Languages and Compilers

Johan Jeuring, Doaitse Swierstra

November 16, 2025

Contents

Preface ...

... for the 2025 version

I am updating the lecture notes of the course on "Languages and Compilers" to reflect the current contents of the course. After being away from the course for many years, I plan to teach it again in the coming years, and to produce an updated version of the lecture notes.

I will gradually process the chapters, and publish them on the course website in installments.

As ever, all comments are welcome, and I will acknowledge your contributions in the updated preface of these notes.

Johan Jeuring

November 2025

... for the 2009 version

This is a work in progress. These lecture notes are in large parts identical with the old lecture notes for "Grammars and Parsing" by Johan Jeuring and Doaitse Swierstra.

The lecture notes have not only been used at Utrecht University, but also at the Open University. Some modifications made by Manuela Witsiers have been reintegrated.

I have also started to make some modifications to style and content – mainly trying to adapt the Haskell code contained in the lecture notes to currently established coding guidelines. I also rearranged some of the material because I think they connect better in the new order.

However, this work is not quite finished, and this means that some of the later chapters look a bit different from the earlier chapters. I apologize in advance for any inconsistencies or mistakes I may have introduced.

Please feel free to point out any mistakes you find in these lecture notes to me – any other feedback is also welcome.

Preface ...

I hope to be able to update the online version of the lecture notes regularly.

Andres Löh

October 2009

... for the first version

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, Alexander Elyasov, Matthias Felleisen, 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

1. Goals

Introduction

A course on *Grammars*, *Parsing* and *Compilation of programming languages* has always been one of the core components of a computer science curriculum. The main reason for this is that these concepts are at the core of the technology we develop using computers: programming languages allow us to express our computational ideas, and compilers help us with creating software that runs on a computer. 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 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 or extremely hard to create by hand.

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 computer scientist to have. The primary, somewhat more traditional, goals are:

- to *describe* structures (i.e., "formulas") using *grammars*;
- to *parse*, i.e., to recognise (build) such structures in (from) a sequence of symbols;
- to *analyse* grammars to determine whether or not specific properties hold;
- to *compose* components such as parsers, analysers, and code generators
- to *apply these techniques* in the construction of all kinds of programs;
- to explain and prove why certain problems can or cannot be described by means of formalisms such as context-free grammars or finite-state automata.

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 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 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 programs by 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 gigabytes), 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 Haskell, 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 grammar 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 determine 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.

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 two of 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 grammars for natural languages is that the grammatical formalism we will use is precisely defined, and doesn't allow for deviations. This enables 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*. This is rather different from grammars for natural languages, where people regularly disagree about whether something is correct English or not. However, this formal approach also comes with a disadvantage; the expressiveness of the class of grammars we are going to describe in this chapter is a bit limited, and there are many languages one might want to describe but which cannot be described, given the limitations of the formalism. But the formalism is powerful enough to be used for almost all programming languages.

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 be able to:

- describe the concepts of *language* and *sentence*;
- describe a language by means of a *context-free grammar*;
- describe the difference between a *terminal* symbol and a *nonterminal* symbol;
- read and interpret the *BNF* notation;
- *derive* a sentence of a language using a context-free grammar;
- construct a *parse tree*;
- construct a datatype corresponding to a context-free grammar;
- read and interpret the *EBNF* notation;
- describe the relation between *concrete* and *abstract syntax*;
- convert a grammar from EBNF-notation into BNF-notation by hand;
- construct a simple context-free grammar in EBNF notation;
- verify whether or not a simple grammar is *ambiguous*;
- *transform* a grammar, for example for removing left recursion.

2.1. Languages

This section introduces 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:

sequence

Definition 2.1 (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 z is a sequence and a is an element of X, then az is also a sequence.

induction

The above definition is an instance of a very common definition pattern: it is a *definition by induction*, i. e., the definition of the concept refers to the concept itself. It is implicitly understood that nothing that cannot be formed by repeated, but finite application of one of the two given rules is a sequence over X.

Furthermore, the definition corresponds almost exactly to the definition of the type $[a]$ of *lists* with elements of type a in Haskell. The one difference is that Haskell lists can be infinite, whereas sequences are always finite.

In the following, we will introduce several concepts based on sequences. They can be implemented easily in Haskell using lists.

Functions that operate on an inductively defined structure such as sequences are typically *structurally recursive*, i. e., such definitions often follow a recursion pattern which is similar to the definition of the structure itself. The function *foldr* which 'folds' over a list is a typical example.

Note that the Haskell notation for lists is generally more precise than the mathematical notation for sequences. When talking about languages and grammars, we often leave the distinction between single symbols and sequences implicit.

We use letters from the beginning of the alphabet to represent single symbols, and letters from the end of the alphabet to represent sequences. We write a to denote both the single symbol a or the sequence $a\varepsilon$, depending on context. We typically use ε only when we want to explicitly emphasize that we are talking about the empty sequence.

Furthermore, we denote concatenation of sequences and symbols in the same way, i. e., az should be understood as the symbol a followed by the sequence z, whereas xy is the concatenation of sequences x and y.

In Haskell, all the differences between these constructions are explicit. Elements are distinguished from lists by their type; there is a clear difference between a and $[a]$. Concatenation of lists is handled by the operator $(+\!\!+)$, whereas a single element can be added to the front of a list using (:). Also, Haskell identifiers often have longer names, so ab in Haskell is to be understood as a single identifier with name ab, not as a combination of two symbols a and b.

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

Definition 2.2 (Alphabet, Language, Sentence)**.**

- An *alphabet* is a finite set of *symbols*.
- A *language* is a subset of T^*, for some alphabet T.
- A *sentence* (often also called *word*) is an element of a language.

alphabet
language
sentence

Note that 'word' and 'sentence' in formal languages are used as synonyms.

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 $\{\mathtt{if}, \mathtt{then}, \mathtt{else}\}$;
- a set of characters $l = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{i}, \mathtt{k}, \mathtt{l}, \mathtt{m}, \mathtt{n}, \mathtt{o}, \mathtt{p}, \mathtt{r}, \mathtt{s}, \mathtt{t}, \mathtt{u}, \mathtt{w}, \mathtt{x}\}$;
- a set of English words $\{\mathtt{course}, \mathtt{practical}, \mathtt{exercise}, \mathtt{exam}\}$.

Examples of languages are:

- T^*, $\emptyset$ (the empty set), $\{\varepsilon\}$ and T are languages over alphabet T;

- the set $\{\texttt{course}, \texttt{practical}, \texttt{exercise}, \texttt{exam}\}$ is a language over the alphabet 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 $\{n \mid n \in \{0,1,\ldots,9\}^*, n \bmod 2 = 0\}$;
- the language *DNA-palindromes* of palindromes in DNA sequences, sequences which read the same forward as backward, over the alphabet $\{\texttt{A}, \texttt{C}, \texttt{T}, \texttt{G}\}$: $\{s | s \in \{\texttt{A}, \texttt{C}, \texttt{T}, \texttt{G}\}^*, s = s^R\}$, where s^R denotes the reverse of sequence s. CRISPR-CAS, the defense mechanism against viruses of DNA makes fundamental use of *DNA-palindromes*. CRISPR is an abbreviation for Clustered Regularly Interspaced Palindromic Repeats. The reverse function on DNA sequences is slightly different from the usual reverse function, but we will ignore that for now.

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.

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 $\overline{L} = \{x \mid x \in T^*, x \notin L\}$.

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 **??**. Note that $\cup$ denotes set union, so $\{1,2\} \cup \{1,3\} = \{1,2,3\}$.

Definition 2.3 (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	$= \{\varepsilon\}$	0^{th} power of L
L^{n+1}	$= LL^n$	$n+1^{\text{st}}$ power of L
L^*	$= \bigcup_{i \in \mathbb{N}} L^i \quad = L^0 \cup L^1 \cup L^2 \cup \ldots$	star-closure of L
L^+	$= \bigcup_{i \in \mathbb{N}, i>0} L^i = L^1 \cup L^2 \cup \ldots$	positive closure of L

The following equations follow immediately from the above definitions.

$$L^* = \{\varepsilon\} \cup LL^*$$
$$L^+ = LL^*$$

Exercise 2.1. Let $L = \{\texttt{ab}, \texttt{aa}, \texttt{baa}\}$, where `a` and `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. $\{\varepsilon\}L = L\{\varepsilon\} = L$

Exercise 2.4. In this section we defined two "star" operators: one for arbitrary sets (Definition 2.1) and one for languages (Definition 2.3). Is there a difference between these operators?

2.2. Grammars

This section introduces the concept of context-free grammars.

Manipulating sets, and proving properties about them, is often challenging. 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 automatic 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 *DNA-palindromes*, the language of palindromes over the alphabet of DNA symbols, 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 2.4 (DNA-palindromes by induction)**.**

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

  $$\texttt{A}P\texttt{A}$$
  $$\texttt{C}P\texttt{C}$$
  $$\texttt{T}P\texttt{T}$$
  $$\texttt{G}P\texttt{G}$$

  are DNA-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 *DNA-palindromes* inductively defined using the above definition read the same forwards and backwards. Therefore this definition is said to be *sound* (every string in *DNA-palindromes* is a DNA-palindrome). Conversely, if a string consisting of `A`'s, `C`'s, `T`'s, and `G`'s reads the same forwards and backwards then it belongs to the language *DNA-palindromes*. Therefore this definition is said to be *complete* (every DNA-palindrome is in *DNA-palindromes*).

sound

complete

Finding an inductive definition for a language which is described by a predicate (like the one for DNA-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 DNA-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:

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

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 *DNA-palindromes*, which we will call P, as follows:

$$\begin{array}{l}
\vdash \varepsilon \in P \\
\vdash \texttt{A} \in P \\
\vdash \texttt{C} \in P \\
\vdash \texttt{T} \in P \\
\vdash \texttt{G} \in P \\
p \in P \vdash \texttt{A}p\texttt{A} \in P \\
p \in P \vdash \texttt{C}p\texttt{C} \in P \\
p \in P \vdash \texttt{T}p\texttt{T} \in P \\
p \in P \vdash \texttt{G}p\texttt{G} \in P
\end{array}$$

Although the definition of P is completely formal, it is still laborious to write. Since in computer science we use many definitions which follow such a pattern, we introduce a shorthand for it, called a *grammar*. A grammar consists of production rules. We can

grammar

give a grammar for P by translating the deduction rules given above into production rules. The rule with which the empty string is constructed is:

$$P \to \varepsilon$$

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 DNA-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 DNA-palindromes consisting of just one character. Each of the one element strings `A`, `C`, `T`, and `G` is a DNA-palindrome, and gives rise to a production:

production rule

nonterminal

$$P \to \texttt{A}$$
$$P \to \texttt{C}$$
$$P \to \texttt{T}$$
$$P \to \texttt{G}$$

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

$$P \to \texttt{A}P\texttt{A}$$
$$P \to \texttt{C}P\texttt{C}$$
$$P \to \texttt{T}P\texttt{T}$$
$$P \to \texttt{G}P\texttt{G}$$

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

- the set of *terminals* $\{\texttt{A}, \texttt{C}, \texttt{T}, \texttt{G}\}$;
- the set of nonterminals $\{P\}$;
- and the set of productions (the nine 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, and we always have to select one to start with.

start symbol

To summarize, we obtain the following grammar for *DNA-palindromes*:

$$
\begin{aligned}
P &\to \varepsilon \\
P &\to \mathtt{A} \\
P &\to \mathtt{C} \\
P &\to \mathtt{T} \\
P &\to \mathtt{G} \\
P &\to \mathtt{A}P\mathtt{A} \\
P &\to \mathtt{C}P\mathtt{C} \\
P &\to \mathtt{T}P\mathtt{T} \\
P &\to \mathtt{G}P\mathtt{G}
\end{aligned}
$$

The definition of the set of terminals, $\{\mathtt{A}, \mathtt{C}, \mathtt{T}, \mathtt{G}\}$, and the set of nonterminals, $\{P\}$, is often implicit. Also the start-symbol is implicitly defined here since there is only one nonterminal.

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

context-free grammar

Definition 2.5 (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 $A \to \alpha$, 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 on 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 $DNA - palindromes$ 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

$$
\begin{aligned}
S &\to \alpha \\
S &\to \beta
\end{aligned}
$$

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

$$S \to \alpha \mid \beta$$

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:

$$P \to \varepsilon \mid \mathtt{A} \mid \mathtt{C} \mid \mathtt{T} \mid \mathtt{G} \mid \mathtt{A}P\mathtt{A} \mid \mathtt{C}P\mathtt{C} \mid \mathtt{T}P\mathtt{T} \mid \mathtt{G}P\mathtt{G}$$

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, in particular for the language ALGOL 60.

BNF

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,

Alpha rule: $S \to \alpha$
Beta rule: $S \to \beta$

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 2.5. Give a context free grammar for the set of sentences over alphabet X where

1. $X = \{\mathtt{a}\}$,
2. $X = \{\mathtt{a}, \mathtt{b}\}$.

Exercise 2.6. Give a context free grammar for the language

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

Exercise 2.7. Give a grammar for *palindromes* over the alphabet $\{\mathtt{a}, \mathtt{b}\}$.

Exercise 2.8. Give a grammar for the language

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

This language is known as the language of *mirror palindromes*.

Exercise 2.9. 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 2.10. Give a grammar for the language

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

where $\#(c, w)$ 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.

In the previous section, we have given an example of how to construct a grammar for a particular language. Now we consider the reverse question: how can we obtain a language from a given grammar? Before we can answer this question we first have to show what we can do with a grammar. The answer is simple: we can derive sequences with it.

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

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

$$\begin{aligned} & P \\ \Rightarrow\ & \texttt{C}P\texttt{C} \\ \Rightarrow\ & \texttt{CA}P\texttt{AC} \\ \Rightarrow\ & \texttt{CATAC} \end{aligned}$$

derivation

Such a construction is called a *derivation*. In the first step of this derivation production $P \to \texttt{C}P\texttt{C}$ is used to rewrite P into $\texttt{C}P\texttt{C}$. In the second step production $P \to \texttt{A}P\texttt{A}$ is used to rewrite $\texttt{C}P\texttt{C}$ into $\texttt{CA}P\texttt{AC}$. Finally, in the last step production $P \to \texttt{T}$ is used to rewrite $\texttt{CA}P\texttt{AC}$ into `CATAC`. Constructing a derivation can be seen as a constructive proof that the string `CATAC` is a DNA-palindrome.

We will now describe derivation steps formally.

Definition 2.6 (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_0, written $\varphi_0 \Rightarrow^* \varphi_n$, if there exist sequences $\varphi_0, \ldots, \varphi_n$ such that

direct derivation

derivation

$$\forall i, 0 \leqslant i < n : \quad \varphi_i \Rightarrow \varphi_{i+1}$$

If $n = 0$, this statement is trivially true, and it follows that we can derive each sentence φ from itself in zero steps:

$$\varphi \Rightarrow^* \varphi$$

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

partial derivation

Finding a derivation $\varphi_0 \Rightarrow^* \varphi_n$ 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

$$P \Rightarrow^* \texttt{CATAC}$$

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 `CATAC` belongs to the language generated by the grammar for DNA-palindromes. In general, we define

language of a grammar

Definition 2.7 (Language of a grammar)**.** The *language of a grammar* $G{=}(T, N, R, S)$, usually denoted by $L(G)$, is defined as

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

The language $L(G)$ is also called *the language generated by the grammar* G. We sometimes also talk about the language of a nonterminal A, which is defined by

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

Different grammars may have the same language. For example, if we extend the grammar for *DNA-palindromes* with the production $P \to \texttt{CATAC}$, we obtain a grammar with exactly the same language as *DNA-palindromes*. 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 construct many grammars that generate the language. To phrase it more mathematically: the mapping between a grammar and its language is not a bijection.

equivalent

Definition 2.8 (Context-free language)**.** A *context-free language* is a language that is generated by a context-free grammar.

context-free language

All DNA-palindromes can be derived from the start symbol P. Thus, the language of our grammar for DNA-palindromes is $DNA - palindromes$, the set of all palindromes over the alphabet $\{\texttt{A}, \texttt{C}, \texttt{T}, \texttt{G}\}$, and *DNA-palindromes* is context-free.

2.3.1. Examples of basic languages

Digits occur in a several programming languages 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 frequently in later sections.

- The language of single digits is specified by a grammar with ten 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:

  $Digs \rightarrow \varepsilon \mid Dig\ Digs$

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

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

  Now we can define the language of natural numbers as follows.

  $Nat \rightarrow 0 \mid Dig\text{-}0\ Digs$

- 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.

  $Sign \rightarrow \texttt{+} \mid \texttt{-}$
  $Int \rightarrow Sign\ Nat \mid Nat$

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

  $SLetter \rightarrow \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z}$
  $CLetter \rightarrow \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z}$

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

  $Letter \rightarrow SLetter \mid CLetter$

- 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:

  $$\begin{aligned} \mathit{Identifier} &\to \mathit{Letter}\ \mathit{AlphaNums} \\ \mathit{AlphaNums} &\to \varepsilon \mid \mathit{Letter}\ \mathit{AlphaNums} \mid \mathit{Dig}\ \mathit{AlphaNums} \end{aligned}$$

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

- Dutch zip codes consist of four digits, of which the first digit is non-zero, followed by two capital letters. So

  $$\mathit{ZipCode} \to \mathit{Dig\text{-}0}\ \mathit{Dig}\ \mathit{Dig}\ \mathit{Dig}\ \mathit{CLetter}\ \mathit{CLetter}$$

Exercise 2.11. A terminal string that belongs to the language of a grammar is always derived in one or more steps from the start symbol of the grammar. Why?

