Implementing Programming Languages

Aarne Ranta

February 6, 2012

Contents

Introduction

This book aims to make programming language implementation as easy as possible. It will guide you through all the phases of the design and implementation of a compiler or an interpreter. You can learn the material in one or two weeks and then build your own language as a matter of hours or days.

The book is different from traditional compiler books in several ways:

- it is much thinner, yet covers all the material needed for the task
- it leaves low-level details to standard tools whenever available
- it has more pure theory (inference rules) but also more actual practice (how to write the code)

Of course, it is not a substitute for the "real" books if you want to do research in compilers, or if you are involved in cutting edge implementations of large programming languages. Things that we have left out include low-level buffering in lexer input, algorithms for building LR parser generators, data flow analysis, register allocation, memory management, and parallelism. Reading hints will be given for material on these topics. In particular, the "Dragon Book"

> Aho, Lam, Sethi & Ullman: *Compilers Principles, Techniques & Tools* Second edition, Pearson/Addison Wesley 2007

covers most of these topics. But it does not cover all the things in this book.

Due to the approach chosen, you will get very quickly into the business of actually implementing your language and running programs written in it. The goal of this exercise is twofold:

1. to design and implement new programming languages, especially domain-specific ones

2. to get an idea of how compilers work

For the latter goal, we have a few theory chapters and sections, marked with an asterisk (*). These chapters can safely be left out if you are only interested in Goal 1. But of course, to talk with any authority about compilers, the knowledge of the underlying theory is essential. The theory sections try to make this interesting and relevant, answering to questions that are likely to arise, such as:

- what exactly can be done in a parser
- why can't a compiler detect all errors in programs

Practical work is an essential part of this book. You cannot claim really to have read this book unless you have done the **main assignment**, which consists of four parts:

1. a grammar and parser for a fragment of C++
2. a type checker of a smaller fragment of C++
3. an interpreter
4. a compiler to Java Virtual Machine

What is here meant by C++ is a small fragment of this immense language. We could as well say C or Java, except for part 1, which contains many of the tricky special features of C++ such as templates. The idea with part 1 is to throw you into cold water and show that you can actually swim. Managing to do this assigment will give you confidence that you can cope with *any* feature of programming language syntax easily.

Assignments 2, 3, and 4 deal with a smaller part of C++, but contain everything that is needed for writing useful programs: arithmetic expressions, declarations and assignments, if-else clauses, while loops, blocks, functions. They will give the basic understanding of how programming languages work, and this understanding can be applied to numerous variations of the same themes.

The main assignment is not only practical but also close to the "real world". Thus we don't use toy languages and home-made virtual machines, but fragments of a real language (C++), and a real virtual machine (JVM).

This makes it for instance possible to produce your own Java class files and link them together with files generated by standard Java compilers. When running your code, you will most certainly experience the embarrassment (and pleasure!) of seeing byte code verification errors, which rarely arise with the standard compilers!

You accomplish all tasks by writing two kinds of code:

- a grammar formalism: BNFC (= BNF Converter; BNF = Backus Naur Form)
- a general-purpose programming language: Java or Haskell

Thus you don't need to write code for traditional compiler tools such as Lex and YACC. Such code, as well as many other parts of the compiler, are automatically derived from the BNFC grammar. For the general-purpose language, you could actually choose any of Java, Haskell, C, C++, C#, or OCaml, since BNFC supports all these languages. But in this book, we will focus on the use of Java and Haskell as implementation language: you can choose either of them in accordance with your taste and experience. If you want to use C++ or C#, you can easily follow the Java code examples, whereas OCaml programmers can follow Haskell. C is a little different, but of course closer to Java than to Haskell.

In addition to the main assignment, the book provides optional minor assignments:

1. an interpreter of a functional language (a fragment of Haskell)
2. the design and implementation of your own language
3. a translator for natural language

The last one of these themes is included to give perspective. The history of programming languages shows a steady development towards higher-level languages—in a sense, coming closer and closer to natural languages. The point of this assignment (and the last chapter as a whole) is to try how far one can get. You will find this to be either surprisingly easy, if you are positively minded, or hopelessly difficult, if you are more pessimistic. But this is an area with a lot of future potential. Applications such as speech-based human-computer interaction and automatic translation are getting commonplace. Their connection with traditional programming language technology

will be illustrated by a natural-language-like specification language, which can be both processed automatically, and understood by layman managers and customers.

Chapter 1

What is a programming language implementation

This chapter introduces the **compilation phases**, fixing the concepts and terminology for most of the later discussion. It explains the difference between compilers and interpreters, the division into low and high level languages, and the datastructures and algorithms involved in each compilation phase.

1.1 From language to binary

As everyone knows, computers manipulate 0's and 1's. This is done by the help of electronic circuits, where 0 means no current goes through whereas 1 means that it does. The reason this is useful is that so many things—in a sense, all information—can be expressed by using just 0's and 1's—by **binary sequences**. One way to see this is to think about information in terms of yes/no questions. A sequence of answers to enough many questions can specify any object. For instance, a popular game in my childhood was to guess a person by means of maximally 20 yes/no questions.

The first thing to encode in binary are the integers:

```
0 = 0
1 = 1
2 = 10
3 = 11
4 = 100
```

and so on. This generalizes easily to letters and to other characters, for instance by the use of the ASCII encoding:

```
A = 65 = 1000001
B = 66 = 1000010
```

and so on. In this way we can see that all **data** manipulated by computers can be expressed by 0's and 1's. But what is crucial is that even the **programs** that manipulate the data can be so expressed. To take a real-world example, programs in the JVM machine language (**Java Virtual Machine**) are sequences of **bytes**, that is, groups of eight 0's or 1's (capable of expressing the numbers from 0 to 255). A byte can encode a numeric value, for instance an integer or a character as above. But it can also encode an **instruction**, that is, a command to do something. For instance, addition and multiplication (of integers) are expressed in JVM as bytes as follows:

```
+ =  96 = 0110 0000
* = 104 = 0110 1000
```

(We will put a space in the middle of each bytes to make it more readable, and more spaces between bytes.)

From the encodings of numbers and operators, one could construct a simple-minded encoding of arithmetic formulas, by just putting together the codes for 5, +, and 6:

```
5 + 6 = 0000 0101    0110 0000    0000 0110
```

While this could be made to work, actual JVM works in a more roundabout way. In the logic that it follows, the expression is first converted to a **postfix** form, where the operands come before the operator:

```
5 + 6 ---> 5 6 +
```

One virtue of the postfix form is that we don't need brackets. For instance,

```
(5 + 6) * 7  --> 5 6 + 7 *
5 + (6 * 7)  --> 5 6 7 * +
```

At least the former expression needs brackets when the usual **infix** order is used, that is, when the operator is between the operands.

The way the JVM machine manipulates such expressions is based on a so-called **stack**, which is the working memory of the machine. The stack is like a pile of plates, where new plates are **pushed** on the stack, and only one plate is available at a time, the one last pushed—known as the **top** of the stack. An arithmetic operation such as + (usually called "add") takes the the two top-most elements from the stack and returns their sum on the top. Thus the computation of, say, 5 + 6, proceeds as follows, where the left column shows the instructions and the right column the stack after each instruction:

```
push 5   ;  5
push 6   ;  5 6
add      ;  11
```

The computation of 5 + (6 * 7) is

```
push 5   ; 5
push 6   ; 5 6
push 7   ; 5 6 7
mul      ; 5 42
add      ; 47
```

In this case, unlike the previous one, the stack at one point contains more numbers than two; but the multiplication ("mul") instruction correctly picks the topmost ones 6 and 7 and returns the value 42 on the stack.

The binary JVM code must make it clear which bytes stand for numeric values and which ones for instructions such as "add". This is obvious if you think that we need to read 0110 0000 sometimes as number 96, and sometimes as addition. The way to make it clear that a byte stands for a numeric value is to prefix it with a special instruction, which is (surprise surprise!) called "push". Thus we get the code for an addition expression:

```
5 + 6 ---> push 5 push 6 add
```

To convert this all into binary, we only need the code for the push instruction,

```
push = 16 = 0001 0000
```

Now we can express the entire arithmetic expression as binary:

```
5 + 6 = 0001 0000  0000 0101  0001 0000  0000 0110  0110 0000
```

We hope to have made two important things clear now:

- Both data and programs can be expressed as **binary code**, i.e. by 0's and 1's.
- There is a systematic translation from conventional ("user-friendly") expressions to binary code.

Of course we will need more instructions to represent variables, assignments, loops, functions, and other constructs found in programming languages, but the principles are the same as in the simple example above. The translation from program code to binary is the very task of the program called a **compiler**. The compiler from arithmetic expressions to JVM byte code works as follows:

1. analyze the expression into an operator F and its operands X and Y
2. compile the code for X, followed by the code for Y, followed by the code for F

This procedure is our first example of a compiler. It shows the two main ideas of compilers, which we will repeat again and again in new configurations:

1. **Syntactic analysis**: here, to find the main operator of an expression
2. **Recursion**: the compiler calls the compiler on parts of the expression, until it reaches the simplest parts (here, the numeric constants).

1.2 Levels of languages

The task of a compiler may be more or less demanding. This depends on the distance of the languages it translates between. The situation is related to translation between human languages: it is easier to translate from English to French than from English to Japanese, because French is closer to English than Japanese is, both in the family tree of languages and because of cultural influences.

But the meaning of "closer" is clearer in the case of computer languages. Often it is directly related to the **level** of the language. The binary machine language is usually defined as the **lowest** level, whereas the highest level might be human language such as English. Usual programming languages are between these levels, as shown by the following very sketchy diagram:

```
---------------------------------------------- human
  human language

                Haskell

                           Lisp     Prolog

  Java

  C

  assembler

  machine language
---------------------------------------------- machine
```

Because of the distance, **high-level languages** are more difficult to compile than **low-level languages**. Notice that "high" and "low" don't imply any value judgements here; the idea is simply that higher levels are closer to human thought, whereas lower levels are closer to the operation of machines. Both humans and machines are needed to make computers work in the way we are used to. Some people might claim that only the lowest level of binary code is necessary, because humans can be trained to write it. But to this one can object that programmers could never write very sophisticated programs by using machine code only—they could just not keep the millions of bytes needed in their heads. Therefore, it is usually much more productive to write high-level code and let a compiler produce the binary.

The history of programming languages indeed shows a steady progress from lower to higher levels. Programmers can usually be more productive when writing in high-level languages, which means that high levels are desirable; at the same time, raising the level implies a challenge to compiler writers. Thus the evolution of programming languages goes hand in hand

with developments in compiler technology. It has of course also helped that the machines have become more powerful. Compilation can be a heavy computation task, and the computers of the 1960's could not have run the compilers of the 2010's. Moreover, it is harder to write compilers that produce efficient code than ones that waste some resources.

Here is a very rough list of programming languages in the history, only mentioning ones that have implied something new in terms of programming language expressivity:

- 1940's: connecting wires to represent 0's and 1's
- 1950's: assemblers, macro assemblers, FORTRAN, COBOL, LISP
- 1960's: ALGOL, BCPL (-> B -> C), SIMULA
- 1970's: Prolog, ML
- 1980's: C++, Perl, Python
- 1990's: Haskell, Java

1.3 Compilation and interpretation

In a way, a compiler reverts the history of programming languages. What we saw before goes from a "1960's" source language:

```
5 + 6 * 7
```

to a "1950's" assembly language

```
push 5 push 6 push 7 mul add
```

and further to a "1940's" machine language

```
0001 0000  0000 0101  0001 0000  0000 0110
0001 0000  0000 0111  0110 1000  0110 0000
```

The second step is very easy: you just look up the binary codes for each symbol in the assembly language and put them together in the same order. It is sometimes not regarded as a part of compilation proper, but as a separate level of **assembly**. The main reason for this is purely practical: modern compilers don't need to go all the way to the binary, but just to the assembly language, since there exist assembly programs that can do the rest.

A compiler is a program that **translates** code to some other code. It does not actually run the program. An **interpreter** does this. Thus a source language expression,

```
5 + 6 * 7
```

is by an interpreter turned to its value,

```
47
```

This computation can be performed without any translation of the source code into machine code. However, a common practice is in fact a *combination* of compilation and interpretation. For instance, Java programs are, as shown above, compiled into JVM code. This code is then in turn interpreted by a JVM interpreter.

The compilation of Java is different from for instance the way C is translated by GCC (GNU Compiler Collection). GCC compiles C into the native code of each machine, which is just executed, not interpreted. JVM code must be interpreted because it is not executable by any actual machine.

Sometimes a distinctin is made between "compiled languages" and "interpreted languages", C being compiled and Java being interpreted. This is really a misnomer, in two ways. First, *any* language could have both an interpreter and a compiler. Second, it's not Java that is interpreted by a "Java interpreter", but JVM, a completely different language to which Java is compiled.

Here are some examples of how some known languages are normally treated:

- C is usually compiled to machine code by GCC
- Java is usually compiled to JVM bytecode by Javac, and this bytecode is usually interpreted using JVM
- JavaScript is interpreted in web browsers

- Unix shell scripts are interpreted by the shell
- Haskell programs are either compiled using GHC, or interpreted (via bytecode) using Hugs or GHCI.

Compilation vs. interpretation is one of the important decisions to make when designing and implementing a new language. Here are some trade-offs:

Advantages of interpretation:

- faster to get going
- easier to implement
- portable to different machines

Advantages of compilation:

- if to machine code: the resulting code is faster to execute
- if to machine-independent target code: the resulting code easier to interpret than the source code

The advent of virtual machines with actual machine language instruction sets, such as VMWare, is blurring the distinction. In general, the best trade-offs are achieved by combinations of compiler and interpreter components, reusing as much as possible (as we saw is done in the reuse of the assembly phase). This leads us to the following topic: how compilers are divided into separate components.

1.4 Compilation phases

A compiler even for a simple language easily becomes a complex program, which is best attacked by dividing it to smaller components. These components typically address different **compilation phases**. Each phase is a part of a pipeline, which transforms the code from one format to another. These formats are typically encoded in different data structures: each phase returns a data structure that is easy for the next phase to manipulate.

The following diagram shows the main compiler phases and how a piece of source code travels through them. The code is on the left, the down-going arrows are annotated by the names of the phases, and the data structure is on the right.

```
57+6*result                          character string

   | lexer
   v

57 + 6 * result                      token string

   | parser
   v

(+ 57 (* 6 result))                  syntax tree

   | type checker
   v

([i+] 57 ([i*] 6 [i result]))        annotated syntax tree

   | code generator
   v

bipush 57                            instruction list
bipush 6
iload 8
imul
iadd
```

With some more explaining words,

- The **lexer** reads a string of **characters** an chops it into **tokens**, i.e. to "meaningful words"; the figure represents the token string by putting spaces between tokens.
- The **parser** reads a string of tokens and groups it into a **syntax tree**, i.e. to a structure indicating which parts belong together and how; the figure represents the syntax tree by using parentheses.
- The **type checker** finds out the **type** of each part of the syntax tree that might have alternative types, and returns an **annotated syntax tree**; the figure represents the annotations by the letter `i` ("integer") in square brackets.

- The **code generator** converts the annotated syntax tree into a list of target code instructions. The figure uses normal JVM assembly code, where `imul` means integer multiplication, `bipush` pushing integer bytes, and `iload` pushing values of integer variables.

We want to point out the role of type checking in particular. In a Java-to-JVM compiler it is an indispensable phase in order to perform **instruction selection** in the code generator. The JVM target language has different instructions for the addition of integers and floats, for instance (`iadd` vs. `dadd`), whereas the Java source language uses the same symbol `+` for both. The type checker analyses the code to find the types of the operands of `+` to decide whether integer or double addition is needed.

The difference between compilers and interpreters is just in the last phase: interpreters don't generate new code, but execute the old code. However, even then they will need to perform the earlier phases, which are independent of what the last step will be. This is with the exception of the type checker: compilers tend to require more type checking than interpreters, to enable instruction selection. It is no coincidence that untyped languages such as JavaScript and Python tend to be interpreted languages.

1.5 Compiler errors

Each compiler phases has both a positive and a negative side, so to say. The positive side is that it converts the code to something that is more useful for the next phase, e.g. the syntax tree into a type-annotated tree. The negative side is that it may fail, in which case it might report an error to the user.

Each compiler phase has its characteristic errors. Here are some examples:

- **Lexer errors**, e.g. unclosed quote,

```
"hello
```

- **Parse errors**, e.g. mismatched parentheses,

```
(4 * (y + 5) - 12))
```

- **Type errors**, e.g. the application of a function to an argument of wrong kind,

```
sort(45)
```

Errors on later phases are usually not supported. One reason is the principle (by Milner), that "well-typed programs cannot go wrong". This means that if a program passes the type checker it will also work on later phases. Another, more general reason is that the compiler phases can be divided into two groups:

- The **front end**, which performs **analysis**, i.e. inspects the program: lexer, parser, type checker.
- The **back end**, which performs **synthesis**: code generator.

It is natural that only the front end (analysis) phases look for errors.

A good compiler finds all errors at the earliest occasion. Thereby it saves work: it doesn't try to type check code that has parse errors. It is also more useful for the user, because it can then give error messages that go to the very root of the problem.

Of course, compilers cannot find all errors, for instance, bugs in the program. Errors such as array index out of bounds are another example of such errors. However, in general it is better to find errors **at compile time** than **at run time**, and this is one aspect in which compilers are constantly improving. One of the most important lessons of this book will be to understand what is possible to do at compile time and what must be postponed to run time.

A typical example is the **binding analysis** of variables: if a variable is used in an expression in Java or C, it must have been declared and given a value. For instance, the following function is incorrect in C:

```
int main () {
  printf("%d",x) ;
  }
```

The reason is that `x` has not been declared, which for instace GCC correctly reports as an error. But the following is correct in C:

```
int main () {
  int x ;
  printf("%d",x) ;
}
```

What is intuitively a problem, though, is that `x` has not been given a value. The corresponding function when compiled in Java would give this as an error.

As we will see in Chapter 3, binding analysis cannot be performed in a parser, but must be done in the type checker. However, the situation is worse than this. Consider the function

```
int main () {
  int x ;
  if (x!=0) x = 1 ;
  printf("%d",x) ;
}
```

Here `x` gets a value under a condition. It may be that this condition is impossible to decide at compile time. Hence it is not decidable at compile time if `x` has a value—neither in the parser, nor in the type checker.

1.6 More compiler phases

The compiler phases discussed above are the main phases. There can be many more—here are a couple of examples:

Desugaring/normalization: remove **syntactic sugar**, i.e. language constructs that are there to make the language more convenient for programmers, without adding to the expressive power. Such constructs can be removed early in compilation, so that the later phases don't need to deal with them. An example is multiple declarations, which can be reduced to sequences of single declarations:

```
int i, j ;   --->  int i ; int j ;
```

Desugaring is normally done at the syntax tree level, and it can be inserted as a phase between parsing and type checking. A disadvantage can be, however, that errors arising in type checking then refer to code that the programmer has never written herself, but that has been created by desugaring.

Optimizations: improve the code in some respect. This can be done on many different levels. For instance, **source code optimization** may precompute values known at compile time:

```
i = 2 + 2 ;            --->  i = 4 ;
```

Target code optimization may replace instructions with cheaper ones:

```
bipush 31 ; bipush 31  --->  bipush 31 ; dup
```

Here the second `bipush 31` is replaced by `dup`, which duplicates the top of the stack. The gain is that the `dup` instruction is just one byte, whereas `bipush 31` is two bytes.

Modern compilers may have dozens of phases. For instance, GCC has several optimization phases performed on the level of **intermediate code**. This code is neither the source nor the target code, but something in between. The advantage is that the optimization phases can be combined with different source and target languages, to make these components reusable.

1.7 Theory and practice

The complex task of compiler writing is greatly helped by the division into phases. Each phase is simple enough to be understood properly; and implementations of different phases can be recombined to new compilers. But there is yet another aspect: many of the phases have a clean mathematical **theory**, which applies to that phase. The following table summarizes those theories:

phase	**theory**
lexer	finite automata
parser	context-free grammars
type checker	type systems
interpreter	operational semantics
code generator	compilation schemes

The theories provide **declarative notations** for each of the phases, so that they can be specified in clean ways, independently of implementatiotion and usually much more concisely. They will also enable **reasoning** about the compiler components. For instance, the way parsers are written by means of context-free grammars can be used for guaranteeing that the language is *unambiguous*, that is, that each program can be compiled in a unique way.

