for defining grammars.

Another notational convention concerns names of productions. Sometimes we want to give names to production rules. The names will be written in front of the production. So, for example,

$$\begin{array}{lrcl} \textit{Alpha rule}: & S & \rightarrow & \alpha \\ \textit{Beta rule}: & S & \rightarrow & \beta \end{array}$$

Finally, if we give a context-free grammar just by means of its productions, the start-symbol is usually the nonterminal in the left hand side of the first production, and the start-symbol is usually called S.

Exercise 2.6 ▷ Give a context free grammar for the set of sentences over alphabet T where

1. $T = \{\mathsf{a}\}$
2. $T = \{\mathsf{a}, \mathsf{b}\}$

◁

Exercise 2.7 ▷ Give a context free grammar for the language

$$L \;=\; \{\, a^n\, b^n \;\mid\; n \in \mathbb{N} \,\}$$

◁

Exercise 2.8 ▷ Give a grammar for *palindromes* over the alphabet $\{\mathsf{a}, \mathsf{b}\}$ ◁

Exercise 2.9 ▷ Give a grammar for the language

$$L \;=\; \{\, s\, s^R \;\mid\; s \in \{\mathsf{a}, \mathsf{b}\}^* \,\}$$

This language is known as the *mirror-palindromes* language. ◁

Exercise 2.10 ▷ A *parity-sequence* is a sequence consisting of 0's and 1's that has an even number of ones. Give a grammar for *parity-sequences*. ◁

Exercise 2.11 ▷ Give a grammar for the language

$$L = \{w \mid w \in \{\mathsf{a}, \mathsf{b}\}^* \wedge nr(\mathsf{a}, w) = nr(\mathsf{b}, w)\}$$

where $nr(c, w)$ is the number of c-occurrences in w. ◁

2.3 The language of a grammar

The goal of this section is to describe the relation between grammars and languages: to show how to derive sentences of a language given its grammar.

We have seen that a grammar for a given language may be given. Now we consider the reverse question: how to obtain a language from a given grammar? Before we can answer this question we first have to say what we can do with a grammar. The answer is simple: we can derive sequences with it.

How do we construct a palindrome? A palindrome is a sequence of terminals, in our case the characters `a`, `b` and `c`, that can be derived in zero or more direct derivation steps from the start-symbol P using the productions of the grammar for palindromes given above.

For example, the sequence `bacab` can be derived using the grammar for palindromes as follows:

$$\begin{array}{ll} & P \\ \Rightarrow & \\ & \texttt{b}P\texttt{b} \\ \Rightarrow & \\ & \texttt{ba}P\texttt{ab} \\ \Rightarrow & \\ & \texttt{bacab} \end{array}$$

derivation

Such a construction is called a *derivation*. In the first step of this derivation production $P \to \texttt{b}P\texttt{b}$ is used to rewrite P into $\texttt{b}P\texttt{b}$. In the second step production $P \to \texttt{a}P\texttt{a}$ is used to rewrite $\texttt{b}P\texttt{b}$ into $\texttt{ba}P\texttt{ab}$. Finally, in the last step production $P \to \texttt{c}$ is used to rewrite $\texttt{ba}P\texttt{ab}$ into `bacab`. Constructing a derivation can be seen as a constructive proof that the string `bacab` is a palindrome.

We will now describe derivation steps more formally.

Definition 6: Derivation

Suppose $X \to \beta$ is a production of a grammar, where X is a nonterminal symbol and β is a sequence of (nonterminal or terminal) symbols. Let $\alpha X \gamma$ be a sequence of (nonterminal or terminal) symbols. We say that $\alpha X \gamma$ *directly derives* the sequence $\alpha\beta\gamma$, which is obtained by replacing the left hand side X of the production by the corresponding right hand side β. We write $\alpha X \gamma \Rightarrow \alpha\beta\gamma$ and we also say that $\alpha X \gamma$ rewrites to $\alpha\beta\gamma$ in one step. A sequence φ_n is *derived* from a sequence φ_1, written $\varphi_1 \stackrel{*}{\Rightarrow} \varphi_n$, if there exist sequences $\varphi_1, \ldots, \varphi_n$ with $n \geq 1$ such that

direct derivation

derivation

$$\forall i,\, 1 \leq i < n \,:\, \varphi_i \Rightarrow \varphi_{i+1}$$

If $n = 1$ this statement is trivially true, and it follows that we can derive each sentence from itself in zero steps:

$$\varphi \stackrel{*}{\Rightarrow} \varphi$$

partial derivation

A *partial derivation* is a derivation of a sequence that still contains nonterminals. □

Finding a derivation $\varphi_1 \stackrel{*}{\Rightarrow} \varphi_n$ is, in general, a nontrivial task. A derivation is only one branch of a whole search tree which contains many more branches. Each branch represents a (successful or unsuccessful) direction in which a possible derivation may proceed. Another important challenge is to arrange things in such a way that finding a derivation can be done in an efficient way.

From the example derivation above it follows that

$$P \stackrel{*}{\Rightarrow} \texttt{bacab}$$

Because this derivation begins with the start-symbol of the grammar and results in a sequence consisting of terminals only (a terminal string), we say that the string `bacab` belongs to the language generated by the grammar for palindromes. In general, we define

Definition 7: Language of a grammar (or language generated by a grammar)
The language of a grammar $G = (T, N, R, S)$, usually denoted by $L(G)$, is defined as

$$L(G) = \{ s \mid S \stackrel{*}{\Rightarrow} s\,,\ s \in T^* \}$$

□

We sometimes also talk about the language of a nonterminal, which is defined by

$$L(A) = \{ s \mid A \stackrel{*}{\Rightarrow} s\,,\ s \in T^* \}$$

for nonterminal A. The language of a grammar could have been defined as the language of its start-symbol.

Note that different grammars may have the same language. For example, if we extend the grammar for PAL with the production $P \rightarrow$ `bacab`, we obtain a grammar with exactly the same language as PAL. Two grammars that generate the same language are called *equivalent*. So for a particular grammar there exists a unique language, but the reverse is not true: given a language we can usually construct many grammars that generate the language. In mathematical language: the mapping between a grammar and its language is not a bijection.

Definition 8: Context-free language
A context-free language is a language that is generated by a context-free grammar. □

context-free language

All palindromes can be derived from the start-symbol P. Thus, the language of our grammar for palindromes is PAL, the set of all palindromes over the alphabet {`a`, `b`, `c`} and PAL is context-free.

2.3.1 Some basic languages

Digits occur in a lot of programming and other languages, and so do letters. In this subsection we will define some grammars that specify some basic languages such as digits and letters. These grammars will be used often in later sections.

- The language of single digits is specified by a grammar with 10 production rules for the nonterminal *Dig*.

  $$Dig \rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}$$

- We obtain sequences of digits by means of the following grammar:

  $$Digs \rightarrow \epsilon \mid Dig\ Digs$$

- Natural numbers are sequences of digits that start with a nonzero digit. So in order to specify natural numbers, we first define the language of nonzero digits.

  $$\textit{Dig-0} \;\rightarrow\; \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}$$

  Now we can define natural numbers as follows.

  $$\textit{Nat} \;\rightarrow\; \texttt{0} \mid \textit{Dig-0 Digs}$$

- Integers are natural numbers preceded by a sign. If a natural number is not preceded by a sign, it is supposed to be a positive number.

  $$\begin{aligned} \textit{Sign} &\rightarrow \texttt{+} \mid \texttt{-} \\ \textit{Z} &\rightarrow \textit{Sign Nat} \mid \textit{Nat} \end{aligned}$$

- The languages of underscore letters and capital letters are each specified by a grammar with 26 productions:

  $$\begin{aligned} \textit{ULetter} &\rightarrow \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \\ \textit{CLetter} &\rightarrow \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \end{aligned}$$

  In the real definitions of these grammars we have to write each of the 26 letters, of course. A letter is now either an underscore or a capital letter.

  $$\textit{Letter} \;\rightarrow\; \textit{ULetter} \mid \textit{CLetter}$$

- Variable names, function names, datatypes, etc., are all represented by identifiers in programming languages. The following grammar for identifiers might be used in a programming language:

  $$\begin{aligned} \textit{Identifier} &\rightarrow \textit{Letter SoS} \\ \textit{SoS} &\rightarrow \epsilon \mid \textit{Letter SoS} \mid \textit{Dig SoS} \end{aligned}$$

  An identifier starts with a letter, and is followed by a sequence of letters and digits (sequence of symbols). We might want to allow more symbols, such as for example underscores and dollars, but then we have to adjust the grammar, of course.

- Dutch zipcodes consist of four digits, of which the first digit is nonzero, followed by two capitals. So

  $$\textit{ZipCode} \;\rightarrow\; \textit{Dig-0 Dig Dig Dig CLetter CLetter}$$

Exercise 2.12 ▷ A terminal string is always derived in one or more steps from the start-symbol. Why? ◁

Exercise 2.13 ▷ What language is generated by the grammar with the single production rule

$$S \rightarrow \epsilon$$

◁

Exercise 2.14 ▷ What language does the grammar with the following productions generate?

$$\begin{aligned} S &\rightarrow A\mathsf{a} \\ A &\rightarrow B \\ B &\rightarrow A\mathsf{a} \end{aligned}$$

◁

Exercise 2.15 ▷ Give a simple description of the language generated by the grammar with productions

$$\begin{aligned} S &\rightarrow \mathsf{a}A \\ A &\rightarrow \mathsf{b}S \\ S &\rightarrow \epsilon \end{aligned}$$

◁

Exercise 2.16 ▷ Is the language L defined in exercise 2.1 context free ? ◁

2.4 Parse trees

The goal of this section is to introduce parse trees, and to show how parse trees relate to derivations. Furthermore, this section defines (non)ambiguous grammars.

For any partial derivation, i.e. a derivation that contains nonterminals in its right hand side, there may be several productions of the grammar that can be used to proceed the partial derivation with and, as a consequence, there may be different derivations for the same sentence. There are two reasons why derivations for a specific sentence differ:

- Only the order in which the derivation steps are chosen differs. All such derivations are considered to be equivalent.
- Different derivation steps have been chosen. Such derivations are considered to be different.

Here is a simple example. Consider the grammar *SequenceOfS* with productions:

$$\begin{aligned} S &\rightarrow SS \\ S &\rightarrow \mathsf{s} \end{aligned}$$

Using this grammar we can derive the sentence `sss` as follows (we have underlined the nonterminal that is rewritten).

$$\underline{S} \Rightarrow S\underline{S} \Rightarrow S\underline{S}S \Rightarrow \underline{S}\mathtt{s}S \Rightarrow \mathtt{ss}\underline{S} \Rightarrow \mathtt{sss}$$

$$\underline{S} \Rightarrow \underline{S}S \Rightarrow \mathtt{s}\underline{S} \Rightarrow \mathtt{s}S\underline{S} \Rightarrow \mathtt{s}\underline{S}\mathtt{s} \Rightarrow \mathtt{sss}$$

These derivations are the same up to the order in which derivation steps are taken. However, the following derivation does not use the same derivation steps:

$$\underline{S} \Rightarrow \underline{S}S \Rightarrow \underline{S}SS \Rightarrow \mathtt{s}\underline{S}S \Rightarrow \mathtt{ss}\underline{S} \Rightarrow \mathtt{sss}$$

In this derivation, the first S is rewritten to SS instead of `s`.

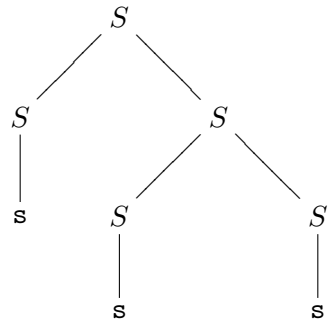
The set of all equivalent derivations can be represented by selecting a, so called, *canonical* element. A good candidate for such a canonical element is the leftmost derivation. In a leftmost derivation, only the leftmost nonterminal is rewritten. If there exists a derivation of a sentence x using the productions of a grammar, then there exists a leftmost derivation of x. The leftmost derivation corresponding to the two equivalent derivations above is

leftmost derivation

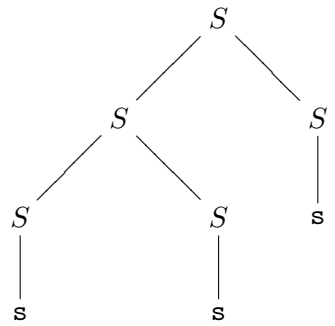
$$\underline{S} \Rightarrow \underline{S}S \Rightarrow \mathtt{s}\underline{S} \Rightarrow \mathtt{s}\underline{S}S \Rightarrow \mathtt{ss}\underline{S} \Rightarrow \mathtt{sss}$$

There exists another convenient way for representing equivalent derivations: they all have the same *parse tree* (or derivation tree). A parse tree is a representation of a derivation which abstracts from the order in which derivation steps are chosen. The internal nodes of a parse tree are labelled with a nonterminal N, and the children of such a node are the parse trees for symbols of the right hand side of a production for N. The parse tree of a terminal symbol is a leaf labelled with the terminal symbol. The resulting parse tree of the first two derivations of the sentence `sss` looks as follows.

parse tree



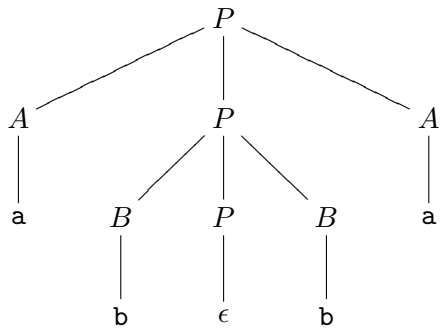
The third derivation of the sentence `sss` results in a different parse tree:



As another example, all derivations of the string `abba` using the productions of the grammar

$$
\begin{aligned}
P &\rightarrow \epsilon \\
P &\rightarrow APA \\
P &\rightarrow BPB \\
A &\rightarrow \texttt{a} \\
B &\rightarrow \texttt{b}
\end{aligned}
$$

are represented by the following derivation tree:



A derivation tree can be seen as a *structural interpretation* of the derived sentence. Note that there might be more than one structural interpretation of a sentence with respect to a given grammar. Such grammars are called ambiguous.

Definition 9: ambiguous, unambiguous grammar

A grammar is *unambiguous* if every sentence has a unique leftmost derivation, or, equivalently, if every sentence has a unique derivation tree. Otherwise it is called *ambiguous*. □

unambiguous

ambiguous

The grammar *SequenceOfS* for constructing sequences of `s`'s is an example of an ambiguous grammar, since there exist two parse trees for the string `sss`.

It is in general undecidable whether or not an arbitrary context-free grammar is ambiguous. This implies that is impossible to write a program that determines the (non)ambiguity of a context-free grammar.

It is usually rather difficult to translate languages with ambiguous grammars. Therefore, you will find that most grammars of programming languages and other languages that are used in processing information are unambiguous.

Grammars have proved very successful in the specification of artificial languages (such as programming languages). They have proved less successful in the specification of natural languages (such as English), partly because is extremely difficult to construct an unambiguous grammar that specifies a nontrivial part of the language. Take for example the sentence 'They are flying planes'. This sentence can be read in two ways, with different meanings: 'They - are - flying planes', and 'They - are flying - planes'. Ambiguity of natural languages

may perhaps be considered as an advantage for their users (e.g. politicians), it certainly is considered a disadvantage for language translators because it is usually impossible to maintain an ambiguous meaning in a translation.

2.4.1 From context-free grammars to datatypes

For each context-free grammar we can define a corresponding datatype in Haskell. Values of these datatypes represent parse trees of the context-free grammar. As an example we take the grammar used in the beginning of this section:

$$\begin{array}{lcl} S & \rightarrow & SS \\ S & \rightarrow & \texttt{s} \end{array}$$

First, we give each of the productions of this grammar a name:

$$\begin{array}{lcl} \mathit{Beside}: \; S & \rightarrow & SS \\ \mathit{Single}: \; S & \rightarrow & \texttt{s} \end{array}$$

And now we interpret the start-symbol of the grammar S as a datatype, using the names of the productions as constructors:

```
data S = Beside S S
       | Single Char
```

Note that this datatype is too general: the type `Char` should really be the single character `s`. This datatype can be used to represent parse trees of sentences of S. For example, the parse tree that corresponds to the first two derivations of the sequence `sss` is represented by the following value of the datatype `S`.

```
Beside (Single 's') (Beside (Single 's') (Single 's'))
```

The third derivation of the sentence `sss` produces the following parse tree:

```
Beside (Beside (Single 's') (Single 's')) (Single 's')
```

Because the datatype `S` is too general, we will reconsider the construction of datatypes from context-free grammars in Section 2.6.

Exercise 2.17 ▷ Consider the grammar for *palindromes* that you have constructed in exercise 2.8. Give parse trees for the palindromes `cPal1 = "abaaba"` and `cPal2 = "baaab"`. Define a datatype `Palc` corresponding to the grammar and represent the parse trees for `cPal1` and `cPal2` as values of `Palc`. ◁

2.5 Grammar transformations

The goal of this section is to discuss properties of grammars, grammar transformations, and to show how grammar transformations can be used to obtain grammars that satisfy particular properties.

Grammars satisfy properties. Examples of properties are:

- a grammar may be unambiguous, that is, every sentence of its language has a unique parse tree;
- a grammar may have the property that only the start-symbol can derive the empty string; no other nonterminal can derive the empty string;
- a grammar may have the property that every production either has a single terminal, or two nonterminals in its right-hand side. Such a grammar is said to be in Chomsky normal form.

So why are we interested in such properties? Some of these properties imply that it is possible to build parse trees for sentences of the language of the grammar in only one way. Some other properties imply that we can build these parse trees very fast. Other properties are used to prove facts about grammars. Yet other properties are used to efficiently compute some other information from parse trees of a grammar.

For example, suppose we have a program p that builds parse trees for sentences of grammars in Chomsky normal form, and that we can prove that each grammar can be transformed in a grammar in Chomsky normal form. (When we say that a grammar G can be transformed in another grammar G', we mean that there exists some procedure to obtain G' from G, and that G and G' generate the same language.) Then we can use this program p for building parse trees for any grammar.

grammar transformation

Since it is sometimes convenient to have a grammar that satisfies a particular property for a language, we would like to be able to *transform* grammars in other grammars that generate the same language, but that possibly satisfy different properties. This section describes a number of grammar transformations

- Removing duplicate productions.
- Substituting right-hand sides for nonterminals.
- Left factoring.
- Removing left recursion.
- Associative separator.
- Introduction of priorities.

There are many more transformations than we describe here; we will only show a small but useful set of grammar transformations. In the following transformations we will assume that u, v, w, x, y, and z denote sequences of terminals and nonterminals, i.e., are elements of $(N \cup T)^*$.

Removing duplicate productions

This grammar transformation is a transformation that can be applied to any grammar of the correct form. If a grammar contains two occurrences of the same production rule, one of these occurrences can be removed. For example,

$$A \rightarrow u \mid u \mid v$$

can be transformed into

$$A \rightarrow u \mid v$$

Substituting right-hand sides for nonterminals
If a nonterminal N occurs in a right-hand side of a production, the production may be replaced by just as many productions as there exist productions for N, in which N has been replaced by its right-hand sides. For example,

$$\begin{aligned} A &\rightarrow uBv \mid z \\ B &\rightarrow x \mid w \end{aligned}$$

may be transformed into

$$\begin{aligned} A &\rightarrow uxv \mid uwv \mid z \\ B &\rightarrow x \mid w \end{aligned}$$

Left factoring

left factoring

Left factoring a grammar is a grammar transformation that is useful when two productions for the same nonterminal start with the same sequence of (terminal and/or nonterminal) symbols. These two productions can then be replaced by a single production, that ends with a new nonterminal, replacing the part of the sequence after the common start sequence. Two productions for the new nonterminal are added: one for each of the two different end sequences of the two productions. For example:

$$A \rightarrow xy \mid xz \mid v$$

where $x \in (N \cup T)^*$, and $x \neq \epsilon$, may be transformed into

$$\begin{aligned} A &\rightarrow xZ \mid v \\ Z &\rightarrow y \mid z \end{aligned}$$

where Z is a new nonterminal.

Removing left recursion

left recursion

A left recursive production is a production in which the right-hand side starts with the nonterminal of the left-hand side. For example, the production

$$A \rightarrow Az$$

is left recursive. A grammar is left recursive if we can derive $A \stackrel{*}{\Rightarrow} Az$ for some nonterminal A of the grammar. Left recursive grammars are sometimes undesirable. The following transformation removes left recursive productions.

To remove the left recursive productions of a nonterminal A, divide the productions for A in sets of left recursive and non left recursive productions. Factorise A as follows:

$$\begin{aligned} A &\rightarrow Ax_1 \mid Ax_2 \mid \ldots \mid Ax_n \\ A &\rightarrow y_1 \mid y_2 \mid \ldots \mid y_m \end{aligned}$$

with x_i, $y_j \in (N \cup T)^*$, *head* $y_j \neq A$ (where *head* returns the first element of a list of elements), and $1 \leq i \leq n$, $1 \leq j \leq m$. Add a new nonterminal Z, and replace A's productions by:

$$\begin{aligned} A &\rightarrow y_1 \mid y_1 Z \mid \ldots \mid y_m \mid y_m Z \\ Z &\rightarrow x_1 \mid x_1 Z \mid \ldots \mid x_n \mid x_n Z \end{aligned}$$

This procedure only works for a grammar that is *direct* left recursive, i.e., a grammar that contains a production of the form $A \rightarrow Ax$. Removing left recursion in general left recursive grammars, which for example contain productions like $A \rightarrow Bx$, $B \rightarrow Ay$ is a bit more complicated, see [1].

For example, the grammar *SequenceOfS*, see Section 2.4, with the productions

$$\begin{aligned} S &\rightarrow SS \\ S &\rightarrow \texttt{s} \end{aligned}$$

is a left recursive grammar. The above procedure for removing left recursion gives the following productions:

$$\begin{aligned} S &\rightarrow \texttt{s} \mid \texttt{s}Z \\ Z &\rightarrow S \mid SZ \end{aligned}$$

Associative separator

The following grammar generates a list of declarations, separated by a semicolon `;`.

$$\begin{aligned} \mathit{Decls} &\rightarrow \mathit{Decls}\ \texttt{;}\ \mathit{Decls} \\ \mathit{Decls} &\rightarrow \mathit{Decl} \end{aligned}$$

where the productions for *Decl*, which generates a single declaration, are omitted. This grammar is ambiguous, for the same reason as *SequenceOfS* is ambiguous. The operator `;` is an associative separator in the generated language, that is: `d1 ; (d2 ; d3)` = `(d1 ; d2) ; d3` where `d1`, `d2`, and `d3` are declarations. Therefore, we may use the following unambiguous grammar for generating a language of declarations:

$$\begin{aligned} \mathit{Decls} &\rightarrow \mathit{Decl}\ \texttt{;}\ \mathit{Decls} \\ \mathit{Decls} &\rightarrow \mathit{Decl} \end{aligned}$$

An alternative grammar for the same language is

$$\begin{aligned} \mathit{Decls} &\rightarrow \mathit{Decls}\ \texttt{;}\ \mathit{Decl} \\ \mathit{Decls} &\rightarrow \mathit{Decl} \end{aligned}$$

This grammar transformation can only be applied because the semicolon is associative in the generated language; it is not a grammar transformation that can be applied blindly to any grammar.

The same transformation can be applied to grammars with productions of the form:

$$A \;\rightarrow\; A\texttt{a}A$$

where `a` is an associative operator in the generated language. As an example you may think of natural numbers with addition.

Introduction of priorities

Another form of ambiguity often arises in the part of a grammar for a programming language which describes expressions. For example, the following grammar generates arithmetic expressions:

$$\begin{aligned} E &\rightarrow E \texttt{ + } E \\ E &\rightarrow E \texttt{ * } E \\ E &\rightarrow \texttt{(}E\texttt{)} \\ E &\rightarrow \mathit{Digs} \end{aligned}$$

where *Digs* generates a list of digits, see Section 2.3.1. This grammar is ambiguous: the sentence `2+4*6` has two parse trees: one corresponding to `(2+4)*6`, and one corresponding to `2+(4*6)`. If we make the usual assumption that `*` has higher priority than `+`, the latter expression is the intended reading of the sentence `2+4*6`. In order to obtain parse trees that respect these priorities, we transform the grammar as follows:

$$\begin{aligned} E &\rightarrow T \\ E &\rightarrow E \texttt{ + } T \\ T &\rightarrow F \\ T &\rightarrow T \texttt{ * } F \\ F &\rightarrow \texttt{(}E\texttt{)} \\ F &\rightarrow \mathit{Digs} \end{aligned}$$

This grammar generates the same language as the previous grammar for expressions, but it respects the priorities of the operators.

In practice, often more than two levels of priority are used. Then, instead of writing a large number of identically formed production rules, we use parameterised nonterminals. For $1 \leq i < n$,

$$\begin{aligned} E_i &\rightarrow E_{i+1} \\ E_i &\rightarrow E_i \; OP_i \; E_{i+1} \end{aligned}$$

Operator OP_i is a parameterised nonterminal that generates operators of priority i. In addition to the above productions, there should also be a production for expressions of the highest priority, for example:

$$E_n \;\rightarrow\; \texttt{(}E_1\texttt{)} \mid \mathit{Digs}$$

A grammar transformation transforms a grammar into another grammar that generates the same language. For each of the above transformations we should prove that the generated language remains the same. Since the proofs are too complicated at this point, they are omitted. The proofs can be found in any of the theoretical books on language and parsing theory [12].

There exist many more grammar transformations, but the ones given in this section suffice for now. Note that everywhere we use 'left' (left recursion, left factoring), we can replace it by 'right', and obtain a dual grammar transformation. We will discuss a larger example of a grammar transformation after the following section.

Exercise 2.18 ▷ Consider the ambiguous grammar with start symbol A

$$\begin{aligned} A &\rightarrow A\mathsf{a}A \\ A &\rightarrow \mathsf{b} \end{aligned}$$

Transform the grammar by applying the rule for associative separator. ◁

Exercise 2.19 ▷ Give a grammar that describes expressions that respect the priorities for the operators `+`, `-`, `*`, `/` and `^` (power raise). ◁

Exercise 2.20 ▷ The standard example of ambiguity in programming languages is the *dangling else*. Let G be a grammar with terminal set { if, b, then, else , a } and productions

$$\begin{aligned} S &\rightarrow \mathsf{if\ b\ then}\ S\ \mathsf{else}\ S \\ S &\rightarrow \mathsf{if\ b\ then}\ S \\ S &\rightarrow \mathsf{a} \end{aligned}$$

1 Give two derivation trees for the sentence if b then if b then a else a.
2 Give an unambiguous grammar that generates the same language as G.
3 How does Java prevent this *dangling else* problem?

◁

Exercise 2.21 ▷ A *bit-list* is a nonempty list of bits separated by commas. A grammar for `Bit-Lists` is given by

$$\begin{aligned} L &\rightarrow B \\ L &\rightarrow L\ ,\ L \\ B &\rightarrow \mathsf{0} \mid \mathsf{1} \end{aligned}$$

Remove the left recursion from this grammar. ◁

Exercise 2.22 ▷ Consider the grammar with start symbol S

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow \epsilon \mid \mathsf{aa}A \\ B &\rightarrow \epsilon \mid B\mathsf{b} \end{aligned}$$

1 What language does this grammar generate?
2 Give an equivalent non left recursive grammar.

◁

2.6 Concrete and abstract syntax

The goal of this section is to introduce abstract syntax, and to show how to obtain an abstract syntax from a concrete syntax.

Recall the grammar *SequenceOfS* for producing sequences of `s`'s:

Beside : $S \rightarrow SS$
Single : $S \rightarrow$ `s`

As explained in Section 2.4.1, the following datatype can be used to represent parse trees of sentences of the language of S.

```
data S = Beside S S
       | Single Char
```

For example, the sequence `sss` may be represented by the parse tree

```
Beside (Beside (Single 's') (Single 's')) (Single 's')
```

The function `s2string` constructs the sentence that corresponds to a value of the datatype `S`:

```
s2string    :: S -> String
s2string x  =  case x of
                 Beside l r -> s2string l ++ s2string r
                 Single x   -> "s"
```

Since in each parse tree for a sentence of the language of S `Single` will always be followed by the character `'s'`, we do not have to include the type `Char` in the datatype definition `S`. We refine the datatype for representing parse trees of *SequenceOfS* as follows:

```
data SA = BesideA SA SA
        | SingleA
```

Note that the type `Char`, representing the terminal symbol, has disappeared now. The sequence `sss` is represented by the parse tree

```
BesideA (BesideA SingleA SingleA) SingleA
```

concrete syntax

A *concrete syntax* of a language describes the appearance of the sentences of a language. So the concrete syntax of the language of S is given by the grammar

abstract syntax

SequenceOfS. An *abstract syntax* of a language describes the parse trees of a language. Parse trees are therefore sometimes also called abstract syntax trees. The datatype `SA` is an example of an abstract syntax for the language of *SequenceOfS*. The adjective abstract says that values of the abstract syntax do not need to have all information about particular sentences, as long as the information is recoverable. For example, function `sa2string` takes a value of the abstract syntax for *SequenceOfS*, and returns the sentence represented by the abstract syntax tree.

```
sa2string    :: SA -> String
sa2string x  =  case x of
                  BesideA l r  ->  sa2string l ++ sa2string r
                  SingleA      ->  "s"
```

semantic function

Such a function is often called a *semantic function*. A semantic function is a function that is defined on an abstract syntax of a language. Semantic functions are used to give semantics (meaning) to values. Here, the meaning of a more abstract representation is expressed in terms of a concrete representation.

Using the removing left recursion grammar transformation, the grammar *SequenceOfS* can be transformed into the grammar with the following productions:

$$\begin{array}{lcl} S & \rightarrow & \mathtt{s}Z \mid \mathtt{s} \\ Z & \rightarrow & SZ \mid S \end{array}$$

An abstract syntax of this grammar may be given by

```
data SA2  =  ConsS Z | SingleS
data Z    =  ConsSA2 SA2 Z | SingleSA2 SA2
```

In fact, the only important information about sequences of `s`'s is how many occurrences of `s` there are. So the ultimate abstract syntax for *SequenceOfS* is

```
data SA3 = Size Int
```

The sequence `sss` is represented by the parse tree `Size 3`.

The *SequenceOfS* example shows that one may choose between many different abstract syntaxes for a given grammar. The application determines which abstract syntax is most convenient.

Exercise 2.23 ▷ Give an implementation of the function `sa32string` that takes a value of type `SA3` and returns the string represented by this value. ◁

Exercise 2.24 ▷ A datatype in Haskell describes an inductively defined set. The following datatype represents a limited form of arithmetic expressions

```
data Expr = Add Expr Expr | Mul Expr Expr | Con Int
```

Give a grammar that corresponds to this datatype. ◁

Exercise 2.25 ▷ Consider your answer to exercise 2.8, which describes the concrete syntax for *palindromes* over {`a`, `b`}.

1. Define a datatype `Pal` that describes the abstract syntax corresponding to your grammar. Give the two abstract palindromes `aPal1` and `aPal2` that correspond to the concrete palindromes `cPal1 = "abaaba"` and `cPal2 = "baaab"`
2. Write a (semantic) function that transforms an abstract representation of a palindrome into a concrete one. Test your function with the abstract palindromes `aPal1` and `aPal2`.
3. Write a function that counts the number of `a`'s occurring in a palindrome. Test your function with the abstract palindromes `aPal1` and `aPal2`.

◁

Exercise 2.26 ▷ Consider your answer to exercise 2.9, which describes the concrete syntax for *mirror-palindromes*.

1. Define a datatype `Mir` that describes the abstract syntax corresponding to your grammar. Give the two abstract mirror-palindromes `aMir1` and `aMir2` that correspond to the concrete mirror-palindromes `cMir1 = "abaaba"` and `cMir2 = "abbbba"`
2. Write a (semantic) function that transforms an abstract representation of a mirror-palindrome into a concrete one. Test your function with the abstract mirror-palindromes `aMir1` and `aMir2`.
3. Write a function that transforms an abstract representation of a mirror-palindrome into the corresponding abstract representation of a palindrome. Test your function with the abstract mirror-palindromes `aMir1` and `aMir2`.

◁

Exercise 2.27 ▷ Consider your anwer to exercise 2.10, which describes the concrete syntax for *parity-sequences*.

1. Describe the abstract syntax corresponding to your grammar. Give the two abstract parity-sequences `aEven1` and `aEven2` that correspond to the concrete parity-sequences `cEven1 = "00101"` and `cEven2 = "01010"`
2. Write a (semantic) function that transforms an abstract representation of a parity-sequence into a concrete one. Test your function with the abstract parity-sequences `aEven1` and `aEven2`.

◁

Exercise 2.28 ▷ Consider your answer to exercise 2.21, which describes the concrete syntax for *bit-lists* by means of a grammar that is not left recursive.

1. Define a datatype `BitList` that describes the abstract syntax corresponding to your grammar. Give the two abstract bit-lists `aBitList1` and `aBitList2` that correspond to the concrete bit-lists `cBitList1 = "0,1,0"` and `cBitList2 = "0,0,1"`.
2. Write a function that transforms an abstract representation of a bit-list into a concrete one. Test your function with the abstract bit-lists `aBitList1` and `aBitList2`.

3 Write a function that concatenates two abstract representations of a bit-lists into a bit-list. Test your function with the abstract bit-lists `aBitList1` and `aBitList2`.

◁

2.7 Constructions on grammars

This section introduces some constructions on grammars that are useful when specifying larger grammars, for example for programming languages. Furthermore, it gives an example of a larger grammar that is transformed in several steps.

The BNF notation, introduced in section 2.2.1, was first used in the early sixties when the programming language ALGOL 60 was defined and until now it is the way of defining programming languages. See for instance the Java Language Grammar. You may object that the Java grammar contains more "syntactical sugar" than the grammars that we considered thus far (and to be honest, this also holds for the Algol 60 grammar): one encounters nonterminals with postfixes '?', '+' and '*'.

This extended BNF notation, *EBNF*, is introduced because the definition of a programming language requires a lot of nonterminals and adding (superfluous) nonterminals for standard construction such as: EBNF

- one or zero occurrences of nonterminal P, $(P?)$,
- one or more occurrences of nonterminal P, (P^+),
- and zero or more occurrences of nonterminal P, (P^*),

decreases the readability of the grammar. In other texts you will sometimes find $[P]$ instead of $P?$, and $\{P\}$ instead of P^*. The same notation can be used for languages, grammars, and sequences of terminal and nonterminal symbols instead of single nonterminals. This section defines the meaning of these constructs.

We introduced grammars as an alternative for the description of languages. Designing a grammar for a specific language may not be a trivial task. One approach is to decompose the language and to find grammars for each of its constituent parts.

In definition 3 we defined a number of operations on languages using operations on sets. Here we redefine these operations using context-free grammars.

Theorem 10: Language operations

Suppose we have grammars for the languages L and M, say $G_L = (T, N_L, R_L, S_L)$ and $G_M = (T, N_M, R_M, S_M)$. We assume that the sets N_L and N_M are disjoint. Then

- $L \cup M$ is generated by the grammar (T, N, R, S) where S is a fresh nonterminal,

$N = N_L \cup N_M \cup \{S\}$ and $R = R_L \cup R_M \cup \{S \to S_L,\ S \to S_M)$

- $L\,M$ is generated by the grammar (T, N, R, S) where S is a fresh nonterminal, $N = N_L \cup N_M \cup \{S\}$ and $R = R_L \cup R_M \cup \{S \to S_L\,S_M\}$
- L^* is generated by the grammar (T, N, R, S) where S is a fresh nonterminal, $N = N_L \cup \{S\}$ and $R = R_L \cup \{S \to \epsilon,\ S \to S_L\,S\}$.
- L^+ is generated by the grammar (T, N, R, S) where S is a fresh nonterminal, $N = N_L \cup \{S\}$ and $R = R_L \cup \{S \to S_L,\ S \to S_L\,S\}$

□

The nice thing about the above definitions is that the set-theoretic operation at the level of languages (i.e. sets of sentences) has a direct counterpart at the level of grammatical description. A straightforward question to ask is now: can I also define languages as the difference between two languages or as the intersection of two languages? Unfortunately there are no equivalent operators for composing grammars that correspond to such intersection and difference operators.

Two of the above constructions are important enough to also define them as grammar operations. Furthermore, we add a new construction for choice.

Definition 11: Grammar operations

Let $G = (T, N, R, S)$ be a context-free grammar and let S' be a fresh nonterminal. Then

$$\begin{array}{lcll} G^* & = & (T,\ N \cup \{S'\},\ R \cup \{S' \to \epsilon, S' \to SS'\},\ S') & \text{star } G \\ G^+ & = & (T,\ N \cup \{S'\},\ R \cup \{S' \to S, S' \to SS'\},\ S') & \text{plus } G \\ G? & = & (T,\ N \cup \{S'\},\ R \cup \{S' \to \epsilon, S' \to S\},\ S') & \text{optional } G \end{array}$$

□

The definition of $P?$, P^+, and P^* for a string P is very similar to the definitions of the operations on grammars. For example, P^* denotes zero or more concatenations of string P, so Dig^* denotes the language consisting of zero or more digits.

Definition 12: EBNF for sequences

Let P be a sequence of nonterminals and terminals, then

$$\begin{array}{lll} L(P^*) = L(Z) & \text{with} & Z \to \epsilon \mid PZ \\ L(P^+) = L(Z) & \text{with} & Z \to P \mid PZ \\ L(P?) = L(Z) & \text{with} & Z \to \epsilon \mid P \end{array}$$

where Z is a new nonterminal in each definition. □

Because the concatenation operator for sequences is associative, the operators $+$ and $*$ can also be defined symmetrically:

$$\begin{array}{lll} L(P^*) = L(Z) & \text{with} & Z \to \epsilon \mid ZP \\ L(P^+) = L(Z) & \text{with} & Z \to P \mid ZP \end{array}$$

There are many variations possible on this theme:

$$L(P^*Q) = L(Z) \quad \text{with} \quad Z \to Q \mid PZ \tag{2.1}$$

2.7.1 SL: an example

To illustrate EBNF and some of the grammar transformations given in the previous section, we give a larger example. The following grammar generates expressions in a very small programming language, called SL.

Expr	→	`if` *Expr* `then` *Expr* `else` *Expr*
Expr	→	*Expr* `where` *Decls*
Expr	→	*AppExpr*
AppExpr	→	*AppExpr Atomic* | *Atomic*
Atomic	→	*Var* | *Number* | *Bool* | `(`*Expr*`)`
Decls	→	*Decl*
Decls	→	*Decls* `;` *Decls*
Decl	→	*Var* `=` *Expr*

where the nonterminals *Var*, *Number*, and *Bool* generate variables, number expressions, and boolean expressions, respectively. Note that the brackets around the *Expr* in the production for *Atomic*, and the semicolon in between the *Decls* in the second production for*Decls* are also terminal symbols. The following 'program' is a sentence of this language:

```
if true then funny true else false where funny = 7
```

It is clear that this is not a very convenient language to write programs in.

The above grammar is ambiguous (why?), and we introduce priorities to resolve some of the ambiguities. Application binds stronger than if, and both application and if bind stronger then where. Using the introduction of priorities grammar transformation, we obtain:

Expr	→	*Expr1*
Expr	→	*Expr1* `where` *Decls*
Expr1	→	*Expr2*
Expr1	→	`if` *Expr1* `then` *Expr1* `else` *Expr1*
Expr2	→	*Atomic*
Expr2	→	*Expr2 Atomic*

where *Atomic*, and *Decls* have the same productions as before.

The nonterminal *Expr2* is left recursive. Removing left recursion gives the following productions for *Expr2*.

Expr2	→	*Atomic* | *Atomic Z*
Z	→	*Atomic* | *Atomic Z*

Since the new nonterminal Z has exactly the same productions as *Expr2*, these productions can be replaced by

$$\mathit{Expr2} \;\rightarrow\; \mathit{Atomic} \mid \mathit{Atomic}\ \mathit{Expr2}$$

So *Expr2* generates a nonempty sequence of atomics. Using the +-notation introduced in this section, we may replace *Expr2* by Atomic^+.

Another source of ambiguity are the productions for *Decls*. *Decls* generates a nonempty list of declarations, and the separator `;` is assumed to be associative. Hence we can apply the associative separator transformation to obtain

$$\mathit{Decls} \;\rightarrow\; \mathit{Decl} \mid \mathit{Decl}\ \texttt{;}\ \mathit{Decls}$$

or, according to (2.1),

$$\mathit{Decls} \;\rightarrow\; (\mathit{Decl}\,\texttt{;})^*\ \mathit{Decl}$$

which, using an omitted rule for that star-operator $*$, may be transformed into

$$\mathit{Decls} \;\rightarrow\; \mathit{Decl}\ (\texttt{;}\,\mathit{Decl})^*$$

The last grammar transformation we apply is left factoring. This transformation applies to *Expr*, and gives

$$\begin{aligned} \mathit{Expr} &\rightarrow \mathit{Expr1}\ Z \\ Z &\rightarrow \epsilon \mid \texttt{where}\ \mathit{Decls} \end{aligned}$$

Since nonterminal Z generates either nothing or a where clause, we may replace Z by an optional where clause in the production for *Expr*.

$$\mathit{Expr} \;\rightarrow\; \mathit{Expr1}\ (\texttt{where}\ \mathit{Decls})?$$

After all these grammar transformations, we obtain the following grammar.

$$\begin{aligned} \mathit{Expr} &\rightarrow \mathit{Expr1}\ (\texttt{where}\ \mathit{Decls})? \\ \mathit{Expr1} &\rightarrow \mathit{Atomic}^+ \\ \mathit{Expr1} &\rightarrow \texttt{if}\ \mathit{Expr1}\ \texttt{then}\ \mathit{Expr1}\ \texttt{else}\ \mathit{Expr1} \\ \mathit{Atomic} &\rightarrow \mathit{Var} \mid \mathit{Number} \mid \mathit{Bool} \mid \texttt{(}\mathit{Expr}\texttt{)} \\ \mathit{Decls} &\rightarrow \mathit{Decl}\ (\texttt{;}\,\mathit{Decl})^* \end{aligned}$$

Exercise 2.29 ▷ Give the EBNF notation for each of the basic languages defined in section 2.3.1. ◁

Exercise 2.30 ▷ What language is generated by G? ? ◁

Exercise 2.31 ▷ In case we define a language by means of a predicate it is almost trivial to define the intersection and the difference of two languages. Show how. ◁

Exercise 2.32 ▷ Let $L_1 = \{\mathsf{a}^n\mathsf{b}^n\mathsf{c}^m \mid n, m \in \mathbb{N}\}$ and $L_2 = \{\mathsf{a}^n\mathsf{b}^m\mathsf{c}^m \mid n, m \in \mathbb{N}\}$.

1. Give grammars for L_1 and L_2.
2. Is $L_1 \cap L_2$ context free, i.e. can you give a context-free grammar for this language?

◁

2.8 Parsing

This section formulates the parsing problem, and discusses some of the future topics of the course.

Definition 13: Parsing problem
Given the grammar G and a string s the *parsing problem* answers the question whether or not $s \in L(G)$. If $s \in L(G)$, the answer to this question may be either a parse tree or a derivation. □

This question may not be easy to answer given an arbitrary grammar. Until now we have only seen simple grammars for which it is easy to determine whether or not a string is a sentence of the grammar. For more complicated grammars this may be more difficult. However, in the first part of this course we will show how given a grammar with certain reasonable properties, we can easily construct parsers by hand. At the same time we will show how the parsing process can quite often be combined with the algorithm we actually want to perform on the recognised object (the semantic function). As such it provides a simple, although surprisingly efficient, introduction into the area of compiler construction.

A compiler for a programming language consists of several parts. Examples of such parts are a scanner, a parser, a type checker, and a code generator. Usually, a parser is preceded by a *scanner*, which divides an input sentence in a list of so-called *tokens*. For example, given the sentence

scanner

```
if true then funny true else false where funny = 7
```

a scanner might return the following list of tokens:

```
["if","true","then","funny","true","else","false"
,"where","funny","=","7"]
```

So a token is a syntactical entity. A scanner usually performs the first step towards an abstract syntax: it throws away layout information such as spacing and newlines. In this course we will concentrate on parsers, but some of the concepts of scanners will sometimes be used.

token

In the second part of this course we will take a look at more complicated grammars, which do not always conform to the restrictions just referred to.

By analysing the grammar we may nevertheless be able to generate parsers as well. Such generated parsers will be in such a form that it will be clear that writing such parsers by hand is far from attractive, and actually impossible for all practical cases.

One of the problems we have not referred to yet in this rather formal chapter is of a more practical nature. Quite often the sentence presented to the parser will not be a sentence of the language since mistakes were made when typing the sentence. This raises another interesting question: *What are the minimal changes that have to be made to the sentence in order to convert it into a sentence of the language?* It goes almost without saying that this is an important question to be answered in practice; one would not be very happy with a compiler which, given an erroneous input, would just reply that the "Input could not be recognised". One of the most important aspects here is to define metric for deciding about the minimality of a change; humans usually make certain mistakes more often than others. A semicolon can easily be forgotten, but the chance that an **if**-symbol was forgotten is far from likely. This is where grammar engineering starts to play a rôle.

SUMMARY

Starting from a simple example, the language of palindromes, we have introduced the concept of a context-free grammar. Associated concepts, such as derivations and parse trees were introduced.

2.9 Exercises

Exercise 2.33 ▷ Do there exist languages L such that $\overline{(L^*)} = \overline{(L)}^*$? ◁

Exercise 2.34 ▷ Give a language L such that $L = L^*$ ◁

Exercise 2.35 ▷ Under what circumstances is $L^+ = L^* - \{\epsilon\}$? ◁

Exercise 2.36 ▷ Let L be a language over alphabet $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ such that $L = L^R$. Does L contain only palindromes? ◁

Exercise 2.37 ▷ Consider the grammar with productions

$$
\begin{aligned}
S &\rightarrow AA \\
A &\rightarrow AAA \\
A &\rightarrow \mathsf{a} \\
A &\rightarrow \mathsf{b}A \\
A &\rightarrow A\mathsf{b}
\end{aligned}
$$

1. Which terminal strings can be produced by derivations of four or fewer steps?

2. Give at least two distinct derivations for the string babbab

3. For any $m, n, p \geq 0$, describe a derivation of the string $\mathsf{b}^m\mathsf{ab}^n\mathsf{ab}^p$.

◁

Exercise 2.38 ▷ Consider the grammar with productions

$$\begin{array}{lcl} S & \rightarrow & \mathsf{a}\mathsf{a}B \\ A & \rightarrow & \mathsf{b}B\mathsf{b} \\ A & \rightarrow & \epsilon \\ B & \rightarrow & A\mathsf{a} \end{array}$$

Show that the string $\mathsf{aabbaabba}$ cannot be derived from S. ◁

Exercise 2.39 ▷ Give a grammar for the language

$$L \;=\; \{\, \omega\mathsf{c}\omega^R \mid \omega \in \{\mathsf{a},\mathsf{b}\}^*\}$$

This language is known as the *center marked palindromes* language. Give a derivation of the sentence abcba. ◁

Exercise 2.40 ▷ Describe the language generated by the grammar:

$$\begin{array}{lcl} S & \rightarrow & \epsilon \\ S & \rightarrow & A \\ A & \rightarrow & \mathsf{a}A\mathsf{b} \\ A & \rightarrow & \mathsf{ab} \end{array}$$

Can you find another (preferably simpler) grammar for the same language? ◁

Exercise 2.41 ▷ Describe the languages generated by the grammars.

$$\begin{array}{lcl} S & \rightarrow & \epsilon \\ S & \rightarrow & A \\ A & \rightarrow & A\mathsf{a} \\ A & \rightarrow & \mathsf{a} \end{array}$$

and

$$\begin{array}{lcl} S & \rightarrow & \epsilon \\ S & \rightarrow & A \\ A & \rightarrow & A\mathsf{a}A \\ A & \rightarrow & \mathsf{a} \end{array}$$

Can you find other (preferably simpler) grammars for the same languages? ◁

Exercise 2.42 ▷ Show that the languages generated by the grammars G_1, G_2 en G_3 are the same.

G_1 :			G_2 :			G_3 :		
S	$\rightarrow$	ϵ	S	$\rightarrow$	ϵ	S	$\rightarrow$	ϵ
S	$\rightarrow$	$\mathsf{a}S$	S	$\rightarrow$	$S\mathsf{a}$	S	$\rightarrow$	a
						S	$\rightarrow$	SS

◁

Exercise 2.43 ▷ Consider the following property of grammars:

1. the start-symbol is the only nonterminal which may have an empty production (a production of the form $X \rightarrow \epsilon$),
2. the start symbol does not occur in any alternative.

A grammar having this property is called non-contracting. The grammar $A = \mathsf{a}A\mathsf{b} \mid \epsilon$ does not have this property. Give a non-contracting grammar which describes the same language as $A = \mathsf{a}A\mathsf{b} \mid \epsilon$. ◁

Exercise 2.44 ▷ Describe the language L of the grammar $A = A\mathsf{a}A \mid \mathsf{a}$. Give a grammar for L that has no left recursive productions. Give a grammar for L that has no right recursive productions. ◁

Exercise 2.45 ▷ Describe the language L of the grammar $X = \mathsf{a} \mid X\mathsf{b}$. Give a grammar for L that has no left recursive productions. Give a grammar for L that has no left recursive productions and which is non-contracting. ◁

Exercise 2.46 ▷ Consider the language L of the grammar

$$\begin{aligned} S &= T \mid US \\ T &= \mathsf{a}S\mathsf{a} \mid U\mathsf{a} \\ U &= S \mid SUT \end{aligned}$$

Give a grammar for L which uses only alternatives of length ≤ 2. Give a grammar for L which uses only 2 nonterminals. ◁

Exercise 2.47 ▷ Give a grammar for the language of all sequences of 0's and 1's which start with a 1 and contain exactly one 0. ◁

Exercise 2.48 ▷ Give a grammar for the language consisting of all nonempty sequences of brackets,

$$\{\,(\,,\,)\,\}$$

in which the brackets match. ( ) (( )) ( ) is a sentence of the language, give a derivation tree for it. ◁

Exercise 2.49 ▷ Give a grammar for the language consisting of all nonempty sequences of two kinds of brackets,

$$\{\,(\,,\,)\,,\,[\,,\,]\,\}$$

in which the brackets match. [( )] ( ) is a sentence of the language. ◁

Exercise 2.50 ▷ This exercise shows an example (attributed to Noam Chomsky) of an ambiguous English sentence. Consider the following grammar for a part of the English language:

$$
\begin{array}{rcl}
\textit{Sentence} & \rightarrow & \textit{Subject Predicate}. \\
\textit{Subject} & \rightarrow & \mathsf{they} \\
\textit{Predicate} & \rightarrow & \textit{Verb NounPhrase} \\
\textit{Predicate} & \rightarrow & \textit{AuxVerb Verb Noun} \\
\textit{Verb} & \rightarrow & \mathsf{are} \\
\textit{Verb} & \rightarrow & \mathsf{flying} \\
\textit{AuxVerb} & \rightarrow & \mathsf{are} \\
\textit{NounPhrase} & \rightarrow & \textit{Adjective Noun} \\
\textit{Adjective} & \rightarrow & \mathsf{flying} \\
\textit{Noun} & \rightarrow & \mathsf{planes}
\end{array}
$$

Give two different left most derivations for the sentence

they are flying planes.

◁

Exercise 2.51 ▷ • Is your grammar of exercise 2.49 unambiguous? If not, find one which *is* unambiguous. ◁

Exercise 2.52 ▷ This exercise deals with a grammar that uses unusual terminal and non-terminal symbols.

$$
\begin{array}{rcl}
\odot & \rightarrow & \odot \triangle \otimes \\
\odot & \rightarrow & \otimes \\
\otimes & \rightarrow & \otimes \diamondsuit \oplus \\
\otimes & \rightarrow & \oplus \\
\oplus & \rightarrow & \clubsuit \\
\oplus & \rightarrow & \spadesuit
\end{array}
$$

Find a derivation for the sentence $\clubsuit\diamondsuit\clubsuit\triangle\spadesuit$. ◁

Exercise 2.53 ▷ • Prove, using induction, that the grammar G for palindromes of section 2.2 does indeed generate the language of palindromes. ◁

Exercise 2.54 ▷ •• Prove that the language generated by the grammar of exercise 2.37 contains all strings over $\{\mathsf{a}, \mathsf{b}\}$ with a number of a's that is even and greater than zero. ◁

Chapter 3

Parser combinators

INTRODUCTION

This chapter is an informal introduction to writing parsers in a lazy functional language using 'parser combinators'. Parsers can be written using a small set of basic parsing functions, and a number of functions that combine parsers into more complicated parsers. The functions that combine parsers are called parser combinators. The basic parsing functions do not combine parsers, and are therefore not parser combinators in this sense, but they are usually also called parser combinators.

Parser combinators are used to write parsers that are very similar to the grammar of a language. Thus writing a parser amounts to translating a grammar to a functional program, which is often a simple task.

Parser combinators are built by means of standard functional language constructs like higher-order functions, lists, and datatypes. List comprehensions are used in a few places, but they are not essential, and could easily be rephrased using the `map`, `filter` and `concat` functions. Type classes are only used for overloading the equality and arithmetic operators.

We will start by motivating the definition of the type of parser functions. Using that type, we can build parsers for the language of (possibly ambiguous) grammars. Next, we will introduce some elementary parsers that can be used for parsing the terminal symbols of a language.

In Section 3.3 the first parser combinators are introduced, which can be used for sequentially and alternatively combining parsers, and for calculating so-called semantic functions during the parse. Semantic functions are used to give meaning to syntactic structures. As an example, we construct a parser for strings of matching parentheses in Section 3.3.1. Different semantic values are calculated for the matching parentheses: a tree describing the structure, and an integer indicating the nesting depth.

In Section 3.4 we introduce some new parser combinators. Not only do these make life easier later, but their definitions are also nice examples of using parser combinators. A real application is given in Section 3.5, where a parser for arithmetical expressions is developed. Finally, the expression parser is generalised to expressions with an arbitrary number of precedence levels. This is done without

coding the priorities of operators as integers, and we will avoid using indices and ellipses.

It is not always possible to directly construct a parser from a context-free grammar using parser combinators. If the grammar is left recursive, it has to be transformed into a non left recursive grammar before we can construct a combinator parser. Another limitation of the parser combinator technique as described in this chapter is that it is not trivial to write parsers for complex grammars that perform reasonably efficient. However, there do exist implementations of parser combinators that perform remarkably well, see [11, 13]. For example, there exist good parsers using parser combinator for Haskell.

Most of the techniques introduced in this chapter have been described by Burge [4], Wadler [14] and Hutton [7]. Recently, the use of so-called *monads* has become quite popular in connection with parser combinators [15, 8]. We will not use them in this article, however, to show that no magic is involved in using parser combinators. You are nevertheless encouraged to study monads at some time, because they form a useful generalisation of the techniques described here.

This chapter is a revised version of [5].

GOALS

This chapter introduces the first programs for parsing in these lecture notes. Parsers are composed from simple parsers by means of parser combinators. Hence, important primary goals of this chapter are:

- to understand how to parse, i.e. how to recognise structure in a sequence of symbols, by means of parser combinators;
- to be able to construct a parser when given a grammar;
- to understand the concept of semantic functions.

Two secondary goals of this chapter are:

- to develop the capability to abstract;
- to understand the concept of domain specific language.

REQUIRED PRIOR KNOWLEDGE

To understand this chapter, you should be able to formulate the parsing problem, and you should understand the concept of context-free grammar. Furthermore, you should be familiar with functional programming concepts such as `type`, `class`, and higher-order functions.

3.1 The type 'Parser'

The goals of this section are:

- develop a type `Parser` that is used to give the type of parsing functions;
- show how to obtain this type by means of several abstraction steps.

The *parsing problem* is (see Section 2.8): Given a grammar G and a string s, determine whether or not $s \in L(G)$. If $s \in L(G)$, the answer to this question may be either a parse tree or a derivation. For example, in Section 2.4 you have seen a grammar for sequences of `s`'s:

$$S \rightarrow SS \mid \texttt{s}$$

A parse tree of an expression of this language is a value of the datatype `SA` (or a value of `S`, `SA2`, `SA3`, see Section 2.6, depending on what you want to do with the result of the parser), which is defined by

```
data SA  =  BesideA SA SA | SingleA
```

A parser for expressions could be implemented as a function of the following type:

```
type Parser  =  String -> SA
```

Exercise 3.1 ▷ In section 2.6 we saw that the sequence `sss` can be represented by the parse tree `BesideA (BesideA SingleA SingleA) SingleA` and that the function `sa2string` applied on this parse tree returns the sequence `sss`.

1. Write a Haskell function `string2sa` that is the reverse function of `sa2string`. The function takes a sequence from the language represented by *SequenceOfS* and returns a parse tree for that sequence.
2. What is the result of the call of `string2sa` on the value `ssa`?

◁

For parsing substructures, a parser can call other parsers, or call itself recursively. These calls do not only have to communicate their result, but also the part of the input string that is left unprocessed. For example, when parsing the string `sss`, a parser will first build a parse tree `BesideA SingleA SingleA` for `ss`, and only then build a complete parse tree

```
BesideA (BesideA SingleA SingleA) SingleA
```

using the unprocessed part `s` of the input string. As this cannot be done using a global variable, the unprocessed input string has to be part of the result of the parser. The two results can be grouped in a tuple. A better definition for the type `Parser` is hence:

```
type Parser  =  String -> (SA,String)
```

Exercise 3.2 ▷

1. In exercise 3.1 the type of the function `string2sa` is of the previous `Parser` type. The function analyses the whole input sequence. Write a function `parserSoS` that also takes a sequence and only tries to recognise a single value of type `SA`, consuming as much as possible from the input string. The type of this function is `String -> (SA, String)`: the type `Parser`.