Exercise 2.12. What language is generated by the grammar with the single production rule

$$S \to \varepsilon$$

Exercise 2.13. What language does the grammar with the following productions generate?

$$\begin{aligned} S &\to A\mathtt{a} \\ A &\to B \\ B &\to A\mathtt{a} \end{aligned}$$

Exercise 2.14. Give a simple description of the language generated by the grammar with productions

$$\begin{aligned} S &\to \mathtt{a}A \\ A &\to \mathtt{b}S \\ S &\to \varepsilon \end{aligned}$$

Exercise 2.15. Is the language L defined in Exercise 2.1 context free ?

2.4. Parse trees

This section introduces parse trees, and shows 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. As a consequence, there may be different derivations for the same sentence.

However, if only the order in which the derivation steps are chosen differs between two derivations, then the derivations are considered to be equivalent. If, however, different derivation steps have been chosen in two derivations, then these derivations are considered to be different.

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

$$S \to SS$$
$$S \to \texttt{0} \mid \texttt{1}$$

Using this grammar, we can derive the sentence `100` in at least the following two ways (the nonterminal that is rewritten in each step appears underlined):

$$\underline{S} \Rightarrow S\underline{S} \Rightarrow S\underline{S}S \Rightarrow \underline{S}\texttt{0}S \Rightarrow \texttt{10}\underline{S} \Rightarrow \texttt{100}$$
$$\underline{S} \Rightarrow \underline{S}S \Rightarrow \texttt{1}\underline{S} \quad \Rightarrow \texttt{1}S\underline{S} \Rightarrow \texttt{1}\underline{S}\texttt{0} \Rightarrow \texttt{100}$$

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:

$$\underline{S} \Rightarrow \underline{S}S \Rightarrow \underline{S}SS \Rightarrow \texttt{1}\underline{S}S \Rightarrow \texttt{10}\underline{S} \Rightarrow \texttt{100}$$

In both derivation sequences above, the first S was rewritten to `1`. In this derivation, however, the first S is rewritten to SS.

leftmost derivation

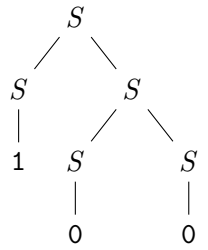
The set of all equivalent derivations can be represented by selecting a so-called *canonical element*. A good candidate for such a canonical element is the *leftmost derivation*. In a leftmost derivation, the leftmost nonterminal is rewritten in each step. If there exists a derivation of a sentence x using the productions of a grammar, then there also exists a leftmost derivation of x. The last of the three derivation sequences for the sentence `100` given above is a leftmost derivation. The two equivalent derivation sequences before, however, are both not leftmost. The leftmost derivation corresponding to these two sequences above is

$$\underline{S} \Rightarrow \underline{S}S \Rightarrow \texttt{1}\underline{S} \quad \Rightarrow \texttt{1}\underline{S}S \Rightarrow \texttt{10}\underline{S} \Rightarrow \texttt{100}$$

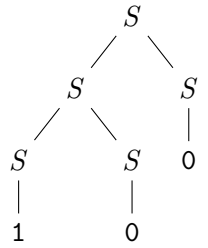
parse tree

There exists another convenient way to represent equivalent derivations: they all correspond to 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 `100` looks as follows:



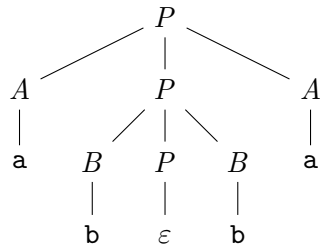
The third derivation of the sentence `100` results in a different parse tree:



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

$P \to \varepsilon$
$P \to APA$
$P \to BPB$
$A \to$ `a`
$B \to$ `b`

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.

unambiguous
ambiguous

Definition 2.9 (ambiguous grammar, 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*.

The grammar *SequenceOfBits* for constructing sequences of bits is an example of an ambiguous grammar, since there exist two parse trees for the sentence `100`.

It is in general undecidable whether or not an arbitrary context-free grammar is ambiguous. This means that it is impossible to write a program that determines for an arbitrary context-free grammar whether or not it is ambiguous.

Translating languages with ambiguous grammars to machine code or another format is often rather difficult. For this reason 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'. While ambiguity of natural languages may perhaps be considered as an advantage for their users (some politicians make frequent use of it), it certainly is considered a disadvantage for language translators, because it is usually impossible to maintain an ambiguous meaning in a translation.

2.5. Grammar transformations

This section studies properties of grammars, and means by which we can systematically transform a grammar into another grammar that describes the same language and satisfies particular properties.

Some examples of properties that are worth considering for a particular grammar 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.

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 certain information from parse trees of a grammar.

Properties are particularly interesting in combination with grammar transformations. A *grammar transformation*is a procedure to obtain a grammar G' from a grammar G such that $L(G') = L(G)$.

grammar transformation

Suppose, for example, that we have a program that builds parse trees for sentences of grammars in Chomsky normal form, and that we can prove that every grammar can be transformed in a grammar in Chomsky normal form. Then we can use this program for building parse trees for any grammar.

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 into other grammars that generate the same language, but that possibly satisfy different properties. In the following, we describe a number of grammar transformations:

- Removing duplicate productions.
- Substituting right hand sides for nonterminals.
- Removing unreachable productions.
- 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, we will assume that N is the set of nonterminals, T is the set of terminals, and that u, v, w, x, y and z denote sequences of terminals and nonterminals, i. e., are elements of $(N \cup T)^*$.

2.5.1. 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,

$$A \to u \mid u \mid v$$

can be transformed into

$$A \to u \mid v$$

2.5.2. Substituting right hand sides for nonterminals

If a nonterminal X occurs in a right hand side of a production, the production may be replaced by just as many productions as there exist productions for X, in which X has been replaced by its right hand sides. For example, we can substitute B in the right hand side of the first production in the following grammar:

$$A \to uBv \mid z$$
$$B \to x \mid w$$

The resulting grammar is:

$$A \to uxv \mid uwv \mid z$$
$$B \to x \mid w$$

2.5.3. Removing unreachable productions

Consider the result of the transformation above:

$$A \to uxv \mid uwv \mid z$$
$$B \to x \mid w$$

If A is the start symbol and B does not occur in any of the symbol sequences u, v, w, x, z, then the second production can never occur in the derivation of a sentence starting from A. In such a case, the unreachable production can be dropped:

$$A \to uxv \mid uwv \mid z$$

2.5.4. Left factoring

left factoring

Left factoring is a grammar transformation that is applicable 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:

$$A \to xy \mid xz \mid v$$

may be transformed into

$$A \to xZ \mid v$$
$$Z \to y \mid z$$

where Z is a new nonterminal. As we will see in Chapter 3, parsers can be constructed systematically from a grammar, and left factoring the grammar before constructing a parser can dramatically improve the performance of the resulting parser.

2.5.5. Removing left recursion

left recursion

A production is called *left-recursive* if the right-hand side starts with the nonterminal of the left-hand side. For example, the production

$$A \to Az$$

is left-recursive. A grammar is left-recursive if we can derive $A \Rightarrow^+ Az$ for some nonterminal A of the grammar (i.e., if we can derive Az from A in one or more steps).

Left-recursive grammars are sometimes undesirable – we will, for instance, see in Chapter 3 that a parser constructed systematically from a left-recursive grammar may loop. Fortunately, left recursion can be removed by transforming the grammar. The following transformation removes left-recursive productions.

To remove the left-recursive productions of a nonterminal A, we divide the productions for A in sets of left-recursive and non left-recursive productions. We can thus factorize the productions for A as follows:

$$A \to Ax_1 \mid Ax_2 \mid \ldots \mid Ax_n$$
$$A \to y_1 \mid y_2 \mid \ldots \mid y_m$$

where none of the symbol sequences $y_1, \ldots, y_m$ starts with A. We now add a new nonterminal Z, and replace the productions for A by:

$$A \to y_1 \mid y_1 Z \mid \ldots \mid y_m \mid y_m Z$$
$$Z \to x_1 \mid x_1 Z \mid \ldots \mid x_n \mid x_n Z$$

Note that this procedure only works for a grammar that is *directly* left-recursive, i. e., a grammar that contains a left-recursive production of the form $A \to Ax$.

Grammars can also be indirectly left recursive. An example is

$$A \to Bx$$
$$B \to Ay$$

None of the two productions is left recursive, but we can still derive $A \Rightarrow^* Ayx$. Removing left recursion in an indirectly left-recursive grammar is also possible, but a bit more complicated [1].

Here is an example of how to apply the procedure described above to a grammar that is directly left recursive, namely the grammar *SequenceOfBits* that we have introduced in Section 2.4:

$$S \to SS$$
$$S \to \texttt{0} \mid \texttt{1}$$

The first production is left-recursive. The second is not. We can thus directly apply the procedure for left recursion removal, and obtain the following productions:

$S \to \texttt{0} \mid \texttt{1} \mid \texttt{0}Z \mid \texttt{1}Z$
$Z \to S \mid SZ$

2.5.6. Associative separator

The following grammar fragment generates a list of declarations, separated by a semicolon '`;`':

$Decls \to Decls\ \texttt{;}\ Decls$
$Decls \to Decl$

The productions for *Decl*, which generates a single declaration, have been omitted. This grammar is ambiguous, for the same reason as *SequenceOfBits* is ambiguous. The operator `;` is an associative separator in the generated language, that is, it does not matter how we group the declarations; given three declarations d_1, d_2, and d_3, the meaning of d_1 `;` (d_2 `;` d_3) and (d_1 `;` d_2) `;` d_3 is the same. Therefore, we may use the following unambiguous grammar for generating a language of declarations:

$Decls \to Decl\ \texttt{;}\ Decls$
$Decls \to Decl$

Note that in this case, the transformed grammar is also no longer left-recursive.

An alternative unambiguous (but still left-recursive) grammar for the same language is

$Decls \to Decls\ \texttt{;}\ Decl$
$Decls \to Decl$

The grammar transformation just described should be handled with care: if the separator is associative in the generated language, like the semicolon in this case, applying the transformation is fine. However, if the separator is not associative, then removing the ambiguity in favour of a particular nesting is dangerous.

This grammar transformation is often useful for expressions that are separated by associative operators, such as for example natural numbers and addition.

2.5.7. 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:

$E \to E$ `+` E
$E \to E$ `*` E
$E \to$ `(` E `)`
$E \to Digs$

where *Digs* generates a list of digits as described in Section 2.3.1.

This grammar is ambiguous: for example, 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`. To obtain parse trees that respect these priorities, we transform the grammar as follows:

$E \to T$
$E \to E$ `+` T
$T \to F$
$T \to T$ `*` F
$F \to$ `(` E `)`
$F \to Digs$

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 nearly identically formed production rules, we can abbreviate the grammar by using parameterised nonterminals. For $1 \leqslant i < n$, we get productions

$E_i \to E_{i+1}$
$E_i \to E_i\ Op_i\ E_{i+1}$

The nonterminal Op_i is parameterised and 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:

$E_n \to$ `(` E_1 `)` $\mid Digs$

2.5.8. Discussion

We have presented several examples of grammar transformations. A grammar transformation transforms a grammar into another grammar that generates the same language. For each of the above transformations we should therefore prove that the generated language remains the same. Since the proofs are too complicated at this point, they are omitted. Proofs can be found in any of the theoretical books on language and parsing theory [11].

There exist many other 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 2.16. Consider the following ambiguous grammar with start symbol A:

$A \to A\texttt{a}A$
$A \to \texttt{b} \mid \texttt{c}$

Transform the grammar by applying the rule for associative separators. Choose the transformation such that the resulting grammar is also no longer left-recursive.

Exercise 2.17. The standard example of ambiguity in programming languages is the *dangling else*. Let G be a grammar with terminal set $\{\texttt{if}, \texttt{b}, \texttt{then}, \texttt{else}, \texttt{a}\}$ and the following productions:

$S \to \texttt{if b then}\ S\ \texttt{else}\ S$
$S \to \texttt{if b then}\ S$
$S \to \texttt{a}$

1. Give two parse 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 2.18. A *bit list* is a nonempty list of bits separated by commas. A grammar for bit lists is given by

$L \to B$
$L \to L\ \texttt{,}\ L$
$B \to \texttt{0} \mid \texttt{1}$

Remove the left recursion from this grammar.

Exercise 2.19. Consider the follwing grammar with start symbol S:

$S \to AB$
$A \to \varepsilon \mid \texttt{aa}A$
$B \to \varepsilon \mid B\texttt{b}$

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

2.6. Concrete and abstract syntax

This section describes how we can represent context-free grammars using Haskell datatypes. To this end, we introduce the notion of abstract syntax, and show how to obtain an abstract syntax from a concrete syntax.

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 *SequenceOfBits*:

$S \to SS$
$S \to \texttt{0} \mid \texttt{1}$

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

Beside: $S \to SS$
Zero: $S \to \texttt{0}$
One: $S \to \texttt{1}$

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
       | Zero
       | One
```

Note that the nonterminals on the right hand side of Beside reappear as arguments of the constructor *Beside*. On the other hand, the terminal symbol `0` (or `1`) in the production Zero (or One) is omitted in the definition of the constructor *Zero* (or *One*.

One might be tempted to create the following definition instead:

```
data S' = Beside' S' S'
        | Zero' Char   — too general
        | One' Char    — too general
```

However, this datatype is too general for the given grammar. An argument of type *Char* can be instantiated to any single character, but we know that this character always has to be `0` or `1`. Since there is no choice anyway, there is no extra value in storing that `0` or `1`, and the first datatype S serves the purpose of encoding the parse trees of the grammar *SequenceOfBits* just fine.

For example, the parse tree that corresponds to the first two derivations of the sequence `100` is represented by the following value of the datatype S:

