Compiler Construction
A Practical Approach

F.J.F. Benders
J.W. Haaring
T.H. Janssen
D. Meffert
A.C. van Oostenrijk

January 29, 2003

Abstract

Plenty of literature is available to learn about compiler construction, but most of it is either too easy, covering only the very basics, or too difficult and accessible only to academics. We find that, most notably, literature about code generation is lacking and it is in this area that this book attempts to fill in the gaps.

In this book, we design a new language (Inger), and explain how to write a compiler for it that compiles to Intel assembly language. We discuss lexical analysis (scanning), LL(1) grammars, recursive descent parsing, syntax error recovery, identification, type checking and code generation using templates and give practical advice on tackling each part of the compiler.

Acknowledgements

The authors would like to extend their gratitude to a number of people who were invaluable in the conception and writing of this book. The compiler construction project of which this book is the result was started with the help of Frits Feldbrugge and Robert Holwerda. The Inger language was named after Inger Vermeir (in the good tradition of naming languages after people, like Ada). The project team was coordinated by Marco Devillers, who proved to be a valuable source of advice.

We would also like to thank Carola Doumen for her help in structuring the project and coordinating presentations given about the project. Cees Haaring helped us getting a number of copies of the book printed.

Furthermore, we thank Mike Wertman for letting us study his source of a Pascal compiler written in Java, and Hans Meijer of the University of Nijmegen for his invaluable compiler construction course.

Finally, we would like to thank the University of Arnhem and Nijmegen for letting us use a project room and computer equipment for as long as we wanted.

Contents

Chapter 1

Introduction

This book is about constructing a compiler. But what, precisely, *is* a compiler? We must give a clear and complete answer to this question before we can begin building our own compiler.

In this chapter, we will introduce the concept of a translator, and more specifically, a compiler. It serves as an introduction to the rest of the book and presents some basic definitions that we will assume to be clear throughout the remainder of the book.

1.1 Translation and Interpretation

A compiler is a special form of a *translator*:

Definition 1.1 (Translator)

A translator is a program, or a system, that converts an input text some language to a text in another language, with the same meaning.

■

One translator could translate English text to Dutch text, and another could translate a Pascal program to assembly or machine code. Yet another translator might translate chess notation to an actual representation of a chess board, or translate a web page description in HTML to the actual web page. The latter two examples are in fact not so much translators as they are *interpreters*:

Definition 1.2 (Interpreter)

An interpreter is a translator that converts an input text to its meaning, as defined by its semantics.

■

A BASIC-interpreter like GW-BASIC is a classic and familiar example of an interpreter. Conversely, a translator translates the expression `2+3` to, for example, machine code that evaluates to `5`. It does *not* translate directly to `5`. The processor (CPU) that executes the machine code is the actual interpreter, delivering the final result. These observations lead to the following relationship:

$$translators \subset interpreters$$

Sometimes the difference between the translation of an input text and its meaning is not immediately clear, and it can be difficult to decide whether a certain translator is an interpreter or not.

A compiler is a translator that converts program source code to some target code, such as Pascal to assembly code, C to machine code and so on. Such translators differ from translators for, for example, natural languages because their input is expected to follow very strict rules for form (syntax) and the meaning of an input text must always be clear, i.e. follow a set of *semantic* rules.

Many programs can be considered translators, not just the ones that deal with text. Other types of input and output can also be viewed as structured text (SQL queries, vector graphics, XML) which adheres to a certain syntax, and therefore be treated the same way. Many conversion tools (conversion between graphics formats, or HTML to LaTeX) are in fact translators. In order to think of some process as a translator, one must find out which alphabet is used (the set of allowed words) and which sentences are spoken. An interesting exercise is writing a program that converts chess notation to a chess board diagram.

Meijer ([1] presents a set of definitions that clarify the distinction between translation and interpretation. If the input text to a translator is a program, then that program can have its own input stream. Such a program can be translated without knowledge of the contents of the input stream, but it cannot be interpreted.

Let p be the program that must be translated, in programming language P, and let i be the input. Then the interpreter is a function v_P, and the result of the translation of p with input i is denoted as:

$$v_P(p, i)$$

If c is a translator, the same result is obtained by applying the translation $c(p)$ in a programming language M to the input stream i:

$$v_M(c(p), i)$$

Interpreters are quite common. Many popular programming languages cannot be compiled but must be interpreted, such as (early forms of) BASIC, Smalltalk, Ruby, Perl and PHP. Other programming languages provide both the option of compilation and the option of interpretation.

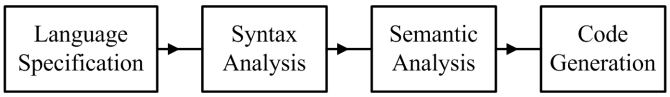
The rest of this book will focus on *compilers*, which translate program input texts to some target language. We are specifically interested in translating program input text in a programming language (in particular, Inger) to *Intel assemby language*.

1.2 Roadmap

Constructing a compiler involves specifying the programming language for which you wish to build a compiler, and then write a grammar for it. The compiler then reads source programs written in the new programming language and

checks that they are syntactically valid (well-formed). After that, the compiler verifies that the meaning of the program is correct, i.e. it checks the program's *semantics*. The final step in the compilation is generating code in the target language.

To help you vizualize where you are in the compiler construction process, every chapter begins with a copy of the *roadmap*:



One of the squares on the map will be highlighted.

1.3 A Sample Interpreter

In this section, we will discuss a very simple sample interpreter that calculates the result of simple mathematical expressions, using the operators + (addition), - (subtraction), * (multiplication) and / (division). We will work only with numbers consisting of one digits (0 through 9).

We will now devise a systematical approach to calculate the result of the expression

```
1 + 2 * 3 - 4
```

This is traditionally done by reading the input string on a character by character basis. Initially, the *read pointer* is set at the beginning of the string, just before the number `1`:

```
 1 + 2 * 3 - 4
^
```

We now proceed by reading the first character (or *code*), which happens to be `1`. This is not an operator so we cannot calculate anything yet. We must store the `1` we just read away for later use, and we do so by creating a *stack* (a last-in-first-out queue abstraction) and placing `1` on it. We illustrate this by drawing a vertical line between the items on the stack (on the left) and the items on the input stream (on the right):

```
1 |  + 2 * 3 - 4
   ^
```

The read pointer is now at the + operator. This operator needs two operands, only one of which is known at this time. So all we can do is store the + on the stack and move the read pointer forwards one position.

```
1 + | 2 * 3 - 4
     ^
```

The next character read is 2. We must now resist the temptation to combine this new operand with the operator and operand already on the stack and evaluate 1 + 2, since the rules of precedence dictate that we must evaluate 2 * 3, and then add this to 1. Therefore, we place (*shift*) the value 2 on the stack:

```
1 + 2 | * 3 - 4
        ^
```

We now read another operator (*) which needs two operands. We shift it on the stack because the second operand is not yet known. The read pointer is once again moved to the right and we read the number 3. This number is also placed on the stack and the read pointer now points to the operator -:

```
1 + 2 * 3 | - 4
            ^
```

We are now in a position to fold up (*reduce*) some of the contents of the stack. The operator - is of lower priority than the operator *. According to the rules of precedence, we may now calculate 2 * 3, which happen to be the topmost three items on the stack (which, as you will remember, is a last-in-first-out data structure). We pop the last three items off the stack and calculate the result, which is shifted back onto the stack. This is the process of *reduction*.

```
1 + 6 | - 4
        ^
```

We now compare the priority of the operator - with the priority of the operator + and find that, according to the rules of precedence, they have equal priority. This means we can either evaluate the current stack contents or continue shifting items onto the stack. In order to keep the contents of the stack to a minimum (consider what would happen if an endless number of + and - operators were encountered in succession) we reduce the contents of the stack first, by calculating 1 + 6:

```
7 | - 4
    ^
```

The stack can be simplied no further, so we direct our attention to the next operator in the input stream (-). This operator needs two operands, so we must shift the read pointer still further to the right:

```
7 - 4 |
        ^
```

We have now reached the end of the stream but are able to reduce the contents of the stack to a final result. The expression 7 - 4 is evaluated, yielding 3. Evaluation of the entire expression 1 + 2 * 3 - 4 is now complete and the algorithm used in the process is simple. There are a couple of interesting points:

1. Since the list of tokens already read from the input stream are placed on a stack in order to wait for evaluation, the operations *shift* and *reduce* are in fact equivalent to the operators *push* and *pop*.

2. The relative precedence of the operators encountered in the input stream determine the order in which the contents of the stack are evaluated.

Operators not only have *priority*, but also *associativity*. Consider the expression

$$1 - 2 - 3$$

The order in which the two operators are evaluated is significant, as the following two possible orders show:

$$(1 - 2) - 3 = -4$$

$$1 - (2 - 3) = 2$$

Of course, the correct answer is -4 and we may conclude that the - operator associates to the left. There are also (but fewer) operators that associate to the right, like the "to the power of" operator (^):

$$(2^3)^2 = 8^2 = 64 \quad (incorrect)$$

$$2^{(}3^2) = 2^9 = 512 \quad (correct)$$

A final class of operators is nonassociative, like +:

$$(1 + 4) + 3 = 5 + 3 = 8$$

$$1 + (4 + 3) = 1 + 7 = 8$$

Such operators may be evaluated either to the left or to the right; it does not really matter. In compiler construction, non-associative operators are often treated as left-associative operators for simplicity.

The importance of priority and associativty in the evaluation of mathematical expressions, leads to the observation, that an operator *priority list* is required by the interpreter. The following table could be used:

operator	**priority**	**associativity**
^	1	right
*	2	left
/	2	left
+	3	left
-	3	left

The parentheses, (and) can also be considered an operator, with the highest priority (and could therefore be added to the priority list). At this point, the priority relation is still incomplete. We also need invisible markers to indicate the beginning and end of an expression. The begin-marker [should be of the lowest priority (in order to cause every other operator that gets shifted onto an otherwise empty stack *not* to evaluate. The end-marker] should be of the lowest priority (just lower than [) for the same reasons. The new, full priority relation is then:

{ [,] } < { +, - } < { *, / } < { ^ }

The language discussed in our example supports only one-digit numbers. In order to support numbers of arbitrary length while still reading one digit at a time and working with the stack-based shift-reduce approach, we could introduce a new implicit concatenation operator:

$$1.2 = 1 * 10 + 2 = 12$$

Numerous other problems accompany the introduction of numbers of arbitrary length, which will not be discussed here (but most certainly in the rest of this book). This concludes the simple interpreter which we have crafted by hand. In the remaining chapter, you will learn how an actual compiler may be built using standard methods and techniques, which you can apply to your own programming language.

Bibliography

[1] H. Meijer: *Inleiding Vertalerbouw*, University of Nijmegen, Subfaculty of Computer Science, 2002.

[2] M.J. Scott: *Programming Language Pragmatics*, Morgan Kaufmann Publishers, 2000.

Chapter 2

Compiler History

This chapter gives an overview of compiler history. Programming languages can roughly be divided into three classes: *procedural* or *imperative* programming languages, *functional* programming languages and *object-oriented* programming languages. Compilers exist for all three classes and each type have their own quirks and specialties.

2.1 Procedural Programming

The first programming languages evolved from machine code. Instead of writing numbers to specify addresses and instructions you could write symbolic names. The computer executes this sequence of instructions when the user runs the program. This style of programming is known as *procedural* or *imperative* programming.

Some of the first procedural languages include:

- *FORTRAN*, created in 1957 by IBM to produce programs to solve mathematical problems. FORTRAN is short for *formula translation*.
- *Algol 60*, created in the late fifties with to goal to provide a "universal programming language". Even though the language was not widely used, its syntax became the standard language for describing algorithms.
- *COBOL*, a "data processing" language developed by Sammett introduced many new data types and implicit type conversion.
- *PL/I*, a programming language that would combine the best features of FORTRAN, Algol and COBOL.
- *Algol 68*, the successor to Algol 60. Also not widely used, though the ideas it introduced have been widely imitated.
- *Pascal*, created by Wirth to demonstrate a powerful programming langauge can be simple too, as opposed to the complex Algol 68, PL/I and others.

- *Modula2*, also created by Wirth as an improvement to Pascal with modules as most important new feature.
- *C*, designed by Ritchie as a low level language mainly for the task of system programming. C became very popular because UNIX was very popular and heavily depended on it.
- *Ada*, a large and complex language created by Whitaker and one of the latest attempts at designing a procedural language.

2.2 Functional Programming

Functional programming is based on the abstract model of programming Turing introduced, known as the *Turing Machine*. Also the theory of recursive functions Kleene and Church introduced play an important role in functional programming. The big difference with procedural programming is the insight that everything can be done with expression as opposed to commands.

Some important functional programming languages:

- *LISP*, the language that introduced functional programming. Developed by John McCarthy in 1958.
- *Scheme*, a language with a syntax and semantics very similar to LISP, but simpler and more consistent. Designed by Guy L. Steele Jr. and Gerald Lay Sussmann.
- *SASL*, short for St. Andrew's Symbolic Language. It was created by David Turner and has an Algol like syntax.
- *SML*, designed by Milner, Tofte and Harper as a "metalanguage".

2.3 Object Oriented Programming

Object oriented programming is entirely focused on objects, not functions. It has some major advantages over procedural programming. Well written code consists of objects that keep their data private and only accesible through certain methods (the interface), this concept is known as *encapsulation*. Another important object oriented programming concept is *inheritance*—a mechanism to have objects inherit state and behaviour from their superclass.

Important object oriented programming languages:

- *Simula*, developed in Norway by Kristiaan Nyaard and Ole-Johan Dahl. A language to model system, wich are collections of interacting processes (objects), wich in turn are represented by multiple procedures.
- *SmallTalk*, originated with Alan Kay's ideas on computers and programming. It is influenced by LISP and Simula.
- *CLU*, a language introduced by Liskov and Zilles to support the idea of *information hiding* (the interface of a module should be public, while the implementation remains private).

- *C++*, created at Bell Labs by Bjarne Soustroup as a programming language to replace C. It is a hybrid language — it supports both imperative and object oriented programming.
- *Eiffel*, a stricly object oriented language wich is strongly focused on software engineering.
- *Java*, a object oriented programming language with a syntax wich looks like C++, but much simpler. Java is compiled to byte code wich makes it portable. This is also the reason it became very popular for web development.
- *Kevo*, a language based on prototypes instead of classes.

2.4 Timeline

In this section, we give a compact overview of the timeline of compiler construction. As described in the overview article [4], the conception of the first computer language goes back as far as 1946. In this year (or thereabouts), Konrad Zuse, a german engineer working alone Konrad Zuse, a German engineer working alone while hiding out in the Bavarian Alps, develops *Plankalkul*. He applies the language to, among other things, chess. Not long after that, the first compiled language appears: *Short Code*, which the first computer language actually used on an electronic computing device. It is, however, a "hand-compiled" language.

In 1951, Grace Hopper, working for Remington Rand, begins design work on the first widely known compiler, named *A-0*. When the language is released by Rand in 1957, it is called *MATH-MATIC*. Less well-known is the fact that almost simulaneously, a rudimentary compiler was developed at a much less professional level. Alick E. Glennie, in his spare time at the University of Manchester, devises a compiler called *AUTOCODE*.

A few years after that, in 1957, the world famous programming language *FORTRAN* (FORmula TRANslation) is conceived. John Backus (responsible for his Backus-Naur Form for syntax specification) leads the development of FORTRAN and later on works on the *ALGOL* programming language. The publication of FORTRAN was quickly followed by *FORTRAN II* (1958), which supported subroutines (a major innovation at the time, giving birth to the concept of modular programming).

Also in 1958, John McCarthy at M.I.T. begins work on *LISP*–LISt Processing, the precursor of (almost) all functional programming languages we know today. Also, this is the year in which the *ALGOL* programming language appears (at least, the specification). The specification of ALGOL does not describe how data will be input or output; that is left to the individual implementations.

1959 was another year of much innovation. *LISP 1.5* appears and the functional programming paradigm is settled. Also, *COBOL* is created by the Conference on Data Systems and Languages (CODASYL). In the next year, the first

actual implementation of ALGOL appears (*ALGOL60*). It is the root of the family tree that will ultimately produce the likes of *Pascal* by Niklaus Wirth. ALGOL goes on to become the most popular language in Europe in the mid- to late-1960s.

Sometime in the early 1960s, Kenneth Iverson begins work on the language that will become *APL* – A Programming Language. It uses a specialized character set that, for proper use, requires APL-compatible I/O devices. In 1962, Iverson publishes a book on his new language (titled, aptly, *A Programming Language*). 1962 is also the year in which *FORTRAN IV* appears, as well as *SNOBOL* (StriNg-Oriented symBOlic Language) and associated compilers.

In 1963, the new language *PL/1* is conceived. This language will later form the basis for many other languages. In the year after, *APL/360* is implemented and at Dartmouth University, professors John G. Kemeny and Thomas E. Kurtz invent *BASIC*. The first implementation is a compiler. The first BASIC program runs at about 4:00 a.m. on May 1, 1964.

Languages start appearing rapidly now: 1965 - *SNOBOL3*. 1966 - *FORTRAN 66* and *LISP 2*. Work begins on *LOGO* at Bolt, Beranek, & Newman. The team is headed by Wally Fuerzeig and includes Seymour Papert. LOGO is best known for its "turtle graphics." Lest we forget: 1967 - *SNOBOL4*.

In 1968, the aptly named ALGOL68 appears. This new language is not altogether a success, and some members of the specifications committee–including C.A.R. Hoare and Niklaus Wirth–protest its approval. ALGOL 68 proves difficult to implement. Wirth begins work on his new language *Pascal* in this year, which also sees the birth of *ALTRAN*, a FORTRAN variant, and the official definition of *COBOL* by the American National Standards Institute (ANSI). Compiler construction attracts a lot of interest – in 1969, 500 people attend an APL conference at IBM's headquarters in Armonk, New York. The demands for APL's distribution are so great that the event is later referred to as "The March on Armonk."

Sometime in the early 1970s , Charles Moore writes the first significant programs in his new language, *Forth*. Work on *Prolog* begins about this time. Also sometime in the early 1970s, work on *Smalltalk* begins at Xerox PARC, led by Alan Kay. Early versions will include Smalltalk-72, Smalltalk-74, and Smalltalk-76. An implementation of *Pascal* appears on a CDC 6000-series computer. *Icon*, a descendant of SNOBOL4, appears.

Remember 1946? In 1972, the manuscript for Konrad Zuse's *Plankalkul* (see 1946) is finally published. In the same year, Dennis Ritchie and Brian Kernighan produces *C*. The definitive reference manual for it will not appear until 1974. The first implementation of *Prolog* – by Alain Colmerauer and Phillip Roussel – appears. Three years later, in 1975, *Tiny BASIC* by Bob Albrecht and Dennis Allison (implementation by Dick Whipple and John Arnold) runs on a microcomputer in 2 KB of RAM. A 4-KB machine is sizable, which left 2 KB available for the program. Bill Gates and Paul Allen write a version of BASIC that they sell to MITS (Micro Instrumentation and Telemetry Systems)

on a per-copy royalty basis. MITS is producing the Altair, an 8080-based microcomputer. Also in 1975, *Scheme*, a LISP dialect by G.L. Steele and G.J. Sussman, appears. Pascal User Manual and Report, by Jensen and Wirth, (also extensively used in the conception of Inger) is published.

B.W. Kerninghan describes *RATFOR* – RATional FORTRAN. It is a preprocessor that allows C-like control structures in FORTRAN. RATFOR is used in Kernighan and Plauger's "Software Tools," which appears in 1976. In that same year, the *Design System Language*, a precursor to *PostScript* (which was not developed until much later), appears.

In 1977, ANSI defines a standard for *MUMPS*: the Massachusetts General Hospital Utility Multi-Programming System. Used originally to handle medical records, MUMPS recognizes only a string data-type. Later renamed M. The design competition (ordered by the Department of Defense) that will produce Ada begins. A team led by Jean Ichbiah, will win the competition. Also, sometime in the late 1970s, Kenneth Bowles produces *UCSD Pascal*, which makes Pascal available on PDP-11 and Z80-based (remember the ZX-spectrum) computers and thus for "home use". Niklaus Wirth begins work on *Modula*, forerunner of *Modula-2* and successor to *Pascal*.

The text-processing language AWK (after the designers: Aho, Weinberger and Kernighan) becomes available in 1978. So does the ANSI standard for *FORTRAN 77*. Two years later, the first "real" implementation of *Smalltalk* (Smalltalk-80) appears. So does *Modula-2*. Bjarne Stroustrup develops "C With Classes", which will eventually become *C++*.

In 1981, design begins on *Common LISP*, a version of LISP that must unify the many different dialects in use at the time. Japan begins the "Fifth Generation Computer System" project. The primary language is *Prolog*. In the next year, the International Standards Organisation (ISO) publishes *Pascal* appears. PostScript is published (after DSL).

The famous book on Smalltalk: *Smalltalk-80: The Language and Its Implementation* by Adele Goldberg is published. *Ada* appears, the language named after Lady Augusta Ada Byron, Countess of Lovelace and daughter of the English poet Byron. She has been called the first computer programmer because of her work on Charles Babbage's analytical engine. In 1983, the Department of Defense '(DoD) directs that all new "mission-critical" applications be written in Ada.

In late 1983 and early 1984, Microsoft and Digital Research both release the first C compilers for microcomputers. The use of compilers by back-bedroom programmers becomes almost feasible. In July , the first implementation of C++ appears. It is in 1984 that Borland produces its famous *Turbo Pascal*. A reference manual for *APL2* appears, an extension of APL that permits nested arrays.

An important year for computer languages is 1985. It is the year in which *Forth* controls the submersible sled that locates the wreck of the Titanic. *Meth-*

ods, a line-oriented Smalltalk for personal computers, is introduced. Also, in 1986, jargonSmalltalk/V appears–the first widely available version of Smalltalk for microcomputers. Apple releases *Object Pascal* for the Mac, greatly popularizing the Pascal language. Borland extends its "Turbo" product line with *Turbo Prolog*.

Charles Duff releases *Actor*, an object-oriented language for developing Microsoft Windows applications. *Eiffel*, an object-oriented language, appears. So does *C++*. Borland produces the fourth incarnation of Turbo Pascal (1987). In 1988, the spefication of CLOS (Common LISP Object System) is fianlly published. Wirth finishes *Oberon*, his follow-up to Modula-2, his third language so far.

In 1989, the ANSI specification for *C* is published, leveraging the already popular language even further. *C++ 2.0* arrives in the form of a draft reference manual. The 2.0 version adds features such as multiple inheritance (not approved by everyone) and pointers to members. A year later, the *Annotated C++ Reference Manual* by Bjarne Stroustrup is published, adding templates and exception-handling features. *FORTRAN 90* includes such new elements as case statements and derived types. Kenneth Iverson and Roger Hui present *J* at the APL90 conference.

Dylan – named for Dylan Thomas – an object-oriented language resembling *Scheme*, is released by Apple in 1992. A year later, ANSI releases the X3J4.1 technical report – the first-draft proposal for object-oriented COBOL. The standard is expected to be finalized in 1997.

In 1994, Microsoft incorporates *Visual Basic for Applications* into Excel and in 1995, ISO accepts the 1995 revision of the *Ada* language. Called *Ada 95*, it includes OOP features and support for real-time systems.

This concludes the compact timeline of the evolution of programming languages. Of course, in the present day, another revolution is taking place, in the form of the Microsoft .NET platform. This platform is worth an entire book unto itself, and much literature is in fact already available. We will not discuss the .NET platform and the *common language specification* any further in this book. It is now time to move on to the first part of building our *own* compiler.

Bibliography

[1] T. Dodd: *An Advanced Logic Programming Language - Prolog-2 Encyclopedia*, Blackwell Scientific Publications Ltd., 1990.

[2] M. Looijen: *Grepen uit de Geschiedenis van de Automatisering*, Kluwer Bedrijfswetenschappen, Deventer, 1992.

[3] G. Moody: *Rebel Code - Inside Linux and the Open Source Revolution*, Perseus Publishing, 2001.

[4] N.N.*A Brief History of Programming Language*, BYTE Magazine, 20th anniversary, 1995.

[5] P.H. Salus: *Handbook of Programming Languages, Volume I: Object-oriented Programming Languages*, Macmillan Technical Publishing, 1998.

[6] P.H. Salus: *Handbook of Programming Languages, Volume II: Imperative Programming Languages*, Macmillan Technical Publishing, 1998.

[7] P.H. Salus: *Handbook of Programming Languages, Volume III: Little Languages and Tools*, Macmillan Technical Publishing, 1998.

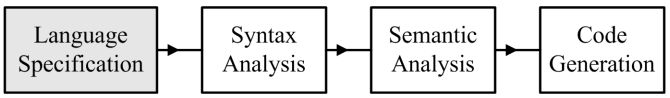
[8] P.H. Salus: *Handbook of Programming Languages, Volume IV: Functional and Logic Programming Languages*, Macmillan Technical Publishing, 1998.

Part I

Inger

Chapter 3

Language Specification



3.1 Introduction

This chapter gives a detailed introduction to the Inger language. The reader is assumed to have some familiarity with the concept of a programming language, and some experience with mathematics.

To give the reader an introduction to programming in general, we cite a short fragment of the introduction to the *PASCAL User Manual and Report* by Niklaus Wirth [7]:

> An *algorithm* or computer program consists of two essential parts, a description of the actions which are to be performed, and a description of the *data*, which are manipulated by so-called *statements*, and data are described by so-called *declarations* and *definitions*.

Inger provides language constructs (declarations) to define the data a program requires, and numerous ways to manipulate that data. In the next sections, we will explore Inger in some detail.

3.2 Program Structure

A program consists of one or more named modules, all of which contribute data or actions to the final program. Every module resides in its own source file. The best way to get to know Inger is to examine a small module source file. The program "factorial" (listing 3.1) calculates the factorial of the number 6. The output is $6! = 720$.

All modules begin with the module name, which can be any name the programmer desires. A module then contains zero or more *functions*, which encapsulate (parts of) algorithms, and zero or more *global variables* (global data). The functions and data declarations can occur in any order, which is made clear in a *syntax diagram* (figure 3.1). By starting from the left, one can trace

```
/* factor.i - test program.
   Contains a function that calculates
   the factorial of the number 6.
   This program tests the while loop. */

module test_module;

factor : int n → int
{
    int factor = 1;
    int i = 1;

    while( i <= n ) do
    {
        factor = factor * n;
        n = n + 1;
    }
    return( factor );
}

start main: void → void
{
    int f;
    f = factor ( 6 );
}
```

Listing 3.1: Inger Factorial Program

the lines leading through boxes and rounded enclosures. Boxes represent additional syntax diagrams, while rounded enclosures contain terminal symbols (those actually written in an Inger program). A syntactically valid program is constructed by following the lines and always taking smooth turns, never sharp turns. Note that dotted lines are used to break a syntax diagram in half that is too wide to fit on the page.

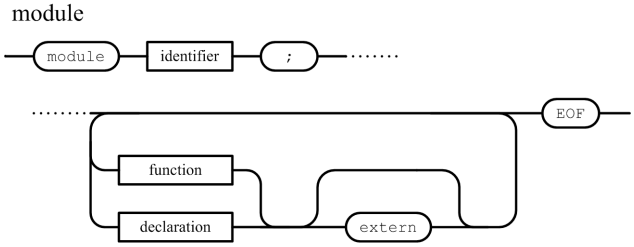


Figure 3.1: Syntax diagram for *module*

Example 3.1 (Tracing a Syntax Diagram)

As an example, we will show two valid programs that are generated by tracing the syntax diagram for *module*. These are not complete programs; they still contain the names of the additional syntax diagrams *function* and *declaration* that must be traced.

```
module Program_One;
```

Program_One is a correct program that contains no functions or declarations. The syntax diagram for *module* allows this, because the loop leading through either *function* or *declaration* is taken zero times.

```
module Program_Two;
extern function ;
 declaration ;
function ;
```

Program_Two is also correct. It contains two functions and one declaration. One of the functions is marked extern; the keyword extern is optional, as the the syntax diagram for *module* shows.

□

Syntax diagrams are a very descriptive way of writing down language syntax, but not very compact. We may also use *Backus-Naur Form* (BNF) to denote the syntax for the program structure, as shown in listing 3.2.

In BNF, each syntax diagram is denoted using one or more lines. The line begins with the name of the syntax diagram (a *nonterminal*), followed by a colon. The contents of the syntax diagram are written after the colon: nonterminals, (which have their own syntax diagrams), and *terminals*, which

```
module:     module identifier ; globals .
globals :   ϵ .
globals :   global  globals .
globals :   extern global globals.
global :    function .
global :    declaration .
```

Listing 3.2: Backus-Naur Form for module

are printed in bold. Since nonterminals may have syntax diagrams of their own, a single syntax diagram may be expressed using multiple lines of BNF. A line of BNF is also called a *production rule*. It provides information on how to "produce" actual code from a nonterminal. In the following example, we produce the programs "one" and "two" from the previous example using the BNF productions.