2. What is the result of applying the function to the values `sss`, `ssa`, `sas`, `s` and `aas`?

◁

Any parser of type `Parser` returns an `SA` and a `String`. However, for different grammars we want to return different parse trees: the type of tree that is returned depends on the grammar for which we want to parse sentences. Therefore it is better to abstract from the type `SA`, and to make the parser type into a polymorphic type. The type `Parser` is parametrised with a type `a`, which represents the type of parse trees.

```
type Parser a  =  String -> (a,String)
```

For example, a parser that returns a structure of type `Oak` (whatever that is) now has type `Parser Oak`. A parser that parses sequences of `s`'s has type `Parser SA`. We might also define a parser that does not return a value of type `SA`, but instead the number of `s`'s in the input sequence. This parser would have type `Parser Int`. Another instance of a parser is a parse function that recognises a string of digits, and returns the number represented by it as a parse 'tree'. In this case the function is also of type `Parser Int`. Finally, a recogniser that either accepts or rejects sentences of a grammar returns a boolean value, and will have type `Parser Bool`.

Until now, we have assumed that every string can be parsed in exactly one way. In general, this need not be the case: it may be that a single string can be parsed in various ways, or that there is no way to parse a string. For example, the string "sss" has the following two parse trees:

```
BesideA (BesideA SingleA SingleA) SingleA
BesideA SingleA (BesideA SingleA SingleA)
```

As another refinement of the type definition of `Parser`, instead of returning one parse tree (and its associated rest string), we let a parser return a *list* of trees. Each element of the result consists of a tree, paired with the rest string that was left unprocessed after parsing. The type definition of `Parser` therefore becomes:

```
type Parser a  =  String -> [(a,String)]
```

```
-- The type of parsers

type Parser symbol result  =  [symbol] -> [(result,[symbol])]
```

Listing 1: ParserType.hs

If there is just one parsing, the result of the parse function is a singleton list. If no parsing is possible, the result is an empty list. In case of an ambiguous grammar, the result consists of all possible parsings.

This method for parsing is called the *list of successes* method, described by Wadler [14]. It can be used in situations where in other languages you would use so-called backtracking techniques. In the Bird and Wadler textbook it is used to solve combinatorial problems like the eight queens problem [3]. If only one solution is required rather than all possible solutions, you can take the `head` of the list of successes. Thanks to lazy evaluation, not all elements of the list are computed if only the first value is needed, so there will be no loss of efficiency. Lazy evaluation provides a backtracking approach to finding the first solution.

list of successes

lazy evaluation

Parsers with the type described so far operate on strings, that is lists of characters. There is however no reason for not allowing parsing strings of elements other than characters. You may imagine a situation in which a preprocessor prepares a list of tokens (see Section 2.8), which is subsequently parsed. To cater for this situation we refine the parser type once more: we let the type of the elements of the input string be an argument of the parser type. Calling it `b`, and as before the result type `a`, the type of parsers is now defined by:

```
type Parser b a  =  [b] -> [(a,[b])]
```

or, if you prefer meaningful identifiers over conciseness:

```
type Parser symbol result  =  [symbol] -> [(result,[symbol])]
```

We will use this type definition in the rest of this chapter. This type is defined in listing 1, the first part of our parser library. The list of successes appears in result type of a parser. Each element of this list is a possible parsing of (an initial part of) the input. We will hardly use the generality provided by the `Parser` type: the type of the input `b` (or `symbol`) will almost always be `Char`.

Exercise 3.3 ▷

1. A function `p1` is of type `Parser Char Int`. How many arguments does this function take and what type do they have? What is the type of the result? Give some examples of values of the type of the result.

2. Same questions for the function `p2` of type `Parser Char (Char -> Bool)`.

◁

3.2 Elementary parsers

The goals of this section are:

- introduce some very simple parsers for parsing sentences of grammars with rules of the form:

  $$\begin{aligned} A &\rightarrow \epsilon \\ A &\rightarrow \texttt{a} \\ A &\rightarrow x \end{aligned}$$

  where x is a sequence of terminals;

- show how one can construct useful functions from simple, trivially correct functions by means of generalisation and partial parametrisation.

This section defines parsers that can only be used to parse fixed sequences of terminal symbols. For a grammar with a production that contains nonterminals in its right-hand side we need techniques that will be introduced in the following section.

We will start with a very simple parse function that just recognises the terminal symbol `a`. The type of the input string symbols is `Char` in this case, and as a parse 'tree' we also simply use a `Char`:

```
symbola  :: Parser Char Char
symbola  []                 =  []
symbola  (x:xs) | x == 'a'  =  [('a',xs)]
                | otherwise =  []
```

Exercise 3.4 ▷ What is the result of applying the parser `symbola` to the strings `"any"` and `"none"`? ◁

The list of successes method immediately pays off, because now we can return an empty list if no parsing is possible (because the input is empty, or does not start with an `a`).

In the same fashion, we can write parsers that recognise other symbols. As always, rather than defining a lot of closely related functions, it is better to abstract from the symbol to be recognised by making it an extra argument of the function. Furthermore, the function can operate on lists of characters, but also on lists of symbols of other types, so that it can be used in other applications than character oriented ones. The only prerequisite is that the symbols to be parsed can be tested for equality. In Hugs, this is indicated by the `Eq` predicate in the type of the function.

Using these generalisations, we obtain the function `symbol` that is given in listing 2. The function `symbol` is a function that, given a symbol, returns a parser for that symbol. A parser on its turn is a function too. This is why two arguments appear in the definition of `symbol`.

Exercise 3.5 ▷

```
-- Elementary parsers

symbol  ::  Eq s  =>  s -> Parser s s
symbol a []                  =  []
symbol a (x:xs) | x == a     =  [(x,xs)]
                | otherwise  =  []

satisfy  ::  (s -> Bool) -> Parser s s
satisfy p []                 =  []
satisfy p (x:xs) | p x       =  [(x,xs)]
                 | otherwise =  []

token  ::  Eq s => [s] -> Parser s [s]
token k xs  |  k == take n xs  =  [(k,drop n xs)]
            |  otherwise       =  []
  where  n = length k

failp     :: Parser s a
failp xs  =  []

succeed       :: a -> Parser s a
succeed r xs  =  [(r,xs)]

-- Applications of elementary parsers

digit  :: Parser Char Char
digit  =  satisfy isDigit
```

Listing 2: ParserType.hs

1. What is the value of the expression `symbol 'a' "any"`?
2. What is the type of the expressions `symbol 'a' "any"`, `symbol 'a'` and `symbol`?
3. Define, using the function `symbol` and partial application, a parser `spaceParser` that recognises a space character.

◁

We will now define some elementary parsers that can do the work traditionally taken care of by lexical analysers (see Section 2.8). For example, a useful parser is one that recognises a fixed string of symbols, such as while or switch. We will call this function `token`; it is defined in listing 2. As in the case of the `symbol` function we have parametrised this function with the string to be recognised, effectively making it into a family of functions. Of course, this function is not confined to strings of characters. However, we do need an equality test on the type of values in the input string; the type of `token` is:

```
token  ::  Eq s  =>  [s] -> Parser s [s]
```

The function `token` is a generalisation of the `symbol` function, in that it recognises a list of symbols instead of a single symbol. Note that we cannot define `symbol` in terms of `token`: the two functions have incompatible types.

Another generalisation of `symbol` is a function which may, depending on the input, return different parse results. Instead of specifying a specific symbol, we can parametrise the function with a *condition* that the symbol should fulfill. Thus the function `satisfy` has a function `s -> Bool` as argument. Where `symbol` tests for equality to a specific value, the function `satisfy` tests for compliance with this predicate. It is defined in listing 2. This generalised function is for example useful when we want to parse digits (characters in between `'0'` and `'9'`):

```
digit  :: Parser Char Char
digit  =  satisfy isDigit
```

where the function `isDigit` is the standard predicate that tests whether or not a character is a digit:

```
isDigit    :: Char -> Bool
isDigit x  =  '0' <= x && x <= '9'
```

In books on grammar theory an empty string is often called 'epsilon'. In this tradition, we will define a function `epsilon` that 'parses' the empty string. It does not consume any input, and hence always returns an empty parse tree and unmodified input. A zero-tuple can be used as a result value: `()` is the only value of the type `()`.

```
epsilon     :: Parser s ()
epsilon xs  =  [((),xs)]
```

A more useful variant is the function `succeed`, which also doesn't consume input, but always returns a given, fixed value (or 'parse tree', if you can call the result of processing zero symbols a parse tree). It is defined in listing 2.

Dual to the function `succeed` is the function `failp`, which fails to recognise any symbol on the input string. As the result list of a parser is a 'list of successes', and in the case of failure there are no successes, the result list should be empty. Therefore the function `failp` always returns the empty list of successes. It is defined in listing 2. Note the difference with `epsilon`, which *does* have one element in its list of successes (albeit an empty one).

Do not confuse `failp` with `epsilon`: there is an important difference between returning one solution (which contains the unchanged input as 'rest' string) and not returning a solution at all!

Exercise 3.6 ▷ Give examples of applications together with their results of the functions in listing 2 and the function `epsilon`. ◁

Exercise 3.7 ▷ Define a function `capital :: Parser Char Char` that parses capital letters. ◁

Exercise 3.8 ▷ Since `satisfy` is a generalisation of `symbol`, the function `symbol` can be defined as an instance of `satisfy`. How can this be done? ◁

Exercise 3.9 ▷ Define the function `epsilon` using `succeed` ◁.

3.3 Parser combinators

Using the elementary parsers from the previous section, parsers can be constructed for terminal symbols from a grammar. More interesting are parsers for *non*terminal symbols. It is convenient to *construct* these parsers by partially parametrising higher-order functions.

The goals of this section are:

- show how parsers can be constructed directly from the productions of a grammar. The kind of productions for which parsers will be constructed are

  $$\begin{aligned} A &\rightarrow x \mid y \\ A &\rightarrow x\ y \end{aligned}$$

  where $x, y \in (N \cup T)^*$;

- show how we can construct a small, powerful combinator language (a domain specific language) for the purpose of parsing;

- understand and use the concept of semantic functions in parsers.

Let us have a look at the grammar for expressions again, see also Section 2.5:

$$\begin{aligned} E &\rightarrow T\texttt{+}E \mid T \\ T &\rightarrow F\texttt{*}T \mid F \\ F &\rightarrow \mathit{Digs} \mid \texttt{(}E\texttt{)} \end{aligned}$$

where *Digs* is a nonterminal that generates the language of sequences of digits, see Section 2.3.1. An expression can be parsed according to any of the two rules for E. This implies that we want to have a way to say that a parser consists of several alternative parsers. Furthermore, the first rule says that in order to parse an expression, we should first parse a term, then a terminal symbol `+`, and then an expression. This implies that we want to have a way to say that a parser consists of several parsers that are applied sequentially.

So important operations on parsers are sequential and alternative composition: a more complex construct can consist of a simple construct *followed* by another construct (sequential composition), or by a *choice* between two constructs (alternative composition). These operations correspond directly to their grammatical counterparts. We will develop two functions for this, which for notational convenience are defined as operators: `<*>` for sequential composition, and `<|>` for alternative composition. The names of these operators are chosen so that they can be easily remembered: `<*>` 'multiplies' two constructs together, and `<|>` can be pronounced as 'or'. Be careful, though, not to confuse the `<|>`-operator with Hugs' built-in construct `|`, which is used to distinguish cases in a function definition.

Priorities of these operators are defined so as to minimise parentheses in practical situations:

```
infixl 6 <*>
infixr 4 <|>
```

So `<*>` has a higher priority than `<|>`.
Both operators take two parsers as argument, and return a parser as result. By again combining the result with other parsers, you may construct even more involved parsers.

In the definitions in listing 3, the functions operate on parsers `p` and `q`. Apart from the arguments `p` and `q`, the function operates on a string, which can be thought of as the string that is parsed by the parser that is the result of combining `p` and `q`.

We start with the definition of operator `<*>`. For sequential composition, `p` must be applied to the input first. After that, `q` is applied to the rest string of the result. The first parser, `p`, returns a list of successes, each of which contains a value and a rest string. The second parser, `q`, should be applied to the rest string, returning a second value. Therefore we use a list comprehension, in which the second parser is applied in all possible ways to the rest string of the first parser:

```
(p <*> q) xs  =  [(combine r1 r2, zs)
```

```
-- Parser combinators

(<|>)         :: Parser s a      -> Parser s a -> Parser s a
(p <|> q) xs  =  p xs ++ q xs

(<*>)         :: Parser s (b -> a) -> Parser s b -> Parser s a
(p <*> q) xs  =  [(f x,zs)
                 |(f  ,ys) <- p xs
                 ,(  x,zs) <- q ys
                 ]

(<$>)         :: (a -> b) -> Parser s a -> Parser s b
(f <$> p) xs  =  [(f y,ys)
                 |(  y,ys) <- p xs
                 ]

-- Applications of parser combinators

newdigit  :: Parser Char Int
newdigit  =  f <$> digit
  where  f c = ord c - ord '0'
```

Listing 3: ParserCombinators.hs

```
                 |(r1,ys) <- p xs
                 ,(r2,zs) <- q ys
                 ]
```

The rest string of the parser for the sequential composition of `p` and `q` is whatever the second parser `q` leaves behind as rest string.

Now, how should the results of the two parsings be combined? We could, of course, parametrise the whole thing with an operator that describes how to combine the parts (as is done in the `zipWith` function). However, we have chosen for a different approach, which nicely exploits the ability of functional languages to manipulate functions. The function `combine` should combine the results of the two parse trees recognised by `p` and `q`. In the past, we have interpreted the word 'tree' liberally: simple values, like characters, may also be used as a parse 'tree'. We will now also accept *functions* as parse trees. That is, the result type of a parser may be a function type.

If the first parser that is combined by `<*>` would return a function of type `b -> a`, and the second parser a value of type `b`, a straightforward choice for the `combine` function would be function application. That is exactly the approach taken in the definition of `<*>` in listing 3. The first parser returns a function, the second parser a value, and the combined parser returns the value that is

obtained by applying the function to the value.

Exercise 3.10 ▷ In this exercise we will define a parser `abParser` that analyses the first two characters of a string and recognises the string `"ab"`. The parser will be defined by using the parser combinator `<*>` and two parsers that each analyse a single character.

1 What is the type of the parser `abParser`?
2 Explain why the parser `(symbol 'a' <*> symbol 'b')` is incorrect. What should the type be of the two parsers that are combined with the combinator `<*>`?
3 What does the function `('a':)` do and what is its type?
4 Define a parser `symbolaParser` that analyses a string and returns the function `('a':)` when the character `'a'` is recognised. What is the type of `symbolaParser`?
5 Define a parser `symbolbParser` that analyses a string and returns the string `"b"` when the character `'b'` is recognised. What is the type of `symbolbParser`?
6 Define the parser `abParser` by using `symbolaParser` and `symbolbParser`. What is the result of the call of `abParser` on the strings `"abcd"` and `"bcd"`?

◁

Exercise 3.11 ▷ Define a parser `gbParser` that analyses the characters of a string and recognises strings that begin with the two tokens `"good "` and `"bye "`, both tokens ending with a space character. What is the type of this parser? ◁

Apart from 'sequential composition' we need a parser combinator for representing 'choice'. For this, we have the parser combinator operator `<|>`. Thanks to the list of successes method, both `p1` and `p2` return lists of possible parsings. To obtain all possible parsings when applying `p1` or `p2`, we only need to concatenate these two lists.

Exercise 3.12 ▷ Define a parser `aOrAParser` that analyses a string and recognises the character `'a'` or `'A'`. ◁

Exercise 3.13 ▷ Define a parser `gParser` that analyses a string and recognises strings that begins with `"good "` or `"good bye "`. What is the result of applying the parser to the strings `"good friends"` and `"good bye friends"`? ◁

By combining parsers with parser combinators we can construct new parsers. The most important parser combinators are `<*>` and `<|>`. The parser combinator `<,>` in exercise 24 is just a variation of `<*>`.

Sometimes we are not quite satisfied with the result value of a parser. The parser might work well in that it consumes symbols from the input adequately (leaving the unused symbols as rest-string in the tuples in the list of successes), but the result value might need some postprocessing. For example, a parser that recognises one digit is defined using the function `satisfy`: `digit = satisfy isDigit`. In some applications, we may need a parser that recognises one digit character, but returns the result as an integer, instead of a character. In a case like this,

we can use a new parser combinator: `<$>`. It takes a function and a parser as argument; the result is a parser that recognises the same string as the original parser, but 'postprocesses' the result using the function. We use the `$` sign in the name of the combinator, because the combinator resembles the operator that is used for normal function application in Haskell: `f $ x = f x`. The definition of `<$>` is given in listing 3. It is an infix operator:

```
infixl 7 <$>
```

Using this postprocessing parser combinator, we can modify the parser `digit` that was defined above:

```
newdigit  :: Parser Char Int
newdigit  =  f  <$>  digit
  where  f c = ord c - ord '0'
```

The auxiliary function `f` determines the ordinal number of a digit character; using the parser combinator `<$>` it is applied to the result part of the `digit` parser.

In practice, the `<$>` operator is used to build a certain value during parsing (in the case of parsing a computer program this value may be the generated code, or a list of all variables with their types, etc.). Put more generally: using `<$>` we can add *semantic functions* to parsers.

Exercise 3.14 ▷

1. What is the result of applying the parsers `digit` and `newdigit` to the string `"1abc"`?

2. Give another definition of the parser `newdigit` using a lambda abstraction.

◁

Exercise 3.15 ▷

1. Define, using the combinator `<$>`, a parser `symbolaParser'` that has the same functionality as `symbolaParser` of exercise 3.10.

2. Test the parser `symbolaParser'` using the parser
   `abParser' = symbolaParser' <*> symbolbParser`
   see also exercise 3.10.6.

3. Define a parser `abParser''` with the same functionality as `abParser'`, but now by using the combinators `<$>` and `<*>`, the parsers `(symbol 'a')` and `(symbol 'b')` and a function defined using a lambda abstraction.

◁

Exercise 3.16 ▷ Consider the grammar G on the alphabet $\{\mathtt{a}, \mathtt{b}\}$ with the following production rules:

$$\begin{array}{lrcl} AnA: & S & \rightarrow & \mathtt{a}S \\ AB: & S & \rightarrow & \mathtt{b} \end{array}$$

This grammar generates the language $L(G) = \mathtt{a}^*\mathtt{b}$.

1. Give a datatype that can be used to represent parse trees for sentences of $L(G)$.
2. Give the parse tree for the string `"aaab"`.
3. Define a parser `sabParser` that recognises sentences of $L(G)$. What is the type of the parser?
4. What is the result of applying the parser `sabParser` to the strings `"aaab"` and `"baaa"`. Give an interpretation of the result.
5. Define a parser `countaParser` that counts the number of occurrences of the character `'a'` in a sentence of $L(G)$.

◁

Exercise 3.17 ▷

1. Define a parser for the language of exercise 2.15.
2. Give a parser `countabParser` that counts the number of occurrences of the string `"ab"`.

◁

A parser for the *SequenceOfS* grammar that returns the abstract syntax tree of the input, i.e., a value of type `SA`, see Section 2.6, is defined as follows:

```
sequenceOfS  :: Parser Char SA
sequenceOfS  =  BesideA <$> sequenceOfS <*> sequenceOfS
            <|> const SingleA <$> symbol 's'
```

But if you try to run this function, you will get a stack overflow! If you apply `sequenceOfS` to a string, the first thing it does is to apply itself to the same string, which ... The problem is that the underlying grammar is left recursive, and you cannot use parser combinators to parse sentences of left recursive grammars. In section 2.5 we have shown how to remove the left recursion in the *SequenceOfS* grammar. The resulting grammar is used to obtain the following parser:

```
sequenceOfS'  :: Parser Char SA2
sequenceOfS'  =
       const ConsS <$> symbol 's' <*> parseZ
   <|> const SingleS <$> symbol 's'
   where parseZ  =  ConsSA2 <$> sequenceOfS' <*> parseZ
                <|> SingleSA2 <$> sequenceOfS'
```

This example is a direct translation of the grammar obtained by using the removing left recursion grammar transformation. There exists a much simpler parser for parsing sequences of s's.

Exercise 3.18 ▷ Prove that for all `f :: a -> b`

```
f <$> succeed a  =  succeed (f a)
```

In the sequel we will often use this rule for constant functions `f`, i.e. `f = \a -> c`, where `c` is a value that does not contain `a`. ◁

Exercise 3.19 ▷ Consider the parser `list <$> symbol 'a'`, where `list a as = a:as`. Give its type and show its results on inputs `[]` and `x:xs`. ◁

Exercise 3.20 ▷

1. Consider the parser `list <$> symbol 'a' <*> p`. Give its type and show its results on inputs `[]` and `x:xs`.
2. What is the type of the parser `p`? Give an example of a parser of this type.

◁

Exercise 3.21 ▷ Define a parser for Booleans. ◁

Exercise 3.22 ▷ Consider the grammar for *palindromes* that you have constructed in exercise 2.8.

1. Give the datatype `Pal2` that corresponds to this grammar.
2. Define a parser `palin2` that returns parse trees for palindromes. Test your function with the palindromes `cPal1 = "abaaba"` and `cPal2 = "baaab"`.
3. Define a parser `palina` that counts the number of `a`'s occurring in a palindrome.

◁

Exercise 3.23 ▷ When defining the priority of the `<|>` operator with the `infixr` keyword, we also specified that the operator associates to the right. Why is this a better choice than association to the left? ◁

Exercise 3.24 ▷ Define a parser combinator `<,>` that combines two parsers. The value returned by the combined parser is a tuple containing the results of the two component parsers. What is the type of this parser combinator? ◁

Exercise 3.25 ▷ The term 'parser combinator' is in fact not an adequate description for `<$>`. Can you think of a better word? ◁

Exercise 3.26 ▷ Compare the type of `<$>` with the type of the standard function `map`. Can you describe your observations in an easy-to-remember, catchy phrase? ◁

Exercise 3.27 ▷ Define `<*>` in terms of `<,>` and `<$>`. Define `<,>` in terms of `<*>` and `<$>`. ◁

Exercise 3.28 ▷ If you examine the definitions of `<*>` and `<$>` in listing 3, you can observe that `<$>` is in a sense a special case of `<*>`. Can you define `<$>` in terms of `<*>`? ◁

3.3.1 Matching parentheses: an example

Using parser combinators, it is often fairly straightforward to construct a parser for a language for which you have a grammar. Consider, for example, the grammar that you wrote in exercise 2.48:

$$S \;\rightarrow\; (\; S\;)\; S \mid \epsilon$$

This grammar can be directly translated to a parser, using the parser combinators `<*>` and `<|>`. We use `<*>` when symbols are written next to each other, and `<|>` when | appears in a production (or when there is more than one production for a nonterminal).

```
parens  :: Parser Char ???
parens  =  symbol '(' <*> parens <*> symbol ')' <*> parens
       <|> epsilon
```

However, this function is not correctly typed: the parsers in the first alternative cannot be composed using `<*>`, as for example `symbol '('` is not a parser returning a function.

But we can postprocess the parser `symbol '('` so that, instead of a character, this parser *does* return a function. So, what function should we use? This depends on the kind of value that we want as a result of the parser. A nice result would be a tree-like description of the parentheses that are parsed. For this purpose we introduce an abstract syntax, see Section 2.6, for the parentheses grammar. A first abstract syntax is given by the datatype `Parentheses1`.

```
data Parentheses1  =  Match1 Char Parentheses1 Char Parentheses1
                   |  Empty1
```

For example, the sentence ()() is represented by

```
Match1 '(' Empty1 ')' (Match1 '(' Empty1 ')' Empty1)
```

Suppose we want to calculate the number of parentheses in a sentence. The number of parentheses is calculated by the function `nrofpars`, which is defined by induction on the datatype `Parentheses1`.

```
nrofpars                        :: Parentheses1 -> Int
nrofpars (Match1 cl pl cr pr)   =  nrofpars pl + nrofpars pr + 2
nrofpars Empty1                 =  0
```

```
data Parentheses  =  Match Parentheses Parentheses
                  |  Empty
                  deriving Show

open   =  symbol '('
close  =  symbol ')'

parens  :: Parser Char Parentheses
parens  =  f  <$> open <*> parens <*> close <*> parens
       <|> succeed Empty
  where f a b c d = Match b d

nesting  :: Parser Char Int
nesting  =  f <$> open <*> nesting <*> close <*> nesting
        <|> succeed 0
  where f a b c d = max (1+b) d
```

Listing 4: ParseParentheses.hs

Since the values `cl` and `cr` in the case for `Match1` are not used (and should never be used) by functions defined on `Parentheses1`, we use the following datatype for the abstract syntax for the grammar of parentheses.

```
data Parentheses  =  Match Parentheses Parentheses
                  |  Empty
```

Now we can add 'semantic functions' to the parser. Thus, we get the definition of `parens` in listing 4.

By varying the function used before `<$>` (the 'semantic function'), we can return other things than parse trees. As an example we construct a parser that calculates the nesting depth of nested parentheses, see the function `nesting` defined in listing 4.

A session in which `nesting` is used may look like this:

```
? nesting "()(())()"
[(2,[]), (2,"()"), (1,"(())()"), (0,"()(())()")]
? nesting "())"
[(1,")"), (0,"())")]
```

As you can see, when there is a syntax error in the argument, there are no solutions with empty rest string. It is fairly simple to test whether a given string belongs to the language that is parsed by a given parser.

Exercise 3.29 ▷ What is the type of the function `f` which appears in function `parens` in listing 4? What is the type of the parser `open`? Using the type of `<$>`, what is the type of `f <$> open` ? Can `f <$> open` be used as a left hand side of `<*> parens`? What is the type of the result? ◁

Exercise 3.30 ▷ What is a convenient way for `<*>` to associate? Does it? ◁

Exercise 3.31 ▷ Write a function `test` that determines whether or not a given string belongs to the language parsed by a given parser. ◁

3.4 More parser combinators

In principle you can build parsers for any context-free language using the combinators `<*>` and `<|>`, but in practice it is easier to have some more parser combinators available. In traditional grammar formalisms, additional symbols are used to describe for example optional or repeated constructions. Consider for example the BNF formalism, in which originally only sequential and alternative composition can be used (denoted by juxtaposition and vertical bars, respectively), but which was later extended to EBNF to also allow for repetition, denoted by a star. The goal of this section is to show how the set of parser combinators can be extended.

3.4.1 Parser combinators for EBNF

It is very easy to make new parser combinators for EBNF. As a first example we consider repetition. Given a parser for a construction, `many` constructs a parser for zero or more occurrences of that construction:

```
many    :: Parser s a  -> Parser s [a]
many p  =  list <$> p <*> many p
       <|> succeed []
```

So the EBNF expression P^* is implemented by `many P`. The auxiliary function `list` takes an element and a list, and combines them in a simple list:

```
list x xs  =  x:xs
```

So `list` is just `(:)`, and we might have written that in the definition of `many`, but then the definition might have looked too cryptic at first sight.

The order in which the alternatives are given only influences the order in which solutions are placed in the list of successes.

For example, the `many` combinator can be used in parsing a natural number:

```
natural   :: Parser Char Int
natural   =  foldl f 0  <$>  many newdigit
  where  f a b = a*10 + b
```

Defined in this way, the `natural` parser also accepts empty input as a number. If this is not desired, we had better use the `many1` parser combinator, which accepts one or more occurrences of a construction, and corresponds to the EBNF expression P^+, see Section 2.7. It is defined in listing 5.

Exercise 3.32 ▷

```
-- EBNF parser combinators

option      :: Parser s a -> a -> Parser s a
option p d  =  p <|> succeed d

many    :: Parser s a  -> Parser s [a]
many p  =  list <$> p <*> many p <|> succeed []

many1    :: Parser s a -> Parser s [a]
many1 p  =  list <$> p <*> many p

pack        :: Parser s a -> Parser s b -> Parser s c -> Parser s b
pack p r q  =  (\x y z -> y) <$> p <*> r <*> q

listOf      :: Parser s a -> Parser s b -> Parser s [a]
listOf p s  =  list <$> p <*> many ((\x y -> y) <$> s <*> p)

-- Auxiliary functions

first  ::  Parser s b -> Parser s b
first p xs  |  null r     =  []
            |  otherwise  =  [head r]
  where r = p xs

greedy, greedy1  ::  Parser s b -> Parser s [b]
greedy   =  first . many
greedy1  =  first . many1

list x xs  =  x:xs
```

Listing 5: EBNF.hs

1. What is the result of the call of `(many (symbol 'a'))` on `"aaab"`?
2. Define using the parser `many` a parser for the language a^*b.

◁

Exercise 3.33 ▷ What is the result of the call of `(many (symbol 'a'))` and `(many1 (symbol 'a'))` on `"baa"`? ◁

Another combinator from EBNF is the `option` combinator $P?$. It takes a parser as argument, and returns a parser that recognises the same construct, but which also succeeds if that construct is not present in the input string. The definition is given in listing 5. It has an additional argument: the value that should be used as result in case the construct is not present. It is a kind of 'default' value.

By the use of the `option` and `many` functions, a large amount of backtracking possibilities are introduced. This is not always advantageous. For example, if we define a parser for identifiers by

```
identifier  =  many1 (satisfy isAlpha)
```

a single word may also be parsed as two identifiers. Caused by the order of the alternatives in the definition of `many` (`succeed []` appears as the second alternative), the 'greedy' parsing, which accumulates as many letters as possible in the identifier is tried first, but if parsing fails elsewhere in the sentence, also less greedy parsings of the identifier are tried – in vain. You will give a better definition of `identifier` in Exercise 3.41.

In situations where from the way the grammar is built we can predict that it is hopeless to try non-greedy results of `many`, we can define a parser transformer `first`, that transforms a parser into a parser that only returns the first possible parsing. It does so by taking the first element of the list of successes.

```
first  ::  Parser a b -> Parser a b
first p xs | null r     =  []
           | otherwise  =  [head r]
  where  r = p xs
```

Using this function, we can create a special 'take all or nothing' version of `many`:

```
greedy   =  first . many
greedy1  =  first . many1
```

If we compose the `first` function with the `option` parser combinator:

```
obligatory p d =  first (option p d)
```

we get a parser which must accept a construction if it is present, but which does not fail if it is not present.

Exercise 3.34 ▷ What is the result of applying `(option (symbol 'a') '?')` to `"ab"` and `"ba"`? ◁

Exercise 3.35 ▷ Define a parser `integer` that parses integers. ◁

Exercise 3.36 ▷

1. Consider the application of the parser `many (symbol 'a')` to the string `"aaa"`. In what order do the four possible parsings appear in the list of successes?

2. What is the result when `many` is replaced by `greedy`?

◁

3.4.2 Separators

The combinators `many`, `many1` and `option` are classical in compiler constructions —there are notations for it in EBNF (*, + and ?, respectively)—, but there is no need to leave it at that. For example, in many languages constructions are frequently enclosed between two meaningless symbols, most often some sort of parentheses. For this case we design a parser combinator `pack`. Given a parser for an opening token, a body, and a closing token, it constructs a parser for the enclosed body, as defined in listing 5. Special cases of this combinator are:

```
parenthesised p  =  pack (symbol '(')    p (symbol ')')
bracketed p      =  pack (symbol '[')    p (symbol ']')
compound p       =  pack (token "begin") p (token "end")
```

Exercise 3.37 ▷ In this exercise we will define a parser that recognises very simple HTML text. A text is enclosed by the tags `<html>` and `</html>`. The text begins after the opening tag and ends when the first opening angle `<` is encountered.

1. Define a parser `notangle` that accepts a character if it is not an opening angle.

2. Define a parser `textwithoutangle` that returns the text before an opening angle.

3. Define using `pack` the parser `simpleHtmlParser` that recognises simple HTML text enclosed by the tags `<html>` and `</html>`. Give examples of applications of the parser.

◁

Another frequently occurring construction is repetition of a certain construction, where the elements are separated by some symbol. You may think of lists of arguments (expressions separated by commas), or compound statements (statements separated by semicolons). For the parse trees, the separators are of no importance. The function `listOf` below generates a parser for a non-empty list, given a parser for the items and a parser for the separators:

```
-- Chain expression combinators

chainr  ::  Parser s a -> Parser s (a -> a -> a) -> Parser s a
chainr pe po  =  h <$> many (j <$> pe <*> po) <*> pe
  where j x op  =  (x `op`)
        h fs x  =  foldr ($) x fs

chainl  ::  Parser s a -> Parser s (a -> a -> a) -> Parser s a
chainl pe po  =  h <$> pe <*> many (j <$> po <*> pe)
  where j op x  =  (`op` x)
        h x fs  =  foldl (flip ($)) x fs
```

Listing 6: Chains.hs

```
listOf      :: Parser s a -> Parser s b -> Parser s [a]
listOf p s  =  list <$> p <*> many ((\x y -> y) <$> s <*> p )
```

Useful instantiations are:

```
commaList, semicList  ::  Parser Char a -> Parser Char [a]
commaList p  =  listOf p (symbol ',')
semicList p  =  listOf p (symbol ';')
```

A somewhat more complicated variant of the function `listOf` is the case where the separators carry a meaning themselves. For example, in arithmetical expressions, where the operators that separate the subexpressions have to be part of the parse tree. For this case we will develop the functions `chainr` and `chainl`. These functions expect that the parser for the separators returns a function (!); that function is used by `chain` to combine parse trees for the items. In the case of `chainr` the operator is applied right-to-left, in the case of `chainl` it is applied left-to-right. The functions `chainr` and `chainl` are defined in listing 6 (remember that `$` is function application: `f $ x = f x`).

The definitions look quite complicated, but when you look at the underlying grammar they are quite straightforward. Suppose we apply operator $\oplus$ ($\oplus$ is an operator variable, it denotes an arbitrary right-associative operator) from right to left, so

$$\begin{aligned} & e_1 \oplus e_2 \oplus e_3 \oplus e_4 \\ = & \\ & e_1 \oplus (e_2 \oplus (e_3 \oplus e_4)) \\ = & \\ & ((e_1\oplus) \cdot (e_2\oplus) \cdot (e_3\oplus))\ e_4 \end{aligned}$$

It follows that we can parse such expressions by parsing many pairs of expressions and operators, turning them into functions, and applying all those functions to the last expression. This is done by function `chainr`, see listing 6.

If operator $\oplus$ is applied from left to right, then

$$
\begin{aligned}
& e_1 \oplus e_2 \oplus e_3 \oplus e_4 \\
= & \\
& ((e_1 \oplus e_2) \oplus e_3) \oplus e_4 \\
= & \\
& ((\oplus e_4) \cdot (\oplus e_3) \cdot (\oplus e_2))\ e_1
\end{aligned}
$$

So such an expression can be parsed by first parsing a single expression (e_1), and then parsing many pairs of operators and expressions, turning them into functions, and applying all those functions to the first expression. This is done by function `chainl`, see listing 6.

Functions `chainl` and `chainr` can be made more efficient by avoiding the construction of the intermediate list of functions. The resulting definitions can be found in [5].

Note that functions `chainl` and `chainr` are very similar, the only difference is that everything is 'turned around': function `j` of `chainr` takes a value and an operator, and returns the function obtained by 'left' applying the operator; function `j` of `chainl` takes an operator and a value, and returns the function obtained by 'right' applying the operator to the value. Such functions are sometimes called *dual*. dual

Exercise 3.38 ▷

1. Define a parser that analyses a string and recognises a list of digits separated by a space character. The result is a list of integers.
2. Define a parser `sumParser` that recognises digits separated by the character `'+'` and returns the sum of these integers.
3. Both parsers return a list of solutions. What should be changed in order to get only one solution?

◁

Exercise 3.39 ▷ As another variation on the theme 'repetition', define a parser combinator `psequence` that transforms a *list of parsers* for some type into a *parser returning a list* of elements of that type. What is the type of `psequence`? Also define a combinator `choice` that iterates the operator `<|>`. ◁

Exercise 3.40 ▷ As an application of `psequence`, define the function `token` that was discussed in Section 3.2. ◁

Exercise 3.41 ▷ In real programming languages, identifiers follow more flexible rules: the first symbol must be a letter, but the symbols that follow (if any) may be a letter, digit, or underscore symbol. Define a more realistic parser `identifier`. ◁

3.5 Arithmetical expressions

The goal of this section is to use parser combinators in a concrete application. We will develop a parser for arithmetical expressions, which have the following

concrete syntax:

$$E \;\rightarrow\; E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E \mid E \texttt{ / } E \mid \texttt{(}E\texttt{)} \mid \mathit{Digs}$$

Besides these productions, we also have productions for identifiers and applications of functions:

$$\begin{aligned} E &\rightarrow \mathit{Identifier} \mid \mathit{Identifier}\ \texttt{(}\mathit{LoA}\texttt{)} \\ \mathit{LoA} &\rightarrow \epsilon \mid E(\texttt{,}E)^* \end{aligned}$$

The parse trees for this grammar are of type `Expr`:

```
data Expr  =  Con Int
           |  Var String
           |  Fun String [Expr]
           |  Expr :+: Expr
           |  Expr :-: Expr
           |  Expr :*: Expr
           |  Expr :/: Expr
```

You can almost recognise the structure of the parser in this type definition. But in order to account for the priorities of the operators, we will use a grammar with three non-terminals 'expression', 'term' and 'factor': an expression is composed of terms separated by + or −; a term is composed of factors separated by ∗ or /, and a factor is a constant, variable, function call, or expression between parentheses.

This grammar appears as a parser in the functions in listing 7.

The first parser, `fact`, parses factors.

```
fact  :: Parser Char Expr
fact  =  Con <$> integer
     <|> Var <$> identifier
     <|> Fun <$> identifier <*> parenthesised (commaList expr)
     <|> parenthesised expr
```

The first alternative is an integer parser which is postprocessed by the 'semantic function' `Con`. The second and third alternative are a variable or function call, depending on the presence of an argument list. In absence of the latter, the function `Var` is applied, in presence the function `Fun`. For the fourth alternative there is no semantic function, because the meaning of an expression between parentheses is the meaning of the expression.

For the definition of a term as a list of factors separated by multiplicative operators we use the function `chainl`. Recall that `chainl` repeatedly recognises its first argument (`fact`), separated by its second argument (a ∗ or /). The parse trees for the individual factors are joined by the constructor functions that appear before `<$>`. We use `chainl` and not `chainr` because the operator `'/'` is considered to be left-associative.

```
-- Type definition for parse tree

data Expr = Con Int
          | Var String
          | Fun String [Expr]
          | Expr :+: Expr
          | Expr :-: Expr
          | Expr :*: Expr
          | Expr :/: Expr

------------------------------------------------------------
-- Parser for expressions with two priorities

fact  :: Parser Char Expr
fact  =  Con <$> integer
     <|> Var <$> identifier
     <|> Fun <$> identifier <*> parenthesised (commaList expr)
     <|> parenthesised expr

integer  ::  Parser Char Int
integer  =  (const negate <$> (symbol '-')) `option` id  <*>  natural

term  :: Parser Char Expr
term  =  chainl fact
               (    const (:*:) <$> symbol '*'
                <|> const (:/:) <$> symbol '/'
               )

expr  :: Parser Char Expr
expr  =  chainl term
               (    const (:+:) <$> symbol '+'
                <|> const (:-:) <$> symbol '-'
               )
```

Listing 7: ExpressionParser.hs

The function `expr` is analogous to `term`, only with additive operators instead of multiplicative operators, and with `term`s instead of `fact`ors.

This example clearly shows the strength of parsing with parser combinators. There is no need for a separate formalism for grammars; the production rules of the grammar are combined with higher-order functions. Also, there is no need for a separate parser generator (like 'yacc'); the functions can be viewed both as description of the grammar and as an executable parser.

Exercise 3.42 ▷

1. Give the parse tree for the expressions `"abc"`, `"(abc)"`, `"a*b+1"`, `"a*(b+1)"`, `"-1-a"`, and `"a(1,b)"`
2. Why is the parse tree for the expression `"a(1,b)"` not the first solution of the parser? Modify the functions in listing 7 in such way that it will be.

◁

Exercise 3.43 ▷ A function with no arguments such as `"f()"` is not accepted by the parser. Explain why and modify the parser in such way that it will be. ◁

Exercise 3.44 ▷ Modify the functions in listing 7, in such a way that $-$ is parsed as a right associative operator, and $+$ is parsed as a left associative operator. ◁

3.6 Generalised expressions

This section generalises the parser in the previous section with respect to priorities. Arithmetical expressions in which operators have more than two levels of priority can be parsed by writing more auxiliary functions between `term` and `expr`. The function `chainl` is used in each definition, with as first argument the function of one priority level lower.

If there are nine levels of priority, we obtain nine copies of almost the same text. This is not as it should be. Functions that resemble each other are an indication that we should write a generalised function, where the differences are described using extra arguments. Therefore, let us inspect the differences in the definitions of `term` and `expr` again. These are:

- The operators and associated tree constructors that are used in the second argument of `chainl`
- The parser that is used as first argument of `chainl`

The generalised function will take these two differences as extra arguments: the first in the form of a list of pairs, the second in the form of a parse function:

```
type Op a  =  (Char,a -> a -> a)

gen        :: [Op a] -> Parser Char a -> Parser Char a
gen ops p  =  chainl p (choice (map f ops))
  where  f (s,c) = const c <$> symbol s
```

If furthermore we define as shorthand:

```
multis  =  [ ('*',(:*:)), ('/',(:/:)) ]
addis   =  [ ('+',(:+:)), ('-',(:-:)) ]
```

then `expr` and `term` can be defined as partial parametrisations of `gen`:

```
expr  =  gen addis  term
term  =  gen multis fact
```

By expanding the definition of `term` in that of `expr` we obtain:

```
expr  =  addis `gen` (multis `gen` fact)
```

which an experienced functional programmer immediately recognises as an application of `foldr`:

```
expr  =  foldr gen fact [addis, multis]
```

From this definition a generalisation to more levels of priority is simply a matter of extending the list of operator-lists.

The very compact formulation of the parser for expressions with an arbitrary number of priority levels is possible because the parser combinators can be used together with the existing mechanisms for generalisation and partial parametrisation in Haskell.

Contrary to conventional approaches, the levels of priority need not be coded explicitly with integers. The only thing that matters is the relative position of an operator in the list of 'list with operators of the same priority'. Also, the insertion of new priority levels is very easy. The definitions are summarised in listing 8.

SUMMARY

This chapter shows how to construct parsers from simple combinators. It shows how a small parser combinator library can be a powerful tool in the construction of parsers. Furthermore, this chapter gives a rather basic implementation of the parser combinator library. More advanced implementations are discussed elsewhere.

3.7 Exercises

Exercise 3.45 ▷ How should the parser of section 3.6 be adapted to also allow raising an expression to the power of an expression? ◁

Exercise 3.46 ▷ Prove the following laws

1. `h <$> (f <$> p) = (h.f) <$> p`

```
-- Parser for expressions with aribitrary many priorities

type Op a  =  (Char,a -> a -> a)

fact'  :: Parser Char Expr
fact'  =  Con <$> integer
      <|> Var <$> identifier
      <|> Fun <$> identifier <*> parenthesised (commaList expr')
      <|> parenthesised expr'

gen        :: [Op a] -> Parser Char a -> Parser Char a
gen ops p  =  chainl p (choice (map f ops))
  where f (s,c) = const c <$> symbol s

expr'  :: Parser Char Expr
expr'  =  foldr gen fact' [addis, multis]

multis  =  [ ('*',(:*:)), ('/',(:/:)) ]
addis   =  [ ('+',(:+:)), ('-',(:-:)) ]
```

Listing 8: GExpressionParser.hs

2. `h <$> (p <|> q) = (h <$> p) <|> (h <$> q)`

3. `h <$> (p <*> q) = ((h.) <$> p) <*> q`

◁

Exercise 3.47 ▷ Consider your answer to exercise 2.26. Define a combinator parser `pMir` that transforms a concrete representation of a mirror-palindrome into an abstract one. Test your function with the concrete mirror-palindromes `cMir1` and `cMir2`. ◁

Exercise 3.48 ▷ Consider your answer to exercise 2.28. Assuming the comma is an associative operator, we can give the following abstract syntax for bit-lists:

```
data BitList = SingleB Bit | ConsB Bit BitList
```

Define a combinator parser `pBitList` that transforms a concrete representation of a bit-list into an abstract one. Test your function with the concrete bit-lists `cBitList1` and `cBitList2`. ◁

Exercise 3.49 ▷ Define a parser for fixed-point numbers, that is numbers like 12.34 and −123.456. Also integers are acceptable. Notice that the part following the decimal point looks like an integer, but has a different semantics! ◁

Exercise 3.50 ▷ Define a parser for floating point numbers, which are fixed point numbers followed by an optional E and an (positive or negative, integer) exponent. ◁

Exercise 3.51 ▷ Define a parser for Java assignments that consist of a variable, an = sign, an expression and a semicolon. ◁

Chapter 4

Grammar and Parser design

The previous chapters have introduced many concepts related to grammars and parsers. The goal of this chapter is to review these concepts, and to show how they are used in the design of grammars and parsers.

The design of a grammar and parser for a language consists of several steps: you have to

1. give a grammar for the language for which you want to have a parser;
2. analyse this grammar to find out whether or not it has some desirable properties;
3. possibly transform the grammar to obtain some of these desirable properties;
4. decide on the type of the parser: `Parser a b`, that is, decide on both the input type `a` of the parser (which may be the result type of a scanner), and the result type `b` of the parser.
5. construct a basic parser;
6. add semantic functions;
7. check whether or not you have obtained what you expected.

We will describe and exemplify each of these steps in detail in the rest of this section.

As a running example we will construct a grammar and parser for travelling schemes for day trips, of the following form:

```
Groningen 8:37 9:44 Zwolle 9:49 10:15 Utrecht 10:21 11:05 Den Haag
```

We might want to do several things with such a schema, for example:

1. compute the net travel time, i.e. the travel time minus the waiting time (2 hours and 17 minutes in the above example);
2. compute the total time one has to wait on the intermediate stations (11 minutes).

This chapter defines functions to perform these computations.

4.1 Step 1: A grammar for the language

The starting point for designing a parser for your language is to define a grammar that describes the language as precisely as possible. It is important to convince yourself from the fact that the grammar you give really generates the desired language, since the grammar will be the basis for grammar transformations, which might turn the grammar into a set of incomprehensible productions.

For the language of travelling schemes, we can give several grammars. The following grammar focuses on the fact that a trip consists of zero or more departures and arrivals.

$$\begin{array}{rcl} \mathit{TS} & \rightarrow & \mathit{TS}\ \mathit{Departure}\ \mathit{Arrival}\ \mathit{TS} \mid \mathit{Station} \\ \mathit{Station} & \rightarrow & \mathit{Identifier} \\ \mathit{Departure} & \rightarrow & \mathit{Time} \\ \mathit{Arrival} & \rightarrow & \mathit{Time} \\ \mathit{Time} & \rightarrow & \mathit{Nat} : \mathit{Nat} \end{array}$$

where *Identifier* and *Nat* have been defined in Section 2.3.1. So a travelling scheme is a sequence of departure and arrival times, separated by stations. Note that a single station is also a travelling scheme with this grammar.

Another grammar focuses on changing at a station:

$$\begin{array}{rcl} \mathit{TS} & \rightarrow & \mathit{Station}\ \mathit{Departure}\ (\mathit{Arrival}\ \mathit{Station}\ \mathit{Departure})^*\ \mathit{Arrival}\ \mathit{Station} \\ & \mid & \mathit{Station} \end{array}$$

So each travelling scheme starts and ends at a station, and in between there is a list of intermediate stations.

4.2 Step 2: Analysing the grammar

To parse sentences of a language efficiently, we want to have a nonambiguous grammar that is left-factored and not left recursive. Depending on the parser we want to obtain, we might desire other properties of our grammar. So a first step in designing a parser is analysing the grammar, and determining which properties are (not) satisfied. We have not yet developed tools for grammar analysis (we will do so in the chapter on *LL(1)* parsing) but for some grammars it is easy to detect some properties.

The first example grammar is left and right recursive: the first production for *TS* starts and ends with *TS*. Furthermore, the sequence *Departure Arrival* is an associative separator in the generated language.

These properties may be used for transforming the grammar. Since we don't mind about right recursion, we will not make use of the fact that the grammar is right recursive. The other properties will be used in grammar transformations in the following subsection.

4.3 Step 3: Transforming the grammar

Since the sequence *Departure Arrival* is an associative separator in the generated language, the productions for *TS* may be transformed into:

$$TS \rightarrow Station \mid Station\ Departure\ Arrival\ TS \tag{4.1}$$

Thus we have removed the left recursion in the grammar. Both productions for *TS* start with the nonterminal *Station*, so *TS* can be left factored. The resulting productions are:

$$\begin{aligned} TS &\rightarrow Station\ Z \\ Z &\rightarrow \epsilon \mid Departure\ Arrival\ TS \end{aligned}$$

We can also apply equivalence (2.1) to the two productions for *TS* from (4.1), and obtain the following single production:

$$TS \rightarrow (Station\ Departure\ Arrival)^*\ Station \tag{4.2}$$

So which productions do we take for *TS*? This depends on what we want to do with the parsed sentences. We will show several choices in the next section.

4.4 Step 4: Deciding on the types

We want to write a parser for travel schemes, that is, we want to write a function `ts` of type

```
ts :: Parser ? ?
```

The question marks should be replaced by the input type and the result type, respectively. For the input type we can choose between at least two possibilities: characters, `Char` or tokens `Token`. The type of tokens can be chosen as follows:

```
data Token  =  Station_Token Station | Time_Token Time

type Station  =  String
type Time     =  (Int,Int)
```

We will construct a parser for both input types in the next subsection. So `ts` has one of the following two types.

```
ts :: Parser Char  ?
ts :: Parser Token ?
```

For the result type we have many choices. If we just want to compute the total travelling time, `Int` suffices for the result type. If we want to compute the total travelling time, the total waiting time, and a nicely printed version of the travelling scheme, we may do several things:

- define three parsers, with `Int` (total travelling time), `Int` (total waiting time), and `String` (nicely printed version) as result type, respectively;
- define a single parser with the triple `(Int,Int,String)` as result type;
- define an abstract syntax for travelling schemes, say a datatype `TS`, and define three functions on `TS` that compute the desired results.

The first alternative parses the input three times, and is rather inefficient compared with the other alternatives. The second alternative is hard to extend if we want to compute something extra, but in some cases it might be more efficient than the third alternative. The third alternative needs an abstract syntax. There are several ways to define an abstract syntax for travelling schemes. The first abstract syntax corresponds to definition (4.1) of grammar *TS*.

```
data TS1  =  Single1 Station
          |  Cons1 Station Time Time TS1
```

where `Station` and `Time` are defined above. A second abstract syntax corresponds to the grammar for travelling schemes defined in (4.2).

```
type TS2  =  ([(Station,Time,Time)],Station)
```

So a travelling scheme is a tuple, the first component of which is a list of triples consisting of a departure station, a departure time, and an arrival time, and the second component of which is the final arrival station. A third abstract syntax corresponds to the second grammar defined in Section 4.1:

```
data TS3 = Single3 Station
         | Cons3 (Station,Time,[(Time,Station,Time)],Time,Station)
```

Which abstract syntax should we take? Again, this depends on what we want to do with the abstract syntax. Since `TS2` and `TS1` combine departure and arrival times in a tuple, they are convenient to use when computing travelling times. `TS3` is useful when we want to compute waiting times since it combines arrival and departure times in one constructor. Often we want to exactly mimic the productions of the grammar in the abstract syntax, so if we use (4.1) for the grammar for travelling schemes, we use `TS1` for the abstract syntax. Note that `TS1` is a datatype, whereas `TS2` is a type. `TS1` cannot be defined as a type because of the two alternative productions for *TS*. `TS2` can be defined as a datatype by adding a constructor. Types and datatypes each have their advantages and disadvantages; the application determines which to use. The result type of the parsing function `ts` may be one of types mentioned earlier (`Int`, etc.), or one of `TS1`, `TS2`, `TS3`.

4.5 Step 5: Constructing the basic parser

Converting a grammar to a parser is a mechanical process that consists of a set of simple replacement rules. Functional programming languages offer some

extra flexibility that we sometimes use, but usually writing a parser is a simple translation. We use the following replacement rules.

→	=
|	`<|>`
(space)	`<*>`
+	`many1`
*	`many`
?	`option`
terminal x	`symbol x`
begin of sequence of symbols	`undefined<$>`

Note that we start each sequence of symbols by `undefined<$>`. The `undefined` has to be replaced by an appropriate semantic function in Step 6, but putting `undefined` here ensures type correctness of the parser. Of course, running the parser will result in an error.

We construct a basic parser for each of the input types `Char` and `Token`.

4.5.1 Basic parsers from strings

Applying these rules to the grammar (4.2) for travelling schemes, we obtain the following basic parser.

```
station  :: Parser Char Station
station  =  undefined <$> identifier

time  :: Parser Char Time
time  =  undefined <$> natural <*> symbol ':' <*> natural

departure, arrival :: Parser Char Time
departure  =  undefined <$> time
arrival    =  undefined <$> time

tsstring  :: Parser Char ?
tsstring  =  undefined <$>
             many (undefined <$>
                   spaces
                   <*> station
                   <*> spaces
                   <*> departure
                   <*> spaces
                   <*> arrival
                  )
          <*> spaces
          <*> station

spaces  :: Parser Char String
spaces  =  undefined <$> many (symbol ' ')
```

The only thing left to do is to add the semantic glue to the functions. The semantic glue also determines the type of the function `tsstring`, which is denoted by `?` for the moment. For the other basic parsers we have chosen some reasonable return types. The semantic functions are defined in the next and final step.

4.5.2 A basic parser from tokens

To obtain a basic parser from tokens, we first write a scanner that produces a list of tokens from a string.

```
scanner  :: String -> [Token]
scanner  =  map mkToken . words

mkToken     :: String -> Token
mkToken xs  =  if isDigit (head xs)
               then Time_Token (mkTime xs)
               else Station_Token (mkStation xs)

parse_result :: [(a,b)] -> a
parse_result xs
  | null xs    =  error "parse_result: could not parse the input"
  | otherwise  =  fst (head xs)

mkTime  :: String -> Time
mkTime  =  parse_result . time

mkStation  :: String -> Station
mkStation  =  parse_result . station
```

This is a basic scanner with very basic error messages, but it suffices for now. The composition of the scanner with the function `tstoken1` defined below gives the final parser.

```
tstoken1  :: Parser Token ?
tstoken1  =  undefined <$>
             many (undefined <$>
                   tstation
                   <*> tdeparture
                   <*> tarrival
                  )
          <*> tstation

tstation :: Parser Token Station
tstation (Station_Token s:xs)  =  [(s,xs)]
tstation _                     =  []

tdeparture, tarrival :: Parser Token Time
```

```
tdeparture (Time_Token (h,m):xs)  =  [((h,m),xs)]
tdeparture _                      =  []

tarrival (Time_Token (h,m):xs)  =  [((h,m),xs)]
tarrival _                      =  []
```

where again the semantic functions remain to be defined. Note that functions `tdeparture` and `tarrival` are the same functions. Their presence reflects their presence in the grammar.

Another basic parser from tokens is based on the second grammar of Section 4.1.

```
tstoken2  :: Parser Token Int
tstoken2  =  undefined <$>
             tstation
             <*> tdeparture
             <*> many (undefined <$>
                       tarrival
                       <*> tstation
                       <*> tdeparture
                      )
             <*> tarrival
             <*> tstation
          <|> undefined <$> tstation
```

4.6 Step 6: Adding semantic functions

Once we have the basic parsing functions, we need to add the semantic glue: the functions that take the results of the elements in the right hand side of a production, and convert them into the result of the left hand side. The basic rule is: Let the types do the work!

First we add semantic functions to the basic parsing functions `station`, `time`, `departure`, `arrival`, and `spaces`. Since function `identifier` already returns a string, we can take the identity function `id` for `undefined` in function `station`. Since `id <$>` is the identity function, it can be omitted. To obtain a value of type `Time` from an integer, a character, and an integer, we have to combine the two integers in a tuple. So we take the following function

```
\x y z -> (x,z)
```

for `undefined` in `time`. Now, since function `time` returns a value of type `Time`, we can take the identity function for `undefined` in `departure` and `arrival`, and then we replace `id <$> time` by just `time`. Finally, the result of `many` is a string, so for `undefined` in `spaces` we can take the identity function too.

The first semantic function for the basic parser `tsstring` defined in Section 4.5.1 returns an abstract syntax tree of type `TS2`. So the first `undefined` in

`tsstring` should return a tuple of a list of things of the correct type (the first component of the type `TS2`) and a `Station`. Since `many` returns a list of things, we can construct such a tuple by means of the function

```
\x y z -> (x,z)
```

provided `many` returns a value of the desired type: `[(Station,Time,Time)]`. Note that this semantic function basically only throws away the value returned by the `spaces` parser: we are not interested in the spaces between the components of our travelling scheme. The `many` parser returns a value of the correct type if we replace the second occurrence of `undefined` in `tsstring` by the function

```
\u v w x y z -> (v,x,z)
```

Again, the results of `spaces` are thrown away. This completes a parser for travelling schemes.

In summary, the parser is defined as follows:

```
station  :: Parser Char Station
station  =  identifier

time  :: Parser Char Time
time  =  (\x y z -> (x,z)) <$> natural <*> symbol ':' <*> natural

departure, arrival :: Parser Char Time
departure  =  time
arrival    =  time

tsstring  :: Parser Char TS2
tsstring  =  (\x y z -> (x,z)) <$>
             many ((\u v w x y z -> (v,x,z)) <$>
                   spaces
                   <*> station
                   <*> spaces
                   <*> departure
                   <*> spaces
                   <*> arrival
                  )
             <*> spaces
             <*> station

spaces  :: Parser Char String
spaces  =  many (symbol ' ')
```

The next semantic functions we define compute the net travel time. To compute the net travel time, we have to compute the travel time of each trip from a

station to a station, and to add the travel times of all of these trips. We obtain the travel time of a single trip if we replace the second occurrence of `undefined` in `tsstring` by:

```
\u v w (xh,xm) y (zh,zm) -> (zh-xh)*60 + zm-xm
```

and Haskell's prelude function `sum` sums these times, so for the first occurrence of `undefined` we take:

```
\x y z -> sum x
```

This gives the following function tsstring':

```
tsstring' :: Parser Char Int
tsstring'  = (\x y z -> sum x) <$>
             many ((\u v w (xh,xm) y (zh,zm)
                           -> (zh - xh) * 60 + zm - xm)
                   <$> spaces
                   <*> station
                   <*> spaces
                   <*> departure
                   <*> spaces
                   <*> arrival
                  )
             <*> spaces
             <*> station
```

The final set of semantic functions we define are used for computing the total waiting time. Since the second grammar of Section 4.1 combines arrival times and departure times, we use a parser based on this grammar: the basic parser `tstoken2`. We have to give definitions of the three `undefined` semantic functions. If a trip consists of a single station, there is now waiting time, so the last occurrence of `undefined` is the function `const 0`. The second occurrence of function `undefined` computes the waiting time for one intermediate station:

```
\(uh,um) v (wh,wm) -> (wh-uh)*60 + wm-um
```

Finally, the first occurrence of `undefined` sums the list of waiting time obtained by means of the function that replaces the second occurrence of `undefined`:

```
\s t x y z -> sum x
```

The definition of the parser `tstoken2` for computing the total waiting time is:

```
tstoken2  :: Parser Token Int
tstoken2  =  (\s t x y z -> sum x) <$>
             tstation
             <*> tdeparture
```

```
<*> many ((\(uh,um) v (wh,wm)
               -> (wh - uh) * 60 + wm - um)
          <$> tarrival
          <*> tstation
          <*> tdeparture
         )
<*> tarrival
<*> tstation
<|> (const 0) <$> tstation
```

Exercise 4.1 ▷ Define the semantic functions for computing the net travel time with the parser `tstoken1`. What is the type of the parser? ◁

4.7 Step 7: Did you get what you expected

In the last step you test your parser(s) to see whether or not you have obtained what you expected, and whether or not you have made errors in the above process.