```
Beside One (Beside Zero Zero)
```

The third derivation of the sentence `100` produces the following parse tree:

$$Beside\ (Beside\ One\ Zero)\ Zero$$

To emphasize that these representations contain sufficient information to reproduce the original strings, we can write a function that performs this conversion:

$$\begin{aligned} &sToString :: S \to String \\ &sToString\ (Beside\ l\ r) = sToString\ l +\!\!+ sToString\ r \\ &sToString\ Zero \quad\quad = \texttt{"0"} \\ &sToString\ One \quad\quad = \texttt{"1"} \end{aligned}$$

Applying the function *sToString* to either *Beside One* (*Beside Zero Zero*) or the alternative *Beside* (*One Single Zero*) *Zero* yields the string `"100"`.

By literally translating the nonterminal S to a datatype name we get a name without a meaning. It would be better to call the datatype *Bits*. We chose to use S to show that the translation process of a context-free grammar to a datatype follows a set of standard rules, but improving names in the process is of course a good thing to do.

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 nonterminal S is given by the grammar *SequenceOfBits*.

abstract syntax

On the other hand, an *abstract syntax* of a language describes the shapes of parse trees of the language, without the need to refer to concrete terminal symbols. Parse trees are therefore often also called abstract syntax trees. The datatype S is an example of an abstract syntax for the language of *SequenceOfBits*. The adjective 'abstract' indicates that values of the abstract syntax do not need to explicitly contain all information about particular sentences, as long as that information is recoverable, as for example by applying function *sToString*.

semantic function

A function such as *sToString* 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. In this example, the meaning of a more abstract representation is expressed in terms of a concrete representation.

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

$$\begin{aligned} &S \to \texttt{0}Z \mid \texttt{1}Z \mid \texttt{0} \mid \texttt{1} \\ &Z \to SZ \mid S \end{aligned}$$

An abstract syntax of this grammar may be given by

$$\begin{aligned} &\textbf{data}\ SA = ConsZero\ Z \mid ConsOne\ Z \mid ZeroS \mid OneS \\ &\textbf{data}\ Z\ \ = ConsZ\ SA\ Z \mid SingleZ\ SA \end{aligned}$$

For each nonterminal in the original grammar, we have introduced a corresponding datatype. For each production for a particular nonterminal (expanding all the alternatives into separate productions), we have introduced a constructor and invented a name for the constructor. The nonterminals on the right hand side of the production rules appear as arguments of the constructors, but the terminals disappear, because that information can be recovered from the constructors.

We obtained the abstract syntax datatype definitions by mechanically applying transformation rules to the context-free grammar for *SequenceOfBits*. Since we know that the grammar describes sequences of bits, we can also use a less precise abstract syntax such as the type of lists of integers $[Int]$, where we assume that the *Int*-values are either 0 or 1. But just as the S' datatype, this abstract syntax definition also includes values that are not representations of sentences of the grammar, such as sequences that contain the integer 2.

The *SequenceOfBits* example shows that one may choose between many different abstract syntaxes for a given grammar. The choice of an abstract syntax over another should therefore be determined by the demands of the application, i.e., by what we ultimately want to compute.

Exercise 2.20. The following Haskell datatype represents a limited form of arithmetic expressions

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

Give a grammar for a suitable concrete syntax corresponding to this datatype.

Exercise 2.21. Consider the grammar for palindromes that you have constructed in Exercise 2.7. Give parse trees for the palindromes $pal_1 =$ `"abaaba"` and $pal_2 =$ `"baaab"`. Define a datatype *Pal* corresponding to the grammar and represent the parse trees for pal_1 and pal_2 as values of *Pal*.

Exercise 2.22. Consider your answers to Exercises 2.7 and 2.21 where we have given a grammar for palindromes over the alphabet $\{\mathtt{a}, \mathtt{b}\}$ and a Haskell datatype describing the abstract syntax of such palindromes.

1. Write a semantic function that transforms an abstract representation of a palindrome into a concrete one. Test your function with the palindromes pal_1 and pal_2 from Exercise 2.21.
2. Write a semantic function that counts the number of `a`'s occurring in a palindrome. Test your function with the palindromes pal_1 and pal_2 from Exercise 2.21.

Exercise 2.23. Consider your answer to Exercise 2.8, 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 $aMir_1$ and $aMir_2$ that correspond to the concrete mirror palindromes $cMir_1 =$ `"abaaba"` and $cMir_2 =$ `"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 $aMir_1$ and $aMir_2$.
3. Write a function that transforms an abstract representation of a mirror palindrome into the corresponding abstract representation of a palindrome (using the datatype from Exercise 2.21). Test your function with the abstract mirror palindromes $aMir_1$ and $aMir_2$.

Exercise 2.24. Consider your answer to Exercise 2.9, which describes the concrete syntax for parity sequences.

1. Define a datatype *Parity* describing the abstract syntax corresponding to your grammar. Give the two abstract parity sequences $aEven_1$ and $aEven_2$ that correspond to the concrete parity sequences $cEven_1 =$ `"00101"` and $cEven_2 =$ `"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 $aEven_1$ and $aEven_2$.

Exercise 2.25. Consider your answer to Exercise 2.18, 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 $aBitList_1$ and $aBitList_2$ that correspond to the concrete bit-lists $cBitList_1 =$ `"0,1,0"` and $cBitList_2 =$ `"0,0,1"`.
2. Write a semantic function that transforms an abstract representation of a bit list into a concrete one. Test your function with the abstract bit lists $aBitList_1$ and $aBitList_2$.
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 $aBitList_1$ and $aBitList_2$.

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 standard way of defining the syntax of programming languages (see, for instance, the Java Language Grammar). The Java grammar contains a bit more "syntactical sugar" than the grammars that we considered thus far (and to be honest, this also holds for the ALGOL 60 grammar): it contains nonterminals with postfixes '?', '+' and '∗'.

EBNF

This extended BNF notation, *EBNF*, helps abbreviating a number of standard constructions that occur quite often in the syntax of a programming language:

- one or zero occurrences of nonterminal P, abbreviated $P?$,
- one or more occurrences of nonterminal P, abbreviated P^+,

- and zero or more occurrences of nonterminal P, abbreviated P^*.

We could easily express these constructions by adding additional nonterminals, but that decreases the readability of the grammar. The notation for the EBNF constructions is not entirely standardized. In some texts, you will for instance find the notation $[P]$ instead of $P?$, and $\{\,P\,\}$ for P^*. The same notation can be used for languages, grammars, and sequences of terminal and nonterminal symbols instead of just single nonterminals. In this section, we define 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 2.3, we have defined a number of operations on languages using operations on sets. We now show that these operations can be expressed in terms of operations on context-free grammars.

Theorem 2.10 (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 nonterminal sets N_L and N_M are disjoint. Then*

- *the language $L \cup M$ is generated by the grammar (T, N, R, S) where S is a fresh nonterminal, $N = N_L \cup N_M \cup \{\,S\,\}$ and $R = R_L \cup R_M \cup \{\,S \to S_L, S \to S_M\,\}$;*
- *the language $L\,M$ is generated by the grammar (T, N, R, S) where S is a fresh nonterminal, $N = N_L \cup N_M \cup \{\,S\,\}$ and $R = R_L \cup R_M \cup \{\,S \to S_L\ S_M\,\}$;*
- *the language L^* is generated by the grammar (T, N, R, S) where S is a fresh nonterminal, $N = N_L \cup \{\,S\,\}$ and $R = R_L \cup \{\,S \to \varepsilon, S \to S_L\ S\,\}$;*
- *the language 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 theorem above establishes that the set-theoretic operations at the level of languages (i. e., sets of sentences) have a direct counterpart at the level of grammatical descriptions. A straightforward question to ask is now: can we also define languages as the difference between two languages or as the intersection of two languages, and translate these operations to operations on grammars? Unfortunately, the answer is negative – there are no operations on grammars that correspond to the language intersection and difference operators.

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

Definition 2.11 (Grammar operations)**.** Let $G = (T, N, R, S)$ be a context-free grammar and let S' be a fresh nonterminal. Then

$$G^* = (T, N \cup \{\,S'\,\}, R \cup \{\,S' \to \varepsilon,\ S' \to S\ S'\,\}, S')$$

$$G^+ = (T, N \cup \{S'\}, R \cup \{S' \to S, S' \to S\ S'\}, S')$$
$$G? = (T, N \cup \{S'\}, R \cup \{S' \to \varepsilon,\ S' \to S\ \ \ \}, S')$$

The definition of $P?$, P^+, and P^* for a sequence of symbols 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 2.12 (EBNF for sequences)**.** Let P be a sequence of nonterminals and terminals, then

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

where Z is a new nonterminal in each definition.

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

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

Many variations are possible on this theme:

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

or also

$$L(P\ Q^*) = L(Z) \quad \text{with} \quad Z \to P \mid ZQ \tag{2.2}$$

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.

$$\begin{array}{ll} \mathit{Expr} & \to \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Expr}\ \texttt{else}\ \mathit{Expr} \\ \mathit{Expr} & \to \mathit{Expr}\ \texttt{where}\ \mathit{Decls} \\ \mathit{Expr} & \to \mathit{AppExpr} \\ \mathit{AppExpr} & \to \mathit{AppExpr}\ \mathit{Atomic} \mid \mathit{Atomic} \\ \mathit{Atomic} & \to \mathit{Var} \mid \mathit{Number} \mid \mathit{Bool} \mid \texttt{(}\ \mathit{Expr}\ \texttt{)} \\ \mathit{Decls} & \to \mathit{Decl} \\ \mathit{Decls} & \to \mathit{Decls}\ \texttt{;}\ \mathit{Decls} \end{array}$$

$$\mathit{Decl} \quad \to \mathit{Var}\ \texttt{=}\ \mathit{Expr}$$

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:

$$\begin{aligned}
\mathit{Expr} &\to \mathit{Expr}_1 \\
\mathit{Expr} &\to \mathit{Expr}_1\ \texttt{where}\ \mathit{Decls} \\
\mathit{Expr}_1 &\to \mathit{Expr}_2 \\
\mathit{Expr}_1 &\to \texttt{if}\ \mathit{Expr}_1\ \texttt{then}\ \mathit{Expr}_1\ \texttt{else}\ \mathit{Expr}_1 \\
\mathit{Expr}_2 &\to \mathit{Atomic} \\
\mathit{Expr}_2 &\to \mathit{Expr}_2\ \mathit{Atomic}
\end{aligned}$$

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

The nonterminal Expr_2 is left-recursive. Removing left recursion gives the following productions for Expr_2:

$$\begin{aligned}
\mathit{Expr}_2 &\to \mathit{Atomic} \mid \mathit{Atomic}\ \mathit{Expr}_2' \\
\mathit{Expr}_2' &\to \mathit{Atomic} \mid \mathit{Atomic}\ \mathit{Expr}_2'
\end{aligned}$$

Since the new nonterminal Expr_2' has exactly the same productions as Expr_2, these productions can be replaced by

$$\mathit{Expr}_2 \to \mathit{Atomic} \mid \mathit{Atomic}\ \mathit{Expr}_2$$

So Expr_2 generates a nonempty sequence of atomics. Using the $\cdot^+$-notation introduced before, we can replace Expr_2 by Atomic^+.

Another source of ambiguity are the productions for *Decls*. The nonterminal *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

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

or, according to (2.2),

$$Decls \to Decl\ (\texttt{;}\ Decl)^*$$

The last grammar transformation we apply is "left factoring". This transformation is applied to the productions for *Expr*, and yields

$$\begin{aligned} Expr &\to Expr_1\ Expr_1' \\ Expr_1' &\to \varepsilon \mid \texttt{where}\ Decls \end{aligned}$$

Since nonterminal $Expr_1'$ generates either nothing or a `where` clause, we can replace $Expr_1'$ by an optional `where` clause in the production for *Expr*:

$$Expr \to Expr_1\ (\texttt{where}\ Decls)?$$

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

$$\begin{aligned} Expr &\to Expr_1\ (\texttt{where}\ Decls)? \\ Expr_1 &\to Atomic^+ \\ Expr_1 &\to \texttt{if}\ Expr_1\ \texttt{then}\ Expr_1\ \texttt{else}\ Expr_1 \\ Atomic &\to Var \mid Number \mid Bool \mid \texttt{(}\ Expr\ \texttt{)} \\ Decls &\to Decl\ (\texttt{;}\ Decl)^* \end{aligned}$$

Exercise 2.26. Give the EBNF notation for each of the basic languages defined in Section 2.3.1.

Exercise 2.27. Let G be a grammar G. Give the language that is generated by $G?$ (i.e., the $\cdot?$ operation applied to G).

Exercise 2.28. Let

$$\begin{aligned} L_1 &= \{\, a^m b^m c^n \mid m, n \in \mathbb{N} \,\} \\ L_2 &= \{\, a^m b^n\ c^n \mid m, n \in \mathbb{N} \,\} \end{aligned}$$

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.

parsing problem

Definition 2.13 (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 relatively 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 recognized object (the semantic function). The techniques we describe comprise 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* (also called *lexer*), which splits an input sentence into a list of so-called tokens. For example, given the sentence

scanner
lexer

```
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

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 other interesting questions: *Can we explain why a sentence is not in a language*, and *What are the minimal changes that have to be made to the sentence 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 that, given an erroneous input, would just reply that the "Input could not be recognised". One of the most important aspects here is to define a 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 is missing is far less likely. This is where grammar engineering starts to play a rôle.

Summary

Starting from a simple example, the language of DNA-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 2.29. Do there exist languages L such that $\overline{(L^*)} = (\overline{L})^*$?

Exercise 2.30. Give a language L such that $L = L^*$.

Exercise 2.31. Under which circumstances is $L^+ = L^* - \{\varepsilon\}$?

Exercise 2.32. Let L be a language over alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ such that $L{=}L^R$. Does L contain only palindromes?

Exercise 2.33. Consider the grammar with productions

$S \to AA$
$A \to AAA$
$A \to \mathtt{a}$
$A \to \mathtt{b}A$
$A \to A\mathtt{b}$

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 $m, n, p \geqslant 0$, describe a derivation of the string $\mathtt{b}^m\mathtt{ab}^n\mathtt{ab}^p$.

Exercise 2.34. Consider the grammar with productions

$S \to \mathtt{aa}B$
$A \to \mathtt{b}B\mathtt{b}$
$A \to \varepsilon$
$B \to A\mathtt{a}$

Show that the string `aabbaabba` cannot be derived from S.

Exercise 2.35. Give a grammar for the language

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

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

Exercise 2.36. Describe the language generated by the grammar:

$S \to \varepsilon$
$S \to A$
$A \to \mathtt{a}A\mathtt{b}$
$A \to \mathtt{ab}$

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

Exercise 2.37. Describe the languages generated by the grammars.

$S \to \varepsilon$
$S \to A$
$A \to A\mathtt{a}$
$A \to \mathtt{a}$

and

$S \to \varepsilon$
$S \to A$
$A \to A\mathtt{a}A$
$A \to \mathtt{a}$

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

Exercise 2.38. Show that the languages generated by the grammars G_1, G_2 en G_3 are the same.

G_1 :	G_2 :	G_3 :
$S \to \varepsilon$	$S \to \varepsilon$	$S \to \varepsilon$
$S \to \mathtt{a}S$	$S \to S\mathtt{a}$	$S \to \mathtt{a}$
		$S \to SS$

Exercise 2.39. 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 $X \to \varepsilon$),
2. the start symbol does not occur in any alternative.

A grammar having this property is called *non-contracting*. The grammar $A \to \mathtt{a}A\mathtt{b} \mid \varepsilon$ does not have this property. Give a non-contracting grammar which describes the same language.

Exercise 2.40. Describe the language L of the grammar

$A \to A\mathtt{a}A \mid \mathtt{a}$

Give a grammar for L that has no left-recursive productions. Give a grammar for L that has no right-recursive productions.

Exercise 2.41. Describe the language L of the grammar

$X \to \mathtt{a} \mid X\mathtt{b}$

Give a grammar for L that has no left-recursive productions. Give a grammar for L that has no left-recursive productions and is non-contracting.

Exercise 2.42. Consider the language L of the grammar

$S \to T \mid US$
$T \to \mathtt{a}S\mathtt{a} \mid U\mathtt{a}$
$U \to S \mid SUT$

Give a grammar for L which uses only productions with two or less symbols on the right hand side. Give a grammar for L which uses only two nonterminals.

Exercise 2.43. 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 2.44. Give a grammar for the language consisting of all nonempty sequences of brackets,

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

in which the brackets match. An example sentence of the language is `( ) ( ( ) ) ( )`. Give a derivation for this sentence.

Exercise 2.45. Give a grammar for the language consisting of all nonempty sequences of two kinds of brackets,

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

in which the brackets match. An example sentence in this language is `[ ( ) ] ( )`.

Exercise 2.46. 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:

Sentence	$\rightarrow$ *Subject Predicate* `.`
Subject	$\rightarrow$ `they`
Predicate	$\rightarrow$ *Verb NounPhrase*
Predicate	$\rightarrow$ *AuxVerb Verb Noun*
Verb	$\rightarrow$ `are`
Verb	$\rightarrow$ `flying`
AuxVerb	$\rightarrow$ `are`
NounPhrase	$\rightarrow$ *Adjective Noun*
Adjective	$\rightarrow$ `flying`
Noun	$\rightarrow$ `planes`

Give two different leftmost derivations for the sentence

```
they are flying planes.
```

Exercise 2.47. Try to find some ambiguous sentences in your own natural language. Here are some ambiguous Dutch sentences seen in the newspapers:

```
Vliegen met hartafwijking niet gevaarlijk
Jessye Norman kan niet zingen
Alcohol is voor vrouwen schadelijker dan mannen
```

Exercise 2.48 (•). Is your grammar for Exercise 2.45 unambiguous? If not, find one which *is* unambiguous.

Exercise 2.49. This exercise deals with a grammar that uses unusual terminal and nonterminal symbols. Assume that $\odot$, $\otimes$, and $\oplus$ are nonterminals, and the other symbols are terminals.

$\odot \rightarrow \odot \triangle \otimes$
$\odot \rightarrow \otimes$

$\otimes \to \otimes \diamondsuit \oplus$
$\otimes \to \oplus$
$\oplus \to \clubsuit$
$\oplus \to \spadesuit$

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

Exercise 2.50 (•)**.** Prove, using induction, that the grammar G for palindromes in Section 2.2 does indeed generate the language of palindromes.

Exercise 2.51 (••, no answer provided)**.** Prove that the language generated by the grammar of Exercise 2.33 contains all strings over $\{\mathtt{a}, \mathtt{b}\}$ where the number of `a`'s is even and greater than zero.

Exercise 2.52 (no answer provided)**.** Consider the natural numbers in unary notation where only the symbol `I` is used; thus 4 is represented as `IIII`. Write an algorithm that, given a string w of `I`'s, determines whether or not w is divisible by 7.

Exercise 2.53 (no answer provided)**.** Consider the natural numbers in reverse binary notation; thus 4 is represented as `001`. Write an algorithm that, given a string w of zeros and ones, determines whether or not w is divisible by 7.

Exercise 2.54 (no answer provided)**.** Let w be a string consisting of `a`'s and `b`'s only. Write an algorithm that determines whether or not the number of `a`'s in w equals the number of `b`'s in w.

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 3.6, 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 [10, 12]. For example, there exist good parsers using parser combinators for the Haskell language.

Most of the techniques introduced in this chapter have been described by Burge [4], Wadler [13] and Hutton [7].

This chapter is a revised version of an article by Fokker [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 parse, i. e., how to recognise structure in a sequence of symbols, by means of parser combinators;
- to construct a parser when given a grammar;
- to define and use 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

As prerequisite knowledge for this chapter, you should be able to describe a language using a context-free grammar, to parse a sentence using a context-free grammar, and to formulate the parsing problem Furthermore, you should be familiar withable to use functional programming concepts such as **type**, **class**, and higher-order functions to develop programs.

3.1. The type of parsers

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 we have seen a grammar for sequences of bits:

$$S \to SS \mid \texttt{0} \mid \texttt{1}$$

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

data $S = Beside\ S\ S \mid Zero \mid One$

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

type $Parser = String \to S$ — preliminary

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 `100`, a parser will first build a parse tree `Beside` *One Zero* for `10`, and only then build a complete parse tree

Beside (*Beside One Zero*) *Zero*

using the unprocessed part `0` 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 paired. A better definition for the type *Parser* is hence:

type $Parser = String \to (S, String)$ — still preliminary

Any parser of type *Parser* returns an S 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 S, and to turn the parser type into a polymorphic type. The type *Parser* is parametrised with a type a, which represents the type of parse trees.

type $Parser\ a = String \to (a, String)$ — still preliminary

For example, a parser that returns a structure of type *Byte* now has type *Parser Byte*. A parser that parses sequences of `0`'s and `1`'s has type *Parser* S. We might also define

a parser that does not return a value of type S, but instead the List of zeroes and ones 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 `"100"` has the following two parse trees:

$$Beside\ (Beside\ One\ Zero)\ Zero$$
$$Beside\ One\ (Beside\ Zero\ Zero)$$

As another refinement of the type *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:

$$\textbf{type}\ Parser\ a = String \to [(a, String)] \quad \text{— useful, but still suboptimal}$$

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

list of successes

This method for parsing is called the *list of successes* method, described by Wadler [13]. 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.