Syntax-directed translation is a common name for the techniques used in type checkers, interpreters, and code generators alike. We will see that these techniques have so much in common that, once you learn how to implement a type checker, the other components are easy variants of this.

1.8 The scope of the techniques

The techniques of compiler construction are by no means restricted to the traditional task of translating programming language to machine language. The target of the translation can also be another programming language—for instance, the Google Web Toolkit is a compiler from Java into JavaScript, enabling the construction of web applications in a higher-level and type-checked language.

Actually, the modular way in which modern compilers are built implies that it is seldom necessary to go all the way to the machine code (or assembler), even if this is the target. A popular way of building native code compilers is via a translation to C. As soon as C code is reached, the compiler for the new language is complete.

The modularity of compilers also enables the use of compiler components to other tasks, such as debuggers, documentation systems, and code analysis of different kinds. But there is still a very good reason to learn the whole chain from source language to machine language: it will help you to decide which phases your task resembles the most, and thereby which techniques are the most useful ones to apply.

Chapter 2

What can a grammar do for you

This chapter is a hands-on introduction to BNFC. It explains step by step how to write a grammar, how to convert it into a lexer and a parser, how to test it, and how to solve the problems that are likely to arise.

This chapter also provides all the concepts and tools needed for solving Assignment 1. BNFC has some features that are not covered here, including the code generation for C and C++. The documentation on BNFC home page gives more information on these matters: the *BNFC tutorial*

> http://www.cse.chalmers.se/research/group/Language-technology/BNFC/doc/tutorial/bnfc-tutorial.html

covers practical issues on using BNFC, whereas the *LBNF Report*,

> http://www.cse.chalmers.se/research/group/Language-technology/BNFC/doc/LBNF-report.pdf

explains more features of the BNF notation used.

2.1 Defining a language

In school teaching, **grammars** are systems of rules used for teaching languages. They specify how words are formed (e.g. that the plural of the noun *baby* is *babies*) and how words are combined to sentences (e.g. that in English the subject usually appears before the verb). Grammars can be more

or less complete, and people who actually speak a language may follow the grammar more or less strictly. In **linguistics**, where grammars are studied in a scientific way, a widely held belief is that *all grammars leak*—that it is not possible to specify a language completely by grammar rules.

In compiler construction, grammars have a similar role: they give rules for forming "words", such as intereger constants, identifiers, and keywords. And they also give rules for combining words into expressions, statements, and programs. But the usefulness of grammars is much more uncontroversial than in linguistics: grammars of programming languages don't leak, because the languages are *defined* by their grammars. This is of course possible just because programming languages are artificial products, rather than results of natural evolution.

Defining a programming language is so easy that we can directly jump into doing it. Let us start with the following grammar. It defines a language of **expressions** built by using the four arithmetic operations (addition, subtraction, multiplication, division) as well as integer constants.

```
EAdd. Exp  ::= Exp  "+" Exp1 ;
ESub. Exp  ::= Exp  "-" Exp1 ;
EMul. Exp1 ::= Exp1 "*" Exp2 ;
EDiv. Exp1 ::= Exp1 "/" Exp2 ;
EInt. Exp2 ::= Integer ;

coercions Exp 2 ;
```

You can copy this code into a file called `Calc.cf`. It will be the source of your first compiler component, which is a parser of integer arithmetic expressions.

The code is written in the notation of BNFC, BNF Converter. It is a brand of the BNF notation, Backus Naur Form, named after the two inventors of this grammar format. BNF grammars are routinely used for the **specification** of programming languages, appearing in language manuals. The parser of a language must of course follow the grammar in order to be correct. If the grammar is written in the BNFC notation, such a correct parser can be automatically derived by the BNFC tool.

The code above should be easy to understand, at least with some experience. We will explain the details of the notation in a while. But we will first show how the code is used in BNFC.

2.2 Using BNFC

The first thing is you have to do is to check that the BNFC tool is available. Assuming you are working in a Unix-style shell, type

```
bnfc
```

and you should get a message specifying the authors and license of BNFC and its usage options. If the command `bnfc` does not work, you can install the software from the BNFC homepage,

```
http://www.cse.chalmers.se/research/group/Language-technology/BNFC/
```

It is available for Linux, Mac OS, and Windows, and there are several installation methods (such as Debian packages), from which you can choose the one most suitable for your platform and taste.

Now, assuming you have BNFC installed, you can run it on the file `Calc.cf` created in the previous section:

```
bnfc -m Calc.cf
```

The system will respond by generating a bunch of files:

```
writing file AbsCalc.hs
writing file LexCalc.x
 (Use Alex 2.0 to compile.)
writing file ParCalc.y
 (Tested with Happy 1.15)
writing file DocCalc.tex
writing file DocCalc.txt
writing file SkelCalc.hs
writing file PrintCalc.hs
writing file TestCalc.hs
writing file ErrM.hs
writing file Makefile
```

These files are different components of a compiler, which can be automatically generated from the BNF grammar. We will go to their details later, but you can certainly guess that there is a file for a lexer (`LexCalc.x`) and another one for a parser (`ParCalc.y`). The suffix `.hs` tells that some of the files are for the Haskell programming language. We shall see later how other languages can be used instead. For instance, you can write

```
bnfc -m -java1.5 Calc.cf
```

to generate the components for Java.

Running BNFC for Haskell

One of the generated files is a `Makefile`, which specifies the commands for compiling the compiler. So let us do this as the next thing:

```
make
```

Again, this can fail at some point if you don't have the Haskell tools installed: the GHC Haskell compiler, the Happy parser generator, and the Alex lexer generator. You don't need to install them, if you aim to work in Java and not in Haskell, but let us first assume you do have GHC, Happy, and Alex. Then your run of `make` will successfully terminate with the message

```
Linking TestCalc ...
```

`TestCalc` is an executable program for testing the parser defined by `Calc.cf`. So let us try it out:

```
echo "5 + 6 * 7" | ./TestCalc
```

Notice that `TestCalc` reads Unix standard input; the easiest thing to provide the parsable expression is thus by a pipe from the `echo` command. Now, the response of `TestCalc` is the following:

```
Parse Successful!

[Abstract Syntax]
EAdd (EInt 5) (EMul (EInt 6) (EInt 7))

[Linearized tree]
5 + 6 * 7
```

It first says that it has succeeded to parse the input, then shows an **abstract syntax tree**, which is the expected result of parsing and gives the tree structure of the expression. Finally, it displays the **linearizion**, which is the string obtained by using the grammar in the direction opposite to parsing. We will later see that this string can sometimes be different from the input string.

Input can also be read from a file. The standard input method for this is

```
./TestCalc < FILE_with_an_expression
```

But the test program also allows a file name argument,

```
./TestCalc FILE_with_an_expression
```

Running BNFC for Java

If you don't want to use Haskell but Java, you will have run BNFC with the Java option,

```
bnfc -m -java1.5 Calc.cf
```

You see that some more files are generated then:

```
writing file Calc/Absyn/Exp.java
writing file Calc/Absyn/EAdd.java
...
writing file Calc/PrettyPrinter.java
writing file Calc/VisitSkel.java
writing file Calc/ComposVisitor.java
writing file Calc/AbstractVisitor.java
writing file Calc/FoldVisitor.java
writing file Calc/AllVisitor.java
writing file Calc/Test.java
writing file Calc/Yylex (saving old file as Calc/Yylex.bak)
  (Tested with JLex 1.2.6.)
writing file Calc/Calc.cup
  (Parser created for category Exp)
  (Tested with CUP 0.10k)
writing file Calc.tex
writing file Makefile
```

There are no Haskell files any more, but files for Java, its parser tool Cup, and its lexer tool JLex. The `Makefile` works exactly like in the case of Haskell:

```
make
```

Well... if you have done exactly as shown above, you will probably fail with the message

```
java  JLex.Main Calc/Yylex
Exception in thread "main" java.lang.NoClassDefFoundError: JLex/Main
make: *** [Calc/Yylex.java] Error 1
```

This problem is typical in Java when using libraries put into unusual places, which often happens with user-installed libraries like Cup and JLex. Fortunately there is an easy solution: you just have to define the **classpath** that Java uses for finding libraries. On my Ubuntu Linux laptop, the following shell command does the job:

```
export CLASSPATH=.:/usr/local/java/Cup:/usr/local/java
```

Now I will get a better result with `make`. Then I can run the parser test in almost the same way as with the version compiled with Haskell:

```
echo "5 + 6 * 7" | java Calc/Test

Parse Succesful!

[Abstract Syntax]
(EAdd (EInt 5) (EMul (EInt 6) (EInt 7)))

[Linearized Tree]
5 + 6 * 7
```

We can conclude this section by two important conclusions about BNFC:

1. We can use a BNF grammar to generate several compiler components.
2. The components can be generated in different languages from the same BNF source.

2.3 Rules, categories, and trees

A BNFC source file is a sequence of **rules**, where most rules have the format

```
LABEL . VALUE_CATEGORY ::= PRODUCTION ;
```

The `LABEL` and `VALUE_CATEGORY` are **identifiers** (without quotes).

The `PRODUCTION` is a sequence of two kinds of items:

- identifiers, called **nonterminals**
- **string literals** (strings in double quotes), called **terminals**

The rule has the following semantics:

- a **tree** of type `VALUE_CATEGORY` can be built with `LABEL` as the topmost node, from any sequence specified by the production, so that whose nonterminals give the subtrees of this new tree

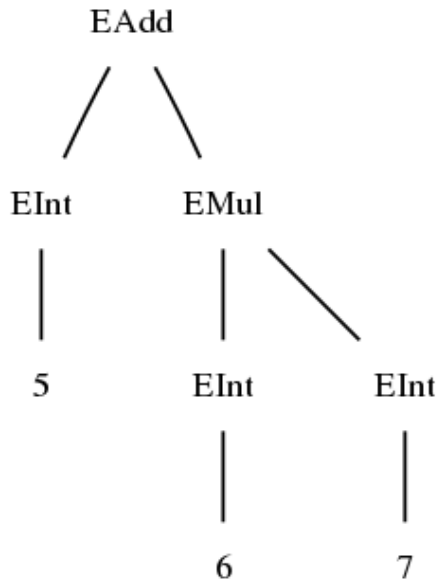
Types of trees are the **categories** of the grammar, i.e. the different kinds of objects that can be built (expressions, statements, programs,...). Tree labels are the **constructors** of those categories. The constructors appear as nodes of abstract syntax trees. Thus we saw above that the string

```
5 + 6 * 7
```

was compiled into a tree displayed as follows:

```
EAdd (EInt 5) (EMul (EInt 6) (EInt 7))
```

This is just a handy (machine-readable!) notation for the "real" tree



You may also notice that it is *exactly* the notation Haskell programmers use for specifying a certain kind of trees: expressions built by function applications.

2.4 Precedence levels

How does BNFC know that multiplication is performed before addition, that is, why the `EMul` node is below the `EAdd` node? Another way for analysing `5 + 6 * 7` could be

```
EMul (EAdd (EInt 5) (EInt 6)) (EInt 7)
```

The reason we don't give this analysis is that multiplication expressions have a **higher precedence**. In BNFC, **precedence levels** are the digits attached to category symbols. Thus `Exp1` has precedence level 1, `Exp2` has precedence level 2, etc. The nonterminal `Exp` without a digit is defined to mean the same as `Exp0`.

The rule

```
EAdd. Exp  ::= Exp  "+" Exp1 ;
```

can be read as saying:

> `EAdd` forms an expression of level 0 from an expression of level 0 on the left of `+` and of level 1 on the right.

Likewise, the rule

```
EMul. Exp1 ::= Exp1 "*" Exp2 ;
```

says: `EMul` form an expression of level 1 from an expression of level 1 on the left of `*` and of level 2 on the right.

The semantics of precedence levels consists of three principles:

1. All precedence variants of a nonterminal denote the same type in the abstract syntax. Thus `2`, `2 + 2`, and `2 * 2` are all of type `Exp`.

2. An expression of higher level can always be used on lower levels as well. This is what makes `2 + 3` correct: level 2 is here used on level 0 on the left and on level 1 on the right.

3. An expression of any level can be lifted to the highest level by putting it in parentheses. Thus `(5 + 6)` is an expression of level 2

What is the highest level? This is specified in the grammar by using a `coercions` statement. For instance, `coercions Exp 2` says that 2 is the highest level for `Exp`. It is actually a shorthand for the following "ordinary" BNF rules:

```
_. Exp0 ::= Exp1 ;
_. Exp1 ::= Exp2 ;
_. Exp2 ::= "(" Exp0 ")" ;
```

These rules are called **coercions**, since they just coerce expressions from one category to another, without doing anything—without creating new nodes in the abstract syntax tree. The underscore _ in front of these rules is a **dummy label**, which indicates that no constructor is added.

2.5 Abstract and concrete syntax

Abstract syntax trees are the hub of a modern compiler: they are the target of the parser and the place where most compilation phases happen, including type checking and code generation.

Abstract syntax is purely about the structure of expressions: what are their immediate parts and the parts of those parts? Abstract syntax thus ignore questions like what the parts look like, or even what order they appear in. From an abstract syntax point of view, all of the following expressions are the same:

```
2 + 3                          -- infix

(+ 2 3)                        -- prefix

(2 3 +)                        -- postfix

bipush 2                       -- JVM
bipush 3
iadd

the sum of 2 and 3             -- English

2:n ja 3:n summa               -- Finnish
```

In fact, the simplest way to build a compiler is the following:

1. Parse the source language expression, e.g. `2 + 3`
2. Obtain an abstract syntax tree, `EAdd (EInt 2) (EInt 3)`
3. Linearize the tree to another format, `bipush 2 bipush 3 iadd`

In practice, compilers don't quite work in this simple way. The main reason is that the tree obtained in parsing may have to be converted to another format before code generation. For instance, type annotations may have to be added to an arithmetic expression tree in order to select the proper JVM instructions.

The BNF grammar specifies the abstract syntax of a language. But it simultaneously specifies its **concrete syntax** as well. The concrete syntax gives more detail than the abstract syntax: it says what the parts look like and in what order they appear. One way to spell out the distinction is by trying to separate these aspects in a BNF rule. Take, for instance, the rule for addition expressions:

```
EAdd. Exp0 ::= Exp0 "+" Exp1
```

The purely abstract syntax part of this rule is

```
EAdd. Exp  ::= Exp      Exp
```

which hides the actual symbol used for addition (and thereby the place where it appears). It also hides the precedence levels, since they don't imply any differences in the abstract syntax trees.

In brief, the abstract syntax is extracted from a BNF grammar as follows:

1. Remove all terminals.
2. Remore all precedence numbers.
3. Remove all `coercions` rules.

If this is performed with the `Calc.cf` file, the following rules remain:

```
EAdd. Exp ::= Exp Exp ;
ESub. Exp ::= Exp Exp ;
EMul. Exp ::= Exp Exp ;
EDiv. Exp ::= Exp Exp ;
EInt. Exp ::= Int ;
```

This is a kind of a "skeleton" of a grammar, which could be filled with terminals, precedence numbers, and coercions in different ways to obtain new languages with the same abstract syntax. For instance, JVM assembler could be constructed as follows:

```
EAdd. Exp ::= Exp Exp "iadd" ;
ESub. Exp ::= Exp Exp "isub" ;
EMul. Exp ::= Exp Exp "imul" ;
EDiv. Exp ::= Exp Exp "idiv" ;
EInt. Exp ::= "bipush" Int ;
```

Now it is easy to see that arithmetic expressions could be compiled to JVM by just combining parsing with `Calc.cf` and linearization with this alternative grammar.

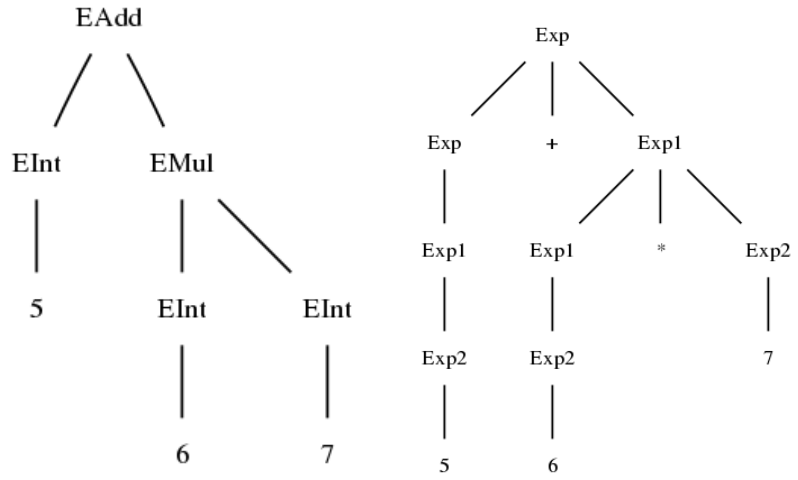
Another way to see the distinction between abstract and concrete syntax is by means of the different kinds of trees they involve. What we have called **abstract syntax trees** have a simple structure:

- their nodes and leaves are constructors (i.e. labels of BNF rules).

What is normally called **parse trees**, and we could call **concrete syntax trees**, look different:

- their nodes are category symbols (i.e. nonterminals)
- their leaves are tokens (i.e. terminals)

Here are the parse tree and the abstract syntax tree for the expression `5 + 6 * 7` as analysed by `Calc.cf`:



2.6 Abstract syntax in Haskell

The purpose of abstract syntax is to provide a suitable platform for further processing in a compiler. Concrete syntax details such as precedences and the shape of the terminals is then irrelevant: it would just complicate the matters, and also weaken the portability of compiler back-end components to other languages.

Haskell is the language with the most straightforward representations of abstract syntax, so let us start with it. Java will be covered in the next section. And we will return to the details of abstract syntax programming later, when discussing later compiler phases. We hope the Haskell code we show is readable by non-Haskellers as well—it is much simpler than the Java code for the same purpose.

When generating Haskell code, BNFC represents the abstract syntax by **algebraic datatypes**. For every category in the grammar, a `data` definition is thus produced. The `Exp` category of `Calc` generates the following definition:

```
data Exp =
   EAdd Exp Exp
 | ESub Exp Exp
 | EMul Exp Exp
 | EDiv Exp Exp
 | EInt Integer
```

```
  deriving (Eq,Ord,Show)
```

The `deriving` part says that the type `Exp` has equality and order predicates, and its objects can be shown as strings, but Java programmers can ignore these details.

The main programming method in most components of the compiler is **syntax-directed translation**, i.e. **structural recursion on abstract syntax trees**. In Haskell, this is performed conveniently by using **pattern matching**. The following code is, in fact, the complete implementation of a calculator, which interprets arithmetic expressions as integer values:

```
module Interpreter where

import AbsCalc

interpret :: Exp -> Integer
interpret x = case x of
  EAdd exp0 exp  -> interpret exp0 + interpret exp
  ESub exp0 exp  -> interpret exp0 - interpret exp
  EMul exp0 exp  -> interpret exp0 * interpret exp
  EDiv exp0 exp  -> interpret exp0 `div` interpret exp
  EInt n  -> n
```

Thus we can now turn our parser into an interpreter! We do this by modifying the generated file `TestCalc.hs`: instead showing the syntax tree, we let it show the value from interpretation:

```
module Main where

import LexCalc
import ParCalc
import AbsCalc
import Interpreter
import ErrM

main = do
  interact calc
  putStrLn ""
```

```
calc s =
  let Ok e = pExp (myLexer s)
  in show (interpret e)
```

This, in a nutshell, is how you can build any compiler on top of BNFC:

1. Write a grammar and convert it into parser and other modules.
2. Write some code that manipulates syntax trees in a desired way.
3. Let the main file show the results of syntax tree manipulation.