Example 3.2 (BNF Derivations)

Here is the listing from program "one" again:

```
module Program_One;
```

To derive this program, we start with the topmost BNF nonterminal, module. This is called the *start symbol*. There is only one production for this nonterminal:

```
module:     module identifier  ;  globals .
```

We now replace the nonterminal module with the right hand side of this production:

module ⟶ module identifier; globals

Note that we have underlined the nonterminal to be replaced. In the new string we now have, there is a new nonterminal to replace: globals. There are multiple production rules for globals:

```
globals :   ϵ .
globals :   global  globals .
globals :   extern global  globals .
```

Program "One" does not have any globals (declarations or functions), so we replace the nonterminal globals with the empty string (ϵ). Finally, we replace the nonterminal identifier. We provide no BNF rule for this, but it suffices to say that we may replace identifier with any word consisting of letters, digits and underscores and starting with either a letter or an underscore:

module
⟶ module identifier; globals
⟶ module Program_One; globals
⟶ module Program_One;

And we have created a valid program! The above list of production rule applications is a called a *derivation*. A derivation is the application of production rules until there are no nonterminals left to replace. We now create a derivation for program "Two", which contains two functions (one of which is extern, more on that later) and a declaration. We will not derive further than the function and declaration level, because these language structures will be explained in a subsequent section Here is the listing for program "Two" again:

```
module Program_Two;
extern function ;
 declaration ;
 function ;
```

module
⟶ module identifier; globals
⟶ module Program_Two; globals
⟶ module Program_Two; extern globals
⟶ module Program_Two; extern global globals
⟶ module Program_Two; extern function globals
⟶ module Program_Two; extern function global globals
⟶ module Program_Two; extern function declaration globals
⟶ module Program_Two; extern function declaration global globals
⟶ module Program_Two; extern function declaration function globals
⟶ module Program_Two; extern function declaration function

And with the last replacement, we have produced to source code for program "Two", exactly the same as in the previous example.

□

BNF is a somewhat rigid notation; it only allows the writer to make explicit the order in which nonterminals and terminals occur, but he must create additional BNF rules to capture repetition and selection. For instance, the syntax diagram for *module* shows that zero or more data declarations or functions may appear in a program. In BNF, we show this by introducing a production rule called *globals*, which calls itself (is recursive). We also needed to create another production rule called *global*, which has two alternatives (*function* and *declaration*) to offer a choice. Note that *globals* has three alternatives. One alternative is needed to end the repetition of functions and declarations (this is denoted with an ϵ, meaning *empty*), and one alternative is used to include the keyword extern, which is optional.

There is a more convenient notation called *Extended Backus-Naur Form* (EBNF), which allows the syntax diagram for *module* to be written like this:

(	)	[	]
!	-	+	~
*	&	*	/
%	+	-	>>
<<	<	<=	>
>=	==	!=	&
^	|	&&	||
?	:	=	,
;	->	{	}
bool	break	case	char
continue	default	do	else
extern	false	float	goto_considered_harmful
if	int	label	module
return	start	switch	true
untyped	while		

Table 3.1: Inger vocabulary

module: **module** identifier **;** { [**extern**]
(function | declaration) } .

In EBNF, we can use vertical bars (`|`) to indicate a choice, and brackets (`[` and `]`) to indicate an optional part. These symbols are called *metasymbols*; they are not part of the syntax being defined. We can also use the metasymbols `(` and `)` to enclose terminals and nonterminals so they may be used as a group. Braces (`{` and `}`) are used to denote repetition zero or more times. In this book, we will use both EBNF and BNF. EBNF is short and clear, but BNF has some advantages which will become clear in chapter 5, *Grammar*.

3.3 Notation

Like all programming languages, Inger has a number of *reserved words*, *operators* and *delimiters* (table 3.3). These words cannot be used for anything else than their intended purpose, which will be discussed in the following sections.

One place where the reserved words may be used freely, along with any other words, is inside a *comment*. A comment is input text that is meant for the programmer, not the compiler, which skips them entirely. Comments are delimited by the special character combinations `/*` and `*/` and may span multiple lines. Listing 3.3 contains some examples of legal comments.

The last comment in the example above starts with `//` and ends at the end of the line. This is a special form of comment called a *single-line comment*.

Functions, constants and variables may be given arbitrary names, or *identifiers* by the programmer, provided reserved words are not used for this purpose. An identifier must begin with a letter or an underscore (`_`) to discern it from a number, and there is no limit to the identifier length (except physical memory). As a rule of thumb, 30 characters is a useful limit for the length of identifiers. Although an Inger compiler supports names much longer than that, more than 30 characters will make for confusing names which are too long to read. All

```
/* This is a comment. */

/* This is also a comment,
   spanning multiple
   lines. */

/*
* This comment is decorated

* with extra asterisks to
* make it stand out.
*/

// This is a single-line comment.
```

Listing 3.3: Legal Comments

identifiers must be different, except when they reside in different *scopes*. Scopes will be discussed in greater detail later. We give a syntax diagram for identifiers in figure 3.2 and EBNF production rules for comparison:

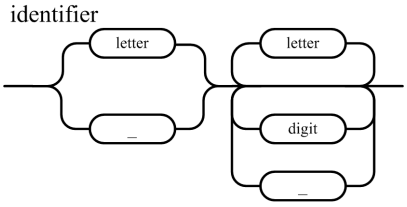


Figure 3.2: Syntax diagram for *identifier*

identifier : (_ | letter) { letter | digit | _ }
letter : **A** | ... | **Z** | **a** | ... | **z**
digit : **0** | ... | **9**

Example 3.3 (Identifiers)

Valid identifiers include:

```
x
_GrandMastaD_
HeLlO_wOrLd
```

Some examples of invalid identifiers:

```
2day
```

```
bool
@a
2+2
```

□

Of course, the programmer is free to choose wonderful names such as `___` or `x234`. Even though the language allows this, the names are not very descriptive and the programmer is encouraged to choose better names that describe the purpose of variables.

Inger supports two types of numbers: *integer numbers* ($x \in \mathbb{N}$), *floating point numbers* ($x \in \mathbb{R}$). Integer numbers consist of only digits, and are 32 bits wide. They have a very simple syntax diagram shown in figure 3.3. Integer numbers also include *hexadecimal numbers*, which are numbers with radix 16. Hexadecimal numbers are written using `0` through `9` and `A` through `F` as digits. The case of the letters is unimportant. Hexadecimal numbers must be prefixed with `0x` to set them apart from ordinary integers. Inger can also work with *binary numbers* (numbers with radix 2). These numbers are written using only the digits `0` and `1`. Binary numbers must be postfixed with `B` or `b` to set them apart from other integers.

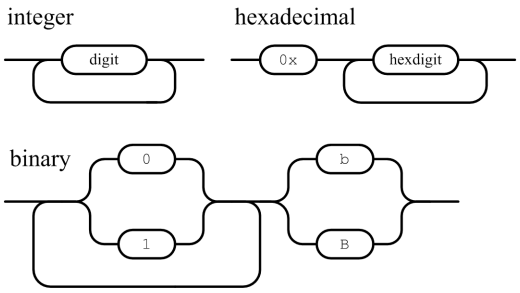


Figure 3.3: Syntax diagram for *integer*

Floating point numbers include a decimal separator (a dot) and an optional fractional part. They can be denoted using scientific notation (e.g. `12e-3`). This makes their syntax diagram (figure 3.4) more involved than the syntax diagram for integers. Note that Inger always uses the dot as the decimal separator, and not the comma, as is customary in some locations.

Example 3.4 (Integers and Floating Pointer Numbers)

Some examples of valid integer numbers:

```
3          0x2e          1101b
```

Some examples of invalid integer numbers (note that these numbers may be perfectly valid floating point numbers):

```
1a         0.2       2.0e8
```

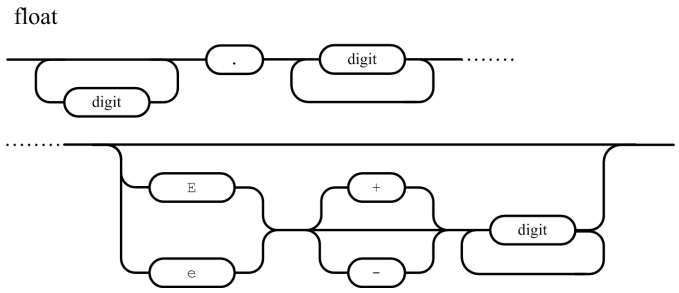


Figure 3.4: Syntax diagram for *float*

Some examples of valid floating point numbers:

```
0.2        2.0e8        .34e-2
```

Some examples of invalid floating point numbers:

```
2e-2        2a
```

□

Alphanumeric information can be encoded in either single *characters* of *strings*. A single character must be inclosed within apostrophes (') and a string must begin and and with double quotes ("). Any and all characters may be used with a string or as a single character, as long as they are printable (control characters cannot be typed). If a control character must be used, Inger offers a way to *escape* ordinary characters to generate control characters, analogous to *C*. This is also the only way to include double quotes in a string, since they are normally used to start or terminate a string (and therefore confuse the compiler if not treated specially). See table 3.2 for a list of escape sequences and the special characters they produce.

A string may not be spanned across multiple lines. Note that while whitespace such as spaces, tabs and end of line are normally used only to separate symbols and further ignored, whitespace within a string remains unchanged by the compiler.

Example 3.5 (Sample Strings and Characters)

Here are some sample single characters:

```
'b'        '&'        '7'        '"'        '''
```

Valid strings include:

```
"hello, world"        "123"
"\r\n"                "\"hi!\""
```

Escape Sequence	Special character
\"	"
\'	'
\\	\
\a	Audible bell
\b	Backspace
\B*nnnnnnnn*	Convert binary value to character
\f	Form feed
\n	Line feed
\o*nnn*	Convert octal value to character
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\x*nn*	Convert hexadecimal value to character

Table 3.2: Escape Sequences

□

This concludes the introduction to the nototional conventions to which valid Inger programs must adhere. In the next section, we will discuss the concept of data (variables) and how data is defined in Inger.

3.4 Data

Almost all computer programs operate on data, with which we mean numbers or text strings. At the lowest level, computers deal with data in the form of bits (binary digits, which a value of either `0` or `1`), which are difficult to manipulate. Inger programs can work at a higher level and offer several data abstractions that provide a more convenient way to handle data than through raw bits.

The data abstractions in Inger are *bool*, *char*, *float*, *int* and *untyped*. All of these except untyped are scalar types, i.e. they are a subset of $\mathbb{R}$. The untyped data abstraction is a very different phenomenon. Each of the data abstractions will be discussed in turn.

3.4.1 bool

Inger supports so-called *boolean*[1] *values* and the means to work with them. Boolean values are truth values, either *true* of *false*. Variables of the boolean data type (keyword `bool`) can only be assigned to using the keywords `true` or `false`, not `0` or `1` as other languages may allow.

[1] In 1854, the mathematician George Boole (1815–1864) published *An investigation into the Laws of Thought, on Which are founded the Mathematical Theories of Logic and Probabilities.* Boole approached logic in a new way reducing it to a simple algebra, incorporating logic into mathematics. He pointed out the analogy between algebraic symbols and those that represent logical forms. It began the algebra of logic called Boolean algebra which now has wide applications in telephone switching and the design of modern computers. Boole's work has to be seen as a fundamental step in today's computer revolution.

There is a special set of operators that work only with boolean values: see table 3.3. The result value of applying one of these operators is also a boolean value.

Operator	Operation
&&	Logical conjunction (and)
||	Logical disjunction (or)
!	Logical negation (not)

A	B	A && B
F	F	F
F	T	F
T	F	F
T	T	T

A	B	A || B
F	F	F
F	T	T
T	F	T
T	T	T

A	!A
F	T
T	F

Table 3.3: Boolean Operations and Their Truth Tables

Some of the relational operators can be applied to boolean values, and all yield boolean return values. In table 3.4, we list the relational operators and their effect. Note that == and != can be applied to other types as well (not just boolean values), but will always yield a boolean result. The assignment operator = can be applied to many types as well. It will only yield a boolean result when used to assign a boolean value to a boolean variable.

Operator	Operation
==	Equivalence
!=	Inequivalence
=	Assignment

Table 3.4: Boolean relations

3.4.2 int

Inger supports only one integral type, i.e. int. A variable of type int can store any $n \in \mathbb{N}$, as long as n is within the range the computer can store using its maximum word size. In table 3.5, we show the size of integers that can be stored using given maximum word sizes.

Word size	Integer Range
8 bits	-128..127
16 bits	-32768..32768
32 bits	-2147483648..2147483647

Table 3.5: Integer Range by Word Size

Inger supports only *signed integers*, hence the negative ranges in the table. Many operators can be used with integer types (see table 3.6), and all return a

value of type int as well. Most of these operators are polymorphic: their return type corresponds to the type of their operands (which must be of the same type).

Operator	Operation
-	unary minus
+	unary plus
~	bitwise complement
*	multiplication
/	division
%	modulus
+	addition
-	subtraction
>>	bitwise shift right
<<	bitwise shift left
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equality
!=	inequality
&	bitwise and
^	bitwise xor
|	bitwise or
=	assignment

Table 3.6: Operations on Integers

Of these operators, the unary minus (-), unary plus (+) and (unary) bitwise complement ( ) associate to the right (since they are unary) and the rest associates to the left, except assignment (=).

The relational operators =, !=, ¡, ¡=, ¿= and ¿ have a boolean result value, even though they have operands of type int. Some operations, such as additions and multiplications, can overflow when their result value exceeds the maximum range of the int type. Consult table 3.5 for the maximum ranges. If a and b are integer expressions, the operation

$$a \underline{op} b$$

will *not* overflow if ($\mathbb{N}$ is the integer range of a given system):

1. $a \underline{op} b \in \mathbb{N}$
2. $a \in \mathbb{N}$
3. $b \in \mathbb{N}$

3.4.3 float

The *float* type is used to represent an element of $\mathbb{R}$, although only a small part of $\mathbb{R}$ is supported, using 8 bytes. A subset of the operators that can be used

with operands of type int can also be used with operands of type float (see table 3.7).

Operator	Operation
-	unary minus
+	unary plus
*	multiplication
/	division
+	addition
-	subtraction
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equality
!=	inequality
=	assignment

Table 3.7: Operations on Floats

Some of these operations yield a result value of type float, while others (the relational operators) yield a value of type bool. Note that Inger supports only floating point values of 8 bytes, while other languages also support 4-byte so-called *float* values (while 8-byte types are called *double*).

3.4.4 char

Variables of type char may be used to store single unsigned bytes (8 bits) or single characters. All operations that can be performed on variables of type int may also be applied to operands of type char. Variables of type char may be initialized with actual characters, like so:

```
char c = 'a';
```

All escape sequences from table 3.2 may be used to initialize a variable of type char, although only one at a time, since a char represents only a single character.

3.4.5 untyped

In contrast to all the types discussed so far, the untyped type does not have a fixed size. untyped is a *polymorphic type*, which can be used to represent any other type. There is one catch: untyped must be used as a *pointer*.

Example 3.6 (Use of Untyped)

The following code is legal:

```
untyped *a; untyped **b;
```

but this code is not:

```
untyped p;
```

□

This example introduces the new concept of a pointer. Any type may have one or more levels of *indirection*, which is denoted using one more more asterisks (*). For an in-depth discussion on pointers, consult *C Programming Language*[1] by Kernighan and Ritchie.

3.5 Declarations

All data and functions in a program must have a name, so that the programmer can refer to them. No module may contain or refer to more than one function with the same name; every function name must be unique. Giving a variable or a function in the program a type (in case of a function: input types and an output type) and a name is called *declaring* the variable or function. All variables must be declared before they can be used, but functions may be used before they are defined.

An Inger program consists of a number of declarations of either *global variables* or functions. The variables are called global because they are declared at the outermost *scope* of the program. Functions can have their own variables, which are then called *local variables* and reside within the scope of the function. In listing 3.4, three global variables are declared and accessed from within the functions f en g. This code demonstrates that global variables can be accessed from within any function.

Local variables can only be accessed from within the function in which they are declared. Listing 3.5 shows a faulty program, in which variable i is accessed from a scope in which it cannot be seen.

Variables are declared by naming their type (bool, char, float, int or untyped, their level of indirection, their name and finally their array size. This structure is shown in a syntax diagram in figure 3.5, and in the BNF production rules in listing 3.6.

The syntax diagram and BNF productions show that it is possible to declare multiple variables using one declaration statement, and that variables can be initialized in a declaration. Consult the following example to get a feel for declarations:

Example 3.7 (Examples of Declarations)

```
char *a, b = 'Q', *c = 0x0;
int number = 0;
bool completed = false, found = true;
```

Here the variable a is a pointer char, which is not initialized. If no initialization value is given, Inger will initialize a variable to 0. b is of type char and is initialized to 'Q', and c is a pointer to data of type char. This pointer is initialized to 0x0 (null) – note that this does not initialize the char data to which c points to null. The variable number is of type int and is initialized to 0. The variables completed and found are of type bool and are initialized to false and true respectively.

□

```
/*
 * globvar.i - demonstration
 * of global variables.
 */
module globvar;

int i;
bool b;
char c;

g: void → void
{
    i = 0;
    b = false;
    c = 'b';
}

start f: void → void
{
    i = 1;
    b = true;
    c = 'a';
}
```

Listing 3.4: Global Variables

```
/*
 * locvar.i - demonstration
 * of local variables.
 */
module locvar;

g: void → void
{
    i = 1; /* will not compile */
}

start f: void → void
{
    int i = 0;
}
```

Listing 3.5: Local Variables

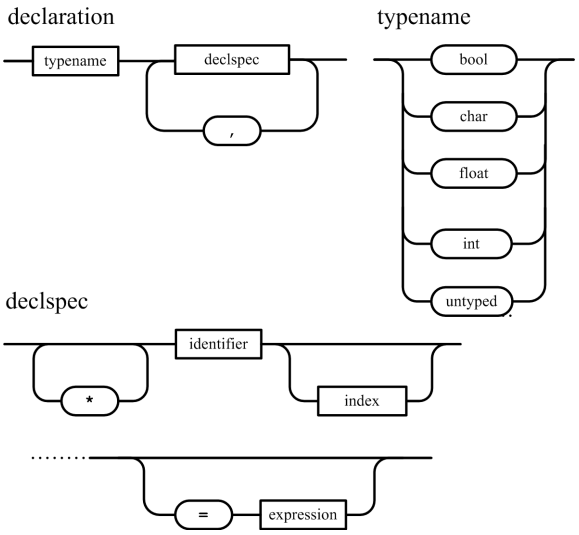


Figure 3.5: Declaration Syntax Diagram

declarationblock: type declaration { , declaration } .
declaration: { * } identifier { [intliteral] } [= expression] .
type: **bool** | **char** | **float** | **int** | **untyped**.

Listing 3.6: BNF for Declaration

3.6 Action

A computer program is not worth much if it does not contain some instructions (*statements*) that execute actions that operate on the data that program declares. Actions come in two categories: *simple statements* and *compound statements*.

3.6.1 Simple Statements

There are many possible action statements, but there is only one statement that actually has a *side effect*, i.e. manipulates data: this is the *assignment statement*, which stores a value in a variable. The form of an assignment is:

<variable> = <expression>

= is the assignment operator. The variable to which an expression value is assigned is called the *left hand side* or *lvalue*, and the expression which is assigned to the variable is called the *right hand side* or *rvalue*.

The expression on the right hand side can consist of any (valid) combination of constant values (numbers), variables, function calls and operators. The expression is *evaluated* to obtain a value to assign to the variable on the left hand side of the assignment. Expression evaluation is done using the well-known rules of mathematics, with regard to operator precedence and associativity. Consult table 3.8 for all operators and their priority and associativity.

Example 3.8 (Expressions)

$2 * 3 - 4 * 5$	$= (2 * 3) - (4 * 5)$	$= -14$
$15 / 4 * 4$	$= (15 / 4) * 4$	$= 12$
$80 / 5 / 3$	$= (80 / 5) / 3$	$= 5$
$4 / 2 * 3$	$= (4 / 2) * 3$	$= 6$
$9.0 * 3 / 2$	$= (9.0 * 3) / 2$	$= 13.5$

□

The examples show that a division of two integers results in an integer type (rounded down), while if either one (or both) of the operands to a division is of type float, the result will be float.

Any type of variable can be assigned to, so long as the expression type and the variable type are equivalent. Assignments may also be *chained*, with multiple variables being assigned the same expression with one statement. The following example shows some valid assignments:

Example 3.9 (Expressions)

```
int a, b;
int c = a = b = 2 + 1;
int my_sum = a * b + c; /* 12 */
```

□

All statements must be terminated with a semicolon (;).

Operator	Priority	Associatity	Description
`()`	1	L	function application
`[]`	1	L	array indexing
`!`	2	R	logical negation
`-`	2	R	unary minus
`+`	2	R	unary plus
`~`	3	R	bitwise complement
`*`	3	R	indirection
`&`	3	R	referencing
`*`	4	L	multiplication
`/`	4	L	division
`%`	4	L	modulus
`+`	5	L	addition
`-`	5	L	subtraction
`>>`	6	L	bitwise shift right
`<<`	6	L	bitwise shift left
`<`	7	L	less than
`<=`	7	L	less than or equal
`>`	7	L	greater than
`>=`	7	L	greater than or equal
`==`	8	L	equality
`!=`	8	L	inequality
`&`	9	L	bitwise and
`^`	10	L	bitwise xor
`|`	11	L	bitwise or
`&&`	12	L	logical and
`||`	12	L	logical or
`?:`	13	R	ternary if
`=`	14	R	assignment

Table 3.8: Operator Precedence and Associativity

3.6.2 Compound Statements

A compount statement is a group of zero or more statements contained within braces ({ and }). These statements are executed as a group, in the sequence in which they are written. Compound statements are used in many places in Inger including the body of a function, the action associated with an if-statement and a while-statement. The form of a compound statement is:

block: { code } .
code: e.
code: block code.
code: statement code.

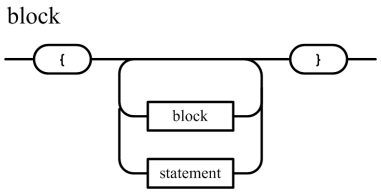


Figure 3.6: Syntax diagram for *block*

The BNF productions show, that a compound statement, or *block*, may contain zero (empty), one or more statements, and may contain other blocks as well. In the following example, the function f has a block of its own (the *function body*), which contains another block, which finally contains a single statement (a declaration).

Example 3.10 (Compound Statement)

```
module compound;

start f: void → void
{
    {

        int a = 1;
    }
}
```

□

3.6.3 Repetitive Statements

Compound statements (including compoung statements with only one statement in their body) can be wrapped inside a repetitive statement to cause it to be

executed multiple times. Some programming languages come with multiple flavors of repetitive statements; Inger has only one: the while statement.

The while statement has the following BNF productions (also consult figure 3.7 for the accompanying syntax diagram):

statement: **while** (expression) **do** block

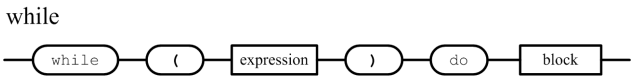


Figure 3.7: Syntax diagram for *while*

The expression between the parentheses must be of type bool. Before executing the compound statement contained in the block, the repetitive statement checks that expression evaluates to true. After the code contained in block has executed, the repetitive statement evaluates expression again and so on until the value of expression is false. If the expression is initially false, the compound statement is executed zero times.

Since the expression between parentheses is evaluated each time the repetitive statement (or *loop*) is executed, it is advised to keep the expression simple so as not to consume too much processing time, especially in longer loops.

The demonstration program in listing 3.7 was taken from the analogous *The while statement* section from Wirth's *PASCAL User Manual* ([7]) and translated to Inger.

The printint function and the #import directive will be discussed in a later section. The output of this program is 2.9287, printed on the console. It should be noted that the compound statement that the while statement must be contained in braces; it cannot be specified by itself (as it can be in the C programming language).

Inger provides some additional control statements, that may be used in conjunction with while: break and continue. The keyword break may be used to prematurely leave a while-loop. It is often used from within the body of an if statement, as shown in listings 3.8 and 3.9.

The continue statement is used to abort the current iteration of a loop and continue from the top. Its use is analogous to break: see listings 3.10 and 3.11.

The use of break and continue is discouraged, since they tend to make a program less readable.

3.6.4 Conditional Statements

Not every statement must be executed. The choice of statements to execute can be made using *conditional statements*. Inger provides the if and switch statements.

The if statement

An if statement consists of a boolean expression, and one or two compound statements. If the boolean expression is true, the first compound statement is

```
/*
 * Compute h(n) = 1 + 1/2 + 1/3 + ... + 1/n
 * for a known n.
 */
module while_demo;

#import "printint.ih"

start main: void → void
{
    int n = 10;
    float h = 0;

    while( n > 0 ) do
    {
        h = h + 1 / n;
        n = n − 1;
    }
    printint ( h );
}
```

Listing 3.7: The While Statement

```
int a = 10;

while( a > 0 )
{
    if ( a == 5 )
    {
        break;
    }

    printint ( a );
    a = a − 1;
}
```

Listing 3.8: The Break Statement

```
10
9
8
7
6
```

Listing 3.9: The Break Statement (output)

```
int a = 10;
while( a > 0 )
{
    if ( a % 2 == 0 )
    {
        continue;
    }

    printint ( a );
    a = a - 1;
}
```

Listing 3.10: The Continue Statement

```
10
8
6
4
2
```

Listing 3.11: The Continue Statement (output)

executed. If the boolean expression evaluates to false, the second compound statement (if any) is executed. Remember that compound statements need not contain multiple statements; they can contain a single statement or no statements at all.

The above definition of the if conditional statement has the following BNF producution associated with it (also consult figure 3.8 for the equivalent syntax diagram):

statement: **if** (expression) block elseblock
elseblock: ϵ.
elseblock: **else** block.

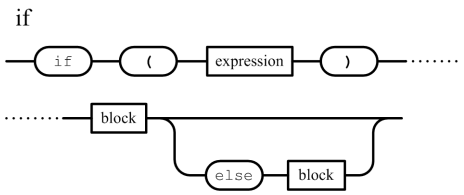


Figure 3.8: Syntax diagram for *if*

The productions for the elseblock show that the if statement may contain a second compound statement (which is executed if the boolean expression argu-

ment evaluates to false) or no second statement at all. If there is a second block, it must be prefixed with the keyword else.

As with the while statement, it is not possible to have the if statement execute single statements, only blocks contained within braces. This approach solves the *dangling else problem* from which the Pascal programming language suffers.

The "roman numerals" program (listing 3.12, copied from [7] and translated to Inger) illustrates the use of the if and while statements. Be aware that the Roman numerals are not translated entirely correctly (4 equals IV, not IIII), but this simplification makes the program easier. This was also done in Wirth's [7].

The case statement

The if statement only allows selection from two alternatives. If more alternatives are required, the else blocks must contain secondary if statements up to the required depth (see listing 3.14 for an example). Inger also provides the switch statement, which constists of an expression (the *selector*) and a list of alternatives *cases*. The cases are labelled with numbers (integers); the switch statement evaluates the selector expression (which must evaluate to type integer) and executes the alternative whose label matches the result. If no case has a matching label, switch executes the *default* case (which is required to be present). The following BNF defines the switch statement more precisely:

statement: **switch**(expression) { cases **default** block } .
cases: ϵ.
cases: **case** <int *literal*> *block cases*.

This is also shown in the syntax diagram in figure 3.9.

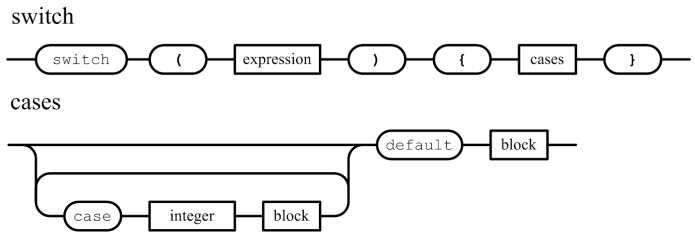


Figure 3.9: Syntax diagram for *switch*

It should be clear the use of the switch statement in listing 3.15 if much clearer than the multiway if statement from listing 3.14.

There cannot be duplicate case labels in a case statement, because the compiler would not know which label to jump to. Also, the order of the case labels is of no concern.

3.6.5 Flow Control Statements

Flow control statements are statements the cause the execution of a program to stop, move to another location in the program, and continue. Inger offers one such statement: the goto_considered_harmful statement. The name of this

```
/* Write roman numerals for the powers of 2. */
module roman_numerals;

#import "stdio.ih"

start main: void → void
{
    int x, y = 1;
    while( y <= 5000 )
    {
        x = y;
        printint ( x );

        while( x >= 1000 ) do
        {
            printstr ( "M" );
            x = x − 1000;
        }
        if ( x >= 500 )
        {
            printstr ( "D" );
            x = x − 500;
        }
        while( x >= 100 ) do
        {
            printstr ( "C" );
            x = x − 100;
        }
        if ( x >= 50 )
        {
            printstr ( "L" );
            x = x − 50;
        }
        while( x >= 10 ) do
        {
            printstr ( "X" );
            x = x − 10;
        }
        if ( x >= 5 )
        {
            printstr ( "V" );
            x = x − 5;
        }
        while( x >= 1 ) do
        {
            printstr ( "I" );
            x = x − 1;
        }
        printstr ( "\n" );
        y = 2 ∗ y;
    }
}
```

Listing 3.12: Roman Numerals

```
Output:
        1    I
        2    II
        4    IIII
        8    VIII
       16    XVI
       32    XXXII
       64    LXIIII
      128    CXXVIII
      256    CCLVI
      512    DXII
     1024    MXXIIII
     2048    MMXXXXVIII
     4096    MMMMLXXXXVI
```

Listing 3.13: Roman Numerals Output

```
if ( a == 0 )
{
    printstr ( "Case 0\n" );
}
else
{
    if ( a == 1 )
    {
            printstr ( "Case 3\n" );
    }
    else
    {
        if ( a == 2 )
        {
            printstr ( "Case 2\n" );
        }
        else
        {
            printstr ( "Case >2\n" );
        }
    }
}
```

Listing 3.14: Multiple If Alternatives

```
switch( a )
{
    case 0
    {
        printstr ( "Case 0\n" );
    }
    case 1
    {
        printstr ( "Case 1\n" );
    }
    case 2
    {
        printstr ( "Case 2\n" );
    }
    default                 ger
    {
        printfstr ( "Case >2\n" );
    }
}
```

Listing 3.15: The Switch Statement

```
int n = 10;
label here;
printstr ( n );
n = n - 1;
if ( n > 0 )
{
    goto_considered_harmful here;
}
```

Listing 3.16: The Goto Statement

statement (instead of the more common goto) is a tribute to the Dutch computer scientist Edger W. Dijkstra.[2]

The goto_considered_harmful statement causes control to jump to a specified (textual) *label*, which the programmer must provide using the label keyword. There may not be any duplicate labels throughout the entire program, regardless of scope level. For an example of the goto statement, see listing 3.16.