Exercise 4.2 ▷ Test the parsers `tsstring`, `tsstring'`, `tstoken1` and `tstoken2` with the scheme given in the introduction. What is your conclusion? ◁

SUMMARY

This chapter describes the different steps that have to be considered in the design of a grammar and a language.

4.8 Exercises

Exercise 4.3 ▷ Write a parser `floatLiteral` for `Java` float-literals. The `EBNF` grammar for float-literals is given by:

$$
\begin{array}{rcl}
\mathit{Digits} & \rightarrow & \mathit{Digits}\ \mathit{Digit} \mid \mathit{Digit} \\
\mathit{FloatLiteral} & \rightarrow & \mathit{IntPart}\ .\ \mathit{FractPart}?\ \mathit{ExponentPart}?\ \mathit{FloatSuffix}? \\
 & \mid & .\ \mathit{FractPart}\ \mathit{ExponentPart}?\ \mathit{FloatSuffix}? \\
 & \mid & \mathit{IntPart}\ \mathit{ExponentPart}\ \mathit{FloatSuffix}? \\
 & \mid & \mathit{IntPart}\ \mathit{ExponentPart}?\ \mathit{FloatSuffix} \\
\mathit{IntPart} & \rightarrow & \mathit{SignedInteger} \\
\mathit{FractPart} & \rightarrow & \mathit{Digits} \\
\mathit{ExponentPart} & \rightarrow & \mathit{ExponentIndicator}\ \mathit{SignedInteger}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{SignedInteger} & \rightarrow & \mathit{Sign}?\ \mathit{Digits} \\
\mathit{ExponentIndicator} & \rightarrow & \mathsf{e} \mid \mathsf{E} \\
\mathit{Sign} & \rightarrow & + \mid - \\
\mathit{FloatSuffix} & \rightarrow & \mathsf{f} \mid \mathsf{F} \mid \mathsf{d} \mid \mathsf{D}
\end{array}
$$

To keep your parser simple, assume that all nonterminals, except for the nonterminal *FloatLiteral*, are represented by a `String` in the abstract syntax. ◁

Exercise 4.4 ▷ Write an evaluator `signedFloat` for `Java` float-literals (the float-suffix may be ignored). ◁

Chapter 5

Regular Languages

INTRODUCTION

The first phase of a compiler takes an input program, and splits the input into a list of terminal symbols: keywords, identifiers, numbers, punctuation, etc. Regular expressions are used for the description of the terminal symbols. A regular grammar is a particular kind of context-free grammar that can be used to describe regular expressions. Finite-state automata can be used to recognise sentences of regular grammars. This chapter discusses all of these concepts, and is organised as follows. Section 5.1 introduces finite-state automata. Finite-state automata appear in two versions, nondeterministic and deterministic ones. Section 5.1.4 shows that a nondeterministic finite-state automaton can be transformed into a deterministic finite-state automaton, so you don't have to worry about whether or not your automaton is deterministic. Section 5.2 introduces regular grammars (context-free grammars of a particular form), and regular languages. Furthermore, it shows their equivalence with finite-state automata. Section 5.3 introduces regular expressions as finite descriptions of regular languages and shows that such expressions are another, equivalent, tool for regular languages. Finally, Section 5.4 gives some of the proofs of the results of the previous sections.

GOALS

After you have studied this chapter you will know that

- regular languages are a subset of context-free languages;
- it is not always possible to give a regular grammar for a context-free grammar;
- regular grammars, finite-state automata and regular expressions are three different tools for regular languages;
- regular grammars, finite-state automata and regular expressions have the same expressive power;
- finite-state automata appear in two, equally expressive, versions: deterministic and nondeterministic.

5.1 Finite-state automata

The classical approach to recognising sentences from a regular language uses finite-state automata. A finite-state automaton can be viewed as a simple form of digital computer with only a finite number of states, no temporary storage, an input file but only the possibility to read it, and a control unit which records the state transitions. A rather limited medium, but a useful tool in many practical subproblems. A finite-state automaton can easily be implemented by a function that takes time linear in the length of its input, and constant space. This implies that problems that can be solved by means of a finite-state automaton, can be implemented by means of very efficient programs.

5.1.1 Deterministic finite-state automata

Finite-state automata come in two flavours: deterministic and nondeterministic. We start with a description of deterministic finite-state automata, the simplest form of automata.

Definition 1: Deterministic finite-state automaton, DFA

deterministic finite-state automaton

A *deterministic finite-state automaton* (DFA) is a 5-tuple (X, Q, d, S, F) where

- X is the input alphabet,
- Q is a finite set of states,
- $d :: Q \to X \to Q$ is the state transition function,
- $S \in Q$ is the start state,
- $F \subseteq Q$ is the set of accepting states.

□

As an example, consider the DFA $M_0 = (X, Q, d, S, F)$ with

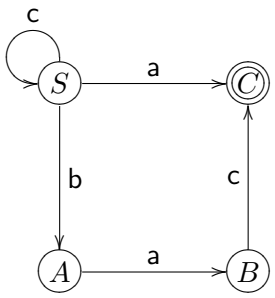
$$\begin{aligned} X &= \{\mathsf{a}, \mathsf{b}, \mathsf{c}\} \\ Q &= \{S, A, B, C\} \\ F &= \{C\} \end{aligned}$$

where state transition function d is defined by

$$\begin{aligned} d\ S\ \mathsf{a} &= C \\ d\ S\ \mathsf{b} &= A \\ d\ S\ \mathsf{c} &= S \\ d\ A\ \mathsf{a} &= B \\ d\ B\ \mathsf{c} &= C \end{aligned}$$

For human beings, a finite-state automaton is more comprehensible in a graphical representation. The following representation is customary: states are depicted as the nodes in a graph; accepting states get a double circle; start states are explicitly mentioned or indicated otherwise. The transition function is represented by the edges: whenever $d\ Q_i\ \mathsf{x}$ is a state Q_j, then there is an arrow

labelled x from Q_i to Q_j. The input alphabet is implicit in the labels. For automaton M_0 above, the pictorial representation is:



Note that d is a partial function: for example $d\ B$ a is not defined. We can make d into a total function by introducing a new 'sink' state, the result state of all undefined transitions. For example, in the above automaton we can introduce a sink state D with $d\ D\ x = D$ for all terminals x, and $d\ E\ x = D$ for all states E and terminals x for which $d\ E\ x$ is undefined. The sink state and the transitions from/to it are almost always omitted.

The action of a DFA on an input string is described as follows: given a sequence w of input symbols, w can be 'processed' symbol by symbol (from left to right) and — depending on the specific input symbol — the DFA (initially in the start state) moves to the state as determined by its state transition function. If no move is possible, the automaton blocks. When the complete input has been processed and the DFA is in one of its accepting states, then we say that w is *accepted by the automaton.* accept

To illustrate the action of an DFA, we will show that the sentence bac is accepted by M_0. We do so by recording the successive configurations, i.e. the pairs of current state and remaining input values.

$$
\begin{array}{l}
\quad (S, \mathsf{bac}) \\
\mapsto \\
\quad (A, \mathsf{ac}) \\
\mapsto \\
\quad (B, \mathsf{c}) \\
\mapsto \\
\quad (C, \epsilon)
\end{array}
$$

Because of the deterministic behaviour of a DFA the definition of acceptance by a DFA is relatively easy. Informally, a sequence $w \in X^*$ is accepted by a DFA (X, Q, d, S, F), if it is possible, when starting the DFA in S, to end in an accepting state after processing w. This operational description of acceptance is formalised in the predicate *dfa_accept*. The predicate will be derived in a top-down fashion, i.e. we formulate the predicate in terms of ("smaller") subcomponents and afterwards we give solutions to the subcomponents.

Suppose *dfa* is a function that reflects the behaviour of the DFA, i.e. a function which given a transition function, a start state and a string, returns the unique state that is reached after processing the string starting from the start state.

Then the predicate *dfa_accept* is defined by:

$$\begin{array}{rcl} \mathit{dfa\_accept} & :: & X^* \to (Q \to X \to Q, Q, \{Q\}) \to \mathit{Bool} \\ \mathit{dfa\_accept}\ w\ (d, S, F) & = & (\mathit{dfa}\ d\ S\ w) \in F \end{array}$$

It remains to construct a definition of function *dfa* that takes a transition function, a start state, and a list of input symbols, and reflects the behaviour of a DFA. The definition of *dfa* is straightforward

$$\begin{array}{lcl} \mathit{dfa} & :: & (Q \to X \to Q) \to Q \to X^* \to Q \\ \mathit{dfa}\ d\ q\ \epsilon & = & q \\ \mathit{dfa}\ d\ q\ (\mathsf{a}x) & = & \mathit{dfa}\ d\ (d\ q\ \mathsf{a})\ x \end{array}$$

Note that both the type and the definition of *dfa* match the pattern of the function *foldl*, and it follows that the function *dfa* is actually identical to *foldl*.

$$\mathit{dfa}\ d\ q \quad = \quad \mathit{foldl}\ d\ q.$$

Definition 2: Acceptance by a DFA
The sequence $w \in X^*$ is accepted by DFA (X, Q, d, S, F) if

$$\mathit{dfa\_accept}\ w\ (d, S, F)$$

where

$$\begin{array}{lcl} \mathit{dfa\_accept}\ w\ (d, qs, fs) & = & \mathit{dfa}\ d\ qs\ w \in fs \\ \mathit{dfa}\ d\ qs & = & \mathit{foldl}\ d\ qs \end{array}$$

□
Using the predicate *dfa_accept*, the language of a DFA is defined as follows.

Definition 3: Language of a DFA
For DFA $M = (X, Q, d, S, F)$, the language of M, $Ldfa(M)$, is defined by

$$\mathit{Ldfa}(M) \quad = \quad \{w \in X^* \mid \ \mathit{dfa\_accept}\ w\ (d, S, F)\}$$

□

5.1.2 Nondeterministic finite-state automata

This subsection introduces nondeterministic finite-state automata and defines their semantics, i.e. the language of a nondeterministic finite-state automaton.

The transition function of a DFA returns a state, which implies that for all terminal symbols x and for all states t there can only be one edge starting in t labelled with x. Sometimes it is convenient to have two or more edges labelled with the same terminal symbol from a state. In these cases one can

use a *nondeterministic finite-state automaton*. Nondeterministic finite state automata are defined as follows.

Definition 4: Nondeterministic finite-state automaton, NFA

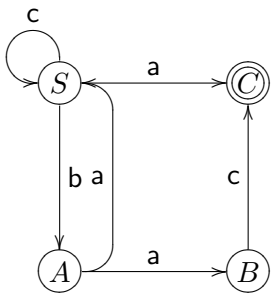
nondeterministic finite-state automaton

A nondeterministic finite-state automaton (NFA) is a 5-tuple (X, Q, d, Q_0, F), where

- X is the input alphabet,
- Q is a finite set of states,
- $d :: Q \to X \to \{Q\}$ is the state transition function,
- $Q_0 \subseteq Q$ is the set of start states,
- $F \subseteq Q$ is the set of accepting states.

□

An NFA differs from a DFA in that there may be more than one start state and that there may be more than one possible move for each state and input symbol. Here is an example of an NFA:



Note that this NFA is very similar to the DFA in the previous section: the only difference is that there are two outgoing arrows labelled with a from state A. Thus the DFA becomes an NFA.

Formally, this NFA is defined as $M_1 = (X, Q, d, Q_0, F)$ with

$$\begin{aligned} X &= \{\mathsf{a}, \mathsf{b}, \mathsf{c}\} \\ Q &= \{S, A, B, C\} \\ Q_0 &= \{S\} \\ F &= \{C\} \end{aligned}$$

where state transition function d is defined by

$$\begin{aligned} d\ S\ \mathsf{a} &= \{C\} \\ d\ S\ \mathsf{b} &= \{A\} \\ d\ S\ \mathsf{c} &= \{S\} \\ d\ A\ \mathsf{a} &= \{S, B\} \\ d\ B\ \mathsf{c} &= \{C\} \end{aligned}$$

Again d is a partial function, which can be made total by adding $d\ D\ x = \{\ \}$ for all states D and all terminal symbols x for which $d\ D\ x$ is undefined.

Since an NFA can make an arbitrary (nondeterministic) choice for one of its possible moves, we have to be careful in defining what it means that a sequence is accepted by an NFA. Informally, sequence $w \in X^*$ is accepted by NFA (X, Q, d, Q_0, F), if it is possible, when starting the NFA in a state from Q_0, to end in an accepting state after processing w. This operational description of acceptance is formalised in the predicate *nfa_accept*.

Assume that we have a function, say *nfa*, which reflects the behaviour of the NFA. That is a function which given a transition function, a set of start states and a string, returns all possible states that can be reached after processing the string starting in some start state. Then the predicate *nfa_accept* can be expressed as

$$\begin{array}{lcl} \mathit{nfa\_accept} & :: & X^* \to (Q \to X \to \{Q\}, \{Q\}, \{Q\}) \to \mathit{Bool} \\ \mathit{nfa\_accept}\ w\ (d, Q_0, F) & = & \mathit{nfa}\ d\ Q_0\ w \cap F \neq \emptyset \end{array}$$

Now it remains to find a function *nfa d qs* of type $X^* \to \{Q\}$ that reflects the behaviour of the NFA. For lists of length 1 such a function, called *deltas*, is defined by

$$\begin{array}{lcl} \mathit{deltas} & :: & (Q \to X \to \{Q\}) \to \{Q\} \to X \to \{Q\} \\ \mathit{deltas}\ d\ qs\ \mathsf{a} & = & \{r \mid q \in qs, r \in d\ q\ \mathsf{a}\} \end{array}$$

The behaviour of the NFA on X-sequences of arbitrary length follows from this "one step" behaviour:

$$\begin{array}{lcl} \mathit{nfa} & :: & (Q \to X \to \{Q\}) \to \{Q\} \to X^* \to \{Q\} \\ \mathit{nfa}\ d\ qs\ \epsilon & = & qs \\ \mathit{nfa}\ d\ qs\ (\mathsf{a}x) & = & \mathit{nfa}\ d\ (\mathit{deltas}\ d\ qs\ \mathsf{a})\ x \end{array}$$

Again, it follows that *nfa* can be written as a *foldl*.

$$\mathit{nfa}\ d\ qs \quad = \quad \mathit{foldl}\ (\mathit{deltas}\ d)\ qs$$

This concludes the definition of predicate *nfa_accept*. In summary we have derived

Definition 5: Acceptance by an NFA
The sequence $w \in X^*$ is accepted by NFA (X, Q, d, Q_0, F) if

$$\mathit{nfa\_accept}\ w\ (d, Q_0, F)$$

where

$$\begin{array}{lcl} \mathit{nfa\_accept}\ w\ (d, qs, fs) & = & \mathit{nfa}\ d\ qs\ w \cap fs \neq \emptyset \\ \mathit{nfa}\ d\ qs & = & \mathit{foldl}\ (\mathit{deltas}\ d)\ qs \\ \mathit{deltas}\ d\ qs\ \mathsf{a} & = & \{r \mid q \in qs, r \in d\ q\ \mathsf{a}\} \end{array}$$

□

Using the *nfa_accept*-predicate, the language of an NFA is defined by

Definition 6: Language of an NFA
For NFA $M = (X, Q, d, Q_0, F)$, the language of M, $Lnfa(M)$, is defined by

$$Lnfa(M) \quad = \quad \{w \in X^* \mid nfa\_accept\ w\ (d, Q_0, F)\}$$

□

Note that it is computationally expensive to determine whether or not a list is an element of the language of a nondeterministic finite-state automaton. This is due to the fact that all possible transitions have to be tried in order to determine whether or not the automaton can end in an accepting state after reading the input. Determining whether or not a list is an element of the language of a deterministic finite-state automaton can be done in time linear in the length of the input list, so from a computational view, deterministic finite-state automata are preferable. Fortunately, for each nondeterministic finite-state automaton there exists a deterministic finite-state automaton that accepts the same language. We will show how to construct a DFA from an NFA in subsection 5.1.4.

5.1.3 Implementation

This section describes how to implement finite state machines. We start with implementing DFA's. Given a DFA M = (X, Q, d, S, F), we define two datatypes:

```
data StateM   =  ...  deriving Eq
data SymbolM  =  ...
```

where the states of M (the elements of the set Q) are listed as constructors of `StateM`, and the symbols of M (the elements of the set X) are listed as constructors of `SymbolM`. Furthermore, we define three values (one of which a function):

```
start   ::  StateM
delta   ::  SymbolM -> StateM -> StateM
finals  ::  [StateM]
```

Note that the first two arguments of `delta` have changed places: this has been done in order to be able to apply 'partial evaluation' later. The extended transition function `dfa` and the accept function `dfaAccept` are now defined by:

```
dfa  :: [SymbolM] -> StateM
dfa  =  foldl (flip delta) start

dfaAccept     :: [SymbolM] -> Bool
dfaAccept xs  =  elem (dfa xs) finals
```

Given a list of symbols `[x1,x2,...,xn]`, the computation of `dfa [x1,x2,...,xn]` uses the following intermediate states:

```
start, delta x1 start, delta x2 (delta x1 start),...
```

This list of states is determined uniquely by the input `[x1,x2,...,xn]` and the start state.

Since we want to use the same function names for different automata, we introduce the following class:

```
class  Eq a => DFA a b  where
   start   ::  a
   delta   ::  b -> a -> a
   finals  ::  [a]

   dfa  :: [b] -> a
   dfa  =  foldl (flip delta) start

   dfaAccept     :: [b] -> Bool
   dfaAccept xs  =  elem (dfa xs) finals
```

Note that the functions `dfa` and `dfaAccept` are defined once and for all for all instances of the class `DFA`.

As an example, we give the implementation of the example DFA (called `MEX` here) given in the previous subsection.

```
data StateMEX   =  A | B | C | S  deriving Eq
data SymbolMEX  =  SA | SB | SC
```

So the state A is represented by `A`, and the symbol `a` is represented by `SA`, and similar for the other states and symbols. The automaton is made an instance of class `DFA` as follows:

```
instance  DFA StateMEX SymbolMEX  where
  start  =  S

  delta x  S  =  case x of SA -> C
                           SB -> A
                           SC -> S
  delta SA A  =  B
  delta SC B  =  C

  finals  =  [C]
```

partial evaluation

We can improve the performance of the automaton (function) `dfa` by means of *partial evaluation*. The main idea of partial evaluation is to replace computations that are performed often at run-time by a single computation that is performed only once at compile-time. A very simple example is the replacement of the expression `if True then f1 else f2` by the expression `f1`. Partial evaluation applies to finite automatons in the following way.

```
  dfa [x1,x2,...,xn]
=
```

```
   foldl (flip delta) start [x1,x2,...,xn]
 =
   foldl (flip delta) (delta x1 start) [x2,...,xn]
 =
   case x1 of
      SA -> foldl (flip delta) (delta SA start) [x2,...,xn]
      SB -> foldl (flip delta) (delta SB start) [x2,...,xn]
      SC -> foldl (flip delta) (delta SC start) [x2,...,xn]
```

All these equalities are simple transformation steps for functional programs. Note that the first argument of `foldl` is always `flip delta`, and the second argument is one of the four states `S`, `A`, `B`, or `C` (the result of `delta`). Since there are only a finite number of states (four, to be precise), we can define a transition function for each state:

```
   dfaS, dfaA, dfaB, dfaC  ::  [Symbol] -> State
```

Each of these functions is a case expression over the possible input symbols.

```
    dfaS []      =  S
    dfaS (x:xs)  =  case x of SA -> dfaC xs
                              SB -> dfaA xs
                              SC -> dfaS xs

    dfaA []      =  A
    dfaA (x:xs)  =  case x of SA -> dfaB xs

    dfaB []      =  B
    dfaB (x:xs)  =  case x of SC -> dfaC xs

    dfaC []      =  C
```

With this definition of the finite automaton, the number of steps required for computing the value of `dfaS xs` for some list of symbols `xs` is reduced considerably.

The implementation of NFA's is similar to the implementation of DFA's. The only difference is that the transition and accept functions have to take care of sets (lists) of states now. We will use the following class, in which we use some names that also appear in the class `DFA`. This is a problem if the two classes appear in the same module.

```
   class  Eq a => NFA a b  where
      start   ::  [a]
      delta   ::  b -> a -> [a]
      finals  ::  [a]

      nfa     :: [b] -> [a]
      nfa     =  foldl (flip deltas) start
```

```
deltas   :: b -> [a] -> [a]
deltas a =  union . map (delta a)

nfaAccept     :: [b] -> Bool
nfaAccept xs  =  intersect (nfa xs) finals /= []
```

Here, functions `union` and `intersect` are implemented as follows:

```
union  :: Eq a => [[a]] -> [a]
union  =  nub . concat

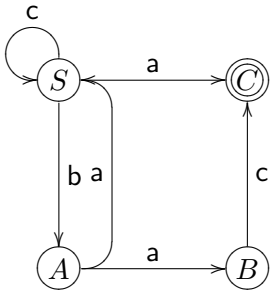
nub  :: Eq a => [a] -> [a]
nub  =  foldr (\x xs -> x:filter (/=x) xs) []

intersect        :: Eq a => [a] -> [a] -> [a]
intersect xs ys  =  intersect' (nub xs)
  where intersect' =
          foldr (\x xs -> if x `elem` ys then x:xs else xs) []
```

5.1.4 Constructing a DFA from an NFA

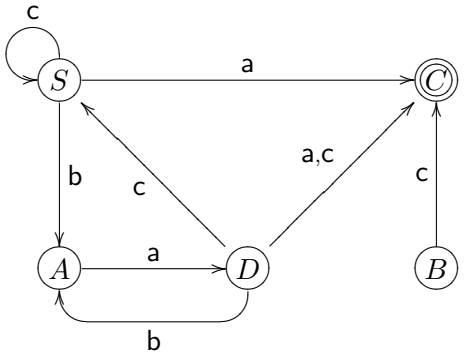
Is it possible to express more languages by means of nondeterministic finite-state automata than by deterministic finite-state automata? For each nondeterministic automaton it is possible to give a deterministic finite-state automaton such that both automata accept the same language, so the answer to the above question is no. Before we give the formal proof of this claim, we illustrate the construction of a DFA for an NFA in an example.

Consider the nondeterministic finite-state automaton corresponding with the example grammar of the previous subsection.

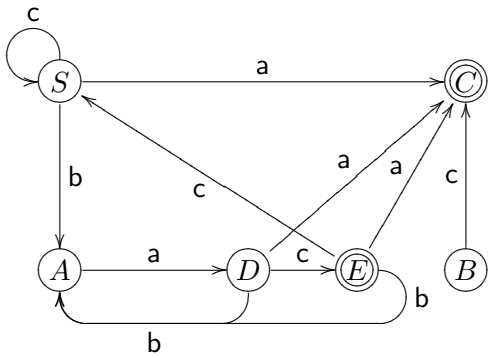


The nondeterminicity of this automaton appears in state A: two outgoing arcs of A are labelled with an `a`. Suppose we add a new state D, with an arc from A to D labelled `a`, and we remove the arcs labelled `a` from A to S and from A to B. Since D is a merge of the states S and B we have to merge the outgoing arcs

from S and B into outgoing arcs of D. We obtain the following automaton.



We omit the proof that the language of the latter automaton is equal to the language of the former one. Although there is just one outgoing arc from A labelled with a, this automaton is still nondeterministic: there are two outgoing arcs labelled with c from D. We apply the same procedure as above. Add a new state E and an arc labelled c from D to E, and remove the two outgoing arcs labelled c from D. Since E is a merge of the states C and S we have to merge the outgoing arcs from C and S into outgoing arcs of E. We obtain the following automaton.



Again, we do not prove that the language of this automaton is equal to the language of the previous automaton, provided we add the state E to the set of accepting states, which until now consisted just of state C. State E is added to the set of accepting states because it is the merge of a set of states among which at least one belongs to the set of accepting states. Note that in this automaton for each state, all outgoing arcs are labelled differently, i.e. this automaton is deterministic. The DFA constructed from the NFA above is the 5-tuple (X, Q, d, S, F) with

$$\begin{aligned} X &= \{\mathsf{a}, \mathsf{b}, \mathsf{c}\} \\ Q &= \{S, A, B, C, D, E\} \\ F &= \{C, E\} \end{aligned}$$

where transition function d is defined by

$$d\ S\ \mathsf{a} \;=\; C$$

$$
\begin{array}{rcl}
d\ S\ \mathsf{b} & = & A \\
d\ S\ \mathsf{c} & = & S \\
d\ A\ \mathsf{a} & = & D \\
d\ B\ \mathsf{c} & = & C \\
d\ D\ \mathsf{a} & = & C \\
d\ D\ \mathsf{b} & = & A \\
d\ D\ \mathsf{c} & = & E \\
d\ E\ \mathsf{a} & = & C \\
d\ E\ \mathsf{b} & = & A \\
d\ E\ \mathsf{c} & = & S
\end{array}
$$

This construction is called the 'subset construction'. In general, the construction works as follows. Suppose $M = (X, Q, d, Q_0, F)$ is a nondeterministic finite-state automaton. Then the finite-state automaton $M' = (X', Q', d', Q_0', F')$, the components of which are defined below, accepts the same language.

$$
\begin{array}{rcl}
X' & = & X \\
Q' & = & \mathit{subs}\ Q
\end{array}
$$

where *subs* returns all subsets of a set. $\mathit{subs}\ Q$ is also called the powerset of Q. For example,

$$
\begin{array}{rcl}
\mathit{subs} & :: & \{X\} \rightarrow \{\{X\}\} \\
\mathit{subs}\ \{A, B\} & = & \{\{\ \}, \{A\}, \{A, B\}, \{B\}\}
\end{array}
$$

For the other components of M' we define

$$
\begin{array}{rcl}
d'\ q\ \mathsf{a} & = & \{t \mid t \in d\ r\ \mathsf{a}, r \in q\} \\
Q_0' & = & \{Q_0\} \\
F' & = & \{p \mid p \cap F \neq \emptyset, p \in Q'\}
\end{array}
$$

The proof of the following theorem is given in Section 5.4.

Theorem 7: DFA for NFA
For every nondeterministic finite-state automaton M there exists a finite-state automaton M' such that

$$\mathit{Lnfa}(M) \quad = \quad \mathit{Ldfa}(M')$$

□

Theorem 7 enables us to freely switch between NFA's and DFA's. Equipped with this knowledge we continue the exploration of regular languages in the following section. But first we show that the transformation from an NFA to a DFA is an instance of partial evaluation.

5.1.5 Partial evaluation of NFA's

Given a nondeterministic finite state automaton we can obtain a deterministic finite state automaton not just by means of the above construction, but also by means of partial evaluation.

Just as for function `dfa`, we can calculate as follows with function `nfa`.

```
  nfa [x1,x2,...,xn]
=
  foldl (flip deltas) start [x1,x2,...,xn]
=
  foldl (flip deltas) (deltas x1 start) [x2,...,xn]
=
  case x1 of
     SA -> foldl (flip deltas) (deltas SA start) [x2,...,xn]
     SB -> foldl (flip deltas) (deltas SB start) [x2,...,xn]
     SC -> foldl (flip deltas) (deltas SC start) [x2,...,xn]
```

Note that the first argument of `foldl` is always `flip deltas`, and the second argument is one of the six sets of states `[S]`, `[A]`, `[B]`, `[C]`, `[B,S]`, `[C,S]` (the possible results of `deltas`). Since there are only a finite number of results of `deltas` (six, to be precise), we can define a transition function for each state:

```
nfaS, nfaA, nfaB, nfaC, nfaBS, nfaCS  ::  [Symbol] -> [State]
```

For example,

```
nfaA []      =  A
nfaA (x:xs)  =  case x of
                  SA  ->  nfaBS xs
                   _  ->  error "no transition possible"
```

Each of these functions is a case expression over the possible input symbols. By partially evaluating the function `nfa` we have obtained a function that is the implementation of the deterministic finite state automaton corresponding to the nondeterministic finite state automaton.

5.2 Regular grammars

This section defines regular grammars, a special kind of context-free grammars. Subsection 5.2.1 gives the correspondence between nondeterministic finite-state automata and regular grammars.

Definition 8: Regular Grammar

A *regular grammar* G is a context free grammar (T, N, R, S) in which all pro- regular grammar

duction rules in R are of one of the following two forms:

$$\begin{aligned} A &\rightarrow xB \\ A &\rightarrow x \end{aligned}$$

with $x \in T^*$ and $A, B \in N$. So in every rule there is at most one nonterminal, and if there is a nonterminal present, it occurs at the end. □

The regular grammars as defined here are sometimes called right-regular grammars. There is a symmetric definition for left-regular grammars.

Definition 9: Regular Language

regular language

A *regular language* is a language that is generated by a regular grammar. □

From the definition it is clear that each regular language is context-free. The question is now: Is each context-free language regular? The answer is: No. There are context-free languages that are not regular; an example of such a language is $\{\mathsf{a}^n\mathsf{b}^n \mid n \in \mathbb{N}\}$. To understand this, you have to know how to prove that a language is not regular. Because of its subtlety, we postpone this kind of proofs until Chapter 9. Here it suffices to know that regular languages form a proper subset of context-free languages and that we will profit from their speciality in the recognition process.

A first similarity between regular languages and context-free languages is that both are closed under union, concatenation and Kleene-star.

Theorem 10:

Let L and M be regular languages, then

$$\begin{aligned} L \cup M &\quad is \quad regular \\ LM &\quad is \quad regular \\ L^* &\quad is \quad regular \end{aligned}$$

□

Proof: Let $G_L = (T, N_L, R_L, S_L)$ and $G_M = (T, N_M, R_M, S_M)$ be regular grammars for L and M respectively, then

- for regular grammars, the well-known union construction for context-free grammars is a regular grammar again;
- we obtain a regular grammar for LM if we replace, in G_L, each production of the form $T \rightarrow x$ and $T \rightarrow \epsilon$ by $T \rightarrow xS_M$ and $T \rightarrow S_M$, respectively;
- since $L^* = \{\epsilon\} \cup LL^*$, it follows from the above that there exists a regular grammar for L^*.

□

In addition to these closure properties, regular languages are closed under intersection and complement too. See the exercises. This is remarkable because context-free languages are not closed under these operations. Recall the language $L = L_1 \cap L_2$ where $L_1 = \{\mathsf{a}^n\mathsf{b}^n\mathsf{c}^m \mid n, m \in \mathbb{N}\}$ and $L_2 = \{\mathsf{a}^n\mathsf{b}^m\mathsf{c}^m \mid n, m \in \mathbb{N}\}$.

As for context-free languages, there may exist more than one regular grammar for a given regular language and these regular grammars may be transformed into each other.

We conclude this section with a grammar transformation:

Theorem 11:

For each regular grammar G there exists a regular grammar G' with start-symbol S' such that

$$L(G) \quad = \quad L(G')$$

and such that G' has no productions of the form $U \rightarrow V$ and $W \rightarrow \epsilon$, with $W \neq S'$.
In other words: every regular grammar can be transformed to a form where every production has a nonempty terminal string in its righthandside (with a possible exception for $S \rightarrow \epsilon$). □

The proof of this transformation is omitted, we only briefly describe the construction of such a regular grammar, and illustrate the construction with an example.

Given a regular grammar G, a regular grammar with the same language but without productions of the form $U \rightarrow V$ and $W \rightarrow \epsilon$ for all U, V, and all $W \neq S$ is obtained as follows. First, consider all pairs Y, Z of nonterminals of G such that $Y \stackrel{*}{\Rightarrow} Z$. Add productions $Y \rightarrow z$ to the grammar, with $Z \rightarrow z$ a production of the original grammar, and z not a single nonterminal. Remove all productions $U \rightarrow V$ from G. Finally, remove all productions of the form $W \rightarrow \epsilon$ for $W \neq S$, and for each production $U \rightarrow \mathsf{x}W$ add the production $U \rightarrow \mathsf{x}$. The following example illustrates this construction.

Consider the following regular grammar G.

$$\begin{array}{lcl} S & \rightarrow & \mathsf{a}A \\ S & \rightarrow & \mathsf{b}B \\ S & \rightarrow & A \\ S & \rightarrow & C \\ A & \rightarrow & \mathsf{b}B \\ A & \rightarrow & S \\ A & \rightarrow & \epsilon \\ B & \rightarrow & \mathsf{b}B \\ B & \rightarrow & \epsilon \\ C & \rightarrow & \mathsf{c} \end{array}$$

The grammar G' of the desired form is constructed in 3 steps.

Step 1
Let G' equal G.

Step 2
Consider all pairs of nonterminals Y and Z. If $Y \stackrel{*}{\Rightarrow} Z$, add the productions $Y \to z$ to G', with $Z \to z$ a production of the original grammar, and z not a single nonterminal. Furthermore, remove all productions of the form $U \to V$ from G'. In the example we remove the productions $S \to A$, $S \to C$, $A \to S$, and we add the productions $S \to \mathsf{b}B$ and $S \to \epsilon$ since $S \stackrel{*}{\Rightarrow} A$, and the production $S \to \mathsf{c}$ since $S \stackrel{*}{\Rightarrow} C$, and the productions $A \to \mathsf{a}A$ and $A \to \mathsf{b}B$ since $A \stackrel{*}{\Rightarrow} S$, and the production $A \to \mathsf{c}$ since $A \stackrel{*}{\Rightarrow} C$. We obtain the grammar with the following productions.

$$
\begin{array}{lcl}
S & \to & \mathsf{a}A \\
S & \to & \mathsf{b}B \\
S & \to & \mathsf{c} \\
S & \to & \epsilon \\
A & \to & \mathsf{b}B \\
A & \to & \mathsf{a}A \\
A & \to & \mathsf{c} \\
A & \to & \epsilon \\
B & \to & \mathsf{b}B \\
B & \to & \epsilon \\
C & \to & \mathsf{c}
\end{array}
$$

This grammar generates the same language as G, and has no productions of the form $U \to V$. It remains to remove productions of the form $W \to \epsilon$ for $W \neq S$.

Step 3
Remove all productions of the form $W \to \epsilon$ for $W \neq S$, and for each production $U \to \mathsf{x}W$ add the production $U \to \mathsf{x}$. Applying this transformation to the above grammar gives the following grammar.

$$
\begin{array}{lcl}
S & \to & \mathsf{a}A \\
S & \to & \mathsf{b}B \\
S & \to & \mathsf{a} \\
S & \to & \mathsf{b} \\
S & \to & \mathsf{c} \\
S & \to & \epsilon \\
A & \to & \mathsf{b}B \\
A & \to & \mathsf{a}A \\
A & \to & \mathsf{a} \\
A & \to & \mathsf{b} \\
A & \to & \mathsf{c} \\
B & \to & \mathsf{b}B \\
B & \to & \mathsf{b}
\end{array}
$$

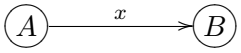
$$C \rightarrow \mathsf{c}$$

Each production in this grammar is of one of the desired forms: $U \rightarrow \mathsf{x}$ or $U \rightarrow \mathsf{x}V$, and the language of the grammar G' we thus obtain is equal to the language of grammar G.

5.2.1 Equivalence of Regular grammars and Finite automata

In the previous section, we introduced finite-state automata. Here we show that regular grammars and nondeterministic finite-state automata are two sides of one coin.

We will prove the equivalence using theorem 7. The equivalence consists of two parts, formulated in the theorems 12 and 13 below. The basis for both theorems is the direct correspondence between a production $A \rightarrow xB$ and a transition



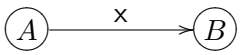
Theorem 12: Regular grammar for NFA
For each NFA M there exists a regular grammar G such that

$$Lnfa(M) = L(G)$$

□

Proof: We will just sketch the construction, the formal proof can be found in the literature. Let (X, Q, d, S, F) be a DFA for NFA M. Construct the grammar $G = (X, Q, R, S)$ where

- the terminals of the grammar are the input alphabet of the automaton;
- the nonterminals of the grammar are the states of the automaton;
- the start state of the grammar is the start state of the automaton;
- the productions of the grammar correspond to the automaton transitions: a rule $A \rightarrow \mathsf{x}B$ for each transition



a rule $A \rightarrow \epsilon$ for each terminal state A.
In formulae:

$$\begin{array}{rcll} R & = & \{A \rightarrow \mathsf{x}B & \mid A, B \in Q, \mathsf{x} \in X, d\ A\ \mathsf{x} = B\} \\ & \cup & \{A \rightarrow \epsilon & \mid A \in F\} \end{array}$$

□

Theorem 13: NFA for regular grammar
For each regular grammar G there exists a nondeterministic finite-state automaton M such that

$$L(G) = Lnfa(M)$$

□

Proof: Again, we will just sketch the construction, the formal proof can be found in the literature. The construction consists of two steps: first we give a direct translation of a regular grammar to an automaton and then we transform the automaton into a suitable shape.

From a grammar to an automaton. Let $G = (T, N, R, S)$ be a regular grammar without productions of the form $U \to V$ and $W \to \epsilon$ for $W \neq S$.
Construct NFA $M = (X, Q, d, \{S\}, F)$ where

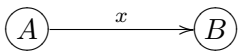
- The input alphabet of the automaton are the nonempty terminal *strings* (!) that occur in the rules of the grammar:

$$\begin{array}{rcll} X & = & \{x \in T^+ & \mid A, B \in N, A \to xB \in R\} \\ & \cup & \{x \in T^+ & \mid A \in N, A \to x \in R\} \end{array}$$

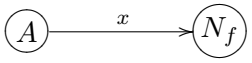
- The states of the automaton are the nonterminals of the grammar extended with a new state N_f.

$$Q = N \cup \{N_f\}$$

- The transitions of the automaton correspond to the grammar productions: for each rule $A \to xB$ we get a transition



for each rule $A \to x$ with nonempty x, we get a transition



In formulae: For all $A \in N$ and $x \in X$:

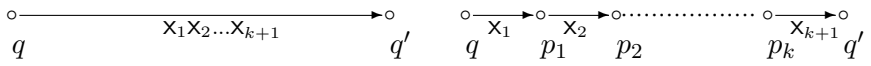
$$\begin{array}{rcl} d\ A\ x & = & \{B \mid B \in N, A \to xB \in R\} \\ & \cup & \{N_f \mid A \to x \in R\} \end{array}$$

- The final states of the automaton are N_f and possibly S, if $S \to \epsilon$ is a grammar production.

$$F = \{N_f\} \cup \{S \mid S \to \epsilon \in R\}$$

$Lnfa(M) = L(G)$, because of the direct correspondence between derivation steps in G and transitions in M.

Transforming the automaton to a suitable shape. There is a minor flaw in the automaton given above: the grammar and the automaton have different alphabets. This shortcoming will be remedied by an automaton transformation which yields an equivalent automaton with transitions labelled by elements of T (instead of T^*). The transformation is relatively easy and is depicted in the diagram below. In order to eliminate transition $d\ q\ x = q'$ where $x = \mathsf{x}_1\mathsf{x}_2 \ldots \mathsf{x}_{k+1}$ with $k > 0$ and $\mathsf{x}_i \in T$ for all i, add new (nonfinal) states $p_1, \ldots, p_k$ to the existing ones and new transitions $d\ q\ \mathsf{x}_1 = p_1, d\ p_1\ \mathsf{x}_2 = p_2, \ldots, d\ p_k\ \mathsf{x}_{k+1} = q'$.



It is intuitively clear that the resulting automaton is equivalent to the original one. Carry out this transformation for each M-transition $d\ q\ x = q'$ with $|x| > 1$ in order to get an automaton for G with the same input alphabet T. □

5.3 Regular expressions

Regular expressions are a classical and convenient way to describe, for example, the structure of terminal words. This section defines regular expressions, defines the language of a regular expression, and shows that regular expressions and regular grammars are equally expressive formalisms. We do not discuss implementations of (datatypes and functions for matching) regular expressions; implementations can be found in the literature, see [9, 6].

Definition 14: RE_T, regular expressions over alphabet T

The set RE_T of regular expressions over alphabet T is inductively defined as follows: for regular expressions R, S

regular expression

$$\begin{aligned} \emptyset &\in RE_T \\ \epsilon &\in RE_T \\ \mathsf{a} &\in RE_T \\ R + S &\in RE_T \\ R\,S &\in RE_T \\ R* &\in RE_T \\ (R) &\in RE_T \end{aligned}$$

where $\mathsf{a} \in T$. The operator $+$ is associative, commutative, and idempotent; the concatenation operator, written as juxtaposition (so x concatenated with y is denoted by xy), is associative, and ϵ is the unit of it. In formulae this reads, for all regular expressions R, S, and V,

$$\begin{aligned} R + (S + U) &= (R + S) + U \\ R + S &= S + R \\ R + R &= R \end{aligned}$$

$$\begin{aligned} R\,(S\,U) &= (R\,S)\,U \\ R\,\epsilon &= R \quad (= \epsilon\,R) \end{aligned}$$

□

Furthermore, the star operator, $*$, binds stronger than concatenation, and concatenation binds stronger than $+$. Examples of regular expressions are:

$$(\mathsf{bc})* + \emptyset$$
$$\epsilon + \mathsf{b}(\epsilon*)$$

The language (i.e. the "semantics") of a regular expression over T is a set of T-sequences compositionally defined on the structure of regular expressions. As follows.

Definition 15: Language of a regular expression
Function $Lre :: RE_T \to \{T^*\}$ returns the language of a regular expression. It is defined inductively by:

$$\begin{aligned} Lre(\emptyset) &= \emptyset \\ Lre(\epsilon) &= \{\epsilon\} \\ Lre(\mathsf{b}) &= \{\mathsf{b}\} \\ Lre(x+y) &= Lre(x) \cup Lre(y) \\ Lre(xy) &= Lre(x)\, Lre(y) \\ Lre(x*) &= (Lre\ (x))^* \end{aligned}$$

□

Since $\cup$ is associative, commutative, and idempotent, set concatenation is associative with $\{\epsilon\}$ as its unit, and function Lre is well defined. Note that the language $Lre\mathsf{b}*$ is the set consisting of zero, one or more concatenations of b, i.e., $Lre(\mathsf{b}*) = (\{\mathsf{b}\})^*$. As an example of a language of a regular expression, we compute the language of the regular expression $(\epsilon + \mathsf{bc})\mathsf{d}$.

$$\begin{aligned} & Lre((\epsilon + \mathsf{bc})\mathsf{d}) \\ = & \\ & (Lre(\epsilon + \mathsf{bc}))\,(Lre(\mathsf{d})) \\ = & \\ & (Lre(\epsilon) \cup Lre(\mathsf{bc}))\{\mathsf{d}\} \\ = & \\ & (\{\epsilon\} \cup (Lre(\mathsf{b}))(Lre(\mathsf{c})))\{\mathsf{d}\} \\ = & \\ & \{\epsilon, \mathsf{bc}\}\,\{\mathsf{d}\} \\ = & \\ & \{\mathsf{d}, \mathsf{bcd}\} \end{aligned}$$

Regular expressions are used to describe the *tokens* of a language. For example, the list

if p **then** $e1$ **else** $e2$

contains six tokens, three of which are identifiers. An *identifier* is an element in the language of the regular expression

$$letter(letter + digit)*$$

where

$$\begin{aligned} letter &= \mathsf{a} + \mathsf{b} + \ldots + \mathsf{z} + \\ &\quad\ \mathsf{A} + \mathsf{B} + \ldots + \mathsf{Z} \\ digit &= \mathsf{0} + \mathsf{1} + \ldots + \mathsf{9} \end{aligned}$$

see subsection 2.3.1.

In the beginning of this section we claimed that regular expressions and regular grammars are equivalent formalisms. We will prove this claim later, but first we illustrate the construction of a regular grammar out of a regular expressions in an example. Consider the following regular expression.

$$R = \mathsf{a}* + \epsilon + (\mathsf{a} + \mathsf{b})*$$

We aim at a regular grammar G such that $Lre(R) = L(G)$ and again we take a top-down approach.

Suppose that nonterminal A generates the language $Lre(\mathsf{a}*)$, nonterminal B generates the language $Lre(\epsilon)$, and nonterminal C generates the language $Lre((\mathsf{a} + \mathsf{b})*)$. Suppose furthermore that the productions for A, B, and C satisfy the conditions imposed upon regular grammars. Then we obtain a regular grammar G with $L(G) = Lre(R)$ by defining

$$\begin{aligned} S &\rightarrow A \\ S &\rightarrow B \\ S &\rightarrow C \end{aligned}$$

where S is the start-symbol of G. It remains to construct productions for nonterminals A, B, and C.

- The nonterminal A with productions

  $$\begin{aligned} A &\rightarrow \mathsf{a}A \\ A &\rightarrow \epsilon \end{aligned}$$

  generates the language $Lre(\mathsf{a}*)$.

- Since $Lre(\epsilon) = \{\epsilon\}$, the nonterminal B with production

  $$B \rightarrow \epsilon$$

  generates the language $\{\epsilon\}$.

- Nonterminal C with productions

$$\begin{aligned} C &\rightarrow \mathsf{a}C \\ C &\rightarrow \mathsf{b}C \\ C &\rightarrow \epsilon \end{aligned}$$

generates the language $Lre((\mathsf{a} + \mathsf{b})*)$.

For a specific example it is not difficult to construct a regular grammar for a regular expression. We now give the general result.

Theorem 16: Regular Grammar for Regular Expression
For each regular expression R there exists a regular grammar G such that

$$Lre(R) = L(G)$$

□

The proof of this theorem is given in Section 5.4.

To obtain a regular expression that generates the same language as a given regular grammar we go via an automaton. Given a regular grammar G, we can use the theorems from the previous sections to obtain a DFA D such that

$$L(G) = Ldfa(D)$$

So if we can obtain a regular expression for a DFA D, we have found a regular expression for a regular grammar. To obtain a regular expression for a DFA D, we interpret each state of D as a regular expression defined as the sum of the concatenation of outgoing terminal symbols with the resulting state. For our example DFA we obtain:

$$\begin{aligned} S &= aC + bA + cS \\ A &= aB \\ B &= cC \\ C &= \epsilon \end{aligned}$$

It is easy to merge these four regular expressions into a single regular expression, partially because this is a simple example. Merging the regular expressions obtained from a DFA that may loop is more complicated, as we will briefly explain in the proof of the following theorem. In general, we have:

Theorem 17: Regular Expression for Regular Grammar
For each regular grammar G there exists a regular expression R such that

$$L(G) = Lre(R)$$

□

The proof of this theorem is given in Section 5.4.

5.4 Proofs

This section contains the proofs of some of the theorems given in this chapter.

Proof: of Theorem 7.
Suppose $M = (X, Q, d, Q_0, F)$ is a nondeterministic finite-state automaton. Define the finite-state automaton $M' = (X', Q', d', Q_0', F')$ as follows.

$$\begin{array}{lcl} X' & = & X \\ Q' & = & \mathit{subs}\ Q \end{array}$$

where *subs* returns the powerset of a set. For example,

$$\mathit{subs}\ \{A, B\} \quad = \quad \{\{\ \}, \{A\}, \{A, B\}, \{B\}\}$$

For the other components of M' we define

$$\begin{array}{rcl} d'\ q\ \mathsf{a} & = & \{t \mid t \in d\ r\ \mathsf{a}, r \in q\} \\ Q_0' & = & Q_0 \\ F' & = & \{p \mid p \cap F \neq \emptyset, p \in Q'\} \end{array}$$

We have

$$\begin{array}{ll} & \mathit{Lnfa}(M) \\ = & \quad \text{definition of } \mathit{Lnfa} \\ & \{w \mid w \in X^*, \mathit{nfa\_accept}\ w\ (d, Q_0, F)\} \\ = & \quad \text{definition of } \mathit{nfa\_accept} \\ & \{w \mid w \in X^*, (\mathit{nfa}\ d\ Q_0\ w \cap F) \neq \emptyset\} \\ = & \quad \text{definition of } F' \\ & \{w \mid w \in X^*, \mathit{nfa}\ d\ Q_0\ w \in F'\} \\ = & \quad \textbf{assume } \mathit{nfa}\ d\ Q_0\ w = \mathit{dfa}\ d'\ Q_0'\ w \\ & \{w \mid w \in X^*, \mathit{dfa}\ d'\ Q_0'\ w \in F'\} \\ = & \quad \text{definition of } \mathit{dfa\_accept} \\ & \{w \mid w \in X^*, \mathit{dfa\_accept}\ w\ (d', Q_0', F')\} \\ = & \quad \text{definition of } \mathit{Ldfa} \\ & \mathit{Ldfa}(M') \end{array}$$

It follows that $\mathit{Lnfa}(M) = \mathit{Ldfa}(M')$ provided

$$\mathit{nfa}\ d\ Q_0\ w \quad = \quad \mathit{dfa}\ d'\ Q_0'\ w \tag{5.1}$$

We prove this equation as follows.

$$\begin{array}{ll} & \mathit{nfa}\ d\ Q_0\ w \\ = & \quad \text{definition of } \mathit{nfa} \end{array}$$

$$
\begin{array}{ll}
 & \mathit{foldl}\ (\mathit{deltas}\ d)\ Q_0\ w \\
= & \quad Q_0 = Q_0'\ ;\ \textbf{assume}\ \mathit{deltas}\ d = d' \\
 & \mathit{foldl}\ d'\ Q_0'\ w \\
= & \quad \text{definition of } \mathit{dfa} \\
 & \mathit{dfa}\ d'\ Q_0'\ w
\end{array}
$$

So if

$$\mathit{deltas}\ d \ = \ d'$$

equation (5.1) holds. This equality follows from the following calculation.

$$
\begin{array}{ll}
 & d'\ q\ \mathsf{a} \\
= & \quad \text{definition of } d' \\
 & \{t \mid r \in q, t \in d\ r\ \mathsf{a}\} \\
= & \quad \text{definition of } \mathit{deltas} \\
 & \mathit{deltas}\ d\ q\ \mathsf{a}
\end{array}
$$

□

Proof: of Theorem 16.
The proof of this theorem is by induction to the structure of regular expressions. For the three base cases we argue as follows.

The regular grammar without productions generates the language $\mathit{Lre}(\emptyset)$. The regular grammar with the production $S \to \epsilon$ generates the language $\mathit{Lre}(\epsilon)$. The regular grammar with production $S \to \mathsf{b}$ generates the language $\mathit{Lre}(\mathsf{b})$.

For the other three cases the induction hypothesis is that there exist a regular grammar with start-symbol S_1 that generates the language $\mathit{Lre}(x)$, and a regular grammar with start-symbol S_2 that generates the language $\mathit{Lre}(y)$.

We obtain a regular grammar with start-symbol S that generates the language $\mathit{Lre}(x + y)$ by defining

$$
\begin{array}{lcl}
S & \to & S_1 \\
S & \to & S_2
\end{array}
$$

We obtain a regular grammar with start-symbol S that generates the language $\mathit{Lre}(xy)$ by replacing, in the regular grammar that generates the language $\mathit{Lre}(x)$, each production of the form $T \to \mathsf{a}$ and $T \to \epsilon$ by $T \to \mathsf{a}S_2$ and $T \to S_2$, respectively.

We obtain a regular grammar with start-symbol S that generates the language $\mathit{Lre}(x*)$ by replacing, in the regular grammar that generates the language $\mathit{Lre}(x)$, each production of the form $T \to \mathsf{a}$ and $T \to \epsilon$ by $T \to \mathsf{a}S$ and $T \to S$, and by adding the productions $S \to S_1$ and $S \to \epsilon$, where S_1 is the start-symbol of the regular grammar that generates the language $\mathit{Lre}(x)$. □

Proof: of Theorem 17.
In sections 5.1.4 and 5.2.1 we have shown that there exists a DFA $D = (X, Q, d, S, F)$

such that

$$L(G) \quad = \quad Ldfa(D)$$

So, if we can show that there exists a regular expression R such that

$$Ldfa(D) \quad = \quad Lre(R)$$

then the theorem follows.
Let $D = (X, Q, d, S, F)$ be a DFA such that $L(G) = Ldfa(D)$. We define a regular expression R such that

$$Lre(R) \quad = \quad Ldfa(D)$$

For each state $q \in Q$ we define a regular expression $\bar{q}$, and we let R be $\bar{S}$. We obtain the definition of $\bar{q}$ by combining all pairs c and C such that $d\ q\ \mathsf{c} = C$.

$$\begin{array}{lll} \bar{q} & = & \mathit{if}\ q \notin F \\ & & \mathit{then}\ \mathit{foldl}\ (+)\ \emptyset\ [\mathsf{c}C \mid d\ q\ \mathsf{c} = C] \\ & & \mathit{else}\ \epsilon + \mathit{foldl}\ (+)\ \emptyset\ [\mathsf{c}C \mid d\ q\ \mathsf{c} = C] \end{array}$$

This gives a set of possibly mutually recursive equations, which we have to solve. In solving these equations we use the fact that concatenation distributes over the sum operator:

$$z(x + y) \quad = \quad zx + zy$$

and that recursion can be removed by means of the star operator $*$:

$$A = xA + z \text{ (where } A \notin z) \quad \equiv \quad A = x{*}z$$

The algorithm for solving such a set of equations is omitted.
We prove $Ldfa(D) = Lre(\bar{S})$.

$$\begin{array}{ll} & Ldfa(D) \\ = & \quad \text{definition of } Ldfa \\ & \{w \mid w \in X^*, \mathit{dfa\_accept}\ w\ (d, S, F)\} \\ = & \quad \text{definition of } \mathit{dfa\_accept} \\ & \{w \mid w \in X^*, (\mathit{dfa}\ d\ S\ w) \in F\} \\ = & \quad \text{definition of } \mathit{dfa} \\ & \{w \mid w \in X^*, (\mathit{foldl}\ d\ S\ w) \in F\} \\ = & \quad \textbf{assumption} \\ & \{w \mid w \in Lre(\bar{S})\} \\ = & \quad \text{equality for set-comprehensions} \\ & Lre(\bar{S}) \end{array}$$

It remains to prove the assumption in the above calculation: for $w \in X^*$,

$$(\mathit{foldl}\ d\ S\ w) \in F \quad \equiv \quad w \in \mathit{Lre}(\bar{S})$$

We prove a generalisation of this equation, namely, for arbitrary q,

$$(\mathit{foldl}\ d\ q\ w) \in F \quad \equiv \quad w \in \mathit{Lre}(\bar{q})$$

This equation is proved by induction to the length of w. For the base case $w = \epsilon$ we calculate as follows.

$$\begin{array}{ll} & (\mathit{foldl}\ d\ q\ \epsilon) \in F \\ \equiv & \quad \text{definition of } \mathit{foldl} \\ & q \in F \\ \equiv & \quad \text{definition of } \bar{q},\ E \text{ abbreviates the fold expression} \\ & \bar{q} = \epsilon + E \\ \equiv & \quad \text{definition of } \mathit{Lre}, \text{ definition of } \bar{q} \\ & \epsilon \in \mathit{Lre}(\bar{q}) \end{array}$$

The induction hypothesis is that for all lists w with $|w| \leq n$ we have $(\mathit{foldl}\ d\ q\ w) \in F \equiv w \in \mathit{Lre}(\bar{q})$. Suppose $\mathsf{a}x$ is a list of length $n{+}1$.

$$\begin{array}{ll} & (\mathit{foldl}\ d\ q\ (\mathsf{a}x)) \in F \\ \equiv & \quad \text{definition of } \mathit{foldl} \\ & (\mathit{foldl}\ d\ (d\ q\ \mathsf{a})\ x) \in F \\ \equiv & \quad \text{induction hypothesis} \\ & x \in \mathit{Lre}(d\ \bar{q}\ \mathsf{a}) \\ \equiv & \quad \text{definition of } \bar{q},\ D \text{ is deterministic} \\ & \mathsf{a}x \in \mathit{Lre}(\bar{q}) \end{array}$$

□

SUMMARY

This chapter discusses methods for recognising sentences from regular languages, and introduces several concepts related to describing and recognising regular languages. Regular languages are used for describing simple languages like 'the language of identifiers' and 'the language of keywords' and regular expressions are convenient for the description of regular languages. The straightforward translation of a regular expression into a recogniser for the language of that regular expression results in a recogniser that is often very inefficient. By means of (non)deterministic finite-state automata we construct a recogniser that requires time linear in the length of the input list for recognising an input list.

5.5 Exercises

Exercise 5.1 ▷ Given a regular grammar G for language L, construct a regular grammar for L^*. ◁

Exercise 5.2 ▷ Transform the grammar with the following productions to a grammar without productions of the form $U \to V$ and $W \to \epsilon$ with $W \neq S$.

$$\begin{array}{lcl} S & \to & \mathsf{a}A \\ S & \to & A \\ A & \to & \mathsf{a}S \\ A & \to & B \\ B & \to & C \\ B & \to & \epsilon \\ C & \to & \mathsf{c}C \\ C & \to & \mathsf{a} \end{array}$$

◁

Exercise 5.3 ▷ Suppose that the state transition function d in the definition of a nondeterministic finite-state automaton has the following type

$$d \;::\; \{Q\} \to X \to \{Q\}$$

Function d takes a set of states V and an element a, and returns the set of states that are reachable from V with an arc labelled a. Define a function *ndfsa* of type

$$(\{Q\} \to X \to \{Q\}) \to \{Q\} \to X^* \to \{Q\}$$

which given a function d, a set of start states, and an input list, returns the set of states in which the nondeterministic finite-state automaton can end after reading the input list. ◁

Exercise 5.4 ▷ Prove the converse of Theorem 7: show that for every deterministic finite-state automaton M there exists a nondeterministic finite-state automaton M' such that

$$Lda(M) \;=\; Lna(M')$$

◁

Exercise 5.5 ▷ Regular languages are closed under complementation. Prove this claim. Hint: construct a finite automaton for $\overline{L}$ out of an automaton for regular language L. ◁

Exercise 5.6 ▷ Regular languages are closed under intersection.