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 the type of symbols s, and as before the result type a, the type of parsers becomes

$$\textbf{type}\ Parser\ s\ a = [s] \to [(a, [s])]$$

or, if you prefer meaningful identifiers over conciseness:

```
— The type of parsers
newtype Parser s r = Parser {runParser :: [s] -> [(r, [s])]}
```

Listing 3.1: ParserType.hs

type *Parser symbol result* = [*symbol*] → [(*result*, [*symbol*])]

Since we want to make the type of parsers abstract, because we might want to change it at a later time, and make the *Parser* type an instance of a number of classes later, we turn it into the following **newtype**

newtype *Parser s r* = *Parser* {*runParser* :: [*s*] → [(*r*, [*s*])]}

We will use this newtype definition in the rest of this chapter. This type is defined in Listing 3.1, the first part of our parser library. If we have a parser p, and input xs, we can apply the parser to the input by means of *runParser p xs*. The list of successes appears in the 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 full generality provided by the *Parser* type: the type of the input s (or *symbol*) will almost always be *Char*.

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:

  $A \to \varepsilon$
  $A \to \mathtt{a}$
  $A \to x$

  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 'parses' the empty string ε. 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 () – both the type and the value are pronounced *unit*.

unit type

$$\begin{aligned}&epsilon :: Parser\ s\ ()\\&epsilon = Parser\ (\lambda xs \to [((), xs)])\end{aligned}$$

A more useful variant of this parser 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).

$$\begin{aligned}&succeed\ \ :: a \to Parser\ s\ a\\&succeed\ r = Parser\ (\lambda xs \to [(r, xs)])\end{aligned}$$

the parser *epsilon* can be defined in terms of *succeed* by

$$epsilon = succeed\ ()$$

We will do this often in this section: define a parser in terms of one or more other parsers. This also partially explains the title 'parser combinators' of this chapter.

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 3.2. Note the difference with *epsilon*, which *does* have one element in its list of successes (albeit an empty one).

Suppose we want to parse 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*:

$$\begin{aligned}&symbola\ \ :: Parser\ Char\ Char\\&symbola\ \ = Parser\ (\lambda xs \to \textbf{case}\ xs\ \textbf{of}\\&\quad []\ \ \ \ \ \ \to []\\&\quad (x : xs) \to \textbf{if}\ x == \texttt{'a'}\ \textbf{then}\ [(\texttt{'a'}, xs)]\ \textbf{else}\ [])\end{aligned}$$

The list of successes method pays off again, 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`).

Function *symbola* parses a character `'a'`. We can generalise it in many ways. First, we sometimes want to parse values from a different type, such as a particular kind of tokens. The parser *anySymbol* parses any kind of symbols, and always succeeds with the first symbol in the input, or fails if the input is empty:

$$\begin{aligned}&anySymbol\ :: Parser\ s\ s\\&anySymbol\ = Parser\ (\lambda xs \to \textbf{case}\ xs\ \textbf{of}\\&\quad (x : xs)\ \ \to [(x, xs)]\\&\quad []\ \ \ \ \ \ \ \to [])\end{aligned}$$

Using the LambdaCase extension of Haskell, we can write this as

```
anySymbol :: Parser s s
anySymbol = Parser (λcase
  (x : xs) → [(x, xs)]
  []       → [])
```

We can use *anySymbol* to define the parser *symbola* by checking that the symbol parsed is equal to the character `'a'`. So an alternative definition of *symbola* is

```
symbola = satisfy (== 'a') anySymbol
```

where *satisfy* takes a condition, a function that returns a *Bool*, and returns a parser that checks the parsed symbol for that condition:

```
satisfy     :: (s → Bool) → Parser s s
satisfy c   = Parser (λcase
  (x : xs) → if c x then [(x, xs)] else []
  []       → [])
```

In the same fashion, we can write parsers that recognise other symbols than the character `'a'`. 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 Haskell, 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 3.2.

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 3.2. 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.

The second elementary parser that typically appears in a lexical analyser is the *digit* parser, which uses the parser combinator *satisfy* to parse digits (characters in between `'0'` and `'9'`):

```
— Elementary parsers
succeed    :: a → Parser s a
succeed r  = Parser (λxs → [(r, xs)])

failp      :: Parser s a
failp xs = Parser (const [])

symbol     :: Eq s ⇒ s → Parser s s
symbol a   = satisfy (== a)

satisfy    :: (s → Bool) → Parser s s
satisfy c  = Parser (λcase
  (x : xs) → if c x then [(x, xs)] else []
  []       → [])
token      :: Eq s ⇒ [s] → Parser s [s]
token k    = Parser (λxs → if k == take n xs then [(k, drop n xs)] else [])
  where n = length k
— Applications of elementary parsers
digit :: Parser Char Char
digit = satisfy isDigit
```

Listing 3.2: Elementary parsers

$$
\begin{aligned}
&\mathit{digit} :: \mathit{Parser}\ \mathit{Char}\ \mathit{Char}\\
&\mathit{digit} = \mathit{satisfy}\ \mathit{isDigit}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{isDigit} \quad :: \mathit{Char} \to \mathit{Bool}\\
&\mathit{isDigit}\ x = \texttt{'0'} \leqslant x \wedge x \leqslant \texttt{'9'}
\end{aligned}
$$

Exercise 3.1. Define a function $\mathit{capital} :: \mathit{Parser}\ \mathit{Char}\ \mathit{Char}$ that parses capital letters.

Exercise 3.2. Since *satisfy* is a generalisation of *symbol*, the function *symbol* can be defined as an instance of *satisfy*. How can this be done?

Exercise 3.3. 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

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

  where x and y are sequences of nonterminal or terminal symbols;
- 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:

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

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 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 Haskell's built-in construct |, which is used to distinguish cases in a function definition or as a separator for the constructors of a datatype.

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

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

So <*> has a higher priority – i. e., it binds stronger – 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 3.3, 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) = Parser (λxs →
                    [ (combine r1 r2, zs)
                    | (r1, ys) ← runParser p xs
                    , (r2, zs) ← runParser q ys
                    ])
```

```
— Parser combinators
(<|>)      :: Parser s a → Parser s a → Parser s a
(p <|> q)  = Parser (λxs → runParser p xs ++ runParser q xs)
(<*>)      :: Parser s (b → a) → Parser s b → Parser s a
(p <*> q)  = Parser (λxs →
                [ (f r, zs)
                | (f, ys) ← runParser p xs
                , (r, zs) ← runParser q ys
                ])
(<$>)      :: (a → b) → Parser s a → Parser s b
(f <$> p)  = Parser (λxs →
                [ (f y, ys   )
                | (   y, ys   ) ← runParser p xs
                ])
— Applications of parser combinators
newdigit :: Parser Char Int
newdigit = f <$> digit
  where f c = ord c − ord '0'
```

Listing 3.3: ParserCombinators.hs

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 parser with an operator that describes how to combine the parts (as is done in the *zipWith* function). However, we choose 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 $b \to a$, 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 3.3. 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.

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 p_1 and p_2 return lists of possible parsings. To obtain all possible parsings when applying p_1 or p_2, we only need to concatenate these two lists.

By combining parsers with parser combinators we can construct new parsers. The most important parser combinators are <*> and <|>. The parser combinator <,> in exercise 3.12 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 3.3. 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.

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

```
sequenceOfBits  ::  Parser Char S
sequenceOfBits  =   Beside <$> sequenceOfBits <*> sequenceOfBits
                <|> const Zero <$> symbol '0'
                <|> const One <$> symbol '1'
```

But if you try to run this function, you will get a stack overflow! If you apply *sequenceOfBits* to a string, the first thing it does is to apply itself to the same string, which loops. The problem stems from the fact that the underlying grammar is left-recursive. For any left-recursive grammar, a systematically constructed parser using parser combinators will exhibit the problem that it loops. However, 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:

```
sequenceOfBits' :: Parser Char SA
sequenceOfBits' =
        const ConsZero <$>  symbol '0' <*> parseZ
  <|> const ConsOne <$>  symbol '1' <*> parseZ
  <|> const ZeroS    <$>  symbol '0'
  <|> const OneS     <$>  symbol '1'
  where parseZ  =  ConsZ  <$> sequenceOfBits' <*> parseZ
                <|> SingleZ <$> sequenceOfBits'
```

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 `0`'s and `1`'s.

Exercise 3.4. Prove for all $f :: a \to b$ that

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

In the sequel we will often use this rule for constant functions f, i. e., $f = \lambda\_ \to c$ for some term c.

Exercise 3.5. Consider the parser (:) <$> *symbol* `'a'`. Give its type and show its results on inputs [] and $x : xs$.

Exercise 3.6. Consider the parser (:) <$> *symbol* `'a'` <*> p. Give its type and show its results on inputs [] and $x : xs$.

Exercise 3.7. Define a parser for Booleans.

Exercise 3.8 (no answer provided)**.** Define parsers for each of the basic languages defined in Section 2.3.1.

Exercise 3.9. Consider the grammar for palindromes that you have constructed in Exercise 2.7.

1. Give the datatype Pal_2 that corresponds to this grammar.

2. Define a parser $palin_2$ that returns parse trees for palindromes. Test your function with the palindromes $cPal_1$ = `"abaaba"` and $cPal_2$ = `"baaab"`. Compare the results with your answer to Exercise 2.21.

3. Define a parser *palina* that counts the number of `a`'s occurring in a palindrome.

Exercise 3.10. Consider the grammar for a part of the English language that is given in Exercise 2.46.

1. Give the datatype *English* that corresponds to this grammar.

2. Define a parser *english* that returns parse trees for the English language. Test your function with the sentence `they are flying planes`. Compare the result to your answer of Exercise 2.46.

Exercise 3.11. 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 3.12. 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 3.13. The term 'parser combinator' is in fact not an adequate description for <$>. Can you think of a better word?

Exercise 3.14. 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 3.15. Define <*> in terms of <,> and <$>. Define <,> in terms of <*> and <$>.

Exercise 3.16. If you examine the definitions of <*> and <$> in Listing 3.3, you can observe that <$> 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 2.44:

$$S \to \texttt{(}\ S\ \texttt{)}\ S \mid \varepsilon$$

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 ???    — ill-typed
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. We obtain the following Haskell datatype:

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

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

```
Match Empty (Match Empty Empty)
```

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 *Parentheses*.

```
nrofpars :: Parentheses → Int
nrofpars (Match pl pr) = 2 + nrofpars pl + nrofpars pr
nrofpars Empty         = 0
```

Using the datatype *Parentheses*, we can add 'semantic functions' to the parser. We then obtain the definition of *parens* in Listing 3.4.

By varying the function used as a first argument of <$> (the 'semantic function'), we can return other things than parse trees. As an example we construct a parser that

```
data Parentheses = Match Parentheses Parentheses
                 | Empty
  deriving Show
open = symbol '('
close = symbol ')'
parens :: Parser Char Parentheses
parens = (λ_ x _ y → Match x y)
           <$> open <*> parens <*> close <*> parens
       <|> succeed Empty
nesting :: Parser Char Int
nesting = (λ_ x _ y → max (1 + x) y)
            <$> open <*> nesting <*> close <*> nesting
        <|> succeed 0
```

Listing 3.4: ParseParentheses.hs

calculates the nesting depth of nested parentheses, see the function *nesting* defined in Listing 3.4.

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

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

It is maybe not easy to think of a good use of the matching parentheses language. But formal languages often use open and close constructs: parentheses around function arguments, an HTML document that starts with <*html*> and ends with </*html*>, code in a `while`-block surrounded by an open brace { and a close brace}, etc.

Exercise 3.17. What is the type of the function $f = \lambda\_\ x\ \_\ y \rightarrow Match\ x\ y$ which appears in function *parens* in Listing 3.4? 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 3.18. What is a convenient way for <*> to associate? Does it?

Exercise 3.19. Write a function *test* that determines whether or not a given string belongs to the language parsed by a given parser.

```
(<$)      :: a → Parser s b → Parser s a
(r <$ p) = const r <$> p
(<*)      :: Parser s a → Parser s b → Parser s a
(p <* q) = (λx _ → x) <$> p <*> q
(*>)      :: Parser s a → Parser s b → Parser s b
(p *> q) = (\_y → y) <$> p <*> q
```

Listing 3.5: ParserCombinatorVariants.hs

```
parens  :: Parser Char Parentheses
parens  =  Match <$ open <*> parens <* close <*> parens
       <|> succeed Empty
nesting :: Parser Char Int
nesting =  (λx y → max (1 + x) y) <$ open <*> nesting <* close <*> nesting
       <|> succeed 0
```

Listing 3.6: ParseParentheses2.hs

3.3.2. Combinator variants

Often we parse a symbol or sequence of symbols that we don't need in calculating the abstract syntax. For example, the parser *parens* for matching parentheses given in Listing 3.4 checks that the matching parentheses appear in the input string, but does not use them to create an abstract syntax tree. All arguments represented by an underscore in the lambda function calculating the abstract syntax tree are superfluous.

In Listing 3.5 we define variants of parser combinators that ignore one of their arguments. Using these parser combinators in the code for *parens* gives a much cleaner definition, see Listing 3.6

Another variant of the combinators we have defined so far is left-biased choice. As its name says, left-biased choice takes two parsers, and returns the results from the first (left) argument if it doesn't fail, and only returns results from the second (right) argument if the left argument fails. This parser combinator is not easily defined in terms of existing parser combinators, and is a primitive.

```
infixr 3 ≪|>
(≪|>)  :: Parser s a → Parser s a → Parser s a
```

```
— 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 = (:) <$> p <*> many p <|> succeed []
many1 :: Parser s a → Parser s [a]
many1 p = (:) <$> p <*> many p
pack :: Parser s a → Parser s b → Parser s c → Parser s b
pack p r q = (λ_ x _ → x) <$> p <*> r <*> q
listOf :: Parser s a → Parser s b → Parser s [a]
listOf p s = (:) <$> p <*> many ((λ_ x → x) <$> 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
```

Listing 3.7: EBNF.hs

```
p <<|> q = Parser (λxs → let r = runParser p xs in
                         if null r then runParser q xs else r)
```

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 p for a construction, *many p* constructs a parser for zero or more occurrences of that construction:

```
many :: Parser s a → Parser s [a]
many p  =  (:) <$> p <*> many p
        <|> succeed []
```

So the EBNF expression P^* is implemented by *many P*. The function (:) is just the cons-operator for lists: it takes a head element and a tail list and combines them.

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 $many_1$ 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 3.7. 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 3.7. 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 3.27.

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.

$$
\begin{aligned}
&\mathit{first} :: \mathit{Parser}\ a\ b \to \mathit{Parser}\ a\ b \\
&\mathit{first}\ p\ xs \mid \mathit{null}\ r \quad\ \ = [] \\
&\qquad\qquad \mid \mathit{otherwise} = [\mathit{head}\ r] \\
&\quad \textbf{where}\ r = p\ xs
\end{aligned}
$$

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

$$
\begin{aligned}
\mathit{greedy} &= \mathit{first}\ .\ \mathit{many} \\
\mathit{greedy}_1 &= \mathit{first}\ .\ \mathit{many}_1
\end{aligned}
$$

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

$$\mathit{obligatory}\ p\ d = \mathit{first}\ (\mathit{option}\ p\ d)$$

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

3.4.2. Separators

The combinators *many*, many_1 and *option* are classical in compiler constructions – there are notations for it in EBNF ($\cdot^*$, $\cdot^+$ and $\cdot?$, 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 3.7. Special cases of this combinator are:

$$
\begin{aligned}
&\mathit{parenthesised}\ p = \mathit{pack}\ (\mathit{symbol}\ \texttt{'('})\ \ \ p\ (\mathit{symbol}\ \texttt{')'}) \\
&\mathit{bracketed}\ p\ \ \ \ = \mathit{pack}\ (\mathit{symbol}\ \texttt{'['})\ \ \ p\ (\mathit{symbol}\ \texttt{']'}) \\
&\mathit{compound}\ p\ \ \ \ = \mathit{pack}\ (\mathit{token}\ \texttt{"begin"})\ p\ (\mathit{token}\ \texttt{"end"})
\end{aligned}
$$

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:

$$
\begin{aligned}
&\mathit{listOf} :: \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ b \to \mathit{Parser}\ s\ [a] \\
&\mathit{listOf}\ p\ s = (:) <\$> p <\!\!*\!\!> \mathit{many}\ ((\lambda\_\ x \to x) <\$> s <\!\!*\!\!> p)
\end{aligned}
$$

Useful instantiations are:

```
— 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 3.8: Chains.hs

```
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 3.8 (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 3.8.

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 3.8.

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 the article by Fokker [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 3.20.

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 3.21. What is the value of

many (*symbol* `'a'`) *xs*

for $xs \in \{[], [\texttt{'a'}], [\texttt{'b'}], [\texttt{'a'}, \texttt{'b'}], [\texttt{'a'}, \texttt{'a'}, \texttt{'b'}]\}$?

Exercise 3.22. 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?

Exercise 3.23 (no answer provided)**.** Using the parser combinators *option*, *many* and $many_1$ define parsers for each of the basic languages defined in Section 2.3.1.

Exercise 3.24. 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 3.25. As an application of *psequence*, define the function *token* that was discussed in Section 3.2.

Exercise 3.26 (no answer provided)**.** Carefully analyse the semantic functions in the definition of *chainl* in Listing 3.8.

Exercise 3.27. In real programming languages, identifiers follow rather 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. Combinator parsers are monads and more

This section gives a number of standard classes of which the type *Parser s* is an instance. For an introduction of these classes we refer to the Haskell literature.

To create an instance of the class *Functor* for *Parser s* we need to define a function *fmap* of type $(a \to b) \to Parser\ s\ a \to Parser\ s\ b$. We have already defined such a function, namely <$>.