If your `Main` module is in a file named `Calculator`, you can compile it with GHC as follows:

```
ghc --make Calculator.hs
```

Then you can run it on command-line input:

```
echo "1 + 2 * 3" | ./Calculator
7
```

Now, let's do the same thing in Java.

2.7 Abstract syntax in Java

Java has no notation for algebraic datatypes. But just types can be encoded by using the class system of Java:

- For each category in the grammar, an abstract base class.
- For each constructor of the category, a class extending the base class.

This means quite a few files, which are for the sake of clarity put to a separate category. In the case of `Calc.cf`, we have the files

```
Calc/Absyn/EAdd.java
Calc/Absyn/EDiv.java
Calc/Absyn/EInt.java
Calc/Absyn/EMul.java
Calc/Absyn/ESub.java
Calc/Absyn/Exp.java
```

This is what the classes look like:

```
public abstract class Exp implements java.io.Serializable {

// some code that we explain later
}

public class EAdd extends Exp {
  public final Exp exp_1, exp_2;
  public EAdd(Exp p1, Exp p2) { exp_1 = p1; exp_2 = p2; }

// some more code that we explain later
}

/* the same for ESub, EMul, EDiv */

public class EInt extends Exp {
  public final Integer integer_;
  public EInt(Integer p1) { integer_ = p1; }
}
```

Tree processing is much less staightforward than in Haskell, because Java doesn't have pattern matching. There are two ways of doing it:

1. Add an interpreter methods to each class.
2. Use a separate **visitor interface** to implement tree traversal in a general way.

The visitor method is the proper choice in more advanced applications, and we will return to it in the chapter on type checking. To get the calculator up and running now, let us follow the simpler way. What we do is take the classes generated in `Calc/Absyn/` by BNFC, and add an `interpret` method to each of them:

```
public abstract class Exp {
  public abstract Integer interpret() ;
}

public class EAdd extends Exp {
  public final Exp exp_1, exp_2;
  public Integer interpret() {return exp_1.interpret() + exp_2.interpret() ;
}

public class EMul extends Exp {
  public final Exp exp_1, exp_2;
  public Integer interpret() {return exp_1.interpret() * exp_2.interpret() ;
}

public class EInt extends Exp {
  public final Integer integer_;
  public Integer interpret() {return integer_ ;}
}
```

(We have hidden most of the other contents of the classes for simplicity.)

Now we can modify the file `Calc/Test.java` into a calculator:

```
public class Calculator {
  public static void main(String args[]) throws Exception
  {
    Yylex l  = new Yylex(System.in) ;
    parser p = new parser(l) ;
    Calc.Absyn.Exp parse_tree = p.pExp() ;
    System.out.println(parse_tree.interpret());
  }
}
```

(We have omitted error handling for simplicity.) If we compile this,

```
javac Calc/Calculator.java
```

we can run it on command-line input as expected:

```
echo "1 + 2 * 3" | java Calc/Calculator
7
```

Run on file input:

```
java Calc/Interpret < ex1.calc
9102
```

2.8 List categories

Lists of various kinds are used everywhere in grammars. Standard BNF defines list categories with pairs of rules. For instance, to define a list of function declarations, one can write

```
NilFunction.  ListFunction ::= ;
ConsFunction. ListFunction ::= Function ListFunction ;
```

The first rule states that a list of functions can be empty ("nil"). The second rule states that a list can be formed by prepending a function to a list ("cons").

Lists often have **terminators**, i.e. tokens that appear after every item of a list. For instance, function declarations might have semicolons (`;`) as terminators. This is expressed as follows:

```
NilFunction.  ListFunction ::= ;
ConsFunction. ListFunction ::= Function ";" ListFunction ;
```

The pattern of list rules is so common that BNFC has some special notations for it. Thus lists of a category C can be denoted as `[`C`]`. Instead of pairs of rules, one can use the shorthand `terminator`. Thus the latter pair of rules for lists of functions can be written concisely

```
terminator Function ";" ;
```

The former pair, where no terminator is used, is written with an "empty terminator",

```
terminator Function "" ;
```

It is important to know that the `terminator` rule expands to a pair of ordinary BNF rules when BNFC is run. Thus there is nothing special with it, and it could be avoided. But it is easier for both the grammar writer and reader to use the concise `terminator` format. This is analogous with the `coercions` shorthand, which BNFC expands to a list of rules relating the precedence levels.

In addition to terminators, programming languages can have **separators**: tokens that appear between list items, just not after the last one. For instance, the list of arguments of a function in C have a comma (`,`) as separator. This is written by a rule for expression lists:

```
separator Exp "," ;
```

This shorthand expands to a set of rules for the category `[Exp]`. The rule for function calls can then be written

```
ECall. Exp ::= Ident "(" [Exp] ")" ;
```

Sometimes lists are required to be **nonempty**, i.e. have at least one element. This is expressed in BNFC by adding the keyword `nonempty`:

```
terminator nonempty Function "" ;
separator nonempty Ident "," ;
```

The bracket notation for lists is borrowed from Haskell. And the abstract syntax build by the parser indeed represents lists as Haskell lists. In Java, list categories are similarty represented as linked lists. To summarize,

- in Haskell, `[C]` is represented as `[C]`
- in Java, `[C]` is represented as `java.util.LinkedList<C>`

2.9 Specifying the lexer

We have defined lexing and parsing as the first two compiler phases. BNF grammars are traditionally used for specifying parsers. So where do we get the lexer from?

In BNFC, the lexer is often implicit. It is based on the use of **predefined token types**, which can be used in BNF rules but only on the right-hand-sides of the rules. Thus it is not legal to write rules that produce these types. There are five such types in BNFC:

- `Integer`, **integer literals**: sequence of digits, e.g. `123445425425436`;
- `Double`, **floating point literals**: two sequences of digits with a decimal point in between, and an optional exponent, e.g. `7.098e-7`;
- `String`, **string literals**: any characters between double quotes, e.g. `"hello world"`, with a backslash (\) escaping a quote and a backslash;
- `Char`, **character literals**: any character between single quotes, e.g. `'7'`;
- `Ident`, **identifiers**: a letter (`A..Za..z`) followed by letters, digits, and characters `_'`, e.g. `r2_out'`

The precise definitions of these types are given in the LBNF report.

The predefined token types are often sufficient for language implementations, especially for new languages, which can be designed to follow the BNFC rules for convenience. But BNFC also allows the definition of new token types. These definitions are written by using **regular expressions**. For instance, one can define a special type of upper-case identifiers as follows:

```
token UIdent (upper (letter | digit | '_')*) ;
```

This defines `UIdent` as a **token type**, which contains strings starting with an upper-case letter and continuing with a (possibly empty) sequence of letters, digits, and underscores.

The following table gives the main regular expressions available in BNFC.

name	**notation**	**explanation**
symbol	'a'	the character `a`
sequence	A B	A followed by B
union	A | B	A or B
closure	A`*`	any number of A's (possibly 0)
empty	`eps`	the empty string
character	`char`	any character (ASCII 0..255)
letter	`letter`	any letter (`A..Za..z`)
upper-case letter	`upper`	any upper-case letter (`A..Z`)
lower-case letter	`lower`	any lower-case letter (`a..z`)
digit	`digit`	any digit (`0..9`)
option	A`?`	optional A
difference	A `-` B	A which is not B

We will return to the semantics and implementation of regular expressions in next chapter.

BNFC can be forced to remember the **position of a token**. This is useful when, for instance, error messages at later compilation phases make reference to the source code. The notation for token types remembering the position is

```
position token CIdent (letter | (letter | digit | '_')*) ;
```

When BNFC is run, bare `token` types are encoded as types of strings. For instance, the standard `Ident` type is in Haskell represented as

```
newtype Ident = Ident String
```

Position token types add to this a pair of integers indicating the line and the column in the input:

```
newtype CIdent = Ident (String, (Int,Int))
```

In addition to tokens, **comments** are language features treated in the lexer. They are parts of source code that the compiler ignores. BNFC permits the definition of two kinds of comments:

- one-line comments, which run from a start token till the end of the line
- multiple-line comments, which run from a start token till a closing token

This is, for instance, how comments of C are defined:

```
comment "//" ;
comment "/*" "*/" ;
```

Thus one-line comments need one token, the start token, whereas multiple-line comments need the start and the closing token.

Since comments are resolved by the lexer, they are processed by using a finite automaton. Therefore nested comments are not possible. A more thorough explanation of this will be given in next chapter.

2.10 Working out a grammar

We conclude this section by working out a grammar for a small C-like programming language. This language is the same as targeted by the Assignments 2 to 4 at the end of this book. Assignment 1 targets a larger language, for which this smaller language is a good starting point. The discussion below goes through the language constructs top-down, i.e. from the largest to the smallest, and builds the appropriate rules at each stage.

- *A program is a sequence of definitions.*

This suggests the following BNFC rules:

```
PDefs.   Program ::= [Def] ;

terminator Def "" ;
```

- *A program may contain comments, which are ignored by the parser.*

This means C-like comments, specified as follows:

```
comment "//" ;
comment "/*" "*/" ;
```

- *A function definition has a type, a name, an argument list, and a body. An argument list is a comma-separated list of argument declarations enclosed in parentheses* `(` *and* `)`*. A function body is a list of statements enclosed in curly brackets* { *and* } *. For example:*

  ```
  int foo(double x, int y)
  {
    return y + 9 ;
  }
  ```

We decide to specify all parts of a function definition in one rule, in addition to which we specify the form of argument and statement lists:

```
DFun.      Def    ::= Type Id "(" [Arg] ")" "{" [Stm] "}" ;
separator  Arg "," ;
terminator Stm "" ;
```

- *An argument declaration has a type and an identifier.*

```
ADecl.   Arg    ::= Type Id ;
```

- *Any expression followed by a semicolon* `;` *can be used as a statement.*

```
SExp.    Stm    ::= Exp ";" ;
```

- *Any declaration followed by a semicolon* `;` *can be used as a statement. Declarations have one of the following formats:*
  - *a type and one variable (as in function parameter lists),*
    ```
    int i ;
    ```
  - *a type and many variables,*
    ```
    int i, j ;
    ```
  - *a type and one initialized variable,*
    ```
    int i = 6 ;
    ```

Now, we could reuse the function argument declarations `Arg` as one kind of statements. But we choose the simpler solution of restating the rule for one-variable declarations.

```
SDecl.   Stm    ::= Type Id ";" ;
SDecls.  Stm    ::= Type Id "," [Id] ";" ;
SInit.   Stm    ::= Type Id "=" Exp ";" ;
```

- *Statements also include*
  - *Statements returning an expression,*
    ```
    return i + 9 ;
    ```
  - *While loops, with an expression in parentheses followed by a statement,*
    ```
    while (i < 10) ++i ;
    ```
  - *Conditionals:* `if` *with an expression in parentheses followed by a statement,* `else`*, and another statement,*
    ```
    if (x > 0) return x ; else return y ;
    ```
  - *Blocks: any list of statements (including the empty list) between curly brackets. For instance,*

```
{
  int i = 2 ;
  {
  }
  i++ ;
}
```

The statement specifications give rise to the following BNF rules:

```
SReturn. Stm    ::= "return" Exp ";" ;
SWhile.  Stm    ::= "while" "(" Exp ")" Stm ;
SBlock.  Stm    ::= "{" [Stm] "}" ;
SIfElse. Stm    ::= "if" "(" Exp ")" Stm "else" Stm ;
```

- *Expressions are specified with the following table that gives their precedence levels. Infix operators are assumed to be left-associative. The arguments in a function call can be expressions of any level. Otherwise, the subexpressions are assumed to be one precedence level above the main expression.*

level	**expression forms**	**explanation**
16	literal	literal (integer, float, boolean)
16	identifier	variable
15	`f(e,...,e)`	function call
14	`v++`, `v--`	in/decrement
13	`++v`, `--v`	in/decrement
12	`e*e`, `e/e`	multiplication, division
11	`e+e`, `e-e`	addition, subtraction
9	`e<e`, `e>e`, `e>=e`, `e<=e`	comparison
8	`e==e`, `e!=e`	(in)equality
4	`e&&e`	conjunction
3	`e||e`	disjunction
2	`v=e`	assignment

The table is straightforward to translate to a set of BNFC rules. On the level of literals, integers and floats ("doubles") are provided by BNFC, whereas the boolean literals `true` and `false` are defined by special rules.

```
EInt.    Exp15  ::= Integer ;
EDouble. Exp15  ::= Double ;
ETrue.   Exp15  ::= "true" ;
EFalse.  Exp15  ::= "false" ;
EId.     Exp15  ::= Id ;

EApp.    Exp15  ::= Id "(" [Exp] ")" ;

EPIncr.  Exp14  ::= Exp15 "++" ;
EPDecr.  Exp14  ::= Exp15 "--" ;

EIncr.   Exp13  ::= "++" Exp14 ;
EDecr.   Exp13  ::= "--" Exp14 ;

ETimes.  Exp12  ::= Exp12 "*"  Exp13 ;
EDiv.    Exp12  ::= Exp12 "/"  Exp13 ;
EPlus.   Exp11  ::= Exp11 "+"  Exp12 ;
EMinus.  Exp11  ::= Exp11 "-"  Exp12 ;
ELt.     Exp9   ::= Exp9  "<"  Exp10 ;
EGt.     Exp9   ::= Exp9  ">"  Exp10 ;
ELtEq.   Exp9   ::= Exp9  "<=" Exp10 ;
EGtWq.   Exp9   ::= Exp9  ">=" Exp10 ;
EEq.     Exp8   ::= Exp8  "==" Exp9 ;
ENEq.    Exp8   ::= Exp8  "!=" Exp9 ;
EAnd.    Exp4   ::= Exp4  "&&" Exp5 ;
EOr.     Exp3   ::= Exp3  "||" Exp4 ;
EAss.    Exp2   ::= Exp3 "=" Exp2 ;
```

Finally, we need a `coercions` rule to specify the highest precedence level, and a rule to form function argument lists.

```
coercions Exp 15 ;
separator Exp "," ;
```

- *The available types are* `bool`, `double`, `int`, and `void`.

```
Tbool.   Type ::= "bool" ;
Tdouble. Type ::= "double" ;
Tint.    Type ::= "int" ;
Tvoid.   Type ::= "void" ;
```

- *An identifier is a letter followed by a list of letters, digits, and underscores.*

Here we cannot use the built-in `Ident` type of BNFC, because apostrophes (`'`) are not permitted! But we can define our identifiers easily by a regular expression:

```
token Id (letter (letter | digit | '_')*) ;
```

The reader is advised to copy all the rules of this section into a file and try this out in BNFC, with various programs as input.

Chapter 3

How do lexers and parsers work*

This is an optional theory chapter, which gives deeper understanding of the things worked through in the previous chapter. It explains the concepts of regular expressions and finite automata, context-free grammars and parsing algorithms, and the limits of each of these methods. For instance, we will show why automata may explode in size, why parentheses cannot be matched by finite automata, and why context-free grammars cannot alone specify the well-formedness of programs. It will also show how the usual parsing algorithms work, to explain what **conflicts** are and how to avoid them.

3.1 The theory of formal languages

BNFC saves a lot of work in a compiler by generating the code needed for the lexer and the parser. The exact saving is by an order of magnitude, compared with hand-written code. The code generated by BNFC is taken care of by other tools, which in turn generate code in some host language—Haskell, Java, or C. Let's call the lexer tool (Alex, JLex, Flex) just **Lex** and the parser tool (Happy, Cup, Bison) just **Yacc**, by references to the first such tools created for C in the 1970's. These tools stand for another order of magnitude of saving, compared to writing host language code by hand.

The generation of Lex and Yacc from a BNFC file is rather straightforward. The next step is much more involved. In a nutshell,

- **Lex** code is **regular expressions**, converted to **finite automata**.

- **Yacc** code is **context-free grammars**, converted to **LALR(1) parsers**.

Regular expressions and context-free grammars, as well as their compilation to automata and parsers, originate in the mathematical theory of **formal languages**. A formal language is, mathematically, just any set of **sequences of symbols**. Programming languages are examples of formal languages. They are rather complex in comparison to the examples usually studied in the theory; but the good news is that their complexity is mostly due to repetitions of simple well-known patterns.

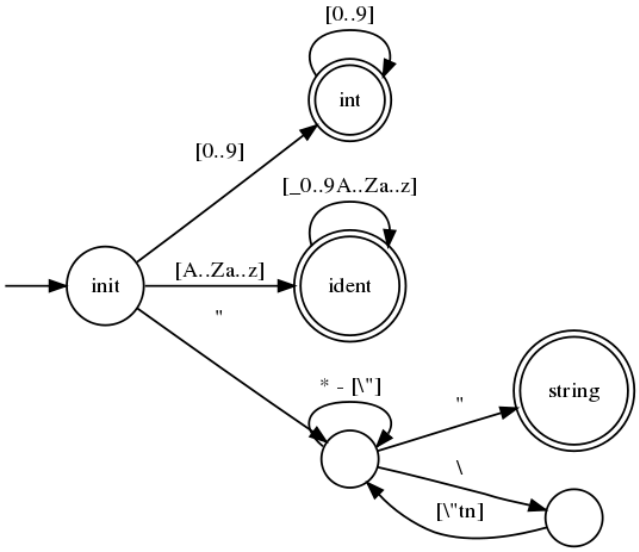
3.2 Regular languages and finite automata

A **regular language** is, like any formal language, a set of **strings**, i.e. sequences of **symbols**, from a finite set of symbols called the **alphabet**. Only some formal languages are regular; in fact, regular languages are exactly those that can be defined by **regular expressions**, which we already saw in Section 2.9. We don't even need all the expressions, but just five of them; the other ones are convenient shorthands. They are showed in the following table, together with the corresponding regular language in set-theoretic notation:

expression	**language**
'a'	$\{\texttt{a}\}$
AB	$\{ab \vert a \in [\![A]\!], b \in [\![B]\!]\}$
$A \mid B$	$[\![A]\!] \cup [\![B]\!]$
$A\texttt{*}$	$\{a_1 a_2 \ldots \vert i = 0, 1, \ldots, a_i \in [\![A]\!]\}$
`eps`	$\{\epsilon\}$ (empty string)

The table uses the notation $[\![A]\!]$ for the set **denoted** by the expression A. This notation is common in computer science to specify the **semantics** of a language formally.

When does a string belong to a regular language? A straightforward answer would be to write a program that *interprets* the sets, e.g. in Haskell by using list comprehensions instead of the set brackets. This implementation, however, would be very inefficient. The usual way to go is to *compile* regular expressions to **finite automata**. Finite automata are graphs that allow traversing their input strings symbol by symbol. For example, the following automaton recognizes a string that is either an integer literal or an identifier or a string literal.



The corresponding regular expression is, using the shorthands of 2.9,

```
  digit digit*
| letter ('_' | letter | digit)*
| '"' (char-[\"] | '\'[\"tn])* '"'
```

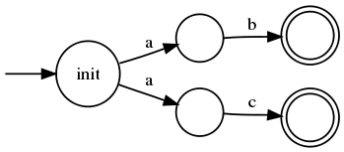
where we have also used the expressions `[abc]` denoting sets of characters and `[a..b]` denoting ranges of characters.

The automaton can be used for the **recognition of strings**. In this case, a recognized string is either a decimal integer, an indentifier, or a string literal. The recognition starts from the **initial state**, that is, the node marked "init". It goes to the next **state** depending on the first character. If it is a digit `0`...`9`, the state is the one marked "int". With more digits, the recognition loops back to this state. The state is marked with a double circle, which means it is a **final state**, also known as an **accepting state**. The other accepting states are "ident" and "string". But on the way to "string", there are non-accepting states: the one before a second quote is read, and the one after an escape is read.

The automaton above is **deterministic**, which means that at any state, any input symbol has at most one **transition**, that is, at most one way to go to a next state. If a symbol with no transition is encountered, the string is not accepted. For instance, `a&b` would not be an accepted string in the above automaton; nor is it covered by the regular expression.