1 Prove this claim using the result from the previous exercise.

2 A direct proof form this claim is the following:
Let $M_1 = (X, Q_1, d_1, S_1, F_1)$ and $M_2 = (X, Q_2, d_2, S_2, F_2)$ be DFA's for the regular languages L_1 and L_2 respectively. Define the (product) automaton $M = (X, Q_1 \times Q_2, d, (S_1, S_2), F_1 \times F_2)$ by $d\ (q_1, q_2)\ x = (d_1\ q_1\ x, d_2\ q_2\ x)$ Now prove that $Ldfa(M) = Ldfa(M_1) \cap Ldfa(M_2)$

◁

Exercise 5.7 ▷ Define nondeterministic finite-state automata that accept languages equal to the languages of the following regular grammars.

1. $$\begin{cases} S & \to & (A \\ S & \to & \epsilon \\ S & \to &)A \\ A & \to &) \\ A & \to & (\end{cases}$$

2. $$\begin{cases} S & \to & \mathsf{0}A \\ S & \to & \mathsf{0}B \\ S & \to & \mathsf{1}A \\ A & \to & \mathsf{1} \\ A & \to & \mathsf{0} \\ B & \to & \mathsf{0} \\ B & \to & \epsilon \end{cases}$$

◁

Exercise 5.8 ▷ Describe the language of the following regular expressions.

1. $\epsilon + \mathsf{b}(\epsilon*)$
2. $(\mathsf{bc})* + \emptyset$
3. $\mathsf{a}(\mathsf{b}*) + \mathsf{c}*$

◁

Exercise 5.9 ▷ Prove that for arbitrary regular expressions R, S, and T the following equivalences hold.

$$\begin{aligned} Lre(R(S+T)) &= Lre(RS+RT) \\ Lre((R+S)T) &= Lre(RT+ST) \end{aligned}$$

◁

Exercise 5.10 ▷ Give regular expressions S and R such that

$$\begin{aligned} Lre(RS) &= Lre(SR) \\ Lre(RS) &\neq Lre(SR) \end{aligned}$$

◁

Exercise 5.11 ▷ Give regular expressions V and W, with $Lre(V) \neq Lre(W)$, such that for all regular expressions R and S with $S \neq \emptyset$

$$Lre(R(S+V)) = Lre(R(S+W))$$

V and W may be expressed in terms of R and S. ◁

Exercise 5.12 ▷ Give a regular expression for the language that consists of all lists of zeros and ones such that the segment 01 occurs nowhere in a list. Examples of sentences of this language are 1110, and 000. ◁

Exercise 5.13 ▷ Give regular grammars that generate the language of the following regular expressions.

1. $((\mathsf{a} + \mathsf{bb})* + \mathsf{c})*$
2. $\mathsf{a}* + \mathsf{b}* + \mathsf{ab}$

◁

Exercise 5.14 ▷ Give regular expressions of which the language equals the language of the following regular grammars.

1. $$\begin{cases} S & \rightarrow & \mathsf{b}A \\ S & \rightarrow & \mathsf{a}C \\ S & \rightarrow & \epsilon \\ A & \rightarrow & \mathsf{b}A \\ A & \rightarrow & \epsilon \\ B & \rightarrow & \mathsf{a}C \\ B & \rightarrow & \mathsf{b}B \\ B & \rightarrow & \epsilon \\ C & \rightarrow & \mathsf{b}B \\ C & \rightarrow & \mathsf{b} \end{cases}$$
2. $$\begin{cases} S & \rightarrow & 0S \\ S & \rightarrow & 1T \\ S & \rightarrow & \epsilon \\ T & \rightarrow & 0T \\ T & \rightarrow & 1S \end{cases}$$

◁

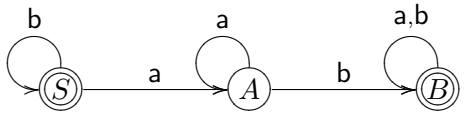
Exercise 5.15 ▷ Construct for each of the following regular expressions a nondeterministic finite-state automaton that accepts the sentences of the language. Transform the nondeterministic finite-state automata into deterministic finite-state automata.

1. $\mathsf{a}* + \mathsf{b}* + (\mathsf{ab})$
2. $(1 + (12) + 0)*(30)*$

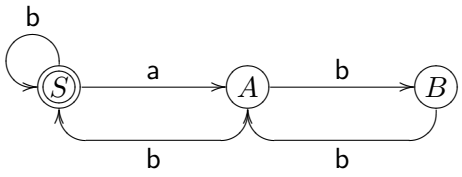
◁

Exercise 5.16 ▷ Define regular expressions for the languages of the following deterministic finite-state automata.

1. Start state is S.



2. Start state is S.



◁

Chapter 6

Compositionality

INTRODUCTION

Many recursive functions follow a common pattern of recursion. These common patterns of recursion can conveniently be captured by higher order functions. For example: many recursive functions defined on lists are instances of the higher order function `foldr`. It is possible to define a function such as `foldr` for a whole range of datatypes other than lists. Such functions are called compositional. Compositional functions on datatypes are defined in terms of algebras of semantic actions that correspond to the constructors of the datatype. Compositional functions can typically be used to define the semantics of programming languages constructs. Such semantics is referred to as algebraic semantics. Algebraic semantics often uses algebras that contain functions from tuples to tuples. Such functions can be seen as computations that read values from a component of the domain and write values to a component of the codomain. The former values are called inherited attributes and the latter values are called synthesised attributes. Attributes can be both inherited and synthesised. As explained in Section 2.4.1, there is an important relationship between grammars and compositionality: with every grammar, which describes the concrete syntax of a language, one can associate a (possibly mutually recursive) datatype, which describes the abstract syntax of the language. Compositional functions on these datatypes are called syntax driven.

GOALS

After studying this chapter and making the exercises you will

- know how to generalise constructors of a datatype to an algebra;
- know how to write compositional functions, also known as folds, on (possibly mutually recursive) datatypes;
- understand the advantages of using folds, and have seen that many problems can be solved with a fold;
- know that a fold applied to the constructors algebra is the identity function;
- have seen the notions of fusion and deforestation;

- know how to write syntax driven code;
- understand the connection between datatypes, abstract syntax and concrete syntax;
- understand the notions of synthesised and inherited attributes;
- be able to associate inherited and synthesised attributes with the different alternatives of (possibly mutually recursive) datatypes (or the different nonterminals of grammars);
- be able to define algebraic semantics in terms of compositional (or syntax driven) code that is defined using algebras of computations which make use of inherited and synthesised attributes.

ORGANISATION

The chapter is organised as follows. Section 6.1 shows how to define compositional recursive functions on built-in lists using a function which is similar to `foldr` and shows how to do the same thing with user-defined lists and streams. Section 6.2 shows how to define compositional recursive functions on several kinds of trees. Section 6.3 defines algebraic semantics. Section 6.4 shows the usefulness of algebraic semantics by presenting an expression evaluator, an expression interpreter which makes use of a stack and expression compiler to a stack machine. They only differ in the way they handle basic expressions (variables and local definitions are handled in the same way). All three examples use an algebra of computations which can read values from and write values to an environment which binds names to values. In a second version of the expression evaluator the use of inherited and synthesised attributes is made more explicit by using tuples. Section 6.5 presents a relatively complex example of the use of tuples in combination with compositionality. It deals with the problem of variable scope when compiling block structured languages.

6.1 Lists

This section introduces compositional functions on the well known datatype of lists. Compositional functions are defined on the built-in datatype `[a]` for lists (Section 6.1.1), on a user-defined datatype `List a` for lists (Section 6.1.2), and on streams or infinite lists (Section 6.1.3). We also show how to construct an algebra that directly corresponds to a datatype.

6.1.1 Built-in lists

The datatype of lists is perhaps the most important example of a datatype. A list is either the empty list `[]` or a nonempty list `(x:xs)` consisting of an element `x` at the head of the list, followed by a tail `xs` which itself is again a list. Thus, the type `[x]` is recursive. In Hugs the type `[x]` for lists is built-in. The informal definition of above corresponds to the following (pseudo) datatype.

```
data [x]  =  x : [x] | []
```

Many recursive functions on lists look very similar. Computing the sum of the elements of a list of integers (`sumL`, where the `L` denotes that it is a sum function defined on lists) is very similar to computing the product of the elements of a list of integers (`prodL`).

```
sumL, prodL   :: [Int] -> Int
sumL  (x:xs)  =  x + sumL  xs
sumL  []      =  0

prodL (x:xs)  =  x * prodL xs
prodL []      =  1
```

The function `sumL` replaces the list constructor `(:)` by `(+)` and the list constructor `[]` by `0` and the function `prodL` replaces the list constructor `(:)` by `(*)` and the list constructor `[]` by `1`. Note that we have replaced the constructor `(:)` (a constructor with two arguments) by binary operators `(+)` and `(*)` (i.e. functions with two arguments) and the constructor `[]` (a constructor with zero variables) by constants `0` and `1` (i.e. 'functions' with zero variables). The similarity between the definitions of the functions `sumL` and `prodL` can be captured by the following higher order recursive function `foldL`, which is nothing else but an uncurried version of the well known function `foldr`. Don't confuse `foldL` with Haskell's prelude function `foldl`, which works the other way around.

```
foldL         :: (x -> l -> l,l) -> [x] -> l
foldL (op,c)  =  fold where
  fold (x:xs)  =  op x (fold xs)
  fold []      =  c
```

The function `foldL` recursively replaces the constructor `(:)` by an operator `op` and the constructor `[]` by a constant `c`. We can now use `foldL` to compute the sum and product of the elements of a list of integers as follows.

```
? foldL ((+),0) [1,2,3,4]
10
? foldL ((*),1) [1,2,3,4]
24
```

The pair `(op,c)` is often referred to as a list-algebra. More precisely, a list-algebra consists of a type `l` (the carrier of the algebra), a binary operator `op` of type `x->l->l` and a constant `c` of type `l`. Note that a type (like `Int`) can be the carrier of a list-algebra in more than one way (for example using `((+),0)` and `((*),1)`). Here is another example of how to turn `Int` into a list-algebra.

```
? foldL (\_ n -> n+1,0) [1,2,3,4]
4
```

This list-algebra ignores the value at the head of a list, and increments the result obtained thus far with one. It corresponds to the function `sizeL` defined by:

```
sizeL         :: [x] -> Int
sizeL (_:xs)  =  1 + sizeL xs
sizeL []      =  0
```

Note that the type of `sizeL` is more general than the types of `sumL` and `prodL`. The type of the elements of the list does not play a role.

6.1.2 User-defined lists

In this subsection we present an example of a fold function defined on another datatype than built-in lists. To keep things simple we redo the list example for user-defined lists.

```
data List x  =  Cons x (List x) | Nil
```

User-defined lists are defined in the same way as built-in ones. The constructors `(:)` and `[]` are replaced by constructors `Cons` and `Nil`. Here are the types of the constructors `Cons` and `Nil`.

```
? :t Cons
Cons :: a -> List a -> List a
? :t Nil
Nil :: List a
```

A algebra type `ListAlgebra` corresponding to the datatype `List` directly follows the structure of that datatype.

```
type ListAlgebra x l  =  (x -> l -> l,l)
```

The left hand side of the type definition is obtained from the left hand side of the datatype as follows: a postfix `Algebra` is added at the end of the name `List` and a type variable `l` is added at the end of the whole left hand side of the type definition. The right hand side of the type definition is obtained from the right hand side of the data definition as follows: all `List x` valued constructors are replaced by `l` valued functions which have the same number of arguments (if any) as the corresponding constructors. The types of recursive constructor arguments (i.e. arguments of type `List x`) are replaced by recursive function arguments (i.e. arguments of type `l`). The types of the other arguments are simply left unchanged. In a similar way, the definition of a fold function can be generated automatically from the data definition.

```
foldList             :: ListAlgebra x l -> List x -> l
foldList (cons,nil)  =  fold where
  fold (Cons x xs)  =  cons x (fold xs)
  fold Nil          =  nil
```

The constructors `Cons` and `Nil` in the left hand sides of the definition of the local function `fold` are replaced by functions `cons` and `nil` in the right hand sides.

The function `fold` is applied recursively to all recursive constructor arguments of type `List x` to return a value of type `l` as required by the functions of the algebra (in this case `Cons` and `cons` have one such recursive argument). The other arguments are left unchanged. Recursive functions on user-defined lists which are defined by means of `foldList` are called compositional. Every algebra defines a unique compositional function. Here are three examples of compositional functions. They correspond to the examples of section 6.1.1.

```
sumList, prodList  :: List Int -> Int
sumList            =  foldList ((+),0)
prodList           =  foldList ((*),1)

sizeList  :: List x -> Int
sizeList  =  foldList (const (1+),0)
```

It is worth mentioning one particular `ListAlgebra`: the trivial `ListAlgebra` that replaces `Cons` by `Cons` and `Nil` by `Nil`. This algebra defines the identity function on user-defined lists.

```
idListAlgebra  :: ListAlgebra x (List x)
idListAlgebra  =  (Cons,Nil)

idList  :: List x -> List x
idList  =  foldList idListAlgebra

? idList (Cons 1 (Cons 2 Nil))
Cons 1 (Cons 2 Nil)
```

6.1.3 Streams

In this section we consider streams (or infinite lists).

```
data Stream x  =  And x (Stream x)
```

Here is a standard example of a stream: the infinite list of fibonacci numbers.

```
fibStream  :: Stream Int
fibStream  =  And 0 (And 1 (restOf fibStream)) where
  restOf (And x stream@(And y _)) = And (x+y) (restOf stream)
```

The algebra type `StreamAlgebra`, and the fold function `foldStream` can be generated automatically from the datatype `Stream`.

```
type StreamAlgebra x s  =  x -> s -> s

foldStream      :: StreamAlgebra x s -> Stream x -> s
foldStream and  =  fold where
  fold (And x xs)  =  and x (fold xs)
```

Note that the algebra has only one component because `Stream` has only one constructor. For the same reason the fold function is defined using only one equation. Here is an example of using a compositional function on user defined streams. It computes the first element of a monotone stream that is greater or equal than a given value.

```
firstGreaterThan    :: Ord x => x -> Stream x -> x
firstGreaterThan n  =  foldStream (\x y -> if x>=n then x else y)
```

6.2 Trees

Now that we have seen how to generalise the `foldr` function on built-in lists to compositional functions on user-defined lists and streams we proceed by explaining another common class of datatypes: trees. We will treat four different kinds of trees in the subsections below:

- binary trees;
- trees for matching parentheses;
- expression trees;
- general trees.

Furthermore, we will briefly mention the concepts of fusion and deforestation.

6.2.1 Binary trees

A binary tree is either a node where the tree splits into two subtrees or a leaf which holds a value.

```
data BinTree x  =  Bin (BinTree x) (BinTree x) | Leaf x
```

One can generate the corresponding algebra type `BinTreeAlgebra` and fold function `foldBinTree` from the datatype automatically. Note that `Bin` has two recursive arguments and that `Leaf` has one non-recursive argument.

```
type BinTreeAlgebra x t  =  (t -> t -> t,x -> t)

foldBinTree             :: BinTreeAlgebra x t -> BinTree x -> t
foldBinTree (bin,leaf)  =  fold where
  fold (Bin l r)  =  bin (fold l) (fold r)
  fold (Leaf x)   =  leaf x
```

In the `BinTreeAlgebra` type, the `bin` part of the algebra has two arguments of type `t` and the `leaf` part of the algebra has one argument of type `x`. Similarly, in the `foldBinTree` function, the local `fold` function is applied recursively to both arguments of `bin` and is not called on the argument of `leaf`. We can now define compositional functions on binary trees much in the same way as we defined them on lists. Here is an example: the function `sizeBinTree` computes the size of a binary tree.

```
sizeBinTree  :: BinTree x -> Int
sizeBinTree  =  foldBinTree ((+),const 1)

?sizeBinTree (Bin (Bin (Leaf 3) (Leaf 7)) (Leaf 11))
3
```

If a tree consists of a leaf, then `sizeBinTree` ignores the value at the leaf and returns `1` as the size of the tree. If a tree consists of two subtrees, then `sizeBinTree` returns the sum of the sizes of those subtrees as the size of the tree. Functions for computing the sum and the product of the integers at the leafs of a binary tree can be defined in a similar way. It suffices to define appropriate semantic actions `bin` and `leaf` on a type `t` (in this case `Int`) that correspond to the syntactic constructs `Bin` and `Leaf` of the datatype `BinTree`.

6.2.2 Trees for matching parentheses

Section 3.3.1 defines the datatype `Parentheses` for matching parentheses.

```
data Parentheses  =  Match Parentheses Parentheses
                  |  Empty
```

For example, the sentence `()()` of the concrete syntax for matching parentheses is represented by the value `Match Empty (Match Empty Empty)` in the abstract syntax `Parentheses`. Remember that the abstract syntax ignores the terminal bracket symbols of the concrete syntax.

We can now define, in the same way as we did for lists and binary trees, an algebra type `ParenthesesAlgebra` and a fold function `foldParentheses`, which can be used to compute the depth (`depthParentheses`) and the width (`widthParentheses`) of matching parenthesis in a compositional way. The depth of a string of matching parentheses s is the largest number of unmatched parentheses that occurs in a substring of s. For example, the depth of the string `(()))()` is 3. The width of a string of matching parentheses s is the the number of substrings that are matching parentheses themselves, which are not a substring of a surrounding string of matching parentheses. For example, the width of the string `(()))()` is 2. Compositional functions on datatypes that describe the abstract syntax of a language are called syntax driven.

```
type ParenthesesAlgebra m  =  (m -> m -> m,m)

foldParentheses  :: ParenthesesAlgebra m -> Parentheses -> m
foldParentheses (match,empty)  =  fold where
  fold (Match l r)  =  match (fold l) (fold r)
  fold Empty        =  empty

depthParenthesesAlgebra  :: ParenthesesAlgebra Int
depthParenthesesAlgebra  =  (\x y -> max (1+x) y,0)

widthParenthesesAlgebra    :: ParenthesesAlgebra Int
```

```
widthParenthesesAlgebra   =  (\_ y -> 1+y,0)

depthParentheses, widthParentheses :: Parentheses -> Int
depthParentheses = foldParentheses depthParenthesesAlgebra
widthParentheses = foldParentheses widthParenthesesAlgebra

parenthesesExample = Match (Match (Match Empty Empty) Empty)
                           (Match Empty
                                  (Match (Match Empty Empty)
                                         Empty
                           )      )

? depthParentheses parenthesesExample
3
? widthParentheses parenthesesExample
3
```

Our example reveals that abstract syntax is not very well suited for interpretation by human beings. What is the concrete representation of the matching parenthesis example represented by `parenthesesExample`? It happens to be `(((()))(()))`. Fortunately, we can easily write a program that computes the concrete representation from the abstract one. We know exactly which terminals we have deleted when going from the concrete syntax to the abstract one. The algebra used by the function `a2cParentheses` simply reinserts those terminals that we have deleted. Note that `a2cParentheses` does not deal with layout such as blanks, indentation and newlines. For a simple example layout does not really matter. For large examples layout is very important: it can be used to let concrete representations look pretty.

```
a2cParenthesesAlgebra  :: ParenthesesAlgebra String
a2cParenthesesAlgebra  =  (\xs ys -> "("++xs++")"++ys,"")

a2cParentheses  :: Parentheses -> String
a2cParentheses  =  foldParentheses a2cParenthesesAlgebra

? a2cParentheses parenthesesExample
(((()))(()))
```

This example illustrates that a computer can easily interpret abstract syntax (something human beings have difficulties with). Strangely enough, human beings can easily interpret concrete syntax (something computers have difficulties with). What we would really like is that computers can interpret concrete syntax as well. This is the place where parsing enters the picture: computing an abstract representation from a concrete one is precisely what parsers are used for.

Consider the functions `parens` and `nesting` of Section 3.3.1 again.

```
open   =  symbol '('
```

```
close  =  symbol ')'

parens  :: Parser Char Parentheses
parens  =  f  <$> open <*> parens <*> close <*> parens
       <|> succeed Empty
  where f a b c d = Match b d

nesting  :: Parser Char Int
nesting  =  f <$> open <*> nesting <*> close <*> nesting
        <|> succeed 0
  where f a b c d = max (1+b) d
```

Function `nesting` could have been defined by means of function `parens` and a fold:

```
nesting'  :: Parser Char Int
nesting'  =  depthParentheses <$> parens
```

(Remember that `depthParentheses` has been defined as a fold.) Functions `nesting` and `nesting'` compute exactly the same result. The function `nesting` is the *fusion* of the fold function with the parser `parens` from the function `nesting'`. Using laws for parsers and folds (which we have not and will not give) we can prove that the two functions are equal.

Note that function `nesting'` first builds a tree by means of function `parens`, and then flattens it by means of the fold. Function `nesting` never builds a tree, and is thus preferable for reasons of efficiency. On the other hand: in function `nesting'` we reuse the parser `parens` and the function `depthParentheses`, in function `nesting` we have to write our own parser, and convince ourselves that it is correct. So for reasons of 'programming efficiency' function `nesting'` is preferable. To obtain the best of both worlds, we would like to write function `nesting'` and have our compiler figure out that it is better to use function `nesting` in computations. The automatic transformation of function `nesting'` into function `nesting` is called *deforestation* (trees are removed). Some (very few) compilers are clever enough to perform this transformation automatically.

6.2.3 Expression trees

The matching parentheses grammar has only one nonterminal. Therefore its abstract syntax is described by a single datatype. In this section we look again at the expression grammar of section 3.5.

$E \rightarrow T$
$E \rightarrow E$ `+` T
$T \rightarrow F$
$T \rightarrow T$ `*` F
$F \rightarrow$ `(`E`)`
$F \rightarrow$ *Digs*

This grammar has three nonterminals, E, T and F. Using the approach from Section 2.6 we transform the nonterminals to datatypes:

```
data E = E1 T | E2 E T
data T = T1 F | T2 T F
data F = F1 E | F2 Int
```

where we have translated *Digs* by the type `Int`. Note that this is a rather inconvenient and clumsy abstract syntax for expressions; the following abstract syntax is more convenient.

```
data Expr = Con Int | Add Expr Expr | Mul Expr Expr
```

However, to illustrate the concept of mutual recursive datatypes, we will study the datatypes `E`, `T`, and `F` defined above. Since `E` uses `T`, `T` uses `F`, and `F` uses `E`, these three types are *mutually recursive*. The main datatype of the three datatypes is the one corresponding to the start-symbol E. Since the datatypes are mutually recursive, the algebra type `EAlgebra` consists of three tuples of functions and three carriers (the main carrier is, as always, the one corresponding to the main datatype and is therefore the one corresponding to the start-symbol).

```
type EAlgebra e t f  =  ((t -> e,e -> t -> e)
                        ,(f -> t,t -> f -> t)
                        ,(e -> f,Int -> f)
                        )
```

The fold function `foldE` for `E` also folds over `T` and `F`, so it uses three mutually recursive local functions.

```
foldE :: EAlgebra e t f -> E -> e
foldE ((e1,e2),(t1,t2),(f1,f2))  =  fold where
  fold (E1 t)     =  e1 (foldT t)
  fold (E2 e t)   =  e2 (fold e) (foldT t)
  foldT (T1 f)    =  t1 (foldF f)
  foldT (T2 t f)  =  t2 (foldT t) (foldF f)
  foldF (F1 e)    =  f1 (fold e)
  foldF (F2 n)    =  f2 n
```

We can now use `foldE` to write a syntax driven expression evaluator `evalE`. In the algebra that is used in the `foldE`, all type variables `e`, `f`, and `t` are instantiated with `Int`.

```
evalE :: E -> Int
evalE = foldE ((id,(+)),(id,(*)),(id,id))
```

```
exE = E2 (E1 (T2 (T1 (F2 2)) (F2 3))) (T1 (F2 1))

? evalE exE
7
```

Once again our example shows that abstract syntax cannot easily be interpreted by human beings. Here is a function `a2cE` which does this job for us.

```
a2cE  :: E -> String
a2cE  =  foldE ((e1,e2),(t1,t2),(f1,f2))
 where e1  =  \t -> t
       e2  =  \e t -> e++"+"++t
       t1  =  \f -> f
       t2  =  \t f -> t++"*"++f
       f1  =  \e -> "("++e++")"
       f2  =  \n -> show n

? a2cE exE
"2*3+1"
```

6.2.4 General trees

A general tree consist of a node, holding a value, where the tree splits into a list of subtrees. Notice that this list may be empty (in which case, of course, only the value at the node is of interest). As usual, the type `TreeAlgebra` and the function `foldTree` can be generated automatically from the data definition.

```
data Tree x  =  Node x [Tree x]

type TreeAlgebra x a  =  x -> [a] -> a

foldTree       :: TreeAlgebra x a -> Tree x -> a
foldTree node  =  fold where
  fold (Node x gts)  =  node x (map fold gts)
```

Notice that the constructor `Node` has a list of recursive arguments. Therefore the node function of the algebra has a corresponding list of recursive arguments. The local `fold` function is recursively called on all elements of a list using the `map` function.

One can compute the sum of the values at the nodes of a general tree as follows:

```
sumTree  :: Tree Int -> Int
sumTree  =  foldTree (\x xs -> x + sum xs)
```

Computing the product of the values at the nodes of a general tree and computing the size of a general tree can be done in a similar way.

6.2.5 Efficiency

A fold takes a value of a datatype, and replaces its constructors by functions. If the evaluation of each of these functions on their arguments takes constant time, evaluation of the fold takes time linear in the number of constructors in its argument. However, some functions require more than constant evaluation time. For example, list concatenation is linear in its left argument, and it follows that if we define the function `reverse` by

```
reverse  :: [a] -> [a]
reverse  =  foldL (\x xs -> xs ++ [x],[])
```

then function `reverse` takes time quadratic in the length of its argument list. So, folds are often efficient functions, but if the functions in the algebra are not constant, the fold is usually not linear. Often such a nonlinear fold can be transformed into a more efficient function. A technique that often can be used in such a case is the accumulating parameter technique. For example, for the `reverse` function we have

```
reverse x  =  reverse' x []

reverse'            :: [a] -> [a] -> [a]
reverse' [] ys      =  ys
reverse' (x:xs) ys  =  reverse' xs (x:ys)
```

The evaluation of `reverse' xs` takes time linear in the length of `xs`.

Exercise 6.1 ▷ Define an algebra type and a fold function for the following datatype.

```
data LNTree a b  =  Leaf a
                 |  Node (LNTree a b) b (LNTree a b)
```

◁

Exercise 6.2 ▷ Define the following functions as folds on the datatype `BinTree`, see Section 6.2.1.

1. `height`, which returns the height of a tree.
2. `flatten`, which returns the list of leaf values in left-to-right order.
3. `maxBinTree`, which returns the maximal value at the leaves.
4. `sp`, which returns the length of a shortest path.
5. `mapBinTree`, which maps a function over the elements at the leaves.

◁

Exercise 6.3 ▷ A *path* through a binary tree describes the route from the root of the tree to some leaf. We choose to represent paths by sequences of `Direction`'s:

```
data Direction  =  Left | Right
```

in such a way that taking the left subtree in an internal node will be encoded by `Left` and taking the right subtree will be encoded by `Right`. Define a compositional function `allPaths` which produces all paths of a given tree. Define this function first using explicit recursion, and then using a fold. ◁

Exercise 6.4 ▷ This exercise deals with resistors. There are some basic resistors with a fixed (floating point) resistance and, given two resistors, they can be put in parallel (`:|:`) or in sequence (`:*:`).

1. Define the datatype `Resist` to represent resistors. Also, define the datatype `ResistAlgebra` and the corresponding function `foldResist`.
2. Define a compositonal function `result` which determines the resistance of a resistors. (Recall the rules $\frac{1}{r} = \frac{1}{r_1} + \frac{1}{r_2}$ and $r = r_1 + r_2$.)

◁

6.3 Algebraic semantics

With every (possibly mutually recursive) datatype one can associate an algebra type and a fold function. The algebra is a tuple (one component for each datatype) of tuples (one component for each constructor of the datatype) of semantic actions. The algebra uses a set of auxiliary carriers (one for each datatype). One of them (the one corresponding to the main datatype) is the main carrier of the algebra. The fold function recursively replaces syntactic constructors of the datatypes by corresponding semantic actions of the algebra. Functions which are defined in terms of a fold function and an algebra are called compositional functions. There is one special algebra: the one whose components are the constructor functions of the mutually recursive datatypes. This algebra defines the identity function on the datatype. The compositional function that corresponds to an algebra is the unique, so called, algebra homomorphism from the datatype to the given algebra. Therefore the datatype is often called the initial algebra and compositional functions are said to define algebraic semantics. We summarise this important statement as follows.

```
algebraicSemantics ::  InitialAlgebra -> Algebra
```

6.4 Expressions

The first part of this section presents a basic expression evaluator. The evaluator is extended with variables in the second part and with local definitions in the third part.

6.4.1 Evaluating expressions

In this subsection we start with a more involved example: an expression evaluator. We will use another datatype for expressions than the one introduced in Section 6.2.3: here we will use a single, and hence non mutual recursive datatype for expressions. We restrict ourselves to float valued expressions on which addition, subtraction, multiplication and division are defined. The datatype and the corresponding algebra type and fold function are as follows:

```
infixl 7 `Mul`
infix  7 `Dvd`
infixl 6 `Add`, `Min`

data Expr  =  Expr `Add` Expr
           |  Expr `Min` Expr
           |  Expr `Mul` Expr
           |  Expr `Dvd` Expr
           |  Num Float

type ExprAlgebra a  =  (a->a->a   -- add
                       ,a->a->a   -- min
                       ,a->a->a   -- mul
                       ,a->a->a   -- dvd
                       ,Float->a) -- num

foldExpr  ::  ExprAlgebra a -> Expr -> a
foldExpr (add,min,mul,dvd,num)  =  fold  where
  fold (expr1 `Add` expr2)  =  fold expr1 `add` fold expr2
  fold (expr1 `Min` expr2)  =  fold expr1 `min` fold expr2
  fold (expr1 `Mul` expr2)  =  fold expr1 `mul` fold expr2
  fold (expr1 `Dvd` expr2)  =  fold expr1 `dvd` fold expr2
  fold (Num n)              =  num n
```

There is nothing special to notice about these definitions except, perhaps, the fact that `Expr` does not have an extra parameter `x` like the list and tree examples. Computing the result of an expression now simply consists of replacing the constructors by appropriate functions.

```
resultExpr  :: Expr -> Float
resultExpr  =  foldExpr ((+),(-),(*),(/),id)
```

6.4.2 Adding variables

Our next goal is to extend the evaluator of the previous subsection such that it can handle variables as well. The values of variables are typically looked up in an environment which binds the names of the variables to values. We implement an environment as a list of name-value pairs. For our purposes names are strings and values are floats. In the following programs we will use the following functions and types:

```
type Env name value  =  [(name,value)]

(?) :: Eq name => Env name value -> name -> value
env ? x = head [ v | (y,v) <- env, x == y]

type Name   =  String
type Value  =  Float
```

The datatype and the corresponding algebra type and fold function are now as follows. Note that we use the same name (`Expr`) for the datatype, although it differs from the previous `Expr` datatype.

```
data Expr  =  Expr `Add` Expr
           |  Expr `Min` Expr
           |  Expr `Mul` Expr
           |  Expr `Dvd` Expr
           |  Num Value
           |  Var Name

type ExprAlgebra a  =  (a->a->a  -- add
                       ,a->a->a  -- min
                       ,a->a->a  -- mul
                       ,a->a->a  -- dvd
                       ,Value->a -- num
                       ,Name->a) -- var

foldExpr :: ExprAlgebra a -> Expr -> a
foldExpr (add,min,mul,dvd,num,var)  =  fold where
  fold (expr1 `Add` expr2)  =  fold expr1 `add` fold expr2
  fold (expr1 `Min` expr2)  =  fold expr1 `min` fold expr2
  fold (expr1 `Mul` expr2)  =  fold expr1 `mul` fold expr2
  fold (expr1 `Dvd` expr2)  =  fold expr1 `dvd` fold expr2
  fold (Num n)              =  num n
  fold (Var x)              =  var x
```

`Expr` now has an extra constructor: the unary constructor `Var`. Similarly, the argument of `foldExpr` now has an extra component: the unary function `var` which corresponds to the unary constructor `Var`. Computing the result of an expression somehow needs to use an environment. Here is a first, bad way of doing this: one can use it as an argument of a function that computes an algebra (we will explain why this is a bad choice in the next subsection; the basic idea is that we use the environment as a global variable here).

```
resultExprBad      :: Env Name Value -> Expr -> Value
resultExprBad env  =  foldExpr ((+),(-),(*),(/),id,(env ?))

?resultExprBad [("x",3)] (Var "x" `Mul` Num 2)
6
```

The good way of using an environment is the following: instead of working with a computation which, given an environment, yields an algebra of values it is better to turn the computation itself into an algebra. Thus we turn the environment in a 'local' variable.

```
(<+>),(<->),(<*>),(</>) :: (Env Name Value -> Value) ->
                           (Env Name Value -> Value) ->
                           (Env Name Value -> Value)
f <+> g = \env -> f env + g env
f <-> g = \env -> f env - g env
f <*> g = \env -> f env * g env
f </> g = \env -> f env / g env

resultExprGood  :: Expr -> (Env Name Value -> Value)
resultExprGood  =
  foldExpr ((<+>),(<->),(<*>),(</>),const,flip (?))

?resultExprGood (Var "x" `Mul` Num 2) [("x",3)]
6
```

The actions (`(+)`, `(-)`, ...) on values are now replaced by corresponding actions (`(<+>)`, `<->)`, ... on computations. Computing the result of the sum of two subexpressions within a given environment consists of computing the result of the subexpressions within this environment and adding both results to yield a final result. Computing the result of a constant does not need the environment at all. Computing the result of a variable consists of looking it up in the environment. Thus, the algebraic semantics of an expression is a computation which yields a value. This important statement can be summarised as follows.

```
algebraicSemantics ::  InitialAlgebra -> Compute Value
```

In this case the computation is of the form `env -> val`. The value type is an example of a synthesised attribute. The value of an expression is synthesised from values of its subexpressions. The environment type is an example of an inherited attribute. The environment which is used by the computation of a subexpression of an expression is inherited from the computation of the expression. Since we are working with abstract syntax we say that the synthesised and inherited attributes are attributes of the datatype `Expr`. If `Expr` is one of the mutually recursive datatypes which are generated from the nonterminals of a grammar, then we say that the synthesised and inherited attributes are attributes of the nonterminal.

6.4.3 Adding definitions

Our next goal is to extend the evaluator of the previous subsection such that it can handle definitions as well. A definition is an expression of the form `Def name expr1 expr2`, which should be interpreted as: let the value of `name` be

equal to `expr1` in expression `expr2`. Variables are typically defined by updating the environment with an appropriate name-value pair.

The datatype (called `Expr` again) and the corresponding algebra type and eval function are now as follows:

```
data Expr  =  Expr `Add` Expr
           |  Expr `Min` Expr
           |  Expr `Mul` Expr
           |  Expr `Dvd` Expr
           |  Num Value
           |  Var Name
           |  Def Name Expr Expr

type ExprAlgebra a  =  (a->a->a        -- add
                       ,a->a->a        -- min
                       ,a->a->a        -- mul
                       ,a->a->a        -- dvd
                       ,Value->a       -- num
                       ,Name->a        -- var
                       ,Name->a->a->a) -- def

foldExpr :: ExprAlgebra a -> Expr -> a
foldExpr (add,min,mul,dvd,num,var,def)  =  fold where
  fold (expr1 `Add` expr2)  =  fold expr1 `add` fold expr2
  fold (expr1 `Min` expr2)  =  fold expr1 `min` fold expr2
  fold (expr1 `Mul` expr2)  =  fold expr1 `mul` fold expr2
  fold (expr1 `Dvd` expr2)  =  fold expr1 `dvd` fold expr2
  fold (Num n)              =  num n
  fold (Var x)              =  var x
  fold (Def x value body)   =  def x (fold value) (fold body)
```

`Expr` now has an extra constructor: the ternary constructor `Def`, which can be used to introduce a local variable. For example, the following expression can be used to compute the number of seconds per year.

```
seconds = Def "days_per_year"      (Num 365) (
          Def "hours_per_day"      (Num 24)  (
          Def "minutes_per_hour"   (Num 60)  (
          Def "seconds_per_minute" (Num 60)  (
            Var "days_per_year"      `Mul`
            Var "hours_per_day"      `Mul`
            Var "minutes_per_hour"   `Mul`
            Var "seconds_per_minute"         ))))
```

Similarly, the parameter of `foldExpr` now has an extra component: the ternary function `def` which corresponds to the ternary constructor `Def`. Notice that the last two arguments are recursive ones. We can now explain why the first use of

environments is inferior to the second one. Trying to extend the first definition gives something like:

```
resultExprBad      :: Env Name Value -> Expr -> Value
resultExprBad env  =
  foldExpr ((+),(-),(*),(/),id,(env ?),error "def")
```

The last component causes a problem: a body that contains a local definition has to be evaluated in an updated environment. We cannot update the environment in this setting: we can read the environment but afterwards it is not accessible any more in the algebra (which consists of values). Extending the second definition causes no problems: the environment is now accessible in the algebra (which consists of computations). We can easily add a new action which updates the environment. The computation corresponding to the body of an expression with a local definition can now be evaluated in the updated environment.

```
f <+> g   =  \env   -> f env + g env
f <-> g   =  \env   -> f env - g env
f <*> g   =  \env   -> f env * g env
f </> g   =  \env   -> f env / g env
x <:=> f  =  \g env -> g ((x,f env):env)

resultExprGood  :: Expr -> (Env Name Value -> Value)
resultExprGood  =
  foldExpr ((<+>),(<->),(<*>),(</>),const,flip (?),(<:=>))

?resultExprGood seconds []
31536000
```

Note that by prepending a pair `(x,y)` to an environment (in the definition of the operator `<:=>`), we add the pair to the environment. By definition of `(?)`, the binding for `x` hides possible other bindings for `x`.

6.4.4 Compiling to a stack machine

In this section we compile expressions to instructions on a stack machine. We can then use this stack machine to evaluate compiled expressions. This section is inspired by an example in [3].

Imagine a simple computer for evaluating arithmetic expressions. This computer has a 'stack' and can execute 'instructions' which change the value of the stack. The class of possible instructions is defined by the following datatype.

```
data MachInstr v = Push v | Apply (v -> v -> v)
type MachProgr v = [MachInstr v]
```

An instruction either pushes a value of type `v` on the stack, or it executes an operator that takes the two top values of the stack, applies the operator, and

pushes the result back on the stack. A stack (a value of type `Stack v` for some value type `v`) is a list of values, from which you can `pop` values, on which you can `push` values, and from which you can take the `top` value. A module `Stack` for stacks can be found in appendix A. The effect of executing an instruction of type `MachInstr` is defined by

```
execute               :: MachInstr v -> Stack v -> Stack v
execute (Push x) s    =  push x s
execute (Apply op) s  =  let a  =  top s
                             t  =  pop s
                             b  =  top t
                             u  =  pop t
                         in push (op a b) u
```

A sequence of instructions is executed by the function `run` defined by

```
run           :: MachProgr v -> Stack v -> Stack v
run []     s  =  s
run (x:xs) s  =  run xs (execute x s)
```

It follows that `run` can be defined as a `foldl`.

An expression can be translated (or compiled) into a list of instructions by the function `compile`, defined by:

```
compileExpr :: Expr ->
               Env Name (MachProgr Value) ->
               MachProgr Value
compileExpr = foldExpr (add,min,mul,dvd,num,var,def) where
  f `add` g   = \env -> f env ++ g env ++ [Apply (+)]
  f `min` g   = \env -> f env ++ g env ++ [Apply (-)]
  f `mul` g   = \env -> f env ++ g env ++ [Apply (*)]
  f `dvd` g   = \env -> f env ++ g env ++ [Apply (/)]
  num v       = \env -> [Push v]
  var x       = \env -> env ? x
  def x fd fb = \env -> fb ((x,fd env):env)
```

We now have for all expressions `e`, environments `e1` and `e2`, and stacks `s`:

```
top (run (compileExpr e e1) s)  =  resultExprGood e e2
```

The proof of this equality, which is by induction on expressions, is omitted.

Exercise 6.5 ▷ Define the following functions as folds on the datatype `Expr` that contains definitions.

1. `isSum`, which determines whether or not an expression is a sum.
2. `vars`, which returns the list of variables that occur in the expression.

◁

Exercise 6.6 ▷ This exercise deals with expressions without definitions. The function `der` is defined by

```
der ::  Expr -> String -> Expr
der (e1 `Add` e2) dx  = der e1 dx `Add` der e2 dx
der (e1 `Min` e2) dx  = der e1 dx `Min` der e2 dx
der (e1 `Mul` e2) dx  = e1 `Mul` (der e2 dx) `Add` (der e1 dx) `Mul` e2
der (e1 `Dvd` e2) dx  = (e2 `Mul` (der e1 dx)
                              `Min` e1 `Mul` (der e2 dx))
                        `Dvd` (e2 `Mul` e2)
der (Num f) dx        = Num 0.0
der (Var s) dx        = if s == dx then Num 1.0 else Num 0.0
```

1. Give an informal description of the function `der`.

2. Why is the function `der` not compositional ?

3. Define a datatype `Exp` to represent expressions consisting of (floating point) constants, variables, addition and substraction. Also, define the type `ExpAlgebra` and the corresponding `foldExp`.

4. Define the function `der` on `Exp` and show that this function is compositional.

◁

Exercise 6.7 ▷ Define the function `replace`, which given a binary tree and an element `m` replaces the elements at the leaves by `m` as a fold on the datatype `BinTree`, see Section 6.2.1. It is easy to write a function with the required functionality if you swap the arguments, but then it is impossible to write `replace` as a fold. Note that the fold returns a function, which when given `m` replaces all the leaves by `m`. ◁

Exercise 6.8 ▷ Consider the datatype of paths introduced in Exercise 3. A path in a tree leads to a unique leaf. Define a compositonal function `path2Value` which, given a tree and a path in the tree, yields the element at the unique leaf. ◁

6.5 Block structured languages

This section presents a more complex example of the use of tuples in combination with compositionality. The example deals with the scope of variables in a block structured language. A variable from a global scope is visible in a local scope only if it is not hidden by a variable with the same name in the local scope.

6.5.1 Blocks

A block is a list of statements. A statement is a variable declaration, a variable usage or a nested block. The concrete representation of an example block of our block structured language looks as follows (`dcl` stands for declaration, `use` stands for usage and `x`, `y` and `z` are variables).

```
use x ; dcl x ;
(use z ; use y ; dcl x ; dcl z ; use x) ;
dcl y ; use y
```

Statements are separated by semicolons. Nested blocks are surrounded by parentheses. The usage of `z` refers to the local declaration (the only declaration of `z`). The usage of `y` refers to the global declaration (the only declaration of `y`). The local usage of `x` refers to the local declaration and the global usage of `x` refers to the global declaration. Note that it is allowed to use variables before they are declared. Here are some mutually recursive (data)types, which describe the abstract syntax of blocks, corresponding to the grammar that describes the concrete syntax of blocks which is used above. We use meaningful names for data constructors and we use built-in lists instead of user-defined lists for the blockalgebra. As usual, the algebra type `BlockAlgebra`, which consists of two tuples of functions, and the fold function `foldBlock`, which uses two mutually recursive local functions, can be generated from the two mutually recursive (data)types.

```
type Block      =  [Statement]
data Statement  =  Dcl Idf | Use Idf | Blk Block

type Idf  =  String

type BlockAlgebra b s = ((s -> b -> b,b)
                        ,(Idf -> s,Idf -> s,b -> s)
                        )

foldBlock :: BlockAlgebra b s -> Block -> b
foldBlock ((cons,empty),(dcl,use,blk)) = fold where
 fold (s:b)     =  cons (foldS s) (fold b)
 fold []        =  empty
 foldS (Dcl x)  =  dcl x
 foldS (Use x)  =  use x
 foldS (Blk b)  =  blk (fold b)
```

6.5.2 Generating code

The goal of this section is to generate code from a block. The code consists of a sequence of instructions. There are three types of instructions.

- `Enter (l,c)`: enters the `l`'th nested block in which `c` local variables are declared.

- `Leave (l,c)`: leaves the `l`'th nested block in which `c` local variables were declared.
- `Access (l,c)`: accesses the `c`'th variable of the `l`'th nested block.

The code generated for the above example looks as follows.

```
[Enter (0,2),Access (0,0)
,Enter (1,2),Access (1,1),Access (0,1),Access (1,0),Leave (1,2)
,Access (0,1),Leave (0,2)
]
```

Note that we start numbering levels (`l`) and counts (`c`) (which are sometimes called displacements) from `0`. The abstract syntax of the code to be generated is described by the following datatype.

```
type Count  =  Int
type Level  =  Int

type Variable   =  (Level,Count)
type BlockInfo  =  (Level,Count)

data Instruction  =  Enter BlockInfo
                  |  Leave BlockInfo
                  |  Access Variable

type Code  =  [Instruction]
```

The function `ab2ac`, which generates abstract code (a value of type `Code`) from an abstract block (a value of type `Block`), uses a compositional function `block2Code`. For all syntactic constructs of `Statement` and `Block` we define appropriate semantic actions on an algebra of computations. Here is a, somewhat simplified, description of these semantic actions.

- `Dcl`: Every time we declare a local variable `x` we have to update the local environment `le` of the block we are in by associating with `x` the current level–local-count pair `(l,lc)`. Moreover we have to increment the local variable count `lc` to `lc+1`. Note that we do not generate any code for a declaration statement. Instead we perform some computations which make it possible to generate appropriate code for other statements.

  ```
  dcl x (le,l,lc) = (le',lc') where
    le' = le `update` (x,(l,lc))
    lc' = lc+1
  ```

  where function `update` is defined in the `AssociationList` module.

- `Use`: Every time we use a local variable `x` we have to generate code `cd'` for it. This code is of the form `[Access (l,lc)]`. The level–local-count pair `(l,lc)` of the variable is looked up in the global environment `e`.

```
use x e = cd' where
  cd' = [Access (l,c)]
  (l,c) = e ? x
```

- `Blk`: Every time we enter a nested block we increment the global level `l` to `l+1`, start with a fresh local variable count `0` and set the local environment of the nested block we enter to the current global environment `e`. The computation for the nested block results in a local variable count `lcB` and a local environment `leB`. Furthermore we need to make sure that the global environment (the one in which we look up variables) which is used by the computation for the nested block is equal to `leB`. The code which is generated for the block is surrounded by an appropriate `[Enter lcB]-[Leave lcB]` pair.

```
blk fB (e,l) = cd' where
 l' = l+1
 (leB,lcB,cdB) = fB (leB,l',e,0)
 cd' = [Enter (l',lcB)]++cdB++[Leave (l',lcB)]
```

- `[]`: No action need to be performed for an empty block.

- `(:)`: For every nonempty block we perform the computation of the first statement of the block which, given a local environment `le` and local variable count `lc`, results in a local environment `leS` and local variable count `lcS`. This environment-count pair is then used by the computation of the rest of the block to result in a local environment `le'` and local variable count `lc'`. The code `cd'` which is generated is the concatenation `cdS++cdB` of the code `cdS` which is generated for the first statement and the code `cdB` which is generated for the rest of the block.

```
cons fS fB (le,lc) = (le',lc',cd') where
 (leS,lcS,cdS) = fS (le,lc)
 (le',lc',cdB) = fB (leS,lcS)
 cd' = cdS++cdB
```

What does our actual computation type look like? For `dcl` we need three inherited attributes: a global level, a local block environment and a local variable count. Two of them: the local block environment and the local variable count are also synthesised attributes. For `use` we need one inherited attribute: a global block environment, and we compute one synthesised attribute: the generated code. For `blk` we need two inherited attributes: a global block environment and a global level, and we compute two synthesised attributes: the local variable count and the generated code. Moreover there is one extra attribute: a local block environment which is both inherited and synthesised. When processing the statements of a nested block we already make use of the global block environment which we are synthesising (when looking up variables). For `cons` we compute three synthesised attributes: the local block environment, the local

variable count and the generated code. Two of them, the local block environment and the local variable count are also needed as inherited attributes. It is clear from the considerations above that the following types fulfill our needs.

```
type BlockEnv   =  [(Idf,Variable)]
type GlobalEnv  =  (BlockEnv,Level)
type LocalEnv   =  (BlockEnv,Count)
```

The implementation of `block2Code` is now a straightforward translation of the actions described above. Attributes which are not mentioned in those actions are added as extra components which do not contribute to the functionality.

```
block2Code :: Block -> GlobalEnv -> LocalEnv -> (LocalEnv,Code)
block2Code = foldBlock ((cons,empty),(dcl,use,blk)) where
 cons fS fB (e,l) (le,lc) = ((le',lc'),cd') where
  ((leS,lcS),cdS) = fS (e,l) (le,lc)
  ((le',lc'),cdB) = fB (e,l) (leS,lcS)
  cd' = cdS++cdB
 empty (e,l) (le,lc) = ((le,lc),[])
 dcl x (e,l) (le,lc) = ((le',lc'),[]) where
  le' = (x,(l,lc)):le
  lc' = lc+1
 use x (e,l) (le,lc) = ((le,lc),cd') where
  cd' = [Access (l,c)]
  (l,c) = e ? x
 blk fB (e,l) (le,lc) = ((le,lc),cd') where
  ((leB,lcB),cdB) = fB (leB,l') (e,0)
  l' = l+1
  cd' = [Enter (l',lcB)] ++ cdB ++ [Leave (l',lcB)]
```

The code generator starts with an empty local environment, a fresh level and a fresh local variable count. The code is a synthesised attribute. The global environment is an attribute which is both inherited and synthesised. When processing a block we already use the global environment which we are synthesising (when looking up variables).

```
ab2ac :: Block -> Code
ab2ac b = [Enter (0,c)] ++ cd ++ [Leave (0,c)] where
 ((e,c),cd) = block2Code b (e,0) ([],0)

aBlock
 = [Use "x",Dcl "x"
   ,Blk [Use "z",Use "y",Dcl "x",Dcl "z",Use "x"]
   ,Dcl "y",Use "y"]

? ab2ac aBlock
[Enter (0,2),Access (0,0)
,Enter (1,2),Access (1,1),Access (0,1),Access (1,0),Leave (1,2)
,Access (0,1),Leave (0,2)]
```

6.6 Exercises

Exercise 6.9 ▷ Consider your answer to exercise 2.25, which gives an abstract syntax for palindromes.

1. Define a type `PalAlgebra` that describes the type of the semantic actions that correspond to the syntactic constructs of `Pal`.
2. Define the function `foldPal`, which describes how the semantics actions that correspond to the syntactic constructs of `Pal` should be applied.
3. Define the functions `a2cPal` and `aCountPal` as `foldPal`'s.
4. Define the parser `pfoldPal` which interprets its input in an arbitrary semantic `PalAlgebra` without building the intermediate abstract syntax tree.
5. Describe the parsers `pfoldPal m1` and `pfoldPal m2` where `m1` and `m2` correspond to the algebras of `a2cPal` and `aCountPal` respectively.

◁

Exercise 6.10 ▷ Consider your answer to exercise 2.26, which gives an abstract syntax for mirror-palindromes.

1. Define the type `MirAlgebra` that describes the semantic actions that correspond to the syntactic constructs of `Mir`.
2. Define the function `foldMir`, which describes how semantic actions that correspond to the syntactic constructs of `Mir` should be applied.
3. Define the functions `a2cMir` and `m2pMir` as `foldMir`'s.
4. Define the parser `pfoldMir`, which interprets its input in an arbitrary semantic `MirAlgebra` without building the intermediate abstract syntax tree.
5. Describe the parsers `pfoldMir m1` and `pfoldMir m2` where `m1` and `m2` correspond to the algebras of `a2cMir` and `m2pMir`, respectively.

◁

Exercise 6.11 ▷ Consider your answer to exercise 2.27, which gives an abstract syntax for parity-sequences.

1. Define the type `ParityAlgebra` that describes the semantic actions that correspond to the syntactic constructs of `Parity`.
2. Define the function `foldParity`, which describes how the semantic actions that correspond to the syntactic constructs of `Parity` should be applied.
3. Define the function `a2cParity` as `foldParity`.

◁

Exercise 6.12 ▷ Consider your answer to exercise 2.28, which gives an abstract syntax for bit-lists.

1. Define the type `BitListAlgebra` that describes the semantic actions that correspond to the syntactic constructs of `BitList`.
2. Define the function `foldBitList`, which describes how the semantic actions that correspond to the syntactic constructs of `BitList` should be applied.
3. Define the function `a2cBitList` as a `foldBitList`.

4 Define the parser `pfoldBitList`, which interprets its input in an arbitrary semantic `BitListAlgebra` without building the intermediate abstract syntax tree.

◁

Exercise 6.13 ▷ The following grammar describes the concrete syntax of a simple block-structured programming language

B	$\rightarrow$	SR	Block
R	$\rightarrow$	$;SR \mid \epsilon$	Rest
S	$\rightarrow$	$D \mid U \mid N$	Statement
D	$\rightarrow$	`x` | `y`	Declaration
U	$\rightarrow$	`X` | `Y`	Usage
N	$\rightarrow$	(B)	Nested Block

1. Define a datatype `Block` that describes the abstract syntax that corresponds to the grammar. What is the abstract representation of `x;(y;Y);X`?

2. Define the type `BlockAlgebra` that describes the semantic actions that correspond to the syntactic constructs of `Block`.

3. Define the function `foldBlock`, which describes how the semantic actions corresponding to the syntactic constructs of `Block` should be applied.

4. Define the function `a2cBlock`, which converts an abstract block into a concrete one. Write `a2cBlock` as a `foldBlock`

5. The function `checkBlock` tests whether or not each variable of a given abstract block is declared before use (declared in the same or in a surrounding block).

◁

Chapter 7

Computing with parsers

Parsers produce results. For example, the parsers for travelling schemes given in Chapter 4 return an abstract syntax, or an integer that represents the net travelling time in minutes. The net travelling time is computed directly by inserting the correct semantic functions. Another way to compute the net travelling time is by first computing the abstract syntax, and then applying a function to the abstract syntax that computes the net travelling time. This section shows several ways to compute results using parsers:

- insert a semantic function in the parser;
- apply a fold to the abstract syntax;
- use a class instead of abstract syntax;
- pass an algebra to the parser.

7.1 Insert a semantic function in the parser

In Chapter 4 we have defined two parsers: a parser that computes the abstract syntax for a travelling schema, and a parser that computes the net travelling time. These functions are obtained by inserting different functions in the basic parser. If we want to compute the total travelling time, we have to insert different functions in the basic parser. This approach works fine for a small parser, but it has some disadvantages when building a larger parser:

- semantics is intertwined with the parsing process;
- it is difficult to locate all positions where semantic functions have to be inserted in the parser.

7.2 Apply a fold to the abstract syntax

Instead of inserting operations in a basic parser, we can write a parser that parses the input to an abstract syntax, and computes the desired result by applying a fold to the abstract syntax.

An example of such an approach has been given in Section 6.2.2, where we defined two functions with the same functionality: `nesting` and `nesting'`; both compute the maximum nesting depth in a string of parentheses. Function `nesting` is defined by inserting functions in the basic parser. Function `nesting'`

is defined by applying a fold to the abstract syntax. Each of these definitions has its own merits; we repeat the main arguments below.

```
parens  :: Parser Char Parentheses
parens  =  (\_ b _ d -> Match b d)  <$>
           open <*> parens <*> close <*> parens
       <|> succeed Empty

nesting  :: Parser Char Int
nesting  =  (\_ b _ d -> max (1+b) d) <$>
            open <*> nesting <*> close <*> nesting
        <|> succeed 0

nesting'  :: Parser Char Int
nesting'  =  depthParentheses <$> parens
```

The first definition (`nesting`) is more efficient, because it does not build an intermediate abstract syntax tree. On the other hand, it might be more difficult to write because we have to insert functions in the correct places in the basic parser. The advantage of the second definition (`nesting'`) is that we reuse both the parser `parens`, which returns an abstract syntax tree, and the function `depthParentheses` (or the function `foldParentheses`, which is used in the definition of `depthParentheses`), which does recursion over an abstract syntax tree. The only thing we have to write ourselves in the second definition is the `depthParenthesesAlgebra`. The disadvantage of the second definition is that it builds an intermediate abstract syntax tree, which is 'flattened' by the fold. We want to avoid building the abstract syntax tree altogether. To obtain the best of both worlds, we would like to write function `nesting'` and have our compiler figure out that it is better to use function `nesting` in computations. The automatic transformation of function `nesting'` into function `nesting` is called *deforestation* (trees are removed). Some (very few) compilers are clever enough to perform this transformation automatically.

7.3 Deforestation

Deforestation removes intermediate trees in computations. The previous section gives an example of deforestation on the datatype `Parentheses`. This section sketches the general idea.

Suppose we have a datatype `AbstractTree`

```
data AbstractTree  =  ...
```

From this datatype we construct an algebra and a fold, see Chapter 6.

```
type AbstractTreeAlgebra a  =  ...

foldAbstractTree :: AbstractTreeAlgebra a -> AbstractTree -> a
```

A parser for the datatype `AbstractTree` (which returns a value of `AbstractTree`) has the following type:

```
parseAbstractTree  ::  Parser Symbol AbstractTree
```

where `Symbol` is some type of input symbols (for example `Char`). Suppose now that we define a function `p` that parses an `AbstractTree`, and then computes some value by folding with an algebra `f` over this tree:

```
p = foldAbstractTree f . parseAbstractTree
```

Then deforestation says that `p` is equal to the function `parseAbstractTree` in which occurrences of the constructors of the datatype `AbstractTree` have been replaced by the corresponding components of the algebra `f`. The following two sections each describe a way to implement such a deforestated function.