The goto_considered_harmful statement is provided for convenience, but its use is strongly discouraged (like the name suggests), since it is detrimental to the structure of a program.

[2]Edger W. Dijkstra (1930-2002) studied mathematics and physics in Leiden, The Netherlands. He obtained his PhD degree with a thesis on computer communications, and has since been a pioneer in computer science, and was awarded the ACM Turing Award in 1972. Dijkstra is best known for his theories about structured programming, including a famous article titled *Goto Considered Harmful*. Dijkstra's scientific work may be found at http://www.cs.utexas.edu/users/EWD.

3.7 Array

Beyond the simple types bool, char, float, int and untyped discussed earlier, Inger supports the advanced data type *array*. An array contains a predetermined number of elements, all of the same type. Examples are an array of elements of type int, or an array whose elements are of type bool. Types cannot be mixed.

The elements of an array are laid out in memory in a sequential manner. Since the number and size of the elements is fixed, the location of any element in memory can be calculated, so that all elements can be accessed equally fast. Arrays are called *random access structures* for this reason. In the section on declarations, BNF productions and a syntax diagram were shown which included array brackets ([and]). We will illustrate their use here with an example:

```
int a[5];
```

declares an array of five elements of type int. The individual elements can be accessed using the [] indexing operator, where the index is zero-based: a[0] accesses the first element in the array, and a[4] accesses the last element in the array. Indexed array elements may be used wherever a variable of the array's type is allowed. As an example, we translate another example program from N. Wirth's *Pascal User Manual* ([7]), in listing 3.17.

Arrays (matrices) may have more than one *dimension*. In declarations, this is specified thus:

```
int a[4][6];
```

which declares a to be a 4×6 matrix. Elements access is similar: a[2][2] accesses the element of a at row 2, column 2. There is no number to the number of dimensions used in an array.

Inger has no way to initialize an array, with the exception of character strings. An array of characters may be initialized with a string constant, as shown in the code below:

```
char a[20] = "hello, world!";
```

In this code, the first 13 elements of array a are initialized with corresponding characters from the string constant "hello, world". a[13] is initialized with zero, to indicate the end of the string, and the remaining characters are uninitialized. This example also shows that Inger works with *zero-minated strings*, just like the C programming language. However, one could say that Inger has no concept of string; a string is just an array of characters, like any other array. The fact that strings are zero-terminated (so-called ASCIIZ-strings) is only relevant to the system support libraries, which provide string manipulation functions.

It is not possible to assign an array to another array. This must be done on an element-by-element basis. In fact, if any operator except the indexing operator ([]) is used with an array, the array is treated like a *typed pointer*.

3.8 Pointers

Any declaration may include some level of indirection, making the variable a *pointer*. Pointers contain addresses; they are not normally used for storage

```
minmax: int a [],  n → void
{
    int  min, max, i , u , v;

    min = a[0]; max = min; i = 2;
    while( i < n−1 ) do
    {
        u = a[i ]; v = a[i+1];
        if ( u > v )
        {
            if ( u > max ) { max = u; }
            if ( v < min ) { min = v; }
        }
        else
        {
            if ( v > max ) { max = v; }
            if ( u < min ) { min = u; }
        }
        i = i + 2;
    }
    if ( i == n )
    {
        if ( a[n] > max )
        {
            max = a[n];
        }
        else  if ( a[n] < min )
        {
            min = a[n];
        }
    }
    printint ( min );
    printint ( max );
}
```

Listing 3.17: An Array Example

themselves, but to point to other variables (hence the name). Pointers are a convenient mechanism to pass large data structures between functions or modules. Instead of copying the entire data structure to the receiver, the receiver is told where it can access the data structure (given the address).

The & operator can be used to retrieve the address of any variable, so it can be assigned to a pointer, and the * operator is used to access the variable at a given address. Examine the following example code to see how this works:

```
int a;
int *b = &a;
*b = 2;
printint ( a ); /* 2 */
```

The variable b is assigned the *address* of variable a. Then, the value 2 is assigned to the variable to which b points (a), using the dereferencing operator (*). After this, a contains the value 2.

Pointers need not always refer to non-pointer variables; it is perfectly possible for a pointer to refer to another pointer. Pointers can also hold multiple levels of indirection, and can be dereferenced multiple times:

```
int a;
int *b = &a;
int **c = &b;
**c = 2;
printint ( a ); /* 2 */
```

Pointers have another use: they can contain the address of a *dynamic variable*. While ordinary variables declared using the declaration statements discussed earlier are called *static variables* and reside on the *stack*, dynamic varibles live on the *heap*. The only way to create them is by using operating system functions to allocate memory for them, and storing their address in a pointer, which must be used to access them for all subsequent operations until the operating system is told to release the memory that the dynamic variable occupies. The allocation and deallocation of memory for dynamic variables is beyond the scope of this text.

3.9 Functions

Most of the examples thus far contained a single *function*, prefixed with they keyword start and often postfixed with something like **void** → **void**. In this section, we discuss how to write additional functions, which are an essential element of Inger is one wants to write larger programs.

The purpose of a function is to encapsulate part of a program and associate it with a name or *identifier*. Any Inger program consists of at least one function: the *start function*, which is marked with the keyword start. To become familiar with the structure of a function, let us examine the syntax diagram for a function (figure 3.10 and 3.11). The associated BNF is a bit lengthy, so we will not print it here.

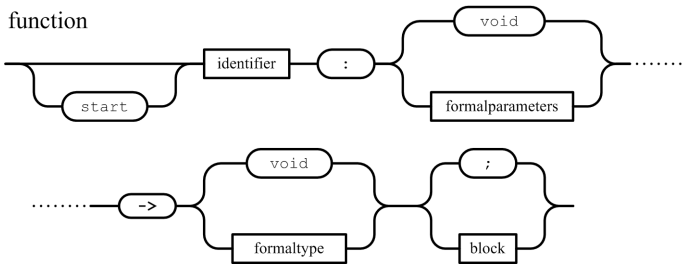


Figure 3.10: Syntax diagram for *function*

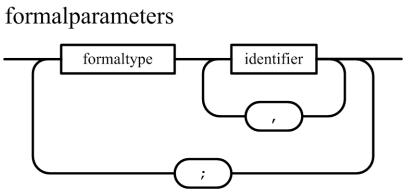


Figure 3.11: Syntax diagram for *formal parameter block*

A function must be declared before it can be used. The declaration does not necessarily have to precede the actual use of the function, but it must take place at some point. The declaration of a function couples an identifier (the function name) to a set of function parameters (which may be empty), a return value (which may be none), and a function body. An example of a function declaration may be found in listing 3.17 (the minmax function).

Function parameters are values given to the function which may influence the way it executes. Compare this to mathematical function definitions: they take an input variable (usually x) and produce a result. The function declarations in Inger are in fact modelled after the style in which mathematical functions are defined. Function parameters must always have names so that the code in the function can refer to them. The return value of a function does not have a name. We will illustrate the declaration of functions with some examples.

Example 3.11 (Function Declarations)

The function f takes no arguments and produces no result. Although such a function may seem useless, it is still possible for it to have a *side effect*, i.e. an influence besides returning a value:

```
f: void → void
```

The function g takes an int and a bool parameter, and returns an int value:

```
g: int a; bool b → int
```

Finally, the function h takes a two-dimensional array of char as an argument, and returns a pointer to an int:

```
h: char str [ ][ ] → int *
```

□

In the previous example, several sample *function headers* were given. Apart from a header, a function must also have a body, which is simply a block of code (contained within braces). From within the *function body*, the programmer may refer to the function parameters as if they were local variables.

Example 3.12 (Function Definition)

Here is sample definition for the function g from the previous example:

```
g: int a; bool b → int
{
    if ( b == true )
       {
        return( a );
    }
    else
    {
        return( -a );
    }
}
```

□

The last example illustrates the use of the return keyword to return from a function call, while at the same time setting the return value. All functions (except functions which return void) must have a return statement somewhere in their code, or their return value may never be set.

Some functions take no parameters at all. This class of functions is called void, and we use the keyword void to identify them. It is also possible that a function has no return value. Again, we use they keyword void to indicate this. There are functions that take no parameters and return nothing: double void.

Now that functions have been defined, they need to be invoked, since that's the reason they exist. The () operator *applies* a function. It must be supplied to call a function, even if that function takes no parameters (void).

Example 3.13 (Function Invocation)

The function f from example 3.11 has no parameters. It is invoked like this:

```
f();
```

Note the use of (), even for a void function. The function g from the same example might be invoked with the following parameters:

```
int result = g( 3, false ); /* -3 */
```

The programmer is free to choose completely different values for the parameters. In this example, constants have been supplied, but it is legal to fill in variables or even complete expressions which can in turn contain function calls:

```
int result = g( g( 3, false ), false ); /* 3 */
```

□

Parameters are always passed *by value*, which means that their value is copied to the target function. If that function changes the value of the parameter, the value of the original variable remains unchanged:

Example 3.14 (By Value vs. By Reference)

Suppose we have the function f, which is defined so:

```
f: int a → void
{
    a = 2;
}
```

To illustrate invocation *by value*, we do this:

```
int i = 1;
f(i);
printint (i); /* 1 */
```

It is impossible to change the value of the input variable i, unless we redefine the function f to accept a pointer:

```
f: int *a → void
{
    *a = 2;
}
```

Now, the address of i is passed *by value*, but still points to the actual memory where i is stored. Thus i can be changed:

```
int i = 1;
f(&i);
printint (i); /* 1 */
```

□

3.10 Modules

Not all code for a program has to reside within the same *module*. A program may consist of multiple modules, one of which is the *main module*, which contains one (and only one) function marked with the keyword start. This is the function that will be executed when the program starts. A start function must always be **void** → **void**, because there is no code that provides it with parameters and no code to receive a return value. There can be only one module with a start

```
/*
 * printint.c
 *
 * Implementation of printint()
 */
void printint ( int x )
{
    printf ( "%d\n", x );
}
```

Listing 3.18: C-implementation of printint Function

```
/*
 *  printint.ih
 *
 *  Header file for printint.c
 */
extern printint : int x → void;
```

Listing 3.19: Inger Header File for printint Function

function. The start function may be called by other functions like any other function.

Data and functions may be shared between modules using the extern keyword. If a variable **int** a is declared in one function, it can be imported by another module with the statement **extern int** a. The same goes for functions. The extern statements are usually placed in a *header file*, with the .ih extension. Such files can be referenced from Inger source code with the #import directive.

In listing 3.18, a C function called printint is defined. We wish to use this function in an Inger program, so we write a header file called printint.ih with contains an extern statement to import the C function (listing 3.19). Finally, the Inger program in listing 3.20 can access the C function by importing the header file with the #import directive.

3.11 Libraries

Unlike other popular programming languages, Inger has no builtin functions (e.g. read, write, sin, cos etc.). The programmer has to write all required functions himself, or import them from a *library*. Inger code can be linked into a static or dynamic using the *linker*. A library consists of one more code modules, all of which do not contain a start function (if one or more of them do, the linker will complain). The compiler not check the existence or nonexistence of start functions, except for printing an error when there is more than one start function in the same module.

Auxiliary functions need not be in an Inger module; they can also be im-

```
/*
 * printint.i
 *
 * Uses C-implementation of
 * printint()
 */
module program;

#import "printint.ih"

int a,b;

start main: void → void
{
    a = b = 1;
    printint ( a + b );
}
```

Listing 3.20: Inger Program Using printint

plemented in the C programming language. In order to use such functions in an Inger program, an Inger header file (.ih) must be provided for the C library, which contains extern function declarations for all the functions used in the Inger program. A good example is the stdio.ih header file supplied with the Inger compiler. This header file is an interface to the ANSI C stdio library.

3.12 Conclusion

This concludes the introduction to the Inger language. Please refer to the appendices, in particular appendices C and D for detailed tables on operator precedence and the BNF productions for the entire language.

Bibliography

[1] B. Kernighan and D. Ritchie: *C Programming Language (2nd Edition)*, Prentice Hall, 1998.

[2] A. C. Hartmann: *A Concurrent Pascal Compiler for Minicomputers*, Lecture notes in computer science, Springer-Verlag, Berlin 1977.

[3] M. Marcotty and H. Ledgard: *The World of Programming Languages*, Springer-Verlag, Berlin 1986., pages 41 and following.

[4] American National Standards Institute: *ANSI X3.159-1989. American National Standard for information systems - Programming Language C*, ANSI, New York, USA 1989.

[5] R. S. Scowen: *Extended BNF a Generic Base Standard*, Final Report, SEG C1 N10 (DITC Software Engineering Group), National Physical Laboratory, Teddington, Middlesex, UK 1993.

[6] W. Waite: *ANSI C Specification*,
http://www.cs.colorado.edu/~eliuser/c_html/c.html

[7] N. Wirth and K. Jensen: *PASCAL User Manual and Report*, Lecture notes in computer science, Springer-Verlag, Berlin 1975.

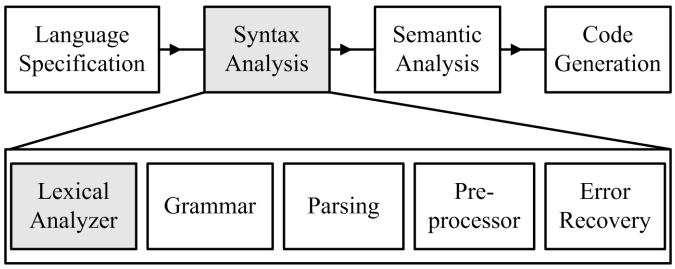
Part II

Syntax

Humans can understand a sentence, spoken in a language, when they hear it (provided they are familiar with the language being spoken). The brain is trained to process the incoming string of words and give meaning to the sentence. This process can only take place if the sentence under consideration obeys the grammatical rules of the language, or else it would be gibberish. This set of rules is called the *syntax* of a language and is denoted using a *grammar*. This part of the book, *syntax analysis*, gives an introduction to formal grammars (notation and manipulation) and how they are used to read (parse) actual sentences in a language. It also discusses ways to vizualize the information gleaned from a sentence in a tree structure (a parse tree). Apart from theoretical aspects, the text treats practical matters such as lexical analysis (breaking a line of text up into individual words and recognizing language keywords among them) and tree traversal.

Chapter 4

Lexical Analyzer



4.1 Introduction

The first step in the compiling process involves reading source code, so that the compiler can check that source code for errors before translating it to, for example, assembly language. All programming languages provide an array of keywords, like IF, WHILE, SWITCH and so on. A compiler is not usually interested in the individual characters that make up these keywords; these keywords are said to be atomic. However, in some cases the compiler does care about the individual characters that make up a word: an integer number (e.g. 12345), a string (e.g. "hello, world") and a floating point number (e.g. 12e-09) are all considered to be words, but the individual characters that make them up are significant.

This distinction requires special processing of the input text, and this special processing is usually moved out of the parser and placed in a module called the *lexical analyzer*, or *lexer* or *scanner* for short. It is the lexer's responsibility to divide the input stream into *tokens* (atomic words). The parser (the module that deals with groups of tokens, checking that their order is valid) requests a token from the lexer, which reads characters from the input stream until it has accumulated enough characters to form a complete token, which it returns to the parser.

Example 4.1 (Tokenizing)

Given the input

```
the quick brown fox jumps over the lazy dog
```

a lexer will split this into the tokens

the, quick, brown, fox, jumps, over, the, lazy and dog

□

The process of splitting input into tokens is called *tokenizing* or *scanning*. Apart from tokenizing a sentence, a lexer can also divide tokens up into classes. This is process is called *screening*. Consider the following example:

Example 4.2 (Token Classes)

Given the input

the sum of 2 + 2 = 4.

a lexer will split this into the following tokens, with classes:

```
Word:   the
Word:   sum
Word:   of
Number: 2
Plus:   +
Number: 2
Equals: =
Number: 4
Dot:    .
```

□

Some token classes are very narrow (containing only one token), while others are broad. For example, the token class Word is used to represent the, sum and of, while the token class Dot can only be used when a . is read. Incidentally, the lexical analyzer must know how to separate individual tokens. In program source text, keywords are usually separated by whitespace (spaces, tabs and line feeds). However, this is not always the case. Consider the following input:

Example 4.3 (Token Separation)

Given the input

```
sum=(2+2)*3;
```

a lexer will split this into the following tokens:

sum, =, (, 2, +, 2,), *, 3 and ;

□

Those familiar with popular programming languages like C or Pascal may know that mathematical tokens like numbers, =, + and * are not required to be separated from each other by whitespace. The lexer must have some way to know when a token ends and the next token (if any) begins. In the next section,

we will discuss the theory of regular languages to further clarify this point and how lexers deal with it.

Lexers have an additional interesting property: they can be used to filter out input that is not important to the parser, so that the parser has less different tokens to deal with. Block comments and line comments are examples of uninteresting input.

A token class may represent a (large) collection of values. The token class OP_MULTIPLY, representing the multiplication operator * contains only one token (*), but the token class LITERAL_INTEGER can represents the collection of all integers. We say that 2 is an integer, and so is 256, 381 and so on. A compiler is not only interested in the fact that a token is a literal integer, but also in the value of that literal integer. This is why tokens are often accompanied by a *token value*. In the case of the number 2, the token could be LITERAL_INTEGER and the token value could be 2.

Token values can be of many types: an integer number token has a token value of type integer, a floating point number token has a token value of type float or double, and a string token has a token value of type char *. Lexical analyzers therefore often store token values using a *union* (a C construct that allows a data structure to map fields of different type on the same memory, provided that only one of these fields is used at the same time).

4.2 Regular Language Theory

The lexical analyzer is a submodule of the parser. While the parser deals with *context-free grammars* (a higher level of abstraction), the lexer deals with individual characters which form tokens (words). Some tokens are simple (IF or WHILE) while others are complex. All integer numbers, for example, are represented using the same token (INTEGER), which covers many cases (1, 100, 5845 and so on). This requires some notation to match all integer numbers, so they are treated the same.

The answer lies in realizing that the collection of integer numbers is really a small language, with very strict rules (it is a so-called *regular language*). Before we can show what regular languages are, we must must discuss some preliminary definitions first.

A language is a set of rules that says which sentences can be generated by stringing together elements of a certain *alphabet*. An alphabet is a collection of symbols (or entire words) denoted as Σ. The set of all strings that can be generated from an alphabet is denoted Σ^*. A language over an alphabet Σ is a subset of Σ^*.

We now define, without proof, several operations that may be performed on languages. The first operation on languages that we present is the binary *concatenation* operation.

Definition 4.1 (Concatenation operation)

Let X and Y be two languages. Then XY is the concatenation of these languages, so that:

$$XY = \{uv \mid u \in X \ \wedge \ v \in Y\}.$$

■

Concatenation of a language with itself is also possible and is denoted X^2. Concatenation of a string can also be performed multiple times, e.g. X^7. We will illustrate the definition of concatenation with an example.

Example 4.4 (Concatenation operation)

Let Σ be the alphabet $\{a, b, c\}$.
Let X be the language over Σ with $X = \{aa, bb\}$.
Let Y be the language over Σ with $Y = \{ca, b\}$.
Then XY is the language $\{aaca, aab, bbca, bbb\}$.

□

The second operation that will need to define regular languages is the binary *union* operation.

Definition 4.2 (Union operation)

Let X and Y be two languages. Then $X \cup Y$ is the union of these languages with

$$X \cup Y = \{u \mid u \in X \ \vee \ u \in Y\}.$$

■

Note that the priority of concatenation is higher than the priority of union. Here is an example that shows how the union operation works:

Example 4.5 (Union operation)

Let Σ be the alphabet $\{a, b, c\}$.
Let X be the language over $\Sigma\{aa, bb\}$.
Let Y be the language over $\Sigma\{ca, b\}$.
Then $X \cup Y$ is the language over $\Sigma\{aa, bb, ca, b\}$.

□

The final operation that we need to define is the unary *Kleene star* operation.[1]

Definition 4.3 (Kleene star)

Let X be a language. Then

$$X^* = \bigcup_{i=0}^{\infty} X^i \tag{4.1}$$

[1]The mathematician Stephen Cole Kleene was born in 1909 in Hartford, Connecticut. His research was on the theory of algorithms and recursive functions. According to Robert Soare, "From the 1930's on Kleene more than any other mathematician developed the notions of computability and effective process in all their forms both abstract and concrete, both mathematical and philosophical. He tended to lay the foundations for an area and then move on to the next, as each successive one blossomed into a major research area in his wake." Kleene died in 1994.

■

Or, in words: X^* means that you can take 0 or more sentences from X and concatenate them. The Kleene star operation is best clarified with an example.

Example 4.6 (Kleene star)

Let Σ be the alphabet $\{a, b\}$.
Let X be the language over $\Sigma\{aa, bb\}$.
Then X^* is the language $\{\lambda, aa, bb, aaaa, aabb, bbaa, bbbb, \ldots\}$.

□

There is also an extension to the Kleene star. XX^* may be written X^+, meaning that at least one string from X must be taken (whereas X^* allows the empty string λ).

With these definitions, we can now give a definition for a regular language.

Definition 4.4 (Regular languages)

1. Basis: $\emptyset$, $\{\lambda\}$ and $\{a\}$ are regular languages.
2. Recursive step: Let X and Y be regular languages. Then

   $X \cup Y$ is a regular language
   XY is a regular language
   X^* is a regular language

■

Now that we have established what regular languages are, it is important to note that lexical analyzer generators (software tools that will be discussed below) use *regular expressions* to denote regular languages. Regular expressions are merely another way of writing down regular languages. In regular expressions, it is customary to write the language consisting of a single string composed of one word, $\{a\}$, as $\boldsymbol{a}$.

Definition 4.5 (Regular expressions)

Using recursion, we can define regular expressions as follows:

1. Basis: $\boldsymbol{\emptyset}$, $\boldsymbol{\lambda}$ and $\boldsymbol{a}$ are regular expressions.
2. Recursive step: Let X and Y be regular expressions. Then

   $X \cup Y$ is a regular expression
   XY is a regular expression
   X^* is a regular expression

■

As you can see, the definition of regular expressions differs from the definition of regular languages only by a notational convenience (less braces to write).

So any language that can be composed of other regular languages or expressions using concatenation, union, and the Kleene star, is also a regular language or expression. Note that the priority of the concatenation, Kleene Star and union operations are listed here from highest to lowest priority.

Example 4.7 (Regular Expression)

Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be regular expressions by definition 4.5(1). Then $\boldsymbol{ab}$ is a regular expression by definition 4.5(2) through concatenation. $(\boldsymbol{ab} \cup \boldsymbol{b})$ is a regular expression by definition 4.5(2) through union. $(\boldsymbol{ab} \cup \boldsymbol{b})^*$ is a regular expression by definition 4.5(2) through union. The sentences that can be generated by $(\boldsymbol{ab} \cup \boldsymbol{b})^*$ are $\{\lambda, ab, b, abb, bab, babab, \ldots\}$.

□

While context-free grammars are normally denoted using production rules, for regular languages it is sufficient to use the easy to read regular expressions.

4.3 Sample Regular Expressions

In this section, we present a number of sample regular expressions to illustrate the theory presented in the previous section. From now on, we will now longer use bold $\boldsymbol{a}$ to denote $\{a\}$, since we will soon move to UNIX regular expressions which do not use bold either.

Regular expression	**Sentences generated**
q	the sentence q
qqq	the sentence qqq
q^*	all sentences of 0 or more q's
q^+	all sentences of 1 or more q's
$q \cup \lambda$	the empty sentence or q. Often denoted as q? (see UNIX regular expressions).
$b^*((b^+a \cup \lambda)b^*$	the collection of sentences that begin with 0 or more b's, followed by either one or more b's followed by an a, or nothing, followed by 0 or more b's.

These examples show that through repeated application of definition 4.5, complex sequences can be defined. This feature of regular expressions is used for constructing lexical analyzers.

4.4 UNIX Regular Expressions

Under UNIX, several extensions to regular expressions have been implemented that we can use. A UNIX regular expression[3] is commonly called a *regex* (multiple: *regexes*).

There is no union operator on UNIX. Instead, we supply a list of alternatives contained within square brackets.

$$\texttt{[abc]} \equiv (a \cup b \cup c)$$

To avoid having to type in all the individual letters when we want to match all lowercase letters, the following syntax is allowed:

$$\texttt{[a-z]} \equiv \texttt{[abcdefghijklmnopqrstuvwxyz]}$$

UNIX does not have a λ either. Here is the alternative syntax:

$\texttt{a?} \equiv a \cup \lambda$

Lexical analyzer *generators* allow the user to directly specify these regular expressions in order to identify lexical tokens (atomic words that string together to make sentences). We will discuss such a generator program shortly.

4.5 States

With the theory of regular languages, we can now find out how a lexical analyzer works. More specifically, we can see how the scanner can divide the input (34+12) into separate tokens.

Suppose the programming language for which we wish to write a scanner consists only of sentences of the form (*number*+*number*). Then we require the following regular expressions to define the tokens.

Token	**Regular expression**
(	(
)	)
+	+
number	[0-9]+

A lexer uses states to determine which characters it can expect, and which may not occur in a certain situation. For simple tokens ((,) and +) this is easy: either one of these characters is read or it is not. For the *number* token, states are required.

As soon as the first digit of a number is read, the lexer enters a state in which it expects more digits, and nothing else. If another digit is read, the lexer remains in this state and adds the digit to the token read so far. It something else (not a digit) is read, the lexer knows the *number* token is finished and leaves the *number* state, returning the token to the caller (usually the parser). After that, it tries to match the unexpected character (maybe a +) to another token.

Example 4.8 (States)

Let the input be (34+12). The lexer starts out in the *base* state. For every character read from the input, the following table shows the state that the lexer is currently in and the action it performs.

□

Token read	**State**	**Action taken**
(	*base*	Return (to caller
3	*base*	Save 3, enter *number* state
4	*number*	Save 4
+	*number*	+ not expected. Leave *number* state and return 34 to caller
+	*base*	Return + to caller
1	*base*	Save 1, enter *number* state
2	*number*	Save 2
)	*number*	) unexpected. Leave *number* state and return 12 to caller
)	*base*	return) to caller

This example did not include whitespace (spaces, line feeds and tabs) on purpose, since it tends to be confusing. Most scanners ignore spacing by matching it with a special regular expression and doing nothing.

There is another rule of thumb used by lexical analyzer generators (see the discussion of this software below): they always try to return the longest token possible.

Example 4.9 (Token Length)

= and == are both tokens. Now if = was read and the next character is also = then == will be returned instead of two times =.

□

In summary, a lexer determines which characters are valid in the input at any given time through a set of states, on of which is the active state. Different states have different valid characters in the input stream. Some characters cause the lexer to shift from its current state into another state.

4.6 Common Regular Expressions

This section discusses some commonly used regular expressions for interesting tokens, such as strings and comments.

Integer numbers

An integer number consists of only digits. It ends when a non-digit character is encountered. The scanner must watch out for an overflow, e.g. `12345678901234` does not fit in most programming languages' type systems and should cause the scanner to generate an overflow error.

The regular expression for integer numbers is

```
[0-9]+
```

This regular expression generates the collection of strings containing at least one digit, and nothing but digits.

Practical advice 4.1 (Lexer Overflow)

If the scanner generates an overflow or similar error, parsing of the source code can continue (but no target code can be generated). The scanner can just replace the faulty value with a correct one, e.g. "1".

■

Floating point numbers

Floating point numbers have a slightly more complex syntax than integer numbers. Here are some examples of floating point numbers:

`1.0`, `.001`, `1e-09`, `.2e+5`

The regular expression for floating point numbers is:

```
[0-9]* . [0-9]+ ( e [+-] [0-9]+ )?
```

Spaces were added for readability. These are not part of the generated strings. The scanner should check each of the subparts of the regular expression containing digits for possible overflow.

Practical advice 4.2 (Long Regular Expressions)

If a regular expression becomes long or too complex, it is possible to split it up into multiple regular expressions. The lexical analyzer's internal state machine will still work.

■

Strings

Strings are a token type that requires some special processing by the lexer. This should become clear when we consider the following sample input:

```
"3+4"
```

Even though this input consists of numbers, and the + operator, which may have regular expressions of their own, the entire expression should be returned to the caller since it is contained within double quotes. The trick to do this is to introduce another state to the lexical analyzer, called an *exclusive state*. When in this state, the lexer will process only regular expressions marked with this state. The resulting regular expressions are these:

Regular expression	Action
`"`	Enter *string* state
string `.`	Store character. A dot (`.`) means anything. This regular expression is only considered when the lexer is in the *string* state.
string `"`	Return to previous state. Return string contents to caller. This regular expression is only considered when the lexer is in the *string* state.

Practical advice 4.3 (Exclusive States)

You can write code for exclusive states yourself (when writing a lexical analyzer from scratch), but AT&T lex and GNU flex can do it for you.