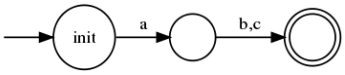
An automaton can also be **nondeterministic**, which means that some symbols may have many transitions. An example is the following automaton, with the corresponding regular expression that recognizes the language {`ab`, `ac`}:

```
a b | a c
```



Now, this automaton and indeed the expression might look like a stupid thing to write anyway: wouldn't it be much smarter to factor out the `a` and write simply as follows?

```
a (b | c)
```



The answer is *no*, both surprisingly and in a way typical to compiler construction. The reason is that one should not try to optimize automata by hand—one should let a compiler do that automatically and much more reliably! Generating a non-deterministic automaton is the standard first step of compiling regular expressions. After that, deterministic and, indeed, minimal automata can be obtained as optimizations.

Just to give an idea of how tedious it can be to create deterministic automata by hand, think about compiling an English dictionary into an automaton. It may start as follows:

```
a able about account acid across act addition adjustment
```

It would be a real pain to write a bracketed expression in the style of `a (c | b)`, and much nicer to just put |'s between the words and let the compiler do the rest!

3.3 The compilation of regular expressions

The standard compilation of regular expressions has the following steps:

1. **NFA generation**: convert the expression into a non-deterministic automaton, **NFA**.

2. **Determination**: convert the NFA into a deterministic automaton, **DFA**.

3. **Minimization**: minimize the size of the deterministic automaton.

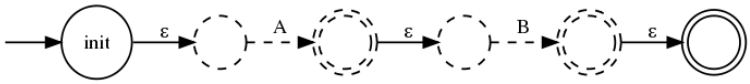
As usual in compilers, each of these phases is simple in itself, but trying to do them all at once would be too complicated.

Step 1. NFA generation

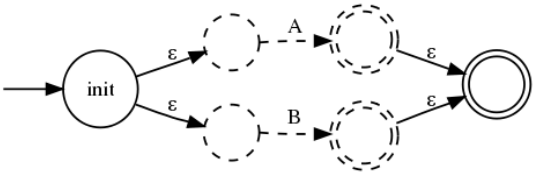
Assuming we have an expression written by just the five basic operators, we can build an NFA which has exactly one initial state and exactly one final state. The "exactly one" condition is crucial to make it easy to combine the automata for sequences, unions, and closures. The easiest way to guarantee this is to use **epsilon transitions**, that is, transitions that consume no input. They are marked with the symbol ϵ in the graphs. They of course increase nondeterminism, but can be eliminated later.

- **Symbol**. The expression `a` is compiled to

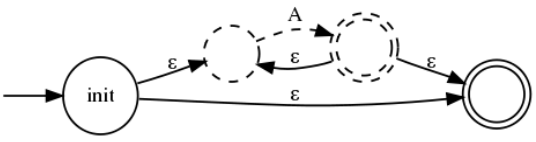
- **Sequence**. The expression $A\ B$ is compiled by combining the automata for A and B (drawn with dashed figures) as follows:



- **Union**. The expression $A \mid B$ is compiled as follows:

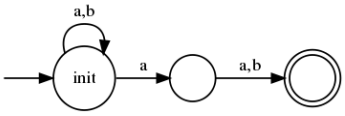


- **Closure**. The expression $A*$ is compiled as follows:



- **Empty**. The expression `eps` is compiled to

NFA generation is an example of **syntax-directed translation**, and could be recommended as an extra assignment for everyone! What is needed is a parser and abstract syntax for regular expressions (by BNFC), and a suitable code representation for automata. From this representation, one could also generate visualizations using e.g. the Graphviz software (as we did when preparing this book). Here is an example of an automaton and its Graphviz code:




```
digraph {
  rankdir = LR ;
  start [label = "",     shape = "plaintext"]
  init  [label = "init", shape = "circle"] ;
  a     [label = "",     shape = "circle"] ;
  end   [label = "",     shape = "doublecircle"] ;
  start -> init ;
  init  -> init [label = "a,b"] ;
  init  -> a    [label = "a"] ;
  a     -> end  [label = "a,b"] ;
}
```

The intermediate abstract representation should encode the mathematical definition of automata:

> *Definition.* A **finite automaton** is a 5-tuple $< \Sigma, S, F, i, t >$ where
>
> - Σ is a finite set of symbols (the **alphabet**)
> - S is a finite set of **states**
> - $F \subset S$ (the **final states**)
> - $i \in S$ (the **initial state**)

- $t : S \times \Sigma \to \mathcal{P}(S)$ (the **transition function**)

An automaton is **deterministic**, if $t(s, a)$ is a singleton for all $s \in S, a \in \Sigma$. Otherwise, it is **nondeterministic**, and then moreover the transition function is generalized to $t : S \times \Sigma \cup \{\epsilon\} -> \mathcal{P}(S)$ (with **epsilon transitions**).

Step 2. Determination

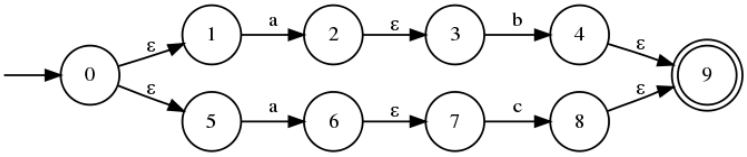
One of the most powerful and amazing properties of finite automata is that they can always be made deterministic by a fairly simple procedure. The procedure is called the **subset construction**. In brief: for every state $s$and symbol a in the automaton, form a new state $\sigma(s, a)$ that "gathers" all those states to which there is a transition from s by a. More precisely:

- $\sigma(s, a)$ is the set of those states s_i to which one can arrive from s by consuming just the symbol a. This includes of course the states to which the path contains epsilon transitions.
- The transitions from $\sigma(s, a) = \{s_1, \ldots, s_n\}$ for a symbol b are all the transitions with b from any s_i. (When this is specified, the subset construction must of course be iterated to build $\sigma(\sigma(s, a), b)$.)
- The state $\sigma(s, a) = \{s_1, \ldots, s_n\}$ is final if any of s_i is final.

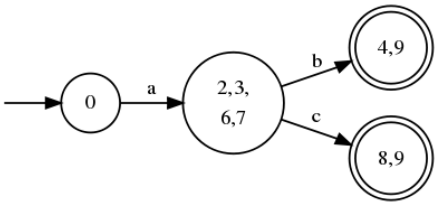
Let us give a complete example. Starting with the "awful" expression

```
a b | a c
```

the NFA generation of Step 1 creates the monstruous automaton



From this, the subset construction gives



How does this come out? First we look at the possible transitions with the symbol `a` from state 0. Because of epsilon transitions, there are no less than four possible states, which we collect to the state named {2,3,6,7}. From this state, `b` can lead to 4 and 9, because there is a `b`-transition from 3 to 4 and an epsilon transition from 4 to 9. Similarly, `c` can lead to 8 and 9.

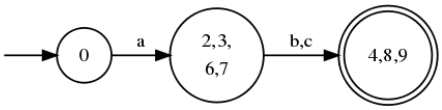
The resulting automaton is deterministic but not yet minimal. Therefore we perform one more optimization.

Step 3. Minimization

Determination may left the automaton with superfluous states. This means that there are states without any **distinguishing strings**. A distinguishing string for states s and u is a sequence x of symbols that ends up in an accepting state when starting from s and in a non-accepting state when starting from u.

For example, in the previous deterministic automaton, the states 0 and {2,3,6,7} are distinguished by the string `ab`. When starting from 0, it leads to the final state {4,9}. When starting from {2,3,6,7}, there are no transitions marked for `a`, which means that any string starting with `a` ends up in a **dead state** which is non-accepting.

But the states {4,9} and {8,9} are not distinguished by any string. The only string that ends to a final state is the empty string, from both of them. The minimization can thus merge these states, and we get the final, optimized automaton



The algorithm for minimization is a bit more complicated than for determination. We omit it here.

3.4 Properties of regular languages

There is a whole branch of discrete mathematics dealing with regular languages and finite automata. A part of the research has to do with **closure properties**. For instance, regular languages are closed under **complement**, i.e. if L is a regular language, then also $\Sigma^* - L$, is one; Σ^* is the set of all strings over the alphabet, also called the **universal language**.

We said that the five operators compiled in the previous section were sufficient to define all regular languages. Other operators can be defined in terms of them; for instance, the non-empty closure A^+ is simply AA^*. The negation operator $-A$ is more complicated to define; in fact, the simplest way to see that it exists is to recall that regular languages are closed under negation.

But how do we *know* that regular languages are closed under negation? The simplest way to do this is to construct an automaton: assume that we have a DFA corresponding to A. Then the automaton for $-A$ is obtained by inverting the status of each accepting state to non-accepting and vice-versa!

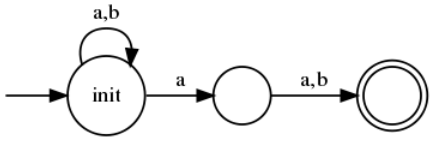
The reasoning above relies on the **correspondence theorem** saying that the following three are equivalent, convertible to each other: regular languages, regular expressions, finite automata. The determination algorithm moreover proves that there is always a *deterministic* automaton. The closure property for regular languages and expressions follows.

Another interesting property is inherent in the subset construction: the size of a DFA can be exponential in the size of the NFA (and therefore of the expression). The subset construction shows a potential for this, because there could in principle be a different state in the DFA for *every* subset of the NFA, and the number of subsets of an n-element set is 2^n.

A concrete example of the **size explosion of automata** is a language of strings of a's and b's, where the nth element *from the end* is an a. Consider this in the case n=2. The regular expression is

```
(a|b)* a (a|b)
```

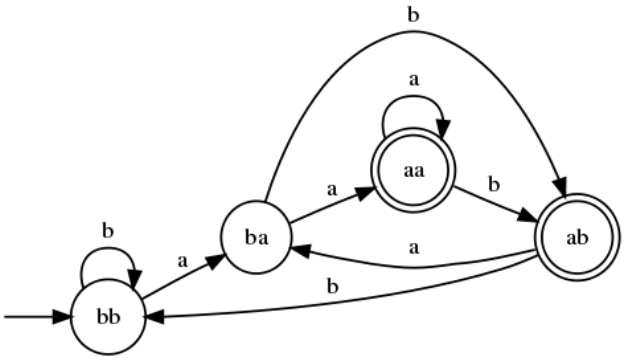
There is also a simple NFA:



But how on earth can we make this deterministic? How can we know, when reading a string, that *this* a is the second-last element and we should stop looping?

It is possible to solve this problem by the subset construction, which is left to an exercise. But there is also an elegant direct construction, which I learned from a student many years ago. The idea is that the *state* must "remember" the last two symbols that have been read. Thus the states can

be named *aa*, *ab*, *ba*, and *bb*. The states *aa* and *ab* are accepting, because they have *a* as the second-last symbol; the other two are not accepting. Now, for any more symbols encountered, one can "forget" the previous second-last symbol and go to the next state accordingly. For instance, if you are in *ab*, then *a* leads to *ba* and *b* leads to *bb*. The complete automaton is below:



Notice that the initial state is *bb*, because a string must have at least two symbols in order to be accepted.

With a similar reasoning, it is easy to see that a DFA for *a* as third-last symbol must have at least 8 states, for fourth-last 16, and so on. Unfortunately, the exponential blow-up of automata is not only a theoretical construct, but often happens in practice and can come as a surprise to those who build lexers by using regular expressions.

The third property of finite-state automata we want to address is, well, their finiteness. Remember from the definition that an automaton has a *finite* set of states. This fact can be used for proving that an automaton cannot match parentheses, i.e. guarantee that a string as as many left and right parentheses.

The argument uses, as customary in formal language theory, *a*'s and *b*'s to stand for left and right parentheses, respectively. The language we want to define is

$$\{a^n b^n | n = 0, 1, 2 \ldots\}$$

Now assume that the automaton is in state s_n after having read n *a*'s and starting to read *b*'s. Assume $s_m = s_n$ for some $m \neq n$. From this it follows that the automaton recognizes an expression $a^n b^m$, which is not in the language!

Now, matching parentheses is usually treated in parsers that use BNF grammars; for the language in question, we can easily write the grammar

```
S ::= ;
S ::= "a" S "b" ;
```

and process it in parser tools. But there is a related construct that one might want to try to treat in a lexer: **nested comments**. The case in point is code of the form

```
a /* b /* c */ d */ e
```

One might expect the code after the removal of comments to be

```
a                   e
```

But actually it is, at least with standard compilers,

```
a               d */ e
```

The reason is that the lexer is implemented by using a finite automaton, which cannot count the number of matching parentheses—in this case comment delimiters.

3.5 Context-free grammars and parsing

A **context-free grammar** is the same as a **BNF grammar**, consisting of rules of the form

$$C ::= t_1 \dots t_n$$

where each t_i is a terminal or a nonterminal. We used this format extensively in Chapter 2 together with labels for building abstract syntax trees. But for most discussion of parsing properties we can ignore the labels.

All regular languages can be also defined by context-free grammars. The inverse does not hold, as proved by matching parentheses. The extra expressive power comes with a price: context-free parsing can be more complex than recognition with automata. It is easy to see that recognition with a finite automaton is *linear* in the length of the string. But for context-free grammars the worst-case complexity is *cubic*. However, programming languages are usually designed in such a way that their parsing is linear. This means that they use a restricted subset of context-free grammars, but still large

enough to deal with matching parentheses and other common programming language features.

We will return to the parsing problem of full context-free grammars later. We first look at the parsing techniques used in compilers, which work for some grammars only. In general, these techniques work for grammars that don't have **ambiguity**. That is, every string has at most one tree. This is not true for context-free grammars in general, but it is guaranteed for most programming languages by design. For parsing, the lack of ambiguity means that the algorithm can stop looking for alternative analyses as soon as it has found one, which is one ingredient of efficiency.

3.6 LL(k) parsing

The simplest practical way to parse programming languages is **LL(k)**, i.e. *left-to-right parsing, leftmost derivations, lookahead k*. It is also called **recursive descent parsing** and has sometimes been used for implementing parsers by hand, that is, without the need of parser generators. The **parser combinators** of Haskell are related to this method.

The idea of recursive descent parsing is the following: for each category, write a function that inspects the first token and tries to construct a tree. Inspecting one token means that the **lookahead** is one; LL(2) parsers inspect two tokens, and so on.

Here is an example grammar:

```
SIf.    Stm ::= "if" "(" Exp ")" Stm ;
SWhile. Stm ::= "while" "(" Exp ")" Stm ;
SExp.   Stm ::= Exp ;
EInt.   Exp ::= Integer ;
```

We need to build two functions, which look as follows in pseudocode,

```
Stm pStm():
  next == "if"    -> ...  build tree with SIf
  next == "while" -> ...  build tree with SWhile
  next is integer -> ...  build tree with SExp

Exp pExp():
  next is integer k -> return EInt k
```

To fill the three dots in this pseudocode, we proceed item by item in each production. If the item is a nonterminal *C*, we call the parser `p`*C*. If it is a terminal, we just check that this terminal is the next input token, but don't save it (since we are constructing an *abstract* syntax tree!). For instance, the first branch in the statement parser is the following:

```
Stm pStm():
  next == "if"    ->
    ignore("if")
    ignore("(")
    Exp e := pExp()
    ignore(")")
    Stm s := pStm()
    return SIf(e,s)
```

Thus we save the expression `e` and the statement `s` and build an `SIf` three from them, and ignore the terminals in the production.

The pseudocode shown is easy to translate to both imperative and functional code; we will return to the functional code in Section 3.9. But we don't recommend this way of implementing parsers, since BNFC is easier to write and more powerful. We show it rather because it is a useful introduction to the concept of **conflicts**, which arise even when BNFC is used.

As an example of a conflict, consider the rules for `if` statements with and without `else`:

```
SIf.     Stm ::= "if" "(" Exp ")" Stm
SIfElse. Stm ::= "if" "(" Exp ")" Stm "else" Stm
```

In an LL(1) parser, which rule should we choose when we see the token `if`? As there are two alternatives, we have a conflict.

One way to solve conflicts is to write the grammar in a different way. In this case, for instance, we can use **left factoring**, which means sharing the common left part of the rules:

```
SIE.   Stm  ::= "if" "(" Exp ")" Stm Rest
RElse. Rest ::= "else" Stm
REmp.  Rest ::=
```

To get the originally wanted abstract syntax, we have to define a function that depends on `Rest`.

```
f(SIE exp stm REmp)          = SIf     exp stm
f(SIE exp stm (RElse stm2)) = SIfElse exp stm stm2
```

It can be tricky ro rewrite a grammar so that it enables LL(1) parsing. Perhaps the most well-known problem is **left recursion**. A rule is left-recursive if it has the form *C ::= C...*, that is, the value category *C* is itself the first on the right hand side. Left recursion is common in programming languages, because operators such as `+` are left associative. For instance, consider the simplest pair of rules for sums of integers:

```
Exp ::= Exp "+" Integer
Exp ::= Integer
```

These rules make an LL(1) parser loop, because, to build an `Exp`, the parser first tries to build an `Exp`, and so on. No input is consumed when trying this, and therefore the parser loops.

The grammar can be rewritten, again, by introducing a new category:

```
Exp  ::= Integer Rest
Rest ::= "+" Integer Rest
Rest ::=
```

The new category `Rest` has **right recursion**, which is harmless. A tree conversion is of course needed to return the originally wanted abstract syntax.

The clearest way to see conflicts and to understand the nature of LL(1) parsing is to build a **parser table** from the grammar. This table has a row for each category and a column for each token. Each shell shows what rule applies when the category is being sought and it begins with the token. For example, the grammar

```
SIf.    Stm ::= "if" "(" Exp ")" Stm ;
SWhile. Stm ::= "while" "(" Exp ")" Stm ;
SExp.   Stm ::= Exp ";" ;
EInt.   Exp ::= Integer ;
```

produces the following table:

-	**if**	**while**	**integer**	**(**	**)**	**;**	**$ (END)**
Stm	SIf	SWhile	SExp	-	-	-	-
Exp	-	-	EInt	-	-	-	-

A conflict means that a cell contains more than one rule. This grammar has no conflicts, but if we added the `SIfElse` rule, the cell (Stm,`if`) would contain both `SIf` and `SIfElse`.

3.7 LR(k) parsing

Instead of LL(1), the standard YACC-like parser tools use **LR(k)**, i.e. *left-to-right parsing, rightmost derivations, lookahead k*. Both algorithms thus read their input left to right. But LL builds the trees from left to right, LR from right to left. The mention of **derivations** refers to the way in which a string can be built by expanding the grammar rules. Thus the **leftmost derivation** of `while(1) if (0) 6 ;` always fills in the leftmost nonterminal first.

```
Stm --> while ( Exp ) Stm
    --> while (   1 ) Stm
    --> while (   1 ) if ( Exp ) Stm
    --> while (   1 ) if (   0 ) Stm
    --> while (   1 ) if (   0 ) Exp ;
    --> while (   1 ) if (   0 )   6 ;
```

The **rightmost derivation** of the same string fills in the rightmost nonterminal first.

```
Stm --> while ( Exp ) Stm
    --> while ( Exp ) if ( Exp ) Stm
    --> while ( Exp ) if ( Exp ) Exp ;
    --> while ( Exp ) if ( Exp )   6 ;
    --> while ( Exp ) if (   0 )   6 ;
    --> while (   1 ) if (   0 )   6 ;
```

The LR(1) parser reads its input, and builds a **stack** of results, which are combined afterwards, as soon as some grammar rule can be applied to the top of the stack. When seeing the next token (lookahead 1), it chooses among five **actions**:

- **shift**: read one more token
- **reduce**: pop elements from the stack and replace by a value

- **goto**: jump to another state and act accordingly
- **accept**: return the single value on the stack when no input is left
- **reject**: report that there is input left but no move to take, or that the input is finished but the stack is not one with a single value.