7.4 Using a class instead of abstract syntax

Classes can be used to implement the deforestated or fused computation of a fold with a parser. This gives a solution of the desired efficiency.

For example, for the language of parentheses, we define the following class:

```
class  Parens a  where
  match  ::  a -> a -> a
  empty  ::  a
```

Note that types of the functions in the class `Parens` correspond exactly to the two types that occur in the type `ParenthesesAlgebra`. This class is used in a parser for parentheses:

```
parens  :: Parens a => Parser Char a
parens  =  (\_ b _ d -> match b d) <$>
           open <*> parens <*> close <*> parens
       <|> succeed empty
```

The algebra is implicit in this function: the only thing we know is that there exist functions `empty` and `match` of the correct type; we know nothing about their implementation. To obtain a function `parens` that returns a value of type `Parentheses` we create the following instance of the class `Parens`.

```
instance  Parens Parentheses  where
  match  =  Match
  empty  =  Empty
```

Now we can write:

```
?(parens :: Parser Char Parentheses) "()()"
[(Match Empty (Match Empty Empty), "")
```

```
,(Match Empty Empty, "()")
,(Empty, "()()")
]
```

Note that we have to supply the type of `parens` in this expression, otherwise Hugs doesn't know which instance of `Parens` to use. This is how we turn the implicit 'class' algebra into an explicit 'instance' algebra. Another instance of `Parens` can be used to compute the nesting depth of parentheses:

```
instance  Parens Int  where
  match b d  =  max (1+b) d
  empty      =  0
```

And now we can write:

```
?(parens :: Parser Char Int) "()()"
[(1, ""), (1, "()"), (0, "()()")]
```

So the answer depends on the type we want our function `parens` to have. This also immediately shows a problem of this, otherwise elegant, approach: it does not work if we want to compute two different results of the same type, because Haskell doesn't allow you to define two (or more) instances with the same type. So once we have defined the instance `Parens Int` as above, we cannot use function `parens` to compute, for example, the width (also an `Int`) of a string of parentheses.

7.5 Passing an algebra to the parser

The previous section shows how to implement a parser with an implicit algebra. Since this approach fails when we want to define different parsers with the same result type, we make the algebras explicit. Thus we obtain the following definition of `parens`:

```
parens                :: ParenthesesAlgebra a -> Parser Char a
parens (match,empty)  =  par where
  par  =  (\_ b _ d -> match b d) <$>
          open <*> par <*> close <*> par
      <|> succeed empty
```

Note that it is now easy to define different parsers with the same result type:

```
nesting, breadth :: Parser Char Int
nesting          =  parens (\b d -> max (1+b) d,0)
breadth          =  parens (\b d -> d+1,0)
```

Chapter 8

Programming with higher-order folds

INTRODUCTION

In the previous chapters we have seen that algebras play an important role when describing the meaning of a recognised structure (a parse tree). For each recursive datatype `T` we have a function `foldT`, and for each constructor of the datatype we have a corresponding function as a component in the algebra. Chapter 6 introduces a language in which local declarations are permitted. Evaluating expressions in this language can be done by choosing an appropriate algebra. The domain of that algebra is a higher order (data)type (a (data)type that contains functions). Unfortunately, the resulting code comes as a surprise to many. In this chapter we will illustrate a related formalism, which will make it easier to construct such involved algebras. This related formalism is the attribute grammar formalism. We will not formally define attribute grammars, but instead illustrate the formalism with some examples, and give an informal definition.

We start with developing a somewhat unconventional way of looking at functional programs, and especially those programs that use functions that recursively descend over datatypes a lot. In our case one may think about these datatypes as abstract syntax trees. When computing a property of such a recursive object (for example, a program) we define two sets of functions: one set that describes how to recursively visit the nodes of the tree, and one set of functions (an algebra) that describes what to compute at each node when visited.

One of the most important steps in this process is deciding what the carrier type of the algebras is going to be. Once this step has been taken, these types are a guideline for further design steps. We will see that such carrier types may be functions themselves, and that deciding on the type of such functions may not always be simple. In this chapter we will present a view on recursive computations that will enable us to "design" the carrier type in an incremental way. We will do so by constructing algebras out of other algebras. In this way we define the meaning of a language in a *semantically compositional* way.

We will start with the rep_min example, which looks a bit artificial, and deals

with a non-interesting, highly specific problem. However, it has been chosen for its simplicity, and to not distract our attention to specific, programming language related, semantic issues. The second example of this chapter demonstrates the techniques on a larger example: a small compiler for part of a programming language.

```
data Tree  =  Leaf Int
           |  Bin  Tree Tree  deriving  Show

type TreeAlgebra a  =  (Int -> a, a -> a -> a)

foldTree  ::  TreeAlgebra a -> Tree -> a
foldTree alg@(leaf, _  ) (Leaf i)   =  leaf i
foldTree alg@(_   , bin) (Bin l r)  =  bin (foldTree alg l)
                                           (foldTree alg r)
```

Listing 9: rm.start.hs

GOALS

In this chapter you will learn:

- how to write 'circular' functional programs, or 'higher-order folds';
- how to combine algebras;
- (informally) the concept of an attribute grammar.

8.1 The rep_min problem

One of the famous examples in which the power of lazy evaluation is demonstrated is the so-called *rep_min* problem [2]. Many have wondered how this program achieves its goal, since at first sight it seems that it is impossible to compute anything with this program. We will use this problem, and a sequence of different solutions, to build up an understanding of a whole class of such programs.

In listing 9 we present the datatype `Tree`, together with its associated algebra. The *carrier type* of an algebra is the type that describes the objects of the algebra. We represent it by a type parameter of the algebra type:

```
type TreeAlgebra a  =  (Int -> a, a -> a -> a)
```

The associated evaluation function `foldTree` systematically replaces the constructors `Leaf` and `Bin` by corresponding operations from the algebra `alg` that is passed as an argument.

We now want to construct a function `rep_min :: Tree -> Tree` that returns a `Tree` with the same "shape" as its argument `Tree`, but with the values in its leaves replaced by the minimal value occurring in the original tree. For example,

```
?rep_min (Bin (Bin (Leaf 1) (Leaf 7)) (Leaf 11))
Bin (Bin (Leaf 1) (Leaf 1)) (Leaf 1)
```

```
minAlg  :: TreeAlgebra Int
minAlg  =  (id, min :: Int->Int->Int)

rep_min    :: Tree -> Tree
rep_min t  =  foldTree repAlg t
  where m      = foldTree minAlg t
        repAlg = (const (Leaf m), Bin)
```

Listing 10: rm.sol1.hs

8.1.1 A straightforward solution

A straightforward solution to the rep_min problem consists of a function in which `foldTree` is used twice: once for computing the minimal value of the leaf values, and once for constructing the resulting `Tree`. The function `rep_min` that solves the problem in this way is given in listing 10. Notice that the variable `m` is a global variable of the `repAlg`-algebra, that is used in the tree constructing call of `foldTree`. One of the disadvantages of this solution is that in the course of the computation the pattern matching associated with the inspection of the tree nodes is performed twice for each node in the tree.

Although this solution as such is no problem, we will try to construct a solution that calls `foldTree` only once.

8.1.2 Lambda lifting

We want to obtain a program for the rep_min problem in which pattern matching is used only once. Program listing 11 is an intermediate step towards this goal. In this program the global variable `m` has been removed and the second call of `foldTree` does not construct a `Tree` anymore, but instead *a function constructing a tree* of type `Int -> Tree`, which takes the computed minimal value as an argument. Notice how we have emphasized the fact that a function is returned through some superfluous notation: the first lambda in the function definitions constituting the algebra `repAlg` is required by the signature of the algebra, the second lambda, which could have been omitted, is there because the carrier set of the algebra contains functions of type `Int -> Tree`. This process is done routinely by functional compilers and is known as *lambda-lifting*.

8.1.3 Tupling computations

We are now ready to formulate a solution in which `foldTree` is called only once. Note that in the last solution the two calls of `foldTree` don't interfere with each other. As a consequence we may perform both the computation of the tree constructing function and the minimal value in one go, by tupling the results of the computations. The solution is given in listing 12. First a function `tuple` is defined. This function takes two `TreeAlgebra`s as arguments

```
repAlg = ( \_          -> \m -> Leaf m
         ,\lfun rfun -> \m -> let lt = lfun m
                                  rt = rfun m
                              in  Bin lt rt
         )

rep_min' t = (foldTree repAlg t) (foldTree minAlg t)
```

Listing 11: rm.sol2.hs

```
infix 9 `tuple`

tuple :: TreeAlgebra a -> TreeAlgebra b -> TreeAlgebra (a,b)
(leaf1, bin1) `tuple` (leaf2, bin2) = (\i   -> (leaf1  i, leaf2 i)
                                      ,\l r -> (bin1 (fst l) (fst r)
                                                ,bin2 (snd l) (snd r)
                                                )
                                      )
min_repAlg :: TreeAlgebra (Int, Int -> Tree)
min_repAlg =  (minAlg `tuple` repAlg)

rep_min'' t = r m
  where (m, r) = foldTree min_repAlg t
```

Listing 12: rm.sol3.hs

and constructs a third one, which has as its carrier tuples of the carriers of the original algebras.

8.1.4 Merging tupled functions

In the next step we transform the type of the carrier set in the previous example, `(Int, Int->Tree)`, into an equivalent type `Int -> (Int, Tree)`. This transformation is not essential here, but we use it to demonstrate that if we compute a cartesian product of functions, we may transform that type into a new type in which we compute only one function, which takes as its arguments the cartesian product of all the arguments of the functions in the tuple, and returns as its result the cartesian product of the result types. In our example the computation of the minimal value may be seen as a function of type `()->Int`. As a consequence the argument of the new type is `((), Int)`, which is isomorphic to just `Int`, and the result type becomes `(Int, Tree)`.

We want to mention here too that the reverse is in general not true; given a function of type `(a, b) -> (c, d)`, it is in general not possible to split this function into two functions of type `a -> c` and `b -> d`, which together achieve

```
mergedAlg :: TreeAlgebra (Int -> (Int,Tree))
mergedAlg = (\i         -> \m -> (i, Leaf m)
            ,\lfun rfun -> \m -> let (lm,lt) = lfun m
                                    (rm,rt) = rfun m
                                in  (lm `min` rm
                                    , Bin lt rt
                                    )
            )

rep_min''' t = r
  where (m, r) = (foldTree mergedAlg t) m
```

Listing 13: rm.sol4.hs

```
rep_min'''' t = r
  where (m, r)         = tree t m
        tree (Leaf i)  = \m -> (i, Leaf m)
        tree (Bin l r) = \m -> let (lm, lt) = tree l m
                                   (rm, rt) = tree r m
                               in (lm `min` rm, Bin lt rt)
```

Listing 14: rm.sol5.hs

the same effect. The new version is given in listing 13.

Notice how we have, in an attempt to make the different rôles of the parameters explicit, again introduced extra lambdas in the definition of the functions of the algebra. The parameters after the second lambda are there because we construct values in a higher order carrier set. The parameters after the first lambda are there because we deal with a `TreeAlgebra`. A curious step taken here is that part of the result, in our case the value `m`, is passed back as an argument to the result of `(foldTree mergedAlg t)`. Lazy evaluation makes this work.

That such programs were possible came originally as a great surprise to many functional programmers, especially to those who used to program in LISP or ML, languages that require arguments to be evaluated completely before a call is evaluated (so-called *strict evaluation* in contrast to lazy evaluation). Because of this surprising behaviour this class of programs became known as *circular programs*. Notice however that there is nothing circular in this program. Each value is defined in terms of other values, and no value is defined in terms of itself (as in `ones=1:ones`).

Finally, listing 14 shows the version of this program in which the function `foldTree` has been unfolded. Thus we obtain the original solution as given in Bird [2].

Concluding, we have systematically transformed a program that inspects each node twice into an equivalent program that inspects each node only once. The resulting solution passes back part of the result of a call as an argument to that same call. Lazy evaluation makes this possible.

Exercise 8.1 ▷ The *deepest_front* problem is the problem of finding the so-called *front* of a tree. The front of a tree is the list of all nodes that are at the deepest level. As in the rep_min problem, the trees involved are elements of the datatype `Tree`, see listing 9. A straightforward solution is to compute the height of the tree and passing the result of this function to a function `frontAtLevel :: Tree -> Int -> [Int]`.

1. Define the functions `height` and `frontAtLevel`
2. Give the four different solutions as defined in the rep_min problem.

◁

Exercise 8.2 ▷ Redo the previous exercise for the *highest_front* problem ◁

8.2 A small compiler

This section constructs a small compiler for (a part of) a small language. The compiler compiles this code into code for a hypothetical stack machine.

8.2.1 The language

The language we consider in this section has integers, booleans, function application, and an if-then-else expression. A language with just these constructs is useless, and you will extend the language in the exercises with some other constructs, which make the language a bit more interesting. We take the following context-free grammar for the concrete syntax of the language.

$$
\begin{array}{lcl}
Expr0 & \rightarrow & \texttt{if}\ Expr1\ \texttt{then}\ Expr1\ \texttt{else}\ Expr1 \mid Expr1 \\
Expr1 & \rightarrow & Expr2\ Expr2^{*} \\
Expr2 & \rightarrow & Int \mid Bool
\end{array}
$$

where Int generates integers, and $Bool$ booleans. An abstract syntax for our language is given in listing 15. Note that we use a single datatype for the abstract syntax instead of three datatypes (one for each nonterminal); this simplifies the code a bit. The listing 15 also contains a definition of a fold and an algebra type for the abstract syntax.

A parser for expressions is given in listing 16.

8.2.2 A stack machine

In section 6.4.4 we have defined a stack machine with which simple arithmetic expressions can be evaluated. Here we define a stack machine that has some

```
data ExprAS  =  If ExprAS ExprAS ExprAS
             |  Apply ExprAS ExprAS
             |  ConInt Int
             |  ConBool Bool deriving Show

type ExprASAlgebra a = (a -> a -> a -> a
                       ,a -> a -> a
                       ,Int -> a
                       ,Bool -> a
                       )

foldExprAS :: ExprASAlgebra a -> ExprAS -> a
foldExprAS (iff,apply,conint,conbool) = fold
  where fold (If ce te ee) = iff (fold ce) (fold te) (fold ee)
        fold (Apply fe ae) = apply (fold fe) (fold ae)
        fold (ConInt i)    = conint i
        fold (ConBool b)   = conbool b
```

Listing 15: ExprAbstractSyntax.hs

```
sptoken :: String -> Parser Char String
sptoken s = (\_ b _ -> b) <$>
            many (symbol ' ') <*> token s <*> many1 (symbol ' ')

boolean = const True <$> token "True" <|> const False <$> token "False"

parseExpr  :: Parser Char ExprAS
parseExpr  =  expr0
  where expr0  =  (\a b c d e f -> If b d f) <$>
                  sptoken "if"
                  <*> parseExpr
                  <*> sptoken "then"
                  <*> parseExpr
                  <*> sptoken "else"
                  <*> parseExpr
              <|> expr1
        expr1  =  chainl expr2 (const Apply <$> many1 (symbol ' '))
              <|> expr2
        expr2  =  ConBool <$> boolean
              <|> ConInt <$> natural
```

Listing 16: ExprParser.hs

```
data InstructionSM  =  LoadInt Int
                    |  LoadBool Bool
                    |  Call
                    |  SetLabel Label
                    |  BrFalse Label
                    |  BrAlways Label

type Label  =  Int
```

Listing 17: InstructionSM.hs

more instructions. The language of the previous section wil be compiled into code for this stack machine in the following section.

The stack machine we will use has the following instructions:

- it can load an integer;
- it can load a boolean;
- given an argument and a function on the stack, it can call the function on the argument;
- it can set a label in the code;
- given a boolean on the stack, it can jump to a label provided the boolean is false;
- it can jump to a label (unconditionally).

The datatype for instructions is given in listing 17.

8.2.3 Compiling to the stackmachine

How do we compile the different expressions to stack machine code? We want to define a function `compile` of type

```
compile :: ExprAS -> [InstructionSM]
```

- A `ConInt i` is compiled to a `LoadInt i`.

  ```
  compile (ConInt i) = [LoadInt i]
  ```

- A `ConBool b` is compiled to a `LoadBool b`.

  ```
  compile (ConBool b) = [LoadBool b]
  ```

- An application `Apply f x` is compiled by first compiling the argument `x`, then the 'function' `f` (at the moment it is impossible to define functions in our language, hence the quotes around 'function'), and finally putting a `Call` on top of the stack.

  ```
  compile (Apply f x) = compile x ++ compile f ++ [Call]
  ```

- An if-then-else expression `If ce te ee` is compiled by first compiling the conditional expression `ce`. Then we jump to a label (which will be set before the code of the else expression `ee` later) if the resulting boolean is false. Then we compile the then expression `te`. After the then expression we always jump to the end of the code of the if-then-else expression, for which we need another label. Then we set the label for the else expression, we compile the else expression `ee`, and, finally, we set the label for the end of the if-then-else expression.

```
compile (If ce te ee)  =  compile ce
                       ++ [BrFalse ?lab1]
                       ++ compile te
                       ++ [BrAlways ?lab2]
                       ++ [SetLabel ?lab1]
                       ++ compile ee
                       ++ [SetLabel ?lab2]
```

  Note that we use labels here, but where do these labels come from?

From the above description we see that we also need labels when compiling an expression. We add a label argument (an integer, used for the first label in the compiled code) to function `compile`, and we want function `compile` to return the first unused label. We change the type of function `compile` as follows:

```
compile  ::  ExprAS -> Label -> ([InstructionSM],Label)

type Label  =  Int
```

The four cases in the definition of `compile` have to take care of the labels. We obtain the following definition of compile:

```
compile (ConInt i)     =  \l -> ([LoadInt i],l)
compile (ConBool b)    =  \l -> ([LoadBool b],l)
compile (Apply f x)    =  \l -> let (xc,l')  = compile x l
                                    (fc,l'') = compile f l'
                                in (xc ++ fc ++ [Call],l'')
compile (If ce te ee)  =  \l -> let (cc,l') = compile ce (l+2)
                                    (tc,l'') = compile te l'
                                    (ec,l''') = compile ee l''
                                in (   cc
                                    ++ [BrFalse l]
                                    ++ tc
                                    ++ [BrAlways (l+1)]
                                    ++ [SetLabel l]
                                    ++ ec
                                    ++ [SetLabel (l+1)]
                                   ,l'''
                                   )
```

```
compile = foldExprAS compileAlgebra

compileAlgebra :: ExprASAlgebra (Label -> ([InstructionSM],Label))
compileAlgebra = (\cce cte cee -> \l ->
                                 let (cc,l') = cce (l+2)
                                     (tc,l'') = cte l'
                                     (ec,l''') = cee l''
                                 in (   cc
                                     ++ [BrFalse l]
                                     ++ tc
                                     ++ [BrAlways (l+1)]
                                     ++ [SetLabel l]
                                     ++ ec
                                     ++ [SetLabel (l+1)]
                                    ,l'''
                                    )
                  ,\cf cx -> \l -> let (xc,l')  = cx l
                                       (fc,l'') = cf l'
                                   in (xc ++ fc ++ [Call],l'')
                  ,\i -> \l -> ([LoadInt i],l)
                  ,\b -> \l -> ([LoadBool b],l)
                  )
```

Listing 18: CompileExpr.hs

Function `compile` is a fold, the carrier type of its algebra is a function of type `Label -> ([InstructionSM],Label)`. The definition of function `compile` as a fold is given in listing 18.

Exercise 8.3 ▷ Extend the code generation example by adding variables to the datatype `Expr`. ◁

Exercise 8.4 ▷ Extend the code generation example by adding definitions to the datatype `Expr` too. ◁

8.3 Attribute grammars

In Section 8.1 we have written a program that solves the rep_min problem. This program computes the minimum of a tree, and it computes the tree in which all the leaf values are replaced by the minimum value. The minimum is computed bottom-up: it is *synthesized* from its children. The minimum value is then passed on to the functions that build the tree with the minimum value in its leaves. These functions receive the minimum value from their parent tree node: they *inherit* the minimum value from their parent.

We can see the rep_min computation as a computation on a value of type `Tree`, on which two attributes are defined: the minimum and result tree attributes.

The minimum is computed bottom-up, and is then passed down to the result tree, and is therefore a synthesized and inherited attribute. The result tree is computed bottom-up, and is hence a synthesized attribute.

The formalism in which it is possible to specify such attributes and computations on datatypes or grammars is called *attribute grammars*, and was originally proposed by Donald Knuth in [10]. Attribute grammars provide a solution for the systematic description of the phases of the compiler that come after scanning and parsing. Although they look different from what we have encountered thus far and are probably a little easier to write, they can straightforwardly be mapped onto a functional program. The programs you have seen in this chapter could also have been obtained by means of such a mapping from an attribute grammar specification. Traditionally such attribute grammars are used as the input of a *compiler generator*. Just as we have seen how by introducing a suitable set of parsing combinators one may avoid the use of a special parser generator and even gain a lot of flexibility in extending the grammatical formalism by introducing more complicated combinators, we have shown how one can do without a special purpose attribute grammar processing system. But, just as the concept of a context free grammar was useful in understanding the fundamentals of parser combinators, understanding attribute grammars will help significantly in describing the semantic part of the recognition and compilation process. This chapter does not further introduce attribute grammars, but they will appear again in the course in implementing programming languages.

Chapter 9

Pumping Lemmas: the expressive power of languages

INTRODUCTION

In these lecture notes we have presented several ways to show that a language is regular or context-free, but until now we did not give any means to show the nonregularity or noncontext-freeness of a language. In this chapter we fill this gap by introducing the so-called *Pumping Lemmas*. For example, the pumping lemma for regular languages says

IF language L is regular,

THEN it has the following property P: each sufficiently long sentence $w \in L$ has a substring that can be repeated any number of times, every time yielding another word of L

In applications, pumping lemmas are used in the contrapositive way. In the regular case this means that one may conclude that L is *not* regular, if P does *not* hold. Although the ideas behind pumping lemmas are very simple, a precise formulation is not. As a consequence, it takes some effort to get familiar with applying pumping lemmas. Regular grammars and context-free grammars are part of the Chomsky hierarchy, which consists of four different kinds of grammars and their corresponding languages. Pumping lemmas are used to show that the expressive power of the different elements of the Chomsky hierarchy is different.

GOALS

After you have studied this chapter you will be able to

- prove that a language is not regular;
- prove that a language is not context-free;
- identify languages and grammars as regular, context-free or none of these;
- give examples of languages that are not regular, and/or not context-free;
- explain the Chomsky hierarchy.

9.1 The Chomsky hierarchy

In the preceding chapters we have seen contex-free grammars and regular grammars. You may now wonder: is it possible to express any language with these grammars? And: is it possible to obtain any context-free language from a regular grammar? The answer to these questions is no. The Chomsky hierarchy explains why the answer is no. The Chomsky hierarchy consists of four elements, each of which is explained below.

9.1.1 Type-0 grammars

The most powerful grammars are the type-0 grammars, in which a production has the form $\phi \rightarrow \psi$, where $\phi \in V^+, \psi \in V^*$, where V is the set of symbols of the grammar. So the left-hand side of a production may consist of a list of nonterminal and terminal symbols, instead of a single nonterminal as in context-free grammars. Type-0 grammars have the same expressive power as Turing machines, and the languages described by these grammars are the recursive enumerable languages. This expressive power comes at a cost though: it is very difficult to parse sentences from type-0 grammars.

9.1.2 Type-1 grammars

We can slightly restrict the form of the productions to obtain type-1 grammars. In a type-1, or context-sensitive grammar, each production has the form $\phi A \psi \rightarrow \phi \delta \psi$, where $\phi, \psi \in V^*, \delta \in V^+$. So a production describes how to rewrite a nonterminal A, in the context of lists of symbols ϕ and ψ. A language generated by a context-sensitive grammar is called a context-sensitive language. Although context-sensitive grammars are less expressive than type-0 grammars, parsing is still very difficult for context-sensitive grammars.

9.1.3 Type-2 grammars

The type-2 grammars are the context-free grammars which you have seen a lot in the preceding chapters. As the name says, in a context-free grammar you can rewrite nonterminals without looking at the context in which they appear. Actually, it is *impossible* to look at the context when rewriting symbols. Context-free grammars are less expressive than context-sensitive grammars. This statement can be proved using the pumping lemma for context-free languages.

However, it is much easier to parse sentences from context-free languages. In fact, a sentence of length n can be parsed in time at most $O(n^3)$ (or even a bit less than this) for any sentence of a context-free language. And if we put some more restrictions on context-free grammars (for example $LL(1)$), we obtain linear-time algorithms for parsing sentences of such grammars.

9.1.4 Type-3 grammars

The type-3 grammars in the Chomsky hierarchy are the regular grammars. Any sentence from a regular language can be processed by means of a finite-

state automaton, which takes linear time and constant space in the size of its input. The set of regular languages is strictly smaller than the set of context-free languages, a fact we will prove below by means of the pumping lemma for regular languages.

9.2 The pumping lemma for regular languages

In this section we give the pumping lemma for regular languages. The lemma gives a property that is satisfied by all regular languages. The property is a statement of the form: in sentences longer than a certain length a substring can be identified that can be duplicated while retaining a sentence. The idea behind this property is simple: regular languages are accepted by finite automata. Given a DFA for a regular language, a sentence of the language describes a path from the start state to some finite state. When the length of such a sentence exceeds the number of states, then at least one state is visited twice; consequently the path contains a cycle that can be repeated as often as desired. The proof of the following lemma is given in Section 9.4.

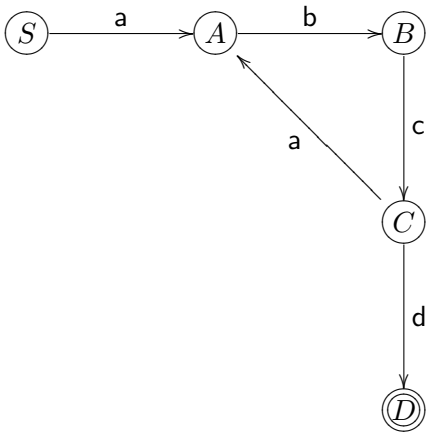
Theorem 1: Regular Pumping Lemma
Let L be a regular language. Then

there exists	$n \in \mathbb{N}$	:
for all	x, y, z	: $xyz \in L$ and $\lvert y\rvert \geq n$:
there exist	u, v, w	: $y = uvw$ and $\lvert v\rvert > 0$:
for all	$i \in \mathbb{N}$	: $xuv^i wz \in L$

□

Note that $|y|$ denotes the length of the string y. Also remember that 'for all $x \in X : \cdots$' is true if $X = \emptyset$, and 'there exists $x \in X : \cdots$' is false if $X = \emptyset$.

For example, consider the following automaton.



This automaton accepts: abcabcd, abcabcabcd, and, in general, a(bca)*bcd. The statement of the pumping lemma amounts to the following. Take for n the

number of states in the automaton (5). Let x, y, z be such that $xyz \in L$, and $|y| \geq n$. Then we know that in order to accept y, the above automaton has to pass at least twice through state A. The part that is accepted in between the two moments the automaton passes through state A can be pumped up to create sentences that contain an arbitrary number of copies of the string $v = \mathsf{bca}$.

This pumping lemma is useful in showing that a language does *not* belong to the family of regular languages. Its application is typical of pumping lemmas in general; they are used *negatively* to show that a given language does not belong to some family.

Theorem 1 enables us to prove that a language L is not regular by showing that

for all	$n \in \mathbb{N}$	:
there exist	x, y, z	: $xyz \in L$ and $\lvert y\rvert \geq n$:
for all	u, v, w	: $y = uvw$ and $\lvert v\rvert > 0$:
there exists	$i \in \mathbb{N}$	: $xuv^i wz \notin L$

In all applications of the pumping lemma in this chapter, this is the formulation we will use.

Note that if $n = 0$, we can choose $y = \epsilon$, and since there is no v with $|v| > 0$ such that $y = uvw$, the statement above holds for all such v (namely none!).
As an example, we will prove that language $L = \{\mathsf{a}^m\mathsf{b}^m \mid m \geq 0\}$ is not regular.
Let $n \in \mathbb{N}$.
Take $s = \mathsf{a}^n\mathsf{b}^n$ with $x = \epsilon$, $y = \mathsf{a}^n$, and $z = \mathsf{b}^n$.
Let u, v, w be such that $y = uvw$ with $v \neq \epsilon$, that is, $u = \mathsf{a}^p$, $v = \mathsf{a}^q$ and $w = \mathsf{a}^r$ with $p + q + r = n$ and $q > 0$.
Take $i = 2$, then

$$
\begin{array}{ll}
 & xuv^2wz \notin L \\
\Leftarrow & \quad \text{defn. } x, u, v, w, z, \text{ calculus} \\
 & \mathsf{a}^{p+2q+r}\mathsf{b}^n \notin L \\
\Leftarrow & \quad p + q + r = n \\
 & n + q \neq n \\
\Leftarrow & \quad \text{arithmetic} \\
 & q \neq 0 \\
\Leftarrow & \quad q > 0 \\
 & \mathit{true}
\end{array}
$$

Exercise 9.1 ▷ Describe the previous proof in your own words. ◁

Note that the language $L = \{\mathsf{a}^m\mathsf{b}^m \mid m \geq 0\}$ is context-free, and together with the fact that each regular grammar is also a context-free grammar it follows immediately that the set of regular languages is strictly smaller than the set of context-free languages.

Note that here we *use* the pumping lemma (and not the proof of the pumping lemma) to prove that a language is not regular. This kind of proof can be viewed as a kind of game: 'for all' is about an arbitrary element which can be chosen by the opponent; 'there exists' is about a particular element which you may choose. Choosing the right elements helps you 'win' the game, where winning means proving that a language is not regular.

Exercise 9.2 ▷ Prove that the following language is not regular

$$\{\mathsf{a}^{k^2} \mid k \geq 0\}$$

◁

Exercise 9.3 ▷ Show that the following language is not regular.

$$\{x \mid x \in \{\mathsf{a}, \mathsf{b}\}^* \ \wedge \ nr\ \mathsf{a}\ x < nr\ \mathsf{b}\ x\}$$

where $nr\ \mathsf{a}\ x$ is the number of occurrences of a in x. ◁

Exercise 9.4 ▷ Prove that the following language is not regular

$$\{\mathsf{a}^k\mathsf{b}^m \mid k \leq m \leq 2k\}$$

◁

Exercise 9.5 ▷ Show that the following language is not regular.

$$\{\mathsf{a}^k\mathsf{b}^l\mathsf{a}^m \mid k > 5 \wedge l > 3 \wedge m \leq l\}$$

◁

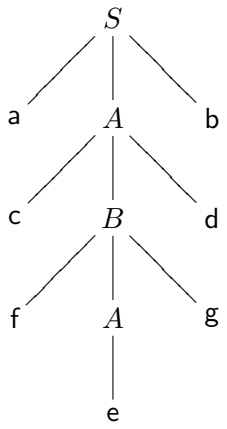
9.3 The pumping lemma for context-free languages

The Pumping Lemma for context-free languages gives a property that is satisfied by all context-free languages. This property is a statement of the form: in sentences exceeding a certain length, two sublists of bounded length can be identified that can be duplicated while retaining a sentence. The idea behind this property is the following. Context-free languages are described by context-free grammars. For each sentence in the language there exists a derivation tree. When sentences have a derivation tree that is higher than the number of nonterminals, then at least one nonterminal will occur twice in a node; consequently a subtree can be inserted as often as desired.

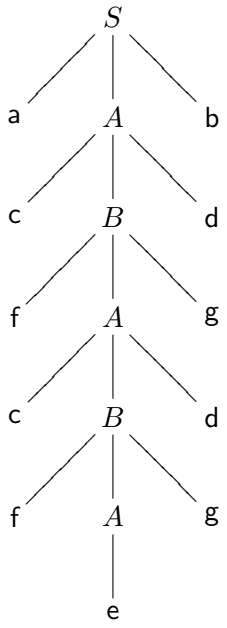
As an example of an application of the Pumping Lemma, consider the context-free grammar with the following productions.

$$\begin{array}{lcl} S & \rightarrow & \mathsf{a}A\mathsf{b} \\ A & \rightarrow & \mathsf{c}B\mathsf{d} \\ A & \rightarrow & \mathsf{e} \\ B & \rightarrow & \mathsf{f}A\mathsf{g} \end{array}$$

The following parse tree represents the derivation of the sentence acfegdb.



If we replace the subtree rooted by the lower occurrence of nonterminal A by the subtree rooted by the upper occurrence of A, we obtain the following parse tree.



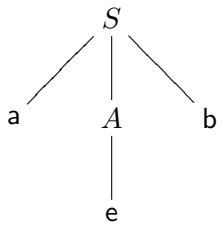
This parse tree represents the derivation of the sentence acfcfegdgdb. Thus we 'pump' the derivation of sentence acfegdb to the derivation of sentence acfcfegdgdb. Repeating this step once more, we obtain a parse tree for the sentence

acfcfcfegdgdgdb

We can repeatedly apply this process to obtain derivation trees for all sentences of the form

$\mathsf{a(cf)}^i\mathsf{e(gd)}^i\mathsf{b}$

for $i \geq 0$. The case $i = 0$ is obtained if we replace in the parse tree for the sentence acfegdb the subtree rooted by the upper occurrence of nonterminal A by the subtree rooted by the lower occurrence of A:



This is a derivation tree for the sentence aeb. This step can be viewed as a negative pumping step.

The proof of the following lemma is given in Section 9.4.

Theorem 2: Context-free Pumping Lemma
Let L be a context-free language. Then

there exist	c, d	: $c, d \in \mathbb{N}$	:
for all	z	: $z \in L$ and $\|z\| \geq c$	:
there exist	u, v, w, x, y	: $z = uvwxy$ and $\|vx\| > 0$ and $\|vwx\| \leq d$	:
for all	$i \in \mathbb{N}$	: $uv^i wx^i y \in L$	

□

The Pumping Lemma is a tool with which we prove that a given language is not context-free. The proof obligation is to show that the property shared by all context-free languages does not hold for the language under consideration.

Theorem 2 enables us to prove that a language L is not context-free by showing that

for all	c, d	: $c, d \in \mathbb{N}$	:
there exists	z	: $z \in L$ and $\|z\| \geq c$	:
for all	u, v, w, x, y	: $z = uvwxy$ and $\|vx\| > 0$ and $\|vwx\| \leq d$	:
there exists	$i \in \mathbb{N}$	: $uv^i wx^i y \notin L$	

As an example, we will prove that the language T defined by

$$T \quad = \quad \{\mathsf{a}^n \mathsf{b}^n \mathsf{c}^n \mid n > 0\}$$

is not context-free.

Proof: Let $c, d \in \mathbb{N}$.
Take $z = \mathsf{a}^r \mathsf{b}^r \mathsf{c}^r$ with $r = max(c, d)$.
Let u, v, w, x, y be such that $z = uvwxy$, $|vx| > 0$ and $|vwx| \leq d$
Note that our choice for r guarantees that substring vwx has one of the following shapes:

- vwx consists of just a's, or just b's, or just c's.
- vwx contains both a's and b's, or both b's and c's.

So vwx does *not* contain a's, b's, and c's.

Take $i = 0$, then

- If vwx consists of just a's, or just b's, or just c's, then it is impossible to write the string uwy as $\mathsf{a}^s\mathsf{b}^s\mathsf{c}^s$ for some s, since only the number of terminals of one kind is decreased.
- If vwx contains both a's and b's, or both b's and c's it lies somewhere on the border between a's and b's, or on the border between b's and c's. Then the string uwy can be written as

  $$\begin{aligned} uwy &= \mathsf{a}^s\mathsf{b}^t\mathsf{c}^r \\ uwy &= \mathsf{a}^r\mathsf{b}^p\mathsf{c}^q \end{aligned}$$

  for some s, t, p, q, respectively. At least one of s and t or of p and q is less than r. Again this list is not an element of T.

□

Exercise 9.6 ▷ Why does vwx not contain a's, b's, and c's? ◁

Exercise 9.7 ▷ Prove that the following language is not context-free

$$\{\mathsf{a}^{k^2} \mid k \geq 0\}$$

◁

Exercise 9.8 ▷ Prove that the following language is not context-free

$$\{\mathsf{a}^i \mid i \text{ is a prime number }\}$$

◁

Exercise 9.9 ▷ Prove that the following language is not context-free

$$\{ww \mid w \in \{\mathsf{a},\mathsf{b}\}^*\}$$

◁

9.4 Proofs of pumping lemmas

This section gives the proof of the pumping lemmas.

Proof: of the Regular Pumping Lemma, Theorem 1.
Since L is a regular language, there exists a deterministic finite-state automaton D such that $L = Ldfa\ D$.
Take for n the number of states of D.
Let s be an element of L with sublist y such that $|y| \geq n$, say $s = xyz$.
Consider the sequence of states D passes through while processing y. Since $|y| \geq n$, this sequence has more than n entries, hence at least one state, say state A, occurs twice.
Take u, v, w as follows

- u is the initial part of y processed until the first occurrence of A,
- v is the (nonempty) part of y processed from the first to the second occurrence of A,
- w is the remaining part of y

Note that D could have skipped processing v, and hence would have accepted $xuwz$. Furthermore, D can repeat the processing in between the first occurrence of A and the second occurrence of A as often as desired, and hence it accepts xuv^iwz for all $i \in \mathbb{N}$. Formally, a simple proof by induction shows that $(\forall i : i \geq 0 : xuv^iwz \in L)$. □

Proof: of the Context-free Pumping Lemma, Theorem 2.
Let $G = (T, N, R, S)$ be a context-free grammar such that $L = L(G)$. Let m be the length of the longest right-hand side of any production, and let k be the number of nonterminals of G.
Take $c = m^k$. In Lemma 3 below we prove that if z is a list with $|z| > c$, then in all derivation trees for z there exists a path of length at least $k+1$.

Let $z \in L$ such that $|z| > c$. Since grammar G has k nonterminals, there is at least one nonterminal that occurs more than once in a path of length $k+1$ (which contains $k+2$ symbols, of which at most one is a terminal, and all others are nonterminals) of a derivation tree for z. Consider the nonterminal A that satisfies the following requirements.

- A occurs at least twice in the path of length $k+1$ of a derivation tree for z.

Call the list corresponding to the derivation tree rooted at the lower A w, and call the list corresponding to the derivation tree rooted at the upper A (which contains the list w) vwx.

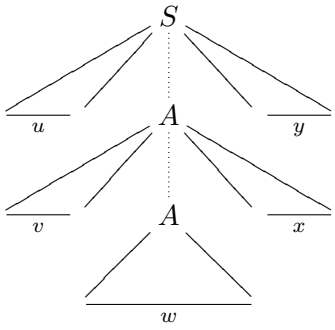
- A is chosen such that at most one of v and x equals ϵ.
- Finally, we suppose that below the upper occurrence of A no other nonterminal that satisfies the same requirements occurs, that is, the two A's are the lowest pair of nonterminals satisfying the above requirements.

First we show that a nonterminal A satisfying the above requirements exists. We prove this statement by contradiction. Suppose that for all nonterminals A that occur twice on a path of length at least $k+1$ both v and x, the border lists of the list vwx corresponding to the tree rooted at the upper occurrence of A, are both equal to ϵ. Then we can replace the tree rooted at the upper A by the tree rooted at the lower A without changing the list corresponding to the tree. Thus we can replace all paths of length at least $k+1$ by a path of length at most k. But this contradicts Lemma 3 below, and we have a contradiction. It follows that a nonterminal A satisfying the above requirements exists.

There exists a derivation tree for z in which the path from the upper A to the leaf has length at most $k+1$, since either below A no nonterminal occurs twice, or there is one or more nonterminal B that occurs twice, but the border lists v' and x' from the list $v'w'x'$ corresponding to the tree rooted at the upper

occurrence of nonterminal B are empty. Since the lists v' and x' are empty, we can replace the tree rooted at the upper occurrence of B by the tree rooted at the lower occurrence of B without changing the list corresponding to the derivation tree. Since we can do this for all nonterminals B that occur twice below the upper occurrence of A, there exists a derivation tree for z in which the path from the upper A to the leaf has length at most $k+1$. It follows from Lemma 3 below that the length of vwx is at most m^{k+1}, so we define $d = m^{k+1}$.

Suppose $z = uvwxy$, that is, the list corresponding to the subtree to the left (right) of the upper occurrence of A is u (y). This situation is depicted as follows.



We prove by induction that $(\forall i : i \geq 0 : uv^i wx^i y \in L)$. In this proof we apply the tree substitution process described in the example before the lemma. For $i = 0$ we have to show that the list uwy is a sentence in L. The list uwy is obtained if the tree rooted at the upper A is replaced by the tree rooted at the lower A. Suppose that for all $i \leq n$ we have $uv^i wx^i y \in L$. The list $uv^{i+1} wx^{i+1} y$ is obtained if the tree rooted at the lower A in the derivation tree for $uv^i wx^i y \in L$ is replaced by the tree rooted above it A. This proves the induction step.
□

The proof of the Pumping Lemma above frequently refers to the following lemma.

Theorem 3:

Let G be a context-free grammar, and suppose that the longest right-hand side of any production has length m. Let t be a derivation tree for a sentence $z \in L(G)$. If *height* $t \leq j$, then $|z| \leq m^j$. □

Proof: We prove a slightly stronger result: if t is a derivation tree for a list z, but the root of t is not necessarily the start-symbol, and *height* $t \leq j$, then $|z| \leq m^j$. We prove this statement by induction on j.

For the base case, suppose $j = 1$. Then tree t corresponds to a single production in G, and since the longest right-hand side of any production has length m, we have that $|z| \leq m = m^j$.

For the induction step, assume that for all derivation trees t of height at most j we have that $|z| \leq m^j$, where z is the list corresponding to t. Suppose we

have a tree t of height $j{+}1$. Let A be the root of t, and suppose the top of the tree corresponds to the production $A \to v$ in G. For all trees s rooted at the symbols of v we have *height* $s \leq j$, so the induction hypothesis applies to these trees, and the lists corresponding to the trees rooted at the symbols of v all have length at most m^j. Since $A \to v$ is a production of G, and since the longest right-hand side of any production has length m, the list corresponding to the tree rooted at A has length at most $m \times m^j = m^{j+1}$, which proves the induction step. □

SUMMARY

This section introduces pumping lemmas. Pumping lemmas are used to prove that languages are not regular or not context-free.

9.5 Exercises

Exercise 9.10 ▷ Show that the following language is not regular.

$$\{x \mid x \in \{\mathsf{0},\mathsf{1}\}^* \ \wedge \ nr\ \mathsf{1}\ x = nr\ \mathsf{0}\ x\}$$

where function nr takes an element a and a list x, and returns the number of occurrences of a in x. ◁

Exercise 9.11 ▷ Consider the following language:

$$\{\mathsf{a}^i\mathsf{b}^j \mid 0 \leq i \leq j\}$$

1. Is this language context-free? If it is, give a context-free grammar and prove that this grammar generates the language. If it is not, why not?

2. Is this language regular? If it is, give a regular grammar and prove that this grammar generates the language. If it is not, why not?

◁

Exercise 9.12 ▷ Consider the following language:

$$\{w\mathsf{c}w \mid w \in \{\mathsf{a},\mathsf{b}\}^*\}$$

1. Is this language context-free? If it is, give a context-free grammar and prove that this grammar generates the language. If it is not, why not?

2. Is this language regular? If it is, give a regular grammar and prove that this grammar generates the language. If it is not, why not?

◁

Exercise 9.13 ▷ Consider the grammar G with the following productions.

$$\begin{cases} S & \to & \{A\} \\ S & \to & \epsilon \\ A & \to & S \\ A & \to & AA \\ A & \to & \}\{ \end{cases}$$

1. Is this grammar
   - Context-free?
   - Regular?

   Why?
2. Give the language of G without referring to G itself.
3. Is the language of G
   - Context-free?
   - Regular?

   Why?

◁

Exercise 9.14 ▷ Consider the grammar G with the following productions.

$$\begin{cases} S & \to & \epsilon \\ S & \to & \mathsf{0} \\ S & \to & \mathsf{1} \\ S & \to & S\mathsf{0} \end{cases}$$

1. Is this grammar
   - Context-free?
   - Regular?

   Why?
2. Give the language of G without referring to G itself.
3. Is the language of G
   - Context-free?
   - Regular?

   Why?

◁

Chapter 10

LL Parsing

INTRODUCTION

This chapter introduces LL(1) parsing. LL(1) parsing is an efficient (linear in the length of the input string) method for parsing that can be used for all LL(1) grammars. A grammar is LL(1) if at each step in a derivation the next symbol in the input uniquely determines the production that should be applied. In order to determine whether or not a grammar is LL(1), we introduce several kinds of grammar analyses, such as determining whether or not a nonterminal can derive the empty string, and determining the set of symbols that can appear as the first symbol in a derivation from a nonterminal.

GOALS

After studying this chapter you will

- know the definition of LL(1) grammars;
- know how to parse a sentence from an LL(1) grammar;
- be able to apply different kinds of grammar analyses in order to determine whether or not a grammar is LL(1).

This chapter is organised as follows. Section 10.1 describes the background of LL(1) parsing, and Section 10.2 describes an implementation in Haskell of LL(1) parsing and the different kinds of grammar analyses needed for checking whether or not a grammar is LL(1).

10.1 LL Parsing: Background

In the previous chapters we have shown how to construct parsers for sentences of context-free languages using combinator parsers. Since these parsers may backtrack, the resulting parsers are sometimes a bit slow. There are several ways in which we can put extra restrictions on context-free grammars such that we can parse sentences of the corresponding languages efficiently. This

chapter discusses one such restriction: $LL(1)$. Other restrictions, not discussed in these lecture notes are $LR(1)$, $LALR(1)$, $SLR(1)$, etc.

10.1.1 A stack machine for parsing

This section presents a stack machine for parsing sentences of context-free grammars. We will use this machine in the following subsections to illustrate why we need grammar analysis.

The stack machine we use in this section differs from the stack machines introduced in Sections 6.4.4 and 8.2. A stack machine for a grammar G has a stack and an input, and performs one of the following two actions.

1. **Expand:** If the top stack symbol is a nonterminal, it is popped from the stack and a right-hand side from a production of G for the nonterminal is pushed onto the stack. The production is chosen nondeterministically.
2. **Match:** If the top stack symbol is a terminal, then it is popped from the stack and compared with the next symbol of the input sequence. If they are equal, then this terminal symbol is 'read'. If the stack symbol and the next input symbol do not match, the machine signals an error, and the input sentence cannot be accepted.

These actions are performed until the stack is empty. A stack machine for G accepts an input if it can terminate with an empty input when starting with the start-symbol from G on the stack.

For example, let grammar G be the grammar with the productions:

$$S \rightarrow \texttt{a}S \mid \texttt{c}S \mid \texttt{b}$$

The stack machine for G accepts the input string `aab` because it can perform the following actions (the first component of the state (before the |in the picture below) is the symbol stack and the second component of the state (after the |) is the unmatched (remaining part of the) input string):

stack	input
S	aab
aS	aab
S	ab
aS	ab
S	b
b	b

and end with an empty input. Note that an expand-action modifies the stack and that a match-action modifies the input.

However, if the machine had chosen the production $S \rightarrow \texttt{c}S$ in its first step, it would have been stuck. So not all possible sequences of actions from the *state* (S,`aab`) lead to an empty input. If there is at least one sequence of actions

that ends with an empty input on an input string, the input string is accepted. In this sense, the stack machine is similar to a nondeterministic finite-state automaton.

10.1.2 Some example derivations

This section gives three examples in which the stack machine for parsing is applied. It turns out that, for all three examples, the nondeterministic stack machine can act in a deterministic way by looking ahead one (or two) symbols of the sequence of input symbols. Each of the examples exemplifies why different kinds of grammar analyses are useful in parsing.

The first example

Our first example is `gramm1`. The set of terminal symbols of `gramm1` is $\{\texttt{a}, \texttt{b}, \texttt{c}\}$, the set of nonterminal symbols is $\{S, A, B, C\}$, the start symbol is S; and the productions are

$$\begin{array}{lcl} S & \rightarrow & \texttt{c}A \mid \texttt{b} \\ A & \rightarrow & \texttt{c}BC \mid \texttt{b}SA \mid \texttt{a} \\ B & \rightarrow & \texttt{cc} \mid C\texttt{b} \\ C & \rightarrow & \texttt{a}S \mid \texttt{ba} \end{array}$$

We want to know whether or not the string `ccccba` is a sentence of the language of this grammar. The stack machine produces, amongst others, the following sequence, corresponding with a leftmost derivation of `ccccba`.

stack	input
S	ccccba
cA	ccccba
A	cccba
cBC	cccba
BC	ccba
ccC	ccba
cC	cba
C	ba
ba	ba
a	a

Starting with S the machine chooses between two productions:

$$S \rightarrow \texttt{c}A \mid \texttt{b}$$

but, since the first symbol of the string `ccccba` to be recognised is `c`, the only applicable production is the first one. After expanding S a match-action removes the leading `c` from `ccccba` and `c`A. So now we have to derive the string `cccba` from A. The machine chooses between three productions:

$$A \rightarrow \texttt{c}BC \mid \texttt{b}SA \mid \texttt{a}$$

and again, since the next symbol of the remaining string `cccba` to be recognised is `c`, the only applicable production is the first one. After expanding A a match-action removes the leading `c` from `cccba` and `c`BC. So now we have to derive the string `ccba` from BC. The top stack symbol of BC is B. The machine chooses between two productions:

$$B \;\rightarrow\; \texttt{cc} \mid C\texttt{b}$$

The next symbol of the remaining string `ccba` to be recognised is, once again, `c`. The first production is applicable, but the second production may be applicable as well. To decide whether it also applies we have to determine the symbols that can appear as the first element of a string derived from B starting with the second production. The first symbol of the alternative C`b` is the nonterminal C. From the productions

$$C \;\rightarrow\; \texttt{a}S \mid \texttt{ba}$$

it is immediately clear that a string derived from C starts with either an `a` or a `b`. The set $\{\texttt{a}, \texttt{b}\}$ is called the *first* set of the nonterminal C. Since the next symbol in the remaining string to be recognised is a `c`, the second production cannot be applied. After expanding B and performing two match-actions it remains to derive the string `ba` from C. The machine chooses between two productions $C \rightarrow$ `a`S and $C \rightarrow$ `ba`. Clearly, only the second one applies, and, after two match-actions, leads to success.

From the above derivation we conclude the following.

> Deriving the sentence `ccccba` using `gramm1` is a deterministic computation: at each step of the derivation there is only one applicable alternative for the nonterminal on top of the stack.

Determinicity is obtained by looking at the set of firsts of the nonterminals.

The second example

A second example is the grammar `gramm2` whose productions are

$$S \;\rightarrow\; \texttt{ab}A \mid \texttt{aa}$$
$$A \;\rightarrow\; \texttt{bb} \mid \texttt{b}S$$

Now we want to know whether or not the string `abbb` is a sentence of the language of this grammar. The stack machine produces, amongst others, the following sequence

stack	input
S	abbb
abA	abbb
bA	bbb
A	bb
bb	bb
b	b

Starting with S the machine chooses between two productions:

$$S \rightarrow \texttt{ab}A \mid \texttt{aa}$$

since both alternatives start with an `a`, it is not sufficient to look at the first symbol `a` of the string to be recognised. The problem is that the *lookahead sets* (the `lookahead` *set* of a production $N \rightarrow \alpha$ is the set of terminal symbols that can appear as the first symbol of a string that can be derived from N starting with the production $N \rightarrow \alpha$, the definition is given in the following subsection) of the two productions for S both contain `a`. However, if we look at the first two symbols `ab`, then we find that the only applicable production is the first one. After expanding and matching it remains to derive the string `bb` from A. Again, looking ahead one symbol in the input string does not give sufficient information for choosing one of the two productions

$$A \rightarrow \texttt{bb} \mid \texttt{b}S$$

for A. If we look at the first two symbols `bb` of the input string, then we find that the first production applies (and, after matching, leads to success). Each string derived from A starting with the second production starts with a `b` and, since it is not possible to derive a string starting with another `b` from S, the second production does not apply.

From the above derivation we conclude the following.

> Deriving the string `abbb` using `gramm2` is a deterministic computation: at each step of the derivation there is only one applicable alternative for the nonterminal on the top of the stack.

Again, determinicity is obtained by analysing the set of firsts (of strings of length 2) of the nonterminals. Alternatively, we can left-factor the grammar to obtain a grammar in which all productions for a nonterminal start with a different terminal symbol.

The third example

A third example is grammar `gramm3` with the following productions:

$$\begin{aligned} S &\rightarrow A\texttt{a}S \mid B \\ A &\rightarrow \texttt{c}S \mid \epsilon \\ B &\rightarrow \texttt{b} \end{aligned}$$

Now we want to know whether or not the string `acbab` is an element of the language of this grammar. The stack machine produces the following sequence

stack	input
S	`acbab`
AaS	`acbab`
aS	`acbab`

Starting with S the machine chooses between two productions:

$$S \;\rightarrow\; A\mathtt{a}S \mid B$$

since each nonempty string derived from A starts with a `c`, and each nonempty string derived from B starts with a `b`, there does not seem to be a candidate production to start a leftmost derivation of `acabb` with. However, since A can also derive the empty string, we can apply the first production, and then apply the empty string for A, producing `a`S which, as required, starts with an `a`. We do not explain the rest of the leftmost derivation since it does not use any empty strings any more.

Nonterminal symbols that can derive the empty sequence will play a central role in the grammar analysis problems which we will consider in section 10.2.

From the above derivation we conclude the following.

> Deriving the string `acbab` using `gramm3` is a deterministic computation: at each step of the derivation there is only one applicable alternative for the nonterminal on the top of the stack.

Determinicity is obtained by analysing whether or not nonterminals can derive the empty string, and which terminal symbols can follow upon a nonterminal in a derivation.

Exercise 10.1 ▷ Give all the states of the stack machine for a derivation of `acbab` with `gramm3`. ◁

10.1.3 $LL(1)$ grammars

The examples in the previous subsection show that the derivations of the example sentences are deterministic, provided we can look ahead one or two symbols in the input. An obvious question now is: for which grammars are all derivations deterministic? Of course, as the second example shows, the answer to this question depends on the number of symbols we are allowed to look ahead. In the rest of this chapter we assume that we may look 1 symbol ahead. A grammar for which all derivations are deterministic with 1 symbol lookahead is called *LL(1)*: *L*eftmost with a *L*ookahead of *1*. Since all derivations of sentences of LL(1) grammars are deterministic, LL(1) is a desirable property of grammars.

To formalise this definition, we define `lookAhead` sets.

Definition 1: `lookAhead` **set**
The `lookahead` *set* of a production $N \rightarrow \alpha$ is the set of terminal symbols that can appear as the first symbol of a string that can be derived from $N\delta$ (where $N\delta$ appears as a tail substring in a derivation from the start-symbol) starting with the production $N \rightarrow \alpha$. So

$$\mathtt{lookAhead}\ (N \rightarrow \alpha) \;=\; \{\mathtt{x} \mid S \stackrel{*}{\Rightarrow} \gamma N\delta \Rightarrow \gamma\alpha\delta \stackrel{*}{\Rightarrow} \gamma\mathtt{x}\beta\}$$

□

For example, for the productions of `gramm1` we have

$$\begin{array}{lcl}
\texttt{lookAhead}\ (S \to \texttt{c}A) & = & \{\texttt{c}\} \\
\texttt{lookAhead}\ (S \to \texttt{b}) & = & \{\texttt{b}\} \\
\texttt{lookAhead}\ (A \to \texttt{c}BC) & = & \{\texttt{c}\} \\
\texttt{lookAhead}\ (A \to \texttt{b}SA) & = & \{\texttt{b}\} \\
\texttt{lookAhead}\ (A \to \texttt{a}) & = & \{\texttt{a}\} \\
\texttt{lookAhead}\ (B \to \texttt{cc}) & = & \{\texttt{c}\} \\
\texttt{lookAhead}\ (B \to C\texttt{b}) & = & \{\texttt{a}, \texttt{b}\} \\
\texttt{lookAhead}\ (C \to \texttt{a}S) & = & \{\texttt{a}\} \\
\texttt{lookAhead}\ (C \to \texttt{ba}) & = & \{\texttt{b}\}
\end{array}$$

We use `lookAhead` sets in the definition of LL(1) grammar.

Definition 2: LL(1) grammar

A grammar G is *LL(1)* if all pairs of productions of the same nonterminal have disjoint lookahead sets, that is: for all productions $N \to \alpha$, $N \to \beta$ of G:

$$\texttt{lookAhead}\ (N \to \alpha) \cap \texttt{lookAhead}\ (N \to \beta) \quad = \quad \emptyset$$

□

Since all `lookAhead` sets for productions of the same nonterminal of `gramm1` are disjoint, `gramm1` is an LL(1) grammar. For `gramm2` we have:

$$\begin{array}{lcl}
\texttt{lookAhead}\ (S \to \texttt{ab}A) & = & \{\texttt{a}\} \\
\texttt{lookAhead}\ (S \to \texttt{aa}) & = & \{\texttt{a}\} \\
\texttt{lookAhead}\ (A \to \texttt{bb}) & = & \{\texttt{b}\} \\
\texttt{lookAhead}\ (A \to \texttt{b}S) & = & \{\texttt{b}\}
\end{array}$$

Here, the `lookAhead` sets for both nonterminals S and A are not disjoint, and it follows that `gramm2` is not LL(1). `gramm2` *is* an LL(2) grammar, where an LL(k) grammar for $k \geq 2$ is defined similarly to an LL(1) grammar: instead of one symbol lookahead we have k symbols lookahead.

How do we determine whether or not a grammar is LL(1)? Clearly, to answer this question we need to know the lookahead sets of the productions of the grammar. The `lookAhead` set of a production $N \to \alpha$, where α starts with a terminal symbol `x`, is simply `x`. But what if α starts with a nonterminal P, that is $\alpha = P\beta$, for some β? Then we have to determine the set of terminal symbols with which strings derived from P can start. But if P can derive the empty string, we also have to determine the set of terminal symbols with which a string derived from β can start. As you see, in order to determine the `lookAhead` sets of productions, we are interested in

- whether or not a nonterminal can derive the empty string (`empty`);
- which terminal symbols can appear as the first symbol in a string derived from a nonterminal (`firsts`);
- and which terminal symbols can follow upon a nonterminal in a derivation (`follow`).