```
instance Functor (Parser s) where
  fmap f p = f <$> p
```

The operator <$> satisfies the laws required for a *Functor*, namely id <$> $p = p$, and $(f\ .\ g)$ <$> $p = f$ <$> $(g$ <$> $p)$.

The type *Parser* is also an instance of the class *Applicative*. The class *Applicative* specifies two functions: a function *pure* of type $a \to Parser\ s\ a$, for which we can use *succeed*, and an operator <*> of type $Parser\ s\ (a \to b) \to Parser\ s\ a \to Parser\ s\ b$. Luckily, our parser combinator <*> has exactly the same type, and we can use it for the *Applicative* instance:

```
instance Applicative (Parser s) where
  pure     = succeed
  p <*> q = p <*> q
  — the <*> in the right-hand side is our parser combinator definition
```

The instance should satisfy four laws. We will look at the first of these: $pure\ id$ <*> $v = v$. In parser combinator terms, the left-hand side is $succeed\ id$ <*> v. *succeed* parses nothing, and returns the identity function, which is applied to the result of v. It follows that this is indeed the same as parsing with only v.

Parser is a *Monad*. For the instance we need a function *return* of type $a \to Parser\ s\ a$, for which we can use *succeed* again, just as for *Applicative*, and we need a bind operator. The instance of the bind operator >>= for *Parser s* has the type $Parser\ s\ a \to (a \to Parser\ s\ b) \to Parser\ s\ b$. The idea is to run the first parser, and apply the function in the second argument to it to obtain a parser that returns a b value.

```
instance Monad (Parser s) where
  return = pure
```

$$
\begin{aligned}
p \gg\!= f = \mathit{Parser}\ (\lambda xs \rightarrow\ & [(z\ , zs) \\
& |(y, ys) \leftarrow \mathit{runParser}\ p\ xs \\
& , (z, zs) \leftarrow \mathit{runParser}\ (f\ y)\ ys \\
&])
\end{aligned}
$$

A monad instance should satisfy the three *Monad* laws. We discuss the first of these: $\mathit{return}\ a \gg\!= h = h\ a$. Since $\mathit{runParser}\ (\mathit{succeed}\ a)\ xs$ doesn't consume input and returns a single pair (a, xs), the instance for $\mathit{Parser}\ s$ of the law $\mathit{return}\ a \gg\!= h$ equals $\mathit{Parser}\ (\lambda xs \rightarrow [(z, zs) \mid (z, zs) \leftarrow \mathit{runParser}\ (h\ a)\ xs])$, which simplifies to $\mathit{Parser}\ (\lambda xs \rightarrow \mathit{runParser}\ (h\ a)\ xs)$, and then to $h\ a$.

The final instance we give for $\mathit{Parser}\ s$ is for the class *Alternative*. An instance of the class *Alternative* needs an *empty* element of type $\mathit{Parser}\ s\ a$, and a choice operator $<|>$ of type $\mathit{Parser}\ s\ a \rightarrow \mathit{Parser}\ s\ a \rightarrow \mathit{Parser}\ s\ a$. Not surprisingly, we can use the choice parser combinator with the same name for the latter, and the failing parser for the former.

```
instance Alternative (Parser s) where
  empty   = failp
  p <|> q = p <|> q
  — the <|> in the right-hand side is our parser combinator definition
```

According to the laws for *Alternative*, *failp* should be the neutral element for choice, so $\mathit{failp} <|> p = p <|> \mathit{failp} = p$, and the choice operator should be associative. Both laws are easily verified for our $\mathit{Parser}\ s$ instance.

Now that we have an instance of $\mathit{Parser}\ s$ for monads, it is easy to use the result of a parser to affect the follow-up parsing process. For example, suppose we want to parse a natural number n, and then parse n lines. So for the input

```
3
In
the
beginning
was
```

the parser would return `["In", "the", "beginning"]`. The parser *parseLine* parses a single line of characters:

```
parseLine :: Parser Char String
parseLine = many (satisfy (≢ '\n')) <* symbol '\n'
```

We use it in *parseNLines*, which uses the **do** notation for monads, and the *sequence* function to turn a list of parsers into a single parser.

```
parseNLines :: Parser Char [String]
parseNLines = do
```

```
n ← natural
_ ← symbol '\ n '
sequence $ replicate n parseLine
```

Another example of the use of monads is in the definition of the *guard* parser combinator. *guard* guards the results of a parser by a condition: only results that satisfy the condition are returned.

```
guard                  :: (a → Bool) → Parser s a → Parser s a
guard cond parser = parser >>= λa → if cond a then succeed a else empty
```

3.6. 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    → E + E
     | E - E
     | E / E
     | ( E )
     | Digs
```

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

```
E    → Identifier
     | Identifier ( Args )
Args → ε | E (, E)*
```

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 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

```
— 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 3.9: ExpressionParser.hs

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 3.9.

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.

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 3.28. 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 3.9 in such way that it will be.

Exercise 3.29. 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 3.30. Modify the functions in Listing 3.9, in such a way that `+` is parsed as a right-associative operator, and `-` is parsed as a left-associative operator.

3.7. 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. We cn improve on this. 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:

$$
\begin{aligned}
&\textbf{type } Op\ a = (Char, a \to a \to a)\\
&gen :: [Op\ a] \to Parser\ Char\ a \to Parser\ Char\ a\\
&gen\ ops\ p = chainl\ p\ (choice\ (map\ f\ ops))\\
&\quad \textbf{where } f\ (s, c) = const\ c <\$> symbol\ s
\end{aligned}
$$

Here the parser combinator *choice* is the generalisation of <|> to lists of parsers, and is defined in exercise 3.24. If furthermore we define as shorthand:

$$
\begin{aligned}
multis &= [(\texttt{'*'}, (:*:)), (\texttt{'/'}, (:/:))]\\
addis &= [(\texttt{'+'}, (:+:)), (\texttt{'-'}, (:-:))]
\end{aligned}
$$

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

$$
\begin{aligned}
expr &= gen\ addis\ term\\
term &= gen\ multis\ fact
\end{aligned}
$$

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.

```
— 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 3.10: GExpressionParser.hs

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 3.10.

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.

3.8. Exercises

Exercise 3.31. How should the parser of Section 3.7 be adapted to also allow raising an expression to the power of an expression?

Exercise 3.32. Prove the following laws

$$h <\$> (f <\$> p) = (h \,.\, f) <\$> p \tag{3.1}$$

$$h <\$> (p <|> q) = (h <\$> p) <|> (h <\$> q) \tag{3.2}$$

$$h <\$> (p <*> q) = ((h.) <\$> p) <*> q \tag{3.3}$$

Exercise 3.33. Consider your answer to Exercise 2.23. 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 $cMir_1$ and $cMir_2$.

Exercise 3.34. Consider your answer to Exercise 2.25. 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 $cBitList_1$ and $cBitList_2$.

Exercise 3.35. 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 3.36. 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 3.37. Define a parser for Java assignments that consist of a variable, an `=` sign, an expression and a semicolon.

Exercise 3.38 (no answer provided)**.** Define a parser for (simplified) Java statements.

Exercise 3.39 (no answer provided)**.** Outline the construction of a parser for Java programs.

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.

Goals

After studying this chapter, solving the exercises, and applying the concepts in the labs, you will have further developed your skills in

- designing a context-free grammar for a simple language;
- developing a parser that parses sentences of the language you have designed;
- transforming a grammar such that it satisfies a number of properties;
- defining a semantic function to obtain the required information;
- decomposing the problem of language and parser design into several steps.

4.1. Decomposing grammar and parser design

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

1. give example sentences of the language for which you want to design a grammar and a parser;
2. give a grammar for the language for which you want to have a parser;
3. test that the grammar can indeed describe the example sentences;
4. analyse this grammar to find out whether or not it has some desirable properties;
5. possibly transform the grammar to obtain some of these desirable properties;
6. 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.
7. construct a basic parser;
8. add semantic functions;

9. test that the parser can parse the example sentences you have given in the first step, and that the parser returns what you expect.

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.

4.2. Step 1: Example sentences for the language

We have already given an example sentence above:

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

Other example sentences are:

```
Utrecht Centraal 10:25 10:58 Amsterdam Centraal
Assen
```

4.3. Step 2: 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.

$$
\begin{array}{ll}
\mathit{TS} & \to \mathit{TS}\ \mathit{Departure}\ \mathit{Arrival}\ \mathit{TS} \mid \mathit{Station} \\
\mathit{Station} & \to \mathit{Identifier}^{+} \\
\mathit{Departure} & \to \mathit{Time} \\
\mathit{Arrival} & \to \mathit{Time} \\
\mathit{Time} & \to \mathit{Nat} : \mathit{Nat}
\end{array}
$$

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:

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

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

4.4. Step 3: Testing the grammar

Both grammars we have given in step 2 describe the example sentences given in step 1. The derivation of these sentences using these grammars is easy.

4.5. Step 4: Analysing the grammar

To parse sentences of a language efficiently, we want to have a unambiguous 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.

4.6. Step 5: Transforming the grammar

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

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

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:

$$\begin{aligned} TS &\to Station\ Z \\ Z &\to \varepsilon \mid Departure\ Arrival\ TS \end{aligned}$$

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

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

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.

4.7. Step 6: 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:

$$\begin{aligned} &\textbf{data}\ Token\ \ = Station\_Token\ Station \mid Time\_Token\ Time \\ &\textbf{type}\ Station = String \\ &\textbf{type}\ Time\ \ \ = (Int, Int) \end{aligned}$$

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

$$\begin{aligned} &ts :: Parser\ Char\ ? \\ &ts :: Parser\ Token\ ? \end{aligned}$$

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.

$$\begin{aligned}&\textbf{data}\ TS_1 = Single_1\ Station\\ &\qquad\quad \mid\ Cons_1\ Station\ Time\ Time\ TS_1\end{aligned}$$

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

$$\textbf{type}\ TS_2 = ([(Station, Time, Time)], Station)$$

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 4.3:

$$\begin{aligned}&\textbf{data}\ TS_3 = Single_3\ Station\\ &\qquad\quad \mid\ Cons_3\ (Station, Time, [(Time, Station, Time)], Time, Station)\end{aligned}$$

Which abstract syntax should we take? Again, this depends on what we want to do with the abstract syntax. Since TS_2 and TS_1 combine departure and arrival times in a tuple, they are convenient to use when computing travelling times. TS_3 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 grammar (4.1) for travelling schemes, we use TS_1 for the abstract syntax. Note that TS_1 is a datatype, whereas TS_2 is a type. TS_1 cannot be defined as a type because of the two alternative productions for TS. TS_2 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 $TS_1,\ TS_2,\ TS_3$.

4.8. Step 7: 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.

grammar construct	Haskell/parser construct
$\rightarrow$	$=$
|	<|>
(space)	<*>
$.^+$	$many_1$
$.^*$	*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*.

4.8.1. 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 <$> many1 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.

4.8.2. 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 . combine . words

combine :: [String] → [String]
combine []          = []
combine [x]         = [x]
combine (x : y : xs) = if isAlpha (head x) ∧ isAlpha (head y)
                       then combine ((x ++ " " ++ y) : xs)
                       else x : combine (y : xs)