Shift and reduce are the most common actions, and it is customary to illustrate the parsing process by showing the sequence of these actions. Take, for instance, the following grammar. We use integers as rule labels, so that we also cover the dummy coercion (label 2).

```
1. Exp  ::= Exp "+" Exp1
2. Exp  ::= Exp1
3. Exp1 ::= Exp1 "*" Integer
4. Exp1 ::= Integer
```

The string `1 + 2 * 3` is parsed as follows:

stack	**input**	**action**
	1 + 2 * 3	shift
1	+ 2 * 3	reduce 4
Exp1	+ 2 * 3	reduce 2
Exp	+ 2 * 3	shift
Exp +	2 * 3	shift
Exp + 2	* 3	reduce 4
Exp + Exp1	* 3	shift
Exp + Exp1 *	3	shift
Exp + Exp1 * 3	3	reduce 3
Exp + Exp1		reduce 1
Exp		accept

Initially, the stack is empty, so the parser must *shift* and put the token `1` to the stack. The grammar has a matching rule, rule 4, and so a *reduce* is performed. Then another reduce is performed by rule *2*. Why? This is because the next token (the lookahead) is `+`, and there is a rule that matches the sequence `Exp +`. If the next token were `*`, then the second reduce would not be performed. This is shown later in the process, when the stack is `Exp + Exp1`.

How does the parser know when to shift and when to reduce? Like in the case of LL(k) parsing, it uses a table. In an LR(1) table, the rows are **parser states**, and there is a column for each terminal and also nonterminal. The cells are parser actions.

So, what is a parser state? It is a grammar rule together with the position that has been reached when trying to match the ruls. This position is conventionally marked by a dot. Thus, for instance,

```
Stm ::= "if" "(" . Exp ")" Stm
```

is the state where an `if` statement is being read, and the parser has read the tokens `if` and `(` and is about to look for an `Exp`.

Here is an example of an LR(1) table. It is the table produced by BNFC and Happy from the previous grammar, so it is actually a variant called LALR(1); see below. The compiler has added two rules to the grammar: rule (0) that produces integer literals (`L_integ`) from the nonterminal `Integer`, and a start rule which adds the extra token `$` to mark the end of the input. Then also the other rules have to decide what to do if they reach the end of input.

```
                              +     *     $     int   Integer  Exp1  Exp
 3 Integer -> L_integ .       r0    r0    r0    -
 4 Exp1 -> Integer .          r4    r4    r4    -
 5 Exp1 -> Exp1 . '*' Integer -     s8    -     -
 6 %start_pExp -> Exp . $     s9    -     a     -
   Exp -> Exp . '+' Exp1
 7 Exp -> Exp1 .              r2    s8    r2    -
   Exp1 -> Exp1 . '*' Integer
 8 Exp1 -> Exp1 '*' . Integer -     -     -     s3    g11
 9 Exp -> Exp '+' . Exp1      -     -     -     s3    g4       g10
10 Exp -> Exp '+' Exp1 .      r1    s8    r1
   Exp1 -> Exp1 . '*' Integer
11 Exp1 -> Exp1 '*' Integer . r3    r3    r3
```

When the dot is before a nonterminal, a *goto* action is performed. Otherwise, either *shift* or *reduce* is performed. For *shift*, the next state is given. For *reduce*, the rule number is given.

The size of LR(1) tables can be large, because it is the number of rule positions multiplied by the number of tokens and categories. For LR(2), it

is the square of the number of tokens and categories, which is too large in practice. Even LR(1) tables are usually not built in their full form. Instead, standard tools like YACC, Bison, CUP, Happy use **LALR(1)**, *look-ahead LR(1)*. In comparison to full LR(1), LALR(1) reduces the number of states by merging some states that are similar to the left of the dot. States 6, 7, and 10 in the above table are examples of this.

In terms of general expressivity, the following inequations hold:

- LR(0) < LALR(1) < LR(1) < LR(2) ...
- LL(k) < LR(k)

That a *grammar* is in LALR(1), or any other of the classes, means that its parsing table has no conflicts. Therefore none of these classes can contain ambiguous grammars.

3.8 Finding and resolving conflicts

In a tabular parsing (LL, LR, LALR), a **conflict** means that there are several items in a cell. In LR and LALR, two kinds of conflicts may occur:

- **shift-reduce conflict**: between shift and reduce actions.
- **reduce-reduce conflict** between two (or more) reduce actions.

The latter are more harmful, but also more easy to eliminate. The clearest case is plain ambiguities. Assume, for instance, that a grammar tries to distinguish between variables and constants:

```
EVar.   Exp ::= Ident ;
ECons.  Exp ::= Ident ;
```

Any `Ident` parsed as an `Exp` can be reduced with both of the rules. The solution to this conflict is to remove one of the rules and wait until the type checher to distinguish constants from variables.

A more tricky case is implicit ambiguities. The following grammar tries to cover a fragment of C++, where a declaration (in a function definition) can be just a type (`DTyp`), and a type can be just an identifier (`TId`). At the same time, a statement can be a declaration (`SDecl`), but also an expression (`SExp`), and an expression can be an identifier (`EId`).

```
SExp.   Stm  ::= Exp ;
SDecl.  Stm  ::= Decl ;
DTyp.   Decl ::= Typ ;
EId.    Exp  ::= Ident ;
TId.    Typ  ::= Ident ;
```

Now the reduce-reduce conflict can be detected by tracing down a chain of rules:

```
Stm -> Exp -> Ident
Stm -> Decl -> Typ -> Ident
```

In other words, an identifier can be used as a statement in two different ways. The solution to this conflict is to redesign the language: `DTyp` should only be valid in function parameter lists, and not as statements!

As for shift-reduce conflicts, the classical example is the **dangling else**, created by the two versions of `if` statements:

```
SIf.     Stm ::= "if" "(" Exp ")" Stm
SIfElse. Stm ::= "if" "(" Exp ")" Stm "else" Stm
```

The problem arises when `if` statements are nested. Consider the following input and position (.):

```
if (x > 0) if (y < 8) return y ; . else return x ;
```

There are two possible actions, which lead to two analyses of the statement. The analyses are made explicit by braces.

```
shift:   if (x > 0) { if (y < 8) return y ;  else return x ;}
reduce:  if (x > 0) { if (y < 8) return y ;} else return x ;
```

This conflict is so well established that it has become a "feature" of languages like C and Java. It is solved by a principle followed by standard tools: when a conflict arises, always choose shift rather than reduce. But this means, strictly speaking, that the BNF grammar is no longer faithfully implemented by the parser.

Hence, if your grammar produces shift-reduce conflicts, this will mean that some programs that your grammar recognizes cannot actually be parsed.

Usually these conflicts are not so "well-understood" ones as the dangling else, and it can take a considerable effort to find and fix them. The most valuable tool in this work are the **info files** generated by some parser tools. For instance, Happy can be used to produce an info file by the flag `-i`:

```
happy -i ParCPP.y
```

The resulting file `ParConf.info` is a very readable text file. A quick way to check which rules are overshadowed in conflicts is to grep for the ignored *reduce* actions:

```
grep "(reduce" ParConf.info
```

Interestingly, conflicts tend cluster on a few rules. If you have very many, do

```
grep "(reduce" ParConf.info | sort | uniq
```

The conflicts are (usually) the same in all standard tools, since they use the LALR(1) method. Since the info file contains no Haskell, you can use Happy's info file if even if you principally work with another tool.

Another diagnostic tool is the **debugging parser**. In Happy,

```
happy -da ParCPP.y
```

When you compile the BNFC test program with the resulting `ParCPP.hs`, it shows the sequence of actions when the parser is executed. With Bison, you can use `gdb` (GNU Debugger), which traces back the execution to lines in the Bison source file.

3.9 The limits of context-free grammars

Parsing with unlimited context-free grammars is decidable, with cubic worst-case time complexity. However, exponential algorithms are often used because of their simplicity. For instance, the Prolog programming language has a built-in parser with this property. Haskell's **parser combinators** are a kind of embedded language for parsers, working in a similar way as Prolog. The method uses recursive descent, just like LL(k) parsers. But this is combined with **backtracking**, which means that the grammar need not make

deterministic choices. Parser combinators can also cope with ambiguity. We will return to them in Chapter 9.

One problem with Prolog and parser combinators—well known in practice—is their unpredictability. Backtracking may lead to exponential behaviour and very slow parsing. Left recursion may lead to non-termination, and can be hard to detect if implicit. Using parser generators is therefore a more reliable, even though more restricted, way to implement parsers.

But even the full class of context-free grammars is not the whole story of languages. There are some very simple formal languages that are *not* context-free. One example is the **copy language**. Each sentence of this language is two copies of some words, and the words can grow arbitrarily long. The simplest copy language has words with consisting of just two symbols, a and b:

$$\{ww | w \in (a|b)^*\}$$

Observe that this is *not* the same as the language defined by the context-free grammar

```
S ::= W W
W ::= "a" W
W ::= "b" W
W ::=
```

In this grammar, there is no guarantee that the two `W`'s are the same.

The copy language is not just a theoretical construct but has an important application in compilers. A common thing one wants to do is to check that every variable is declared before it is used. Language-theoretically, this can be seen as an instance of the copy language:

```
Program ::= ... Var ... Var ...
```

Consequently, checking that variables are declared before use is a thing that cannot be done in the parser but must be left to a later phase.

One way to obtain stronger grammars than BNF is by a separation of abstract and concrete syntax rules. For instance, the rule

```
EMul. Exp ::= Exp "*" Exp
```

then becomes a pair of rules: a `fun` rule declaring a tree-building function, and a `lin` rule defining the **linearization** of the tree into a string:

```
fun EMul : Exp -> Exp -> Exp
lin EMul x y = x ++ "*" ++ y
```

This notation is used in the grammar formalism **GF**, the **Grammatical Framework** (http://grammaticalframework.org). In GF, the copy language can be defined by the following grammar:

```
-- abstract syntax
cat S ; W ;
fun s : W -> S ;
fun e : W ;
fun a : W -> W ;
fun b : W -> W ;

-- concrete syntax
lin s w = w ++ w ;
lin e   = "" ;
lin a w = "a" ++ w ;
lin b w = "b" ++ w ;
```

For instance, `abbababbab` has the tree `s (a (b (b (a (b e)))))`.

GF corresponds to a class of grammars known as **parallel multiple context-free grammars**, which is useful in natural language description. Its worst-case parsing complexity is polynomial, where the exponent depends on the grammar; parsing the copy language is just linear. We will return to GF in Chapter 10.

Chapter 4

When does a program make sense

This chapter is about types and type checking. It defines the traditional notion of well-formedness as exemplified by e.g. Java and C, extended by overloading and some tricky questions of variable scopes. Many of these things are trivial for a human to understand, but the Main Assignment 2 will soon show that it requires discipline and stamina to make the machine check well-formedness automatically.

4.1 The purposes of type checking

Type checking annoys many programmers. You write a piece of code that makes complete sense to you, but you get a stupid type error. For this reason, untyped languages like LISP, Python, and JavaScript attract many programmers. They trade type errors for run-time errors, which means they spend relatively more time on debugging than on trying to compile, compared to Java or Haskell programmers. Of course, the latter kind of programmers learn to appreciate type checking, because it is a way in which the compiler can find bugs automatically.

The development of programming languages shows a movement to more and more type checking. This was one of the main additions of C++ over C. On the limit, a type checker could find *all* errors, that is, all violations of the specification. This is not the case with today's languages, not even the strictest ones like ML and Haskell. To take an example, a sorting function

`sort` might have the type

```
sort : List -> List
```

This means that the application `sort([2,1,3])` to a list is type-correct, but the application `sort(0)` to an integer isn't. However, the sorting function could still be defined in such a way that, for instance, sorting any list returns an empty list. This would have a correct type but it wouldn't be a correct sorting function. Now, this problem is one that could be solved by an even stronger type checker in a language such as **Agda**. Agda uses the **propositions as types principle** and which in particular makes it possible to express **specifications** as types. For instance, the sorting function could be declared

```
sort : (x : List) -> (y : List) & Sorted(x,y)
```

where the condition that the value `y` is a sorted version of the argument `x` is a part of the type. But at the time of writing this is still in the avant-garde of programming language technology.

Coming back to more standard languages, type checking has another function completely different from correctness control. It is used for **type annotations**, which means that it enables the compiler to produce more efficient machine code. For instance, JVM has separate instructions for integer and double-precision float addition (`iadd` and `dadd`, respectively). One might always choose `dadd` to be on the safe side, but the code becomes more efficient if `iadd` is used whenever possible.

Since Java source code uses `+` ambiguously for integer and float addition, the compiler must decide which one is in question. This is easy if the operands are integer or float constants: it could be made in the parser. But if the operands are variables, and since Java uses the same kind of variables for all types, the parser cannot decide this. Ultimately, recalling the previous chapter, this is so because context-free grammars cannot deal with the copy language! It is the type checker that is aware of the **context**, that is, what variables have been declared and in what types. Luckily, the parser will already have analysed the source code into a tree, so that the task of the type checker is not hopelessly complicated.

4.2 Specifying a type checker

There is no standard tool for type checkers, which means they have to be written in a general-purpose programming language. However, there are standard notations that can be used for specifying the **type systems** of a language and easily converted to code in any host language. The most common notation is **inference rules**. An inference rule has a set of **premisses** $J_1, \ldots, J_n$ and a **conclusion** J, conventionally separated by a line:

$$\frac{J_1 \ \ldots \ J_n}{J}$$

This inference rule is read:

> *From the premisses $J_1, \ldots, J_n$, we can conclude J.*

The symbols $J_1, \ldots, J_n, J$ stand for **judgements**. The most common judgement in type systems is

$$e : T$$

read, *expression e has type T*. An example of an inference rule for C or Java is

$$\frac{a : bool \ \ b : bool}{a \texttt{\&\&} b : bool}$$

that is, *if a has type bool and b has type bool. then a* `&&` *b has type bool*.

4.3 Type checking and type inference

The first step from an inference rule to implementation is pseudo-code for **syntax-directed translation**. Typing rules for expression forms in an abstract syntax can be seen as clauses in a recursive function definition that traverses expression trees. There are two kinds of functions:

- **Type checking**: given an expression e and a type T, decide if $e : T$.
- **Type inference**: given an expression e, find a type T such that $e : T$.

When we translate a typing rule to type checking code, its conclusion becomes a case for pattern matching, and its premisses become recursive calls for type checking. For instance, the above `&&` rule becomes

```
check (a && b : bool) =
  check (a : bool)
  check (b : bool)
```

There are no patterns matching other types than `bool`, so type checking fails for them.

In a type inference rule, the premisses become recursive calls as well, but the type in the conclusion becomes the value returned by the function:

```
infer (a && b) =
  check (a : bool)
  check (b : bool)
  return bool
```

Notice that the function should not just return bool outright: it must also check that the operands are of type bool.

4.4 Context, environment, and side conditions

How do we type-check variables? Variables symbols like `x` can have *any* of the types available in a programming language. The type it has in a particular program depends on the **context**. In C and Java, the context is defined by declarations of variables. It is a data structure where one can look up a variable and get its type. So we can think of the context as as **lookup table** of (variable,type) pairs.

In inference rules, the context is denoted by the Greek letter Gamma, Γ. The judgement form for typing is generalized to

$$\Gamma \Longrightarrow e : T$$

which is read, *expression e has type T in context* Γ. Most typing rules are generalized by adding the same Γ to all judgements, because the context doesn't change.

$$\frac{\Gamma \Longrightarrow a : bool \;\; \Gamma \Longrightarrow b : bool}{\Gamma \Longrightarrow a \,\texttt{\&\&}\, b : bool}$$

This would be silly if it was *always* the case. However, as we shall see, the context does change in the rules for type checking declarations.

The places where contexts are needed for expressions are those that involve variables. First of all, the typing rule for variable expressions is

$$\frac{}{\Gamma \Longrightarrow x : T} \text{ if } x : T \text{ in } \Gamma$$

What does this mean? The condition "if $x : T$ in Γ" is not a judgement but a sentence in the **metalanguage** (English). Therefore it cannot appear above the inference line as one of the premisses, but beside the line, as a **side condition**. The situation becomes even cleared if we look at the pseudocode:

```
infer (Gamma,x) =
  T := lookup(x,Gamma)
  return T
```

Looking up the type of the variable is not a recursive call to `infer` or `check`, but another function, `lookup`.

One way to make this fully precise is to look at concrete Haskell code. Here we have the type inference and lookup functions

```
infer :: Context -> Exp -> Type
look  :: Ident -> Context -> Type
```

We also have to make the abstract syntax constructors explicit: we cannot write just `x` bur `EVar x`, when we infer the type of a variable expression. Then the type inference rule comes out as a pattern matching case in the definition of `infer`:

```
infer Gamma (EVar x) =
  let typ = look x Gamma
  in return typ
```

If the language has function definitions, we also need to look up the types of functions when type checking function calls ($f(a, b, c)$). We will assume that the context Γ also includes the type information for functions. Then Γ is more properly called the **environment** for type checking, and not just the context.

The only place where the function storage part of the environment ever changes is when type checking function definitions. The only place where it is needed is when type checking function calls. The typing rule involves

a lookup of the function in Γ as a side condition, and the typings of the arguments as premisses:

$$\frac{\Gamma \Longrightarrow a_1 : T_1 \quad \cdots \quad \Gamma \Longrightarrow a_n : T_n}{\Gamma \Longrightarrow f(a_1, \ldots, a_n) : T} \text{ if } f : (T_1, \ldots, T_n) \to T \text{ in } \Gamma$$

For the purpose of expressing the value of function lookup, we use the notation $(T_1, \ldots, T_n) \to T$ for the type of functions, even though it is not used in C and Java themselves.

4.5 Proofs in a type system

Inference rules are designed for the construction of **proofs**, which are structured as **proof trees**. A proof tree can be seen as a trace of the steps that the type checker performs when checking or inferring a type. For instance, if we want to prove that `x + y > y` is a boolean expressions when `x` and `y` are integer variables, we have to prove the judgement

$$(x : \texttt{int})(y : \texttt{int}) \Longrightarrow x\texttt{+}y\texttt{>}y : \texttt{bool}$$

Notice the notation for contexts:

$$(x_1 : T_1) \ldots (x_n : T_n)$$

This is also handy when writing inference rules, because it also allows us to write simply

$$\Gamma(x : T)$$

when we add a new variable to the context Γ.

Here is a proof tree for the judgement we wanted to prove:

$$\dfrac{\dfrac{(x : \texttt{int})(y : \texttt{int}) \Longrightarrow x : \texttt{int} \quad (x : \texttt{int})(y : \texttt{int}) \Longrightarrow y : \texttt{int}}{(x : \texttt{int})(y : \texttt{int}) \Longrightarrow x\texttt{+}y : \texttt{int}} \qquad (x : \texttt{int})(y : \texttt{int}) \Longrightarrow y : \texttt{int}}{(x : \texttt{int})(y : \texttt{int}) \Longrightarrow x\texttt{+}y\texttt{>}y : \texttt{bool}}$$

The tree can be made more explicit by adding explanations for the side conditions. Here they appear beside the top-most judgments, just making it clear that the typing of a variable is justified by the context:

$$\dfrac{\dfrac{\dfrac{}{(x : \texttt{int})(y : \texttt{int}) \Longrightarrow x : \texttt{int}}\ (x : \texttt{int}) \quad \dfrac{}{(x : \texttt{int})(y : \texttt{int}) \Longrightarrow y : \texttt{int}}\ (x : \texttt{int})}{(x : \texttt{int})(y : \texttt{int}) \Longrightarrow x\texttt{+}y : \texttt{int}} \qquad \dfrac{}{(x : \texttt{int})(y : \texttt{int}) \Longrightarrow y : \texttt{int}}\ (x : \texttt{int})}{(x : \texttt{int})(y : \texttt{int}) \Longrightarrow x\texttt{+}y\texttt{>}y : \texttt{bool}}$$

4.6 Overloading and type casts

Variables are examples of expressions that can have different types in different contexts. Another example is **overloaded operators**. The binary arithmetic operations (`+ - * /`) and comparisons (`== != < > <= >=`) are in many languages usable for different types.