In each of the following definitions we assume that a grammar G is given.

Definition 3: Empty

Function `empty` takes a nonterminal N, and determines whether or not the empty string can be derived from the nonterminal:

$$\texttt{empty}\ N \;=\; N \stackrel{*}{\Rightarrow} \epsilon$$

□

For example, for `gramm3` we have:

$$\begin{aligned}\texttt{empty}\ S &= \texttt{False}\\ \texttt{empty}\ A &= \texttt{True}\\ \texttt{empty}\ B &= \texttt{False}\end{aligned}$$

Definition 4: First

The *set of firsts* of a nonterminal N is the set of terminal symbols that can appear as the first symbol of a string that can be derived from N:

$$\texttt{firsts}\ N \;=\; \{\texttt{x} \mid N \stackrel{*}{\Rightarrow} \texttt{x}\beta\}$$

□

For example, for `gramm3` we have:

$$\begin{aligned}\texttt{firsts}\ S &= \{\texttt{a}, \texttt{b}, \texttt{c}\}\\ \texttt{firsts}\ A &= \{\texttt{c}\}\\ \texttt{firsts}\ B &= \{\texttt{b}\}\end{aligned}$$

We could have given more restricted definitions of `empty` and `firsts`, by only looking at derivations from the start-symbol, for example,

$$\texttt{empty}\ N \;=\; S \stackrel{*}{\Rightarrow} \alpha N \beta \stackrel{*}{\Rightarrow} \alpha\beta$$

but the simpler definition above suffices for our purposes.

Definition 5: Follow

The *follow set* of a nonterminal N is the set of terminal symbols that can follow on N in a derivation starting with the start-symbol S from the grammar G:

$$\texttt{follow}\ N \;=\; \{\texttt{x} \mid S \stackrel{*}{\Rightarrow} \alpha N \texttt{x} \beta\}$$

□

For example, for `gramm3` we have:

$$\begin{aligned} \texttt{follow}\ S &= \{\texttt{a}\} \\ \texttt{follow}\ A &= \{\texttt{a}\} \\ \texttt{follow}\ B &= \{\texttt{a}\} \end{aligned}$$

In the following section we will give programs with which `lookahead`, `empty`, `firsts`, and `follow` are computed.

Exercise 10.2 ▷ Give the results of the function `empty` for the grammars `gramm1` and `gramm2`. ◁

Exercise 10.3 ▷ Give the results of the function `firsts` for the grammars `gramm1` and `gramm2`. ◁

Exercise 10.4 ▷ Give the results of the function `follow` for the grammars `gramm1` and `gramm2`. ◁

Exercise 10.5 ▷ Give the results of the function `lookahead` for grammar `gramm3`. Is `gramm3` an LL(1) grammar ? ◁

Exercise 10.6 ▷ Grammar `gramm2` is not LL(1), but it can be transformed into an LL(1) grammar by left factoring. Give this equivalent grammar `gramm2'` and give the results of the functions `empty`, `first`, `follow` and `lookAhead` on this grammar. Is `gramm2'` an LL(1) grammar? ◁

Exercise 10.7 ▷ A non-leftrecursive grammar for `Bit-Lists` is given by the following grammar (see your answer to exercise 2.21):

$$\begin{aligned} L &\rightarrow BR \\ R &\rightarrow \epsilon \mid ,BR \\ B &\rightarrow \texttt{0} \mid \texttt{1} \end{aligned}$$

Give the results of functions `empty`, `firsts`, `follow` and `lookAhead` on this grammar. Is this grammar LL(1)? ◁

10.2 LL Parsing: Implementation

Until now we have written parsers with parser combinators. Parser combinators use backtracking, and this is sometimes a cause of inefficiency. If a grammar is LL(1) we do not need backtracking anymore: parsing is deterministic. We can use this fact by either adjusting the parser combinators so that they don't use backtracking anymore, or by writing a special purpose LL(1) parsing program. We present the latter in this section.

This section describes the implementation of a program that parses sentences of LL(1) grammars. The program works for arbitrary context-free LL(1) grammars, so we first describe how to represent context-free grammars in Haskell. Another consequence of the fact that the program parses sentences of arbitrary

context-free LL(1) grammars is that we need a generic representation of parse trees in Haskell. The second subsection defines a datatype for parse trees in Haskell. The third subsection presents the program that parses sentences of LL(1) grammars. This program assumes that the input grammar is LL(1), so in the fourth subsection we give a function that determines whether or not a grammar is LL(1). Both this and the LL(1) parsing function use a function that determines the lookahead of a production. This function is presented in the fifth subsection. The last subsections of this section define functions for determining the empty, first, and follow symbols of a nonterminal.

10.2.1 Context-free grammars in Haskell

A context-free grammar may be represented by a pair: its start symbol, and its productions. How do we represent terminal and nonterminal symbols? There are at least two possibilities.

- The rigorous approach uses a datatype `Symbol`:

  ```
  data Symbol a b = N a | T b
  ```

  The advantage of this approach is that nonterminals and terminals are strictly separated, the disadvantage is the notational overhead of constructors that has to be carried around. However, a rigorous implementation of context-free grammars should keep terminals and nonterminals apart, so this is the preferred implementation. But in this section we will use the following implementation:

- ```
  class  Eq s => Symbol s  where
    isT  ::  s -> Bool
    isN  ::  s -> Bool
    isT  =   not . isN
  ```

  where `isN` and `isT` determine whether or not a symbol is a nonterminal or a terminal, respectively. This notation is compact, but terminals and nonterminals are no longer strictly separated, and symbols that are used as nonterminals cannot be used as terminals anymore. For example, the type of characters is made an instance of this class by defining:

  ```
  instance  Symbol Char  where
    isN c  =  'A' <= c && c <= 'Z'
  ```

  that is, capitals are nonterminals, and, by definition, all other characters are terminals.

A context-free grammar is a value of the type `CFG`:

```
type CFG s  =  (s,[(s,[s])])
```
```

where the list in the second component associates nonterminals to right-hand sides. So an element of this list is a production. For example, the grammar with productions

$$\begin{aligned} S &\to A\texttt{a}S \mid B \mid CB \\ A &\to SC \mid \epsilon \\ B &\to A \mid \texttt{b} \\ C &\to D \\ D &\to \texttt{d} \end{aligned}$$

is represented as:

```
exGrammar  :: CFG Char
exGrammar  =
  ('S', [('S',"AaS"),('S',"B"),('S',"CB")
        ,('A',"SC"),('A',"")
        ,('B',"A"),('B',"b")
        ,('C',"D")
        ,('D',"d")
        ]
  )
```

On this type we define some functions for extracting the productions, nonterminals, terminals, etc. from a grammar.

```
start  :: CFG s -> s
start  =  fst

prods  :: CFG s -> [(s,[s])]
prods  =  snd

terminals  :: (Symbol s, Ord s) => CFG s -> [s]
terminals  =  unions . map (filter isT . snd) . snd
```

where `unions :: Ord s => [[s]] -> [s]` returns the union of the 'sets' of symbols in the lists.

```
nonterminals  :: (Symbol s, Ord s) => CFG s -> [s]
nonterminals  =  nub . map fst . snd
```

Here, `nub :: Ord s => [s] -> [s]` removes duplicates from a list.

```
symbols          :: (Symbol s, Ord s) => CFG s -> [s]
symbols grammar  =
  union (terminals grammar) (nonterminals grammar)

nt2prods  ::  Eq s => CFG s -> s -> [(s,[s])]
nt2prods grammar s  =
  filter (\(nt,rhs) -> nt==s) (prods grammar)
```

where function `union` returns the set union of two lists (removing duplicates). For example, we have

```
?start exGrammar
'S'

? terminals exGrammar
"abd"
```

10.2.2 Parse trees in Haskell

A parse tree is a tree, in which each internal node is labelled with a nonterminal, and has a list of children (corresponding with a right-hand side of a production of the nonterminal). It follows that parse trees can be represented as rose trees with symbols, where the datatype of rose trees is defined by:

```
data Rose a = Node a [Rose a] | Nil
```

Recall that a rose tree is a tree where each node has a list of children.
The constructor `Nil` has been added to simplify error handling when parsing: when a sentence cannot be parsed, the 'parse tree' `Nil` is returned. Strictly speaking it should not occur in the datatype of rose trees.

Exercise 10.8 ▷ Give the `Rose` tree representation of the parse tree corresponding to the derivation of the sentence `ccccba` using grammar `gramm1`. ◁

Exercise 10.9 ▷ Give the `Rose` tree representation of the parse tree corresponding to the derivation of the sentence `abbb` using grammar `gramm2'` defined in the exercises of the previous section. ◁

Exercise 10.10 ▷ Give the `Rose` tree representation of the parse tree corresponding to the derivation of the sentence `acbab` using grammar `gramm3`. ◁

10.2.3 LL(1) parsing

This subsection defines a function `ll1` that takes a grammar and a terminal string as input, and returns one tuple: a parse tree, and the rest of the input-string that has not been parsed. So `ll1` takes a grammar, and returns a parser with `Rose s` as its result type. As mentioned in the beginning of this section, our parser doesn't need backtracking anymore, since parsing with an LL(1) grammar is deterministic. Therefore, the parser type is adjusted as follows:

```
type Parser b a = [b] -> (a,[b])
```

Using this parser type, the type of the function `ll1` is:

```
ll1 :: (Symbol s, Ord s) => CFG s -> Parser s (Rose s)
```

Function `ll1` is defined in terms of two functions. Function `isll1 :: CFG s -> Bool` is a function that checks whether or not a grammar is LL(1). And function `gll1` (for *generalised* LL(1)) produces a *list* of rose trees for a *list* of symbols. `ll1` is obtained from `gll1` by giving `gll1` the singleton list containing the start symbol of the grammar as argument.

```
ll1 grammar input =
  if isll1 grammar
  then let ([rose], rest) = gll1 grammar [start grammar] input
       in (rose, rest)
  else error "ll1: grammar not LL(1)"
```

Exercise 10.11 ▷ Describe the result of applying `ll1` to `gramm1` and the sentence `ccccba`. ◁

So now we have to implement functions `isll1` and `gll1`. Function `isll1` is implemented in the following subsection. Function `gll1` also uses two functions. Function `grammar2ll1table` takes a grammar and returns the LL(1) table: the association list that associates productions with their lookahead sets. And function `choose` takes a terminal symbol, and chooses a production based on the LL(1) table.

```
gll1 :: (Symbol s, Ord s) => CFG s -> [s] -> Parser s [Rose s]
gll1 grammar =
  let ll1table = grammar2ll1table grammar
        -- The LL(1) table.
      nt2prods nt = filter (\((n,l),r) -> n==nt) ll1table
        -- nt2prods returns the productions for nonterminal
        -- nt from the LL(1) table
      selectprod nt t = choose t (nt2prods nt)
        -- selectprod selects the production for nt from the
        -- LL(1) table that should be taken when t is the next
        -- symbol in the input.
  in \stack input ->
       case stack of
         []     -> ([], input)
         (s:ss) ->
           if isT s
           then -- match
                let (rts,rest) = gll1 grammar ss (tail input)
                in if s == head input
                   then (Node s []: rts, rest)
                     -- The parse tree is a leaf (a node with
                     -- no children).
                   else ([Nil], input)
                     -- The input cannot be parsed
```

```
else -- expand
     let t = head input
         (rts,zs) = gll1 grammar (selectprod s t) input
           -- Try to parse according to the production
           -- obtained from the LL(1) table from s.
         (rrs,vs) = gll1 grammar ss zs
           -- Parse the rest of the symbols on the
           -- stack.
     in ((Node s rts): rrs, vs)
```

Exercise 10.12 ▷

1. Function `gll1` takes two arguments. The first argument is a grammar. What is the second argument and what is the result?
2. Describe the given implementation of function `gll1`.

◁

Functions `grammar2ll1table` and `choose`, which are used in the above function `gll1`, are defined as follows. These functions use function `lookaheadp`, which returns the lookahead set of a production and is defined in one of the following subsections.

```
grammar2ll1table :: (Symbol s, Ord s) => CFG s -> [((s,[s]),[s])]
grammar2ll1table grammar  =
  map (\x -> (x,lookaheadp grammar x)) (prods grammar)

choose      :: Eq a => a -> [((b,c),[a])] -> c
choose t l  =
  let [((s,rhs), ys)] = filter (\((x,p),q) -> t `elem` q) l
  in rhs
```

Note that function `choose` assumes that there is exactly one element in the association list in which the element `t` occurs.

10.2.4 Implementation of isLL(1)

Function `isll1` checks whether or not a context-free grammar is LL(1). It does this by computing for each nonterminal of its argument grammar the set of lookahead sets (one set for each production of the nonterminal), and checking that all of these sets are disjoint. It is defined in terms of a function `lookaheadn`, which computes the lookahead sets of a nonterminal, and a function `disjoint`, which determines whether or not all sets in a list of sets are disjoint. All sets in a list of sets are disjoint if the length of the concatenation of these sets equals the length of the union of these sets.

```
isll1          :: (Symbol s, Ord s) => CFG s -> Bool
isll1 grammar  =
  and (map (disjoint . lookaheadn grammar) (nonterminals grammar))

disjoint      :: Ord s => [[s]] -> Bool
disjoint xss  =  length (concat xss) == length (unions xss)
```

Function `lookaheadn` computes the lookahead sets of a nonterminal by computing all productions of a nonterminal, and computing the lookahead set of each of these productions by means of function `lookaheadp`.

```
lookaheadn          :: (Symbol s, Ord s) => CFG s -> s -> [[s]]
lookaheadn grammar  =
  map (lookaheadp grammar) . nt2prods grammar
```

10.2.5 Implementation of lookahead

Function `lookaheadp` takes a grammar and a production, and returns the lookahead set of the production. It is defined in terms of four functions. Each of the first three functions will be defined in a separate subsection below, the fourth function is defined in this subsection.

- `isEmpty :: (Ord s,Symbol s) => CFG s -> s -> Bool`

  Function `isEmpty` takes a grammar and a nonterminal and determines whether or not the empty string can be derived from the nonterminal in the grammar. (This function was called `empty` in Definition 3.)

- `firsts :: (Ord s, Symbol s) => CFG s -> [(s,[s])]`

  Function `firsts` takes a grammar and computes the set of firsts of each symbol (the set of firsts of a terminal is the terminal itself).

- `follow :: (Ord s, Symbol s) => CFG s -> [(s,[s])]`

  Function `follow` takes a grammar and computes the follow set of each nonterminal (so it associates a list of symbols with each nonterminal).

- ```
  lookSet :: Ord s       =>
             (s -> Bool) ->  -- isEmpty
             (s -> [s])  ->  -- firsts?
             (s -> [s])  ->  -- follow?
             (s, [s])    ->  -- production
             [s]             -- lookahead set
  ```

  Note that we use the operator `?`, see Section 6.4.2, on the `firsts` and `follow` association lists. Function `lookSet` takes a predicate, two functions that given a nonterminal return the first and follow set, respectively, and a production, and returns the lookahead set of the production. Function `lookSet` is introduced after the definition of function `lookaheadp`.

Now we define:
  ```

```
lookaheadp :: (Symbol s, Ord s) => CFG s -> (s,[s]) -> [s]
lookaheadp grammar  =
  lookSet (isEmpty grammar) ((firsts grammar)?) ((follow grammar)?)
```

We will exemplify the definition of function `lookSet` with the grammar `exGrammar`, with the following productions:

$$
\begin{aligned}
S &\rightarrow A\mathtt{a}S \mid B \mid CB \\
A &\rightarrow SC \mid \epsilon \\
B &\rightarrow A \mid \mathtt{b} \\
C &\rightarrow D \\
D &\rightarrow \mathtt{d}
\end{aligned}
$$

Consider the production $S \rightarrow A\mathtt{a}S$. The lookahead set of the production contains the set of symbols which can appear as the first terminal symbol of a sequence of symbols derived from A. But, since the nonterminal symbol A can derive the empty string, the lookahead set also contains the symbol `a`.

Consider the production $A \rightarrow SC$. The lookahead set of the production contains the set of symbols which can appear as the first terminal symbol of a sequence of symbols derived from S. But, since the nonterminal symbol S can derive the empty string, the lookahead set also contains the set of symbols which can appear as the first terminal symbol of a sequence of symbols derived from C.

Finally, consider the production $B \rightarrow A$. The lookahead set of the production contains the set of symbols which can appear as the first terminal symbol of a sequence of symbols derived from A. But, since the nonterminal symbol A can derive the empty string, the lookahead set also contains the set of terminal symbols which can follow the nonterminal symbol B in some derivation.

The examples show that it is useful to have functions `firsts` and `follow` in which, for every nonterminal symbol `n`, we can look up the terminal symbols which can appear as the first terminal symbol of a sequence of symbols in some derivation from `n` and the set of terminal symbols which can follow the nonterminal symbol `n` in a sequence of symbols occurring in some derivation respectively. It turns out that the definition of function `follow` also makes use of a function `lasts` which is similar to the function `firsts`, but which deals with last nonterminal symbols rather than first terminal ones.

The examples also illustrate a control structure which will be used very often in the following algorithms: we will fold over right-hand sides. While doing so we compute sets of symbols for all the symbols of the right-hand side which we encounter and collect them into a final set of symbols. Whenever such a list for a symbol is computed, there are always two possibilities:

- either we continue folding and return the result of taking the union of the set obtained from the current element and the set obtained by recursively folding over the rest of the right-hand side
- or we stop folding and immediately return the set obtained from the current element.

We continue if the current symbol is a nonterminal which can derive the empty sequence and we stop if the current symbol is either a terminal symbol or a nonterminal symbol which cannot derive the empty sequence. The following function makes this statement more precise.

```
foldrRhs :: Ord s =>
            (s -> Bool) ->
            (s -> [s]) ->
            [s] ->
            [s] ->
            [s]
foldrRhs p f start = foldr op start
  where op x xs = f x `union` if p x then xs else []
```

The function `foldrRhs` is, of course, most naturally defined in terms of the function `foldr`. This function is somewhere in between a general purpose and an application specific function (we could easily have made it more general though). In the exercises we give an alternative characterisation of `foldRhs`. We will also need a function `scanrRhs` which is like `foldrRhs` but accumulates intermediate results in a list. The function `scanrRhs` is most naturally defined in terms of the function `scanr`.

```
scanrRhs :: Ord s =>
            (s -> Bool) ->
            (s -> [s]) ->
            [s] ->
            [s] ->
            [[s]]
scanrRhs p f start = scanr op start
  where op x xs = f x `union` if p x then xs else []
```

Finally, we will also need a function `scanlRhs` which does the same job as `scanrRhs` but in the opposite direction. The easiest way to define `scanlRhs` is in terms of `scanrRhs` and `reverse`.

```
scanlRhs p f start = reverse . scanrRhs p f start . reverse
```

We now return to the function `lookSet`.

```
lookSet :: Ord s =>
           (s -> Bool) ->
           (s -> [s]) ->
           (s -> [s]) ->
           (s,[s]) ->
           [s]
lookSet p f g (nt,rhs)  =  foldrRhs p f (g nt) rhs
```

The function `lookSet` makes use of `foldrRhs` to fold over a right-hand side. As stated above, the function `foldrRhs` continues processing a right-hand side

only if it encounters a nonterminal symbol for which `p` (so `isEmpty` in the `lookSet` instance `lookaheadp`) holds. Thus, the set `g nt` (`follow?nt` in the `lookSet` instance `lookaheadp`) is only important for those right-hand sides for `nt` that consist of nonterminals that can all derive the empty sequence. We can now (assuming that the definitions of the auxiliary functions are given) use the function `lookaheadp` instance of `lookSet` to compute the lookahead sets of all productions.

```
look nt rhs = lookaheadp exGrammar (nt,rhs)

? look 'S' "AaS"
dba
? look 'S' "B"
dba
? look 'S' "CB"
d
? look 'A' "SC"
dba
? look 'A' ""
ad
? look 'B' "A"
dba
? look 'B' "b"
b
? look 'C' "D"
d
? look 'D' "d"
d
```

It is clear from this result that `exGrammar` is not an LL(1)-grammar. Let us have a closer look at how these lookahead sets are obtained. We will have to use the functions `firsts` and `follow` and the predicate `isEmpty` for computing intermediate results. The corresponding subsections explain how to compute these intermediate results.

For the lookahead set of the production $A \to A\mathtt{a}S$ we fold over the right-hand side $A\mathtt{a}S$. Folding stops at `'a'` and we obtain

```
 firsts? 'A' `union` firsts? 'a'
==
 "dba" `union` "a"
==
 "dba"
```

For the lookahead set of the production $A \to SC$ we fold over the right-hand side SC. Folding stops at C since it cannot derive the empty sequence, and we obtain

```
 firsts? 'S' `union` firsts? 'C'
```

```
==
 "dba" `union` "d"
==
 "dba"
```

Finally, for the lookahead set of the production $B \to A$ we fold over the right-hand side A In this case we fold over the complete (one element) list and and we obtain

```
 firsts? 'A' `union` follow? 'B'
==
 "dba" `union` "d"
==
 "dba"
```

The other lookahead sets are computed in a similar way.

10.2.6 Implementation of empty

Many functions defined in this chapter make use of a predicate `isEmpty`, which tests whether or not the empty sequence can be derived from a nonterminal. This subsection defines this function. Consider the grammar `exGrammar`. We are now only interested in deriving sequences which contain only nonterminal symbols (since it is impossible to derive the empty string if a terminal occurs). Therefore we only have to consider the productions in which no terminal symbols appear in the right-hand sides.

$$\begin{array}{lcl} S & \to & B \mid CB \\ A & \to & SC \mid \epsilon \\ B & \to & A \\ C & \to & D \end{array}$$

One can immediately see from those productions that the nonterminal A derives the empty string in one step. To know whether there are any nonterminals which derive the empty string in more than one step we eliminate the productions for A and we eliminate all occurrences of A in the right hand sides of the remaining productions

$$\begin{array}{lcl} S & \to & B \mid CB \\ B & \to & \epsilon \\ C & \to & D \end{array}$$

One can now conclude that the nonterminal B derives the empty string in two steps. Doing the same with B as we did with A gives us the following productions

$$\begin{array}{lcl} S & \to & \epsilon \mid C \\ C & \to & D \end{array}$$

One can now conclude that the nonterminal S derives the empty string in three steps. Doing the same with S as we did with A and B gives us the following productions

$$C \rightarrow D$$

At this stage we can conclude that there are no more new nonterminals which derive the empty string.

We now give the Haskell implementation of the algorithm described above. The algorithm is iterative: it does the same steps over and over again until some desired condition is met. For this purpose we use function `fixedPoint`, which takes a function and a set, and repeatedly applies the function to the set, until the set does not change anymore.

```
fixedPoint       :: Ord a => ([a] -> [a]) -> [a] -> [a]
fixedPoint f xs  |  xs == nexts  =  xs
                 |  otherwise    =  fixedPoint f nexts
  where nexts = f xs
```

`fixedPoint f` is sometimes called the *fixed-point* of `f`. Function `isEmpty` determines whether or not a nonterminal can derive the empty string. A nonterminal can derive the empty string if it is a member of the `emptySet` of a grammar.

```
isEmpty         :: (Symbol s, Ord s) => CFG s -> s -> Bool
isEmpty grammar =  (`elem` emptySet grammar)
```

The `emptySet` of a grammar is obtained by the iterative process described in the example above. We start with the empty set of nonterminals, and at each step n of the computation of the `emptySet` as a `fixedPoint`, we add the nonterminals that can derive the empty string in n steps. Function `emptyStepf` adds a nonterminal if there exists a production for the nonterminal of which all elements can derive the empty string.

```
emptySet :: (Symbol s, Ord s) => CFG s -> [s]
emptySet grammar = fixedPoint (emptyStepf grammar) []

emptyStepf :: (Symbol s, Ord s) => CFG s -> [s] -> [s]
emptyStepf grammar set =
  nub (map fst (filter (\(nt,rhs) -> all (`elem` set) rhs)
                       (prods grammar)
      )        )
```

10.2.7 Implementation of first and last

Function `firsts` takes a grammar, and returns for each symbol of the grammar (so also the terminal symbols) the set of terminal symbols with which a sentence derived from that symbol can start. The first set of a terminal symbol is the terminal symbol itself.

The set of firsts each symbol consists of that symbol itself, plus the (first) symbols that can be derived from that symbol in one or more steps. So the set of firsts can be computed by an iterative process, just as the function `isEmpty`. Consider the grammar `exGrammar` again. We start the iteration with

```
[('S',"S"),('A',"A"),('B',"B"),('C',"C"),('D',"D")
,('a',"a"),('b',"b"),('d',"d")
]
```

Using the productions of the grammar we can derive in one step the following lists of first symbols.

```
[('S',"ABC"),('A',"S"),('B',"Ab"),('C',"D"),('D',"d")]
```

and the union of these two lists is

```
[('S',"SABC"),('A',"AS"),('B',"BAb"),('C',"CD"),('D',"Dd")
,('a',"a"),('b',"b"),('d',"d")]
```

In *two* steps we can derive

```
[('S',"SAbD"),('A',"ABC"),('B',"S"),('C',"d"),('D',"")]
```

and again we have to take the union of this list with the previous result. We repeat this process until the list doesn't change anymore. For `exGrammar` this happens when:

```
[('S',"SABCDabd")
,('A',"SABCDabd")
,('B',"SABCDabd")
,('C',"CDd")
,('D',"Dd")
,('a',"a")
,('b',"b")
,('d',"d")
]
```

Function `firsts` is defined as the `fixedPoint` of a step function that iterates the first computation one more step. The `fixedPoint` starts with the list that contains all symbols paired with themselves.

```
firsts  ::  (Symbol s, Ord s) => CFG s -> [(s,[s])]
firsts grammar  =
  fixedPoint (firstStepf grammar) (startSingle grammar)

startSingle :: (Ord s, Symbol s) => CFG s -> [(s,[s])]
startSingle grammar  =  map (\x -> (x,[x])) (symbols grammar)
```

The step function takes the old approximation and performs one more iteration step. At each of these iteration steps we have to add the start list with which the iteration started again.

```
firstStepf :: (Ord s, Symbol s) =>
              CFG s -> [(s,[s])] -> [(s,[s])]
firstStepf grammar approx  =     (startSingle grammar)
                       `combine` (compose (first1 grammar) approx)

combine :: Ord s => [(s,[s])] -> [(s,[s])] -> [(s,[s])]
combine xs = foldr insert xs
  where insert (a,bs) [] = [(a,bs)]
        insert (a,bs) ((c,ds):rest)
          | a == c     =  (a, union bs ds) : rest
          | otherwise  =  (c,ds) : (insert (a,bs) rest)

compose :: Ord a => [(a,[a])] -> [(a,[a])] -> [(a,[a])]
compose r1 r2 =  [(a, unions (map (r2?) bs)) | (a,bs) <- r1]
```

Finally, function `first1` computes the direct first symbols of all productions, taking into account that some nonterminals can derive the empty string, and combines the results for the different nonterminals.

```
first1 :: (Symbol s, Ord s) => CFG s -> [(s,[s])]
first1 grammar  =
  map (\(nt,fs) -> (nt,unions fs))
      (group (map (\(nt,rhs) -> (nt,foldrRhs (isEmpty grammar)
                                             single
                                             []
                                             rhs
                  )             )
                  (prods grammar)
      )      )
```

where `group` groups elements with the same first element together

```
group  :: Eq a => [(a,b)] -> [(a,[b])]
group  =  foldr insertPair []

insertPair :: Eq a => (a,b) -> [(a,[b])] -> [(a,[b])]
insertPair (a,b) []            = [(a,[b])]
insertPair (a,b) ((c,ds):rest) =
  if a==c then (c,(b:ds)):rest else (c,ds):(insertPair (a,b) rest)
```

function `single` takes an element and returns the set with the element, and `unions` returns the union of a set of sets.

Function `lasts` is defined using function `firsts`. Suppose we reverse the right-hand sides of all productions of a grammar. Then the set of firsts of this reversed

grammar is the set of lasts of the original grammar. This idea is implemented in the following functions.

```
reverseGrammar  :: Symbol s => CFG s -> CFG s
reverseGrammar  =
  \(s,al) -> (s,map (\(nt,rhs) -> (nt,reverse rhs)) al)

lasts  :: (Symbol s, Ord s) => CFG s -> [(s,[s])]
lasts  =  firsts . reverseGrammar
```

10.2.8 Implementation of follow

The final function we have to implement is the function `follow`, which takes a grammar, and returns an association list in which nonterminals are associated to symbols that can follow upon the nonterminal in a derivation. A nonterminal `n` is associated to a list containing terminal `t` in `follow` if `n` and `t` follow each other in some sequence of symbols occurring in some leftmost derivation. We can compute pairs of such adjacent symbols by splitting up the right-hand sides with length at least 2 and, using `lasts` and `firsts`, compute the symbols which appear at the end resp. at the beginning of strings which can be derived from the left resp. right part of the split alternative. Our grammar `exGrammar` has three alternatives with length at least 2: `"AaS"`, `"CB"` and `"SC"`. Function `follow` uses the functions `firsts` and `lasts` and the predicate `isEmpty` for intermediate results. The previous subsections explain how to compute these functions.

Let's see what happens with the alternative `"AaS"`. The lists of all nonterminal symbols that can appear at the end of sequences of symbols derived from `"A"` and `"Aa"` are `"ADC"` and `""` respectively. The lists of all terminal symbols which can appear at the beginning of sequences of symbols derived from `"aS"` and `"S"` are `"a"` and `"dba"` respectively. Zipping together those lists shows that an `'A'`, a `'D'` and a `'C'` can be be followed by an `'a'`. Splitting the alternative `"CB"` in the middle produces sets of firsts and sets of lasts `"CD"` and `"dba"`. Splitting the alternative `"SC"` in the middle produces sets of firsts and sets of lasts `"SDCAB"` and `"d"`. From the first pair we can see that a `'C'` and a `'D'` can be followed by a `'d'`, a `'b'` and an `'a'` From the second pair we see that an `'S'`, a `'D'`, a `'C'`, an `'A'`, and a `'B'` can be followed by a `'d'`. Combining all these results gives:

```
[('S',"d"),('A',"ad"),('B',"d"),('C',"adb"),('D',"adb")]
```

The function `follow` uses the functions `scanrAlt` and `scanlAlt`. The lists produced by these functions are exactly the ones we need: using the function `zip` from the standard prelude we can combine the lists. For example: for the alternative `"AaS"` the functions `scanlAlt` and `scanrAlt` produce the following lists:

```
[[], "ADC", "DC", "SDCAB"]
["dba", "a", "dba", []]
```

Only the two middle elements of both lists are important (they correspond to the nontrivial splittings of `"AaS"`). Thus, we only have to consider alternatives of length at least 2. We start the computation of `follow` with assigning the empty follow set to each symbol:

```
follow  :: (Symbol s, Ord s) => CFG s -> [(s,[s])]
follow grammar  =  combine (followNE grammar) (startEmpty grammar)

startEmpty grammar = map (\x -> (x,[])) (symbols grammar)
```

The real work is done by functions `followNE` and function `splitProds`. Function `followNE` passes the right arguments on to function `splitProds`, and removes all nonterminals from the set of firsts, and all terminals from the set of lasts. Function `splitProds` splits the productions of length at least 2, and pairs the last nonterminals with the first terminals.

```
followNE :: (Symbol s, Ord s) => CFG s -> [(s,[s])]
followNE grammar  =  splitProds
                       (prods       grammar)
                       (isEmpty     grammar)
                       (isTfirsts   grammar)
                       (isNlasts    grammar)
  where isTfirsts = map (\(x,xs) -> (x,filter isT xs)) . firsts
        isNlasts  = map (\(x,xs) -> (x,filter isN xs)) . lasts

splitProds :: (Symbol s, Ord s) =>
              [(s,[s])]         ->   -- productions
              (s -> Bool)       ->   -- isEmpty
              [(s,[s])]         ->   -- terminal firsts
              [(s,[s])]         ->   -- nonterminal lasts
              [(s,[s])]
splitProds prods p fset lset  =
  map (\(nt,rhs) -> (nt,nub rhs)) (group pairs)
  where  pairs = [(l, f)
                 | rhs <- map snd prods
                 , length rhs >= 2
                 , (fs, ls) <- zip (rightscan rhs) (leftscan rhs)
                 , l <- ls
                 , f <- fs
                 ]
         leftscan  = scanlRhs p (lset?) []
         rightscan = scanrRhs p (fset?) []
```

Exercise 10.13 ▷ In this exercise we will take a closer look at the functions `foldrRhs` and `scanrRhs` which are the essential ingredients of the implementation of the grammar analysis algorithms. From the definitions it is clear that grammar analysis is easily expressed via a calculus for (finite) sets. A calculus for finite

sets is implicit in the programs for LL(1) parsing. Since the code in this module is obscured by several implementation details we will derive the functions `foldrRhs` and `scanrRhs` in a stepwise fashion. In this derivation we will use the following:
A (finite) set is implemented by a list with no duplicates. In order to construct a set, the following operations may be used:

`[]`	::	`[a]`	the empty set of `a`-elements
`union`	::	`[a] → [a] → [a]`	the union of two sets
`unions`	::	`[[a]] → [a]`	the generalised union
`single`	::	`a → [a]`	the singleton function

These operations satisfy the well-known laws for set operations.

1. Define a function `list2Set :: [a] → [a]` which returns the set of elements occurring in the argument.

2. Define `list2Set` as a `foldr`.

3. Define a function `pref p :: [a] → [a]` which given a list `xs` returns the set of elements corresponding to the longest prefix of `xs` all of whose elements satisfy `p`.

4. Define a function `prefplus p :: [a] → [a]` which given a list `xs` returns the set of elements in the longest prefix of `xs` all of whose elements satisfy `p` together with the first element of `xs` that does not satisfy `p` (if this element exists at all).

5. Define `prefplus p` as a `foldr`.

6. Show that `prefplus p = foldrRhs p single []`.

7. It can be shown that

   ```
   foldrRhs p f [] = unions . map f . prefplus p
   ```

   for all set-valued functions `f`. Give an informal description of the functions `foldrRhs p f []` and `foldrRhs p f start`.

8. The standard function `scanr` is defined by

   ```
   scanr f q0 = map (foldr f q0) .  tails
   ```

   where `tails` is a function which takes a list `xs` and returns a list with all tailsegments (postfixes) of `xs` in decreasing length. The function `scanrRhs` is defined is a similar way

   ```
   scanrRhs p f start = map (foldrRhs p f start) .  tails
   ```

   Give an informal description of the function `scanrRhs`.

◁

Exercise 10.14 ▷ The computation of the functions `empty` and `firsts` is not restricted to nonterminals only. For terminal symbols `s` these functions are defined by

```
empty s  =  False
firsts s =  {s}
```

Using the definitions in the previous exercise, compute the following.

1. For the example grammar `gramm1` and two of its productions $A \to \mathtt{b}SA$ and $B \to C\mathtt{b}$.
   (a) `foldrRhs empty firsts []` $\mathtt{b}SA$
   (b) `foldrRhs empty firsts []` $C\mathtt{b}$
2. For the example grammar `gramm3` and its production $S \to A\mathtt{a}S$
   (a) `foldrRhs empty firsts []` $A\mathtt{a}S$
   (b) `scanrRhs empty firsts []` $A\mathtt{a}S$

◁

Chapter 11

LL versus LR parsing

The parsers that were constructed using parser combinators in chapter 3 are nondeterministic. They can recognize sentences described by ambiguous grammars by returning a *list of solutions*, rather than just one solution; each solution contains a parse tree and the part of the input that was left unprocessed. Nondeterministic parsers are of the following type:

```
type Parser a b  =  [a]  ->  [ (b, [a]) ]
```

where `a` denotes the alphabet type, and `b` denotes the parse tree type.

In chapter 10 we turned to *deterministic* parsers. Here, parsers have only one result, consisting of a parse tree of type `b` and the remaining part of the input string:

```
type Parser a b  =  [a]  ->  (b, [a])
```

Ambiguous grammars are not allowed anymore. Also, in the case of parsing input containing syntax errors, we cannot return an empty list of successes anymore; intead there should be some mechanism of raising an error.

There are various algorithms for deterministic parsing. They impose some additional constraints on the form of the grammar: not every context free grammar is allowable by these algorithms. The parsing algorithms can be modelled by making use of a *stack machine*. There are two fundamentally different deterministic parsing algorithms:

- LL parsing, also known as *top-down parsing*
- LR parsing, also known as *bottom-up parsing*

The first 'L' in these acronyms stands for 'Left-to-right', that is, input in processed in the order it is read. So these algorithms are both suitable for reading an input stream, e.g. from a file. The second 'L' in 'LL-parsing' stands for 'Leftmost derivation', as the parsing mimics doing a leftmost derivation of a sentence (see section 2.4). The 'R' in 'LR-parsing' stands for 'Rightmost derivation' of the sentences. The parsing algorithms are normally refered to as $LL(k)$ or ($LR(k)$, where k is the number of unread symbols that the parser is allowed to 'look ahead'. In most practical cases, k is taken to be 1.

In chapter 10 we studied the $LL(1)$ parsing algorithm extensively, including the so-called $LL(1)$-property to which grammars must abide. In this chapter we start with an example application of the $LL(1)$ algorithm. Next, we turn to the $LR(1)$ algorithm.

11.1 LL(1) parser example

The $LL(1)$ parsing algorithm was implemented in section 10.2.3. Function `ll1` defined there takes a grammar and an input string, and returns a single result consisting of a parse tree and rest string. The function was implemented by calling a generalized function named `gll1`; generalized in the sense that the function takes an additional list of symbols that need to be recognized. That generalization is then called with a singleton list containing only the root nonterminal.

11.1.1 An LL(1) checker

Here, we define a slightly simplified version of the algorithm: it doesn't return a parse tree, so it need not be concerned with building it; it merely *checks* whether or not the input string is a sentence. Hence the result of the function is simply a boolean value:

```
check :: String -> Bool
check input  =  run ['S'] input
```

As in chapter 10, the function is implemented by calling a generalized function `run` with a singleton containing just the root nonterminal. Now the function `run` takes, in addition to the input string, a list of symbols (which is initially called with the abovementioned singleton). That list is used as some kind of stack, as elements are prepended at its front, and removed at the front in other occasions. Therefore we refer to the whole operation as a *stack machine*, and that's why the function is named `run`: it *runs* the stack machine. Function `run` is defined as follows:

```
run  ::  Stack -> String -> Bool
run []     []     = True
run []     (x:xs) = False
run (a:as) input  | isT a  =     not(null input)
                                 && a==hd input
                                 && run      as (tl input)
                  | isN a  =     run (rs++as)    input
          where rs = select a (hd input)
```

So, when called with an empty stack and an empty string, the function succeeds. If the stack is empty, but the input is not, it fails, as there is junk input present. If the stack is nonempty, case distinction is done on `a`, the top of the stack. If it is a terminal, the input should begin with it, and the machine is called recursively for the rest of the input. In the case of a noterminal, we push `rs` on the stack, and leave the input unchanged in the recursive call. This is a simple tail recursive function, and could imperatively be implemented in a single loop that runs while the stack is non-empty.

The new symbols `rs` that are pushed on the stack is the right hand side of a production rule for nonterminal `a`. It is selected from the available alternatives by function `select`. For making the choice, the first input symbol is passed to `select` as well. This is where you can see that the algorithm is $LL(\mathbf{1})$: it is allowed to look ahead 1 symbol.

This is how the alternative is selected:

```
select a x  =  (snd . hd . filter ok . prods) gram
    where ok p@(n,_)  =  n==a && x `elem` lahP gram p
```

So, from all productions of the grammar returned by `prods`, the `ok` ones are taken, of which there should be only one (this is ensured by the $LL(1)$-property); that single production is retrieved by `hd`, and of it only the right hand side is needed, hence the call to `snd`. Now a production is `ok`, if the nonterminal `n` is `a`, the one we are looking for, and moreover the first input symbol `x` is member of the *lookahead set* of this production.

Determining the lookahead sets of all productions of a grammar is the tricky part of the LL(1) parsing algorithm. It is described in section 10.2.5–8. Though the general formulation of the algorithm is quite complex, its application in a simple case is rather intuitive. So let's study an example grammar: arithmetic expressions with operators of two levels of precedence.

11.1.2 An LL(1) grammar

The idea for the grammar for arithmetical expressions was given in section 2.5: we need auxiliary notions of 'Term' and 'Factor' (actually, we need as much additional notions as there are levels of precedence). The most straightforward definition of the grammar is shown in the left column below.

Unfortunately, this grammar doesn't abide to the LL(1)-property. The reason for this is that the grammar contains rules with common prefixes on the right hand side (for E and T). A way out is the application of a grammar transformation known as *left factoring*, as described in section 2.5. The result is a grammar where there is only one rule for E and T, and the non-common part of the right hand sides is described by additional nonterminals, P and M. The resulting grammar is shown in the right column below. For being able to treat end-of-input as if it were a character, we also add an additional rule, which says that the input consists of an expression followed by end-of-input (designated as '#' here).

$E \to T$	$S \to E\ \#$
$E \to T$ `+` E	$E \to TP$
$T \to F$	$P \to \epsilon$
$T \to F$ `*` T	$P \to$ `+` E
$F \to N$	$T \to FM$
$F \to$ `(` E `)`	$M \to \epsilon$
$N \to$ `1`	$M \to$ `*` T
$N \to$ `2`	$F \to N$
$N \to$ `3`	$F \to$ `(` E `)`
	$N \to$ `1`
	$N \to$ `2`
	$N \to$ `3`

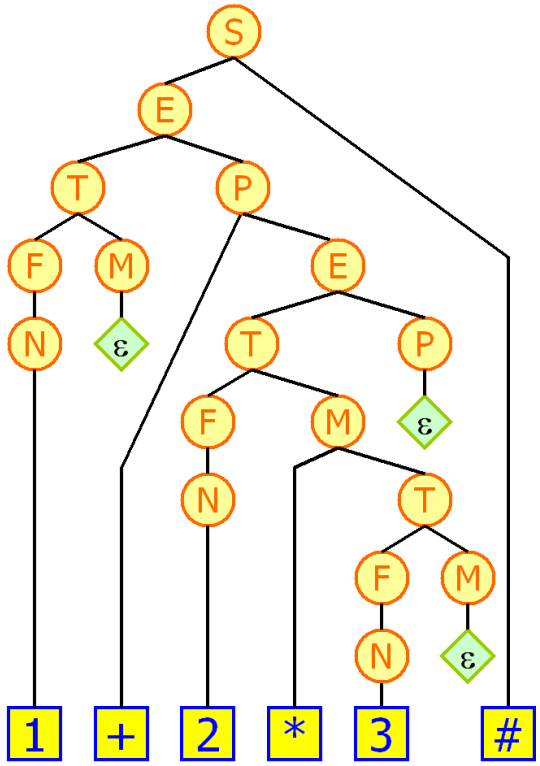
For determining the lookahead sets of each nonterminal, we also need to analyze whether a nonterminal can produce the empty string, and which terminals

can be the first symbol of any string produced by each nonterminal. These properties are named *empty* and *first* respecively. All properties for the example grammar are summarized in the table below. Note that *empty* and *first* are properties of a nonterminal, whereas *lookahead* is a property of a single production rule.

production	*empty*	*first*	*lookahead*	
$S \rightarrow E\ \#$	no	`( 1 2 3`	first(E)	`( 1 2 3`
$E \rightarrow TP$	no	`( 1 2 3`	first(T)	`( 1 2 3`
$P \rightarrow \epsilon$	yes	`+`	follow(P)	`) #`
$P \rightarrow$ `+` E			immediate	`+`
$T \rightarrow FM$	no	`( 1 2 3`	first(F)	`( 1 2 3`
$M \rightarrow \epsilon$	yes	`*`	follow(M)	`+ ) #`
$M \rightarrow$ `*` T			immediate	`*`
$F \rightarrow N$	no	`( 1 2 3`	first(N)	`1 2 3`
$F \rightarrow$ `(` E `)`			immediate	`(`
$N \rightarrow$ `1`	no	`1 2 3`	immediate	`1`
$N \rightarrow$ `2`			immediate	`2`
$N \rightarrow$ `3`			immediate	`3`

11.1.3 Using the LL(1) parser

A sentence of the language described by the example grammar is `1+2*3`. Because of the high precedence of `*` relative to `+`, the parse tree should reflect that the `2` and `3` should be multiplied, rather than the `1+2` and `3`. Indeed, the parse tree does so:



Now when we do a step-by-step analysis of how the parse tree is constructed by the stack machine, we notice that the parse tree is traversed in a depth-first fashion, where the left subtrees are analysed first. Each node corresponds to the application of a production rule. The order in which the production rules are applied is a pre-order traversal of the tree. The tree is constructed top-down: first the root is visited, an then each time the first remaining nonterminal is expanded. From the table in figure 11.1 it is clear that the contents of the stack describes what is still to be expected on the input. Initially, of course, this is the root nonterminal S. Whenever a terminal is on top of the stack, the corresponing symbol is read from the input.

11.2 LR parsing

Another algorithm for doing deterministic parsing using a stack machine, is $LR(1)$-parsing. Actually, what is described in this section is known as *Simple* LR parsing or $SLR(1)$-parsing. It is still rather complicated, though.

A nice property of LR-parsing is that it is in many ways exactly the opposite, or *dual*, of LL-parsing. Some of these ways are:

- LL does a leftmost derivation, LR does a rightmost derivation
- LL *starts* with only the root nonterminal on the stack, LR *ends* with only the root nonterminal on the stack
- LL *ends* when the stack is empty, LR *starts* with an empty stack
- LL uses the stack for designating what is still to be *expected*, LR uses the stack for designating what has already been *seen*
- LL builds the parse tree *top down*, LR builds the parse tree *bottom up*
- LL continuously pops a nonterminal off the stack, and pushes a corresponding right hand side; LR tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal
- LL thus *expands* nonterminals, while LR *reduces* them
- LL reads terminal when it pops one *off* the stack, LR reads terminals while it pushes them *on* the stack
- LL uses grammar rules in an order which corresponds to *pre-order* traversal of the parse tree, LR does a *post-order* traversal.

11.2.1 A stack machine for SLR parsing

As in Section 10.1.1 a stack machine is used to parse sentences. A stack machine for a grammar G has a stack and an input. When the parsing starts the stack is empty. The stack machine performs one of the following two actions.

1. **Shift:** Move the first symbol of the remaining input to the top of the stack.

2. **Reduce:** Choose a production rule $N \to \alpha$; pop the sequence α from the top of the stack; push N onto the stack.

These actions are performed until the stack only contains the start symbol. A stack machine for G accepts an input if it can terminate with only the start symbol on the stack when the whole input has been processed. Let us take the example of Section 10.1.1.

stack	input
	aab
a	ab
aa	b
baa	
Saa	
Sa	
S	

Note that with our notation the top of the stack is at the left side. To decide which production rule is to be used for the reduction, the symbols on the stack must be read in reverse order.
The stack machine for G accepts the input string `aab` because it terminates with the start symbol on the stack and the input is completely processed. The stack machine performs three shift actions and then three reduce actions.
The SLR parser only performs a reduction with a grammar rule $N \rightarrow \alpha$ if the next symbol on the input is in the follow set of N.

Exercise 11.1 ▷ Give for `gramm1` of Section 10.1.2 all the states of the stack machine for a derivation of the input `ccccba`. ◁

Exercise 11.2 ▷ Give for `gramm2` of Section 10.1.2 all the states of the stack machine for a derivation of the input `abbb`. ◁

Exercise 11.3 ▷ Give for `gramm3` of Section 10.1.2 all the states of the stack machine for a derivation of the input `acbab`. ◁

11.3 LR parse example

11.3.1 An LR checker

for a start, the main function for LR parsing calls a generalized stack function with an empty stack:

```
check' :: String -> Bool
check' input  =  run' [] input
```

The stack machine terminates when it finds the stack containing just the root nonterminal. Otherwise it either pushes the first input symbol on the stack ('Shift'), or it drops some symbols off the stack (which should be the right hand side of a rule) and pushes the corresponding nonterminal ('Reduce').

```
run'  :: Stack -> String -> Bool
run' ['S'] []     = True
run' ['S'] (x:xs) = False
run' stack (x:xs) = case action of
                      Shift      -> run' (x:       stack) xs
                      Reduce a n -> run' (a:drop n stack) (x:xs)
                      Error      -> False
            where action = select' stack x
```

In the case of LL-parsing the hard part was selecting the right rule to expand; here we have the hard decision of whether to reduce according to a (and which?) rule, or to shift the next symbol. This is done by the `select'` function, which is allowed to inspect the first input symbol `x` and the *entire* stack: after all, in needs to find the right hand side of a rule on the stack.

In the right column in figure 11.1 the LR derivation of sentence `1+2*3` is shown. Compare closely to the left column, which shows the LL derivation, and note the duality of the processes.

LL derivation

rule	read	↓stack	remaining
		S	`1+2*3`
$S \to E$		E	`1+2*3`
$E \to TP$		TP	`1+2*3`
$T \to FM$		FMP	`1+2*3`
$F \to N$		NMP	`1+2*3`
$N \to$ `1`		`1`MP	`1+2*3`
read	`1`	MP	`+2*3`
$M \to \epsilon$	`1`	P	`+2*3`
$P \to$ `+`E	`1`	`+`E	`+2*3`
read	`1+`	E	`2*3`
$E \to TP$	`1+`	TP	`2*3`
$T \to FM$	`1+`	FMP	`2*3`
$F \to N$	`1+`	NMP	`2*3`
$N \to$ `2`	`1+`	`2`MP	`2*3`
read	`1+2`	MP	`*3`
$M \to$ `*`T	`1+2`	`*`TP	`*3`
read	`1+2*`	TP	`3`
$T \to FM$	`1+2*`	FMP	`3`
$F \to N$	`1+2*`	NMP	`3`
$N \to$ `3`	`1+2*`	`3`MP	`3`
read	`1+2*3`	MP	
$M \to \epsilon$	`1+2*3`	P	
$P \to \epsilon$	`1+2*3`		

LR derivation

rule	read	stack↓	remaining
			`1+2*3`
shift	`1`	`1`	`+2*3`
$N \to$ `1`	`1`	N	`+2*3`
$F \to N$	`1`	F	`+2*3`
$M \to \epsilon$	`1`	FM	`+2*3`
$T \to FM$	`1`	T	`+2*3`
shift	`1+`	T`+`	`2*3`
shift	`1+2`	T`+2`	`*3`
$N \to$ `2`	`1+2`	T`+`N	`*3`
$F \to N$	`1+2`	T`+`F	`*3`
shift	`1+2*`	T`+`F`*`	`3`
shift	`1+2*3`	T`+`F`*3`	
$N \to$ `3`	`1+2*3`	T`+`F`*`N	
$F \to N$	`1+2*3`	T`+`F`*`F	
$M \to \epsilon$	`1+2*3`	T`+`F`*`FM	
$T \to FM$	`1+2*3`	T`+`F`*`T	
$M \to$ `*`T	`1+2*3`	T`+`FM	
$T \to FM$	`1+2*3`	T`+`T	
$P \to \epsilon$	`1+2*3`	T`+`TP	
$E \to TP$	`1+2*3`	T`+`E	
$P \to$ `+`E	`1+2*3`	TP	
$E \to TP$	`1+2*3`	E	
$S \to E$	`1+2*3`	S	

Figure 11.1: LL and LR derivations of `1+2*3`

11.3.2 LR action selection

The choice whether to shift or to reduce is made by function `select'`. It is defined as follows:

```
select' as x
   | null items    = Error
   | null redItems = Shift
   | otherwise     = Reduce a (length rs)
       where items    =  dfa as
             redItems =  filter red items
             (a,rs,_) =  hd redItems
```

In the selection process a set (list) of so-called *items* plays a role. If the set is empty, there is an error. If the set contains at least one item, we filter the `red`, or *reducible* items from it. There should be only one, if the grammar has the LR-property. (Or rather: it is the LR-property that there is only one element in this situation). The reducible item is the production rule that can be reduced by the stack machine.

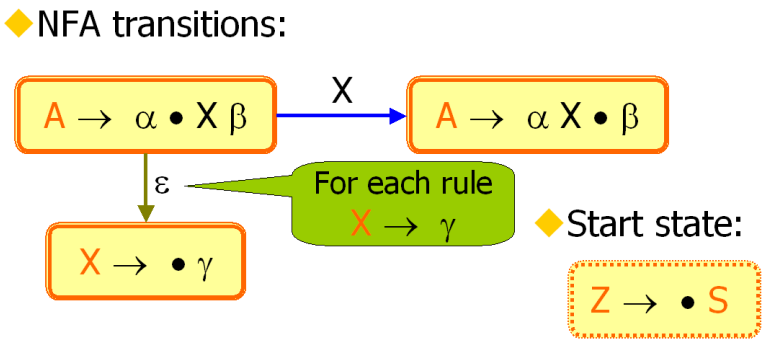
Now what are these items? An *item* is defined to be a production rule, augmented with a 'cursor position' somewhere in the right hand side of the rule. So for the rule $F \rightarrow$ `(`E`)`, there are four possible items: $F \rightarrow \cdot$`(`E`)`, $F \rightarrow$ `(`$\cdot E$`)`, $F \rightarrow$ `(`$E \cdot$ `)` and $F \rightarrow$ `(`E`)`$\cdot$, where $\cdot$ denotes the cursor.

for the rule $F \rightarrow$ `2` we have two items: one having the cursor in front of the single symbol, and one with the cursor after it: $F \rightarrow \cdot$ `2` and $F \rightarrow$ `2` $\cdot$. For epsilon-rules there is a single item, where the cursor is at position 0 in the right hand side.

In the Haskell representation, the cursor is an integer which is added as a third element of the tupel, which already contains nonterminal and right hand side of the rule.

The items thus constructed are taken to be the states of a NFA (Nondeterministic Finite-state Automaton), as described in section 5.1.2. We have the following transition relations in this NFA:

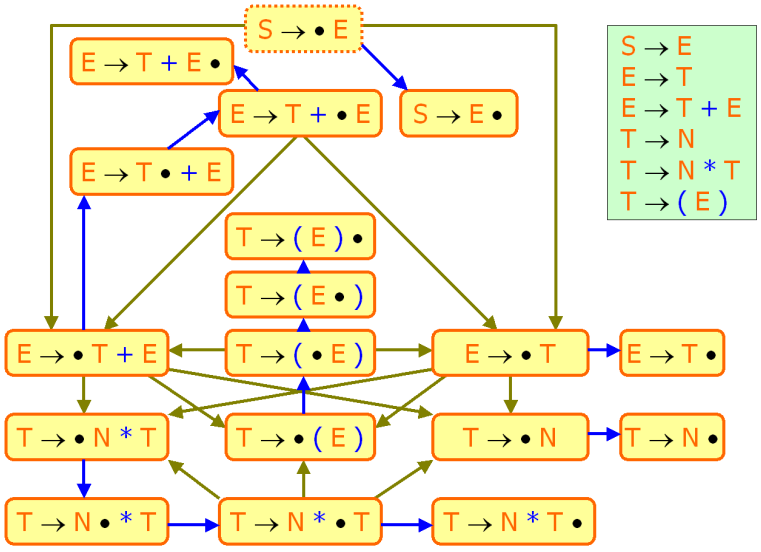
- The cursor can be 'stepped' to the next position. This transition is labeled with the symbol that is hopped over
- If the cursor is on front of a nonterminal, we can jump to an item describing the application of that nonterminal, where the cursor is at the beginning. This relation is an epsilon-transition of the NFA.



As an example, let's consider an simplification of the arithmetic expression grammar:

$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow N$
$T \rightarrow N * T$
$T \rightarrow (E)$

it skips the 'factor' notion as compared to the grammar earlier in this chapter, so it misses some well-formed expressions, but it serves only as an example for creating the states here. There are 18 states in this machine, as depicted here:



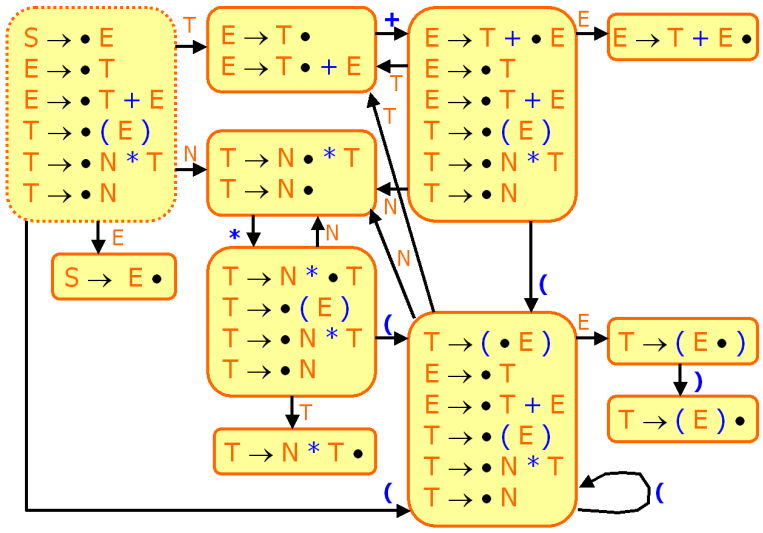
Exercise 11.4 ▷ How can you predict the number of states from inspecting the grammar? ◁

Exercise 11.5 ▷ In the picture, the transition arrow labels are not shown. Add them. ◁

Exercise 11.6 ▷ What makes this FA nondeterministic? ◁

As was shown in section 5.1.4, another automaton can be constructed that is deterministic (a DFA). That construction involves defining states which are *sets*

of the original states. So in this case, the states of the DFA are *sets of items* (where items are rules with a cursor position). In the worst case we would have 2^{18} states for the DFA-version of our example NFA, but it turns out that we are in luck: there are only 11 states in the DFA. Its states are rather complicated to depict, as they are sets of items, but it can be done:



Exercise 11.7 ▷ Check that this FA is indeed deterministic. ◁

Given this DFA, let's return to our function that selects whether to shift or to reduce:

```
select' as x
   | null items    = Error
   | null redItems = Shift
   | otherwise     = Reduce a (length rs)
       where items    =  dfa as
             redItems =  filter red items
             (a,rs,_) =  hd redItems
```

It runs the DFA, *using the contents of the stack (*`as`*) as input.* From the state where the DFA ends, which is by construction a set of items, the 'red' ones are filtered out. An item is 'red' (that is: reducible) if the cursor is at the end of the rule.