mkToken :: String → Token
mkToken xs = if   isDigit (head xs)
             then Time_Token (mkTime xs)
             else Station_Token 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
```

This is a basic scanner with very basic error messages, but it suffices for now. The composition of the scanner with the function $tstoken_1$ 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 4.3.

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

4.9. Step 8: 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 $many_1$ *identifier* returns a list of strings, and we want to obtain the concatenation of these strings for the station name, we can take the concatenation function *concat* for *undefined* in function *station*. 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

$$\lambda x \_ y \rightarrow (x, y)$$

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 4.8.1 returns an abstract syntax tree of type TS_2. 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 TS_2) and a *Station*. Since *many* returns a list of things, we can construct such a tuple by means of the function

$$\lambda x \_ y \rightarrow (x, y)$$

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

$$\lambda\_ x \_ y \_ z \rightarrow (x, y, z)$$

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

$$\lambda\_ \_ \_ (xh, xm) \_ (zh, zm) \rightarrow (zh - xh) * 60 + zm - xm$$

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

$$\lambda x \_ \_ \rightarrow sum\ x$$

The final set of semantic functions we define are used for computing the total waiting time. Since the second grammar of Section 4.3 combines arrival times and departure times, we use a parser based on this grammar: the basic parser $tstoken_2$. 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:

$$\lambda(uh, um)\ \_\ (wh, wm) \rightarrow (wh - uh) * 60 + wm - um$$

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*:

$$\lambda\_\ \_\ x\ \_\ \_ \rightarrow sum\ x$$

4.10. Step 9: 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.

Summary

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

4.11. Exercises

Exercise 4.1. Write a parser *floatLiteral* for Java float-literals. The EBNF grammar for float-literals is given by:

FloatLiteral	→ *IntPart* . *FractPart*? *ExponentPart*? *FloatSuffix*?
	| . *FractPart ExponentPart*? *FloatSuffix*?
	| *IntPart ExponentPart FloatSuffix*?
	| *IntPart ExponentPart*? *FloatSuffix*
IntPart	→ *SignedInteger*
FractPart	→ *Digits*
ExponentPart	→ *ExponentIndicator SignedInteger*
SignedInteger	→ *Sign*? *Digits*
Digits	→ *Digits Digit* | *Digit*
ExponentIndicator	→ `e` | `E`
Sign	→ `+` | `-`
FloatSuffix	→ `f` | `F` | `d` | `D`

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

Exercise 4.2. Write an evaluator *signedFloat* for Java float-literals (the float-suffix may be ignored).

Exercise 4.3. Up to the definition of the semantic functions, parsers constructed on a (fixed) abstract syntax have the same shape. Give this parsing scheme for Java float literals.

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] B.W. Kernighan and R. Pike. Regular expressions — languages, algorithms, and software. *Dr. Dobb's Journal*, April:19 – 22, 1999.

[9] D.E. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127–145, 1968.

[10] Niklas Röjemo. *Garbage collection and memory efficiency in lazy functional languages*. PhD thesis, Chalmers University of Technology, 1995.

[11] 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.

[12] S.D. Swierstra and P.R. Azero Alcocer. Fast, error correcting parser combinators: a short tutorial. In *SOFSEM'99*, 1999.

[13] 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.

A. Answers to exercises

2.1 Three of the four strings are elements of L^*: `abaabaaabaa`, `aaaabaaaa`, `baaaaabaa`.

2.2 $\{\varepsilon\}$.

2.3

$$
\begin{array}{ll}
 & \emptyset L \\
= & \quad \{ \text{ Definition of concatenation of languages } \} \\
 & \{ st \mid s \in \emptyset, t \in L \} \\
= & \quad \{ \ s \in \emptyset \ \} \\
 & \emptyset
\end{array}
$$

The other equalities can be proved in a similar fashion.

2.4 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.

2.5 Section 2.1 contains an inductive definition of the set of sequences over an arbitrary set X. Syntactical definitions for such sets follow immediately from this.

1. A grammar for $X = \{\mathtt{a}\}$ is given by

   $$
   \begin{array}{l}
   S \to \varepsilon \\
   S \to \mathtt{a}S
   \end{array}
   $$

2. A grammar for $X = \{\mathtt{a}, \mathtt{b}\}$ is given by

   $$
   \begin{array}{l}
   S \ \to \varepsilon \\
   S \ \to X \ S \\
   X \to \mathtt{a} \mid \mathtt{b}
   \end{array}
   $$

2.6 A context free grammar for L is given by

$$
\begin{array}{l}
S \to \varepsilon \\
S \to \mathtt{a}S\mathtt{b}
\end{array}
$$

2.7 Analogous to the construction of the grammar for PAL.

$$
\begin{aligned}
P \rightarrow\ & \varepsilon \\
\mid\ & \texttt{a} \\
\mid\ & \texttt{b} \\
\mid\ & \texttt{a}P\texttt{a} \\
\mid\ & \texttt{b}P\texttt{b}
\end{aligned}
$$

2.8 Analogous to the construction of PAL.

$$
\begin{aligned}
M \rightarrow\ & \varepsilon \\
\mid\ & \texttt{a}M\texttt{a} \\
\mid\ & \texttt{b}M\texttt{b}
\end{aligned}
$$

2.9 First establish an inductive definition for parity sequences. An example of a grammar that can be derived from the inductive definition is:

$$P \rightarrow \varepsilon \mid \texttt{1}P\texttt{1} \mid \texttt{0}P \mid P\texttt{0}$$

There are many other solutions.

2.10 Again, establish an inductive definition for L. An example of a grammar that can be derived from the inductive definition is:

$$S \rightarrow \varepsilon \mid \texttt{a}S\texttt{b} \mid \texttt{b}S\texttt{a} \mid SS$$

Again, there are many other solutions.

2.11 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.

2.12 The language consisting of the empty string only, i.e., $\{\varepsilon\}$.

2.13 This grammar generates the empty language, i.e., $\emptyset$. In general, grammars such as this one that have no production rules without nonterminals on the right hand side, cannot produce any sentences with only terminal symbols. Each derivation will always contain nonterminals, so no sentences can be derived.

2.14 The sentences in this language consist of zero or more concatenations of `ab`, i.e., the language is the set $\{\texttt{ab}\}^*$.

2.15 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 procedure yields

$S \to \texttt{ab}$
$S \to \texttt{aa}$
$S \to \texttt{baa}$

2.16 To bring the grammar into the form where we can directly apply the rule for associative separators, we introduce a new nonterminal:

$A \to A\texttt{a}A$
$A \to B$
$B \to \texttt{b} \mid \texttt{c}$

Now we can remove the ambiguity:

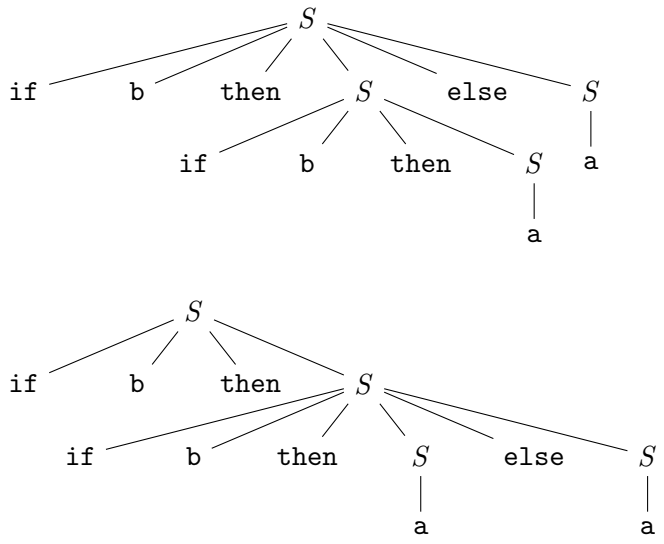
$A \to B\texttt{a}A$
$A \to B$
$B \to \texttt{b} \mid \texttt{c}$

It is now (optionally) possible to undo the auxiliary step of introducing the additional nonterminal by applying the rules for substituting right hand sides for nonterminal and removing unreachable productions. We then obtain:

$A \to \texttt{ba}A \mid \texttt{ca}A$
$A \to \texttt{b} \mid \texttt{c}$

2.17

1. Here are two parse trees for the sentence `if b then if b then a else a`:



2. The rule we apply is: match `else` with the closest previous unmatched `else`. This means we prefer the second of the two parse trees above. The disambiguating rule is incorporated directly into the grammar:

$$\begin{array}{ll} S & \to \mathit{MatchedS} \mid \mathit{UnmatchedS} \\ \mathit{MatchedS} & \to \texttt{if b then } \mathit{MatchedS} \texttt{ else } \mathit{MatchedS} \\ & \mid \texttt{a} \\ \mathit{UnmatchedS} & \to \texttt{if b then } S \\ & \mid \texttt{if b then } \mathit{MatchedS} \texttt{ else } \mathit{UnmatchedS} \end{array}$$

3. An `else` clause is always matched with the closest previous unmatched `if`.

2.18 An equivalent grammar for bit lists is

$$\begin{array}{l} L \to B\ Z \mid B \\ Z \to \texttt{,}\ L\ Z \mid \texttt{,}\ L \\ B \to \texttt{0} \mid \texttt{1} \end{array}$$

2.19

1. The grammar generates the language $\{\, a^{2n} b^m \mid m, n \in \mathbb{N} \}$.
2. An equivalent non left recursive grammar is

$$\begin{array}{l} S \to AB \\ A \to \varepsilon \mid \texttt{aa}A \\ B \to \varepsilon \mid \texttt{b}B \end{array}$$

2.20 Of course, we can choose how we want to represent the different operators in concrete syntax. Choosing standard symbols, this is one possibility:

$$\begin{array}{ll} \mathit{Expr} & \to \mathit{Expr} \texttt{ + } \mathit{Expr} \\ & \mid \mathit{Expr} \texttt{ * } \mathit{Expr} \\ & \mid \mathit{Int} \end{array}$$

where *Int* is a nonterminal that produces integers. Note that the grammar given above is ambiguous. We could also give an unambiguous version, for example by introducing operator priorities.

2.21 Recall the grammar for palindromes from Exercise 2.7, now with names for the productions:

$$\begin{array}{ll} \text{Empty:} & P \to \varepsilon \\ \text{A:} & P \to \texttt{a} \\ \text{B:} & P \to \texttt{b} \\ \text{A}_2\text{:} & P \to \texttt{a}P\texttt{a} \\ \text{B}_2\text{:} & P \to \texttt{b}P\texttt{b} \end{array}$$

We construct the datatype P by interpreting the nonterminal as datatype and the names of the productions as names of the constructors:

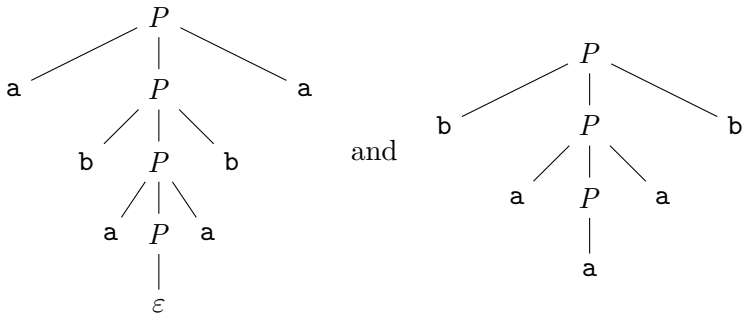
$$\textbf{data}\ P = \mathit{Empty} \mid A \mid B \mid A_2\ P \mid B_2\ P$$

Note that once again we keep the nonterminals on the right hand sides as arguments to the constructors, but omit all the terminal symbols.

The strings `abaaba` and `baaab` can be derived as follows:

$$P \Rightarrow \texttt{a}P\texttt{a} \Rightarrow \texttt{ab}P\texttt{ba} \Rightarrow \texttt{aba}P\texttt{aba} \Rightarrow \texttt{abaaba}$$
$$P \Rightarrow \texttt{b}P\texttt{b} \Rightarrow \texttt{ba}P\texttt{ab} \Rightarrow \texttt{baaab}$$

The parse trees corresponding to the derivations are the following:



Consequently, the desired Haskell definitions are

$$\mathit{pal}_1 = A_2\ (B_2\ (A_2\ \mathit{Empty}))$$
$$\mathit{pal}_2 = B_2\ (A_2\ A)$$

2.22

1.

$$\begin{aligned}
&\mathit{printPal} :: P \to \mathit{String}\\
&\mathit{printPal}\ \mathit{Empty} = \texttt{""}\\
&\mathit{printPal}\ A \quad\ \ = \texttt{"a"}\\
&\mathit{printPal}\ B \quad\ \ = \texttt{"b"}\\
&\mathit{printPal}\ (A_2\ p) = \texttt{"a"} +\!\!+ \mathit{printPal}\ p +\!\!+ \texttt{"a"}\\
&\mathit{printPal}\ (B_2\ p) = \texttt{"b"} +\!\!+ \mathit{printPal}\ p +\!\!+ \texttt{"b"}
\end{aligned}$$

Note how the function follows the structure of the datatype *Pal* closely, and calls itself recursively wherever a recursive value of *Pal* occurs in the datatype. Such a pattern is typical for semantic functions.

2.

$$
\begin{aligned}
&aCountPal :: P \to Int \\
&aCountPal\ Empty = 0 \\
&aCountPal\ A \quad\ = 1 \\
&aCountPal\ B \quad\ = 0 \\
&aCountPal\ (A_2\ p) = aCountPal\ p + 2 \\
&aCountPal\ (B_2\ p) = aCountPal\ p
\end{aligned}
$$

2.23 Recall the grammar from Exercise 2.8, this time with names for the productions:

MEmpty: $M \to \varepsilon$
MA: $M \to \mathtt{a}M\mathtt{a}$
MB: $M \to \mathtt{b}M\mathtt{b}$

1. By systematically transforming the grammar, we obtain the following datatype:

   data $Mir = MEmpty \mid MA\ Mir \mid MB\ Mir$

   The concrete mirror palindromes $cMir_1$ and $cMir_2$ correspond to the following terms of type Mir:

   $$
   \begin{aligned}
   &aMir_1 = MA\ (MB\ (MA\ Empty)) \\
   &aMir_2 = MA\ (MB\ (MB\ Empty))
   \end{aligned}
   $$

2.

$$
\begin{aligned}
&printMir :: Mir \to String \\
&printMir\ MEmpty = \texttt{""} \\
&printMir\ (MA\ m) = \texttt{"a"} +\!\!+ printMir\ m +\!\!+ \texttt{"a"} \\
&printMir\ (MB\ m) = \texttt{"b"} +\!\!+ printMir\ m +\!\!+ \texttt{"b"}
\end{aligned}
$$

3.

$$
\begin{aligned}
&mirToPal :: Mir \to Pal \\
&mirToPal\ MEmpty = Empty \\
&mirToPal\ (MA\ m) = A_2\ (mirToPal\ m) \\
&mirToPal\ (MB\ m) = B_2\ (mirToPal\ m)
\end{aligned}
$$

2.24 Recall the grammar from Exercise 2.9, this time with names for the productions:

Stop: $P \to \varepsilon$
POne: $P \to \mathtt{1}P\mathtt{1}$
PZeroL: $P \to \mathtt{0}P$
PZeroR: $P \to P\mathtt{0}$

1.

$$\begin{aligned}
&\textbf{data}\ \mathit{Parity} = \mathit{Stop} \mid \mathit{POne}\ \mathit{Parity} \mid \mathit{PZeroL}\ \mathit{Parity} \mid \mathit{PZeroR}\ \mathit{Parity}\\
&\mathit{aEven}_1 = \mathit{PZeroL}\ (\mathit{PZeroL}\ (\mathit{POne}\ (\mathit{PZeroL}\ \mathit{Stop})))\\
&\mathit{aEven}_2 = \mathit{PZeroL}\ (\mathit{PZeroR}\ (\mathit{POne}\ (\mathit{PZeroL}\ \mathit{Stop})))
\end{aligned}$$

Note that the grammar is ambiguous, and other representations for cEven_1 and cEven_2 are possible, for instance:

$$\begin{aligned}
&\mathit{aEven}'_1 = \mathit{PZeroL}\ (\mathit{PZeroL}\ (\mathit{POne}\ (\mathit{PZeroR}\ \mathit{Stop})))\\
&\mathit{aEven}'_2 = \mathit{PZeroR}\ (\mathit{PZeroL}\ (\mathit{POne}\ (\mathit{PZeroL}\ \mathit{Stop})))
\end{aligned}$$

2.

$$\begin{aligned}
&\mathit{printParity} :: \mathit{Parity} \to \mathit{String}\\
&\mathit{printParity}\ \mathit{Stop} = \texttt{""}\\
&\mathit{printParity}\ (\mathit{POne}\ p) = \texttt{"1"} +\!\!+ \mathit{printParity}\ p +\!\!+ \texttt{"1"}\\
&\mathit{printParity}\ (\mathit{PZeroL}\ p) = \texttt{"0"} +\!\!+ \mathit{printParity}\ p\\
&\mathit{printParity}\ (\mathit{PZeroR}\ p) = \mathit{printParity}\ p +\!\!+ \texttt{"0"}
\end{aligned}$$

2.25 A grammar for bit lists that is not left-recursive is the following:

$$\begin{aligned}
&L \to B\ Z \mid B\\
&Z \to \texttt{,}\ L\ Z \mid \texttt{,}\ L\\
&B \to \texttt{0} \mid \texttt{1}
\end{aligned}$$

1.

$$\begin{aligned}
&\textbf{data}\ \mathit{BitList} = \mathit{ConsBit}\ \mathit{Bit}\ Z \mid \mathit{SingleBit}\ \mathit{Bit}\\
&\textbf{data}\ Z = \mathit{ConsBitList}\ \mathit{BitList}\ Z \mid \mathit{SingleBitList}\ \mathit{BitList}\\
&\textbf{data}\ \mathit{Bit} = \mathit{Bit}_0 \mid \mathit{Bit}_1\\
&\mathit{aBitList}_1 = \mathit{ConsBit}\ \mathit{Bit}_0\ (\mathit{ConsBitList}\ (\mathit{SingleBit}\ \mathit{Bit}_1)\\
&\qquad\qquad (\mathit{SingleBitList}\ (\mathit{SingleBit}\ \mathit{Bit}_0)))\\
&\mathit{aBitList}_2 = \mathit{ConsBit}\ \mathit{Bit}_0\ (\mathit{ConsBitList}\ (\mathit{SingleBit}\ \mathit{Bit}_0)\\
&\qquad\qquad (\mathit{SingleBitList}\ (\mathit{SingleBit}\ \mathit{Bit}_1)))
\end{aligned}$$

2.

$$\begin{aligned}
&\mathit{printBitList} :: \mathit{BitList} \to \mathit{String}\\
&\mathit{printBitList}\ (\mathit{ConsBit}\ b\ z) = \mathit{printBit}\ b +\!\!+ \mathit{printZ}\ z\\
&\mathit{printBitList}\ (\mathit{SingleBit}\ b) = \mathit{printBit}\ b\\
&\mathit{printZ} :: Z \to \mathit{String}\\
&\mathit{printZ}\ (\mathit{ConsBitList}\ bs\ z) = \texttt{","} +\!\!+ \mathit{printBitList}\ bs +\!\!+ \mathit{printZ}\ z\\
&\mathit{printZ}\ (\mathit{SingleBitList}\ bs) = \texttt{","} +\!\!+ \mathit{printBitList}\ bs
\end{aligned}$$

$$
\begin{aligned}
&\mathit{printBit} :: \mathit{Bit} \to \mathit{String} \\
&\mathit{printBit}\ \mathit{Bit}_0 &&= \texttt{"0"} \\
&\mathit{printBit}\ \mathit{Bit}_1 &&= \texttt{"1"}
\end{aligned}
$$

When multiple datatypes are involved, semantic functions typically still follow the structure of the datatypes closely. We get one function per datatype, and the functions call each other recursively where appropriate – we say they are *mutually recursive*.

3. We can still make the concatenation function structurally recursive in the first of the two bit lists. We never have to match on the second bit list:

$$
\begin{aligned}
&\mathit{concatBitList} :: \mathit{BitList} \to \mathit{BitList} \to \mathit{BitList} \\
&\mathit{concatBitList}\ (\mathit{ConsBit}\ b\ z)\ cs = \mathit{ConsBit}\ b\ (\mathit{concatZ}\ z\ cs) \\
&\mathit{concatBitList}\ (\mathit{SingleBit}\ b)\ \ cs = \mathit{ConsBit}\ b\ (\mathit{SingleBitList}\ cs) \\
&\mathit{concatZ} :: Z \to \mathit{BitList} \to Z \\
&\mathit{concatZ}\ (\mathit{ConsBitList}\ bs\ z)\ \ cs = \mathit{ConsBitList}\ bs\ (\mathit{concatZ}\ z\ xs) \\
&\mathit{concatZ}\ (\mathit{SingleBitList}\ bs)\ \ cs = \mathit{ConsBitList}\ bs\ (\mathit{SingleBitList}\ cs)
\end{aligned}
$$

2.26 We only give the EBNF notation for the productions that change.

$$
\begin{aligned}
&\mathit{Digs} &&\to \mathit{Dig}^* \\
&\mathit{Int} &&\to \mathit{Sign}?\ \mathit{Nat} \\
&\mathit{AlphaNums} &&\to \mathit{AlphaNum}^* \\
&\mathit{AlphaNum} &&\to \mathit{Letter} \mid \mathit{Dig}
\end{aligned}
$$

2.27 $L(G?) = L(G) \cup \{\varepsilon\}$

2.28

1. L_1 is generated by:

$$
\begin{aligned}
S &\to ZC \\
Z &\to \texttt{a}Z\texttt{b} \mid \varepsilon \\
C &\to \texttt{c}^*
\end{aligned}
$$

and L_2 is generated by:

$$
\begin{aligned}
S &\to AZ \\
A &\to \texttt{a}^* \\
Z &\to \texttt{b}Z\texttt{c} \mid \varepsilon
\end{aligned}
$$

2. We have that

$$L_1 \cap L_2 = \{\, a^n b^n c^n \mid n \in \mathbb{N} \,\}$$

However, this language is not context-free, i. e., there is no context-free grammar that generates this language. We will see in Chapter **??** how to prove such a statement.

2.29 No. Furthermore, for any language L, since $\varepsilon \in L^*$, we have $\varepsilon \notin \overline{(L^*)}$. On the other hand, $\varepsilon \in (\overline{L})^*$. Thus $\overline{(L^*)}$ and $(\overline{L})^*$ cannot be equal.

2.30 For example $L = \{ x^n \mid n \in \mathbb{N} \}$. For any language L, it holds that

$$L^* = (L^*)^*$$

so, given any language L, the language L^* fulfills the desired property.

2.31 This is only the case when $\varepsilon \notin L$.

2.32 No. the language $L = \{ \mathtt{aab}, \mathtt{baa} \}$ also satisfies $L = L^R$.

2.33

1. The shortest derivation is three steps long and yields the sentence `aa`. The sentences `baa`, `aba`, and `aab` can all be derived in four steps.
2. Several derivations are possible for the string `babbab`. Two of them are

$$\begin{aligned} &\underline{S} \Rightarrow \underline{A}A \Rightarrow \mathtt{b}A\underline{A} \Rightarrow \mathtt{b}\underline{A}\mathtt{b}A \Rightarrow \mathtt{bab}\underline{A} \Rightarrow \mathtt{babb}\underline{A} \Rightarrow \mathtt{babb}\underline{A}\mathtt{b} \Rightarrow \mathtt{babbab} \\ &\underline{S} \Rightarrow A\underline{A} \Rightarrow \underline{A}A\mathtt{b} \Rightarrow \mathtt{b}\underline{A}A\mathtt{b} \Rightarrow \mathtt{b}\underline{A}\mathtt{b}A\mathtt{b} \Rightarrow \mathtt{b}\underline{A}\mathtt{bb}A\mathtt{b} \Rightarrow \mathtt{babb}\underline{A}\mathtt{b} \Rightarrow \mathtt{babbab} \end{aligned}$$

3. A leftmost derivation is:

$$\begin{aligned} \underline{S} &\Rightarrow \underline{A}A \Rightarrow^* \mathtt{b}^m\underline{A}A \Rightarrow^* \mathtt{b}^m\underline{A}\mathtt{b}^nA \Rightarrow \mathtt{b}^m\mathtt{ab}^n\underline{A} \Rightarrow^* \mathtt{b}^m\mathtt{ab}^n\underline{A}\mathtt{b}^p \\ &\Rightarrow \mathtt{b}^m\mathtt{ab}^n\mathtt{ab}^p \end{aligned}$$

2.34 The grammar is equivalent to the grammar

$$\begin{aligned} S &\to \mathtt{aa}B \\ B &\to \mathtt{b}B\mathtt{ba} \\ B &\to \mathtt{a} \end{aligned}$$

This grammar generates the string `aaa` and the strings $\mathtt{aab}^m\mathtt{a}(\mathtt{ba})^m$ for $m \in \mathbb{N}$, $m \geqslant 1$. The string `aabbaabba` does not appear in this language.

2.35 The language L is generated by:

$$S \to \mathtt{a}S\mathtt{a} \mid \mathtt{b}S\mathtt{b} \mid \mathtt{c}$$

The derivation is:

$$\underline{S} \Rightarrow \mathtt{a}\underline{S}\mathtt{a} \Rightarrow \mathtt{ab}\underline{S}\mathtt{ba} \Rightarrow \mathtt{abcba}$$

2.36 The language generated by the grammar is

$$\{ \mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N} \}$$

The same language is also generated by the grammar

$$S \to \mathtt{a}A\mathtt{b} \mid \varepsilon$$

2.37 The first language is

$$\{\mathtt{a}^n \mid n \in \mathbb{N}\}$$

This language is also generated by the grammar

$$S \to \mathtt{a}S \mid \varepsilon$$

The second language is

$$\{\varepsilon\} \cup \{\mathtt{a}^{2n+1} \mid n \in \mathbb{N}\}$$

This language is also generated by the grammar

$$S \to A \mid \varepsilon$$
$$A \to \mathtt{a} \mid \mathtt{a}A\mathtt{a}$$

or using EBNF notation

$$S \to (\mathtt{a}(\mathtt{aa})^*)?$$

2.38 All three grammars generate the language

$$\{\mathtt{a}^n \mid n \in \mathbb{N}\}$$

2.39

$$S \to A \mid \varepsilon$$
$$A \to \mathtt{a}A\mathtt{b} \mid \mathtt{ab}$$

2.40 The language is

$$L = \{\mathtt{a}^{2n+1} \mid n \in \mathbb{N}\}$$

A grammar for L without left-recursive productions is

$$A \to \mathtt{aa}A \mid \mathtt{a}$$

And a grammar without right-recursive prodcutions is

$$A \to A\mathtt{aa} \mid \mathtt{a}$$

2.41 The language is

$\{\mathtt{ab}^n \mid n \in \mathbb{N}\}$

A grammar for L without left-recursive productions is

$X \to \mathtt{a}Y$
$Y \to \mathtt{b}Y \mid \varepsilon$

A grammar for L without left-recursive productions that is also non-contracting is

$X \to \mathtt{a}Y \mid \mathtt{a}$
$Y \to \mathtt{b}Y \mid \mathtt{b}$

2.42 A grammar that uses only productions with two or less symbols on the right hand side:

$S \to T \mid US$
$T \to X\mathtt{a} \mid U\mathtt{a}$
$X \to \mathtt{a}S$
$U \to S \mid YT$
$Y \to SU$

The sentential forms $\mathtt{a}S$ and SU have been abstracted to nonterminals X and Y.

A grammar for the same language with only two nonterminals:

$S \to \mathtt{a}S\mathtt{a} \mid U\mathtt{a} \mid US$
$U \to S \mid SU\mathtt{a}S\mathtt{a} \mid SUU\mathtt{a}$

The nonterminal T has been substituted for its alternatives $\mathtt{a}S\mathtt{a}$ and $U\mathtt{a}$.

2.43

$S \to \mathtt{1}O$
$O \to \mathtt{1}O \mid \mathtt{0}N$
$N \to \mathtt{1}^*$

2.44 The language is generated by the grammar:

$S \to \mathtt{(}A\mathtt{)} \mid SS$
$A \to S \mid \varepsilon$

A derivation for `( ) ( ( ) ) ( )` is:

$\underline{S} \Rightarrow \underline{S}S \Rightarrow \underline{S}SS \Rightarrow \mathtt{(}\underline{A}\mathtt{)}SS \Rightarrow \mathtt{()}\underline{S}S \Rightarrow \mathtt{()(}\underline{A}\mathtt{)}S \Rightarrow \mathtt{()(}\underline{S}\mathtt{)}S \Rightarrow \mathtt{()((}\underline{A}\mathtt{))}S$
$\Rightarrow \mathtt{()(())}\underline{S} \Rightarrow \mathtt{()(())(}\underline{A}\mathtt{)} \Rightarrow \mathtt{()(())()}$

2.45 The language is generated by the grammar

$S \to \texttt{(}A\texttt{)} \mid \texttt{[}A\texttt{]} \mid SS$
$A \to S \mid \varepsilon$

A derivation for `[ ( ) ] ( )` is:

$$\underline{S} \Rightarrow \underline{S}S \Rightarrow \texttt{[}\underline{A}\texttt{]}S \Rightarrow \texttt{[}\underline{S}\texttt{]}S \Rightarrow \texttt{[(}\underline{A}\texttt{)]}S \Rightarrow \texttt{[()]}\underline{S} \Rightarrow \texttt{[()](}\underline{A}\texttt{)} \Rightarrow \texttt{[()]()}$$

2.46 First leftmost derivation:

$\underline{\textit{Sentence}}$
$\Rightarrow \underline{\textit{Subject}}\ \textit{Predicate}$
$\Rightarrow$ `they` $\underline{\textit{Predicate}}$
$\Rightarrow$ `they` $\underline{\textit{Verb}}\ \textit{NounPhrase}$
$\Rightarrow$ `they are` $\underline{\textit{NounPhrase}}$
$\Rightarrow$ `they are` $\underline{\textit{Adjective}}\ \textit{Noun}$
$\Rightarrow$ `they are flying` $\underline{\textit{Noun}}$
$\Rightarrow$ `they are flying planes`

Second leftmost derivation:

$\underline{\textit{Sentence}}$
$\Rightarrow \underline{\textit{Subject}}\ \textit{Predicate}$
$\Rightarrow$ `they` $\underline{\textit{Predicate}}$
$\Rightarrow$ `they` $\underline{\textit{AuxVerb}}\ \textit{Verb}\ \textit{Noun}$
$\Rightarrow$ `they are` $\underline{\textit{Verb}}\ \textit{Noun}$
$\Rightarrow$ `they are flying` $\underline{\textit{Noun}}$
$\Rightarrow$ `they are flying planes`

2.48 Here is an unambiguous grammar for the language from Exercise 2.45:

$S \to \texttt{(}E\texttt{)}E \mid \texttt{[}E\texttt{]}\ E$
$E \to \varepsilon \mid S$

2.49 Here is a leftmost derivation for ♣ ◇ ♣ △ ♠.

$$\underline{\odot} \Rightarrow \underline{\odot} \triangle \otimes \Rightarrow \underline{\otimes} \triangle \otimes \Rightarrow \underline{\otimes} \diamond \oplus \triangle \otimes \Rightarrow \underline{\oplus} \diamond \oplus \triangle \otimes \Rightarrow \clubsuit \diamond \underline{\oplus} \triangle \otimes$$
$$\Rightarrow \clubsuit \diamond \clubsuit \triangle \underline{\otimes} \Rightarrow \clubsuit \diamond \clubsuit \triangle \underline{\oplus} \Rightarrow \clubsuit \diamond \clubsuit \triangle \spadesuit$$

Notice that the grammar of this exercise is the same, up to renaming, as the grammar

$E \to E\ \texttt{+}\ T \mid T$
$T \to T\ \texttt{*}\ F \mid F$
$F \to \texttt{0} \mid \texttt{1}$

2.50 The palindrome ε with length 0 can be generated by de grammar with the derivation $P \Rightarrow \varepsilon$. The palindromes `a`, `b`, and `b` are the palindromes with length 1. They can be derivated with $P \Rightarrow$ `a`, $P \Rightarrow$ `b`, $P \Rightarrow$ `c`, respectively.

Suppose that the palindrome s with a length of 2 or more. Then s can be written as `a`t`a` or `b`t`b` or `c`t`c` where the length of t is strictly smaller, and t also is a palindrome. Thus, by induction hypothesis, there is a derivation $P \Rightarrow^* t$. But then, there is also a derivation for s, for example $P \Rightarrow^* t \Rightarrow$ `a`t`a` in the first situation – the other two cases are analogous.

We have now proved that any palindrome can be generated by the grammar – we still have to prove that anything generated by the grammar is a palindrome, but this is easy to see by induction over the length of derivations. Certainly ε, a, b, and c are palindromes. And if s is a palindrome that can be derived, so are `a`s`a`, `b`s`b`, and `c`s`c`.

3.1 Either we use the predefined predicate *isUpper* in module *Data.Char*,