■

The regular expressions proposed above for strings do not heed line feeds. You may want to disallow line feeds within strings, though. Then you must add another regular expressions that matches the line feed character (\n in some languages) and generates an error when it is encountered within a string.

The lexer writer must also be wary of a buffer overflow; if the program source code consists of a `"` and hundreds of thousands of letters (at least, not another `"`), a compiler that does not check for buffer overflow conditions will eventually crash for lack of memory. Note that you could match strings using a single regular expression:

```
"(.)*"
```

but the state approach makes it much easier to check for buffer overflow conditions since you can decide at any time whether the current character must be stored or not.

Practical advice 4.4 (String Limits)

To avoid a buffer overflow, limit the string length to about 64 KB and generate an error if more characters are read. Skip all the offending characters until another " is read (or end of file).

■

Comments

Most compilers place the job of filtering comments out of the source code with the lexical analyzer. We can therefore create some regular expressions that do just that. This once again requires the use of an exclusive state. In programming languages, the beginning and end of comments are usually clearly marked:

Language	Comment style
C	`/* comment */`
C++	`// comment (line feed)`
Pascal	`{ comment }`
BASIC	`REM comment :`

We can build our regular expressions around these delimiters. Let's build sample expressions using the C comment delimiters:

Regular expression	Action
`/*`	Enter *comment* state
comment `.`	Ignore character. A dot (`.`) means anything. This regular expression is only considered when the lexer is in the *comment* state.
comment `*/`	Return to previous state. Do not return to caller but read next token, effectively ignoring the comment. This regular expression is only considered when the lexer is in the *comment* state.

Using a minor modification, we can also allow nested comments. To do this, we must have the lexer keep track of the comment nesting level. Only when the nesting level reaches 0 after leaving the final comment should the lexer leave the *comment* state. Note that you could handle comments using a single regular expression:

```
/* (.)* */
```

But this approach does not support nested comments. The treatment of line comments is slightly easier. Only one regular expression is needed:

```
//(.)*\n
```

4.7 Lexical Analyzer Generators

Although it is certainly possible to write a lexical analyzer by hand, this task becomes increasingly complex as your input language gets richer. It is therefore more practical use a lexical analyzer generator. The code generated by such a generator program is usually faster and more efficient that any code you might write by hand[2].

Here are several candidates you could use:

AT&T lex	Not free, ancient, UNIX and Linux implementations
GNU flex	Free, modern, Linux implementation
Bumblebee lex	Free, modern, Windows implementation

The *Inger* compiler was constructed using GNU flex; in the next sections we will briefly discuss its syntax (since flex takes lexical analyzer specifications as its input) and how to use the output flex generates.

Practical advice 4.5 (Lex)

We heard that some people think that a lexical analyzer must be written in lex or flex in order to be called a *lexer*. Of course, this is blatant nonsense (it is the other way around).

■

Flex syntax

The layout of a flex input file (extension `.l`) is, in pseudocode:

```
%{
    Any preliminary C code (inclusions, defines) that
    will be pasted in the resulting .C file
%}
Any flex definitions
%%
Regular expressions
%%
Any C code that will be appended to
the resulting .C file
```

When a regular expression matches some input text, the lexical analyzer must execute an action. This usually involves informing the caller (the parser) of the token class found. With an action included, the regular expressions take the following form:

```
[0-9]+ {
            intValue_g = atoi( yytext );
            return( INTEGER );
        }
```

Using `return( INTEGER )`, the lexer informs the caller (the parser) that is has found an integer. It can only return one item (the token class) so the actual value of the integer is passed to the parser through the global variable `intValue_g`. Flex automatically stores the characters that make up the current token in the global string `yytext`.

Sample flex input file

Here is a sample flex input file for the language that consists of sentences of the form (*number*+*number*), and that allows spacing anywhere (except within tokens).

```
%{
    #define NUMBER 1000
    int intValue_g;
%}
%%
"("         { return( '(' ); }
")"         { return( ')' ); }
"+"         { return( '+' ); }
[0-9]+      {
                intValue_g = atoi( yytext );
                return( NUMBER );
            }
%%
int main()
{
    int result;
    while( ( result = yylex() ) != 0 )
    {
        printf( "Token class found: %d\n", result );
    }
    return( 0 );
}
```

For many more examples, consult J. Levine's *Lex and yacc* [2].

4.8 Inger Lexical Analyzer Specification

As a practical example, we will now discuss the token categories in the *Inger* language, and all regular expressions used for complex tokens. The full source for the *Inger* lexer is included in appendix F.

Inger discerns several token categories: keywords (IF, WHILE and so on), operators (+, % and more), complex tokens (integer numbers, floating point numbers, and strings), delimiters (parentheses, brackets) and whitespace.

We will list the tokens in each category and show which regular expressions is used to match them.

Keywords

Inger expects all keywords (sometimes called *reserved words*) to be written in lowercase, allowing the literal keyword to be used to match the keyword itself. The following table illustrates this:

Token	Regular Expression	Token identifier
break	break	KW_BREAK
case	case	KW_CASE
continue	continue	KW_CONTINUE
default	default	KW_DEFAULT
do	do	KW_DO
else	else	KW_ELSE
false	false	KW_FALSE
goto_considered _harmful	goto_considered _harmful	KW_GOTO
if	if	KW_IF
label	label	KW_LABEL
module	module	KW_MODULE
return	return	KW_RETURN
start	start	KW_START
switch	switch	KW_SWITCH
true	true	KW_TRUE
while	while	KW_WHILE

Types

Type names are also tokens. They are invariable and can therefore be matched using their full name.

Token	Regular Expression	Token identifier
bool	bool	KW_BOOL
char	char	KW_CHAR
float	float	KW_FLOAT
int	int	KW_INT
untyped	untyped	KW_UNTYPED

Note that the *untyped* type is equivalent to void in the C language; it is a polymorphic type. One or more reference symbols (*) must be added after the untyped keyword. For instance, the declaration

```
untyped ** a;
```

declares a to be a double polymorphic pointer.

Complex tokens

Inger's complex tokens variable identifiers, integer literals, floating point literals and character literals.

Token	Regular Expression	Token identifier
integer literal	`[0-9]+`	INT
identifier	`[_A-Za-z][_A-Za-z0-9]*`	IDENTIFIER
float	`[0-9]*\.[0-9]+([eE][\+-][0-9]+)?`	FLOAT
char	`\'.\'`	CHAR

Strings

In Inger, strings cannot span multiple lines. Strings are read using and exlusive lexer *string* state. This is best illustrated by some flex code:

```
\"                  { BEGIN STATE_STRING; }
<STATE_STRING>\"    { BEGIN 0; return( STRING ); }
<STATE_STRING>\n    { ERROR( "unterminated string" ); }
<STATE_STRING>.     { (store a character) }
<STATE_STRING>\\\"  { (add " to string)   }
```

If a linefeed is encountered while reading a string, the lexer displays an error message, since strings may not span lines. Every character that is read while in the string state is added to the string, except `"`, which terminates a string and causes the lexer to leave the exclusive `string` state. Using the `\"` control code, the programmer can actually add the " (double quotes) character to a string.

Comments

Inger supports two types of comments: line comments (which are terminated by a line feed) and block comments (which must be explicitly terminated). Line comments can be read (and subsequently skipped) using a single regular expression:

```
"//"[^\n]*
```

whereas block comments need an exclusive lexer state (since they can also be nested). We illustrate this again using some flex code:

```
/*                  { BEGIN STATE_COMMENTS;
                      ++commentlevel; }
<STATE_COMMENTS>"/*"  { ++commentlevel; }
<STATE_COMMENTS>.     { }
<STATE_COMMENTS>\n    { }
<STATE_COMMENTS>"*/"  { if( --commentlevel == 0 )
                      BEGIN 0; }
```

Once a comment is started using `/*`, the lexer sets the comment level to 1 and enters the comment state. The comment level is increased every time a `/*` is encountered, and decreased every time a `*/` is read. While in comment state, all characters but the comment start and end delimiters are discarded. The lexer leaves the comment state after the last comment block terminates.

Operators

Inger provides a large selection of operators, of varying priority. They are listed here in alphabetic order of the token identifiers. This list includes only atomic operators, not operators that delimit their argument on both sides, like function application.

funcname `(` *expr[,expr...]* `)`

or array indexing

arrayname `[` *index* `]`.

In the next section, we will present a list of all operators (including function application and array indexing) sorted by priority.

Some operators consist of multiple characters. The lexer can discern between the two by looking one character ahead in the input stream and switching states (as explained in section 4.5.

Token	Regular Expression	Token identifier
addition	`+`	`OP_ADD`
assignment	`=`	`OP_ASSIGN`
bitwise and	`&`	`OP_BITWISE_AND`
bitwise complement	`~`	`OP_BITWISE_COMPLEMENT`
bitwise left shift	`<<`	`OP_BITWISE_LSHIFT`
bitwise or	`|`	`OP_BITWISE_OR`
bitwise right shift	`>>`	`OP_BITWISE_RSHIFT`
bitwise xor	`^`	`OP_BITWISE_XOR`
division	`/`	`OP_DIVIDE`
equality	`==`	`OP_EQUAL`
greater than	`>`	`OP_GREATER`
greater or equal	`>=`	`OP_GREATEREQUAL`
less than	`<`	`OP_LESS`
less or equal	`<=`	`OP_LESSEQUAL`
logical and	`&&`	`OP_LOGICAL_AND`
logical or	`||`	`OP_LOGICAL_OR`
modulus	`%`	`OP_MODULUS`
multiplication	`*`	`OP_MULTIPLY`
logical negation	`!`	`OP_NOT`
inequality	`!=`	`OP_NOTEQUAL`
subtract	`-`	`OP_SUBTRACT`
ternary if	`?`	`OP_TERNARY_IF`

Note that the `*` operator is also used for dereferencing (in unary form) besides multiplication, and the `&` operator is also used for indirection besides bitwise and.

Delimiters

Inger has a number of delimiters. There are listed here by there function description.

Token	Regexp	Token identifier
precedes function return type	`->`	ARROW
start code block	`{`	LBRACE
end code block	`}`	RBRACE
begin array index	`[`	LBRACKET
end array index	`]`	RBRACKET
start function parameter list	`:`	COLON
function argument separation	`,`	COMMA
expression priority, function application	`(`	LPAREN
expression priority, function application	`)`	RPAREN
statement terminator	`;`	SEMICOLON

The full source to the Inger lexical analyzer is included in appendix F.

Bibliography

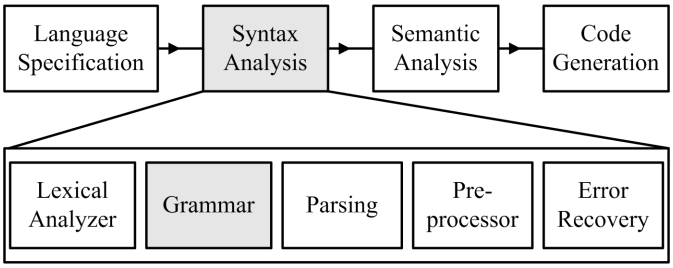
[1] G. Goos, J. Hartmanis: *Compiler Construction - An Advanced Course*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1974.

[2] J. Levine: *Lex and Yacc*, O'Reilly & sons, 2000

[3] H. Spencer: *POSIX 1003.2 regular expressions*, UNIX man page regex(7), 1994

Chapter 5

Grammar



5.1 Introduction

This chapter will introduce the concepts of *language* and *grammar* in both informal and formal terms. After we have established exactly what a grammar is, we offer several example grammars with documentation.

This introductory section discusses the value of the material that follows in writing a compiler. A compiler can be thought of as a sequence of actions, performed on some code (formulated in the source language) that transform that code into the desired output. For example, a Pascal compiler transforms Pascal code to assembly code, and a Java compiler transforms Java code to its corresponding Java bytecode.

If you have used a compiler in the past, you may be familiar with "syntax errors". These occur when the input code does not conform to a set of rules set by the language specification. You may have forgotten to terminate a statement with a semicolon, or you may have used the THEN keyword in a C program (the C language defines no THEN keyword).

One of the things that a compiler does when transforming source code to target code is check the structure of the source code. This is a required step before the compiler can move on to something else.

The first thing we must do when writing a compiler is write a grammar for the source language. This chapter explains what a grammar is, how to create one. Furthermore, it introduces several common ways of writing down a grammar.

5.2 Languages

In this section we will try to formalize the concept of a *language*. When thinking of languages, the first languages that usually come to mind are *natural languages* like English or French. This is a class of languages that we will only consider in passing here, since they are very difficult to understand by a computer. There is another class of languages, the *computer* or *formal* languages, that are far easier to parse since they obey a rigid set of rules. This is in constrast with natural languages, whose leniant rules allow the speaker a great deal of freedom in expressing himself.

Computers have been and are actively used to translate natural languages, both for professional purposes (for example, voice-operated computers or Microsoft SQL Server's English Query) and in games. This first so-called *adventure game*[1] was written as early as 1975 and it was played by typing in English commands.

All languages draw the words that they allow to be used from a pool of words, called the *alphabet*. This is rather confusing, because we tend to think of the alphabet as the 26 latin letters, A through Z. However, the definition of a language is not concerned with how its most basic elements, the words, are constructed from individual letters, but how these words are strung together. In definitions, an alphabet is denoted as Σ.

A language is a collection of sentences or *strings*. From all the words that a language allows, many sentences can be built but only some of these sentences are valid for the language under consideration. All the sentences that may be constructed from an alphabet Σ are denoted Σ^*. Also, there exists a special sentence: the sentence with no words in it. This sentence is denoted λ.

In definitions, we refer to words using lowercase letters at the beginning of our alphabet $(a, b, c...)$, while we refer to sentences using letters near the end of our alphabet $(u, v, w, x...)$. We will now define how sentences may be built from words.

Definition 5.1 (Alphabet)

Let Σ be an alphabet. Σ^*, the set of strings over Σ, is defined recursively as follows:

1. Basis: $\lambda \in \Sigma^*$.
2. Recursive step: If $w \in \Sigma^*$, then $wa \in \Sigma^*$.
3. Closure: $w \in \Sigma^*$ only if it can be obtained from λ by a finite number of applications of the recursive step.

■

[1] In early 1977, Adventure swept the ARPAnet. Willie Crowther was the original author, but Don Woods greatly expanded the game and unleashed it on an unsuspecting network. When Adventure arrived at MIT, the reaction was typical: after everybody spent a lot of time doing nothing but solving the game (it's estimated that Adventure set the entire computer industry back two weeks), the true lunatics began to think about how they could do it better [proceeding to write Zork] (Tim Anderson, "The History of Zork – First in a Series" New Zork Times; Winter 1985)

This definition may need some explanation. It is put using *induction*. What this means will become clear in a moment.

In the *basis* (line 1 of the defintion), we state that the empty string (λ) is a sentence over Σ. This is a statement, not proof. We just state that for any alphabet Σ, the empy string λ is among the sentences that may be constructed from it.

In the *recursive step* (line 2 of the definition), we state that given a string w that is part of Σ^*, the string wa is also part of Σ^*. Note that w denotes a string, and a denotes a single word. Therefore what we mean is that given a string generated from the alphabet, we may append any word from that alphabet to it and the resulting string will still be part of the set of strings that can be generated from the alphabet.

Finally, in the *closure* (line 3 of the definition), we add that all the strings that can be built using the basis and recursive step are part of the set of strings over Σ^*, and all the other strings are not. You can think of this as a sort of safeguard for the definition. In most inductive defintions, we will leave the closure line out.

Is Σ^*, then, a language? The answer is no. Σ^* is the set of all possible strings that may be built using the alphabet Σ. Only some of these strings are actually valid for a language. Therefore a language over an alphabet Σ is a subset of Σ^*.

As an example, consider a small part of the English language, with the alphabet { 'dog', 'bone,', the', 'eats' } (we cannot consider the actual English language, as it has far too many words to list here). From this alphabet, we can derive strings using definition 5.1:

λ
dog
dog dog dog
bone dog the
the dog eats the bone
the bone eats the dog

Many more strings are possible, but we can at least see that most of the strings above are not valid for the English language: their structure does not obey the rules of English grammar. Thus we may conclude that a language over an alphabet Σ is a subset of Σ^* *that follows certain grammar rules*.

If you are wondering how all this relates to compiler construction, you should realize that one of the things that a compiler does is check the structure of its input by applying grammar rules. If the structure is off, the compiler prints a syntax error.

5.3 Syntax and Semantics

To illustrate the concept of grammar, let us examine the following line of text:

```
jumps the fox the dog over
```

Since it obviously does not obey the rules of English grammar, this sentence is meaningless. It is said to be *syntactically incorrect*. The *syntax* of a sentence is

its form or structure. Every sentence in a language must obey to that language's syntax for it to have meaning.

Here is another example of a sentence, whose meaning is unclear:

```
the fox drinks the color red
```

Though possibly considered a wondrous statement in Vogon poetry, this statement has no meaning in the English language. We know that the color red cannot be drunk, so that although the sentence is syntactically correct, it conveys no useful information, and is therefore considered incorrect. Sentences whose structure conforms to a language's syntax but whose meaning cannot be understood are said to be *semantically* incorrect.

The purpose of a grammar is to give a set of rules which all the sentences of a certain language must follow. It should be noted that speakers of a natural language (like English or French) will generally understand sentences that differ from these rules, but this is not so for formal languages used by computers. All sentences are required to adhere to the grammar rules without deviation for a computer to be able to understand them.

In *Compilers: Principles, Techniques and Tools* ([1]), Aho defines a grammar more formally:

> *A grammar is a formal device for specifying a potentially infinite language (set of strings) in a finite way.*

Because language semantics are hard to express in a set of rules (although we will show a way to deal with sematics in part III), rammars deal with syntax only: a grammar defines the structure of sentences in a language.

5.4 Production Rules

In a grammar, we are not usually interested in the individual letters that make up a word, but in the words themselves. We can give these words names so that we can refer to them in a grammar. For example, there are very many words that can be the subject or object of an English sentence ('fox', 'dog', 'chair' and so on) and it would not be feasible to list them all. Therefore we simply refer to them as 'noun'. In the same way we can give all words that precede nouns to add extra information (like 'brown', 'lazy' and 'small') a name too: 'adjective'. We call the set of all articles ('the', 'a', 'an') 'article'. Finally, we call all verbs 'verb'. Each of these names represent a set of many words, with the exception of 'article', which has all its members already listed.

Armed with this new terminology, we are now in a position to describe the form of a very simple English sentence:

sentence: article adjective noun verb adjective noun.

From this lone *production rule*, we can generate (produce) English sentences. We can replace every set name to the right of the colon with one of its elements. For example, we can replace article with 'the', adjective with 'quick', noun with 'fox' and so on. This way we can build sentences such as

```
the quick fox eats a delicious banana
the delicious banana thinks the quick fox
a quick banana outruns a delicious fox
```

The structure of these sentences matches the preceding rule, which means that they conform to the syntax we specified. Incidentally, some of these sentences have no real meaning, thus illustrating that semantic rules are not included in the grammar rules we discuss here.

We have just defined a grammar, even though it contains only one rule that allows only one type of sentence. Note that our grammar is a so-called *abstract grammar*, since it does not specify the actual words that we may use to replace the word classes (article, noun, verb) that we introduced.

So far we have given names to classes of individual words. We can also assign names to common combinations of words. This requires multiple rules, making the individual rules simpler:

sentence: object verb object.
object: article adjective noun.

This grammar generates the same sentences as the previous one, but is somewhat easier to read. Now we will also limit the choices that we can make when replacing word classes by introducing some more rules:

noun: **fox**.
noun: **banana**.
verb: **eats**.
verb: **thinks**.
verb: **outruns**.
article: **a**.
article: **the**.
adjective: **quick**.
adjective: **delicious**.

Our grammar is now extended so that it is no longer an abstract grammar. The rules above dictate how *nonterminals* (abstract grammar elements like object or article) may be replaced with concrete elements of the language's *alphabet*. The alphabet consists of all a the *terminal symbols* or *terminals* in a language (the actual words). In the productions rules listed above, terminal symbols are printed in bold.

Nonterminal symbols are sometimes called *auxiliary symbols*, because they must be removed from any sentential form in order to create a concrete sentence in the language.

Production rules are called production rules for a reason: they are used to produce concrete sentences from the topmost nonterminal, or *start symbol*. A concrete sentence may be *derived* from the start symbol by systematically selecting nonterminals and replacing them with the right hand side of a suitable production rule. In the listing below, we present the production rules for the grammar we have constructed so far during this chapter. Consult the following example, in which we use this grammar to derive a sentence.

sentence:	object verb object.
object:	article adjective noun.
noun:	**fox**.
noun:	**banana**.
verb:	**eats**.
verb:	**thinks**.
verb:	**outruns**.
article :	**a**.
article :	**the**.
adjective :	**quick**.
adjective :	**delicious**.

Derivation		**Rule Applied**
	'sentence'	
$\Longrightarrow$	'object' verb object	1
$\Longrightarrow$	' article ' adjective noun verb object	2
$\Longrightarrow$	**the** ' adjective ' noun verb object	9
$\Longrightarrow$	**the quick** 'noun' verb object	2
$\Longrightarrow$	**the quickfox** 'verb' object	3
$\Longrightarrow$	**the quickfox eats** 'object'	5
$\Longrightarrow$	**the quickfox eats** ' article ' adjective noun	2
$\Longrightarrow$	**the quickfox eats a** ' adjective ' noun	8
$\Longrightarrow$	**the quickfox eats a delicious** 'noun'	11
$\Longrightarrow$	**the quickfox eats a delicious banana**	4

The symbol $\Longrightarrow$ indicates the application of a production rule, which the rule number of the rule applied in the right column. The set of all sentences which can be derived by repeated application of the production rules (deriving) is the language defined by these production rules.

The string of terminals and nonterminals in each step is called a *sentential form*. The last string, which contains only terminals, is the actual *sentence*. This means that the process of derivation ends once all nonterminals have been replaced with terminals.

You may have noticed that in every step, we consequently replaced the *leftmost* nonterminal in the sentential form with one of its productions. This is why the derivation we have performed is called a *leftmost derivation*. It is also correct to perform a rightmost derivation by consequently replacing the rightmost nonterminal in each sentential form, or any derivation in between.

Our current grammar states that every noun is preceded by precisely one adjective. We now want to modify our grammar so that it allows us to specify zero, one or more adjectives before each noun. This can be done by introducing *recursion*, where a production rule for a nonterminal may again contain that nonterminal:

object:	article adjectivelist noun.
adjectivelist :	adjective adjectivelist .
adjectivelist :	ϵ.

The rule for the nonterminal object has been altered to include adjectivelist

instead of simply adjective. An adjective list can either be empty (nothing, indicated by ϵ), or an adjective, followed by another adjective list and so on. The following sentences may now be derived:

	sentence
$\Longrightarrow$	*object* verb object
$\Longrightarrow$	*article* adjectivelist noun verb object
$\Longrightarrow$	**the** *adjectivelist* noun verb object
$\Longrightarrow$	**the** *noun*verb object
$\Longrightarrow$	**the banana***verb*object
$\Longrightarrow$	**the bananaoutruns***object*
$\Longrightarrow$	**the bananaoutruns***article* adjectivelist noun
$\Longrightarrow$	**the bananaoutrunsthe** *adjectivelist* noun
$\Longrightarrow$	**the bananaoutrunsthe***adjective* adjectivelist noun
$\Longrightarrow$	**the bananaoutrunsthequick** *adjectivelist* noun
$\Longrightarrow$	**the bananaoutrunsthequick***adjective* adjectivelist noun
$\Longrightarrow$	**the bananaoutrunsthequickdelicious** *adjectivelist* noun
$\Longrightarrow$	**the bananaoutrunsthequickdeliciousnoun**
$\Longrightarrow$	**the bananaoutrunsthequickdeliciousfox**

5.5 Context-free Grammars

After the introductory examples of sentence derivations, it is time to deal with some formalisms. All the grammars we will work with in this book are *context-free grammars*:

Definition 5.2 (Context-Free Grammar)

A *context-free grammar* is a quadruple (V, Σ, P, S) where V is a finite set of variables (nonterminals), Σ is a finite set of terminals, P is a finite set of production rules and $S \in V$ is an element of V designated as the *start symbol*.

■

The grammar listings you have seen so far were context-free grammars, consisting of a single nonterminal on the left-hand side of each production rule (the mark of a context-free grammar). In fact, a symbol is a nonterminal only when it acts as the left-hand side of a production rule. The right side of every production rule may either be empty (denoted using an epsilon, ϵ), or contain any combination of terminals and nonterminals. This notation is called the *Backus-Naur form*[2], after its inventors, John Backus and Peter Naur.

[2] John Backus and Peter Naur introduced for the first time a formal notation to describe the syntax of a given language (this was for the description of the ALGOL 60 programming language). To be precise, most of BNF was introduced by Backus in a report presented at an earlier UNESCO conference on ALGOL 58. Few read the report, but when Peter Naur read it he was surprised at some of the differences he found between his and Backus's interpretation of ALGOL 58. He decided that for the successor to ALGOL, all participants of the first design had come to recognize some weaknesses, should be given in a similar form so that all participants should be aware of what they were agreeing to. He made a few modificiations that are almost universally used and drew up on his own the BNF for ALGOL 60 at the meeting where it was designed. Depending on how you attribute presenting it to the world, it was either by Backus in 59 or Naur in 60. (For more details on this period of programming languages history, see the introduction to Backus's Turing award article in Communications of the ACM, Vol. 21, No. 8, august 1978. This note was suggested by William B. Clodius from Los Alamos Natl. Lab).

```
expression:    expression + expression.
expression:    expression - expression.
expression:    expression * expression.
expression:    expression / expression.
expression:    expression ^ expression.
expression:    number.
expression:    ( expression ) .
number:        0.
number:        1.
number:        2.
number:        3.
number:        4.
number:        5.
number:        6.
number:        7.
number:        8.
number:        9.
```

Listing 5.1: Sample Expression Language

The process of deriving a valid sentence from the start symbol (in our previous examples, this was sentence), is executed by repeatedly replacing a nonterminal by the right-hand side of any one of the production rules of which it acts as the left-hand side, until no nonterminals are left in the *sentential form*. Nonterminals are always abstract names, while terminals are often expressed using their actual (real-world) representations, often between quotes (e.g. "+", "while", "true") or printed bold (like we do in this book).

The left-hand side of a production rule is separated from the right-hand side by a colon, and every production rule is terminated by a period. Whether you do this does not affect the meaning of the production rules at all, but is considered good style and part of the specificiation of Backus Naur Form (BNF). Other notations are used.

As a running example, we will work with a simple language for mathematical expressions, analogous to the language discussed in the introduction to this book. The language is capable of expressing the following types of sentences:

```
1 + 2 * 3 + 4
2 ^ 3 ^ 2
2 * (1 + 3)
```

In listing 5.1, we give a sample context-free grammar for this language.

Note the periods that terminate each production rule. You can see that there are only two nonterminals, each of which have a number of alternative production rules associated with them. We now state that expression will be the distinguished nonterminal that will act as the start symbol, and we can use these production rules to derive the sentence `1 + 2 * 3` (see table 5.5).

	expression
$\Longrightarrow$	*expression* $*$ expression
$\Longrightarrow$	*expression* $+$expression $*$ expression
$\Longrightarrow$	*number*$+$expression $*$ expression
$\Longrightarrow$	**1** $+$*expression* $*$ expression
$\Longrightarrow$	**1** $+$*number*$*$ expression
$\Longrightarrow$	**1** $+$**2** $*$ *expression*
$\Longrightarrow$	**1** $+$**2** $*$ *number*
$\Longrightarrow$	**1** $+$**2** $*$ **3**

Table 5.1: Derivation scheme for `1 + 2 * 3`

The grammar in listing 5.1 has all its keywords (the operators and digits) defined in it as terminals. One could ask how this grammar deals with whitespace, which consists of spaces, tabs and (possibly) newlines. We would naturally like to allow an abitrary amount of whitespace to occur between two tokens (digits, operators, or parentheses), but the term *whitespace* occurs nowhere in the grammar. The answer is that whitespace is not usually included in a grammar, although it could be. The *lexical analyzer* uses whitespace to see where a word ends and where a new word begins, but otherwise discards it (unless the witespace occurs within comments or strings, in which case it is significant). In our language, whitespace does not have any significance at all so we assume that it is discarded.

We would now like to extend definition 5.2 a little further, because we have not clearly stated what a production rule is.

Definition 5.3 (Production Rule)

In the quadruple (V, Σ, S, P) that defines a context-free grammar, $P \subseteq N \times (V \cup \Sigma)^*$ is a finite set of production rules.

■

Here, $(V \cup \Sigma)$ is the union of the set of nonterminals and the set of terminals, yielding the set of all symbols. $(V \cup \Sigma)^*$ denotes the set of finite strings of elements from $(V \cup \Sigma)$. In other words, P is a set of 2-tuples with on the left-hand side a nonterminal, and on the right-hand side a string constructed from items from V and Σ. It should now be clear that the following are examples of production rules:

expression:	expression $*$ expression.
number:	**3**.

We have already shown that production rules are used to derive valid sentences from the start symbol (sentences that may occur in the language under consideration). The formal method used to derive such sentences are (also see *Languages and Machines* by Thomas Sudkamp ([8]):

Definition 5.4 (String Derivation)

Let $G = (V, \Sigma, S, P)$ be a context-free grammar and $v \in (V \cup \Sigma)^*$. The set of strings derivable from v is recursively defined as follows:

1. Basis: v is derivable from v.
2. Recursive step: If $u = xAy$ is derivable from v and $A \longrightarrow w \in P$, then xwy is derivable from v.
3. Closure: Precisely those strings constructed from v by finitely many applications of the recursive step are derivable from v.