Exercise 11.8 ▷ Which of the states in the picture of the DFA contain red items? How many? ◁

This is not the only condition that makes an item reducible; the second condition is that the lookahead input symbol is in the *follow* set of the nonterminal that is reduced to. Function `follow` was also needed in the LL analysis in section 10.2.8. Both conditions are in the formal definition:

```
... where red (a,r,c)  =  c==length r   &&  x `elem` follow a
```

Exercise 11.9 ▷ How does this condition reflect that 'the cursor is at the end'? ◁

11.3.3 LR optimizations and generalizations

The stack machine `run'`, by its nature, pushes and pops the stack continuously, and does a recursive call afterwards. In each call, for making the shift/reduce decision, the DFA is run on the (new) stack. In practice, it is considered a waste of time to do the full DFA transitions each time, as most of the stack remains the same after some popping and pushing at the top. Therefore, as an optimization, at each stack position, the corresponding DFA state is also stored. The states of the DFA can easily be numbered, so this amounts to just storing extra integers on the stack, tupled eith the symbols that used to be on the stack. (An artificial bottom element should be placed on the stack initially, containing a dummy symbol and the number of the initial DFA state).

By analysing the grammar, two tables can be precomputed:

- Shift, that decides what is the new state when pushing a terminal symbol on the stack. This basically is the transition relation on the DFA.
- Action, that decides what action to take from a given state seeing a given input symbol.

Both tables can be implemented as a two-dimensional table of integers, of dimensions the number of states (typically, 100s to 1000s) times the number of symbols (typically, under 100).

As said in the introduction, the algorithm described here is a mere *Simple* LR parsing, or SLR(1). Its simplicity is in the reducibility test, which says:

```
... where red (a,r,c)  =  c==length r   &&  x `elem` follow a
```

The *follow* set is a rough approximation of what might follow a given nonterminal. But this set is not dependent of the context in which the nonterminal is used; maybe, in some contexts, the set is smaller. So, the SLR `red` function, may designate items as reducuble, where it actually should not. For some grammars this might lead to a decision to reduce, where it should have done a shift, with a failing parser as a consequence.

An improvement, leading to a wider class of grammars that are allowable, would be to make the *follow* set context dependent. This means that it should vary for each *item* instead of for each *nonterminal*. It leads to a dramatic increase of the number of states in the DFA. And the full power of LR parsing is sledomly needed.

A compromise position is taken by the so-called *LALR* parsers, or *Look Ahead LR* parsing. (A rather silly name, as *all* parsers look ahead...). In LALR parsers, the follow sets are context dependent, but when states in the DFA differ only with respect to the follow-part of their set-members (and not with respect to the item-part of them), the states are merged. Grammars that do not give rise to shift/reduce conflicts in this situation are said to be LALR-grammars. It is not really a natural notion; rather, it is a performance hack that gets most of LR power while keeping the size of the goto- and action-tables reasonable.

A widely used parser generator tool named *yacc* (for 'yet another compiler compiler') is based on an LALR engine. It comes with Unix, and was originally created for implementing the first C compilers. A commonly used clone of yacc is named *Bison*. Yacc is designed for doing the context-free aspects of analysing a language. The micro structure of identifiers, numbers, keywords, comments etc. is not handled by this grammar. Instead, it is described by regular expressions, which are analysed by a accompagnying tool to yacc named *lex* (for 'lexical scanner'). Lex is a preprocessor to yacc, that subdivides the input character stream into a stream of meaningful *tokens*, such as numbers, identifiers, operators, keywords etc.

Bibliographical notes

The example DFA and NFA for LR parsing, and part of the description of the algorithm were taken from course notes by Alex Aiken and George Necula, which can be found at `www.cs.wright.edu/~tkprasad/courses/cs780`

Bibliography

[1] A.V. Aho, Sethi R., and J.D. Ullman. *Compilers — Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] R.S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[3] R.S. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[4] W.H. Burge. Parsing. In *Recursive Programming Techniques*. Addison-Wesley, 1975.

[5] J. Fokker. Functional parsers. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[6] R. Harper. Proof-directed debugging. *Journal of Functional Programming*, 1999. To appear.

[7] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323 – 343, 1992.

[8] G. Hutton and E. Meijer. Monadic parsing in haskell. *Journal of Functional Programming*, 8(4):437 – 444, 1998.

[9] B.W. Kernighan and R. Pike. Regular expressions — languages, algorithms, and software. *Dr. Dobb's Journal*, April:19 – 22, 1999.

[10] D.E. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127–145, 1968.

[11] Niklas Röjemo. *Garbage collection and memory efficiency in lazy functional languages*. PhD thesis, Chalmers University of Technology, 1995.

[12] S. Sippu and E. Soisalon-Soininen. *Parsing Theory, Vol. 1: Languages and Parsing*, volume 15 of *EATCS Monographs on THeoretical Computer Science*. Springer-Verlag, 1988.

[13] S.D. Swierstra and P.R. Azero Alcocer. Fast, error correcting parser combinators: a short tutorial. In *SOFSEM'99*, 1999.

[14] P. Wadler. How to replace failure by a list of successes: a method for exception handling, backtracking, and pattern matching in lazy functional languages. In J.P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 113 – 128. Springer, 1985. LNCS 201.

[15] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*. Springer, 1995. LNCS 925.

Appendix A

The Stack module

```
module Stack
 ( Stack
 , emptyStack
 , isEmptyStack
 , push
 , pushList
 , pop
 , popList
 , top
 , split
 , mystack
 )

where

data Stack x =  MkS [x] deriving (Show,Eq)

emptyStack :: Stack x
emptyStack = MkS []

isEmptyStack :: Stack x -> Bool
isEmptyStack (MkS xs) = null xs

push :: x -> Stack x -> Stack x
push x (MkS xs) = MkS (x:xs)

pushList :: [x] -> Stack x -> Stack x
pushList xs (MkS ys) = MkS (xs ++ ys)


pop :: Stack x -> Stack x
pop (MkS xs) = if isEmptyStack (MkS xs)
               then error "pop on emptyStack"
               else MkS (tail xs)
```

```
popIf :: Eq x => x -> Stack x -> Stack x
popIf x stack = if top stack == x
                then pop stack
                else error "argument and top of stack don't match"

popList :: Eq x => [x] -> Stack x -> Stack x
popList xs stack = foldr popIf stack (reverse xs)


top :: Stack x -> x
top (MkS xs) = if isEmptyStack (MkS xs)
               then error "top on emptyStack"
               else head xs

split :: Int -> Stack x -> ([x], Stack x)
split 0 stack = ([], stack)
split n (MkS []) = error "attempt to split the emptystack"
split (n+1) (MkS (x:xs)) = (x:ys, stack')
                           where
                           (ys, stack') = split n (MkS xs)

mystack = MkS [1,2,3,4,5,6,7]
```

Appendix B

Answers to exercises

2.1 Three of the fours strings are elements of L^*: abaabaaabaa, aaaabaaaa, baaaaabaa.

2.2 $\{\epsilon\}$.

2.3

$$
\begin{array}{ll}
 & \emptyset\, L \\
= & \quad \text{Definition of concatenation of languages} \\
 & \{st \mid s \in \emptyset, t \in L\} \\
= & \quad s \in \emptyset \\
 & \emptyset
\end{array}
$$

and the other equalities are proved in a similar fashion.

2.4 L^n should satisfy:

$$
\begin{array}{rcl}
L^0 & = & ? \\
L^{n+1} & = & L^n L
\end{array}
$$

It follows that we want L^0 to satisfy

$$L^0 L = L$$

If we choose $L^0 = \{\epsilon\}$, we have

$$
\begin{array}{ll}
 & L^0 L \\
= & \quad \text{Definition of concatenation of languages} \\
 & \{st \mid s \in \{\epsilon\}, t \in L\} \\
= & \quad \text{Definition of string concatenation} \\
 & \{t \mid t \in L\} \\
= & \quad \text{Definition of set comprehension} \\
 & L
\end{array}
$$

as desired.

2.5 The star operator on sets injects the elements of a set in a list; the star operator on languages concatenates the sentences of the language. The former star operator preserves more structure.

2.6 Section 2.1 contains an inductive definition of the set of sequences over an arbitrary set T. Syntactical definitions for such sets follow immediately from this.

1. A grammar for $T = \{\mathsf{a}\}$ is given by

$$\begin{array}{lcl} S & \rightarrow & \epsilon \\ S & \rightarrow & \mathsf{a}S \end{array}$$

2. A grammar for $T = \{\mathsf{a}, \mathsf{b}\}$ is given by

$$\begin{array}{lcl} S & \rightarrow & \epsilon \\ S & \rightarrow & XS \\ X & \rightarrow & \mathsf{a} \mid \mathsf{b} \end{array}$$

2.7 A context free grammar for L is given by

$$\begin{array}{lcl} S & \rightarrow & \epsilon \\ S & \rightarrow & \mathsf{a}S\mathsf{b} \end{array}$$

2.8 Analogous to the construction of the grammar for PAL .

$$P \rightarrow \epsilon \mid \mathsf{a} \mid \mathsf{b} \mid \mathsf{a}P\mathsf{a} \mid \mathsf{b}P\mathsf{b}$$

2.9 Analogous to the construction of PAL.

$$M \rightarrow \epsilon \mid \mathsf{a}M\mathsf{a} \mid \mathsf{b}M\mathsf{b}$$

2.10 First establish an inductive definition for *parity-sequences*. An example of a grammar that can be derived from the inductive definition is:

$$P \rightarrow \epsilon \mid \mathsf{1}P\mathsf{1} \mid \mathsf{0}P \mid P\mathsf{0}$$

There are many other solutions.

2.11 Again, establish an inductive definition for L. An example of a grammar that can be derived from the inductive definition is:

$$S \rightarrow \epsilon \mid \mathsf{a}S\mathsf{b} \mid \mathsf{b}S\mathsf{a} \mid SS$$

This grammar has four production rules. The first one says that the empty string belongs to the language. The second and the third one say that the string obtained by prepending and appending a different symbol to a string of the language also belongs to the language. The last production rules is for strings with identical first and last symbol. In this case a string that belongs to the language can be split into two parts with each a different first and last symbol.

Again, there are many other solutions.

2.12 A sentence is a sentential form consisting only of terminals which can be derived in *zero* or more derivation steps from the start-symbol (to be more precise: the sentential form consisting only of the start-symbol). The start-symbol is a nonterminal. The nonterminals of a grammar do not belong to the alphabet (the set of terminals) of the language we describe using the grammar. Therefore the start-symbol cannot be a sentence of the language. As a consequence we have to perform at least *one* derivation step from the start-symbol before we end up with a sentence of the language.

2.13 The language consisting of the empty string only, i.e. $\{\epsilon\}$.

2.14 This grammar generates the empty language, i.e. $\emptyset$. The grammar cannot produce any sentence with only terminal symbols because there is no production rule without nonterminals. Therefore each derivation will always contain nonterminals.

2.15 The sentences in this language consist of zero or more concatenations of `ab`.

2.16 Yes. Each finite language is context free. A context free grammar can be obtained by taking one nonterminal and adding a production rule for each sentence in the language. For the language in exercise 2.1 this yields

$$\begin{aligned} S &\rightarrow \texttt{ab} \\ S &\rightarrow \texttt{aa} \\ S &\rightarrow \texttt{baa} \end{aligned}$$

2.17 Draw a tree yourself from the following derivations: $P \stackrel{*}{\Rightarrow} \texttt{a}P\texttt{a} \stackrel{*}{\Rightarrow} \texttt{ab}P\texttt{ba} \stackrel{*}{\Rightarrow} \texttt{aba}P\texttt{aba} \stackrel{*}{\Rightarrow} \texttt{abaaba}$ and $P \stackrel{*}{\Rightarrow} \texttt{b}P\texttt{b} \stackrel{*}{\Rightarrow} \texttt{ba}P\texttt{ab} \stackrel{*}{\Rightarrow} \texttt{baaab}$. The datatype `Palc` can be defined by:

```
data Palc = Empty | A Char | B Char | A2 Char Palc Char | B2 Char Palc Char
```

The two example palindromes are represented by the following values:

```
cPal1 = A2 'a' (B2 'b' (A2 'a' Empty 'a') 'b') 'a'
cPal2 = B2 'b' (A2 'a' (A 'a') 'a') 'b')
```

Note that here too the datatype is too general. The datatype allows characters different from `a` and `b`, and in the constructors `A2` and `B2` we can use different characters at the left-hand and right-hand sides.

2.18 The equivalent grammar after transformation is

$$\begin{aligned} A &\rightarrow \texttt{ba}A \\ A &\rightarrow \texttt{b} \end{aligned}$$

2.19 In the following grammar power raise has the highest priority:

$$\begin{aligned} E &\rightarrow T \mid E \texttt{ + } T \mid E \texttt{ - } T \\ T &\rightarrow F \mid T \texttt{ * } F \mid T \texttt{ / } F \\ F &\rightarrow P \mid F \texttt{ ^ } P \\ P &\rightarrow \texttt{(}E\texttt{)} \mid \mathit{Digs} \end{aligned}$$

2.20

1 $S \stackrel{*}{\Rightarrow}$ if b then S else $S \stackrel{*}{\Rightarrow}$ if b then if b then S else $S \stackrel{*}{\Rightarrow}$ if b then if b then a else $S \stackrel{*}{\Rightarrow}$ if b then if b then a else a, and $S \stackrel{*}{\Rightarrow}$ if b then $S \stackrel{*}{\Rightarrow}$ if b then if b then S else $S \stackrel{*}{\Rightarrow}$ if b then if b then a else $S \stackrel{*}{\Rightarrow}$ if b then if b then a else a

2 The rule we apply is: match else with the closest previous unmatched then. This disambiguating rule is incorporated directly into the grammar:

$$\begin{aligned} S &\rightarrow \mathit{MatchedS} \mid \mathit{UnmatchedS} \\ \mathit{MatchedS} &\rightarrow \text{if b then } \mathit{MatchedS} \text{ else } \mathit{MatchedS} \mid \text{a} \\ \mathit{UnmatchedS} &\rightarrow \text{if b then } S \mid \text{if b then } \mathit{MatchedS} \text{ else } \mathit{UnmatchedS} \end{aligned}$$

3 An else clause is always matched with the closest previous unmatched if.

2.21 An equivalent grammar for `Bit-Lists` is

$$\begin{aligned} L &\rightarrow BZ \mid B \\ Z &\rightarrow ,LZ \mid ,L \\ B &\rightarrow 0 \mid 1 \end{aligned}$$

2.22

1 This grammar generates $\{ \text{a}^{2n}\text{b}^m \mid n \geq 0, m \geq 0\}$

2 An equivalent non left recursive grammar is

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow \epsilon \mid \text{aa}A \\ B &\rightarrow \epsilon \mid \text{b}B \end{aligned}$$

2.23 SequenceOfS

```
sa32string :: SA3 -> String
sa32string (Size i) = replicate i 's'
```

A session in which `sa32string` is used may look like this:

```
? sa32string (Size 5)
"sssss"
```

2.24

$$\mathit{Expr} \rightarrow \mathit{Expr}+\mathit{Expr} \mid \mathit{Expr} \times \mathit{Expr} \mid \mathit{Int}$$

where *Int* is a nonterminal that produces integers.

2.25 Palindromes

1 The following datatype describes the structure of palindromes. `Pal1` represents the empty string and `Pal2` and `Pal3` represent the characters `a` and `b`, respectively. `Pal4` represents a palindrome that begins and ends with an `a` and `Pal5` represents a palindrome that begins and ends with a `b`.

```
data Pal = Pal1 | Pal2 | Pal3 | Pal4 Pal | Pal5 Pal
aPal1 = Pal4 (Pal5 (Pal4 Pal1))
aPal2 = Pal5 (Pal4 Pal2)
```

```
2   a2cPal :: Pal -> String
    a2cPal Pal1 = ""
    a2cPal Pal2 = "a"
    a2cPal Pal3 = "b"
    a2cPal (Pal4 p) = "a" ++ a2cPal p ++ "a"
    a2cPal (Pal5 p) = "b" ++ a2cPal p ++ "b"
3   aCountPal :: Pal -> Int
    aCountPal Pal1 = 0
    aCountPal Pal2 = 1
    aCountPal Pal3 = 0
    aCountPal (Pal4 p) = aCountPal p + 2
    aCountPal (Pal5 p) = aCountPal p
```

2.26 Mirror-Palindromes

```
1   data Mir = Mir1 | Mir2 Mir | Mir3 Mir
    aMir1 = Mir2 (Mir3 (Mir2 Mir1))
    aMir2 = Mir2 (Mir3 (Mir3 Mir1))
2   a2cMir :: Mir -> String
    a2cMir Mir1 = ""
    a2cMir (Mir2 m) = "a" ++ a2cMir m ++ "a"
    a2cMir (Mir3 m) = "b" ++ a2cMir m ++ "b"
3   m2pMir :: Mir -> Pal
    m2pMir Mir1 = Pal1
    m2pMir (Mir2 m) = Pal4 (m2pMir m)
    m2pMir (Mir3 m) = Pal5 (m2pMir m)
```

2.27 Parity-Sequences

```
1   data Parity = Empty | Parity1 Parity | ParityL0 Parity | ParityR0 Parity
    aEven1 = ParityL0 (ParityL0 (Parity1 (ParityL0 Empty)))
    aEven2 = ParityL0 (ParityR0 (Parity1 (ParityL0 Empty)))
2   a2cParity :: Parity -> String
    a2cParity Empty = ""
    a2cParity (Parity1 p) = "1" ++ a2cParity p ++ "1"
    a2cParity (ParityL0 p) = "0" ++ a2cParity p
    a2cParity (ParityR0 p) = a2cParity p ++ "0"
```

2.28 Bit-Lists

```
1   data BitList = ConsB Bit Z | SingleB Bit
    data Z = ConsBL BitList Z | SingleBL BitList
    data Bit = Bit0 | Bit1
    aBitList1 = ConsB Bit0 (ConsBL (SingleB Bit1) (SingleBL (SingleB Bit0)))
    aBitList2 = ConsB Bit0 (ConsBL (SingleB Bit0) (SingleBL (SingleB Bit1)))
2   a2cBitList :: BitList -> String
    a2cBitList (ConsB b z) = bit2c b ++ a2cz z
    a2cBitList (SingleB b) = bit2c b
    a2cz (ConsBL bl z) = "," ++ a2cBitList bl ++ a2cz z
    a2cz (SingleBL bl) = "," ++ a2cBitList bl
    bit2c Bit0 = "0"
    bit2c Bit1 = "1"
```

3
```
concatBL :: BitList -> BitList -> BitList
concatBL (SingleB b) bl  = ConsB b (SingleBL bl)
concatBL (ConsB b z) bl  = ConsB b (concatZBL z bl)
concatZBL (ConsBL bl z) bl' = ConsBL bl (concatZBL z bl')
concatZBL (SingleBL bl) bl' = ConsBL bl (SingleBL bl')
```

2.29 We only give the EBNF notation for the productions that change.

$$\begin{array}{rcl} \mathit{Digs} & \rightarrow & \mathit{Dig}^* \\ Z & \rightarrow & \mathit{Sign?}\,\mathit{Nat} \\ \mathit{SoS} & \rightarrow & \mathit{LoD}^* \\ \mathit{LoD} & \rightarrow & \mathit{Letter} \mid \mathit{Dig} \end{array}$$

2.30

$$L(G?) \;=\; L(G) \cup \{\epsilon\}$$

2.31 Let $L(P)$ be the language defined by means of predicate P. Then we have $L(P1) \cap L(P2) = L(\lambda x.P1(x) \wedge P2(x)) = \{x \mid P1(x) \wedge P2(x)\}$, etc.

2.32

1. L_1 is generated by:

$$\begin{array}{rcl} S & \rightarrow & ZC \\ Z & \rightarrow & \mathsf{a}Z\mathsf{b} \mid \epsilon \\ C & \rightarrow & \mathsf{c}^* \end{array}$$

and L_2 is generated by:

$$\begin{array}{rcl} S & \rightarrow & AZ \\ Z & \rightarrow & \mathsf{b}Z\mathsf{c} \mid \epsilon \\ A & \rightarrow & \mathsf{a}^* \end{array}$$

2.

$$L_1 \cap L_2 \;=\; \{\mathsf{a}^n\mathsf{b}^n\mathsf{c}^n \mid n \in \mathbb{N}\}$$

and as we will see in Chapter 9, this language is not context-free.

2.33 No, since $\epsilon \in L^*$ we have $\epsilon \notin \overline{(L^*)}$ but $\epsilon \in \overline{(L)}^*$, so $\overline{(L^*)} \neq \overline{(L)}^*$ for all languages L.

2.34 $L = \{\mathsf{x}^n \mid n \in \mathbb{N}\}$.

2.35 This is only the case when $\epsilon \notin L$.

2.36 No, the language $L = \{\mathsf{aab}, \mathsf{baa}\}$ also satisfies $L = L^R$.

2.37

1. The shortest derivation gives in three steps the sentence aa. The sentences baa, aba and aab can be derived in four steps.

2. Several derivations are possible for the string babbab. Two of them are

$$\underline{S} \overset{*}{\Rightarrow} \underline{A}A \overset{*}{\Rightarrow} \mathtt{b}A\underline{A} \overset{*}{\Rightarrow} \mathtt{b}\underline{A}\mathtt{b}A \overset{*}{\Rightarrow} \mathtt{bab}\underline{A} \overset{*}{\Rightarrow} \mathtt{babb}\underline{A} \overset{*}{\Rightarrow} \mathtt{babb}\underline{A}\mathtt{b} \overset{*}{\Rightarrow} \mathtt{babbab}$$

   and

$$\underline{S} \overset{*}{\Rightarrow} A\underline{A} \overset{*}{\Rightarrow} \underline{A}A\mathtt{b} \overset{*}{\Rightarrow} \mathtt{b}\underline{A}A\mathtt{b} \overset{*}{\Rightarrow} \mathtt{b}\underline{A}\mathtt{b}A\mathtt{b} \overset{*}{\Rightarrow} \mathtt{b}\underline{A}\mathtt{bb}A\mathtt{b} \overset{*}{\Rightarrow} \mathtt{babb}\underline{A}\mathtt{b} \overset{*}{\Rightarrow} \mathtt{babbab}$$

3. A leftmost derivation is:

$$\underline{S} \overset{*}{\Rightarrow} \underline{A}A \overset{*}{\Rightarrow} \mathtt{b}^m\underline{A}A \overset{*}{\Rightarrow} \mathtt{b}^m\underline{A}\mathtt{b}^nA \overset{*}{\Rightarrow} \mathtt{b}^m\mathtt{ab}^n\underline{A} \overset{*}{\Rightarrow} \mathtt{b}^m\mathtt{ab}^n\underline{A}\mathtt{b}^p \overset{*}{\Rightarrow} \mathtt{b}^m\mathtt{ab}^n\mathtt{ab}^p$$

2.38 The grammar is equivalent to the grammar

$$\begin{aligned} S &\rightarrow \mathtt{aa}B \\ B &\rightarrow \mathtt{b}B\mathtt{ba} \\ B &\rightarrow \mathtt{a} \end{aligned}$$

This grammar generates the string aaa and the strings $\mathtt{aab}^m\mathtt{a}(\mathtt{ba})^m$ for $m \in \mathbb{N}^+$. The string aabbaabba does not appear in this language.

2.39 L is generated by:

$$S \rightarrow \mathtt{a}S\mathtt{a} \mid \mathtt{b}S\mathtt{b} \mid \mathtt{c}$$

The derivation is:

$$\underline{S} \overset{*}{\Rightarrow} \mathtt{a}\underline{S}\mathtt{a} \overset{*}{\Rightarrow} \mathtt{ab}\underline{S}\mathtt{ba} \overset{*}{\Rightarrow} \mathtt{abcba}$$

2.40 $L = \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$. This language is also generated by the grammar:

$$S \rightarrow \mathtt{a}S\mathtt{b} \mid \epsilon$$

2.41 $L_1 = \{\mathtt{a}^n \mid n \in \mathbb{N}\}$. This language is also generated by the grammar:

$$S \rightarrow \mathtt{a}S \mid \epsilon$$

$L_2 = \{\epsilon \cup \mathtt{a}^{2n+1} \mid n \in \mathbb{N}\}$. This language is also generated by the grammar:

$$\begin{aligned} S &\rightarrow A \mid \epsilon \\ A &\rightarrow \mathtt{a} \mid \mathtt{a}A\mathtt{a} \end{aligned}$$

2.42 The three grammars generate the language $L = \{\mathtt{a}^n \mid n \in \mathbb{N}\}$.

2.43

$$\begin{array}{lcl} S & = & A \mid \epsilon \\ A & = & \mathsf{a}A\mathsf{b} \mid \mathsf{ab} \end{array}$$

2.44

1. $\{ \mathsf{a}^{2n+1} \mid n \geq 0 \}$
2. $A = \mathsf{aa}A \mid \mathsf{a}$
3. $A = A\mathsf{aa} \mid \mathsf{a}$

2.45

1. $\{ \mathsf{ab}^n \mid n \geq 0 \}$
2. $$\begin{array}{lcl} X & = & \mathsf{a}Y \\ Y & = & \mathsf{b}Y \mid \epsilon \end{array}$$
3. $$\begin{array}{lcl} X & = & \mathsf{a}Y \mid \mathsf{a} \\ Y & = & \mathsf{b}Y \mid \mathsf{b} \end{array}$$

2.46

1. $$\begin{array}{lcl} S & = & T \mid US \\ T & = & X\mathsf{a} \mid U\mathsf{a} \\ X & = & \mathsf{a}S \\ U & = & S \mid YT \\ Y & = & SU \end{array}$$

   The sentential forms $\mathsf{a}S$ and SU have been abstracted to nonterminals X and Y.

2. $$\begin{array}{lcl} S & = & \mathsf{a}S\mathsf{a} \mid U\mathsf{a} \mid US \\ U & = & S \mid SU\mathsf{a}S\mathsf{a} \mid SUU\mathsf{a} \end{array}$$

   The nonterminal T has been substituted for its alternatives $\mathsf{a}S\mathsf{a}$ and $U\mathsf{a}$.

2.47

$$\begin{array}{lcl} S & \rightarrow & 1O \\ O & \rightarrow & 1O \mid 0N \\ N & \rightarrow & 1^* \end{array}$$

2.48 The language is generated by the grammar:

$$\begin{array}{lcl} S & \rightarrow & (A) \mid SS \\ A & \rightarrow & S \mid \epsilon \end{array}$$

A derivation for ( ) (( )) ( ) is:

$$\begin{array}{l} \underline{S} \overset{*}{\Rightarrow} \underline{S}S \overset{*}{\Rightarrow} \underline{S}SS \overset{*}{\Rightarrow} (\underline{A})SS \overset{*}{\Rightarrow} ()\underline{S}S \overset{*}{\Rightarrow} ()(\underline{A})S \\ \quad \overset{*}{\Rightarrow} ()(\underline{S})S \overset{*}{\Rightarrow} ()((\underline{A}))S \overset{*}{\Rightarrow} ()(())\underline{S} \overset{*}{\Rightarrow} ()(())(\underline{A}) \overset{*}{\Rightarrow} ()(())() \end{array}$$

2.49 The language is generated by the grammar:

$$\begin{aligned} S &\rightarrow (A) \mid [A] \mid SS \\ A &\rightarrow S \mid \epsilon \end{aligned}$$

A derivation for [( )] ( ) is:

$$\begin{aligned} \underline{S} &\stackrel{*}{\Rightarrow} \underline{S}S \stackrel{*}{\Rightarrow} [\underline{A}]S \stackrel{*}{\Rightarrow} [\underline{S}]S \\ &\stackrel{*}{\Rightarrow} [(\underline{A})]S \stackrel{*}{\Rightarrow} [()]\underline{S} \stackrel{*}{\Rightarrow} [()](\underline{A}) \stackrel{*}{\Rightarrow} [()]() \end{aligned}$$

2.50

1. first leftmost derivation:

   Sentence
   ⇒
   Subject Predicate
   ⇒
   they*Predicate*
   ⇒
   they *Verb NounPhrase*
   ⇒
   theyare*NounPhrase*
   ⇒
   theyare*Adjective Noun*
   ⇒
   theyareflying*Noun*
   ⇒
   theyareflyingplanes

2. second leftmost derivation:

   Sentence
   ⇒
   Subject Predicate
   ⇒
   they*Predicate*
   ⇒
   they *AuxVerb Verb Noun*
   ⇒
   theyare *Verb Noun*
   ⇒
   theyareflying*Noun*
   ⇒
   theyareflyingplanes

2.51 Here is an unambiguous grammar for the double parentheses language (although it may not be trivial to convince yourself that it actually *is* unambiguous).

$$S \rightarrow () \mid S() \mid (S) \mid S(S) \mid [] \mid S[] \mid [S] \mid S[S]$$

2.52 Here is a leftmost derivation for $\clubsuit\diamond\clubsuit\triangle\spadesuit$

$$\begin{aligned}\odot &\Rightarrow \odot\triangle\otimes \Rightarrow \otimes\triangle\otimes \Rightarrow \otimes\diamond \oplus \triangle\otimes \Rightarrow \oplus\diamond \oplus \triangle\otimes\\ &\Rightarrow \clubsuit\diamond \oplus \triangle\otimes \Rightarrow \clubsuit\diamond\clubsuit\triangle\otimes \Rightarrow \clubsuit\diamond\clubsuit\triangle\oplus \Rightarrow \clubsuit\diamond\clubsuit\triangle\spadesuit\end{aligned}$$

Notice that the grammar of this exercise is the same, up to renaming, as the grammar

$$\begin{aligned}E &= E{+}T \mid T\\ T &= T{*}F \mid F\\ F &= \mathsf{0} \mid \mathsf{1}\end{aligned}$$

2.53 The palindrome $s = \epsilon$ with length 0 can be generated by de grammar with the derivation $P \Rightarrow \epsilon$. The palindromes a, b and c are the palindromes with length 1. They can be derivated with $P \Rightarrow \mathsf{a}$, $P \Rightarrow \mathsf{b}$ and $P \Rightarrow \mathsf{c}$.
Suppose that the palindrome s with length $\mathsf{n} > 1$ can be generated by the grammar with some derivation $P \stackrel{*}{\Rightarrow} s$. Then the palindrome $\mathsf{a}s\mathsf{a}$ with length $\mathsf{n}+2$ can be generated by the grammar with the derivation $P \Rightarrow \mathsf{a}\underline{P}\mathsf{a} \stackrel{*}{\Rightarrow} \mathsf{a}s\mathsf{a}$. The others palindromes with length $\mathsf{n}+2$ are $\mathsf{b}s\mathsf{b}$ and $\mathsf{c}s\mathsf{c}$. They also can be generated in the same way.
This proves that the language of palindromes is generated by the grammar.

2.54 A derivation of S contains occurrences (possibly zero) of the nonterminal A and of the characters a and b and no other symbols. Let x be the number of occurrences of the nonterminal A and y the number of occurrences of the character a. We prove using induction that for each derivation from S the sum $\mathsf{x}+\mathsf{y}$ is even and greater than zero.
This is true for AA, the only direct derivation of S. In this case $\mathsf{x} = 2$ and $\mathsf{y} = 0$.
Suppose X is an indirect derivation of S and $\mathsf{x}+\mathsf{y} = 2\mathsf{n}$. If X does not contain the nonterminal A then $\mathsf{x} = 0$ and $\mathsf{y} = 2\mathsf{n}$. If X does contain an occurrence of A then the next derivation uses one of the four production rules:

$$\begin{aligned}A &\rightarrow AAA &&: \mathsf{x}\text{ increases by 2, }\mathsf{y}\text{ remains unchanged}\\ A &\rightarrow \mathsf{a} &&: \mathsf{x}\text{ decreases by 1, }\mathsf{y}\text{ increases by 1}\\ A &\rightarrow \mathsf{b}A &&: \mathsf{x}\text{ and }\mathsf{y}\text{ remain unchanged}\\ A &\rightarrow A\mathsf{b} &&: \mathsf{x}\text{ and }\mathsf{y}\text{ remain unchanged}\end{aligned}$$

It follows that in the next step $\mathsf{x}+\mathsf{y}$ remains even and the sum never decreases. So our proposition holds for each derivation of S.
A string of the language is a derivation without a nonterminal. Then $\mathsf{x} = 0$ and y, the number of occurrences of a, is even and greater than zero.

It remains to prove that in a sentence of the language the character b can occur at any position. Let s be a sentence of the language containing n

occurrences of the character a and m occurrences of the character b, with n even and greater than zero and m at least zero. A derivation for s is :

$$S \stackrel{*}{\Rightarrow} A^n \stackrel{*}{\Rightarrow} \mathsf{s}$$

In A^n the character b can then be introduced at any position by using one of the production rules:

$$A \rightarrow A\mathsf{b} \mid \mathsf{b}A$$

After that the remaining n nonterminals A are substituted by a using the production rule:

$$A \rightarrow \mathsf{a}$$

It follows that the grammar generates all strings over $\{\mathsf{a}, \mathsf{b}\}$ in which the number of a's is even and greater than zero and b can occur at any position.

3.1

1. To be able to test the function `string2sa` we adapt the datatype `SA` to make it an instance of the type class `Show`:

```
data SA = BesideA SA SA
        | SingleA
        deriving Show
```

   There are different possibilities for defining function `string2sa`. The following definition analyses the string from left to right and produces a parse tree that grows to the right:

```
string2sa           :: String -> SA
string2sa ['s']     =  SingleA
string2sa ('s':xs)  =  BesideA SingleA (string2sa xs)

? string2sa "sss"
BesideA SingleA (BesideA SingleA SingleA)
```

   The next definition of the function `sa2string` analyses the sequence from the right to left and returns a parse tree that grows to the left:

```
string2sa'          :: String -> SA
string2sa' ['s']    =  SingleA
string2sa' y        =  BesideA  (string2sa' (init y))
                                (string2sa' [last y])
```

   Many other definitions are possible. Note that we did not define function `string2sa` on the empty string, or on characters different from 's'. To obtain good error messages, both versions should be extended with the lines

```
string2sa []      =
  error "string2sa: Can not parse the empty string"
string2sa (x:xs)  |  x /= 's' =
  error "string2sa: Can not parse characters other than 's'"
```

2. The error message: string2sa: Can not parse characters other than 's'.

3.2

1. The following function tries to consume as many s's as possible from the input string.

```
parserSoS           :: String -> (SA, String)
parserSoS []        =
  error "parserSoS: Cannot parse the empty string"
parserSoS ('s':xs)  |  null xs || head xs /= 's'  =
                         (SingleA,xs)
                    |  otherwise                  =
                         let (result,rest) = parserSoS xs
                         in  (BesideA SingleA result,rest)
parserSoS _         =
  error "parserSoS: Cannot parse characters different from 's'"
```

2. The results are:

```
?> parserSoS "sss"
(BesideA SingleA (BesideA SingleA SingleA),"")
?> parserSoS "ssa"
(BesideA SingleA SingleA,"a")
?> parserSoS "sas"
(SingleA,"as")
?> parserSoS "s"
(SingleA,"")
```

Applying the function to the string `"aas"` produces a program error.

Note that to avoid program errors we could extend the function definition for a sequence that does not begin with an `'s'`, but then we would need an extra value of type `SA` to represent an empty or non-existing element:

```
data SA = BesideA SA SA
        | SingleA
        | None
    deriving Show
```

```
parserSoS           :: String -> (SA, String)
parserSoS ('s':xs)  |  null xs || head xs /= 's'  =
                         (SingleA,xs)
                    |  otherwise                  =
                         let (result,rest) = parserSoS xs
                         in  (BesideA SingleA result,rest)
parserSoS xs        =  (None,xs)

?>  parserSoS "aas"
(None,"aas")
```

3.3

1. The function `p1` takes one argument of type `[Char]` and a result of type `[(Int, [Char])]`. Two examples of values of this type are:

   ```
   [(1, "is a string"), (2, "a string"), (3, "string")]
   [(3, ['4', '5'])]
   ```

2. The function `p2` takes one argument of type `[Char]` and returns a value of type `[(Char -> Bool, [Char])]`. A value of this type is `[(isUpper, "This is a string"), (isAlpha, "")]` where the functions `isUpper` and `isAlpha` are standard functions from the prelude.

3.4

```
? symbola "any"
[('a',"ny")]
? symbola "none"
[]
```

3.5

1. ```
   ? symbol 'a' "any"
   [('a',"ny")]
   ```

2. ```
   ? :t symbol 'a' "any"
   symbol 'a' "any" :: [(Char,[Char])]
   ? :t symbol 'a'
   symbol 'a' :: [Char] -> [(Char,[Char])]
   ? :t symbol
   symbol :: Eq a => a -> Parser a a
   ```

   Note that `symbol 'a'` also is of type `Parser Char Char`.

3. ```
   spaceParser :: Parser Char Char
   spaceParser = symbol ' '
   ```

3.6
   ```

```
?> symbol 'h' "hallo"
[('h',"allo")]
?> symbol 'h' "Hallo"
[]
?> satisfy even [2,3,4]
[(2,[3,4])]
?> satisfy even [1,2,3]
[]
?> token "if" "if a then b"
[("if"," a then b")]
?> token "if" "else c"
[]
?> succeed "if" "then else"
[("if","then else")]
?> digit "1234"
[('1',"234")]
?> digit "abc"
[]
?> (failp :: Parser Char Int) "abc"
[]
?> epsilon "abc"
 [((),"abc")]
```

Note that we must specify a type for the parser `failp` when it is applied in order to be able to show the result.

3.7

```
capital  =  satisfy (\s -> ('A' <= s) && (s <= 'Z'))
```

or

```
capital  =  satisfy isUpper
```

3.8 A symbol equal to `a` satisfies `(==a)`:

```
symbol a  =  satisfy (==a)
```

3.9 The function `epsilon` is a special case of `succeed`:

```
epsilon  :: Parser s ()
epsilon  =  succeed ()
```

3.10

1 The parser `abParser` parses two symbols of type `Char` and returns a string:
```
abParser :: Parser Char String
```
2 The types of `symbol 'a'` and `symbol 'b'` are `Parser Char Char`. The operator `<*>` expects as its first argument a parser of type `Parser Char (String -> String)` and as its second argument a parser of type `Parser Char String`.

3 The function `('a':)` takes a string as argument. It returns the string with the character `'a'` in front. Its type is:

```
('a':) :: String -> String
```

4 The parser `symbolaParser` returns the function `('a':)`:

```
symbolaParser         :: Parser Char (String -> String)
symbolaParser []      =  []
symbolaParser (x:xs)  |  x == 'a'   =  [(('a':), xs)]
                      |  otherwise  =  []
```

5 The parser `symbolbParser` returns the string `"b"`:

```
symbolbParser         :: Parser Char String
symbolbParser []      =  []
symbolbParser (x:xs)  |  x == 'b'   =  [("b", xs)]
                      |  otherwise  =  []
```

6 The parser first recognises the character `'a'`, then the character `'b'`, and then produces the string `"ab"`. This is exactly what the result function of `symbolaParser` does when applied to the result of `symbolbParser`.

```
abParser = symbolaParser <*> symbolbParser
```

```
?> abParser "abcd"
[("ab","cd")]
?> abParser "bcd"
[]
```

3.11 The parser can be defined using the combinator `<*>`, and the parsers `token` and `goodParser`, which is based on the parser `token`. Parser `goodParser` recognises the token `"good "` and returns a function that concatenates the token with the result of the parser `token "bye "`.

```
goodParser     :: Parser Char (String -> String)
goodParser xs  |  "good " == take 5 xs  =
                     [(("good " ++),drop 5 xs)]
               |  otherwise             =  []

gbParser  :: Parser Char String
gbParser  =  goodParser <*> token "bye "
```

```
?> gbParser "good bye friends"
[("good bye ","friends")]
?> gbParser "good friends"
[]
```

3.12

```
aOrAParser  :: Parser Char Char
aOrAParser  =  (symbol 'a') <|> (symbol 'A')
```

```
?> aOrAParser "abcd"
[('a',"bcd")]
?> aOrAParser "ABCD"
[('A',"BCD")]
```

3.13 The parser can be defined using the combinator `<|>` and the parsers `gbParser` from exercise 3.11 and `token`.

```
gParser  :: Parser Char String
gParser  =  gbParser <|> token "good "

?> gParser "good friends"
[("good ","friends")]
?> gParser "good bye friends"
[("good bye ","friends"),("good ","bye friends")]
```

Note that the last application has two solutions.

3.14

1. The parser `digit` returns a character, the parser `newdigit` returns an integer.

   ```
   ?> digit "1abc"
   [('1',"abc")]
   ?> newdigit "1abc"
   [(1,"abc")]
   ```

2. ```
   newdigit' :: Parser Char Int
   newdigit' = (\c -> ord c - ord '0') <$> digit
   ```

3.15

1. ```
   symbolaParser'  :: Parser Char (String -> String)
   symbolaParser'  =  (:) <$> (symbol 'a')
   ```

2. ```
   abParser'  :: Parser Char String
   abParser'  =  symbolaParser' <*> symbolbParser

   ?> abParser' "abcd"
   [("ab","cd")]
   ?> abParser' "bcd"
   []
   ```

   To combine parsers with the parser combinator `<*>` it is not necessary to first build a new parser with the appropriate type like `symbolaParser` in exercise 3.10. A parser with the appropriate type can be built by applying a function to an existing parser using the combinator `<$>` as in `symbolaParser'`.

3. The lambda expression is a function that takes two arguments, the results of `(symbol 'a')` and `(symbol 'b')`, and returns the string `"ab"`:
   ```

```
abParser''  :: Parser Char String
abParser''  =  (\ x y -> x:[y]) <$> (symbol 'a')
                                <*> (symbol 'b')
```

3.16

1. `data SAB  =  AnA SAB | AB deriving Show`

2. `AnA (AnA (AnA AB))`

3. The parser recognises sentences constructed by means of the two production rules. It is defined with the combinator `<|>` and two parsers `p1` and `p2`:

   ```
   sabParser  =  p1 <|> p2
   ```

   The parser `p1` recognises strings that begin with the character `'a'` and parser `p2` recognises the character `'b'`.

   Let first look at parser `p2`. When `p2` encounters the character `'b'` the result should be `AB`. The parser can be defined as follow:

   ```
   p2  :: Parser Char SAB
   p2  =  const AB <$> symbol 'b'
   ```

   Recall that function `const` is defined in the prelude as follow:

   ```
   const      :: a -> a -> a
   const k _  =  k
   ```

   When parser `p1` encounters the character `'a'` the result should be `AnA` combined with the result of parsing the rest of the string. The parser is defined as follow:

   ```
   p1  :: Parser Char SAB
   p1  =  const AnA <$> symbol 'a' <*> sabParser
   ```

   So the complete definition of the parser `sabParser` looks as follows:

   ```
   sabParser  :: Parser Char SAB
   sabParser  =  const AnA <$> symbol 'a' <*> sabParser
             <|> const AB  <$> symbol 'b'
   ```

   Note that `sabParser` is a recursive function. The input string will be processed as long as the parser encounters a character `'a'`.

4.
```
?> sabParser "aaab"
[(AnA (AnA (AnA AB)),"")]
?> sabParser "baaa"
[(AB,"aaa")]
```

   The string `"aaab"` is completely parsed. The remaining string is empty. This string belongs to the language of the grammar. The string `"baaa"` is not parsed completely. The remaining string `"aaa"` cannot be derived using the production rules. It follows that the string `"baaa"` does not belong to the language.

5. A parse tree is no longer needed for the result of the parser `countaParser`. Instead, an integer is returned. The left argument of de combinator `<$>` is a function that increments an integer by 1 each time an `'a'` is encountered.

```
countaParser  :: Parser Char Int
countaParser  =  (\ x y -> y + 1) <$> symbol 'a'
                                  <*> countaParser
              <|> const 0         <$> symbol 'b'
```

   The lambda abstraction is a function that takes two arguments, the result of symbol `'a'` and the result of the recursive call of `countaParser`. The lambda abstraction increments the result of the recursive call by 1. When the character `'b'` is encountered the result of `countaParser` is 0.

```
?> countaParser "aaab"
[(3,"")]
?> countaParser "b"
[(0,"")]
?> countaParser "baa"
[(0,"aa")]
```

3.17

1. The datatype for the language is:

```
data SAB2  =  AnA2 A | EMPTY
  deriving Show
data A  =  AB2 SAB2
  deriving Show
```

   Because two datatypes are defined, we define two parsers:

```
sab2Parser  =  const AnA2  <$> symbol 'a' <*> aParser
            <|> const EMPTY <$> epsilon
aParser     =  const AB2 <$> symbol 'b' <*> sab2Parser
```

   Parser `sab2Parser` implements the production rules for S and parser `aParser` implements the production rule for A. Note that the two parsers are mutually recursive because `sab2Parser` calls `aParser` and vice versa.

```
?> sab2Parser "abab"
[(AnA2 (AB2 (AnA2 (AB2 EMPTY))),"")
,(AnA2 (AB2 EMPTY),"ab")
,(EMPTY,"abab")]
?> sab2Parser "ba"
[(EMPTY,"ba")]
```

The string `"abab"` belongs to the language, the string `"ba"` does not.

2. Each time a character `b` is encountered the count is incremented:

```
countabParser  :: Parser Char Int
countabParser  =  (\x y -> y) <$> symbol 'a'
                              <*> countbParser
               <|> const 0    <$> epsilon
countbParser   =  (\x y -> y+1) <$> symbol 'b'
                                <*> countabParser
```

3.18 Let `xs :: [s]`. Then

```
  (f <$> succeed a) xs
=     { definition of <$> }
  [ (f x,ys) | (x,ys) ← succeed a xs ]
=     { definition of succeed }
  [ (f a,xs) ]
=     { definition of succeed }
  succeed (f a) xs
```

3.19 The type and results of `list <$> symbol 'a'` are (note that you cannot write this as a definition in Haskell):

```
(list <$> symbol 'a') :: Parser Char (String -> String)
(list <$> symbol 'a') []                  =  []
(list <$> symbol 'a') (x:xs) | x == 'a'   =  [(list x,xs)]
                             | otherwise  =  []
```

3.20

1. The type and results of `list <$> symbol 'a' <*> p` are:

```
(list <$> symbol 'a' <*> p)           :: Parser Char String
(list <$> symbol 'a' <*> p) []        =  []
(list <$> symbol 'a' <*> p) (x:xs)
  |  x == 'a'   =  [(list 'a' x,ys) | (x,ys) <- p xs]
  |  otherwise  =  []
```

2. The parser `p` is of type `Parser Char String`. It is the type of the parser `token`:

```
?> (list <$> symbol 'a' <*> token "b") "abc"
[("ab","c")]
?> (list <$> symbol 'a' <*> token "b") "bcd"
[]
```

3.21

```
pBool  :: Parser Char Bool
pBool  =  const True  <$> token "True"
      <|> const False <$> token "False"
```

3.22

```
1  data Pal2 = Nil | Leafa | Leafb | Twoa Pal2 | Twob Pal2
2  palin2  :: Parser Char Pal2
   palin2  =  (\x y z -> Twoa y) <$>
              symbol 'a' <*> palin2 <*> symbol 'a'
          <|> (\x y z -> Twob y) <$>
              symbol 'b' <*> palin2 <*> symbol 'b'
          <|> const Leafa <$> symbol 'a'
          <|> const Leafb <$> symbol 'b'
          <|> succeed Nil
3  palina :: Parser Char Int
   palina =  (\x y z -> y+2) <$>
             symbol 'a' <*> palina <*> symbol 'a'
         <|> (\x y z -> y)   <$>
             symbol 'b' <*> palina <*> symbol 'b'
         <|> const 1 <$> symbol 'a'
         <|> const 0 <$> symbol 'b'
         <|> succeed 0
```

3.23 As `<|>` uses `++`, it is more efficiently evaluated right associative.

3.24 The function is the same as `<*>`, but instead of applying the result of the first parser to that of the second, it pairs them together:

```
(<,>)         :: Parser s a -> Parser s b -> Parser s (a,b)
(p <,> q) xs  =  [((x,y),zs)
                 |(x,ys) <- p xs
                 ,(y,zs) <- q ys
                 ]
```

3.25 'Parser transformator', or 'parser modifier' or 'parser postprocessor', etcetera.

3.26 The transformator `<$>` does to the result part of parsers what `map` does to the elements of a list.

3.27 The parser combinators `<*>` and `<,>` can be defined in terms of each other:

```
p <*> q  =  h <$> (p <,> q)
  where -- h :: (b -> a,b) -> a
        h (f,y) = f y
p <,> q  =  (h <$> p) <*> q
  where -- h :: a -> (b -> (a,b))
        h x y = (x, y)
```

3.28 Yes. You can combine the parser parameter of `<$>` with a parser that consumes no input and always yields the function parameter of `<$>`:

```
f <$> p  =  succeed f <*> p
```

3.29

```
f                       ::
   Char->Parentheses->Char->Parentheses->Parentheses
      open              ::
   Parser Char Char
f <$> open              ::
   Parser Char (Parentheses->Char->Parentheses->Parentheses)
                 parens ::
   Parser Char Parentheses
(f <$> open) <*> parens ::
   Parser Char (Char->Parentheses->Parentheses)
```

3.30 To the left. Yes.

3.31 The result of applying the parser to a string is a list of solutions. A string belongs to the language when there is a solution in the result of which the second element of the tuple is the empty string.
Function `test` first selects (using `filter`) all solutions of which the second element of the tuple is empty. If this selection is non-empty, then the parsed string belongs to the language parsed by the parser.

```
test p s  =  not (null (filter (null . snd) (p s)))
test p    =  not . null . filter (null . snd) . p
```

3.32

1. ```
   ?> (many (symbol 'a'))  "aaab"
   [("aaa","b"),("aa","ab"),("a","aab"),("","aaab")]
   ```

2. The parser must recognise many characters `'a'` and then a character `'b'`. To combine `many (symbol 'a')` and `symbol 'b'` we must use a semantic function. In the following parser the semantic function appends the result of the second parser to the result of the first parser. Note that the result of the second parser must first be transformed to a string.

   ```
   sabParser''' :: Parser Char String
   sabParser''' = (\ x y -> x ++ [y]) <$> (many (symbol 'a'))
                  <*> (symbol 'b')
   ```
   ```

```
?> sabParser''' "aaab"
[("aaab","")]
```

3.33 The first call has the empty string as solution, the second call has no solution:

```
?> (many (symbol 'a')) "baa"
[("","baa")]
?> (many1 (symbol 'a')) "baa"
[]
```

3.34 The first application has two solutions, the second application has one solution:

```
?> (option (symbol 'a') '?') "ab"
[('a',"b"),('?',"ab")]
?> (option (symbol 'a') '?') "ba"
[('?',"ba")]
```

3.35 When the character `'-'` is encountered the function `negate` is applied to the result of `natural`. Otherwise the function `id` is applied to the result:

```
integer  ::  Parser Char Int
integer  =  (const negate <$> (symbol '-')) `option` id  <*>  natural
```

3.36

1. The empty alternative is presented last, because the `<|>` combinator uses list concatenation for concatenating lists of successes. This also holds for the recursive calls; thus the 'greedy' parsing of all three `a`'s is presented first, then two `a`'s with a singleton rest string, then one `a`, and finally the empty result with the original input as rest string.

   ```
   ?> (many (symbol 'a')) "aaa"
   [("aaa",""),("aa","a"),("a","aa"),("","aaa")]
   ```

2. The result includes only the first solution where the hole string is parsed.

   ```
   ?> (greedy (symbol 'a')) "aaa"
   [("aaa","")]
   ```

3.37

1. ```
   notangle          :: Parser Char Char
   notangle []       =  []
   notangle (x:xs)   |  x /= '<'    =  [(x, xs)]
                     |  otherwise   =  []
   ```
   ```

2. Parser `greedy` is used instead of `many` because only the first solution is needed.

```
textwithoutangle  :: Parser Char String
textwithoutangle  =  greedy notangle
```

3.
```
simpleHtmlParser  :: Parser Char String
simpleHtmlParser  =
    pack (token "<html>") textwithoutangle (token "</html>")

?> simpleHtmlParser "<html>this is text</html>"
[("this is text","")]
?>  simpleHtmlParser "<html>this is text<html>"
[]
```

3.38

1.
```
listofdigits  :: Parser Char [Int]
listofdigits  =  listOf newdigit (symbol ' ')

?> listofdigits "1 2 3"
[([1,2,3],""),([1,2]," 3"),([1]," 2 3")]
?> listofdigits "1 2 a"
[([1,2]," a"),([1]," 2 a")]
```

2. In order to use the parser `chainr` we first define a parser `plusParser` that recognises the character `'+'` and returns the function `(+)`.

```
plusParser          :: Num a => Parser Char (a -> a -> a)
plusParser []       =  []
plusParser (x:xs)   |  x == '+'   =  [((+) , xs)]
                    |  otherwise  =  []
```

The definiton of the parser `sumParser` is:

```
sumParser  :: Parser Char Int
sumParser  =  chainr newdigit plusParser

?> sumParser "1+2+3"
[(6,""),(3,"+3"),(1,"+2+3")]
?> sumParser "1+2+a"
[(3,"+a"),(1,"+2+a")]
?> sumParser "1"
[(1,"")]
```

Note that the parser also recognises a single integer.

3. The parser `many` should be replaced by the parser `greedy` in de definition of `listOf`.

3.39

```
-- Combinators for repetition

psequence         :: [Parser s a] -> Parser s [a]
psequence []      =  succeed []
psequence (p:ps)  =  list <$> p <*> psequence ps

psequence'  :: [Parser s a] -> Parser s [a]
psequence'  =  foldr f (succeed [])
  where  f p q = list <$> p <*> q

choice  :: [Parser s a] -> Parser s a
choice  =  foldr (<|>) failp
```

```
  ?> (psequence [digit, satisfy isUpper]) "1A"
  [("1A","")]
  ?> (psequence [digit, satisfy isUpper]) "1Ab"
  [("1A","b")]
  ?> (psequence [digit, satisfy isUpper]) "1ab"
  []
```

```
  ?> (choice [digit, satisfy isUpper]) "1ab"
  [('1',"ab")]
  ?> (choice [digit, satisfy isUpper]) "Ab"
  [('A',"b")]
  ?> (choice [digit, satisfy isUpper]) "ab"
  []
```

3.40

```
   token  :: Eq s  =>  [s] -> Parser s [s]
   token  =  psequence . map symbol
```

3.41

```
identifier :: Parser Char String
identifier =  list <$> satisfy isAlpha <*> greedy (satisfy isAlphaNum)
```

3.42

1. ```
   "abc":     Var "abc"
   "(abc)":   Var "abc"
   "a*b+1":   Var "a" :*: Var "b" :+: Con 1
   "a*(b+1)": Var "a" :*: (Var "b" :+: Con 1)
   "-1-a":    Con (-1) :-: Var "a"
   "a(1,b)":  Fun "a" [Con 1,Var "b"]
   ```

2. The parser `fact` first tries to parse an integer, then a variable, then a function application and finally a parenthesised expression. A function application is a variable followed by an argument list. When the parser encounters a function application, a variable will first be recognised. This first solution will however not lead to a parse tree for the complete expression because the list of arguments that comes after the variable cannot be parsed.
   If we swap the second and the third line in the definition of the parser `fact`, the parse tree for a function application will be the first solution of the parser:

   ```
   fact  :: Parser Char Expr
   fact  =  Con <$> integer
        <|> Fun <$> identifier <*> parenthesised (commaList expr)
        <|> Var <$> identifier
        <|> parenthesised expr

   ?> expr "a(1,b)"
   [(Fun "a" [Con 1,Var "b"],""),(Var "a","(1,b)")]
   ```

3.43 A function with no arguments is not accepted by the parser:

```
?> expr "f()"
[(Var "f","()")]
```

The parser `parenthesised (commaList expr)` that is used in the parser `fact` does not accept an empty list of arguments because `commaList` does not. To accept an empty list we modify the parser `fact` as follows:

```
fact  :: Parser Char Expr
fact  =  Con <$> integer
     <|> Fun <$> identifier
             <*> parenthesised (commaList expr <|> succeed [])
     <|> Var <$> identifier
     <|> parenthesised expr

?> expr "f()"
[(Fun "f" [],""),(Var "f","()")]
```

3.44
```

```
expr = chainl (chainr term (const (:-:) <$> symbol '-'))
              (const (:+:) <$> symbol '+')
```

3.45 The datatype `Expr` is extended as follows to allow raising an expression to the power of an expression:

```
data Expr = Con Int
          | Var String
          | Fun String [Expr]
          | Expr :+: Expr
          | Expr :-: Expr
          | Expr :*: Expr
          | Expr :/: Expr
          | Expr :^: Expr
  deriving Show
```

Now the parser `expr'` of listing 8 can be extended with a new level of priorities:

```
powis = [ ('^',(:^:))]

expr' :: Parser Char Expr
expr' = foldr gen fact' [addis, multis, powis]
```

Note that because of the use of `chainl` all the operators listed in `addis`, `multis` and `powis` are treated as left-associative.