```
capital = satisfy isUpper
```

or we make use of the ordering defined characters,

```
capital = satisfy (λs → ('A' ⩽ s) ∧ (s ⩽ 'Z'))
```

3.2 A symbol equal to a satisfies the predicate $(== a)$:

```
symbol a = satisfy (== a)
```

3.3 The function *epsilon* is a special case of *succeed*:

```
epsilon :: Parser s ()
epsilon = succeed ()
```

3.4 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
```

3.5 The type and results of (:) <$> *symbol* `'a'` are (note that you cannot write this as a definition in Haskell):

```
((:) <$> symbol 'a') :: Parser Char (String → String)
((:) <$> symbol 'a') []                    = []
((:) <$> symbol 'a') (x : xs) | x == 'a'   = [((x:), xs)]
                              | otherwise = []
```

3.6 The type and results of (:) <$> *symbol* 'a' <*> *p* are:

```
((:) <$> symbol 'a' <*> p) :: Parser Char String
((:) <$> symbol 'a' <*> p) [] = []
((:) <$> symbol 'a' <*> p) (x : xs)
  | x == 'a'   = [('a' : x, ys) | (x, ys) ← p xs]
  | otherwise = []
```

3.7

```
pBool :: Parser Char Bool
pBool  =  const True <$> token "True"
       <|> const False <$> token "False"
```

3.9

1.

```
data Pal2 = Nil | Leafa | Leafb | Twoa Pal2 | Twob Pal2
```

2.

```
palin2 :: Parser Char Pal2
palin2  =  (\_y _ → Twoa y) <$>
           symbol 'a' <*> palin2 <*> symbol 'a'
       <|> (\_y _ → Twob y) <$>
           symbol 'b' <*> palin2 <*> symbol 'b'
       <|> const Leafa <$> symbol 'a'
       <|> const Leafb <$> symbol 'b'
       <|> succeed Nil
```

3.

```
palina :: Parser Char Int
palina  =  (λ_ y _ → y + 2) <$>
           symbol 'a' <*> palina <*> symbol 'a'
       <|> (λ_ y _ → y)      <$>
           symbol 'b' <*> palina <*> symbol 'b'
       <|> const 1 <$> symbol 'a'
       <|> const 0 <$> symbol 'b'
       <|> succeed 0
```

3.10

1.

```
data English = E1 Subject Pred
data Subject = E2 String
data Pred    = E3 Verb NounP | E4 AuxV Verb Noun
data Verb    = E5 String | E6 String
data AuxV    = E7 String
data NounP   = E8 Adj Noun
data Adj     = E9 String
data Noun    = E10 String
```

2.

```
english :: Parser Char English
english  =  E1  <$> subject <*> pred
subject  =  E2  <$> token "they"
pred     =  E3  <$> verb <*> nounp
         <|> E4  <$> auxv <*> verb <*> noun
verb     =  E5  <$> token "are"
         <|> E6  <$> token "flying"
auxv     =  E7  <$> token "are"
nounp    =  E8  <$> adj <*> noun
adj      =  E9  <$> token "flying"
noun     =  E10 <$> token "planes"
```

3.11 As <|> uses ++, it is more efficiently evaluated if right-associative.

3.12 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
               ]
```

3.13 'Parser transformator', or 'parser modifier' or 'parser postprocessor', etcetera.

3.14 The transformator <$> does to the result part of parsers what *map* does to the elements of a list.

3.15 The parser combinators <*> and <,> can be defined in terms of each other:

```
p <*> q = uncurry ($) <$> (p <,> q)
p <,> q = (,) <$> p <*> q
```

3.16 Yes. You can combine the parser parameter of <$> with a parser that consumes no input and always yields the function parameter of <$>:

$$f \mathbin{<\$>} p = succeed\ f \mathbin{<*>} p$$

3.17

$$\begin{aligned}
&f && :: \\
&\quad Char \to Parentheses \to Char \to Parentheses \to Parentheses \\
&\quad open && :: \\
&\quad Parser\ Char\ Char \\
&f \mathbin{<\$>} open && :: \\
&\quad Parser\ Char\ (Parentheses \to Char \to Parentheses \to Parentheses) \\
&\quad parens :: \\
&\quad Parser\ Char\ Parentheses \\
&(f \mathbin{<\$>} open) \mathbin{<*>} parens :: \\
&\quad Parser\ Char\ (Char \to Parentheses \to Parentheses)
\end{aligned}$$

3.18 To the left. Yes.

3.19 The function has to check whether applying the parser p to input s returns at least one result with an empty rest sequence:

$$test\ p\ s = not\ (null\ (filter\ (null\ .\ snd)\ (p\ s)))$$

3.20

1. $$\begin{aligned}
   &listofdigits :: Parser\ Char\ [Int] \\
   &listofdigits = listOf\ newdigit\ (symbol\ \texttt{' '})
   \end{aligned}$$

   ```
   ?> 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 (+).

   $$\begin{aligned}
   &plusParser :: Num\ a \Rightarrow Parser\ Char\ (a \to a \to a) \\
   &plusParser\ [] && = [] \\
   &plusParser\ (x : xs) \mid x == \texttt{'+'} && = [((+), xs)] \\
   &\qquad\qquad\qquad\quad \mid otherwise && = []
   \end{aligned}$$

The definiton of the parser *sumParser* is:

$$\begin{aligned}&sumParser :: Parser\ Char\ Int\\&sumParser = chainr\ newdigit\ plusParser\end{aligned}$$

```
?> 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*.

3.21 We introduce the abbreviation

$$listOfa = (:) <\$> symbol\ \texttt{'a'}$$

and use the results of Exercises 3.5 and 3.6.

$xs = []$:

$$\begin{aligned}&many\ (symbol\ \texttt{'a'})\ []\\=\quad&\{\text{ definition of } many \text{ and } listOfa\ \}\\&(listOfa <*> many\ (symbol\ \texttt{'a'}) <|> succeed\ []) []\\=\quad&\{\text{ definition of } <|>\ \}\\&(listOfa <*> many\ (symbol\ \texttt{'a'})) [] +\!\!+ succeed\ [] []\\=\quad&\{\text{ Exercise 3.6, definition of } succeed\ \}\\&[] +\!\!+ [([], [])]\\=\quad&\{\text{ definition of } +\!\!+\ \}\\&[([], [])]\end{aligned}$$

$xs = [\texttt{'a'}]$:

$$\begin{aligned}&many\ (symbol\ \texttt{'a'})\ [\texttt{'a'}]\\=\quad&\{\text{ definition of } many \text{ and } listOfa\ \}\\&(listOfa <*> many\ (symbol\ \texttt{'a'}) <|> succeed\ []) [\texttt{'a'}]\\=\quad&\{\text{ definition of } <|>\ \}\\&(listOfa <*> many\ (symbol\ \texttt{'a'})) [\texttt{'a'}] +\!\!+ succeed\ [] [\texttt{'a'}]\\=\quad&\{\text{ Exercise 3.6, previous calculation }\}\end{aligned}$$

$[([\texttt{'a'}], []), ([], [\texttt{'a'}])]$

$xs = [\texttt{'b'}]$:

$$
\begin{aligned}
& \mathit{many}\ (\mathit{symbol}\ \texttt{'a'})\ [\texttt{'b'}] \\
= \quad & \{\ \text{as before}\ \} \\
& (\mathit{listOfa} <\!\!*\!\!> \mathit{many}\ (\mathit{symbol}\ \texttt{'a'}))\ [\texttt{'b'}] +\!\!+ \mathit{succeed}\ []\ [\texttt{'b'}] \\
= \quad & \{\ \text{Exercise 3.6, previous calculation}\ \} \\
& [([], [\texttt{'b'}])]
\end{aligned}
$$

$xs = [\texttt{'a'}, \texttt{'b'}]$:

$$
\begin{aligned}
& \mathit{many}\ (\mathit{symbol}\ \texttt{'a'})\ [\texttt{'a'}, \texttt{'b'}] \\
= \quad & \{\ \text{as before}\ \} \\
& (\mathit{listOfa} <\!\!*\!\!> \mathit{many}\ (\mathit{symbol}\ \texttt{'a'}))\ [\texttt{'a'}, \texttt{'b'}] +\!\!+ \mathit{succeed}\ []\ [\texttt{'a'}, \texttt{'b'}] \\
= \quad & \{\ \text{Exercise 3.6, previous calculation}\ \} \\
& [([\texttt{'a'}], [\texttt{'b'}]), ([], [\texttt{'a'}, \texttt{'b'}])]
\end{aligned}
$$

$xs = [\texttt{'a'}, \texttt{'a'}, \texttt{'b'}]$:

$$
\begin{aligned}
& \mathit{many}\ (\mathit{symbol}\ \texttt{'a'})\ [\texttt{'a'}, \texttt{'a'}, \texttt{'b'}] \\
= \quad & \{\ \text{as before}\ \} \\
& (\mathit{listOfa} <\!\!*\!\!> \mathit{many}\ (\mathit{symbol}\ \texttt{'a'}))\ [\texttt{'a'}, \texttt{'a'}, \texttt{'b'}] +\!\!+ \mathit{succeed}\ []\ [\texttt{'a'}, \texttt{'a'}, \texttt{'b'}] \\
= \quad & \{\ \text{Exercise 3.6, previous calculation}\ \} \\
& [([\texttt{'a'}, \texttt{'a'}], [\texttt{'b'}]), ([\texttt{'a'}], [\texttt{'a'}, \texttt{'b'}]), ([], [\texttt{'a'}, \texttt{'a'}, \texttt{'b'}])]
\end{aligned}
$$

3.22 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.

3.24