For simplicity, let us assume that the only possible types for arithmetic and comparisons are `int` and `double`. The typing rules then look as follows:

$$\frac{\Gamma \Longrightarrow a : t \quad \Gamma \Longrightarrow b : t}{\Gamma \Longrightarrow a \texttt{+} b : t} \text{ if } t \text{ is } \texttt{int} \text{ or } \texttt{double}$$

$$\frac{\Gamma \Longrightarrow a : t \quad \Gamma \Longrightarrow b : t}{\Gamma \Longrightarrow a \texttt{==} b : \mathit{bool}} \text{ if } t \text{ is } \texttt{int} \text{ or } \texttt{double}$$

and similarly for the other operators. Notice that a `+` expression has the same type as its operands, whereas a `==` is always a boolean. The type of `+` can in both cases be inferred from the first operand and used for checking the second one:

```
infer (a + b) =
  t := infer (a)
  check (b : t)
  return t
```

Yet another case of expressions having different type is **type casts**. For instance, an integer can be cast into a double. This may sound trivial from the ordinary mathematical point of view, because integers are a subset of reals. But for most machines this is not the case, because the integers and doubles have totally different binary representations and different sets of instructions. Therefore, the compiler usually has to generate a conversion instruction for type casts, both explicit and implicit ones. We will leave out type casts from the language implemented in this book.

4.7 The validity of statements and function definitions

Expressions have types, which can be checked and inferred. But what happens when we type-check a statement? Then we are not interested in a type,

but just in whether the judgement is **valid**. For the validity of a statement, we need a new judgement form,

$$\Gamma \Longrightarrow s \text{ valid}$$

which is read, *statement s is valid in environment* Γ.

Checking whether a statement is valid often requires type checking some expressions. For instance, in a `while` statement the condition expression has to be boolean:

$$\frac{\Gamma \Longrightarrow e : \mathit{bool} \;\; \Gamma \Longrightarrow s \text{ valid}}{\Gamma \Longrightarrow \texttt{while (}e\texttt{)} \;\; s \text{ valid}}$$

What about expressions used as statements, for instance, assignments and some function calls? We don't need to care about what the type of the expression is, just that it has one—which means that we are able to infer one. Hence the expression statement rule is

$$\frac{\Gamma \Longrightarrow e : t}{\Gamma \Longrightarrow e\texttt{;} \text{ valid}}$$

A similar rule could be given to `return` statements. However, when they occur within function bodies, they can more properly be checked with respect to the return types of the functions.

Similarly to statements, function definitions just checked for validity:

$$\frac{(x_1 : T_1) \ldots (x_1 : T_m) \Longrightarrow s_1 \ldots s_n \text{ valid}}{T \; f(T_1 \, x_1, \ldots, T_m \, x_m)\{s_1 \; \ldots, s_n\} \text{ valid}}$$

The variables declared as parameters of the function define the context in which the body is checked. The body consists of a list of statements $s_1 \ldots s_n$, which are checked in this context. One can think of this as a shorthand for n premisses, where each statement is in turn checked in the same context. But this is not quite true, because the context may change from one statement to the other. We return to this in next section.

To be really picky, the type checker of function definitions should also check that all variables in the parameter list are distinct. We shall see in the next section that variables introduced in declarations are checked to be new. Then they must also be new with respect to the function parameters.

One *could* add to the conclusion of this rule that Γ is extended by the new function and its type. However, this would not be enough for allowing **recursive functions**, that is, functions whose body includes calls to the

function itself. Therefore we rather assume that the functions in Γ are added at a separate first pass of the type checker, which collects all functions and their types (and also checks that all functions have different names). We return to this in Section 4.9.

One *could* also add a condition that the function body contains a `return` statement of expected type. A more sophisticated version of this could also allow returns in `if branches`, for example,

```
if (fail()) return 1 ; else return 0 ;
```

4.8 Declarations and block structures

Variables get their types in **declarations**. Each declaration has a **scope**, which is within a certain **block**. Blocks in C and Java correspond roughly to parts of code between curly brackets, { and }. Two principles regulate the use of variables:

1. A variable declared in a block has its scope till the end of that block.

2. A variable can be declared again in an inner block, but not otherwise.

To give an example of these principles at work, let us look at a code with some blocks, declarations, and assignments:

```
{
  int x ;
  {
    x = 3 ;      // x : int
    double x ;   // x : double
    x = 3.14 ;
    int z ;
  }
  x = x + 1 ;    // x : int, receives the value 3 + 1
  z = 8 ;        // ILLEGAL! z is no more in scope
  double x ;     // ILLEGAL! x may not be declared again
}
```

Our type checker has to control that the block structure is obeyed. This requires a slight revision of the notion of context. Instead of a simple lookup table, Γ must be made into a **stack of lookup tables**. We denote this with a dot notation, for example,

$$\Gamma_1.\Gamma_2$$

where Γ_1 is an old (i.e. outer) context and Γ_2 an inner context. The innermost context is the top of the stack.

The lookup function for variables must be modified accordingly. With just one context, it looks for the variable everywhere. With a stack of contexts, it starts by looking in the top-mosts and goes deeper in the stack only if it doesn't find the variable.

A declaration introduces a new variable in the current scope. This variable is checked to be fresh with respect to the context. But how do we express that the new variable is added to the context in which the later statements are checked? This is done by a slight modification of the judgement that a statement is valid: we can write rules checking that a **sequence of statements** is valid,

$$\Gamma \Longrightarrow s_1 \dots s_n \text{ valid}$$

A declaration extends the context used for checking the statements that follow:

$$\frac{\Gamma(x : T) \Longrightarrow s_2 \dots s_n \text{ valid}}{\Gamma \Longrightarrow Tx; s_2 \dots s_n \text{ valid}}$$

In other words: a declaration followed by some other statements $s_2 \dots s_n$ is valid, if these other statements are valid in a context where the declared variable is added. This addition causes the type checker to recognize the effect of the declaration.

For block statements, we push a new context on the stack. In the rule notation, this is seen as the appearance of a dot after Γ. Otherwise the logic is similar to the declaration rule—but now, it is the statements inside the block that are affected by the context change, not the statements after:

$$\frac{\Gamma. \Longrightarrow r_1 \dots r_m \text{ valid} \quad \Gamma \Longrightarrow s_2 \dots s_n \text{ valid}}{\Gamma \Longrightarrow \{r_1 \dots r_m\} s_2 \dots s_n \text{ valid}}$$

The reader should now try out her hand in building a proof tree for the judgement

$$\Longrightarrow \texttt{int x ; x = x + 1 ;} \text{ valid}$$

This is a proof from the **empty context**, which means no variables are given beforehand. You should first formulate the proper rules of assignment expressions and integer literals, which we haven't shown. But they are easy.

4.9 Implementing a type checker

Implementing a type checker is our first large-scale lesson in **syntax-directed translation**. As shown in Section 4.3, this is done by means of inference and checking functions, together with some auxiliary functions for dealing with contexts shown in Section 4.4. The block structure (Section 4.8) creates the need for some more. Here is a summary of the functions we need:

```
Type     infer  (Env env, Exp e)
Void     check  (Env env, Type t, Exp e)
Void     check  (Env env, Statements t)
Void     check  (Env env, Definition d)
Void     check  (Program p)

Type     look   (Ident x, Env env)
FunType  look   (Ident x, Env env)
Env      extend (Env env, Ident x, Type t)
Env      extend (Env env, Definition d)
Env      push   (Env env)
Env      pop    (Env env)
Env      empty  ()
```

We make the `check` functions return a `Void`. Their job is to go through the code and silently return if the code is correct. If they encounter an error, they emit an error message. So does `infer` if type inference fails, and `look` if the variable or function is not found in the environment. The `extend` functions can be made to fail if the inserted variable or function name already exists in the environment.

Most of the types involved in the signature above come from the abstract syntax of the implemented language, hence ultimately from its BNF grammar. The exceptions are `Void`, `FunType`, and `Env`. `FunType` is a data structure that contains a list of argument types and a value type. `Env` contains a lookup table for functions and a stack of contexts. These are our first examples of **symbol tables**, which are essential in all compiler components.

We don't need the definitions of these types in the pseudocode, but just the functions for lookup and environment construction. But we will show possible Haskell and Java definitions below.

Here is the pseudocode for the top-level function checking that a program is valid. We assume that a program is a sequence of function definitions. It is checked in two passes: first, collect the type signatures of each function by running `extend` on each definition in turn. Secondly, check each function definition in the environment that now contains all the functions with their types.

```
check (def_1...def_n) =
  env := empty
  for each i = 1,...,n: extend(env,def_i)
  for each i = 1,...,n: check(env,def_i)
```

We assume that the `extend` function updates the environment `env` by a side effect, so that it contains all function types on the last line where `check` is called.

Checking a function definition is derived from the rule in Section 4.7:

```
check (env, typ fun (typ_1 x_1,...,typ_m x_m) {s_1...s_n}) =
  for each i = 1,...,m: extend(env,x_i, typ_i)
  check(env, s_1...s_n)
```

Checking a statement list needs pattern matching over different forms of statements. The most critical parts are declarations and blocks:

```
check (env, typ x ; s_2...s_n) =
  env' := extend(env, x, typ)
  check (env', s_2...s_n)

check (env, {r_1...r_m} s_2...r_n) =
  env1 := push(env)
  check(env1, r_1...r_m)
  env2 := pop(env1)
  check(env2, s_2...s_m)
```

The type checker we have been defining just checks the validity of programs without changing them. But usually the type checker is expected to return a more informative syntax tree to the later phases, a tree with **type annotations**. Then the type signatures become

```
<Type,Exp>     infer  (Env env, Exp e)
Exp            check  (Env env, Type t, Exp e)
Statements     check  (Env env, Statements t)
```

and so on. The abstract syntax needs to be extended by a constructor for type-annotated expressions. We will denote them with <exp : type> in the pseudocode. Then, for instance, the type inference rule for addition expression becomes

```
infer(env, a + b) =
  <typ,a'> := infer(env, a)
  b'       := check(env, b, typ)
  return <typ, <a' + b' : typ>>
```

4.10 Type checker in Haskell

The compiler pipeline

To implement the type checker in Haskell, we need three things:

- define the appropriate auxiliary types and functions;
- implement type checker and inference functions;
- put the type checker into the compiler pipeline.

A suitable pipeline looks as follows. It calls the lexer within the parser, and reports a syntax error if the parser fails. Then it proceeds to type checking, showing an error message at failure and saying "OK" if the check succeeds. When more compiler phases are added, the next one takes over from the OK branck of type checking.

```
compile :: String -> IO ()
compile s = case pProgram (myLexer s) of
  Bad err  -> do
    putStrLn "SYNTAX ERROR"
    putStrLn err
    exitFailure
  Ok tree -> case typecheck tree of
```

```
    Bad err -> do
      putStrLn "TYPE ERROR"
      putStrLn err
      exitFailure
    Ok _ -> putStrLn "OK"  -- or go to next compiler phase
```

The compiler is implementer in the **IO monad**, which is the most common example of Haskell's monad system. Internally, it will also use an **error monad**, which is here implemented by the **error type** defined in the BNFC generated code:

```
data Err a = Ok a | Bad String
```

The value is either `Ok` of the expected type or `Bad` with an error message.

Whatever monad is used, its actions can be **sequence**d. For instance, if

```
checkExp :: Env -> Exp -> Type -> Err ()
```

then you can make several checks one after the other by using `do`

```
do checkExp env exp1 typ
   checkExp env exp2 typ
```

You can **bind** variables returned from actions, and **return** values.

```
do typ1 <- inferExp env exp1
   checkExp env exp2 typ1
   return typ1
```

If you are only interested in side effects, you can use the dummy value type `()` (corresponds to `void` in C and `void` or `Object` in Java).

Symbol tables

The environment has separate parts for function type table and the stack of variable contexts. We use the `Map` type for symbol tables, and a list type for the stack. Using lists for symbol tables is also possible, but less efficient and moreover not supported by built-in update functions.

```
type Env = (Sig,[Context])      -- functions and context stack
type Sig = Map Id ([Type],Type) -- function type signature
type Context = Map Id Type      -- variables with their types
```

Auxiliary operations on the environment have the following types:

```
lookVar   :: Env -> Id -> Err Type
lookFun   :: Env -> Id -> Err ([Type],Type)
updateVar :: Env -> Id -> Type -> Err Env
updateFun :: Env -> Id -> ([Type],Type) -> Err Env
newBlock  :: Env -> Err Env
emptyEnv  :: Env
```

You should keep the datatypes abstract, i.e. use them only via these operations. Then you can switch to another implementation if needed, for instance to make it more efficient or add more things in the environment. You can also more easily modify your type checker code to work as an interpreter, where the environment is different but the same operations are needed.

Pattern matching for type checking and inference

Here is the statement checker for expression statements, declaratins, and `while` statements:

```
checkStm :: Env -> Type -> Stm -> Err Env
checkStm env val x = case x of
  SExp exp  -> do
    inferExp env exp
    return env
  SDecl type' x  ->
    updateVar env id type'
  SWhile exp stm  -> do
    checkExp env Type_bool exp
    checkStm env val stm
```

Checking expressions is defined in terms of type inference:

```
checkExp :: Env -> Type -> Exp -> Err ()
checkExp env typ exp = do
```

```
    typ2 <- inferExp env exp
    if (typ2 == typ) then
        return ()
      else
        fail $ "type of " ++ printTree exp
```

Here is type inference for some expression forms:

```
inferExp :: Env -> Exp -> Err Type
inferExp env x = case x of
  ETrue      -> return Type_bool
  EInt n     -> return Type_int
  EId id     -> lookVar env id
  EAdd exp0 exp -> inferArithmBin env exp0 exp
```

Checking the overloaded addition uses a generic auxiliary for binary arithmetic operations:

```
inferArithmBin :: Env -> Exp -> Exp -> Err Type
inferArithmBin env a b = do
  typ <- inferExp env a
  if elem typ [Type_int, Type_double] then do
      checkExp env b typ
    else
      fail $ "type of expression " ++ printTree exp -- ...
```

4.11 Type checker in Java

The visitor pattern

In Section 2.7, we showed a first example of syntax-directed translation in Java: a calculator defined by adding the `interpret()` method to each abstract syntax class. This is the most straightforward way to implement pattern matching in Java. However, it is not very modular, because it requires us to go through and change every class whenever we add a new method. In a compiler, we need to add a type checker, an interpreter, a code generator, perhaps some optimizations—and none of these methods will come out as a cleanly separated piece of code.

To solve this problem, Java programmers are recommended to use the **visitor pattern**. It is also supported by BNFC, which generates the **visitor interface** and skeleton code to implement a visitor. With this method, you can put each compiler component into a separate class, which implements the visitor interface.

Before attacking the type checker itself, let us look at a simpler example—the calculator. The abstract syntax class `Exp` now contains an interface called `Visitor`, which depends on two class parameters, `A` and `R`. It is these parameters that make the visitor applicable to different tasks. In type inference, for instance, `A` is a context and `R` is a type. Let us look at the code:

```
public abstract class Exp {
  public abstract <R,A> R accept(Exp.Visitor<R,A> v, A arg);
  public interface Visitor <R,A> {
    public R visit(Arithm.Absyn.EAdd p, A arg);
    public R visit(Arithm.Absyn.EMul p, A arg);
    public R visit(Arithm.Absyn.EInt p, A arg);
  }
}
public class EAdd extends Exp {
  public final Exp exp_1, exp_2;
  public <R,A> R accept(Arithm.Absyn.Exp.Visitor<R,A> v, A arg) {
    return v.visit(this, arg);
  }
}
public class EInt extends Exp {
  public final Integer integer_;
  public <R,A> R accept(Arithm.Absyn.Exp.Visitor<R,A> v, A arg) {
    return v.visit(this, arg);
  }
}
```

There are three ingredients in the visitor pattern:

- `Visitor<R,A>`, the interface to be implemented by each application
- `R visit(Tree p, A arg)`, the interface methods in `Visitor` for each constructor
- `R accept(Visitor v, A arg)`, the abstract class method calling the visitor

Let us see how the calculator is implemented with the visitor pattern:

```
public class Interpreter {
  public Integer interpret(Exp e) {
    return e.accept(new Value(), null ) ;
  }
  private class Value implements Exp. Visitor<Integer, Object> {
    public Integer visit (EAdd p, Object arg) {
      return interpret(p.exp_1) + interpret(p.exp_2) ;
    }
    public Integer visit (EMul p, Object arg) {
      return interpret(p.exp_1) * interpret(p.exp_2) ;
    }
    public Integer visit (EInt p, Object arg) {
      return p.integer_ ;
    }
  }
}
```

This is the summary of the components involved:

- the return type `R` is `Integer`.
- the additional argument `A` is just `Object`; we don't need it for anything.
- the main class is `Interpreter` and contains
  - the public main method, `Integer interpret(Exp e)`, calling the visitor with `accept`
  - the private class `Value`, which implements `Visitor` by making the `visit` method evaluate the expression

At least to me, the most difficult thing to understand with visitors is the difference between `accept` and `visit`. It helps to look at what exactly happens when the interpreter is run on an expression—let's say `2 + 3`:

```
interpret(EAdd(EInt(2),(EInt(3))))      --> [interpret calls accept]
EAdd(EInt(2),(EInt(3))).accept(v,null)  --> [accept calls visit]
visit(EAdd(EInt(2),(EInt(3))),null)     --> [visit calls interpret]
interpret(EInt(2)) + interpret(EInt(3)) --> [interpret calls accept, etc]
```

Of course, the logic is less direct than in Haskell's pattern matching:

```
interpret (EAdd (EInt 2) (EInt 3))      --> [interpret calls interpret]
interpret (EInt 2) + interpret (EInt 3) --> [interpret calls interpret, etc]
```

But this is how Java can after all make it happen in a modular, type-correct way.

Type checker components

To implement the type checker in Java, we need three things:

- define the appropriate `R` and `A` classes;
- implement type checker and inference visitors with `R` and `A`;
- put the type checker into the compiler pipeline.

For the return type `R`, we already have the class `Type` from the abstract syntax. But we also need a representation of function types:

```
public static class FunType {
  public LinkedList<Type> args ;
  public Type val ;
}
```

Now we can define the environment with two components: a symbol table (`Map`) of function type signatures, and a stack (`LinkedList`) of variable contexts. We also need lookup and update methods:

```
public static class Env {
  public Map<String,FunType> signature ;
  public LinkedList<Map<String,Type>> contexts ;

  public static Type lookVar(String id) { ...} ;
  public static FunType lookFun(String id) { ...} ;
  public static void updateVar (String id, Type ty) {...} ;
  // ...
}
```

We also need something that Haskell gives for free: a way to compare types for equality. This we can implement with a special enumeration type of **type codes**:

```
public static enum TypeCode { CInt, CDouble, CBool, CVoid } ;
```

Now we can give the headers of the main classes and methods:

```
public void typecheck(Program p) {
  }
public static class CheckStm implements Stm.Visitor<Env,Env> {
  public Env visit(SDecl p, Env env) {
  }
  public Env visit(SExp p, Env env) {
  }
  // ... checking different statements
public static class InferExp implements Exp.Visitor<Type,Env> {
  public Type visit(EInt p, Env env) {
  }
  public Type visit(EAdd p, Env env) {
  }
  // ... inferring types of different expressions
}
```

On the top level, the compiler ties together the lexer, the parser, and the type checker. Exceptions are caught at each level:

```
try {
  l = new Yylex(new FileReader(args[0]));
  parser p = new parser(l);
  CPP.Absyn.Program parse_tree = p.pProgram();
  new TypeChecker().typecheck(parse_tree);
} catch (TypeException e) {
    System.out.println("TYPE ERROR");
    System.err.println(e.toString());
    System.exit(1);
} catch (IOException e) {
    System.err.println(e.toString());
    System.exit(1);
} catch (Throwable e) {
    System.out.println("SYNTAX ERROR");
    System.out.println ("At line " + String.valueOf(l.line_num())
 + ", near \"" + l.buff() + "\" :");
    System.out.println("     " + e.getMessage());
    System.exit(1);
}
```

Visitors for type checking

Now, finally, let us look at the visitor code itself. Here is checking statements, with declarations and expression statements as examples:

```
public static class CheckStm implements Stm.Visitor<Env,Env> {
  public Env visit(SDecl p, Env env) {
    env.updateVar(p.id_,p.type_) ;
    return env ;
  }
  public Env visit(SExp s, Env env) {
    inferExp(s.exp_, env) ;
    return env ;
  }
 //...
}
```

Here is an example of type inference, for overloaded addition expressions:

```
public static class InferExpType implements Exp.Visitor<Type,Env> {
  public Type visit(demo.Absyn.EPlus p, Env env) {
    Type t1 = p.exp_1.accept(this, env);
    Type t2 = p.exp_2.accept(this, env);
    if (typeCode(t1) == TypeCode.CInt && typeCode(t2) == TypeCode.CInt)
       return TInt;
    else
    if (typeCode(t1) == TypeCode.CDouble && typeCode(t2) == TypeCode.CDouble)
       return TDouble;
    else
      throw new TypeException("Operands to + must be int or double.");
    }
  //...
}
```

The function `typeCode` converts source language types to their type codes:

```
public static TypeCode typeCode (Type ty) ...
```

It can be implemented by writing yet another visitor :-)

Chapter 5

How to run programs in an interpreter

This chapter concludes what is needed in a minimal full-scale language implementation: you can now run your program and see what it produces. This part is the Main Assignment 3, but it turns out to be almost the same as Main Assignment 2, thanks to the powerful method of **syntax-directed translation**. Of course, it is not customary to interpret Java or C directly on source code; but languages like JavaScript are actually implemented in this way, and it is the quickest way to get your language running. The chapter will conclude with another kind of an interpreter, one for the Java Virtual Machine. It is included more for theoretical interest than as a central task in this book.