■

This definition illustrates how we use lowercase latin letters to represent strings of zero or more terminal symbols, and uppercase latin letters to represent a nonterminal symbol. Furthermore, we use lowercase greek letters to denote strings of terminal and nonterminal symbols.

A close formula may be given for all the sentences derivable from a given grammar, simultaneously introducing a new operator:

$$\{s \in \Sigma^* : S \Longrightarrow^* s\} \tag{5.1}$$

We have already discussed the operator $\Longrightarrow$, which denotes the derivation of a sentential form from another sentential form by applying a production rule. The $\Longrightarrow$ relation is defined as

$$\{(\sigma A\tau, \sigma\alpha\tau) \in (V \cup \Sigma)^* \times (V \cup \Sigma)^* : \sigma, \tau \in (V \cup \Sigma)^* \wedge (A, \alpha) \in P\}$$

which means, in words: $\Longrightarrow$ is a collecton of 2-tuples, and is therefore a relation which binds the left element of each 2-tuple to the right-element, thereby defining the possible replacements (productions) which may be performed. In the tuples, the capital latin letter A represents a nonterminal symbol which gets replaced by a string of nonterminal and terminal symbols, denoted using the greek lowercase letter α. σ and τ remain unchanged and serve to illustrate that a replacement (production) is *context-insensitive* or *context-free*. Whatever the actual value of the strings σ and τ, the replacement can take place. We will encounter other types of grammars which include context-sensitive productions later.

The $\Longrightarrow^*$ relation is the reflexive closure of the relation $\Longrightarrow$. $\Longrightarrow^*$ is used to indicate that multiple production rules have been applied in succession to achieve the result stated on the right-hand side. The formula $\alpha \Longrightarrow^* \beta$ denotes an arbitrary derivation starting with α and ending with β. It is perfectly valid to rewrite the derivation scheme we presented in table 5.5 using the new operator (see table 5.5). We can use this approach to leave out derivations that are obvious, analogous to the way one leaves out trivial steps in a mathematical proof.

The concept of *recursion* is illustrated by the production rule *expression* : "("*expression*")". When deriving a sentence, the nonterminal expression may be replaced by itself (but between parentheses). This recursion may continue indefinitely, termination being reached when another production rule for expression is applied (in particular, *expression* : *number*).

	expression
$\Longrightarrow^*$	*expression* +expression ∗ expression
$\Longrightarrow^*$	**1** +*number*∗ expression
$\Longrightarrow^*$	**1 +2 ∗ 3**

Table 5.2: Compact derivation scheme for 1 + 2 * 3

Recursion is considered *left recursion* when the nonterminal on the left-hand side of a production also occurs as the first symbol in the right-hand side of the production. This applies to most of the production rules for expression, for example:

expression: expression + expression.

While this property does not affect our ability to derive sentences from the grammar, it does prohibit a machine from automatically parsing an input text using *determinism*. This will be discussed shortly. Left recursion can be obvious (as it is in this example), but it can also be buried deeply in a grammar. It some cases, it takes a keen eye to spot and remove left recursion. Consider the following example of *indirect recursion* (in this example, we use capital latin letters to indicate nonterminal symbols and lowercase latin letters to indicate strings of terminal symbols, as is customary in the compiler construction literature):

Example 5.1 (Indirect Recursion)

A: Bx
B: Cy
C: Az
C: x

A may be replaced by Bx, thus removing an instance of A from a sentential form, and B may be replaced by Cy. C may be replaced by Az, which reintroduces A in the sentential form: indirect recursion.

This example was taken from [2].

□

5.6 The Chomsky Hierarchy

An *unrestricted rewriting system*[1] (grammar), the collection of production rules is:

$$P \subseteq (V \cup \Sigma)^* \times (V \cup \Sigma)^*$$

This means that the most leniant form of grammar allows multiple symbols, both terminals and nonterminals on the left hand side of a production rule. Such a production rule is often denoted

$$(\alpha, \omega)$$

since greek lowercase letters stand for a finite string of terminal and non-terminal symbols, i.e. $(V \cup \Sigma)^*$. The unrestricted grammar generates a *type 0 language* according to the *Chomsky hierarchy*. Noam Chomsky has defined four levels of grammars which successively more severe restrictions on the form of production rules which result in interesting classes of grammars.

A *type 1 grammar* or *context-sensitive grammar* is one in which each production $\alpha \longrightarrow \beta$ is such that $\mid \beta \mid \geq \mid \alpha \mid$. Alternatively, a context-sensitive grammar is sometimes defined as having productions of the form

$$\gamma A \rho \longrightarrow \gamma \omega \rho$$

where ω cannot be the empty string (ϵ). This is, of course, the same definition. A type 1 grammar generates a *type 1 language*.

A *type 2 grammar* or *context-free grammar* is one in which each production is of the form

$$A \longrightarrow \omega$$

where ω can be the empty string (ϵ). A context-free grammar generates a *type 2 language*.

A *type 3 grammar* or *regular grammar* is either *right linear*, with each production of the form

$$A \longrightarrow a \qquad or \qquad A \longrightarrow aB$$

or *left-linear*, in which each production is of the form:

$$A \longrightarrow a \qquad or \qquad A \longrightarrow Ba$$

A regular grammar generates a *type 3 language*. Regular grammars are very easy to parse (analyze the structure of a text written using such a grammar) but are not very powerful at the same time. They are often used to write lexical analyzers and were discussed in some detail in the previous chapter. Grammars for most actual programming languages are context free, since this type of grammar turns out to be easy to parse and yet powerful. The higher classes (0 and 1) are not often used.

As it happens, the class of context-free languages (type 2) is not very large. It turns out that there are almost no interesting languages that are context-free. But this problem is easily solved by first defining a superset of the language that is being designed, in order to formalize the context-free aspects of this language. After that, the remaining restrictions are defined using other means (i.e. semantic analysis).

As an example of a context-sensitive aspect (from Meijer [6]), consider the fact that in many programming languages, variables must be declared before they may be used. More formally, in sentences of the form $\alpha X \beta X \gamma$, in which the number of possible productions for X and the length of the production for β are not limited, both instances of X must always have the same production. Of

course, this cannot be expressed in a context-free manner.[3] This is an immediate consequence of the fact that the productions are context-free: every nonterminal may be replaced by one of its right-hand sides *regardless of its context*. Context-free grammars can therefore be spotted by the property that the left-hand side of their production rules consist of precisely one nonterminal.

5.7 Additional Notation

In the previous section, we have shown how a grammar can be written for a simple language using the Backus-Naur form (BNF). Because BNF can be unwieldy for languages which contain many alternative production rules for each nonterminal or involve many recursive rules (rules that refer to themselves), we also have the option to use the *extended Backus-Naur form* (EBNF). EBNF introduces some meta-operators (which are only significant in EBNF and have no function in the language being defined) which make the life of the grammar writer a little easier. The operators are:

Operator	**Function**
(and)	Group symbols together so that other meta-operators may be applied to them as a group.
[and]	Symbols (or groups of symbols) contained within square brackets are optional.
{ and }	Symbols (or groups of symbols) between braces may be repeated zero or more times.
|	Indicates a choice between two symbols (usually grouped with parentheses).

Our sample grammar can now easily be rephrased using EBNF (see listing 5.2). Note how we are now able to combine multiple productions rules for the same nonterminal into one production rule, but *be aware* that the alternatives specified between pipes (|) still constitute multiple production rules. EBNF is the syntax description language that is most often used in the compiler construction literature.

Yet another, very intuitive way of describing syntax that we have already used extensively in the Inger language specification in chapter 3, is the *syntax diagram*. The production rules from listing 5.2 have been converted into two syntax diagrams in figure 5.1.

Syntax diagrams consist of terminals (in boxes with rounded corners) and nonterminals (in boxes with sharp corners) connected by lines. In order to produce valid sentences, the user begins with the syntax diagram designated as the top-level diagram. In our case, this is the syntax diagram for expression, since expression is the *start symbol* in our grammar. The user then traces the line leading into the diagram, evaluating the boxes he encounters on the way. While tracing lines, the user may follow only rounded corners, never sharp ones, and may not reverse direction. When a box with a terminal is encountered, that terminal is placed in the sentence that is written. When a box containing a nonterminal

[3] That is, unless there were a (low) limit on the number of possible productions for X and/or the length of β were fixed and small. In that case, the total number of possibilities is limited and one could write a separate production rule for each possibility, thereby regaining the freedom of context.

```
expression:     expression + expression
              | expression - expression
              | expression * expression
              | expression / expression
              | expression ^ expression
              | number
              | ( expression ) .
number:       0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.
```

Listing 5.2: Sample Expression Language in EBNF

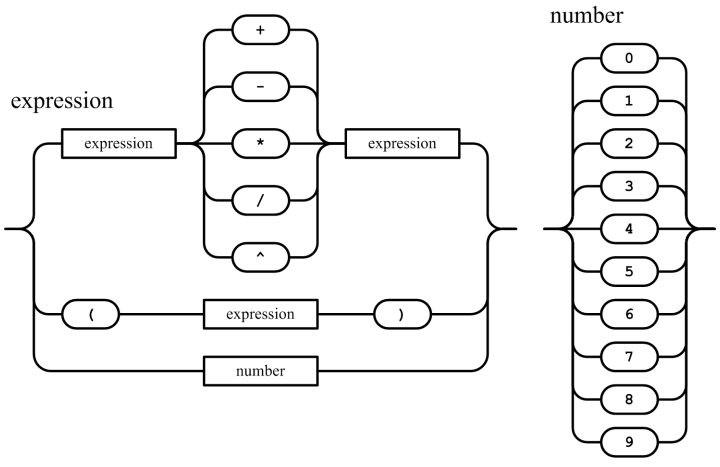


Figure 5.1: Syntax Diagrams for Mathematical Expressions

is encountered, the user switches to the syntax diagram for that nonterminal. In our case, there is only one nonterminal besides expression (number) and thus there are only two syntax diagrams. In a grammar for a more complete language, there may be many more syntax diagrams (consult appendix E for the syntax diagrams of the Inger language).

Example 5.2 (Tracing a Syntax Diagram)

Let's trace the syntax diagram in figure 5.1 to generate the sentence

```
1 + 2 * 3 - 4
```

We start with the expression diagram, since expression is the start symbol. Entering the diagram, we face a selection: we can either move to a box containing expression, move to a box containing the terminal (or kmove to a box containing number. Since there are no parentheses in the sentence that we want to generate, the second alternative is eliminated. Also, if we were to move to number now, the sentence generation would end after we generate only one digit, because after the number box, the line we are tracing ends. Therefore we are left with only one alternative: move to the expression box.

The expression box is a nonterminal box, so we must restart tracing the expression syntax diagram. This time, we move to the number box. This is also a nonterminal box, so we must pause our current trace and start tracing the number syntax diagram. The number diagram is simple: it only offers use one choice (pick a digit). We trace through 1 and leave the number diagram, picking up where we left off in the expression diagram. After the number box, the expression diagram also ends so we continue our first trace of the expression diagram, which was paused after we entered an expression box. We must now choose an operator. We need a +, so we trace through the corresponding box. Following the line from + brings us to a second expression box. We must once again pause our progress and reenter the expression diagram. In the following interations, we pick 2, *, 3, - and 4. Completing the trace is left as an exercise to the reader.

□

Fast readers may have observed that converting (E)BNF production rules to syntax diagrams does not yield very efficient syntax diagrams. For instance, the syntax diagrams in figure 5.2 for our sample expression grammar are simpler than the original ones, because we were able to remove most of the recursion in the expression diagram.

At a later stage, we will have more to say about syntax diagrams. For now, we will direct our attention back to the sentence generation process.

5.8 Syntax Trees

The previous sections included some example of sentence generation from a given grammar, in which the generation process was visualized using a *derivation scheme* (such as table 5.5 on page 86. Much more insight is gained from drawing a so-called *parse tree* or *syntax tree* for the derivation.

We return to our sample expression grammar (listing 5.3, printed here again for easy reference) and generate the sentence

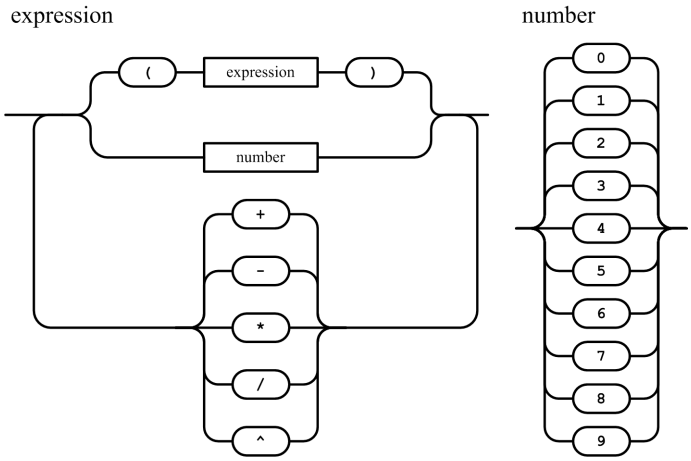


Figure 5.2: Improved Syntax Diagrams for Mathematical Expressions

expression: expression + expression
| expression − expression
| expression ∗ expression
| expression / expression
| expression ^ expression
| number
| (expression) .
number: **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**.

Listing 5.3: Sample Expression Language in EBNF

```
1 + 2 * 3
```

We will derive this sentence using *leftmost derivation* as shown in the derivation scheme in table 5.8.

The resulting parse tree is in figure 5.3. Every nonterminal encountered in the derivation has become a node in the tree, and the terminals (the digits and operators themselves) are the leaf nodes. We can now easily imagine how a machine would calculate the value of the expression `1 + 2 * 3`: every nonterminal node retrieves the value of its children and performs an operation on them (addition, subtraction, division, multiplication), and stores the result inside itself. This process occurs recursively, so that eventually the topmost node of the tree, known as the `root node`, contains the final value of the expression. Not all nonterminal nodes perform an operation on the values of their children; the `number` node does not change the value of its child, but merely serves as a placeholder. When a parent node queries the `number` node for its value, it merely passes the value of its child up to its parent. The following recursive definition states this approach more formally:

	expression
$\Longrightarrow$	*expression* $*$ *expression*
$\Longrightarrow$	*expression* $+$expression $*$ expression
$\Longrightarrow$	*number*$+$expression $*$ expression
$\Longrightarrow$	**1** $+$*expression* $*$ expression
$\Longrightarrow$	**1** $+$*number*$*$ expression
$\Longrightarrow$	**1** $+$**2** $*$ *expression*
$\Longrightarrow$	**1** $+$**2** $*$ *number*
$\Longrightarrow$	**1** $+$**2** $*$ **3**

Table 5.3: Leftmost derivation scheme for `1 + 2 * 3`

Definition 5.5 (Tree Evaluation)

The following algorithm may be used to evaluate the final value of an expression stored in a tree.

Let n be the root node of the tree.

- If n is a leaf node (i.e. if n has no children), the final value of n its current value.
- If n is not a leaf node, the value of n is determined by retrieving the values of its children, from left to right. If one of the children is an operator, it is applied to the other children and the result is the final result of n.

■

The tree we have just created is not unique. In fact, their are multiple valid trees for the expression `1 + 2 * 3`. In figure 5.4, we show the parse tree for the *rightmost derivation* of our sample expression. This tree differs slightly (but significantly) from our original tree. Apparently out grammar is *ambiguous*: it can generate multiple trees for the same expression.

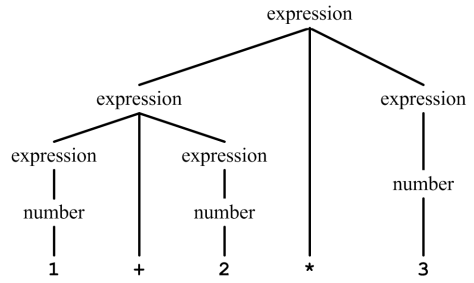


Figure 5.3: Parse Tree for Leftmost Derivation of `1 + 2 * 3`

The existence of multiple trees is not altogether a blessing, since it turns out that different trees produce different expression results.

Example 5.3 (Tree Evaluation)

	expression
$\Longrightarrow$	expression $+$*expression*
$\Longrightarrow$	expression $+$expression $*$ *expression*
$\Longrightarrow$	expression $+$expression $*$ *number*
$\Longrightarrow$	expression $+$*expression* $*$ **3**
$\Longrightarrow$	expression $+$*number*$*$ **3**
$\Longrightarrow$	*expression* $+$**2** $*$ **3**
$\Longrightarrow$	*number*$+$**2** $*$ **3**
$\Longrightarrow$	**1** $+$**2** $*$ **3**

Table 5.4: Rightmost derivation scheme for 1 + 2 * 3

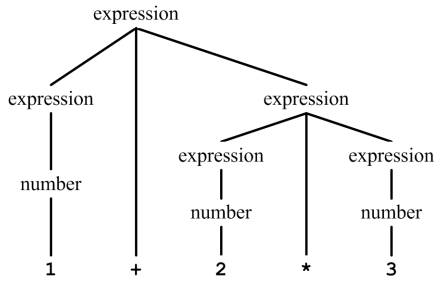


Figure 5.4: Parse Tree for Rightmost Derivation of 1 + 2 * 3

In this example, we will calculate the value of the expression 1 + 2 * 3 using the parse tree in figure 5.4. We start with the root node, and query the values of its three expression child nodes. The value of the left child node is 1, since it has only one child (number) and its value is 1. The value of the right expression node is determined recursively by retrieving the values of its two expression child nodes. These nodes evaluate to 2 and 3 respectively, and we apply the middle child node, which is the multiplication (*) operator. This yields the value 6 which we store in the expression node.

The values of the left and right child nodes of the root expression node are now known and we can calculate the final expression value. We do so by retrieving the value of the root node's middle child node (the + operator) and applying it to the values of the left and right child nodes (1 and 6 respectively). The result, 7 is stored in the root node. Incidentally, it is also the correct answer.

At the end of the evaluation, the expression result is known and resides inside the root node.

□

In this example, we have seen that the value 7 is found by evaluating the tree corresponding to the rightmost derivation of the expression 1 + 2 * 3. This is illustrated by the *annotated parse tree*, which is shown in figure 5.5.

We can now apply the same technique to calculate the final value of the parse tree corresponding to the leftmost derivation of the expression 1 + 2 * 3, shown in figure 5.6. We find that the answer (9) is incorrect, which is caused by the order in which the nodes are evaluated.

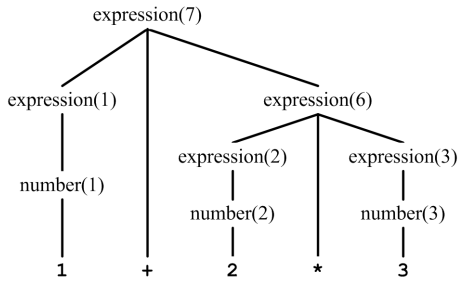


Figure 5.5: Annotated Parse Tree for Rightmost Derivation of 1 + 2 * 3

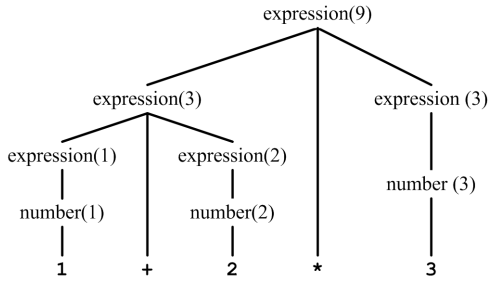


Figure 5.6: Annotated Parse Tree for Leftmost Derivation of 1 + 2 * 3

The nodes in a parse tree must reflect the precedence of the operators used in the expression in the parse tree. In case of the tree for the rightmost derivation of 1 + 2 * 3, the precedence was correct: the value of 2 * 3 was evaluated before the 1 was added to the result. In the parse tree for the leftmost derivation, the value of 1 + 2 was calculated before the result was multiplied by 3, yielding an incorrect result. Should we, then, always use rightmost derivations? The answer is no: it is mere coindence that the rightmost derivation happens to yield the correct result – it is the grammar that is flawed. With a correct grammar, any derivation order will yield the same results and only one parse tree correspons to a given expression.

5.9 Precedence

The problem of ambiguity in the grammar of the previous section is solved for a big part by introducing new nonterminals, which will serve as placeholders to introduce operator precedence levels. We know that multiplication (*) and division (/) bind more strongly than addition (+) and subtraction(-), but we need a means to visualize this concept in the parse tree. The solution lies in adding the nonterminal term (see the new grammar in listing 5.4, which will deal with multiplications and additions. The original expression nonterminal is now only used for additions and subtractions. The result is, that whenever a multiplication or division is encountered, the parse tree will contain a term node in which all multiplications and divisions are resolved until an addition or

```
expression:   term + expression
            | term − expression
            | term.
term:         factor ∗ term
            | factor / term
            | factor ^ term
            | factor .
factor:       0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.
            | ( expression ) .
```

Listing 5.4: Unambiguous Expression Language in EBNF

	expression
$\Longrightarrow$	*term* +expression
$\Longrightarrow$	*factor* +expression
$\Longrightarrow$	**1** +*expression*
$\Longrightarrow$	**1** +*term*
$\Longrightarrow$	**1** +*factor* ∗ term
$\Longrightarrow$	**1** +**2** ∗ *term*
$\Longrightarrow$	**1** +**2** ∗ *factor*
$\Longrightarrow$	**1** +**2** ∗ **3**

Table 5.5: Leftmost derivation scheme for 1 + 2 * 3

subtraction arrives.

We also introduce the nonterminal factor to replace number, and to deal with parentheses, which have the highest precedence. It should now become obvious that the lower you get in the grammar, the higher the priority of the operators dealt with. Tables 5.9 and 5.9 show the leftmost and rightmost derivation of 1 + 2 * 3. Careful study shows that they are the same. In fact, the corresponding parse trees are exactly identical (shown in figure 5.7). The parse tree is already annotated for convience and yields the correct result for the expression it holds.

It should be noted that in some cases, an instance of, for example, term actually adds an operator (* or /) and sometimes it is merely included as a placeholder that holds an instance of factor. Such nodes have no function in a syntax tree and can be safely left out (which we will do when we generate *abstract syntax trees*.

There is an amazing (and amusing) trick that was used in the first FORTRAN compilers to solve the problem of operator precedence. An excerpt from a paper by Donald Knuth (1962):

> *An ingenious idea used in the first FORTRAN compiler was to surround binary operators with peculiar-looking parentheses:*
>
> $+$ *and* $-$ *were replaced by* $)))+((($ *and* $)))-((($
> $*$ *and* $/$ *were replaced by* $))*(($ *and* $))/(($
> $**$ *was replaced by* $)**($

	expression
$\Longrightarrow$	term $+$*expression*
$\Longrightarrow$	term $+$*term*
$\Longrightarrow$	term $+$factor "$*$" term
$\Longrightarrow$	term $+$factor $*$ *factor*
$\Longrightarrow$	term $+$*factor* $* \mathbf{3}$
$\Longrightarrow$	term $+\mathbf{2} * \mathbf{3}$
$\Longrightarrow$	*term* $+\mathbf{2} * \mathbf{3}$
$\Longrightarrow$	*factor* $+\mathbf{2} * \mathbf{3}$
$\Longrightarrow$	$\mathbf{1} + \mathbf{2} * \mathbf{3}$

Table 5.6: Rightmost derivation scheme for `1 + 2 * 3`

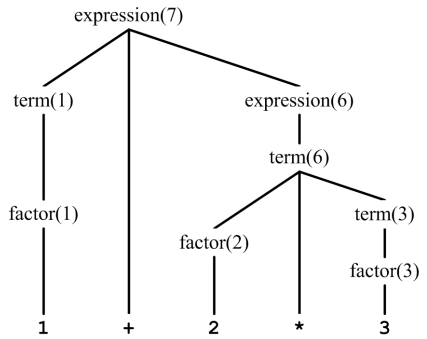


Figure 5.7: Annotated Parse Tree for Arbitrary Derivation of `1 + 2 * 3`

and then an extra "(((" at the left and ")))" at the right were tacked on. For example, if we consider "$(X + Y) + W/Z$," we obtain

$$((((X))) + (((Y)))) + (((W))/(((Z)))$$

This is admittedly highly redundant, but extra parentheses need not affect the resulting machine language code.

Another approach to solve the precedence problem was invented by the Polish scientist J. Lukasiewicz in the late 20s. Today frequently called *prefix notation*, the parenthesis-free or *polish notation* was a perfect notation for the output of a compiler, and thus a step towards the actual mechanization and formulation of the compilation process.

Example 5.4 (Prefix notation)

`1 + 2 * 3` becomes `+ 1 * 2 3`
`1 / 2 - 3` becomes `- / 1 2 3`

□

5.10 Associativity

When we write down the syntax tree for the expression `2 - 1 - 1` according to our example grammar, we discover that our grammar is still not correct (see figure 5.8). The parse tree yields the result `2` while the correct result is `0`, even though we have taken care of operator predence. It turns out that apart from precedence, operator *associativity* is also important. The subtraction operator - associates to the left, so that in a (sub) expression which consists only of operators of equal precedence, the order in which the operators must be evaluated is still fixed. In the case of subtraction, the order is from left to right. In the case of ^ (power), the order is from right to left. After all,

$$2^{2^2} = 512 \neq 64.$$

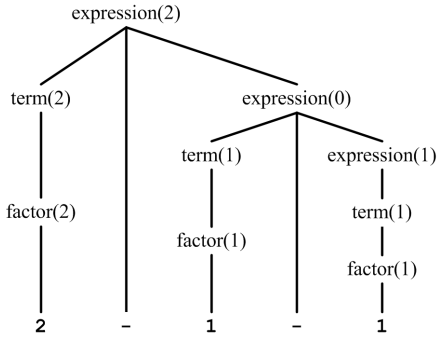


Figure 5.8: Annotated Parse Tree for `2 - 1 - 1`

```
expression:     expression + term
              | expression - term
              | term.
term:           factor * term
              | factor / term
              | factor ^ term
              | factor.
factor:         0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.
              | ( expression ) .
```

Listing 5.5: Expression Grammar Modified for Associativity

It turns out that our grammar works only for right-associative operators (or for non-associative operators like addition or multiplication, since these may be treated like right-associative operators), because its production rules are right-recursive. Consider the following excerpt:

```
expression:     term + expression
              | term - expression
              | term.
```

The nonterminal expression acts as the left-hand side of these three production rules, and in two of them also occurs on the far right. This causes right recursion which can be spotted in the parse tree in figure 5.8: the right child node of every expression node is again an expression node. Left recursion can be recognized the same way. The solution, then, to the associativity problem is to introduce left-recursion in the grammar. The grammar in listing 5.5 can deal with left-associativity and right-associativity, because expression is left-recursive, causing + and - to be treated as left-associative operators, and term is right-recursive, causing *, / and ^ to be treated as right-associative operators.

And presto–the expressions `2 - 1 - 1` and `2 ^ 3 ^ 2` now have correct parse trees (figures 5.9 and 5.10). We will see in the next chapter that we are not quite out of the woods yet, but never fear, the worst is behind us.

5.11 A Logic Language

As a final and bigger example, we present a complete little language that handles propositional logic notation (proposition, implication, conjunction, disjunction and negation). This language has operator precedence and associativity, but very few terminals (and an interpreter can therefore be completely implemented as an exercise. We will do so in the next chapter). Consult the following sample program:

Example 5.5 (Proposition Logic Program)

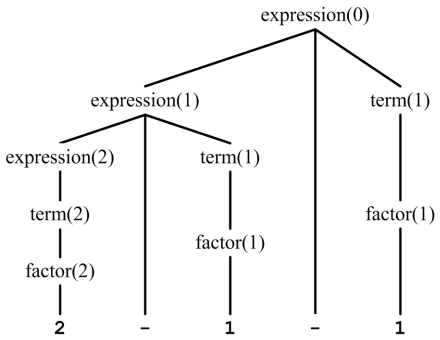


Figure 5.9: Correct Annotated Parse Tree for `2 - 1 - 1`

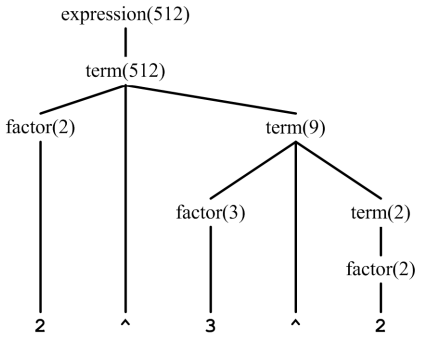


Figure 5.10: Correct Annotated Parse Tree for `2 ^ 3 ^ 2`

```
A = 1
B = 0
C = (~A) | B
RESULT = C -> A
```

□

The language allows the free declaration of variables, for which capital letters are used (giving a range of 26 variables maximum). In the example, the variable A is declared and set to *true* (1), and B is set to *false* (0). The variable C is declared and set to (~A) | B, which is false (0). Incidentally, the parentheses are not required because ~ has higher priority than |. Finally, the program is terminated with an instruction that prints the value of C -> B, which is true (1). Termination of a program with such an instruction is required.