```
— Combinators for repetition
psequence :: [Parser s a] → Parser s [a]
psequence []       = succeed []
psequence (p : ps) = (:) <$> p <*> psequence ps

psequence' :: [Parser s a] → Parser s [a]
psequence' = foldr f (succeed [])
  where f p q = (:) <$> p <*> q
```

$choice :: [Parser\ s\ a] \to Parser\ s\ a$
$choice = foldr\ (<|>)\ failp$

```
?> (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"
[]
```

3.25

$token :: Eq\ s \Rightarrow [s] \to Parser\ s\ [s]$
$token = psequence\ .\ map\ symbol$

3.27

$identifier :: Parser\ Char\ String$
$identifier = (:) <\$> satisfy\ isAlpha <*> greedy\ (satisfy\ isAlphaNum)$

3.28

1. As Haskell terms:

`"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)")]
```

3.29 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","()")]
```

3.30

```
expr = chainr (chainl term (const (:−:) <$> symbol '-'))
              (const (:+:) <$> symbol '+')
```

3.31 The datatype *Expr* is extended as follows to allow raising an expression to the power of an expression:

```
data Expr = Con Int
          | Var String
```

$$
\begin{aligned}
&\mid \mathit{Fun}\ \mathit{String}\ [\mathit{Expr}] \\
&\mid \mathit{Expr}\ \mathit{:+:}\ \mathit{Expr} \\
&\mid \mathit{Expr}\ \mathit{:-:}\ \mathit{Expr} \\
&\mid \mathit{Expr}\ \mathit{:*:}\ \mathit{Expr} \\
&\mid \mathit{Expr}\ \mathit{:/:}\ \mathit{Expr} \\
&\mid \mathit{Expr}\ \mathit{:\hat{}:}\ \mathit{Expr} \\
&\textbf{deriving}\ \mathit{Show}
\end{aligned}
$$

Now the parser $expr'$ of Listing 3.10 can be extended with a new level of priorities:

$$
\begin{aligned}
&\mathit{powis} = [('\hat{}', (\mathit{:\hat{}:}))] \\
&\mathit{expr}' :: \mathit{Parser}\ \mathit{Char}\ \mathit{Expr} \\
&\mathit{expr}' = \mathit{foldr}\ \mathit{gen}\ \mathit{fact}'\ [\mathit{addis}, \mathit{multis}, \mathit{powis}]
\end{aligned}
$$

Note that because of the use of *chainl* all the operators listed in *addis*, *multis* and *powis* are treated as left-associative.

3.32 The proofs can be given by using laws for list comprehension, but here we prefer to exploit the following equation

$$(f <\$> p)\ xs = \mathit{map}\ (f \ast\!\!\ast\!\!\ast\ \mathit{id})\ (p\ xs) \tag{A.1}$$

where $(\ast\!\!\ast\!\!\ast)$ is defined by

$$
\begin{aligned}
&(\ast\!\!\ast\!\!\ast) :: (a \to c) \to (b \to d) \to (a, b) \to (c, d) \\
&(f \ast\!\!\ast\!\!\ast\ g)\ (a, b) = (f\ a, g\ b)
\end{aligned}
$$

It has the following property:

$$(f \ast\!\!\ast\!\!\ast\ g)\ .\ (h \ast\!\!\ast\!\!\ast\ k) = (f\ .\ h) \ast\!\!\ast\!\!\ast\ (g\ .\ k) \tag{A.2}$$

Furthermore, we will use the following laws about *map* in our proof: map distributes over composition, concatenation, and the function *concat*:

$$\mathit{map}\ f\ .\ \mathit{map}\ g\ = \mathit{map}\ (f\ .\ g) \tag{A.3}$$

$$\mathit{map}\ f\ (x +\!\!+ y) = \mathit{map}\ f\ x +\!\!+ \mathit{map}\ f\ y \tag{A.4}$$

$$\mathit{map}\ f\ .\ \mathit{concat}\ = \mathit{concat}\ .\ \mathit{map}\ (\mathit{map}\ f) \tag{A.5}$$

1.

$$
\begin{aligned}
&\quad (h <\$> (f <\$> p))\ xs \\
&= \quad \{\ (\text{A.1})\ \} \\
&\quad \mathit{map}\ (h \ast\!\!\ast\!\!\ast\ \mathit{id})\ ((f <\$> p)\ xs) \\
&= \quad \{\ (\text{A.1})\ \} \\
&\quad \mathit{map}\ (h \ast\!\!\ast\!\!\ast\ \mathit{id})\ (\mathit{map}\ (f \ast\!\!\ast\!\!\ast\ \mathit{id})\ (p\ xs))
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{\ \text{(A.3)}\ \} \\
&\quad map\ ((h \mathbin{***} id)\ .\ (f \mathbin{***} id))\ (p\ xs) \\
&= \quad \{\ \text{(A.2)}\ \} \\
&\quad map\ ((h\ .\ f) \mathbin{***} id)\ (p\ xs) \\
&= \quad \{\ \text{(A.1)}\ \} \\
&\quad ((h\ .\ f) <\$>\ p)\ xs
\end{aligned}
$$

2.

$$
\begin{aligned}
&\quad (h <\$> (p <|> q))\ xs \\
&= \quad \{\ \text{(A.1)}\ \} \\
&\quad map\ (h \mathbin{***} id)\ ((p <|> q)\ xs) \\
&= \quad \{\ \text{definition of } <|>\ \} \\
&\quad map\ (h \mathbin{***} id)\ (p\ xs +\!\!+ q\ xs) \\
&= \quad \{\ \text{(A.4)}\ \} \\
&\quad map\ (h \mathbin{***} id)\ (p\ xs) +\!\!+ map\ (h \mathbin{***} id)\ (q\ xs) \\
&= \quad \{\ \text{(A.1)}\ \} \\
&\quad (h <\$>\ p)\ xs +\!\!+ (h <\$>\ q)\ xs \\
&= \quad \{\ \text{definition of } <|>\ \} \\
&\quad ((h <\$>\ p) <|> (h <\$>\ q))\ xs
\end{aligned}
$$

3. First note that $(p <\!*\!> q)\ xs$ can be written as

$$(p <\!*\!> q)\ xs = concat\ (map\ (mc\ q)\ (p\ xs)) \tag{A.6}$$

where

$$mc\ q\ (f, ys) = map\ (f \mathbin{***} id)\ (q\ ys)$$

Now we calculate

$$
\begin{aligned}
&\quad (((h.) <\$>\ p) <\!*\!> q)\ xs \\
&= \quad \{\ \text{(A.6)}\ \} \\
&\quad concat\ (map\ (mc\ q)\ (((h.) <\$>\ p)\ xs)) \\
&= \quad \{\ \text{(A.1)}\ \} \\
&\quad concat\ (map\ (mc\ q)\ (map\ ((h.) \mathbin{***} id)\ (p\ xs))) \\
&= \quad \{\ \text{(A.3)}\ \} \\
&\quad concat\ (map\ (map\ (h \mathbin{***} id)\ (map\ (mc\ q)\ (p\ xs)))) \\
&= \quad \{\ \text{(A.7), see below}\ \} \\
&\quad concat\ (map\ ((map\ (h \mathbin{***} id))\ .\ mc\ q)\ (p\ xs))
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{\ (A.3)\ \} \\
&\quad\ concat\ (map\ (map\ (h \mathbin{***} id))\ (map\ (mc\ q)\ (p\ xs))) \\
&= \quad \{\ (A.5)\ \} \\
&\quad\ map\ (h \mathbin{***} id)\ (concat\ (map\ (mc\ q)\ (p\ xs))) \\
&= \quad \{\ (A.6)\ \} \\
&\quad\ map\ (h \mathbin{***} id)\ ((p \mathbin{<\!\!*\!\!>} q)\ xs) \\
&= \quad \{\ (A.1)\ \} \\
&\quad\ (h \mathbin{<\!\!\$\!\!>} (p \mathbin{<\!\!*\!\!>} q))\ xs
\end{aligned}
$$

It remains to prove the claim

$$
mc\ q\ .\ ((h.) \mathbin{***} id) = map\ (h \mathbin{***} id)\ .\ mc\ q \tag{A.7}
$$

This claim is also proved by calculation:

$$
\begin{aligned}
&\quad\ ((map\ (h \mathbin{***} id))\ .\ mc\ q)\ (f, ys) \\
&= \quad \{\ \text{definition of .}\ \} \\
&\quad\ map\ (h \mathbin{***} id)\ (mc\ q\ (f, ys)) \\
&= \quad \{\ \text{definition of } mc\ q\ \} \\
&\quad\ map\ (h \mathbin{***} id)\ (map\ (f \mathbin{***} id)\ (q\ ys)) \\
&= \quad \{\ map \text{ and } *** \text{ distribute over composition}\ \} \\
&\quad\ map\ ((h\ .\ f) \mathbin{***} id)\ (q\ y) \\
&= \quad \{\ \text{definition of } mc\ q\ \} \\
&\quad\ mc\ q\ (h\ .\ f, ys) \\
&= \quad \{\ \text{definition of } ***\ \} \\
&\quad\ (mc\ q\ .\ ((h.) \mathbin{***} id))\ (f, ys)
\end{aligned}
$$

3.33

```
pMir :: Parser Char Mir
pMir  =  (λ_ m _ → MB m) <$> symbol 'b' <*> pMir <*> symbol 'b'
      <|> (λ_ m _ → MA m) <$> symbol 'a' <*> pMir <*> symbol 'a'
      <|> succeed MEmpty
```

3.34

```
pBitList :: Parser Char BitList
pBitList  =  SingleB <$> pBit
          <|> (λb _ bs → ConsB b bs) <$> pBit <*> symbol ',' <*> pBitList
pBit      =  const Bit0 <$> symbol '0'
          <|> const Bit1 <$> symbol '1'
```

3.35

```
— Parser for floating point numbers
fixed :: Parser Char Float
fixed = (+) <$> (fromIntegral <$> greedy integer)
            <*> (((λ_ y → y) <$> symbol '.' <*> fractpart) `option` 0.0)
fractpart :: Parser Char Float
fractpart = foldr f 0.0 <$> greedy newdigit
  where f d n = (n + fromIntegral d) / 10.0
```

3.36

```
float :: Parser Char Float
float =   f <$> fixed
          <*> (((λ_ y → y) <$> symbol 'E' <*> integer) `option` 0)
  where f m e             = m * power e
        power e | e < 0   = 1.0 / power (−e)
                | otherwise = fromIntegral (10e)
```

3.37 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
       <$> identifier
        <*> ((λ_ y _ → y) <$> 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

```
data FloatLiteral  =  FL1 IntPart FractPart ExponentPart FloatSuffix
                   |  FL2 FractPart ExponentPart FloatSuffix
                   |  FL3 IntPart ExponentPart FloatSuffix
```

$$
\begin{array}{lcl}
 & | & FL_4\ IntPart\ ExponentPart\ FloatSuffix \\
\multicolumn{3}{l}{\textbf{deriving}\ Show} \\
\multicolumn{3}{l}{\textbf{type}\ ExponentPart = String} \\
\multicolumn{3}{l}{\textbf{type}\ ExponentIndicator = String} \\
\multicolumn{3}{l}{\textbf{type}\ SignedInteger = String} \\
\multicolumn{3}{l}{\textbf{type}\ IntPart = String} \\
\multicolumn{3}{l}{\textbf{type}\ FractPart = String} \\
\multicolumn{3}{l}{\textbf{type}\ FloatSuffix = String} \\
digit & = & satisfy\ isDigit \\
digits & = & many_1\ digit \\
floatLiteral & = & (\lambda a\ b\ c\ d\ e \rightarrow FL_1\ a\ c\ d\ e) \\
 & & <\$>\ intPart <*> period <*> optfract <*> optexp <*> optfloat \\
 & <|> & (\lambda a\ b\ c\ d \rightarrow FL_2\ b\ c\ d) \\
 & & <\$>\ period <*> fractPart <*> optexp <*> optfloat \\
 & <|> & (\lambda a\ b\ c \rightarrow FL_3\ a\ b\ c) \\
 & & <\$>\ intPart <*> exponentPart <*> optfloat \\
 & <|> & (\lambda a\ b\ c \rightarrow FL_4\ a\ b\ c) \\
 & & <\$>\ intPart <*> optexp <*> floatSuffix \\
intPart & = & signedInteger \\
fractPart & = & digits \\
exponentPart & = & (+\!\!+) <\$>\ exponentIndicator <*> signedInteger \\
signedInteger & = & (+\!\!+) <\$>\ option\ sign\ \texttt{""} <*> digits \\
exponentIndicator & = & token\ \texttt{"e"} <|> token\ \texttt{"E"} \\
sign & = & token\ \texttt{"+"} <|> token\ \texttt{"-"} \\
floatSuffix & = & token\ \texttt{"f"} <|> token\ \texttt{"F"} \\
 & <|> & token\ \texttt{"d"} <|> token\ \texttt{"D"} \\
period & = & token\ \texttt{"."} \\
optexp & = & option\ exponentPart\ \texttt{""} \\
optfract & = & option\ fractPart\ \texttt{""} \\
optfloat & = & option\ floatSuffix\ \texttt{""}
\end{array}
$$

4.2 The data and type definitions are the same as before, only the parsers return another (semantic) result.

```
digit             = f <$> satisfy isDigit
                    where f c = ord c - ord '0'
digits            = foldl f 0 <$> many1 digit
                    where f a b = 10*a + b
floatLiteral      =  (\a b c d e -> (fromIntegral a + c) * power d) <$>
                     intPart <*> period <*> optfract <*> optexp <*> optfloat
                   <|> (\a b c d -> b * power c) <$>
                     period <*> fractPart <*> optexp <*> optfloat
```

```
                  <|> (\a b c -> (fromIntegral a) * power b) <$>
                      intPart <*> exponentPart <*> optfloat
                  <|> (\a b c -> (fromIntegral a) * power b) <$>
                      intPart <*> optexp <*> floatSuffix
intPart           =  signedInteger
fractPart         =  foldr f 0.0 <$> many1 digit
                     where f a b = (fromIntegral a + b)/10
exponentPart      =  (\x y -> y) <$> exponentIndicator <*> signedInteger
signedInteger     =  (\ x y -> x y) <$> option sign id <*> digits
exponentIndicator =  symbol 'e' <|> symbol 'E'
sign              =  const id <$> symbol '+'
                  <|> const negate <$> symbol '-'
floatSuffix       =  symbol 'f' <|> symbol 'F'<|> symbol 'd' <|> symbol 'D'
period = symbol '.'
optexp = option exponentPart 0
optfract = option fractPart 0.0
optfloat = option floatSuffix ' '
power e  | e < 0     = 1 / power (-e)
         | otherwise = fromIntegral (10^e)
```

4.3 The parsing scheme for Java floats is

```
digit             = f <$> satisfy isDigit
                    where f c = .....
digits            = f <$> many1 digit
                    where f ds = ..
floatLiteral      =  f1 <$>
                     intPart <*> period <*> optfract <*> optexp <*> optfloat
                  <|> f2 <$>
                     period <*> fractPart <*> optexp <*> optfloat
                  <|> f3 <$>
                     intPart <*> exponentPart <*> optfloat
                  <|> f4 <$>
                     intPart <*> optexp <*> floatSuffix
                     where
                     f1 a b c d e = .....
                     f2 a b c d   = .....
                     f3 a b c     = .....
                     f4 a b c     = .....
intPart           = signedInteger
fractPart         = f <$> many1 digit
                    where f ds = ....
exponentPart      = f <$> exponentIndicator <*> signedInteger
                    where f x y = .....
```

```
signedInteger     = f <$> option sign ?? <*> digits
                    where f x y = .....
exponentIndicator = f1 <$> symbol 'e' <|> f2 <$> symbol 'E'
                    where
                    f1 c = ....
                    f2 c = ..
sign              = f1 <$> symbol '+' <|> f2 <$> symbol '-'
                    where
                    f1 h = .....
                    f2 h = .....
floatSuffix       =  f1 <$> symbol 'f'
                  <|> f2 <$> symbol 'F'
                  <|> f3 <$> symbol 'd'
                  <|> f4 <$> symbol 'D'
                     where
                     f1 c = .....
                     f2 c = .....
                     f3 c = .....
                     f4 c = .....
period = symbol '.'
optexp = option exponentPart ??
optfract = option fractPart ??
optfloat = option floatSuffix ??
```