5.1 Specifying an interpreter

Just like type checkers, interpreters can be abstractly specified by means of inference rules. The rule system of an interpreter is called the **operational semantics** of the language. The rules tell how to **evaluate** expressions and how to **execute** statements and whole programs.

The basic judgement form for expressions is

$$\gamma \Longrightarrow e \Downarrow v$$

which is read, *expression e evaluates to value v in environment γ*. It involves the new notion of **value**, which is what the evaluation returns, for instance,

an integer or a double. Values can be seen as a special case of expressions, mostly consisting of literals; we can also eliminate booleans by defining *true* as the integer 1 and *false* as 0.

The environment γ (which is a small Γ) now contains values instead of types. We will denote value environments as follows:

$$(x_1 := v_1) \ldots (x_n := v_n)$$

When interpreting (i.e. evaluating) a variable expression, we look up its value from γ. Thus the rule for evaluating variable expressions is

$$\frac{}{\gamma \Longrightarrow x \Downarrow v} \text{ if } x := v \text{ in } \gamma$$

The rule for interpreting `&&` expressions is

$$\frac{\gamma \Longrightarrow a \Downarrow u \quad \gamma \Longrightarrow b \Downarrow v}{\gamma \Longrightarrow a \,\&\&\, b \Downarrow u \times v}$$

where we use integer multiplication to interpret the boolean conjunction. Notice how similar this rule is to the typing rule,

$$\frac{\Gamma \Longrightarrow a : bool \quad \Gamma \Longrightarrow b : bool}{\Gamma \Longrightarrow a \,\&\&\, b : bool}$$

One could actually see the typing rule as a special case of interpretation, where the value of an expression is always its type.

5.2 Side effects

Evaluation can have **side effects**, that is, do things other than just return a value. The most typical side effect is changing the environment. For instance, the assignment expression `x = 3` on one hand returns the value 3, on the other changes the value of `x` to 3 in the environment.

Dealing with side effects needs a more general form of judgement: evaluating an expression returns, not only a value, but also a new environment γ'. We write

$$\gamma \Longrightarrow e \Downarrow < v, \gamma' >$$

which is read, *expression e evaluates to value v and the new environment* γ' *in environment* γ. The original form without γ' can still be used as a shorthand for the case where $\gamma' = \gamma$.

Now we can write the rule for assignment expressions:

$$\frac{\gamma \Longrightarrow e \Downarrow < v, \gamma' >}{\gamma \Longrightarrow x \texttt{=} e \Downarrow < v, \gamma'(x := v) >}$$

The notation $\gamma(x := v)$ means that we **update** the value of x in γ to v, which means that we **overwrite** any old value that x might have had.

Operational semantics is an easy way to explain the difference between **preincrements** (`++x`) and **postincrements** (`x++`). In preincrement, the value of the expression is $x+1$. In postincrement, the value of the expression is x. In both cases, x is incremented in the environment. With rules,

$$\frac{}{\gamma \Longrightarrow \texttt{++}x \Downarrow < v+1, \gamma(x := v+1) >} \text{ if } x := v \text{ in } \gamma$$

$$\frac{}{\gamma \Longrightarrow x\texttt{++} \Downarrow < v, \gamma(x := v+1) >} \text{ if } x := v \text{ in } \gamma$$

One might think that side effects only matter in expressions that have side effect themselves, such as assignments. But also other forms of expressions must be given all those side effects that occur in their parts. For instance, `++x - x++` is, even if perhaps bad style, a completely valid expression that should be interpreted properly. The interpretation rule for subtraction thus takes into account the changing environment:

$$\frac{\gamma \Longrightarrow a \Downarrow < u, \gamma' > \quad \gamma' \Longrightarrow b \Downarrow < v, \gamma'' >}{\gamma \Longrightarrow a \texttt{-} b \Downarrow < u - - - v, \gamma'' >}$$

What is the value of `++x - x++` in the environment $(x := 1)$? This is easy to calculate by building a proof tree:

$$\frac{(x := 1) \Longrightarrow \texttt{++x} \Downarrow < 2, (x := 2) > \quad (x := 2) \Longrightarrow \texttt{x++} \Downarrow < 2, (x := 3) >}{(x := 1) \Longrightarrow a \texttt{-} b \Downarrow < 0, (x := 3) >}$$

But what other value could the expression have in C, where the evaluation order of operands is specified to be undefined?

Another kind of side effects are **IO actions**, that is, **input and output**. For instance, printing a value is an output action side effect. We will not treat them with inference rules here, but show later how they can be implemented in the interpreter code.

5.3 Statements

Statements are executed for their side effects, not to receive values. Lists of statements are executed in order, where each statement may change the environment for the next one. Therefore the judgement form is

$$\gamma \Longrightarrow s_1 \ldots s_n \Downarrow \gamma'$$

This can, however, be reduced to the interpretation of single statements as follows:

$$\frac{\gamma \Longrightarrow s \Downarrow \gamma' \quad \gamma' \Longrightarrow s_2 \ldots s_n \Downarrow \gamma''}{\gamma \Longrightarrow s_1 \ldots s_n \Downarrow \gamma''}$$

Expression statements just ignore the value of the expression:

$$\frac{\gamma \Longrightarrow e \Downarrow < v, \gamma' >}{\gamma \Longrightarrow e\,; \Downarrow \gamma'}$$

For `if` and `while` statements, the interpreter differs crucially from the type checker, because it has to consider the two possible values of the condition expression. Therefore, `if` statements have two rules: one where the condition is true (1), one where it is false (0). In both cases, just one of the statements in the body is executed. But recall that the condition can have side effects!

$$\frac{\gamma \Longrightarrow e \Downarrow < 1, \gamma' > \quad \gamma' \Longrightarrow s \Downarrow \gamma''}{\gamma \Longrightarrow \texttt{if}\,(e)\,s\,\texttt{else}\,t \Downarrow \gamma''} \qquad \frac{\gamma \Longrightarrow e \Downarrow < 0, \gamma' > \quad \gamma' \Longrightarrow t \Downarrow \gamma''}{\gamma \Longrightarrow \texttt{if}\,(e)\,s\,\texttt{else}\,t \Downarrow \gamma''}$$

For `while` staments, the truth of the condition results in a loop where the body is executed and the condition tested again. Only if the condition becomes false (since the environment has changed) can the loop be terminated.

$$\frac{\gamma \Longrightarrow e \Downarrow < 1, \gamma' > \quad \gamma' \Longrightarrow s \Downarrow \gamma'' \quad \gamma'' \Longrightarrow \texttt{while}\,(e)\,s \Downarrow \gamma'''}{\gamma \Longrightarrow \texttt{while}\,(e)\,s \Downarrow \gamma'''}$$

$$\frac{\gamma \Longrightarrow e \Downarrow < 0, \gamma' >}{\gamma \Longrightarrow \texttt{while}\,(e)\,s \Downarrow \gamma'}$$

Declarations extend the environment with a new variable, which is first given a "null" value. Using this null value in the code results in a run-time error, but this is of course impossible to prevent by the compilation.

$$\frac{}{\gamma \Longrightarrow Tx; \Downarrow \gamma(x := \text{null})}$$

We don't check for the freshness of the new variable, because this has been done in the type checker! Here we follow the principle of Milner, the inventor of ML:

Well-typed programs can't go wrong.

However, in this very case we *would* gain something with a run-time check, if the language allows declarations in branches of `if` statements.

For block statements, we push a new environment on the stack, just as we did in the type checker. The new variables declared in the block are added to this new environment, which is popped away at exit from the block.

$$\frac{\gamma. \Longrightarrow s_1 \ldots s_n \Downarrow \gamma'.\delta}{\gamma \Longrightarrow \{s_1 \ldots s_n\} \Downarrow \gamma'}$$

What is happening in this rule? The statements in the block are interpreted in the environment $\gamma.$, which is the same as γ with a new, empty, variable storage on the top of the stack. The new variables declared in the block are collected in this storage, which we denote by δ. After the block, δ is discarded. But the old γ part may still have changed, because the block may have given new values to some old variables! Here is an example of how this works, with the environment after each statement shown in a comment.

```
{
  int x ;         // (x := null)
  {               // (x := null).
    int y ;       // (x := null).(y := null)
    y = 3 ;       // (x := null).(y := 3)
    x = y + y ;   // (x := 6).(y := 3)
  }               // (x := 6)
  x = x + 1 ;     // (x := 7)
}
```

5.4 Programs, function definitions, and function calls

How do we interpret whole programs and function definitions? We well assume the C convention that the entire program is executed by running its

`main` function. This means the evaluation of an expression that calls the `main` function. Also following C conventions, `main` has no arguments:

$$\gamma \Longrightarrow \texttt{main()} \Downarrow < v, \gamma' >$$

The environment γ is the **global environment** of the program. It contains no variables (as we assume there are no global variables). But it does contain all functions. It allows us to look up a function name f and get the parameter list and the function body.

In any function call, we execute body of the function in an environment where the parameters are given the values of the arguments:

$$\frac{\begin{array}{c}\gamma \Longrightarrow a_1 \Downarrow < v_1, \gamma_1 > \ \cdots \ \gamma_{m-1} \Longrightarrow a_m \Downarrow < v_m, \gamma_m > \\ \gamma.(x_1 := v_1) \ldots (x_m := v_m) \Longrightarrow s_1 \ldots s_n \Downarrow < v, \gamma' >\end{array}}{\gamma \Longrightarrow f(a_1, \ldots, a_n) \Downarrow < v, \gamma_m >} \quad \text{if } T\ f(T_1\, x_1, \ldots, T_m\, x_m)\{s_1\ \ldots, s_n\} \text{ in } \gamma$$

This is quite a mouthful. Let us explain it in detail:

- The first m premisses evaluate the arguments of the function call. As the environment can change, we show m versions of γ.
- The last premiss evaluates the body of the function. This is done in a new environment, which binds the parameters of f to its actual arguments.
- No other variables can be accessed when evaluating the body. This is indicated by the use of the dot (.). Hence the local variables in the body won't be confused with the old variables in γ. Actually, the old variables cannot be updated either, but this is already guaranteed by type checking. For the same reason, using γ_m instead of γ here wouldn't make any difference.
- The value that is returned by evaluating the body comes from the `return` statement in the body.

We have not yet defined how function bodies, which are lists of statements, can return values. We do this by a simple rule saying that the value returned comes from the expression of the last statement, which must be a `return` statement:

$$\frac{\gamma \Longrightarrow s_1 \ldots s_{n-1} \Downarrow \gamma' \quad \gamma' \Longrightarrow e \Downarrow < v, \gamma'' >}{\gamma \Longrightarrow s_1 \ldots s_{n-1}\, \texttt{return}\, e \Downarrow < v, \gamma'' >}$$

5.5 Laziness

The rule for interpreting function calls is an example of the **call by value** evaluation strategy. This means that the arguments are evaluated *before* the function body is evaluated. Its alternative is **call by name**, which means that the arguments are inserted into the function body as *expressions*, before evaluation. One advantage of call by name is that it doesn't need to evaluate expressions that don't actually occur in the function body. Therefore it is also known as **lazy evaluation**. A disadvantage is that, if the variable is used more than once, it has to be evaluated again and again. This, in turn, is avoided by a more refined technique of **call by need**, which is the one used in Haskell.

We will return to evaluation strategies in Chapter 7. Most languages, in particular C and Java, use call by value, which is why we have used it here, too. But they do have some exceptions to it. Thus the boolean expressions `a && b` and `a || b` are evaluated lazily. Thus in `a && b`, `a` is evaluated first. If the value is false (0), the whole expression comes out false, and `b` is not evaluated at all. This is actually important, because it allows the programmer to write

```
x != 0 && 2/x > 1
```

which would otherwise result in a division-by-zero error when `x == 0`.

The operational semantics resembles `if` and `while` statements in Section 5.3. Thus it is handled with two rules—one for the 0 case and one for the 1 case:

$$\frac{\gamma \Longrightarrow a \Downarrow < 0, \gamma' >}{\gamma \Longrightarrow a \&\& b \Downarrow < 0, \gamma' >} \qquad \frac{\gamma \Longrightarrow a \Downarrow < 1, \gamma' > \quad \gamma' \Longrightarrow b \Downarrow < v, \gamma'' >}{\gamma \Longrightarrow a \&\& b \Downarrow < v, \gamma'' >}$$

For `a || b`, the evaluation stops if `x == 1`.

5.6 Debugging interpreters

One advantage of interpreters is that one can easily extend them to **debuggers**. A debugger traverses the code just like an interpreter, but also prints intermediate results such as the current environment, accurately linked to each statement in the source code.

5.7 Implementing the interpreter

The code for the interpreter is mostly a straightforward variant of the type checker. The biggest difference is in the return types, and in the contents of the environment:

```
<Value,Env> eval   (Env env, Exp e)
Env         exec   (Env env, Statement s)
Void        exec   (Program p)

Value       look   (Ident x, Env env)
Fun         look   (Ident x, Env env)
Env         extend (Env env, Ident x, Value v)
Env         extend (Env env, Definition d)
Env         push   (Env env)
Env         pop    (Env env)
Env         empty  ()
```

The top-level interpreter first gathers the function definition to the environment, then executes the `main` function.

```
exec (def_1 ... def_n) =
  env := empty
  for each i = 1,...,n: extend(env, def_i)
  eval(env, main())
```

Executing statements and evaluating expressions follows from the semantic rules in the same way as type checking follows from typing rules. In fact, it is easier now, because we don't have to decide between type checking and type inference. For example:

```
exec(env, e;) =
  <_,env'> := eval(env,e)
  return env'

exec(env, while e s) =
  <v,env'> := eval(env,e)
  if v == 0
    return env'
```

```
    else
      env'' := exec(env',s)
      exec(env'',while e s)

  eval(env, a-b) =
    <env', u> := eval(env, a)
    <env'',v> := eval(env',b)
    return <env'', u-v>

  eval(env, f(a_1,...,a_m) =
    for each i = 1,...,m: <v_i,env_i> := eval(env_i-1, a_i)
    t f(t_1 x_1,...,t_m x_m){s_1...s_m}
    envf := extend(push(env),(x_1 := v_1)...(x_m := v_m)
    <_,v> := eval(envf, s_1...s_m)
    return <env_m,v>
```

The implementation language takes care of the operations on values, for instance, comparisons like `v == 0` and calculations like `u - v`.

The implementation language may also need to define some **predefined functions**, in particular ones needed for input and output. Four such functions are needed in the assignment of this book: reading and printing integers and doubles. The simplest way to implement them is as special cases of the `eval` function:

```
  eval(env, printInt(e)) =
    <env',v> := eval(env,e)
    print integer v to standard output
    return <void, env'>

  eval(env, readInt()) =
    read integer v from standard input
    return <v,env>
```

The type `Value` can be thought of as a special case of `Exp`, only containing literals, but it would be better implemented as an algebraic datatype. One way to do this is to derive the implementation from a BNFC grammar! This time, we don't use this grammar for parsing, but only for generating the datatype implementation and perhaps the function for printing integer and double values.

```
VInteger.   Value ::= Integer ;
VDouble.    Value ::= Double ;
VVoid.      Value ::= ;
VUndefined. Value ::= ;
```

But some work remains to be done with the arithmetic operations. You cannot simply write

```
VInteger(2) + VInteger(3)
```

because `+` in Haskell and Java is not defined for the type `Value`. Instead, you have to define a special function `addValue` to the effect that

```
addValue(VInteger(u),VInteger(v)) = VInteger(u+v)
addValue(VDouble(u), VDouble(v))  = VDouble(u+v)
```

You won't need any other cases because, once again, *well-typed programs can't go wrong!*

5.8 Interpreting Java bytecode

It is a common saying that "Java is an interpreted language". We saw already Chapter 1 that this is not quite true. The truth is that Java is compiled to another language, **JVM**, **Java Virtual Machine** or **Java bytecode**, and JVM is then interpreted.

JVM is very different from Java, and its implementation is quite a bit simpler. In Chapter 1, we saw an example, the execution of the bytecode compiled from the expression 5 + (6 * 7):

```
bipush 5  ; 5
bipush 6  ; 5 6
bipush 7  ; 5 6 7
imul      ; 5 42
iadd      ; 47
```

After `;` (the comment delimiter in JVM assembler), we see the **stack** as it evolves during execution. At the end, there value of the expression, 47, is found on the **top** of the stack. In our representation, the "top" is the right-most element.

Like most machine languages, JVM has neither expressions nor statements but just **instructions**. Here is a selections of instructions that we will use in the next chapter to compile into:

instruction	**explanation**
`bipush` n	push byte constant n
`iadd`	pop topmost two values and push their sum
`imul`	pop topmost two values and push their product
`iload` i	push value stored in address i
`istore` i	pop topmost value and store it in address i
`goto` L	go to code position L
`ifeq` L	pop top value; if it is 0 go to position L

The instructions working on integers have variants for other types in the full JVM; see next chapter.

The load and store instructions are used to compile variables. The code generator assigns a **memory address** to every variable. This address is an integer. Declarations are compiled so that the next available address is reserved to the variable in question; no instruction is generated. Using a variable as an expression means loading it, whereas assigning to it means storing it. The following code example with both C and JVM illustrates the workings:

```
int i ;            ; reserve address 0 for i
i = 9 ;            bipush 9
                   istore 0
int j = i + 3 ;    ; reserve address 1 for j
                   iload 0
                   bipush 3
                   iadd
                   istore 1
```

Control structures such as `while` loops are compiled to **jump instructions**: `goto`, which is an **unconditional jump**, and `ifeq`, which is a **conditional jump**. The jumps go to **labels**, which are positions in the code. Here is how `while` statements can be compiled:

```
                   TEST:
```

```
while (exp)    ===>    ; code to evaluate exp
  stm                  if (exp==0) goto END
                       ; code to execute stm
                       goto TEST
                     END:
```

We have been explaining the JVM in informal English. To build an interpreter, it is useful to have formal semantics. This time, the semantics is built by the use of **transitions**: simple rules that specify what each instruction does. This kind of semantics is also known as **small-step semantics**, as each rule specifies just one step of computation. In fact the big-step relation $\Downarrow$ can be seen as the **transitive closure** of the small-step relation `-->`:

$e \Downarrow v$ means that e `-->` $\ldots$ `-->` v in some number of steps.

To make this completely precise, we of course have to specify how the big and small step environments correspond to each other. But in the JVM case $e \Downarrow v$ can be taken to mean that executing the instructions in e returns the value v on top of the stack after some number of steps and then terminates.

The operational semantics for C/Java source code that we gave earlier in this chapter is correspondingly called **big-step semantics**. For instance, `a + b` is there specified by saying that `a` is evaluated first; but this can take any number of intermediate steps.