Since our language is a proposition logic language, we must define truth tables for each operator (see table 5.7). You may already be familiar with all the operators. Pay special attention to the operator precedence relation:

Operator	**Priority**	**Operation**
~	1	Negation (not)
&	2	Conjunction (and)
|	2	Disjunction (or)
->	3	Right Implication
<-	3	Left Implication
<->	3	Double Implication

A	B	A & B
F	F	F
F	T	F
T	F	F
T	T	T

A	B	A | B
F	F	F
F	T	T
T	F	T
T	T	T

A	~A
F	T
T	F

A	B	A -> B
F	F	T
F	T	T
T	F	F
T	T	T

A	B	A <- B
F	F	T
F	T	F
T	F	T
T	T	T

A	B	A <-> B
F	F	T
F	T	F
T	F	F
T	T	T

Table 5.7: Proposition Logic Operations and Their Truth Tables

Now that we are familiar with the language and with the operator precedence relation, we can write a grammar in BNF. Incidentally, all operators are non-associative, and we will treat them as if they associated to the right (which is easiest for parsing by a machine, in the next chapter). The BNF grammar is in listing 5.6. For good measure, we have also written the grammar in EBNF (listing 5.7).

You may be wondering why we have built our BNF grammar using complex constructions with empty production rules (ϵ) while our running example, the

```
program:          statementlist RESULT = implication.
statementlist :   ϵ.
statementlist :   statement statementlist.
statement:        identifier = implication ;.
implication :     conjunction restimplication.
restimplication : ϵ.
restimplication : -> conjunction restimplication.
restimplication : <- conjunction restimplication.
restimplication : <-> conjunction restimplication.
conjunction:      negation restconjunction.
restconjunction : ϵ.
restconjunction : & negation restconjunction.
restconjunction : | negation restconjunction.
negation:         ~ negation.
negation:         factor.
factor :          ( implication ) .
factor :          identifier .
factor :          1.
factor :          0.
identifier :      A.
                  ...
identifier :      Z.
```

Listing 5.6: BNF for Logic Language

```
program:          { statement ; } RESULT = implication.
statement:        identifier = implication.
implication :     conjunction { ( -> | <- | <-> ) implication } .
conjunction:      negation { ( & | | ) conjunction } .
negation:         { ~ } factor.
factor :          ( implication )
                  | identifier
                  | 1
                  | 0.
identifier :      A | ... | Z.
```

Listing 5.7: EBNF for Logic Language

mathematical expression grammar, was so much easier. The reason is that in our expression grammar, multiple individual production rules with the same nonterminal on the left-hand side (e.g. factor), also start with that nonterminal. It turns out that this property of a grammar makes it difficult to implement in an automatic parser (which we will do in the next chapter). This is why we must go out of our way to create a more complex grammar.

5.12 Common Pitfalls

We conclude our chapter on grammar with some practical advice. Grammars are not the solution to everything. They can describe the basic structure of a language, but fail to capture the details. You can easily spend much time trying to formulate grammars that contain the intricate details of some shadowy corner of your language, only to find out that it would have been far easier to handle those details in the semantic analysis phase. Often, you will find that some things just cannot be done with a context free grammar.

Also, if you try to capture a high level of detail in a grammar, your grammar will grow rapidly grow and become unreadable. Extended Backus-Naur form may cut you some notational slack, but in the end you will be moving towards attribute grammars or affix grammars (discussed in Meijer, [6]).

Visualizing grammars using syntax diagrams can be a big help, because Backus-Naur form can lure you into recursion without termination. Try to formulate your entire grammar in syntax diagrams before moving to BNF (even though you will have to invest more time). Refer to the syntax diagrams for the Inger language in appendix E for an extensive example, especially compared to the BNF notation in appendix D.

Bibliography

[1] A.V. Aho, R. Sethi, J.D. Ullman: *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] F.H.J. Feldbrugge: *Dictaat Vertalerbouw*, Hogeschool van Arnhem en Nijmegen, edition 1.0, 2002.

[3] J.D. Fokker, H. Zantema, S.D. Swierstra: *Programmeren en correctheid*, Academic Service, Schoonhoven, 1991.

[4] A. C. Hartmann: *A Concurrent Pascal Compiler for Minicomputers*, Lecture notes in computer science, Springer-Verlag, Berlin 1977.

[5] J. Levine: *Lex and Yacc*, O'Reilly & sons, 2000

[6] H. Meijer: *Inleiding Vertalerbouw*, University of Nijmegen, Subfaculty of Computer Science, 2002.

[7] M.J. Scott: *Programming Language Pragmatics*, Morgan Kaufmann Publishers, 2000.

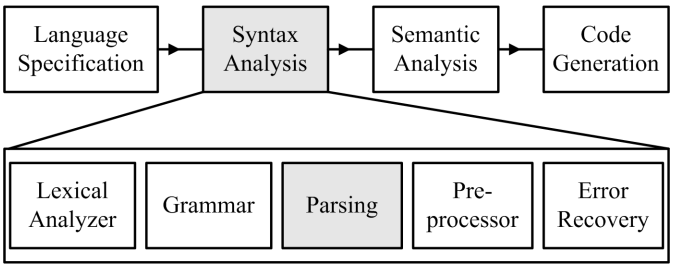
[8] T. H. Sudkamp: *Languages & Machines*, Addison-Wesley, 2nd edition, 1998.

[9] N. Wirth: *Compilerbouw*, Academic Service, 1987.

[10] N. Wirth and K. Jensen: *PASCAL User Manual and Report*, Lecture notes in computer science, Springer-Verlag, Berlin 1975.

Chapter 6

Parsing



6.1 Introduction

In the previous chapter, we have devised grammars for formal languages. In order to generate valid sentences in these languages, we have written derivation schemes, and syntax trees. However, a compiler does not work by generating sentences in some language, buy by recognizing (parsing) them and then translating them to another language (usually assembly language or machine language).

In this chapter, we discuss how one writes a program that does exactly that: parse sentences according to a grammar. Such a program is called a *parser*. Some parsers build up a syntax tree for sentences as they recognize them. These syntax trees are identical to the ones presented in the previous chapter, but they are generated inversely: from the concrete sentence instead of from a derivation scheme. In short, a parser is a program that will read an input text and tell you if it obeys the rules of a grammar (and if not, why – if the parser is worth anything). Another way of saying it would be that a parser determines if a sentence can be generated from a grammar. The latter description states more precisely what a parser does.

Only the more elaborate compilers build up a syntax tree in memory, but we will do so explicitly because it is very enlightening. We will also discuss a technique to simplify the syntax tree, thus creating an *abstract syntax tree*, which is more compact than the original tree. The abstract syntax tree is very important: it is the basis for the remaining compilation phases of semantic analysis and code generation.

Parsing techniques come in many flavors and we do not presume to be able to discuss them all in detail here. We will only fully cover LL(1) parsing (recursive descent parsing), and touch on LR(k) parsing. No other methods are discussed.

6.2 Prefix code

The grammar chapter briefly touched on the subject of *prefix notation*, or *polish notation* as it is sometimes known. Prefix notation was invented by the Polish J. Lukasiewicz in the late 20s. This parenthesis-free notation was a perfect notation for the output of a compiler:

Example 6.1 (Prefix notation)

1 + 2 * 3 becomes + 1 * 2 3
1 / 2 - 3 becomes - / 1 2 3

□

In the previous chapter, we found that operator precedence and associativity can and should be handled in a grammar. The later phases (semantic analysis and code generation) should not be bothered with these operator properties anymore–the parser should convert the input text to an intermediate format that implies the operator priority and associativity. An unambiguous syntax tree is one such structure, and prefix notation is another.

Prefix notation may not seem very powerful, but consider that fact that it can easily be used to denote complex constructions like if ... then and while .. do with which you are no doubt familiar (if not, consult chapter 3):

if (var > 0) { a + b } **else** { b − a }	becomes	`?>var'0'+a'b'-b'a'`
while (n > 0) { n = n − 1 }	becomes	`W>n'0'=n'-'n'1'`

Apostrophes (') are often used as monadic operators that delimit a variable name, so that two variables are not actually read as one. As you can deduct from this example, prefix notation is actually a flattened tree. As long as the number of operands that each operator takes is known, the tree can easily be traversed using a recursive function. In fact, a very simple compiler can be constructed that translates the mathematical expression language that we devised into prefix code. A second program, the *evaluator*, could then interpret the prefix code and calculate the result.

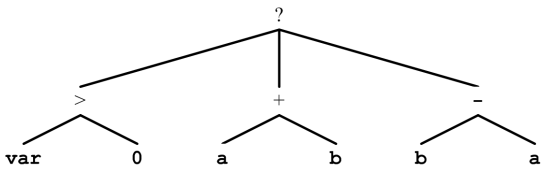


Figure 6.1: Syntax Tree for If-Prefixcode

To illustrate these facts as clearly as possible, we have placed the prefix expressions for the if and while examples in syntax trees (figures 6.1 and 6.2 respectively).

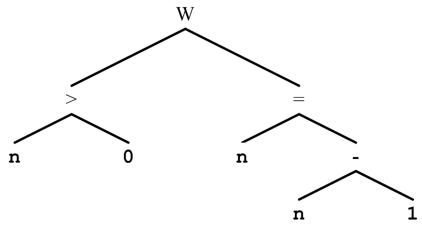


Figure 6.2: Syntax Tree for While-Prefixcode

Notice that the original expressions may be regained by walking the tree in a pre-order fashion. Conversely, try walking the tree in-order or post-order, and examine the result as an interesting exercise.

The benefits of prefix notation do not end there: it is also an excellent means to eliminate unnecessary *syntactic sugar* like whitespace and comments, without loss of meaning.

The evaluator program is a recursive affair: it starts reading the prefix string from left to right, and for every operator it encounters, it calls itself to retrieve the operands. The recursion terminates when a constant (a variable name or a literal value) is found. Compare this to the method we discussed in the introduction to this book. We said that we needed a *stack* to place (*shift*) values and operators on that could not yet be evaluated (*reduce*). The evaluator works by this principle, and uses the recursive function as a stack.

The translator-evaluator construction we have discussed so far may seem rather artificial to you. But real compilers. although more complex, work the same way. The big difference is that the evaluator is the computer processor (CPU) - it cannot be changed, and the code that your compiler outputs must obey the processor's rules. In fact, the machine code used by a real machine like the Intel x86 processor is a language unto itself, with a real grammar (consult the Intel instruction set manual [3] for details).

There is one more property of the prefix code and the associated trees: operators are no longer leaf nodes in the trees, but have become internal nodes. We could have used nodes like expression and term as we have done before, but these nodes would then be void of content. By making the operators nodes themselves, we save valuable space in the tree.

6.3 Parsing Theory

We have discussed prefix notations and associated syntax trees (or parse trees), but how is such a tree constructed from the original input (it *is* called a parse tree, after all)? In this section we present some of the theory that underlies parsing. Note that in a book of limited size, we do not presume to be able to treat all of the theory. In fact, we will limit our discussion to LL(1) grammars and mention LR parsing in a couple of places.

Parsing can occur in two basic fashions: *top-down* and *bottom-up*. With top-down, you start with a grammar's *start symbol* and work toward the concrete sentence under consideration by repeatedly applying production rules (replacing

nonterminals with one of their right-hand sides) until there are no nonterminals left. This method is by far the easiest method, but also places the most restrictions on the grammar.

Bottom-up parsing starts with the sentence to be parsed (the string of terminals), and repeatedly applies production rules inversely, i.e. replaces substrings of terminals nonterminals with the left-hand side of production rules. This method is more powerful than top-down parsing, but much harder to write by hand. Tools that construct bottom-up parsers from a grammar (*compiler-compilers*) exist for this purpose.

6.4 Top-down Parsing

Top-down parsing relies on a grammar's *determinism* property to work. A top down or *recursive descent* parser always takes the leftmost nonterminal in a *sentential form* (or the rightmost nonterminal, depending on the flavor of parser you use) and replaces it with one of its right-hand sides. Which one, depends on the next terminal character the parser reads from the input stream. Because the parser must constantly make choices, it must be able to do so without having to retrace its steps after making a wrong choice. There exist parsers that work this way, but these are obviously not very efficient and we will not give them any further thought.

If all goes well, eventually all nonterminals will have been replaced with terminals and the input sentence should be readable. If it is not, something went wrong along the way and the input text did not obey the rules of the grammar; it is said to be syntactically incorrect–there were *syntax errors*. We will later see ways to pinpoint the location of syntax errors precisely.

Incidentally, there is no real reason why we always replace the leftmost (or rightmost) nonterminal. Since our grammar is context-free, it does not matter which nonterminal gets replaced since there is no dependency between the nonterminals (context-insensitive). It is simply tradition to pick the leftmost nonterminal, and hence the name of the collection of recursive descent parsers: *LL*, which means "recognizable while reading from *left* to right, and rewriting *leftmost* nonterminals." We can also define RL right away, which means "recognizable while reading from *right* to left, and rewriting *leftmost* nonterminals." – this type of recursive descent parsers would be used in countries where text is read from right to left.

As an example of top-down parsing, consider the BNF grammar in listing 6.1. This is a simplified version of the mathematical expression grammar, made suitable for LL parsing (the details of that will follow shortly).

Example 6.2 (Top-down Parsing by Hand)

We will now parse the sentence `1 + 2 + 3` by hand, using the top-down approach. A top-down parser always starts with the start symbol, which in this case is `expression`. It then reads the first character from the input stream, which happens to be `1`, and determines which production rule to apply. Since there is only one producton rule that can replace `expression` (it only acts as the left-hand side of one rule), we replace `expression` with `factor restexpression`:

$$\texttt{expression} \quad \Longrightarrow_L \texttt{factor} \;\; \texttt{restexpression}$$

expression :	factor restexpression .
restexpression :	ϵ .
restexpression :	+ factor restexpression .
restexpression :	− factor restexpression .
factor :	**0**.
factor :	**1**.
factor :	**2**
factor :	**3**.
factor :	**4**.
factor :	**5**.
factor :	**6**.
factor :	**7**.
factor :	**8**.
factor :	**9**.

Listing 6.1: Expression Grammar for LL Parser

In LL parsing, we always replace the leftmost nonterminal (here, it is factor). factor has ten alternative production rules, but know exactly which one to pick, since we have the character 1 in memory and there is only one production rule whose right-hand side starts with 1:

$$\begin{aligned} \text{expression} &\Longrightarrow_L \text{factor restexpression} \\ &\Longrightarrow_L \mathbf{1} \text{ restexpression} \end{aligned}$$

We have just eliminated one terminal from the input stream, so we read the next one, which is "+". The leftmost nonterminal which we need to replace is restexpression, which has only one alternative that starts with +:

$$\begin{aligned} \text{expression} &\Longrightarrow_L \text{factor restexpression} \\ &\Longrightarrow_L \mathbf{1} \text{ restexpression} \\ &\Longrightarrow_L \mathbf{1} + \text{factor restexpression} \end{aligned}$$

We continue this process until we run out of terminal tokens. The situation at that point is:

$$\begin{aligned} \text{expression} &\Longrightarrow_L \text{factor restexpression} \\ &\Longrightarrow_L \mathbf{1} \text{ restexpression} \\ &\Longrightarrow_L \mathbf{1} + \text{factor restexpression} \\ &\Longrightarrow_L \mathbf{1} + \mathbf{2} \text{ restexpression} \\ &\Longrightarrow_L \mathbf{1} + \mathbf{2} + \text{factor restexpression} \\ &\Longrightarrow_L \mathbf{1} + \mathbf{2} + \text{factor restexpression} \\ &\Longrightarrow_L \mathbf{1} + \mathbf{2} + \mathbf{3} \text{ restexpression} \end{aligned}$$

The current terminal symbol under consideration is empty, but we could also use *end of line* or *end of file*. In that case, we see that of the three alternatives for restexpression, the ones that start with + and - are invalid. So we pick the production rule with the empty right-hand side, effectively removing restexpression from the sentential form. We are now left with the input sentence, having eliminated all the terminal symbols. Parsing was successful.

If we were to parse `1 + 2 * 3` using this grammar, parsing will not be successful. Parsing will fail as soon as the terminal symbol ***** is encountered. If the lexical analyzer cannot handle this token, parsing will end for that reason. If it can (which we will assume here), the parser is in the following situation:

$$\text{expression} \quad \Longrightarrow_L^* \mathbf{1\ +2}\ \text{restexpression}$$

The parser must now find a production rule starting with *****. There is none, so it replaces `restexpression` with the empty alternative. After that, there are no more nonterminals to replace, but there are still terminal symbols on the input stream, thus the sentence cannot be completely recognized.

□

There are a couple of caveats with the LL approach. Consider what happens if a nonterminal is replaced by a collection of other nonterminals, and so on, until at some point this collection of nonterminals is replaced by the original nonterminal, while no new terminals have been processed along the way. This process will then continue indefinitely, because there is no termination condition. Some grammars cause this behaviour to occur. Such grammars are called *left-recursive*.

Definition 6.1 (Left Recursion)

A context-free grammar (V, Σ, S, P) is left-recursive if

$$\exists X \in V, \alpha, \beta \in (V \cup \Sigma)^* : S \Longrightarrow^* X\alpha \Longrightarrow_L^* X\beta$$

in which $\Longrightarrow_L^*$ is the reflexive transitive closure of $\Longrightarrow_L$, defined as:

$$\{(uA\tau) \in (V \cup \Sigma)^* \times (V \cup \Sigma)^* : u \in \Sigma^*, \tau \in (V \cup \Sigma)^* \wedge (A, \alpha) \in P\}$$

The difference between $\Longrightarrow$ and $\Longrightarrow_L$ is that the former allows arbitrary strings of terminals and nonterminals to precede the nonterminal that is going to be replaced (A), while the latter insists that only terminals occur before A (thus making A the leftmost nonterminal).

Equivalently, we may as well define $\Longrightarrow_R$:

$$\{(\tau Au) \in (V \cup \Sigma)^* \times (V \cup \Sigma)^* : u \in \Sigma^*, \tau \in (V \cup \Sigma)^* \wedge (A, \alpha) \in P\}$$

$\Longrightarrow_R$ insists that A be the rightmost nonterminal, followed only by terminal symbols. Using $\Longrightarrow_R$, we are also in a position to define right-recursion (which is similar to left-recursion):

$$\exists X \in V, \alpha, \beta \in (V \cup \Sigma)^* : S \Longrightarrow^* \alpha X \Longrightarrow_R^* \beta X$$

Grammars which contain left-recursion are not guaranteed to terminated, although they may. Because this may introduce hard to find bugs, it is important to weed out left-recursion from the outset if at all possible.

■

Removing left-recursion can be done using *left-factorisation*. Consider the following excerpt from a grammar (which may be familiar from the previous chapter):

expression: expression + term.
expression: expression − term.
expression: term.

Obviously, this grammar is left-recursive: the first two production rules both start with expression, which also acts as their left-hand side. So expression may be replaced with expression without processing a nonterminal along the way. Let it be clear that there is nothing *wrong* with this grammar (it will generate valid sentences in the mathematical expression language just fine), it just cannot be recognized by a top-down parser.

Left-factorisation means recognizing that the nonterminal expression occurs multiple times as the leftmost symbol in a production rule, and should therefore be in a production rule on its own. Firstly, we swap the order in which term and expression occur:

expression: term + expression.
expression: term − expression.
expression: term.

The + and - operators are now treated as if they were right-associative, which - is definitely not. We will deal with this problem later. For now, assume that associativity is not an issue. This grammar is no longer left-recursive, and it obviously produces the same sentences as the original grammar. However, we are not out of the woods yet.

It turns out that when multiple production rule alternatives start with the same terminal or nonterminal symbol, it is impossible for the parser to choose an alternative based on the token it currently has in memory. This is the situation we have now; three production rules which all start with term. This is where we apply the left-factorisation: term may be removed from the beginning of each production rule and placed in a rule by itself. This is called "factoring out" a nonterminal on the left side, hence *left-factorisation*. The result:

expression: term restexpression.
restexpression: + term restexpression.
restexpression: − term restexpression.
restexpression: ϵ.

Careful study will show that this grammar produces exactly the same sentences as the original one. We have had to introduce a new nonterminal (restexpression) with an empty alternative to solve the left-recursion, in addition to wrong associativity for the - operator, so we were not kidding when we said that top-down parsing imposes some restrictions on grammar. On the flipside, writing a parser for such a grammar is a snap.

So far, we have assumed that the parser selects the production rule to apply

based on one terminal symbol, which is has in memory. There are also parsers that work with more than one token at a time. A recursive descent parser which works with 3 tokens is an LL(3) parser. More generally, an LL(k) parser is a top-down parser with a k tokens *lookahead*.

Practical advice 6.1 (One Token Lookahead)

Do not be tempted to write a parser that uses a lookahead of more than one token. The complexity of such a parser is much greater than the one-token lookahead LL(1) parser, and it will not really be necessary. Most, if not all, language constructs can be parsed using an LL(1) parser.

■

We have now found that grammars, suitable for recursive descent parsing, must obey the following two rules:

1. There most not be left-recursion in the grammar.
2. Each alternative production rule with the same left-hand side must start with a distinct terminal symbol. If it starts with a nonterminal symbol, examine the production rules for that nonterminal symbol and so on.

We will repeat these definitions more formally shortly, after we have discussed bottom-up parsing and compared it to recursive descent parsing.

6.5 Bottom-up Parsing

Bottom-up parsers are the inverse of top-down parsers: they start with the full input sentence (string of terminals) and work by replacing substrings of terminals and nonterminals in the sentential form by nonterminals, effectively reversely applying the production rules.

In de remainder of this chapter, we will focus on top-down parsing only, but we will illustrate the concept of bottom-up parsing (also known as LR) with an example. Consider the grammar in listing 6.2, which is not LL.

Example 6.3 (Bottom-up Parsing by Hand)

We will now parse the sentence 1 + 2 + 3 by hand, using the bottom-up approach. A bottom-up parser begins with the entire sentence to parse, and replaces groups of terminals and nonterminals with the left-hand side of production rules. In the initial situation, the parser sees the first terminal symbol, 1, and decides to replace it with factor (which is the only possibility). Such a replacement is called a *reduction*.

$$\mathbf{1} + \mathbf{2} + \mathbf{3} \implies \text{factor } + \mathbf{2} + \mathbf{3}$$

Starting again from the left, the parser sees the nonterminal factor and decides to replace it with expression (which is, once again, the only possibility):

$$\begin{aligned} \mathbf{1} + \mathbf{2} + \mathbf{3} &\implies \text{factor } + \mathbf{2} + \mathbf{3} \\ &\implies \text{expression } + \mathbf{2} + \mathbf{3} \end{aligned}$$

```
expression:     expression + expression.
expression:     expression − expression.
expression:     factor .
factor:         0.
factor:         1.
factor:         2
factor:         3.
factor:         4.
factor:         5.
factor:         6.
factor:         7.
factor:         8.
factor:         9.
```

Listing 6.2: Expression Grammar for LR Parser

There is now no longer a suitable production rule that has a lone expression on the right-hand side, so the parser reads another symbol from the input stream (+). Still, there is no production rule that matches the current input. The *tokens* expression and + are stored on a stack (*shifted*) for later reference. The parser reads another symbol from the input, which happens to be 2, which it can replace with factor, which can in turn be replaced by expression

$$\begin{aligned} \mathbf{1}\ \mathbf{+2}\ +\ \mathbf{3} &\Longrightarrow \text{factor}\ \mathbf{+2}\ \mathbf{+3} \\ &\Longrightarrow \text{expression}\ \mathbf{+2}\ \mathbf{+3} \\ &\Longrightarrow \text{expression}\ +\text{factor}\ \mathbf{+3} \\ &\Longrightarrow \text{expression}\ +\text{expression}\ \mathbf{+3} \end{aligned}$$

All of a sudden, the first three tokens in the sentential form (expression + expression), two of which were stored on the stack, form the right hand side of a production rule:

```
expression:     expression + expression.
```

The parser replaces the three tokens with expression and continues the process until the situation is thus:

$$\begin{aligned} \mathbf{1}\ \mathbf{+2}\ +\ \mathbf{3} &\Longrightarrow \text{factor}\ \mathbf{+2}\ \mathbf{+3} \\ &\Longrightarrow \text{expression}\ \mathbf{+2}\ \mathbf{+3} \\ &\Longrightarrow \text{expression}\ +\text{factor}\ \mathbf{+3} \\ &\Longrightarrow \text{expression}\ +\text{expression}\ \mathbf{+3} \\ &\Longrightarrow \text{expression}\ \mathbf{+3} \\ &\Longrightarrow \text{expression}\ +\text{factor} \\ &\Longrightarrow \text{expression}\ +\text{expression} \\ &\Longrightarrow \text{expression} \end{aligned}$$

In the final situation, the parser has reduced the entire original sentence to the start symbol of the grammar, which is a sign that the input text was syntactically correct.

□

Formally put, the *shift-reduce method* constructs a right derivation $S \Longrightarrow_R^* s$, but in reverse order. This example shows that bottom up parsers can deal with left-recursion (in fact, left recursive grammars make more efficient bottom up parsers), which helps keep grammars simple. However, we stick with top down parsers since they are by far the easiest to write by hand.

6.6 Direction Sets

So far, we have only informally defined which restrictions are placed on a grammar for it to be LL(k). We will now present these limitations more precisely. We must start with several auxiliary definitions.

Definition 6.2 (FIRST-Set of a Production)

The FIRST set of a production for a nonterminal A is the set of all terminal symbols, with which the strings generated from A can start.

■

Note that for an LL(k) grammar, the first k terminal symbols with which a production starts are included in the FIRST set, as a string. Also note that this definition relies on the use of BNF, not EBNF. It is important to realize that the following grammar excerpt:

factor : **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**.

actually consists of 10 different production rules (all of which happen to share the same left-hand side). The FIRST set of a production is often denoted PFIRST, as a reminder of the fact that it is the FIRST set of a single Production.

Definition 6.3 (FIRST-Set of a Nonterminal)

The FIRST set of a nonterminal A is the set of all terminal symbols, with which the strings generated from A can start.

If the nonterminal X has n productions in which it acts as the left-hand side, then

$$FIRST(X) := \bigcup_{i=1}^{n} PFIRST(X_i)$$

■

The LL(1) FIRST set of factor in the previous example is {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Its individual PFIRST sets (per production) are {0} through {9}. We will deal only with LL(1) FIRST sets in this book.

We also define the FOLLOW set of a nonterminal. FOLLOW sets are determined only for entire nonterminals, not for productions:

Definition 6.4 (FOLLOW-Set of a Nonterminal)

The FOLLOW set of a nonterminal A is the set of all terminal symbols, that may follow directly *after* A.

■

To illustrate FOLLOW-sets, we need a bigger grammar:

Example 6.4 (FOLLOW-Sets)

expression:	factor restexpression .
restexpression:	ϵ .
	| + factor restexpression
	| − factor restexpression .
factor:	**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**.

FOLLOW(expression) = $\{\perp\}$[1]
FOLLOW(restexpresson) = $\{\perp\}$
FOLLOW(factor) = $\{\perp, +, -\}$

□

We are now in a position to formalize the property of unambiguity for LL(k) grammars:

Definition 6.5 (Unambiguity of LL(k) Grammars)

A grammar is unambiguous when

1. If a nonterminal acts as the left-hand side of multiple productions, then the PFIRST sets of these productions must be disjunct.

2. If a nonterminal can produce the empty string (ϵ), then its FIRST set must be disjunct with its FOLLOW set.

■

How does this work in practice? The first conditions is easy. Whenever an LL parser reads a terminal, it must decide which production rule to apply. It does this by looking at the first k terminal symbols that each production rule can produce (its PFIRST set). In order for the parser to be able to make the choice, these sets must not have any overlap. If there is no overlap, the grammar is said to be *deterministic*.

If a nonterminal can be replaced with the empty string, the parser must check whether it is valid to do so. Inserting the empty string is an option when no other rule can be applied, and the nonterminals that come after the nonterminal that will produce the empty string are able to produce the terminal that the parser is currently considering. Hence, to make the decision, the FIRST set of the nonterminal must not have any overlap with its FOLLOW set.

[1] We use $\perp$ to denote end of file.

Production		PFIRST
program:	statementlist **RESULT**=implication.	$\{A \ldots Z\}$
statementlist :	ϵ .	$\emptyset$
statementlist :	statement statementlist .	$\{A \ldots Z\}$
statement:	identifier =implication ; .	$\{A \ldots Z\}$
implication :	conjunction restimplication .	$\{\sim, (, 0, 1, A \ldots Z\}$
restimplication :	ϵ .	$\emptyset$
restimplication :	$->$ conjunction restimplication .	{ -> }
restimplication :	$<-$ conjunction restimplication .	{ <- }
restimplication :	$<->$ conjunction restimplication.	{ <-> }
conjunction:	negation restconjunction .	$\{\sim, (, 0, 1, A \ldots Z\}$
restconjunction :	ϵ .	$\emptyset$
restconjunction :	**&**negation restconjunction .	$\{\&\}$
restconjunction :	| negation restconjunction .	$\{\vert\}$
negation:	~ negation.	$\{\sim\}$
negation:	factor .	$\{(, 0, 1, A \ldots Z\}$
factor :	(implication) .	$\{(\}$
factor :	identifier .	$\{A \ldots Z\}$
factor :	**1**.	$\{1\}$
factor :	**0**.	$\{0\}$
identifier :	**A**.	$\{A\}$
identifier :	**Z**.	$\{Z\}$

Table 6.1: PFIRST Sets for Logic Language

6.7 Parser Code

A wondrous amd most useful property of LL(k) grammars (henceforth referred to as LL since we will only be working with LL(1) anyway) is that a parser can be written for them in a very straightforward fashion (as long as the grammar is truly LL).

A top-down parser needs a stack to place its nonterminals on. It it easiest to use the stack offered by the C compiler (or whatever language you work with) for this purpose. Now, for every nonterminal, we produce a function. This function checks that the terminal symbol currently under consideration is an element of the FIRST set of the nonterminal that the function represents, or else it reports a syntax error.

After a syntax error, the parser may recover from the error using a synchronization approach (see chapter 8 on *error recovery* for details) and continue if possible, in order to find more errors.