3.46 The proofs can be given by using laws for list comprehension, but here we prefer to exploit the following equation

```
(f <$> p) xs  =  map (cross (f,id)) (p xs)        (B.1)
```

where `cross` is defined by:

```
cross :: (a -> c,b -> d) -> (a, b) -> (c, d)
cross (f,g) (a,b) = (f a,g b)
```

It has the following property

```
cross (f,g) . cross (h,k)  =  cross (f.h,g.k)     (B.2)
```

Furthermore, we will use the following laws about `map` in our proof: map distributes over composition, concatenation, and function `concat`.

```
 map f . map g  =  map (f.g)                       (B.3)
map f (x ++ y)  =  map f x ++ map f y              (B.4)
map f . concat  =  concat . map (map f)            (B.5)
```

1.
```
(h <$> (f <$> p)) xs
=
map (cross (h,id)) ((f <$> p) xs)
=
map (cross (h,id)) (map (cross (f,id)) (p xs))
```
= { map distributes over composition: B.3}
```
map ((cross (h,id)) . (cross (f,id))) (p xs)
```
= { equation B.2 for cross }
```
map (cross (h.f,id)) (p xs)
```
= { equation B.1 for <$> }
```
((h.f) <$> p) xs
```

2.
```
(h <$> (p <|> q)) xs
=
map (cross (h,id)) ((p <|> q) xs)
=
map (cross (h,id)) (p xs ++ q xs)
```
= { map distributes over concatenation: B.4 }
```
map (cross (h,id)) (p xs) ++ map (cross (h,id)) (q xs)
```
= { equation B.1 for <$> }
```
(h <$> p) xs ++ (h <$> q) xs
```
= { definition of <|> }
```
((h <$> p) <|> (h <$> q)) xs
```

3. First note that `(p <*> q) xs` can be written as

```
(p <*> q) xs  =  concat (map (mc q) (p xs))          (B.6)
```

where

```
mc q (f,ys) = map (cross (f,id)) (q ys)
```

Now we calculate

```
( ((h.) <$> p) <*> q) xs
=
concat (map (mc q) (((h.) <$> p) xs))
```
= { equation B.1 for <$> }
```
concat (map (mc q) (map (cross ((h.),id)) (p xs)))
```
= { map distributes over composition }
```
concat (map (mc q . (cross ((h.),id))) (p xs))
```

```
=          { claim B.7 below }
           concat (map ((map (cross (h,id))) . mc q) (p xs))
=          { map distributes over composition }
           concat (map (map (cross (h,id)) (map (mc q) (p xs))))
=          { map distributes over concat: B.5 }
           map (cross (h,id)) (concat (map (mc q) (p xs)))
=          { equation B.6 }
           map (cross (h,id)) ((p <*> q) xs)
=          { equation B.1 for <$> }
           (h <$> (p <*> q)) xs
```

It remains to prove the claim

```
mc q . cross ((h.),id)  =  map (cross (h,id)) . mc q    (B.7)
```

This claim is also proved by calculation.

```
           ((map (cross (h,id))) . mc q) (f,ys)
=
           map (cross (h,id)) (mc q (f,ys))
=          { definition of mc q }
           map (cross (h,id)) (map (cross (f,id)) (q ys))
=          { map and cross distribute over composition }
           map (cross (h.f,id)) (q ys)
=          { definition of mc q }
           mc q (h.f,ys)
=          { definition of cross }
           (mc q . (cross ((h.),id))) (f,ys)
```

3.47

```
pMir  :: Parser Char Mir
pMir  =  (\o p q -> Mir3 p) <$> symbol 'b' <*> pMir <*> symbol 'b'
     <|> (\o p q -> Mir2 p) <$> symbol 'a' <*> pMir <*> symbol 'a'
     <|> succeed Mir1
```

3.48

```
pBitList  :: Parser Char BitList
pBitList  =  SingleB <$> pBit
         <|> (\b c bl -> ConsB b bl) <$> pBit <*> symbol ',' <*> pBitList
pBit      =  const Bit0 <$> symbol '0'
         <|> const Bit1 <$> symbol '1'
```

3.49 The parser returns the floating-point value of the fixed-point number. A fixed-point number is built from a whole part eventually followed by a decimal point and a fractional part. The whole part is a natural number or a negative sign followed by a natural number. In the former case the fixed-point number is positive and in the latter case it is negative. The parser can be defined by combining three parsers: a parser for the sign, a parser for a natural number and a parser for the fractional part.
The parser for the sign gives the float value -1.0 when the character `'-'` is encounterd and the float value 1.0 otherwise, leaving the string unprocessed:

```
signParser         :: Parser Char Float
signParser (x:xs)  |  x == '-'   =  [(-1.0, xs)]
                   |  otherwise  =  [(1.0, x:xs)]
```

The parser `natural` from section 3.4 with type `Parser Char Int` can be used to process the number of the whole part. The parser for the fractional part must compute the floating-point value after the decimal point:

```
fractpart  :: Parser Char Float
fractpart  =  foldr f 0.0 <$> greedy newdigit
    where f d n = (n + fromInt d) / 10.0
```

The parser `fixed` parses a fixed-point number. When it does not encounter a decimal point the value 0.0 will be used as fractional part:

```
fixed :: Parser Char Float
fixed = f <$> signParser
          <*> natural
          <*> (((\x y -> y) <$> symbol '.' <*> fractpart) `option` 0.0)
  where f a b c = a * (fromInt b + c)
```

Function `f` adds the results of the parser for the whole part and the parser for the fractional part, and multiplies the resulting value by the result of the parser for the sign.

```
?> fixed "-0.012"
[(-0.012,""),(0.0,".012")]
?> fixed "0.012"
[(0.012,""),(0.0,".012")]
?> fixed "-2"
[(-2.0,"")]
?> fixed "-2."
[(-2.0,""),(-2.0,".")]
```

3.50

```
float  :: Parser Char Float
float  =  f <$> fixed
                <*>
                (((\x y -> y) <$> symbol 'E' <*> integer) `option` 0)
  where  f m e   =  m * power e
         power e | e<0       = 1.0 / power (-e)
                 | otherwise = fromInteger (10^e)
```

3.51 Parse trees for Java assignments are of type:

```
data JavaAssign = JAssign String Expr
  deriving Show
```

The parser is defined as follows:

```
assign  :: Parser Char JavaAssign
assign  =  JAssign
       <$> identifier
       <*> (( \x y z -> y) <$> symbol '=' <*> expr <*> symbol ';')

?> assign "x1=(a+1)*2;"
[(JAssign "x1" (Var "a" :+: Con 1 :*: Con 2),"")]
?> assign "x=a+1"
[]
```

Note that the second example is not recognised as an assignment because the string does not end with a semicolon.

4.1 We obtain the net travel time if we replace the second occurrence of function `undefined` by:

```
\x (yh,ym) (zh,zm) -> (zh - yh) * 60 + zm - ym
```

The value returned by the parser `tstation` is thrown away. The values returned by the parsers `tdeparture` and `tarrival` are used to compute the net travel time.
The first occurrence of `undefined` sums the list of times:

```
\x y -> sum x
```

De definition of the parser is:

```
tstoken1  :: Parser Token Int
tstoken1  = (\x y -> sum x) <$>
            many ((\x (yh,ym) (zh,zm) -> (zh - yh) * 60 + zm - ym)
                  <$> tstation
                  <*> tdeparture
                  <*> tarrival
                 )
            <*> tstation
```

```
? tstoken1 (scanner "Groningen 8:37 9:44 Zwolle")
[(67,[]),
 (0,[Time_Token (8,37),Time_Token (9,44),Station_Token "Zwolle"])]
```

Note that the string is first processed by the scanner.

4.2 With

```
scheme =
  "Groningen 8:37 9:44 Zwolle 9:49 10:15 Utrecht 10:21 11:05 Den Haag"
```

none of the parsers succeed because all expect a name without a space character for a station.
To allow parser `tsstring` to accept a station with a space character in its name we modify the definition of parser `station` as follows:

```
station  :: Parser Char Station
station = (\x y z -> x ++ " " ++ z) <$> identifier
                                    <*> spaces
                                    <*> station
          <|> identifier
```

The parser parses groups of identifiers separated by spaces.
Similar changes can be made to parser `tstation` too.

4.3 The abstract syntax and the parser are rather straightforward.

```
data FloatLiteral = FL1 IntPart FractPart ExponentPart FloatSuffix
                  | FL2 FractPart ExponentPart FloatSuffix
                  | FL3 IntPart ExponentPart FloatSuffix
                  | FL4 IntPart ExponentPart FloatSuffix
                  deriving Show
type ExponentPart      = String
type ExponentIndicator = String
type SignedInteger     = String
type IntPart           = String
type FractPart         = String
type FloatSuffix       = String
digit             = satisfy isDigit
digits            = many1 digit
floatLiteral      =  (\a b c d e -> FL1 a c d e) <$>
                     intPart <*> period <*> optfract <*>
                     optexp <*> optfloat
                  <|> (\a b c d -> FL2 b c d) <$>
                     period <*> fractPart <*> optexp <*> optfloat
                  <|> (\a b c -> FL3 a b c) <$>
                     intPart <*> exponentPart <*> optfloat
                  <|> (\a b c -> FL4 a b c) <$>
                     intPart <*> optexp <*> floatSuffix
```

```
intPart           =  signedInteger
fractPart         =  digits
exponentPart      =  (++) <$> exponentIndicator <*> signedInteger
signedInteger     =  (++) <$> option sign "" <*> digits
exponentIndicator =  token "e" <|> token "E"
sign              =  token "+" <|> token "-"
floatSuffix       =  token "f" <|> token "F"
                  <|> token "d" <|> token "D"
period            =  token "."
optexp            =  option exponentPart ""
optfract          =  option fractPart ""
optfloat          =  option floatSuffix ""
```

4.4 The data and type definitions are the same as before, only the parsers return another (semantic) result. In order to evaluate the number correctly the parser `signedFloat` first parses a positive float-value and then transforms the value according to the (possibly absent) sign. The parser `intPart` that is used to evaluate the whole part of the number for the parser `floatLiteral` does not take the sign in account any more.

```
digits             :: Parser Char Int
digits             =  foldl f 0 <$> many1 newdigit
                        where f a b = 10*a + b
signedFloat        = (\ x y -> x y) <$> option sign id
                                    <*> floatLiteral
int2float :: Int -> Float
int2float          = fromInteger . toInteger
floatLiteral       =  (\a b c d e -> (int2float a + c) * power d) <$>
                        intPart <*> period <*>
                        optfract <*> optexp  <*> optfloat
                   <|> (\a b c d -> b * power c) <$>
                        period <*> fractPart <*> optexp <*> optfloat
                   <|> (\a b c -> int2float a * power b) <$>
                        intPart <*> exponentPart <*> optfloat
                   <|> (\a b c -> int2float a * power b) <$>
                        intPart <*> optexp <*> floatSuffix
signedInteger      =  (\ x y -> x y) <$> option sign id <*> digits
exponentIndicator  =  symbol 'e' <|> symbol 'E'
sign               :: Num a => Parser Char (a->a)
sign               =  const id <$> symbol '+'
                   <|> const negate <$> symbol '-'
intPart            =  digits
period             =  token "."
fractPart          =  foldr f 0.0 <$> many1 newdigit
                        where f a b = (int2float a + b)/10
exponentPart       =  (\x y -> y) <$> exponentIndicator <*>
                                      signedInteger
```

```
floatSuffix       =  symbol 'f' <|> symbol 'F'<|>
                     symbol 'd' <|>  symbol 'D'
optexp            = option exponentPart 0
optfract          = option fractPart 0.0
optfloat          = option floatSuffix ' '
power e  | e < 0     = 1 / power (-e)
         | otherwise =  (10^e)
```

Note that the function `int2float` is used to convert a value of type `Int` into a value of type `Float`.

5.1 Let $G = (T, N, R, S)$. Define the regular grammar $G' = (T, N \cup \{S'\}, R', S')$ as follows. S' is a new nonterminal symbol. For the productions R', divide the productions of R in two sets: the productions $R1$ of which the right-hand side consists just of terminal symbols, and the productions $R2$ of which the right-hand side ends with a nonterminal. Define a set $R3$ of productions by adding the nonterminal S at the right end of each production in $R1$. Define $R' = R \cup R3 \cup S' \rightarrow S \mid \epsilon$. The grammar G' thus defined is regular, and generates the language L^*.

5.2 In step 2, add the productions

$$
\begin{aligned}
&S \rightarrow \mathsf{a}S\\
&S \rightarrow \epsilon\\
&S \rightarrow \mathsf{c}C\\
&S \rightarrow \mathsf{a}\\
&A \rightarrow \epsilon\\
&A \rightarrow \mathsf{c}C\\
&A \rightarrow \mathsf{a}\\
&B \rightarrow \mathsf{c}C\\
&B \rightarrow \mathsf{a}
\end{aligned}
$$

and remove the productions

$$
\begin{aligned}
&S \rightarrow A\\
&A \rightarrow B\\
&B \rightarrow C
\end{aligned}
$$

In step 3, add the production $S \rightarrow \mathsf{a}$, and remove the productions $A \rightarrow \epsilon, B \rightarrow \epsilon$.

5.3

$$
\begin{aligned}
\mathit{ndfsa}\ d\ qs\ [\,] &= qs\\
\mathit{ndfsa}\ d\ qs\ (x : xs) &= \mathit{ndfsa}\ d\ (d\ qs\ x)\ xs
\end{aligned}
$$

5.4 Let $M = (X, Q, d, S, F)$ be a deterministic finite-state automaton. Then the nondeterministic finite-state automaton M' defined by $M' = (X, Q, d', \{S\}, F)$, where $d'\ q\ x = \{d\ q\ x\}$ accepts the same language.

5.5 Let $M = (X, Q, d, S, F)$ be a deterministic finite-state automaton that accepts language L. Then the deterministic finite-state automaton $M' =$

$(X, Q, d, S, Q - F)$, where $Q - F$ is the set of states Q from which all states in F have been removed, accepts the language $\overline{L}$. Here we assume that $d\ q\ \mathsf{a}$ is defined for all q and a.

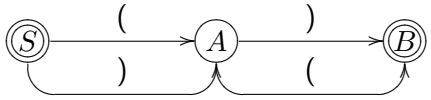
5.6

1 $L_1 \cap L_2 = \overline{(\overline{L_1} \cup \overline{L_2})}$, so it follows that regular languages are closed under intersection if they are closed under complementation and union. Regular languages are closed under union, see Theorem 10, and they are closed under complementation, see exercise 5.5.

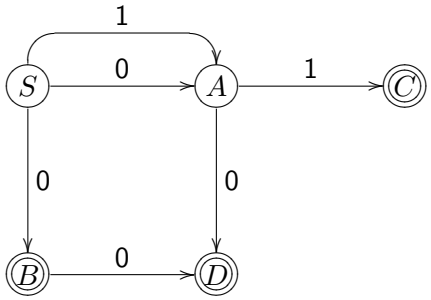
2
$$
\begin{array}{ll}
 & \mathit{Ldfa}\ M \\
= & \\
 & \{w \in X^* \mid \mathit{dfa\_accept}\ w\ (d, (S_1, S_2), (F_1 \times F_2))\} \\
= & \\
 & \{w \in X^* \mid \mathit{dfa}\ d\ (S_1, S_2)\ w \in F_1 \times F_2\} \\
= & \quad \text{This requires a proof by induction} \\
 & \{w \in X^* \mid (\mathit{dfa}\ d_1\ S_1\ w, \mathit{dfa}\ d_2\ S_2\ w) \in F_1 \times F_2\} \\
= & \\
 & \{w \in X^* \mid \mathit{dfa}\ d_1\ S_1\ w \in F_1 \wedge \mathit{dfa}\ d_2\ S_2\ w \in F_2\} \\
= & \\
 & \{w \in X^* \mid \mathit{dfa}\ d_1\ S_1\ w \in F_1\} \cap \{w \in X^* \mid \mathit{dfa}\ d_2\ S_2\ w \in F_2\} \\
= & \\
 & \mathit{Ldfa}\ M_1 \cap \mathit{Ldfa}\ M_2
\end{array}
$$

5.7

1



2



5.8 Use the definition of *Lre* to calculate the languages:

1 $\{\epsilon, \mathsf{b}\}$
2 (bc^*)
3 $\{\mathsf{a}\}\mathsf{b}^* \cup \mathsf{c}^*$

5.9

1
$$
\begin{array}{ll}
 & \mathit{Lre}(R(S + T)) \\
= & \\
 & (\mathit{Lre}(R))(\mathit{Lre}(S + T)) \\
= &
\end{array}
$$

$$
\begin{array}{ll}
 & (Lre(R))(Lre(S) \cup Lre(T)) \\
= & \\
 & (Lre(R))(Lre(S)) \cup (Lre(R))(Lre(T)) \\
= & \\
 & Lre(RS + RT)
\end{array}
$$

2 Similar to the above calculation.

5.10 Take $R = \mathsf{a}$, $S = \mathsf{a}$, and $R = \mathsf{a}$, $S = \mathsf{b}$.

5.11 If both V and W are subsets of S, then $Lre(R(S+V)) = Lre(R(S+W))$. Since $S \neq \emptyset$, $V = S$ and $W = \emptyset$ satisfy the requirement. Another solution is

$$
\begin{array}{rcl}
V & = & S \cap R \\
W & = & S \cap \overline{R}
\end{array}
$$

Since $S \neq \emptyset$, at least one of $S \cap R$ and $S \cap \overline{R}$ is not empty, and it follows that $V \neq W$. There exist other solutions than these.

5.12 The string $\mathsf{01}$ may not occur in a string in the language of the regular expression. So when a $\mathsf{0}$ appears somewhere, only $\mathsf{0}$'s can follow. Take $\mathsf{1}*\mathsf{0}*$.

5.13

1 If we can give a regular grammar for $(\mathsf{a} + \mathsf{bb})* + \mathsf{c}$, we can use the procedure constructed in exercise 5.1 to obtain a regular grammar for the complete regular expression. The following regular grammar generates $(\mathsf{a} + \mathsf{bb})* + \mathsf{c}$:

$$S_0 \rightarrow S_1 \mid \mathsf{c}$$

provided S_1 generates $(\mathsf{a} + \mathsf{bb})*$. Again, we use exercise 5.1 and a regular grammar for $\mathsf{a} + \mathsf{bb}$ to obtain a regular grammar for $(\mathsf{a} + \mathsf{bb})*$. The language of the regular expression $\mathsf{a} + \mathsf{bb}$ is generated by the regular grammar

$$S_2 \rightarrow \mathsf{a} \mid \mathsf{bb}$$

2 The language of the regular expression $\mathsf{a}*$ is generated by the regular grammar

$$S_0 \rightarrow \epsilon \mid \mathsf{a}S_0$$

The language of the regular expression $\mathsf{b}*$ is generated by the regular grammar

$$S_1 \rightarrow \epsilon \mid \mathsf{b}S_1$$

The language of the regular expression ab is generated by the regular grammar

$$S_2 \rightarrow \mathsf{ab}$$

It follows that the language of the regular expression $\mathsf{a}* + \mathsf{b}* + \mathsf{ab}$ is generated by the regular grammar

$$S_3 \rightarrow S_0 \mid S_1 \mid S_2$$

5.14 First construct a nondeterministic finite-state automaton for these grammars, and use the automata to construct the following regular expressions.

1 $\mathsf{a}(\mathsf{b} + \mathsf{b}(\mathsf{b} + \mathsf{ab})*(\mathsf{ab} + \epsilon)) + \mathsf{bb}* + \epsilon$

2 $(\mathsf{0} + \mathsf{10}*\mathsf{1})*$

5.16

1 $\mathsf{b} * + \mathsf{b} * (\mathsf{aa} * b(\mathsf{a} + \mathsf{b}) *)$
2 $(\mathsf{b} + \mathsf{a}(\mathsf{bb}) * \mathsf{b}) *$

6.1

```
type LNTreeAlgebra a b x  =  (a -> x,x -> b -> x -> x)

foldLNTree :: LNTreeAlgebra a b x -> LNTree a b -> x
foldLNTree (leaf,node)  =  fold where
  fold (Leaf a)      =  leaf a
  fold (Node l m r)  =  node (fold l) m (fold r)
```

6.2

1. ```
   -- height (Leaf x) = 0
   -- height (Bin lt rt) = 1 + (height lt) `max` (height rt)
   height :: BinTree x -> Int
   height = foldBinTree heightAlgebra
   heightAlgebra = (\u v -> 1 + u `max` v, const 0)
   ```
2. ```
   -- flatten (Leaf x) = [x]
   -- flatten (Bin lt rt) = (flatten lt) ++ (flatten rt)
   flatten :: BinTree x -> [x]
   flatten = foldBinTree ((++) ,\x -> [x])
   ```
3. ```
   -- maxBinTree (Leaf x) = x
   -- maxBinTree (Bin lt rt) = (maxBinTree lt) `max` (maxBinTree rt)
   maxBinTree :: Ord x => BinTree x -> x
   maxBinTree = foldBinTree (max, id)
   ```
4. ```
   -- sp (Leaf x) = 0
   -- sp (Bin lt rt) = 1 + (sp lt) `min` (sp rt)
   sp :: BinTree x -> Int
   sp = foldBinTree spAlgebra
   spAlgebra = (\u v -> 1 + u `min` v, const 0)
   ```
5. ```
   -- mapBinTree f (Leaf x) = Leaf (f x)
   -- mapBinTree f (Bin lt rt) = Bin (mapBinTree f lt) (mapBinTree f rt)
   mapBinTree :: (a -> b) -> BinTree a -> BinTree b
   mapBinTree f = foldBinTree (Bin, Leaf . f)
   ```

6.3

```
--allPaths (Leaf x) = [[]]
--allPaths (Bin lt rt)  =  map (Left:) (allPaths lt)
--                      ++ map (Right:) (allPaths rt)
allPaths :: BinTree a -> [[Direction]]
allPaths = foldBinTree psAlgebra
psAlgebra :: BinTreeAlgebra a [[Direction]]
psAlgebra = (\u v -> map (Left:) u ++ map (Right:) v, const [[]])
```
```

6.4

1. ```
   data Resist = Resist :|: Resist
               | Resist :*: Resist
               | BasicR Float
               deriving Show
   type ResistAlgebra a = (a -> a -> a, a -> a -> a, Float -> a)
   foldResist :: ResistAlgebra a -> Resist -> a
   foldResist (par, seq,basic) = fold where
     fold (r1 :|: r2) = par (fold r1) (fold r2)
     fold (r1 :*: r2) = seq (fold r1) (fold r2)
     fold (BasicR f)  = basic f
   ```

2. ```
   result :: Resist -> Float
   result = foldResist resultAlgebra
   resultAlgebra :: ResistAlgebra Float
   resultAlgebra = (\u v -> (u * v) /(u + v), (+), id)
   ```

6.5

1. ```
   isSum = foldExpr ((&&)
                    ,\x y -> False
                    ,\x y -> False
                    ,\x y -> False
                    ,const True
                    ,const True
                    ,\x ->   (&&)
                    )
   ```

2. ```
   vars = foldExpr ((++)
                   ,(++)
                   ,(++)
                   ,(++)
                   ,const []
                   ,\x -> [x]
                   ,\x y z -> x : (y ++ z)
                   )
   ```

6.6

1. `der` computes a symbolic differentiation of an expression.

2. The function `der` is not a compositional function on `Expr` because the righthand sides of the ‘`Mul`‘ and ‘`Dvd`‘ expressions do not only use `der e1 dx` and `der e2 dx`, but also `e1` and `e2` themselves.

3. ```
   data Exp = Exp ‘Plus‘ Exp
            | Exp ‘Sub‘ Exp
            | Con Float
   ```
   ```

```
          | Idf String
          deriving Show
type ExpAlgebra a = (a -> a -> a
                    ,a -> a -> a
                    ,Float -> a
                    ,String -> a
                    )
foldExp :: ExpAlgebra a -> Exp -> a
foldExp (plus, sub, con, idf)  =  fold where
  fold (e1 `Plus` e2)  =  plus (fold e1) (fold e2)
  fold (e1 `Sub` e2)   =  sub (fold e1) (fold e2)
  fold (Con n)         =  con n
  fold (Idf s)         =  idf s
```

4.
```
-- der :: Exp -> String -> Exp
-- der (e1 `Plus` e2) dx = (der e1 dx) `Plus` (der e2 dx)
-- der (e1 `Sub` e2) dx = (der e1 dx) `Sub` (der e2 dx)
-- der (Con f) dx       = Con 0.0
-- der (Idf s) dx       = if s == dx then Con 1.0 else Con 0.0
der = foldExp derAlgebra
derAlgebra :: ExpAlgebra (String -> Exp)
derAlgebra = (\f g -> \s -> f s `Plus` g s
             ,\f g -> \s -> f s `Sub` g s
             ,\n -> \s -> Con 0.0
             ,\s -> \t -> if s == t then Con 1.0 else Con 0.0
             )
```

6.7

```
-- replace (Leaf x) y = Leaf y
-- replace (Bin lt rt) y = Bin (replace lt y) (replace rt y)
replace :: BinTree a -> a -> BinTree a
replace = foldBinTree repAlgebra
repAlgebra = (\f g -> \y -> Bin (f y) (g y), \x y -> Leaf y)
```

6.8

```
--path2Value (Leaf x) = \bs -> if null bs then x else error"no roothpath"
--path2Value (Bin lt rt) =
-- \bs -> case bs of
--          []         -> error "no roothpath"
--          (Left:rs)  -> path2Value lt rs
--          (Right:rs) -> path2Value rt rs
--
path2Value :: BinTree a -> [Direction] -> a
path2Value = foldBinTree pvAlgebra
pvAlgebra :: BinTreeAlgebra a ([Direction] -> a)
pvAlgebra = (\fl fr -> \ bs -> case bs of
```

```
                          {[]         -> error "no roothpath"
                          ;(Left:rs)  -> fl rs
                          ;(Right:rs) -> fr rs
                          }
        , \x -> \ bs -> if null bs
                        then x
                        else error "no roothpath")
```

6.9

1
```
type PalAlgebra p = (p, p, p, p -> p, p -> p)
```

2
```
foldPal :: PalAlgebra p -> Pal -> p
foldPal (pal1,pal2,pal3,pal4,pal5) = fPal where
  fPal Pal1      =  pal1
  fPal Pal2      =  pal2
  fPal Pal3      =  pal3
  fPal (Pal4 p)  =  pal4 (fPal p)
  fPal (Pal5 p)  =  pal5 (fPal p)
```

3
```
a2cPal    = foldPal (""
                    ,"a"
                    ,"b"
                    ,\p ->"a"++p++"a"
                    ,\p ->"b"++p++"b"
                    )
aCountPal = foldPal (0,1,0,\p->p+2,\p->p)
```

4
```
pfoldPal :: PalAlgebra p -> Parser Char p
pfoldPal (pal1, pal2, pal3, pal4, pal5) = pPal where
  pPal =     const pal1 <$> epsilon
         <|> const pal2 <$> syma
         <|> const pal3 <$> symb
         <|> (\_ p _ -> pal4 p) <$> syma <*> pPal <*> syma
         <|> (\_ p _ -> pal5 p) <$> symb <*> pPal <*> symb
  syma = symbol 'a'
  symb = symbol 'b'
```

5 The parser `pfoldPal m1` returns the concrete representation of a palindrome. The parser `pfoldPal m2` returns the number of `a`'s occurring in a palindrome.

6.10

1
```
type MirAlgebra m = (m,m->m,m->m)
```

2
```
foldMir :: MirAlgebra m -> Mir -> m
foldMir (mir1,mir2,mir3) = fMir where
  fMir Mir1      =  mir1
  fMir (Mir2 m)  =  mir2 (fMir m)
```

```
  fMir (Mir3 m)  =  mir3 (fMir m)
```

3
```
a2cMir = foldMir ("",\m->"a"++m++"a",\m->"b"++m++"b")
m2pMir = foldMir (Pal1,Pal4,Pal5)
```

4
```
pfoldMir :: MirAlgebra m -> Parser Char m
pfoldMir (mir1, mir2, mir3) = pMir where
  pMir =     const mir1 <$> epsilon
         <|> (\_ m _ -> mir2 m) <$> syma <*> pMir <*> syma
         <|> (\_ m _ -> mir3 m) <$> symb <*> pMir <*> symb
  syma = symbol 'a'
  symb = symbol 'b'
```

5 The parser `pfoldMir m1` returns the concrete representation of a palindrome. The parser `pfoldMir m2` returns the abstract representation of a palindrome.

6.11

1
```
type ParityAlgebra p = (p,p->p,p->p,p->p)
```

2
```
foldParity :: ParityAlgebra p -> Parity -> p
foldParity (empty,parity1,parityL0,parityR0) = fParity where
  fParity Empty  =  empty
  fParity (Parity1 p)  =  parity1 (fParity p)
  fParity (ParityL0 p)  =  parityL0 (fParity p)
  fParity (ParityR0 p)  =  parityR0 (fParity p)
```

3
```
a2cParity = foldParity (""
                       ,\x->"1"++x++"1"
                       ,\x->"0"++x
                       ,\x->x++"0"
                       )
```

6.12

1
```
type BitListAlgebra bl z b = ((b->z->bl,b->bl),(bl->z->z,bl->z),(b,b))
```

2
```
foldBitList :: BitListAlgebra bl z b -> BitList -> bl
foldBitList ((consb,singleb),(consbl,singlebl),(bit0,bit1)) = fBitList
  where
  fBitList (ConsB b z)  =  consb (fBit b) (fZ z)
  fBitList (SingleB b)  =  singleb (fBit b)
  fZ (ConsBL bl z)      =  consbl (fBitList bl) (fZ z)
  fZ (SingleBL bl)      =  singlebl (fBitList bl)
  fBit Bit0             =  bit0
  fBit Bit1             =  bit1
```

3
```
a2cBitList = foldBitList (((++),id)
```

```
                              ,((++),id)
                              ,("0","1")
                              )
```

4
```
pfoldBitList :: BitListAlgebra bl z b -> Parser Char bl
pfoldBitList ((consb,singleb),(consbl,singlebl),(bit0,bit1)) = pBitList
  where
  pBitList  =  consb <$> pBit <*> pZ
           <|> singleb <$> pBit
  pZ        =  (\_ bl z -> consbl bl z) <$> symbol ','
                                        <*> pBitList
                                        <*> pZ
           <|> singlebl <$> pBitList
  pBit      =  const bit0 <$> symbol '0'
           <|> const bit1 <$> symbol '1'
```

6.13

1.
```
data Block  =  B1 Stat Rest
data Rest   =  R1 Stat Rest | Nix
data Stat   =  S1 Decl | S2 Use | S3 Nest
data Decl   =  Dx | Dy
data Use    =  UX | UY
data Nest   =  N1 Block
```

The abstract representation of `x;(y;Y);X` is

```
B1 (S1 Dx) (R1 stat2 rest2) where
  stat2 = S3 (N1 block)
  block = B1 (S1 Dy) (R1 (S2 UY) Nix)
  rest2 = R1 (S2 UX) Nix
```

2.
```
type BlockAlgebra b r s d u n =
  (s -> r -> b
  ,(s -> r -> r, r)
  ,(d -> s, u -> s, n -> s)
  ,(d,d)
  ,(u,u)
  ,b -> n
  )
```

3.
```
foldBlock :: BlockAlgebra b r s d u n -> Block -> b
foldBlock (b1, (r1, nix), (s1, s2, s3), (dx, dy), (ux, uy), n1) =
  foldB
  where
  foldB (B1 stat rest) = b1 (foldS stat) (foldR rest)
  foldR (R1 stat rest) = r1 (foldS stat) (foldR rest)
  foldR Nix            = nix
```

```
  foldS (S1 decl)      = s1 (foldD decl)
  foldS (S2 use)       = s2 (foldU use)
  foldS (S3 nest)      = s3 (foldN nest)
  foldD Dx             = dx
  foldD Dy             = dy
  foldU UX             = ux
  foldU UY             = uy
  foldN (N1 block)     = n1 (foldB block)
```

4.
```
a2cBlock :: Block -> String
a2cBlock = foldBlock a2cBlockAlgebra
a2cBlockAlgebra ::
  BlockAlgebra String String String String String String
a2cBlockAlgebra =
  (b1, (r1, nix), (s1, s2, s3), (dx, dy), (ux, uy), n1)
  where b1 u v = u ++ v
        r1 u v = ";" ++ u ++ v
        nix    = ""
        s1 u   = u
        s2 u   = u
        s3 u   = u
        dx    = "x"
        dy    = "y"
        ux    = "X"
        uy    = "Y"
        n1 u   = "(" ++ u ++ ")"
```

8.1 The type `TreeAlgebra` and the function `foldTree` are defined in the `rep-min` problem.

1.
```
height = foldTree heightAlgebra
heightAlgebra = (const 0, \l r -> (l `max` r) + 1)
--frontAtLevel :: Tree -> Int -> [Int]
--frontAtLevel (Leaf i) h  = if h == 0 then [i] else []
--frontAtLevel (Bin l r) h =
--  if h > 0  then frontAtLevel l (h-1) ++ frontAtLevel r (h-1)
--            else []
frontAtLevel        = foldTree frontAtLevelAlgebra
frontAtLevelAlgebra =
  ( \i h    -> if h == 0 then [i] else []
  , \f g h  -> if h > 0 then f (h-1) ++ g (h-1) else []
  )
```

2. (a) Straightforward Solution: impossible to give a solution like rm.sol1.

```
heightAlgebra = (const 0, \l r -> (l `max` r) + 1)
front t = foldTree frontAtLevelAlgebra t h
  where h = foldTree heightAlgebra t
```

(b) Lambda Lifting

```
front' t = foldTree
             frontAtLevelAlgebra
             t
             (foldTree heightAlgebra t)
```

(c) Tupling Computations

```
htupfr :: TreeAlgebra (Int, Int -> [Int])
htupfr
--     = (heightAlgebra `tuple` frontAtLevelAlgebra)
       = ( \i             -> ( 0
                             , \h -> if h == 0 then [i] else []
                             )
         , \(lh,f) (rh,g) -> ( (lh `max` rh) + 1
                             , \h -> if h > 0
                                     then f (h-1) ++ g (h-1)
                                     else []
                             )
         )
front'' t = fr h
  where (h, fr) = foldTree htupfr t
```

(d) Merging Tupled Functions
It is helpful to note that the merged algebra is constructed such that

```
foldTree mergedAlgebra t i = (height t, frontAtLevel t i)
```

Therefore, the definitions corresponding to the fourth solution are

```
mergedAlgebra :: TreeAlgebra (Int -> (Int, [Int]))
mergedAlgebra =
  (\i          -> \h -> ( 0
                        , if h == 0 then [i] else []
                        )
  , \lfun rfun -> \h -> let (lh, xs) = lfun (h-1)
                            (rh, ys) = rfun (h-1)
                        in
                        ( (lh `max` rh) + 1
                        , if h > 0 then xs ++ ys else []
                        )
   )
front''' t = fr
  where (h, fr) = foldTree mergedAlgebra t h
```

8.2 The highest_front problem is the problem of finding the first non-empty list of nodes which are at the lowest level. The solutions are similar to these of the deepest_front problem.

1. `--lowest :: Tree -> Int`

```
--lowest (Leaf i) = 0
--lowest (Bin l r) = ((lowest l) `min` (lowest r)) + 1
lowest = foldTree lowAlgebra
lowAlgebra = (const 0, \ l r -> (l `min` r) + 1)
```

2. (a) Straightforward Solution: impossible to give a solution like rm.sol1.

```
lfront t = foldTree frontAtLevelAlgebra t l
  where l = foldTree lowAlgebra t
```

   (b) Lambda Lifting

```
lfront' t = foldTree
              frontAtLevelAlgebra
              t
              (foldTree lowAlgebra t)
```

   (c) Tupling Computations

```
ltupfr :: TreeAlgebra (Int, Int -> [Int])
ltupfr =
  ( \i              -> ( 0
                       , (\l -> if l == 0 then [i] else [])
                       )
  , \ (lh,f) (rh,g) -> ( (lh `min` rh) + 1
                       , \l -> if l > 0
                               then f (l-1) ++ g (l-1)
                               else []
                       )
  )
lfront'' t = fr l
  where (l, fr) = foldTree ltupfr t
```

   (d) Merging Tupled Functions
   It is helpful to note that the merged algebra is constructed such that

```
foldTree mergedAlgebra t i = (lowest t, frontAtLevel t i)
```

   Therefore, the definitions corresponding to the fourth solution are

```
lmergedAlgebra :: TreeAlgebra (Int -> (Int, [Int]))
lmergedAlgebra =
  ( \i        -> \l -> ( 0
                       , if l == 0 then [i] else []
                       )
  , \lfun rfun -> \l ->
      let (ll, lres) = lfun (l-1)
          (rl, rres) = rfun (l-1)
      in
      ( (ll `min` rl) + 1
      , if l > 0 then lres ++ rres else []
      )
```

```
    )
lfront''' t = fr
  where (l, fr) = foldTree lmergedAlgebra t l
```

9.1 The conditions are:

Let $n \in \mathbb{N}$.
Take $s = \mathsf{a}^n\mathsf{b}^n$ with $x = \epsilon$, $y = \mathsf{a}^n$, and $z = \mathsf{b}^n$.
Let u, v, w be such that $y = uvw$ with $v \neq \epsilon$, that is, $u = \mathsf{a}^p$, $v = \mathsf{a}^q$ and $w = \mathsf{a}^r$ with $p + q + r = n$ and $q > 0$.
Take $i = 2$.

For the given conditions we want to prove that $xuv^2wz \notin L$.
In the first implication x, u, v, w and z are replaced by their value. Now we have to prove that $\mathsf{a}^{p+2q+r}\mathsf{b}^n \notin L$.
In the second implication $p + 2q + r$ is substituted by n. So $s = \mathsf{a}^{n+q}\mathsf{b}^n$. Now we have to prove that $n + q \neq n$.
In the third implication we have to prove that $q \neq 0$. Because $q > 0$ is a given condition, we conclude in the fourth implication that the proof is done.

9.2 We follow the proof pattern for non-regular languages given in the text.
Let $n \in \mathbb{N}$.
Take $s = \mathsf{a}^{n^2}$ with $x = \epsilon$, $y = \mathsf{a}^n$, and $z = \mathsf{a}^{n^2-n}$.
Let u, v, w be such that $y = uvw$ with $v \neq \epsilon$, that is, $u = \mathsf{a}^p$, $v = \mathsf{a}^q$ and $w = \mathsf{a}^r$ with $p + q + r = n$ and $q > 0$.
Take $i = 2$, then

$$
\begin{array}{ll}
 & xuv^2wz \notin L \\
\Leftarrow & \quad \text{defn. } x, u, v, w, z, \text{ calculus} \\
 & \mathsf{a}^{p+2q+r}\mathsf{a}^{n^2-n} \notin L \\
\Leftarrow & \quad p + q + r = n \text{ and } q > 0 \\
 & n^2 + q \text{ is not a square} \\
\Leftarrow & \\
 & n^2 < n^2 + q < (n+1)^2 \\
\Leftarrow & \quad p \geq 0,\ r \geq 0 \ so\ q \leq n \\
 & true
\end{array}
$$

9.3 Using the proof pattern.
Let $n \in \mathbb{N}$.
Take $s = \mathsf{a}^n\mathsf{b}^{n+1}$ with $x = \mathsf{a}^n$, $y = \mathsf{b}^n$, and $z = \mathsf{b}$.
Let u, v, w be such that $y = uvw$ with $v \neq \epsilon$, that is, $u = \mathsf{b}^p$, $v = \mathsf{b}^q$ and $w = \mathsf{b}^r$ with $p + q + r = n$ and $q > 0$.
Take $i = 0$, then

$$
\begin{array}{ll}
 & xuwz \notin L \\
\Leftarrow & \quad \text{defn. } x, u, v, w, z, \text{ calculus}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{a}^n\mathsf{b}^{p+r+1} \notin L \\
\Leftarrow & \\
 & p+q+r \geq p+r+1 \\
\Leftarrow & \quad q > 0 \\
 & true
\end{array}
$$

9.4 Using the proof pattern.
Let $n \in \mathbb{N}$.
Take $s = \mathsf{a}^n\mathsf{b}^{2n}$ with $x = \mathsf{a}^n$, $y = \mathsf{b}^n$, and $z = \mathsf{b}^n$.
Let u, v, w be such that $y = uvw$ with $v \neq \epsilon$, that is, $u = \mathsf{b}^p$, $v = \mathsf{b}^q$ and $w = \mathsf{b}^r$ with $p + q + r = n$ and $q > 0$.
Take $i = 2$, then

$$
\begin{array}{ll}
 & xuv^2wz \notin L \\
\Leftarrow & \quad \text{defn. } x, u, v, w, z, \text{ calculus} \\
 & \mathsf{a}^n\mathsf{b}^{2n+q} \notin L \\
\Leftarrow & \\
 & 2n + q > 2n \\
\Leftarrow & \quad q > 0 \\
 & true
\end{array}
$$

9.5 Using the proof pattern.
Let $n \in \mathbb{N}$.
Take $s = \mathsf{a}^6\mathsf{b}^{n+4}\mathsf{a}^n$ with $x = \mathsf{a}^6\mathsf{b}^{n+4}$, $y = \mathsf{a}^n$, and $z = \epsilon$.
Let u, v, w be such that $y = uvw$ with $v \neq \epsilon$, that is, $u = \mathsf{a}^p$, $v = \mathsf{a}^q$ and $w = \mathsf{a}^r$ with $p + q + r = n$ and $q > 0$.
Take $i = 6$, then

$$
\begin{array}{ll}
 & xuv^6wz \notin L \\
\Leftarrow & \quad \text{defn. } x, u, v, w, z, \text{ calculus} \\
 & \mathsf{a}^6\mathsf{b}^{n+4}\mathsf{a}^{n+5q} \notin L \\
\Leftarrow & \\
 & n + 5q > n + 4 \\
\Leftarrow & \quad q > 0 \\
 & true
\end{array}
$$

9.6 The length of a substring with a's, b's, and c's is at least $r+2$, and $|vwx| \leq d \leq r$.

9.7 We follow the proof pattern for proving that a language is not context-free given in the text.
Let $c, d \in \mathbb{N}$.
Take $z = \mathsf{a}^{k^2}$ with $k = max(c, d)$.
Let u, v, w, x, y be such that $z = uvwxy$, $|vx| > 0$ and $|vwx| \leq d$
That is $u = \mathsf{a}^p$, $v = \mathsf{a}^q$, $w = \mathsf{a}^r$, $x = \mathsf{a}^s$ and $y = \mathsf{a}^t$ with $p + q + r + s + t = k^2$, $q + r + s \leq d$ and $q + s > 0$.

Take $i = 2$, then

$$
\begin{array}{ll}
 & uv^2wx^2y \notin L \\
\Leftarrow & \quad \text{defn. } u,v,w,x,y, \text{ calculus} \\
 & \mathsf{a}^{k^2+q+s} \notin L \\
\Leftarrow & \quad q+s>0 \\
 & k^2+q+s \text{ is not a square} \\
\Leftarrow & \\
 & k^2 < k^2+q+s < (k+1)^2 \\
\Leftarrow & \quad 0 < q+s \leq d \leq k \\
 & \mathit{true}
\end{array}
$$

9.8 Using the proof pattern.
Let $c, d \in \mathbb{N}$.
Take $z = \mathsf{a}^k$ with k is prime and $k > max(c, d)$.
Let u, v, w, x, y be such that $z = uvwxy$, $|vx| > 0$ and $|vwx| \leq d$
That is $u = \mathsf{a}^p$, $v = \mathsf{a}^q$, $w = \mathsf{a}^r$, $x = \mathsf{a}^s$ and $y = \mathsf{a}^t$ with $p + q + r + s + t = k$, $q + r + s \leq d$ and $q + s > 0$.
Take $i = k + 1$, then

$$
\begin{array}{ll}
 & uv^{k+1}wx^{k+1}y \notin L \\
\Leftarrow & \quad \text{defn. } u,v,w,x,y, \text{ calculus} \\
 & \mathsf{a}^{k+kq+ks} \notin L \\
\Leftarrow & \\
 & k(1+q+s) \text{ is not a prime} \\
\Leftarrow & \quad q+s>0 \\
 & \mathit{true}
\end{array}
$$

9.9 Using the proof pattern.
Let $c, d \in \mathbb{N}$.
Take $z = \mathsf{a}^k\mathsf{b}^k\mathsf{a}^k\mathsf{b}^k$ with $k = max(c, d)$.
Let u, v, w, x, y be such that $z = uvwxy$, $|vx| > 0$ and $|vwx| \leq d$
Note that our choice for k guarantees that substring vwx has one of the following shapes:

- vwx consists of just a's, or just b's.
- vwx contains both a's and b's.

Take $i = 0$, then

- If vwx consists of just a's, or just b's, then it is impossible to write the string uwy as ww for some string w, since only the number of terminals of one kind is decreased.
- If vwx contains both a's and b's, it lies somewhere on the border between a's and b's, or on the border between b's and a's. Then the string uwy can be written as

$$uwy \quad = \quad \mathsf{a}^s\mathsf{b}^t\mathsf{a}^p\mathsf{b}^q$$

for some s, t, p, q, respectively. At least one of s ,t, p and q is less than k, while two of them are equal to k. Again this sentence is not an element of the language.

9.10 Using the proof pattern.
Let $n \in \mathbb{N}$.
Take $s = 1^n 0^n$ with $x = 1^n$, $y = 0^n$, and $z = \epsilon$.
Let u, v, w be such that $y = uvw$ with $v \neq \epsilon$, that is, $u = \mathsf{b}^p$, $v = \mathsf{b}^q$ and $w = \mathsf{b}^r$ with $p + q + r = n$ and $q > 0$.
Take $i = 2$, then

$$
\begin{array}{ll}
 & xuv^2w \notin L \\
\Leftarrow & \quad \text{defn. } x, u, v, w, z, \text{ calculus} \\
 & 1^n 0^{p+2q+r} \notin L \\
\Leftarrow & \quad p + q + r = n \\
 & 1^n 0^{n+q} \notin L \\
\Leftarrow & \quad q > 0 \\
 & true
\end{array}
$$

9.11

1. The language $\{\mathsf{a}^i\mathsf{b}^j \mid 0 \leq i \leq j\}$ is context-free. The language can be generated by the following context-free grammar:

$$
\begin{array}{lcl}
S & \rightarrow & \mathsf{a}S\mathsf{b}A \\
S & \rightarrow & \epsilon \\
A & \rightarrow & \mathsf{b}A \\
A & \rightarrow & \epsilon
\end{array}
$$

   This grammar generates the strings $\mathsf{a}^m\mathsf{b}^m\mathsf{b}^n$ for $m, n \in \mathbb{N}$. These are exactly the strings of the given language.

2. The language is not regular. We prove this using the proof pattern.
   Let $n \in \mathbb{N}$.
   Take $s = \mathsf{a}^n\mathsf{b}^{n+1}$ with $x = \epsilon$, $y = \mathsf{a}^n$, and $z = \mathsf{b}^{n+1}$.
   Let u, v, w be such that $y = uvw$ with $v \neq \epsilon$, that is, $u = \mathsf{a}^p$, $v = \mathsf{a}^q$ and $w = \mathsf{a}^r$ with $p + q + r = n$ and $q > 0$.
   Take $i = 3$, then

$$
\begin{array}{ll}
 & xuv^3wz \notin L \\
\Leftarrow & \quad \text{defn. } x, u, v, w, z, \text{ calculus} \\
 & \mathsf{a}^{p+3q+r}\mathsf{b}^{n+1} \notin L \\
\Leftarrow & \quad p + q + r = n
\end{array}
$$

$$
\begin{array}{ll}
 & n + 2q > n + 1 \\
\Leftarrow & \quad q > 0 \\
 & true
\end{array}
$$

9.12

1. The language $\{w\mathsf{c}w \mid w \in \{\mathsf{a}, \mathsf{b}\}^*\}$ is not context-free. We prove this using the proof pattern.
   Let $c, d \in \mathbb{N}$.
   Take $z = \mathsf{a}^k\mathsf{b}^k\mathsf{c}\mathsf{a}^k\mathsf{b}^k$ with $k = max(c, d)$.
   Let u, v, w, x, y be such that $z = uvwxy$, $|vx| > 0$ and $|vwx| \leq d$
   Note that our choice for k guarantees that $|vwx| \leq k$. The substring vwx has one of the following shapes:

   - vwx does not contain c.
   - vwx contains c.

   Take $i = 0$, then

   - If vwx does not contain c it is a substring at the left-hand side or at the right-hand side of c. Then it is impossible to write the string uwy as $s\mathsf{c}s$ for some string s, since only the number of terminals on one side of c is decreased.
   - If vwx contains c then the string vwx can be written as

     $$vwx \quad = \quad \mathsf{b}^s\mathsf{c}\mathsf{a}^t$$

     for some s, t respectively. For uwy there are two possibilities.
     The string uwy does not contain c so it is not an element of the language.
     If the string uwy does contain c then it has either fewer b's at the left-hand side than at the right-hand side or fewer a's at the right-hand side than at the left-hand side, or both. So it is impossible to write the string uwy as $s\mathsf{c}s$ for some string s.

2. The language is not regular because it is not context-free. The set of regular languages is a subset of the set of context-free languages.

9.13

1. The grammar G is context-free because there is only one nonterminal at the left hand side of the production rules and the right hand side is a sequence of terminals and nonterminals. The grammar is not regular because there is a production rule with two nonterminals at the right hand side.

2. $L(G) = \{(\mathsf{st})^* \mid \mathsf{s} \in \ \{^* \ \wedge \ \mathsf{t} \in \ \}^* \ \wedge \ |\mathsf{s}| = |\mathsf{t}| \ \}$
   $L(G)$ is all sequences of nested braces.

3. The language is context-free because it is generated by a context-free grammar.
   The language is not regular. We use the proof pattern and choose $\mathsf{s} = \{^n\}^n$. The proof is exactly the same as the proof for non-regularity of $L = \{\mathsf{a}^m\mathsf{b}^m \mid m \geq 0\}$ in Section 9.2.

9.14

1. The grammar is context-free. The grammar is not right-regular but left-regular because the nonterminal appears at the left hand side in the last production rule.

2. $L(G) = 0^* \cup 10^*$
   Note that the language is also generated by the following right-regular grammar:

$$\begin{array}{lcl} S & \rightarrow & \epsilon \\ S & \rightarrow & 1A \\ S & \rightarrow & 0A \\ A & \rightarrow & \epsilon \\ A & \rightarrow & 0 \\ A & \rightarrow & 0A \end{array}$$

3. The language is context-free and regular because it is generated by a regular grammar.

10.1 The stack machine produces the following sequence:

stack	input
S	acbab
AaS	acbab
aS	acbab
S	cbab
AaS	cbab
cSaS	cbab
SaS	bab
BaS	bab
baS	bab
aS	ab
S	b
B	b
b	b

10.2 For both grammars we have:

```
empty  =  const False
```

10.3 For `grammar1` we have:

```
firsts S = {b,c}
firsts A = {a,b,c}
firsts B = {a,b,c}
firsts C = {a,b}
```

For `grammar2`:

```
firsts S = {a}
firsts A = {b}
```

10.4 For `gramm1` we have:

```
follow S = {a,b,c}
follow A = {a,b,c}
follow B = {a,b}
follow C = {a,b,c}
```

For `gramm2`:

```
follow  =  const {}
```

10.5 For the productions of `gramm3` we have:

$$\begin{array}{lcl}
\texttt{lookAhead}\ (S \rightarrow A\texttt{a}S) & = & \{\texttt{a,c}\ \} \\
\texttt{lookAhead}\ (S \rightarrow B) & = & \{\texttt{b}\} \\
\texttt{lookAhead}\ (A \rightarrow \texttt{c}S) & = & \{\texttt{c}\} \\
\texttt{lookAhead}\ (A \rightarrow []) & = & \{\texttt{a}\} \\
\texttt{lookAhead}\ (B \rightarrow \texttt{b}) & = & \{\texttt{b}\}
\end{array}$$

Since all `lookAhead` sets for productions of the same nonterminal are disjoint, `gramm3` is an LL(1) grammar.

10.6 After left factoring `gramm2` we obtain `gramm2'` with productions

$$\begin{array}{lcl}
S & \rightarrow & \texttt{a}C \\
C & \rightarrow & \texttt{b}A \mid \texttt{a} \\
A & \rightarrow & \texttt{b}D \\
D & \rightarrow & \texttt{b} \mid S
\end{array}$$

For this transformed grammar `gramm2'` we have:

The `empty` function

```
empty  =  const False
```

The `first` function

```
firsts S = {a}
firsts C = {a,b}
firsts A = {b}
firsts D = {a,b}
```

The `follow` function

```
follow  =  const {}
```

The `lookAhead` function

$$\begin{array}{lcl}
\texttt{lookAhead}\ (S \rightarrow \texttt{a}C) & = & \{\texttt{a}\} \\
\texttt{lookAhead}\ (C \rightarrow \texttt{b}A) & = & \{\texttt{b}\} \\
\texttt{lookAhead}\ (C \rightarrow \texttt{a}) & = & \{\texttt{a}\} \\
\texttt{lookAhead}\ (A \rightarrow \texttt{b}D) & = & \{\texttt{b}\} \\
\texttt{lookAhead}\ (D \rightarrow \texttt{b}) & = & \{\texttt{b}\} \\
\texttt{lookAhead}\ (D \rightarrow S) & = & \{\texttt{a}\}
\end{array}$$

Clearly, `gramm2'` is an LL(1) grammar.

10.7 For the `empty` function we have:

```
empty R  =  True
empty _  =  False
```

For the `firsts` function we have

```
firsts L  =  {0,1}
firsts R  =  {,}
firsts B  =  {0,1}
```

The `follow` function is

```
follow B  =  {,}
follow _  =  {}
```

The `lookAhead` function is

$$\begin{array}{lcl}
\texttt{lookAhead}\ (L \rightarrow B\ R) & = & \{\texttt{0, 1}\} \\
\texttt{lookAhead}\ (R \rightarrow \epsilon) & = & \{\} \\
\texttt{lookAhead}\ (R \rightarrow \texttt{,}\ B\ R) & = & \{\texttt{,}\} \\
\texttt{lookAhead}\ (B \rightarrow \texttt{0}) & = & \{\texttt{0}\} \\
\texttt{lookAhead}\ (B \rightarrow \texttt{1}) & = & \{\texttt{1}\}
\end{array}$$

Since all `lookAhead` sets for productions of the same nonterminal are disjoint, the grammar is LL(1).

10.8

```
Node S [ Node c []
       , Node A [ Node c []
                , Node B [ Node c [], Node c [] ]
                , Node C [ Node b [], Node a [] ]
                ]
       ]
```

10.9

```
Node S [ Node a []
       , Node C [ Node b []
                , Node A [ Node b [], Node D [ Node b [] ] ]
                ]
       ]
```

10.10

```
Node S [ Node A []
       , Node a []
       , Node S [ Node A [ Node c [], Node S [ Node B [ Node b []]]]
                , Node a []
                , Node S [ Node B [ Node b [] ]]
                ]
       ]
```

10.11 The result of applying `ll1` to `gramm1` and the sentence `ccccba` is a tuple with the rose tree of exercise 10.8 as first element and the empty string as second element.

10.12

1. The second argument of the function `gll1` is a list of symbols representing the stack of the stack machine. The result of the function is a parser. Applying an input sentence (the input of the stack machine) to this parser results in a tuple. The first element of the tuple is a list of rose trees and the second argument is the rest of the input that could not be processed.

2. The function `gll1` is defined using pattern matching by describing the function call `gll1 grammar stack input` and defining the resulting tuple according to the value of the first symbol of the stack. Induction is used to process the whole stack.
   If the stack is empty, the input remains unprocessed. If the first symbol on the stack is a terminal then a match-operation is performed. If the first symbol is a nonterminal then an expand-operation is performed.

10.13

1.
```
list2Set :: Ord s => [s] -> [s]
list2Set = unions . map single
```

2.
```
list2Set :: Ord s => [s] -> [s]
list2Set = foldr op []
           where
           op x xs = single x `union` xs
```

3.
```
pref :: Ord s => (s -> Bool) -> [s] -> [s]
pref p = list2Set . takeWhile p
```

or

```
pref p [] = []
pref p (x:xs) = if p x then single x `union` pref p xs else []
```

or

```
pref p = foldr op []
          where
          op x xs = if p x then single x `union` xs else []
```

4.
```
prefplus p [] = []
prefplus p (x:xs) = if p x then single x `union` prefplus p xs
                           else single x
```

5.
```
prefplus p = foldr op []
             where
             op x us = if p x then single x `union` us
                              else single x
```

6.
```
  prefplus p
=
  foldr op []
  where
  op x us = if p x then single x `union` us else single x
=
  foldr op []
  where
  op x us = single x `union` rest
            where
            rest = if p x then us else []
=
  foldrRhs p single []
```

7. The function `foldrRhs p f []` takes a list `xs` and returns the set of all `f`-images of the elements of the prefix of `xs` all of whose elements satisfy `p` together with the `f`-image of the first element of `xs` that does not satisfy `p` (if this element exists).
   The function `foldrRhs p f start` takes a list `xs` and returns the set `start `union` foldrRhs p f [] xs`.

8.

10.14

1. For `gramm1`

   (a)
   ```
           foldrRhs empty first [] bSA
   =
           unions (map first (prefplus empty bSA))
   =           empty b = False
           unions (map first [b])
   =
           unions [{b}]
   =
           {b}
   ```
   (b)
   ```
   foldrRhs empty first [] Cb
   =
           unions (map first (prefplus empty Cb))
   =           empty C = False
           unions (map first [C])
   =
           unions [{a, b }]
   =
           { a, b }
   ```

2. For `gramm3`

   (a)
   ```
           foldrRhs empty first [] AaS
   =
           unions (map first (prefplus empty AaS))
   =           empty A = True
           unions (map first ([A] `union` prefplus empty aS))
               empty a = False
           unions (map first ([A] `union` [a]))
   =
           unions (map first ([A, a]))
   =
           unions [first A, first a]
   =
           unions [{c}, {a}]
   =
           {a, c}
   ```
   (b)
   ```
           scanrRhs empty first [] AaS
   =
           map (foldrRhs empty first []) (tails AaS)
   =
   ```

```
        map (foldrRhs empty first []) [AaS, aS, S, []]
=
        [ foldrRhs empty first [] AaS
        , foldrRhs empty first [] aS
        , foldrRhs empty first [] S
        , foldrRhs empty first [] []
        ]
=           calculus
        [ {a, c}, {a}, {a, b, c }, [] ]
```

11.1 Recall, for `gramm1` we have:

```
follow S = {a,b,c}
follow A = {a,b,c}
follow B = {a,b}
follow C = {a,b,c}
```

The production rule $B \rightarrow$ `cc` cannot be used for a reduction when the input begins with the symbol `c`.

stack	input
	ccccba
c	cccba
cc	ccba
ccc	cba
cccc	ba
Bcc	ba
bBcc	a
abBcc	
CBcc	
Ac	
S	

11.2 Recall, for `gramm2` we have:

```
follow S = {}
follow A = {}
```

No reduction can be made until the input is empty.

stack	input
	abbb
a	bbb
ba	bb
bba	b
bbba	
Aba	
S	

11.3 Recall, for `gramm3` we have:

```
follow S = {a}
follow A = {a}
follow B = {a}
```

At the beginning a reduction is applied using the production rule $A \to \varepsilon$.

stack	input
	acbab
A	acbab
aA	cbab
caA	bab
bcaA	ab
BcaA	ab
ScaA	ab
AaA	ab
aAaA	b
baAaA	
SaAaA	
SaA	
S	