The format of our small-step rules for JVM is

```
< Instruction , Env > --> < Env' >
```

The environment *Env* has the following parts:

- a **code pointer** P,
- a **stack** S,
- a **variable storage** V,

The rules work on instructions, executed one at a time. The next instruction is determined by the code pointer. Each instruction can do some of the following:

- increment the code pointer: `P+1`

- change the code pointer according to a label: `P(L)`
- copy a value from a storage address: `V(i)`
- write a value in a storage address: `V(i := v)`
- push values on the stack: `S.v`
- pop values from the stack

Here are the small-step semantic rules for the instructions we have introduced:

```
<bipush v, P; V; S>      --> <P+1;  V;        S.v>
<iadd,     P; V; S.v.w>  --> <P+1;  V:        S.v+w>
<imul,     P; V; S.v.w>  --> <P+1;  V:        S.v*w>
<iload i,  P; V; S>      --> <P+1;  V;        S.V(i)>
<istore i, P: V; S.v>    --> <P+1;  V(i:=v); S>
<goto L,   P; V; S>      --> <P(L); V;        S>
<ifeq L,   P; V; S.0>    --> <P(L); V;        S>
<ifeq L,   P; V; S.v>    --> <P+1;  V;        S>  (v not 0)
```

The semantic rules are a precise, declarative specification of an interpreter. They can guide its implementation. But they also make it possible, at least in principle, to perform **reasoning about compilers**. If both the source language and the target language have a formal semantics, it is possible to define the **correctness of a compiler** precisely. For instance:

> An expression compiler c is *correct* if, for all expressions e, $e \Downarrow w$ if and only if $c(e) \Downarrow v$.

Chapter 6

Compiling to machine code

There is **semantic gap**, a gap between the basic language constructs, which make machine languages look frighteningly different from source languages. However, the syntax-directed translation method can be put into use once again, and Main Assignment 4 will be an easy piece for anyone who has completed the previous assignments.

6.1 The semantic gap

Java and JVM are based on different kinds of constructions. These differences create the **semantic gap**, which a compiler has to bridge. Here is a summary, which works for many other source and target languags as well:

high-level code	**machine code**
statement	instruction
expression	instruction
variable	memory address
value	bit vector
type	memory layout
control structure	jump
function	subroutine
tree structure	linear structure

The general picture is that machine code is simpler. This is what makes the correspondence of concepts into *many-one*: for instance, both statements

and expressions are compiled to instructions. The same property makes compilation of constructs into *one-many*: typically, one statement or expression translates to many instructions. For example,

```
x + 3   ==>  iload 0
             bipush 3
             iadd
```

But the good news resulting from this is that compilation is easy, because it can proceed by just *ignoring* some information in the source language! This comes with the qualification that some information not present in the source language must first be extracted from the code. This means, in particular, that the type checker has to annotate the syntax tree with type information.

6.2 Specifying the code generator

Just like type checkers and interpreters, we could specify a code generator by means of inference rules. One judgement form could be

$$\gamma \Longrightarrow e \downarrow c$$

which is read, *expression e generates code c in environment* γ. The rules for compiling `+` expressions could be

$$\frac{\gamma \Longrightarrow a \downarrow c \quad \gamma \Longrightarrow b \downarrow d}{\gamma \Longrightarrow \texttt{<}a\texttt{+}b\texttt{:int>} \downarrow c\,d\,\texttt{iadd}} \qquad \frac{\gamma \Longrightarrow a \downarrow c \quad \gamma \Longrightarrow b \downarrow d}{\gamma \Longrightarrow \texttt{<}a\texttt{+}b\texttt{:double>} \downarrow c\,d\,\texttt{dadd}}$$

thus one rule for each type, and with type annotations assumed to be in place.

However, we will use the linear, non-tree notation of pseudocode from the beginning. One reason is that inference rules are not traditionally used for this task, so the notation would be a bit self-made. Another, more important reason is that the generated code is sometimes quite long, and the rules could become too wide to fit on the page. But in any case, rules and pseudocode are just two concrete syntaxes for the same abstract ideas.

Thus the pseudocode for compiling `+` expressions becomes

```
compile(env, <a + b : t>) =
  c := compile(env,a)
```

```
d := compile(env,b)
if t == int
   return c d iadd
else
   return c d dadd
```

The type of this function is

```
Code compile (Env env, Exp e)
```

Even this is not the most common and handy way to specify the compiler. We will rather use the following format:

```
Void compile (Exp e)

compile(<a + b : t>) =
  compile(a)
  compile(b)
  if t == int
     emit(iadd)
  else
     emit(dadd)
```

This format involves two simplifications:

- the environment is kept implicit—as a global variable, which may be consulted and changed by the compiler;
- code is generated as a side effect—by the function `Void emit(Code c)`, which writes the code into a file.

6.3 The compilation environment

As in type checkers and interpreters, the environment stores information on functions and variables. More specifically,

- for each function, its type in the JVM notation;
- for each variable, its address as an integer.

The exact definition of the environment need not bother us in the pseudocode. We just need to know the utility functions that form its interface:

```
Address     look   (Ident x)
FunType     look   (Ident f)
Void        extend (Ident x, Size s)
Void        extend (Definition d)
Void        push   ()   // new context when entering block
Void        pop    ()   // exit from block, discard new context
Void        empty  ()   // discard all variables
Label       label  ()   // get fresh code label
```

The `label` function gives a fresh label to be used in jump instructions. All labels in the code for a function must be distinct, because the they must uniquely identify a code position.

When extending the environment with a new variable, the **size** of its value must be known. For integers, the size is 1, for doubles, 2. The addresses start from 0, which is given to the first variable declared. The first variables are the function parameters, after which the locals follow. Blocks can overshadow old variables as usual. Here is an example of how the variable storage develops in the course of a function:

```
int foo (double x, int y)
{                // x -> 0, y -> 2
  int i ;        // x -> 0, y -> 2, i -> 3
  bool b ;       // x -> 0, y -> 2, i -> 3, b -> 4
  {              // x -> 0, y -> 2, i -> 3, b -> 4 .
    double i ;   // x -> 0, y -> 2, i -> 3, b -> 4 . i -> 4
  }              // x -> 0, y -> 2, i -> 3, b -> 4
  int z ;        // x -> 0, y -> 2, i -> 3, b -> 4, z -> 5
}
```

6.4 Simple expressions and statements

The simplest expressions are the integer and double literals. The simplest instructions to compile them to are

- `ldc` i, for pushing an integer i

- `ldc2_w` *d*, for pushing a double *d*

These instructions are implemented in a special way by using a separate storage called the **runtime constant pool**. Therefore they are not the most efficient instructions to use for small numbers: for them, the JVM also has

- `bipush` *b*, for integers whose size is one byte
- `iconst_m1` for -1, `iconst_0` for 0, ..., `iconst_5` for 5
- `dconst_0` for 0.0, `dconst_1` for 1.1

The `dconst` and `iconst` sets are better than `bipush` because they need no second bit for the argument. It is of course easy to optimize the code generation to one of these. But let us assume, for simplicity, the use of the worst-case instructions:

```
compile(i) = emit(ldc i)
compile(d) = emit(ldc2_w d)
```

Arithmetic operations were already covered. The following scheme works for all eight cases:

```
compile(<a + b : t>) =  // - * /
  compile(a)
  compile(b)
  if t == int
     emit(iadd)  // isub imul idiv
  else
     emit(dadd)  // dsub dmul ddiv
```

Variables are loaded from the storage:

```
compile(<x : int>)    = emit(iload look(x))
compile(<x : double>) = emit(dload look(x))
```

Like for constants, there are special instructions available for small addresses.

Assignments need some care, since we are treating them as expressions which both have side effects and return values. A simple-minded compilation would give

```
i = 3 ;      ===>  iconst_3 ; istore_1
```

It follows from the semantics in Section 5.8 that after `istore`, the value 3 is no more on the stack. This is fine as long as the expression is used only as a statement. But if its value is needed, then we need both to store it and have it on the stack. One way to guarantee this is

```
iconst_3 ; istore_1 ; iload_1
```

Anther way is to **duplicate** the top of the stack with the instruction `dup`:

```
<dup, P; V; S.v>  --> <P+1; V; S.v.v>
```

This works for integers; the variant for doubles is `dup2`. Thus we can use the following compilation scheme for assignments:

```
compile(<x = e : t>) =
  compile(e)
  if t == int
    emit (dup)
    emit (istore look(x))
  else
    emit (dup2)
    emit (dstore look(x))
```

What about if the value is *not* needed? Then we can use the `pop` instruction,

```
<pop, P; V; S.v>  --> <P+1; V; S>
```

and its big sister `pop2`. The rule is common for all uses of expressions as statements:

```
compile(<e : t> ;) =
  compile(e)
  if t == int    emit (pop)
  else
  if t == double emit (pop2)
  else return
```

Notice that the `int` case in compilation schemes covers booleans as well. The last "else" case for expression statements takes care of expressions of type `void`: these leave nothing on the stack to pop. The only such expressions in our language are function calls with `void` as return type.

Declarations have a compilation scheme that emits no code, but just reserves a place in the variable storage:

```
compile(t x ;) =
  extend(x,size(t))
```

The size of a type is 1 for integers and booleans, 2 for doubles. The `extend` helper function looks up the smalles available address for a variable, say i, and updates the compilation environment with the entry $(x \to i)$. The "smallest available address" is incremented by the size of the type.

Blocks are likewise compiled without emitting any code:

```
compile({s_1 ... s_n}) =
  push
  for each i = 1,...,n: compile(s_i)
  pop
```

6.5 Expressions and statements with jumps

The expressions and statements of the previous section are "simple" in the sense that they are compiled into **straight code**, that is, code without **jumps**, executed in the order of instructions. Code that is not straight is needed for `if` and `while` statements but also, as we will see now, many expressions involving booleans.

The basic way to compile `while` statements is as follows:

```
compile (while (exp) stm) =
  TEST := label
  END  := label
  emit (TEST)
  compile (exp)
  emit (ifeq END)
  compile (stm)
  emit (goto TEST)
  emit (END)
```

The generated code looks as follows:

```
                        TEST:
while (exp)     ===>      exp
  stm                     ifeq END
                          stm
                          goto TEST
                        END:
```

As specificed in Section 5.8, the `ifeq` instruction checks if the top of the stack is 0. If yes, the execution jumps to the label; if not, it continues to the next instruction. The checked value is the value of `exp` in the `while` condition. Value 0 means that the condition is false, hence the body is not executed. Otherwise, the value is 1 and the body `stm` is executed. After this, we take a jump back to the test of the condition.

`if` statements are compiled in a similar way:

```
if (exp)     ===>       evaluate exp
  stm1                  if (exp==0) goto FALSE
else                    execute stm1
  stm2                  goto TRUE
                      FALSE:
                        execute stm2
                      TRUE:
```

The idea is to have a label for false case, similar to the label `END` in `while` statements. But we also need a label for true, to prevent the execution of the `else` branch. The compilation scheme is straightforward to extreact from this example.

JVM has no booleans, no comparison operations, no conjunction or disjunction. Therefore, if we want to get the value of `exp1` $<$ `exp2`, we execute code corresponding to

```
if (exp1 < exp2) 1 ; else 0 ;
```

We use the conditional jump `if_icmplt LABEL`, which compares the two elements at the top of the stack and jumps if the second-last is less than the last:

```
<if_icmplt L, P; V; S.a.b>  -->  <P(L); V; S>  if a < b
<if_icmplt L, P; V; S.a.b>  -->  <P+1;  V; S>  otherwise
```

We can use code that first pushes 1 on the stack. This is overwritten by 0 if the comparison does not succeed.

```
  bipush 1
  exp1
  exp3
  if_icmplt TRUE
  pop
  bipush 0
TRUE:
```

There are instructions similar to `if_icmplt` for all comparisons of integers: `eq`, `ne`, `lt`, `gt`, `ge`, and `le`. For doubles, the mechanism is different. There is one instruction, `dcmpg`, which works as follows:

```
<dcmpg, P; V; S.a.b> --> <P+1; V; S.v>
```

where $v = 1$ if $a > b, v = 0$ if $a == b, and v = -1$ if $a < b$. We leave it as an exercise to produce the full compilation schemes for both integer and double comparisons.

Putting together the compilation of comparisons and `while` loops gives terrible spaghetti code, shown in the middle column.

```
                              TEST:                 TEST:
while (x < 9) stm   ===>        bipush 1
                                iload 0               iload 0
                                bipush 9              bipush 9
                                if_icmplt TRUE        if_icmpge END
                                pop
                                bipush 0
                              TRUE:
                                ifeq goto END
                                stm                   stm
                                goto TEST             goto TEST
                              END:                  END:
```

The right column shows a better code doing the sam job. It makes the comparison directly in the `while` jump, by using its *negation* `if_icmpge`; recall that `!(a < b) == (a >= b)`. The problem is: how can we get this code by using the compilation schemes?

6.6 Compositionality

A syntax-directed translation function T is **compositional**, if the value returned for a tree is a function of the values for its immediate subtrees:

$$T(Ct_1 \dots t_1) = f(T(t_1), \dots, T(t_1))$$

In the implementation, this means that,

- in Haskell, pattern matching does not need patterns deeper than one;
- in Java, one visitor definition per class and function is enough.

In Haskell, it would be easy to use **noncompositional** compilation schemes, by deeper patterns:

```
compile (SWhile (ELt exp1 exp2) stm) = ...
```

In Java, another visitor must be written to define what can happen depending on the condition part of `while`.

Another approach is to use compositional code generation followed by a separate phase of **back-end optimization** of the generated code: run through the code and look for code fragments that can be improved. This technique is more modular and therefore usually preferable to noncompositional hacks in code generation.

6.7 Function calls and definitions

Function calls in JVM are best understood as a generalization of arithmetic operations:

1. Push the function arguments on the stack.
2. Evaluate the function (with the arguments as parameters).

3. Return the value on the stack, popping the arguments.

For instance, in function call `f(a,b,c)`, the stack evolves as follows:

```
S         // before the call
S.a.b.c   // entering f
S.        // executing f, with a,b,c in variable storage
S.v       // returning from f
```

The procedure is actually quite similar to what the interpreter did in Section 5.4. Entering a function `f` means that the the JVM jumps to the code for `f`, with the arguments as the first available variables. The evaluation doesn't have access to old variables or to the stack of the calling code, but these become available again when the function returns.

The compilation scheme looks as follows:

```
compile(f(a_1,...,a_n)) =
  for each i = 1,...,n: compile a_i
  typ := look f
  emit(invokestatic C/f typ)
```

The JVM instruction for function calls is `invokestatic`. As the name suggests, we are only considering `static` methods here. The instruction needs to know the type of the function. It also needs to know its class. But we assume for simplicity that there is a global class `C`where all the called functions reside. The precise syntax for `invokestatic` is shown by the following example:

```
invokestatic C/mean(II)I
```

This calls a function `int mean (int x, int y)` in class `C`. So the type is written with a special syntax where the argument types are in parentheses before the value type. The types have one-letter symbols corresponding to Java types as follows:

```
I = int, D = double, V = void, Z = boolean
```

There is no difference between integers and booleans in execution, but the JVM interpreter may use the distinction for **bytecode verification**, that is, type checking at run time. Notice that the class, function, and type are written without spaces between in the assembly code.

The top level structure in JVM (as in Java) is a **class**. Function definitions are included in classed as **methods**. Here is a function and the compiled method in JVM assembler:

```
int mean (int x, int y)          .method public static mean(II)I
{                                .limit locals 2
                                 .limit stack 2
                          ===>      iload_0
                                    iload_1
                                    iadd
                                    iconst_2
                                    idiv
  return ((n+m) / 2) ;              ireturn
}                                 .end method
```

The first line obviously shows the function name and type. The function body is in the indented part. Before the body, two limits are specified: the storage needed for local variables (V in the semantic rules) and the storage needed for the evaluation stack (S in the semantics).

The local variables include the two arguments but nothing else, and since they are integers, the limit is 2. The stack can be calculated by simulating the JVM: it reaches 2 when pushing the two variables, but never beyond that. The code generator can easily calculate these limits by maintaining them in the environment; otherwise, one can use rough limits such as 1000.

Now we can give the compilation scheme for function definitions:

```
compile (t f (t_1 x_1,...,t_m f_m) {s_1 ... s_n} =
  empty
  emit (.method public static f type(t_1 ... t_m t)
  emit (.limit locals locals(f))
  emit (.limit stack stack(f))
  for each i = 1,...,m: extend(x_i, size(t_i))
  for each i = 1,...,n: compile(s_i)
  emit (.end method)
```

We didn't show yet how to compile return statements. JVM has separate instructions for different types. Thus:

```
compile(return <e : t>;) =
  compile(e)
  if t==double emit(dreturn) else emit(ireturn)

compile(return;) =
  emit(return)
```

6.8 Putting together a class file

Class files can be built with the following template:

```
.class public Foo
.super java/lang/Object

.method public <init>()V
  aload_0
  invokenonvirtual java/lang/Object/&lt;init>()V
  return
.end method

; user's methods one by one
```

The methods are compiled as described in the previous section. Each method has its own stack, locals, and labels; in particular, a jump from one method can never reach a label in another method.

If we follow the C convention as in Chapter 5, the class must have a `main` method. In JVM, its type signature of is different from C:

```
.method public static main([Ljava/lang/String;)V
```

The code generator must therefore treat `main` as a special case.

The class name, `Foo` in the above template, can be generated by the compiler from the file name (without suffix). The IO functions (reading and printing integers and doubles; cf. Section 5.7) can be put into a separate class, say `IO`, and then called as usual:

```
invokestatic IO/printInt(I)V
invokestatic IO/readInt()I
```

The easiest way to produce the IO class is by writing a Java program `IO.java` and compile it to `IO.class`. Then you will be able run "standard" Java code together with code generated by your compiler.

The class file and all JVM code show so far is not binary code but assemply code. It follows the format of **Jasmin**, which is a **JVM assembler**. In order to create the class file `Foo.class`, you have to compile your source code into a Jasmin file `Foo.j`. This is assembled by the call

```
jasmin Foo.j
```

To run your own program, write

```
java Foo
```

This executes the `main` function.

The Jasmin program can be obtained from http://jasmin.sourceforge.net/

6.9 Compiling to native code

6.10 Memory management

Chapter 7

Functional programming languages

The Main Assignment material is over, and this chapter takes a look at a new, fascinating world, where the languages are much simpler but much more powerful. If the grammar for the C++ fragment treated before was 100 lines, this language can be defined on less than 20 lines. But the simplicity is more on the user's side than the compiler writer's: you are likely to bang your head against the wall a few times, until you get it right with recursion, call by name, closures, and polymorphism. This work is helped by a rigorous and simple rule system; more than ever before, you need your discipline and stamina to render it correctly in your implementation code.

Chapter 8

How simple can a language be*

The functional language shown in the previous chapter was very simple already, but it can be made even simpler: the minimal language of Lambda Calculus has just three grammar rules. It needs no integers, no booleans—almost nothing, since everything can be defined by those three rules. This leads us to the notion of Turing Completeness, and we will show another Turing complete language, which is an imperative language similar to C, but definable on less than ten lines. Looking at these languages gives us ideas to assess the popular thesis that "it doesn't matter what language you use, since it's the same old Turing machine anyway".

Chapter 9

Designing your own language

You are not likely to implement C++ or Haskell in your professional career. You are much more likely to implement a DSL, domain specific language. However, the things you have learned by going through the Main Assignments and the optional functional assignment give you all the tools you need to create your own language. You will feel confident to do this, and you also know the limits of what is realistic and what is not.

Chapter 10

Compiling natural language*

This last chapter introduces GF, Grammatical Framework, which is similar to BNFC but much more powerful. GF can be used for definining programming languages, which will be used as an introductory example. Not only can it parse them, but also type check them, and actually the translation from Java to JVM could be defined by a GF grammar. However, the main purpose of the additional power of GF is to cope with the complexities of natural languages. The assignment is to build a little system that documents a program by automatically generating text that describes it; this text can be rendered into your favourite language, be it English, French, German, Persian, Urdu, or any of the more than 20 languages available in GF libraries.

Index