The body of the function reads any terminals that are specified in the production rules for the nonterminal, and calls other functions (that represent other nonterminals) in turn, thus putting frames on the stack. In the next section, we will show that this approach is ideal for construction a syntax tree.

Writing parser code is best illustrated with an (elaborate) example. Please refer to the grammar for the logic language (section 5.11), for which we will write a parser. In table 6.7, we show the PFIRST set for every individual production, while in table 6.7, we show the FIRST and FOLLOW sets for every nonterminal.

Nonterminal	**FIRST**	**FOLLOW**
program	$\{A \ldots Z\}$	$\{\perp\}$
statementlist	$\{A \ldots Z\}$	$\{RESULT\}$
statement	$\{A \ldots Z\}$	$\{\sim, (, 0, 1, A \ldots Z, RESULT\}$
implication	$\{\sim, (, 0, 1, A \ldots Z\}$	$\{;, \perp,)\}$
restimplication	{ ->, <-, <-> }	$\{;, \perp,)\}$
conjunction	$\{\sim, (, 0, 1, A \ldots Z\}$	$\{$->, <-, <->, $;, \perp,)\}$
restconjunction	$\{\&, \mid\}$	$\{$->, <-, <->, $;, \perp,)\}$
negation	$\{\sim, (, 0, 1, A \ldots Z\}$	$\{\&, \mid,$ ->, <-, <->, $;, \perp,)\}$
factor:	$\{(, 0, 1, A \ldots Z\}$	$\{\&, \mid,$ ->, <-, <->, $;, \perp,)\}$
identifier:	$\{A \ldots Z\}$	$\{=, \&, \mid,$ ->, <-, <->, $;, \perp,)\}$

Table 6.2: FIRST and FOLLOW Sets for Logic Language

With this information, we can now build the parser. Refer to appendix G for the complete source code (including a lexical analyzer built with flex). We will discuss the C-function for the nonterminal conjunction here (shown in listing 6.3.

The conjunction function first checks that the current terminal input symbol (stored in the global variable token) is an element of $FIRST(conjunction)$ (lines 3–6). If not, conjunction returns an error.

If token is an element of the FIRST set, conjunction calls negation,which is the first token in the production rule for conjunction (lines 11-14):

```
conjunction:        negation restconjunction.
```

If negation returns without errors, conjunction must now decide whether to call restconjunction (which may produce the empty string). It does so by looking at the current terminal symbol under consideration. If it is a & or a | (both part of $FIRST(restconjunction)$, it calls restconjunction (lines 16-19). If not, it skips restconjunction, assuming it produces the empty string.

The other functions in the parser are constructed using a similar approach. Note that the parser presented here only performs a syntax check; the parser in appendix G also interprets its input (it is an interpreter), which makes for more interesting reading.

6.8 Conclusion

Our discussion of parser construction is now complete. The results of parsing are placing in a syntax tree and passed on to the next phase, semantic analysis.

```
1   int conjunction()
2   {
3       if ( token != '~' && token != '('
4           && token != IDENTIFIER
5           && token != TRUE
6           && token != FALSE )
7       {
8           return( ERROR );
9       }
10
11      if ( negation() == ERROR )
12      {
13          return( ERROR );
14      }
15
16      if ( token == '&' || token == '|' )
17      {
18          return( restconjunction () );
19      }
20      else
21      {
22          return( OK );
23      }
24  }
```

Listing 6.3: Conjunction Nonterminal Function

Bibliography

[1] A.V. Aho, R. Sethi, J.D. Ullman: *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] F.H.J. Feldbrugge: *Dictaat Vertalerbouw*, Hogeschool van Arnhem en Nijmegen, edition 1.0, 2002.

[3] Intel: *IA-32 Intel Architecture - Software Developer's Manual - Volume 2: Instruction Set*, Intel Corporation, Mt. Prospect, 2001.

[4] J. Levine: *Lex and Yacc*, O'Reilly & sons, 2000

[5] H. Meijer: *Inleiding Vertalerbouw*, University of Nijmegen, Subfaculty of Computer Science, 2002.

[6] M.J. Scott: *Programming Language Pragmatics*, Morgan Kaufmann Publishers, 2000.

[7] T. H. Sudkamp: *Languages & Machines*, Addison-Wesley, 2nd edition, 1998.

[8] N. Wirth: *Compilerbouw*, Academic Service, 1987.

Chapter 7

Preprocessor

7.1 What is a preprocessor?

A preprocessor is a tool used by a compiler to transform a program before actual compilation. The facilities a preprocessor provides may vary, but the four most common functions a preprocessor could provide are:

- header file inclusion
- conditional compilation
- macro expansion
- line control

Header file inclusion is the substitution of files for include declarations (in the C preprocessor this is the `#include` directive). *Conditional compilation* provides a mechanism to include and exclude parts of a program based on various conditions (in the C preprocessor this can be done with `#define` directives). *Macro expansion* is probably the most powerful feature of a preprocessor. Macros are short abbreviations of longer program constructions. The preprocessor replaces these macros with their definition throughout the program (in the C preprocessor a macro is specified with `#define`). *Line control* is used to inform the compiler where a source line originally came from when different source files are combined into an intermediate file. Some preprocessors also remove comments from the source file, though it is also perfectly acceptable to do this in the lexical analyzer.

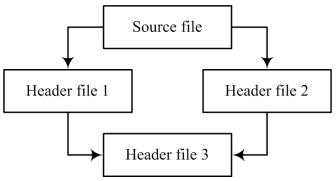
7.2 Features of the Inger preprocessor

The preprocessor in Inger only supports header file inclusion for now. In the near future other preprocessor facilities may be added, but due to time constraints header file inclusion is the only feature. The preprocessor directives in Inger always start at the beginning of a line with a `#`, just like the C preprocessor. The directive for header inclusion is `#import` followed by the name of the file to include between quotes.

7.2.1 Multiple file inclusion

Multiple inclusion of the same header might give some problems. In C we prevent this through conditional compiling with a `#define` or with a `#pragma` once directive. The Inger preprocessor automatically prevents multiple inclusion by keeping a list of files that are already included for this source file.

Example 7.1 (Multiple inclusion)



Multiple inclusion – this should be perfectly acceptable for a programmer so no warning is shown, though hdrfile3 is included only once.

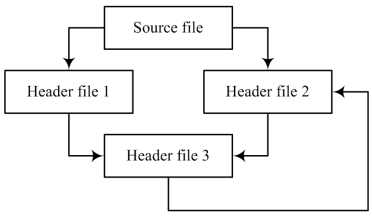
□

Forcing the user not to include files more than once is not an option since sometimes multiple header files just need the same other header file. This could be solved by introducing conditional compiling into the preprocessor and have the programmers solve it themselves, but it would be nice if it happened automatically so the Inger preprocessor keeps track of included files to prevent it.

7.2.2 Circular References

Another problem that arises from header files including other header files is the problem of circular references. Again unlike the C preprocessor, the Inger preprocessor detects circular references and shows a warning while ignoring the circular include.

Example 7.2 (Circular References)



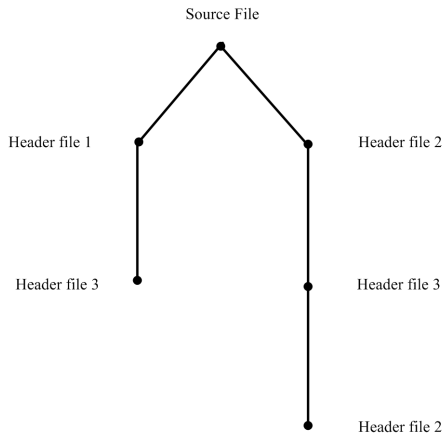
Circular inclusion – this always means that there is an error in the source so the preprocessor gives a warning and the second inclusion of hdrfile2 is ignored.

□

This is realized by building a tree structure of includes. Everytime a new file is to be included, the tree is checked upwards to the root node, to see if this file has already been included. When a file already has been included the preprocessor shows a warning and the import directive is ignored. Because every

include creates a new child node in the tree, the preprocessor is able to distinct between a multiple inclusion and a circular inclusion by only going up in the tree.

Example 7.3 (Include tree)

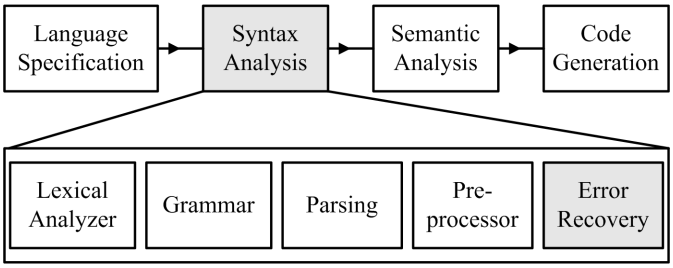


Include tree structure – for every inclusion, a new child node is added. This example shows how the circular inclusion for header 2 is detected by going upwards in the tree, while the multiple inclusion of header 3 is not seen as a circular inclusion because it is in a different branch.

□

Chapter 8

Error Recovery



As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programmes." - Maurice Wilkes discovers debugging, 1949.

8.1 Introduction

Almost no programs are ever written from scratch that contain no errors at all. Since programming languages, as opposed to natural languages, have very rigid syntax rules, it is very hard to write error-free code for a complex algorithm on the first attempt. This is why compilers must have excellent features for error handling. Most programs will require several runs of the compiler before they are free of errors.

Error detection is a very important aspect of the compiler; it is the outward face of the compiler and the most important bit that the user will become rapidly familiar with. It is therefore imperative that error messages be clear, correct and, above all, useful. The user should not have to look up additional error information is a dusty manual, but the nature of the error should be clear from the information that the compiler gives.

This chapter discusses the different natures of errors, and shows ways detecting, reporting and recovering from errors.

8.2 Error handling

Parsing is all about detecting syntax errors and displaying them in the most useful manner possible. For every compiler run, we want the parser to detect and display as many syntax errors as it can find, to alleviate the need for the user to run the compiler over and over, correcting the syntax errors one by one.

There are three stages in error handling:

- Detection
- Reporting
- Recovery

The detection of errors will happen during compilation or during execution. Compile-time errors are detected by the compiler during translation. Runtime errors are detected by the operating system in conjunction with the hardware (such as a division by zero error). Compile-time errors are the only errors that the compiler should have to worry about, although it should make an effort to detect and warn about all the potential runtime errors that it can. Once an error is detected, it must be reported both to the user and to a function which will process the error. The user must be informed about the nature of the error and its location (its line number, and possibly its character number).

The last and also most difficult task that the compiler faces is recovering from the error. Recovery means returning the compiler to a position in which it is able to resume parsing normally, so that subsequent errors do not result from the original error.

8.3 Error detection

The first stage of error handling is detection which is divided into compile-time detection and runtime detection. Runtime detection is not part of the compiler so therefore it will not be discussed here. Compile-time however is divided into four stages which will be discussed here. These stages are:

- Detecting lexical errors
- Detecting syntactic errors
- Detecting semantic errors
- Detecting compiler errors

- *Detecting lexical errors* Normally the scanner reports a lexical error when it encounters an input character that cannot be the first character of any lexical token. In other words, an error is signalled when the scanner is unfamiliar with a token found in the input stream. Sometimes, however, it is appropriate to recognize a specific sequence of input characters as an invalid token. An example of a error detected by the scanner is an unterminated comment. The scanner must remove all comments from the source code. It is not correct, of course, to begin a comment but never terminate it. The scanner will reach the end of the source file before it encounters the end of the comment. Another (similar) example is when the scanner is unable to determine the end of a string.

Lexical errors also include the error class known as overflow errors. Most languages include the integer type, which accepts integer numbers of a certain bit length (32 bits on Intel x86 machines). Integer numbers that exceed the maximum bit length generate a lexical error. These errors cannot be detected using regular expressions, since regular expressions cannot interpret the value of a token, but only calculate its length. The lexer can rule that no integer number can be longer than 10 digits, but that would mean that 000000000000001 is not a valid integer number (although it is!). Rather, the lexer must verify that the literal value of the integer number does not exceed the maximum bit length using a so-called lexical action. When the lexer matches the complete token and is about to return it to the parser, it verifies that the token does not overflow. If it does, the lexer reports a lexical error and returns zero (as a placeholder value). Parsing may continue as normal.

- *Detecting syntactic errors* The parser uses the grammar's production rules to determine which tokens it expects the lexer to pass to it. Every nonterminal has a FIRST set, which is the set of all the terminal tokens that the nonterminal be replaced with, and a FOLLOW set, which is the set of all the terminal tokens that may appear after the nonterminal. After receiving a terminal token from the lexical analyzer, the parser must check that it matches the FIRST set of the nonterminal it is currently evaluating. If so, then it continues with its normal processing, otherwise the normal routine of the parser is interrupted and an error processing function is called.

- *Detecting semantic errors* Semantic errors are detected by the action routines called within the parser. For example, when a variable is encountered it must have an entry in the symbol table. Or when the value of variable `"a"` is assigned to variable `"b"` they must be of the same type.

- *Detecting compiler errors* The last category of compile-time errors deals with malfunctions within the compiler itself. A correct program could be incorrected compiled because of a bug in the compiler. The only thing the user can do is report the error to the system staff. To make the compiler as error-free as possible, it contains extensive self-tests.

8.4 Error reporting

Once an error is detected, it must be reported to the user and to the error handling function. Typically, the user recieves one or more messages that report the error. Errors messages displayed to the user must obey a few style rules, so that they may be clear

and easy to understand.

1. The message should be specific, pinpointing the place in the program where the error was detected as closely as possible. Some compilers include only the line number in the source file on which the error occurred, while others are able to highlight the character position in the line containing the error, making the error easier to find.

2. The messages should be written in clear and complete English sentences, never in cryptic terms. Never list just a message code number such as `"error number 33"` forcing the user to refer to a manual.

3. The message should not be redundant. For example when a variable is not declared, it is not be nessesary to print that fact each time the variable is referenced.

4. The messages should indicate the nature of the error discovered. For example, if a colon were expected but not found, then the message should just say that and not just `"syntax error"` or

   `"missing symbol"`.

5. It must be clear that the given error is actually an error (so that the compiler did not generate an executable), or that the message is a warning (and an executable may still be generated).

Example 8.1 (Error Reporting)

Error

The source code contains an overflowing integer value (e.g. 1234578901234567890).

Response

This error may be treated as a warning, since compilation can still take place. The offending overflowing value will be replaced with some neutral value (say, zero) and this fact should be reported:

```
test.i (43): warning: integer value overflow
(12345678901234567890). Replaced with 0.
```

Error

The source code is missing the keyword THEN where it is expected according to the grammar.

Response

This error cannot be treated as a warning, since an essential piece of code can not be compiled. The location of the error must be pinpointed so that the user can easily correct it:

```
test.i (54): error: THEN expected after IF condition.
```

Note that the compiler must now recover from the error; obviously an important part of the IF statement is missing and it must be skipped somehow. More information on error recovery will follow below.

□

8.5 Error recovery

There are three ways to perform error recovery:

1. When an error is found, the parser stops and does not attempt to find other errors.
2. When an error is found, the parser reports the error and continues parsing. No attempt is made at error correction (recovery), so the next errors may be irrelevant because they are caused by the first error.
3. When an error is found, the parser reports it and recovers from the error, so that subsequent errors do not result from the original error. This is the method discussed below.

Any of these three approaches may be used (and have been), but it should be obvious that approach 3 is most useful to the programmer using the compiler. Compiling a large source program may take a long time, so it is advantageous to have the compiler report multiple errors at once. The user may then correct all errors at his leisure.

8.6 Synchronization

Error recovery uses so-called synchronization points that the parser looks for after an error has been detected. A synchronization point is a location in the source code from which the parser can safely continue parsing without printing further errors resulting from the original error.

Error recovery uses two sets of terminal tokens, the so-called direction sets:

1. The *FIRST* set - is the set of all terminal symbols with which the strings, generated by all the productions for this nonterminal begin.
2. The *FOLLOW* set - a set of all terminal symbols that can be generated by the grammar directly after the current nonterminal.

As an example for direction sets, we will consider the following very simple grammar and show how the FIRST and FOLLOW sets may be constructed for it.

number:	digit morenumber.
morenumber:	digit morenumber.
morenumber:	ϵ.
digit :	**0**.
digit :	**1**.

Any nonterminal has at least one, but frequently more than one production rule. Every production rule has its own FIRST set, which we will call PFIRST. The PFIRST set for a production rule contains all the leftmost terminal tokens that the production rule may eventually produce. The FIRST set of any nonterminal is the union of all its PFIRST sets. We will now construct the FIRST and PFIRST sets for our sample grammar.

PFIRST sets for every production:

number:	digit morenumber.	PFIRST = { **0**, **1** }
morenumber:	digit morenumber.	PFIRST = { **0**, **1** }
morenumber:	ϵ.	PFIRST = { }
digit :	**0**.	PFIRST = { **0** }
digit :	**1**.	PFIRST = { **1** }

FIRST sets per terminal:

FIRST(number) = { **0**, **1** }
FIRST(morenumber) = { **0**, **1** } V { } = { **0**, **1** }
FIRST(digit) = { **0** } V { **1** } = { **0**, **1** }

Practical advice 8.1 (Construction of PFIRST sets)

PFIRST sets may be most easily constructed by working from bottom to top: find the PFIRST sets for 'digit' first (these are easy since the production rules for digit contain only terminal tokens). When finding the PFIRST set for a production rule higher up (such as number), combine the FIRST sets of the nonterminals it uses (in the case of number, that is digit). These make up the PFIRST set.

■

Every nonterminal must also have a FOLLOW set. A FOLLOW set contains all the terminal tokens that the grammar accepts after the nonterminal to which the FOLLOW set belongs. To illustrate this, we will now determine the FOLLOW sets for our sample grammar.

number:	digit morenumber.
morenumber:	digit morenumber.
morenumber:	ϵ.
digit :	**0**.
digit :	**1**.

FOLLOW sets for every nonterminal:

FOLLOW(number) = { EOF }
FOLLOW(morenumber) = { EOF }
FOLLOW(digit} = { EOF, **0**, **1** }

The terminal tokens in these two sets are the synchronization points. After the parser detects and displays an error, it must synchronize (recover from the error). The parser does this by ignoring all further tokens until it reads a token that occurs in a synchronization point set, after which parsing is resumed. This point is best illustrated by a example, describing a Sync routine. Please refer to listing 8.1.

```
/* Forward declarations. */
/* If current token is not in FIRST set, display
 * specified error.
 * Skip tokens until current token is in FIRST
 * or in FOLLOW set.
 * Return TRUE if token is in FIRST set, FALSE
 * if it is in FOLLOW set.
 */
BOOL Sync( int first [], int follow [], char *error )
{
    if ( !Element( token, first ) )
    {
        AddPosError( error, lineCount, charPos );
    }

    while( !Element( token, first ) && !Element( token, follow ) )
    {
        GetToken();
        /* If EOF reached, stop requesting tokens and just
         * exit, claiming that the current token is not
         * in the FIRST set. */
        if ( token == 0 )
        {
            return( FALSE );
        }
    }

    /* Return TRUE if token in FIRST set, FALSE
     * if token in FOLLOW set.
     */
    return( Element( token, first ) );
}
```

Listing 8.1: Sync routine

```
/* Call this when an unexpected token occurs halfway a
 * nonterminal function. It prints an error, then
 * skips tokens until it reaches an element of the
 * current nonterminal's FOLLOW set. */
void SyncOut( int follow [] )
{
    /* Skip tokens until current token is in FOLLOW set. */
    while ( !Element( token, follow ) )
    {
        GetToken();
        /* If EOF is reached, stop requesting tokens and
         * exit. */
        if ( token == 0 ) return;
    }
}
```

Listing 8.2: SyncOut routine

Tokens are requested from the lexer and discarded until a token occurs in one of the synchronization point lists.

At the beginning of each production function in the parser the FIRST and FOLLOW sets are filled. Then the function Sync should be called to check if the token given by the lexer is available in the FIRST or FOLLOW set. If not then the compiler must display the error and search for a token that is part of the FIRST or FOLLOW set of the current production. This is the synchronization point. From here on we can start checking for other errors.

It is possible that an unexpected token is encountered halfway a nonterminal function. When this happens, it is nessesary to synchronize until a token of the FOLLOW set is found. The function SyncOut provides this functionality (see listing 8.2).

Morgan (1970) claims that up to 80% of the spelling errors occurring in student programs may be corrected in this fashion.

Part III

Semantics

We are now in a position to continue to the next level and take a look at the shady side of compiler construction; *semantics*. This part of the book will provide answers to questions like: What are semantics good for? What is the difference between syntax and semantics? Which checks are performed? What is typechecking? And what is a *symbol table*? In other words, this chapter will unleash the riddles of *semantic analysis*. Firstly it is important to know the difference between syntax and semantics. Syntax is the grammatical arrangement of words or tokens in a language which establishes their necessary relations. Hence, *syntax analysis* checks the correctness of the relation between elements of a sentence. Let's explain this with an example using a *natural language*. The sentence

```
Loud purple flowers talk
```

is incorrect according to the English grammar, hence the syntax of the sentence is flawed. This means that the relation between the words is incorrect due to its bad syntactical construction which results in a meaningless sentence.

Whereas *syntax* is about the relation between elements of a sentence, semantics is concerned with the meaning of the production. The relation of the elements in a sentence can be right, while the construction as a whole has no meaning at all. The sentence

```
Purple flowers talk loud
```

is correct according to the English grammar, but the meaning of the sentence is not flawless at all since purple flowers cannot talk! At least, not yet. The semantic analysis checks for meaningless constructions, erroneous productions which could have multiple meanings and generates *error* en *warning* messages.

When we apply this theory on programming languages we see that *syntax analysis* finds syntax errors such as typos and invalid constructions such as illegal variable names. However, it is possible to write programs that are syntactically correct, but still violate the rules of the language. For example, the following sample code conforms to the Inger syntax, but is invalid nonetheless: we cannot assign a value to a function.

```
myFunc() = 6;
```

The *semantic analysis* is of great importance since code with assignments like this may act strange when executing the program after successful compilation. If the program above does not crash with a *segmentation fault* on execution and apparently executes the way it should, there is a chance that something fishy is going on: it is possible that a new address is assigned to the function myFunc(), or not? We do not assume [1] that everything will work the way we think it will work.

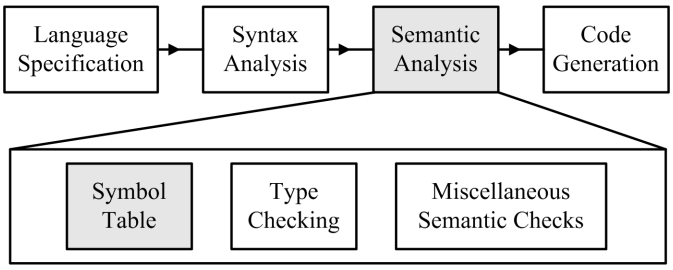
Some things are too complex for *syntax analysis*, this is where *semantic analysis* comes in. *Type checking* is necessary because we cannot force correct use of *types* in the syntax because too much additional information is needed. This additional information, like (return) types, will be available to the *type checker* stored in the *AST* and *symbol table*.

Let us begin, with the *symbol table*.

[1] *Tip 27 from the Pragrammatic Programmer [1]: Don't Assume It - Prove It* Prove your assumptions in the actual environment - with real data and boundary conditions

Chapter 9

Symbol table



9.1 Introduction to symbol identification

At compile time we need to keep track of the different symbols (functions and variables) declared in the program source code. This is a requirement because every *symbol* must be identifiable with a unique name so we can find the symbols back later.

Example 9.1 (An Incorrect Program)

```
module flawed;
start main: void → void
{
    2 ∗ 4;           // result is lost
    myfunction ( 3 ); // function does not exist
}
```

□

Without *symbol identification* it is impossible to make assignments, or define functions. After all, *where* do we assign the value to? And how can we call a function if it has no name? We do not think we would make many programmers happy if they could only reference values and functions through memory addresses. In the example the mathematical production yields `8`, but the result is not assigned to a uniquely identified *symbol*. The call to `myfunction` yields a

compiler error as myfunction is not *declared* anywhere in the program source so there is no way for the compiler to know what code the programmer actually wishes to call. This explains only *why* we need *symbol identification* but does not yet tell anything practical about the subject.

9.2 Scoping

We would first like to introduce *scoping*. What actually is *scoping*? In Webster's Revised Unabridged Dictionary (1913) a scope is defined as:

> *Room or opportunity for free outlook or aim; space for action; amplitude of opportunity; free course or vent; liberty; range of view, intent, or action.*

When discussing scoping in the context of a programming language the description comes closest to Webster's *range of view*. A scope limits the *view* a statement or expression has when it comes to other symbols. Let us illustrate this with an example in which every block, delimited by { and }, results in a new scope.

Example 9.2 (A simple scope example)

```
module example; // begin scope (global)

int a = 4;
int b = 3;

start main: void → void
{ // begin scope (main)
  float a = 0;

  { // begin scope (free block}
    char a = 'a';
    int x;
    print( a );
  } // end of scope {free block}
  x = 1; // x is not in range of view!
  print( b );
} // end of scope (main)

// end of scope 0 (global)
```

□

Example 9.2 contains an Inger program with 3 nested scopes containing variable declarations. Note that there are 3 declarations for variables named a, which is perfectly legal as each declaration is made in a scope that did not yet contain a symbol called 'a' of its own. Inger only allows referencing of symbols

in the local scope and (grand)parent scopes. The expression `x = 1;` is illegal since x was declared in a scope that could be best described as a *nephew* scope.

Using identification in scopes enables us to declare a symbol name multiple times in the same program but in different scopes, this implicates that a *symbol* is *unique* in its own *scope* and not necessarily in the program, this way we do not run out of useful variable names within a program.

Now that we know what scoping is, it is probably best to continue with some theory on how to store information about these scopes and their symbols during compile time.

9.3 The Symbol Table

Symbols collected during parsing must be stored for later reference during *semantic analysis*. In order to have access to the symbols during in a later stage it is important to define a clear data structure in which we store the necessary information about the symbols and the scopes they are in. This data structure is called a *Symbol Table* and can be implemented in a variety of ways (e.g. *arrays*, *linked lists*, *hash tables*, *binary search trees* & *n-ary search trees*). Later on in this chapter we will discuss what data structure we considered best for our symbol table implementation.

9.3.1 Dynamic vs. Static

There are two possible types of symbol tables, dynamic or static symbol tables. What exactly are the differences between the two types, and why is one better than the other? A dynamic symbol table can only be used when both the gathering of symbol information and usage of this information happen in one pass. It works in a stack like manner: a symbol is pushed on the stack when it is encountered. When a scope is left, we pop all symbols which belong to that scope from the stack. A static table is built once and can be walked as many times as required. It is only deconstructed when the compiler is finished.

Example 9.3 (Example of a dynamic vs. static symbol table)

```
module example;

int v1, v2;

f: int v1, v2 → int
{
    return (v1 + v2);
}

start g: int v3 → int
```

```
{
    return  (v1 + v3);
}
```

The following example illustrates now a dynamic table grows and shrinks over time.
After line 3 the symbol table is a set of symbols

$$T = \{v1, v2\}$$

After line 5 the symbol table also contains the function `f` and the local variables `v1` and `v2`

$$T = \{v1, v2, f, v1, v2\}$$

At line 9 the symbol table is back to its global form

$$T = \{v1, v2\}$$

After line 10 the symbol table is expanded with the function `g` and the local variable `v3`

$$T = \{v1, v2, g, v3\}$$

Next we illustrate now a static table's set of symbols only grows.
After line 3 the symbol table is a set of symbols

$$T = \{v1, v2\}$$

After line 5 the symbol table also contains the function `f` and the local variables `v1` and `v2`

$$T = \{v1, v2, f, v1, v2\}$$

After line 10 the symbol table is expanded with the function `g` and the local variable `v3`

$$T = \{v1, v2, f, v1, v2, g, v3\}$$

□

In earlier stages of our research we assumed local symbols should only be present in the symbol table when the scope it is declared in is being processed (this includes, off course, children of that scope). After all, storing a symbol that is no longer accessable seems pointless. This assumption originated when we were working on the idea of a single-pass compiler (tokenizing, parsing, semantics and code generation all in one single run) so for a while we headed in the direction of a dynamic symbol table. Later on we decided to make the compiler multi-pass which resulted in the need to store symbols over a longer timespan. Using multiple passes, the local symbols should remain available for as long as the compiler lives as they might be needed every pass around, thus we decided to switch to a static symbol table.

When using a static symbol table, the table will not shrink but only grow. Instead of building the symbol table during pass 1 which happens when using a dynamic table, we will construct the symbol table from the AST. The AST will be available after parsing the source code.

9.4 Data structure selection

9.4.1 Criteria

In order to choose the right data structure for implementing the symbol table we look at its primary goal and what the criteria are. Its primary goal is storing symbol information an provide a fast lookup facility, for easy access to all the stored symbol information.

Since we have only a short period of time to develop our language Inger and the compiler, the only criteria we had in choosing a suitable data structure was that it was easy to use, and implement.

9.4.2 Data structures compared

One of the first possible data structures which comes to mind when thinking of how to store a list of symbols is perhaps an array. Although an array is very convenient to store symbols, it has quite a few practical limitations. Arrays are defined with a static size, so chances are you define a symbol table with array size 256, and end up with 257 symbols causing a buffer overflow (an internal compiler error). Writing beyond an array limit will produce unpredictable results. Needless to say, this situation is undesirable. Searching an array is not efficient at all due to its linear searching algorithm. A binary search algorithm could be applied, but this would require the array to be sorted at all times. For sorted arrays, searching is a fairly straightforward operation and easy to implement. It is also notable that an array would probably only be usable when using either a dynamic table or if no scoping would be allowed.

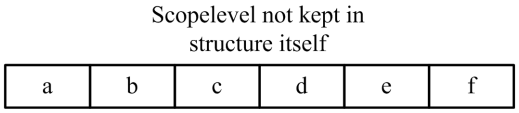


Figure 9.1: Array

If the table would be implemented as a stack, finding a symbol is a simple matter of searching for the desired symbol from the top of the stack. The first variable found is automatically the last variable added and thus the variable in the nearest scope. This implementation makes it easy to use multi-level scoping, but is a heavy burden on performance as with every search the stack has to be deconstructed and stored on a second stack (until the first occurrence is found) and reconstructed, a very expensive operation. This implementation would probably be the best for a dynamic table if it were not for its expensive search operations.

Another dynamic implementation would be a linked list of symbols. If implemented as an unsorted double linked list it would be possible to use it just

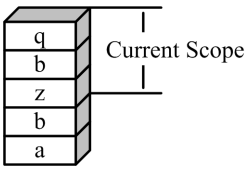


Figure 9.2: Stack

like the stack (append to the back and search from the back) without the disadvantage of a lot of push and pop operations. A search still takes place in a linear time frame but the operations themselves are much cheaper than the stack implementation.

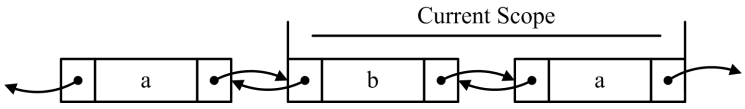


Figure 9.3: Linked List

Binary search trees improve search time massively, but only in sorted form (an unsorted tree after all, is not a tree at all). This results in the loss of an advantage the stack and double linked list offered: easy scoping. Now the first symbol found is not per definition the latest definition of that symbol name. It in fact is probably the first occurrence. This means that the search is not complete until it is impossible to find another occurrence. This also means that we have to include some sort of scope field with every symbol to separate the symbols: (a,1) and (a,2) are symbols of the same name, but a is in a higher scope and therefore the correct symbol. Another big disadvantage is that when a function is processed we need to rid the tree of all symbols in that function's scope. This requires a complete search and rebalancing of the tree. Since the tree is sorted by string value every operation (insert, search, etc...) is quite expensive. These operations could be made more time efficient by using an hash algorithm as explained in the next paragraph.

String comparisons are relatively heavy compared to comparison of simple types such as integers or bytes so using an hash algorithm to convert symbol names to simple types would speed all operations on the tree considerably.

The last option we discuss is the n-ary tree. Every node has n children each of which implicate a new scope as a child of its parent scope. Every node is a scope and all symbols in that scope are stored inside that node. When the AST is walked, all the code has to do is make sure that the symbol table walks along. Then when information about a symbol is requested, we only have to search the current scope and its (grand) parents. This seems in our opinion to be the only valid static symbol table implementation.

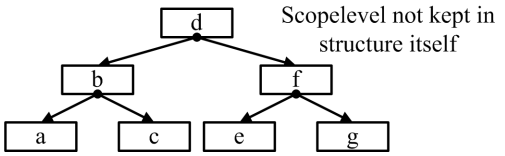


Figure 9.4: Binary Tree

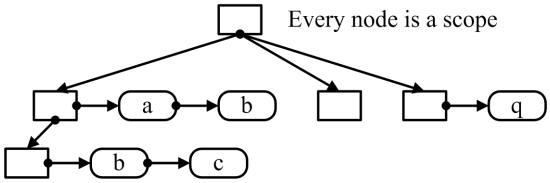


Figure 9.5: N-ary Tree

9.4.3 Data structure selection

We think a combination of an n-ary tree combined with linked lists is a suitable solution for us. Initially we thought using just one linked list was a good idea. Each list node, representing a scope, would contain the root node of a binary tree. The major advantage of this approach is that adding and removing a scope was easy and fast. This advantage was based on the idea that the symbol table would grow and shrink during the first pass of compilation, which means that the symbol table is not available anymore after the first pass. This is not what we want since the symbol table must be available at all times after the first pass and therefore favour a new data structure that is less efficient in removing scopes (we do not remove scopes anyway) but faster in looking up symbols in the symbol table.

9.5 Types

The symbol table data structure is not enough, it is just a tree and should be decorated with symbol information like (return) types and modifiers. To store this information correctly we designed several logical structures for symbols and types. It basicly comes down to a set of functions which wrap a Type structure. These functions are for example: CreateType(), AddSimpleType(), AddDimension(), AddModifier(), etc.... There is a similar set of accessor functions.

9.6 An Example

To illustrate how the symbol table is filled from the *Abstract Syntax Tree* we show which steps have to be taken to fill the symbol table.

1. Start walking at the root node of the AST in a pre-order fashion.

2. For each block we encounter we add a new child to the *current scope* and make this child the new *current scope*
3. For each variable declaration found, we extract:
   - Variable name
   - Variable type[1]
4. For each function found, we extract:
   - Function name
   - Function types, starting with the return type[1]
5. After the end of a block is encountered we move back to the parent scope.

To conclude we will show a simple example.

Example 9.4 (Simple program to test expressions)

```
module test_module;

int z = 0;

inc : int a → int
{
    return( a + 1 );
}

start main: void → void
{
    int i;
    i = (z * 5) / 10 + 20 - 5 * (132 + inc ( 3 ) );
}
```

□

We can distinguish the following steps in parsing the example source code.

1. found z, add symbol to current scope (global)
2. found inc, add symbol to current scope (global)
3. enter a new scope level as we now parse the function inc
4. found the parameter a, add this symbol to the current scope (inc)
5. as no new symbols are encountered, leave this scope
6. found main, add symbol to current scope (global)

[1] for every type we also store optional information such as modifiers (start, extern, etc. . .) and dimensions (for pointers and arrays)

7. enter a new scope level as we now parse the function main
8. found i, add symbol to current scope (main)
9. as no new symbols are encountered, leave this scope

After these steps our symbol table will look like this.

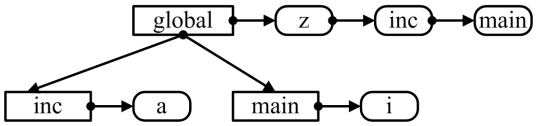
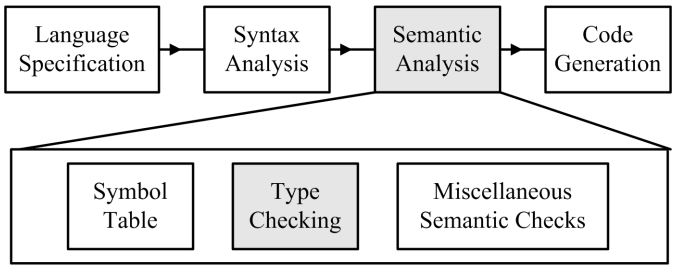


Figure 9.6: Conclusion

Chapter 10

Type Checking



10.1 Introduction

Type checking is part of the symantic analysis. The purpose of type checking is to evaluate each operator, application and return statement in the AST (Abstract Syntax Tree) and search for its operands or arguments. The operands or arguments must both be of compatible types and form a valid combination with the operator. For instance: when the operator is +, the left operand is a integer and the right operand is a char pointer, it is not a valid addition. You can not add a char pointer to an integer without explicit coercion.

The type checker evaluates all nodes where types are used in an expression and produces an error when it cannot find a decent solution (through *coercion*) to a type conflict.

Type checking is one of the the last steps to detect semantic errors in the source code. After there are a few symantic checks left before code generation can commence.

This chapter discusses the process of type checking, how to modify the AST by including type info and produce proper error messages when necessary.

10.2 Implementation

The process of type checking consists of two parts:

- Decorate the AST with types for literals.
- Propagate these types up the tree taking into account:

- Type correctness for operators
- Type correctness for function arguments
- Type correctness for **return** statements
- If types do not match in their simple form (**int**, **float** etc...) try to coerce these types.

- Perform a last type check to make sure indirection levels are correct (e.g. assigning an **int** to a pointer variable.

10.2.1 Decorate the AST with types

To decorate the AST with types it is advisable to walk post-order through the AST and search for all literal identifiers or values. When a literal is found in the AST the type must be located in the symbol table. Therefore it is necessary when walking through the AST to keep track of the current scope level. The symbol table provides the information of the literal (all we are interested in is the type) and this will be stored in the AST.

The second step is to move up in the AST and evaluate types for unary, binary and application nodes.

The 10.1 illustrates the process of expanding the tree. It shows the AST decoration process for the expression `a = b + 1;`. The variable `a` and `b` are both declared as an integer.

Example 10.1 (Decorating the AST with types.)

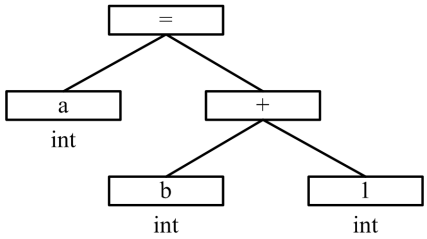


Figure 10.1: AST type expanding

The nodes `a`, `b` and `1` are the literals. These are the first nodes we encounter when walking post-order through the tree. The second part is to determine the types of node + and node =. After we passed the the literals `b` and `1` we arrive at node +. Because we have already determined the type of its left and right child we can evaluate its type. In this case the outcome (futher referred to in the text as the *result type*) is easy. Because node `b` and `1` are both of the type `int`, node + will also become an `int`.

Because we are still walking post-order through the AST we finally arrive at node =. The right and left child are also both integers so this node will also become an integer.

□

The advantage by walking post-order through the AST is that all the type-checking can be done in one pass. If you were to walk pre-order through the AST it would be advisible to decorate the AST with types in two passes. The first pass should walk pre-order through the AST and decorate only the literal nodes, and the second pass which walks pre-order through the AST evaluates the parent nodes from the literals. This cannot be done in one pass because the first time walking pre-order through the AST you will first encounter the = node. When you try to evaluate its type you will find that the children do not have a type.

The above example was easy; all the literals were integers so the result type will also be an integer. But what would happen if one of the literals was a float and the others are all integers.

One way of dealing with this problem is to create a table with conversion priorities. When for example a float and an int are located, the highest priority operator wins. These priorities can be found in the table for each operator. For an example of this table, see table 10.1. In this table the binary operators assign = and add + are implemented. The final version of this table has all binary implemented. The same goes for all unary operators like the not (!) operator.

Node	**Type**
NODE_ASSIGN	FLOAT
NODE_ASSIGN	INT
NODE_ASSIGN	CHAR
NODE_BINARY_ADD	FLOAT
NODE_BINARY_ADD	INT

Table 10.1: Conversion priorities

The highest priority is on top for each operator.

The second part is to make a list of types which can be converted for each operator. In the Inger language it is possible to convert an integer to a float but the conversion from integer to string is not possible. This table is called the coercion table. For an example see table 10.2.

From type	**To type**	**New node**
INT	FLOAT	NODE_INT_TO_FLOAT
CHAR	INT	NODE_CHAR_TO_INT
CHAR	FLOAT	NODE_CHAR_TO_FLOAT

Table 10.2: Coercion table

A concrete example is explained in section 10.2. It shows the AST for the expression `a = b + 1.0;`. The variable `a` and is declared as a float and `b` is declared by the type of integer. The literal `1.0` is also a float.

Example 10.2 (float versus int.)

The literals `a`, `b` and `1.0` are all looked up in the symbol table. The variable `a` and the literal `1.0` are both floats. The variable `b` is an integer. Because we

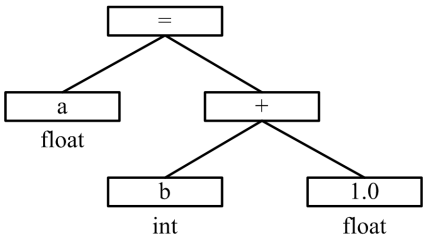


Figure 10.2: Float versus int

are walking post-order through the AST the first operator we encounter is the +. Operator + has as its left child an integer and the right child is a float. Now it is time to use the lookup table to find out of what type the + operator must be. It appears that the first entry for the operator + in the lookup table is of the type float. This type has the highest priority. Because one of the two types is also a float, the result type for the operator + will be a float.

It is still nessesary to check if the other child can be converted to the float. If not, an error message should appear on the screen.

The second operator is the operator =. This will be exactly the same process as for the + operator. The left child (`a`) is of type float and the right child + of type float so operator = will also become a float.

However, what would happen if the left child of the assignment operator = was an integer? Normally the result type should be looked up in the table 10.1, but in case of a assignment there is an exception. For the assignment operator = its right child determines the result. So if the left child is an integer, the assignment operator will also become an integer. When you declare a variable as an integer and an assignment takes place of which the right child differs from the original declared type, an error must occur. It is not possible to change the original declaration type of any variable. This is the only operator exception you should take care of.

We just illustrated an example of what would happen if two different types are encountered, belonging to the same operator. After the complete pass the AST is decorated with types and finally looks like 10.3.

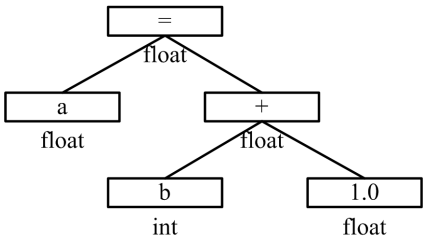


Figure 10.3: Float versus int result

□

10.2.2 Coercion

After the decoration of the AST is complete, and all the checks are executed the main goal of the typechecker mudule is achieved. At this point it is nessesary to make a choice. There are two ways to continue, the first way is to start with the code generation. The type checking module is in this case completely finished. The second way is to prepare the AST tree for the code generation module.

In the first approach, the type checker's responsibility is now finished, and it is up to the code generation module to perform the necessary conversions. In the sample source line

```
int b;
float a = b + 1.0;
```

Listing 10.1: Coercion

the code generation module finds that since a is a float, the result of b + 1.0 must also be a float. This implies that the value of b must be converted to float in order to add 1.0 to it and return the sum. To determine that variable b must be converted to a float it is nessecary to evalute the expression just like the way it is done in the typechecker module.

In the second approach, the typechecking module takes the responsibility to convert the variable b to a float. Because the typechecker module already decorates the AST with all types and therefore concludes any conversion to be made it can easily apply the conversion so the code generation module does not have to repeat the evaluation process.

To prepare the AST for the above problem we have to apply the coercion technique. Coercion means the conversion form one type to another. However it is not possible to convert any given type to any other given type. Since all natural numbers (integers) are elements in the set $\mathbb{N}$ and all real numbers (float) are in the set $\mathbb{R}$ the following formula applies:

$$\mathbb{N} \subset \mathbb{R}$$

A practical application of this theory is it nessesary to modify the AST by adding new nodes. These new nodes are the so called coercion nodes. The best way to explain this is by a practical example. For this example, refer to the source of listing 10.1.

Example 10.3 (coercion)

In the first approach were we let the code generation module take care of the coercion technique, the AST would end up looking like figure 10.3. In the second approach, were the typechecker module takes responsibility for the coercion technique, the AST will have the structure shown in figure 10.4.

Notice that the position of node b is repleaced by node IntToFloat and node b has become a child of node IntToFloat. The node IntToFloat is called the coercion node. When we arrive during the typechecker pass at node +, the left and right child are both evaluated. Because the right child is a float and the right child an integer the outcome must be a float. This is determined by the type lookup table 10.1. Since we now know the result type for node + we can apply the

coercion technique for its childs. This is only required for the child of which the type differs from its parent (node +).

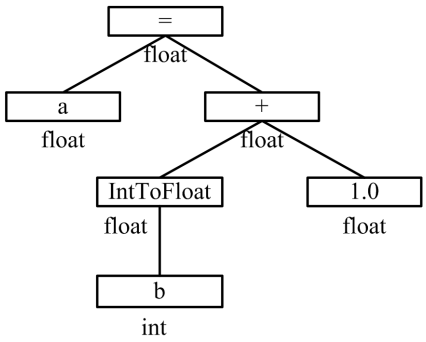


Figure 10.4: AST coercion

When we find a child which type differs from its parent we use the coercion table 10.2 to check if it is possible to convert the type of the child node (node b) to its parent type. If this is not possible an error message must be produced and the compilation progress will stop. When it is possible to apply the conversion it is required to insert a new node in the AST. This node will replace node b and the type becomes a float. Node b will be its child.

□

10.3 Overview.

Now all the steps for the typechecker module are completed. The AST is decorated with types and prepared for the code generation module. Example 10.4 gives a complete display of the AST befor and after the type checking pass.

Example 10.4 (AST decoration)

Consult the sample Inger program in listing 10.2. The AST before decoration is shown in figure 10.5, notice that all types are unknown (no type). The AST after the decoration is shown in figure 10.6.

□

10.3.1 Conclusion

Typechecking is the most important part of the semantic analysis. When the typechecking is completed there could still be some errors in the source. For example

- unreachable code, statements are located after a return keyword. These statements will never be executed;

```
module example;
start f: void → float
{
    float a = 0;
    int b = 0;

    a = b + 1.0;

    return (a);
}
```

Listing 10.2: Sample program listing

- when the function header is declared with a return type other than void, the return keyword must exist in the function body. It will not check if the return type is valid, this already took place in the typechecker pass;
- check for double case labels in a switch;
- lvalue check, when a assignment = is located in the AST its left child can not be a function. This is a rule we applied for the Inger language, other languages may allow this.
- when a goto statement is encountered the label which the goto points at must exists.
- function parameter count, when a function is declared with two parameters (return type excluded), the call to the function must also have two parameters.

All these small checks are also part of the semantic analysis and will be discussed in the next chapter. After these checks are preformed the code generation can finally take place.

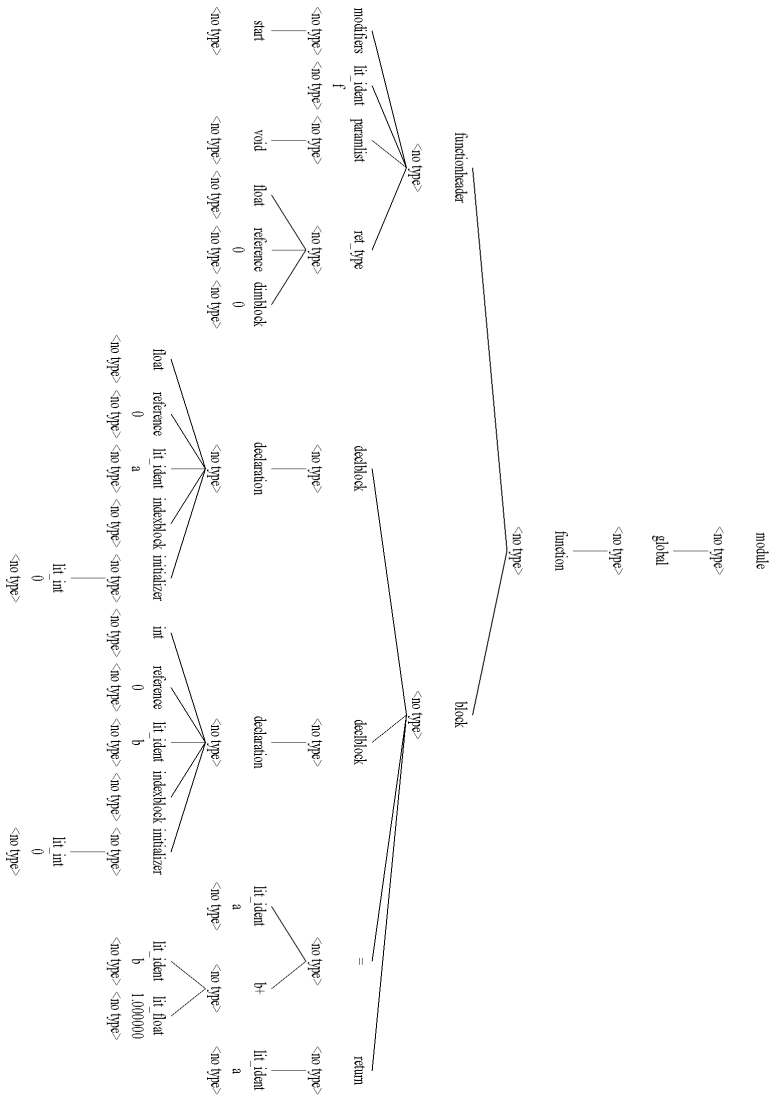


Figure 10.5: AST before decoration

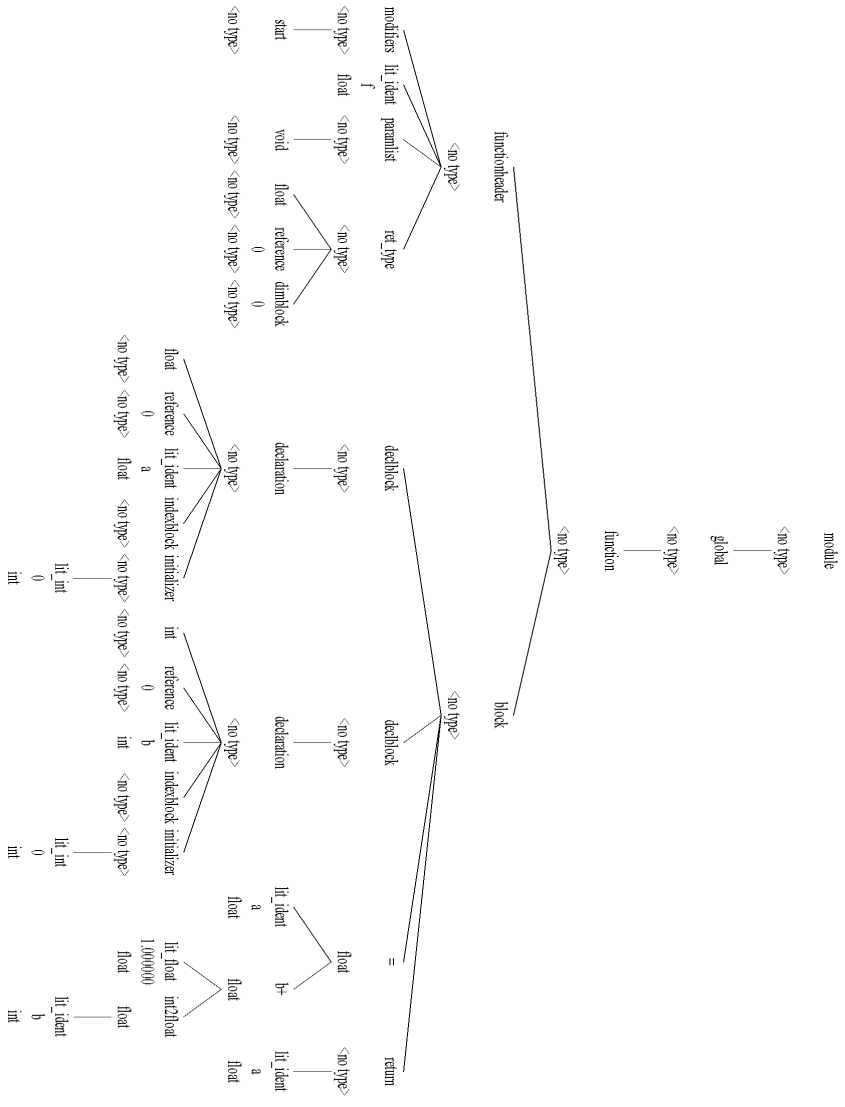


Figure 10.6: AST after decoration

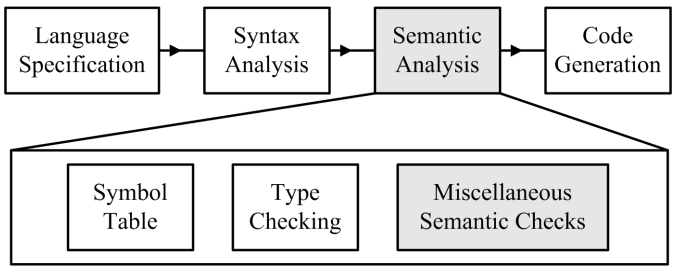
Bibliography

[1] A.B. Pyster: *Compiler Design and Construction*, Van Nostrand Reinhold Company, 1980

[2] G. Goos, J. Hartmanis: *Compiler Construction - An Advanced Course*, Springer-Verlag, Berlin, 1974

Chapter 11

Miscellaneous Semantic Checks



11.1 Left Hand Values

An *lvalue*, short for *left hand value*, is that expression or identifier reference that can be placed on the left hand side of an assignment. The *lvalue check* is one of the necessary checks in the semantic stage. An *lvalue* check makes sure that no invalid assignments are done in the source code. Examples 11.1 and 11.2 show us what lvalues are valid in the Inger compiler and which are not.

Example 11.1 (Invalid Lvalues)

```
function() = 6;
2 = 2;
"somestring" = "somevalue";
```

□

What makes a valid *lvalue*? An *lvalue* must be a modifiable entity. One can define the invalid *lvalues* and check for them, in our case it is better to check for the *lvalues* that *are* valid, because this list is much shorter.

Example 11.2 (Valid Lvalues)

```
int a = 6;
name = "janwillem";
```

□

11.1.1 Check Algorithm

To check the validity of the *lvalues* we need a filled *AST* (*Abstract Syntax Tree*) in order to have access to the all elements of the source code. To get a better grasp of the checking algorithm have a look at the pseudo code in example 11.3. This algorithm results in a list of error messages if any.

Example 11.3 (Check Algorithm)

```
Start at the root of the AST

for each node found in the AST do
  if the node is an '=' operator then
    check its most left child in the AST
    which is the lvalue and see if this is
    a valid one.

    if invalid report an error

    else go to next node
  else go to the next node
```

□

Not all the *lvalues* are as straightforward as they seem. A valid but bizarre example of a semantically correct assignment is:

Example 11.4 (Bizarre Assignment)

```
int a[20];
int b = 4;

a = a * b;
```

□

We choose to make this a valid assignment in Inger and to provide some address arithmetic possibilities. The code in example 11.4 multiplies the base address by the absolute value of identifier b. Lets say that the base address of array a was initially 0x2FFFFFA0, then the base address of a will be 0xBFFFFE80.

11.2 Function Parameters

This section covers argument count checking. Amongst other things, function parameters must be checked before we actually start generating code. Apart from checking the use of a correct number of function arguments in function calls and the occurence of multiple definitions of the *main* function,we also check whether the passed arguments are of the correct type. Argument type checking is explained in 10

The idea of checking the number of arguments passed to a function is pretty straightforward. The check consists of two steps: firstly, we collect all the function header nodes from the *AST* and store them in a list. Secondly, we compare the number of arguments used in each function call to the number of arguments required by each function and check that the numbers match.

To build a list of all nodes that are function headers we make a pass through the *AST* and collect all nodes that are of type NODE_FUNCTIONHEADER, and put them in a list structure provided by the generic *list* module. It is faster to go through the *AST* once and build a list of the *nodes* we need, than to make a pass through the *AST* to look for a node each time we need it. After building the list for the example program 11.5 it will contain the *header nodes* for the functions main and AddOne.

Example 11.5 (Program With Function Headers)

```
module example;

start main : void → void
{
    AddOne( 2 );
}

AddOne : int a → int
{
    return( a + 1 );
}
```

□

The next step is to do a second pass through the *AST* and look for *nodes* of type NODE_APPLICATION which represent a function call in the source code. When such a *node* is found we first retrieve the actual number of arguments passed in the function application with the helper function GetArgumentCount-FromApplication. Secondly we get the number of arguments as defined in the function declaration, to do this we use the function GetArgumentCount. Then it is just a matter of comparing the number of arguments we expect and the number of arguments we found. We only print an error message when a function was called with too many or few arguments.

11.3 Return Keywords

The typechecking mechanism of the Inger compiler checks if a function returns the right type when assigning a function return value to a *variable*.

Example 11.6 (Correct Variable Assignment)

```
int a;
a = myfunction();
```

□

The source code in example 11.6 is correct and implies that the function myfunction returns a value. As in most programming languages we introduced a *return* keyword in our language and we define the following semantic rules: *unreachable code* and *non-void function returns* (definition 11.1 and 11.2).

11.3.1 Unreachable Code

Definition 11.1 (Unreachable code)

> Code after a return keyword in Inger source will not be executed. A warning for this unreachable code will be generated.

■

For this check we run over the *AST* pre-order and check each *code block* for the *return* keyword. If the *child node* containing the *return* keyword is not the last *child node* in the *code block*, the remaining statements will be unreachable; *unreachable code*. An example of *unreachable code* can be found in example 11.7 in which function print takes an *integer* as parameter and prints this to the screen. The *statement* 'print(2)' will never be executed since the function main returns before the print function is reached.

Example 11.7 (Unreachable Code)

```
start main : void → int
{
    int a = 8;

    if ( a == 8 )
    {
        print ( 1 );
        return( a );
        print ( 2 );
    }
}
```

□

Unreachable code is, besides useless, not a problem and the compilation process can continue, therefor a *warning* messages is printed.

11.3.2 Non-void Function Returns

Definition 11.2 (Non-void function returns)

> The last statement in a non-void function should be the keyword 'return' in order to return a value. If the last statement in a non-void function is not 'return' we generate a warning 'control reaches end of non-void function'.

■

It is nice that *unreachable code* is detected, but it is not essential to the next phase the process of compilation. *Non-void function returns*, on the contrary, have a greater impact. Functions that should return a value but never do, can result in an errorneous program. In example 11.8 *variable* a is assigned the result value of function myfunction, but the function myfunction never returns a value.

Example 11.8 (Non-void Function Returns)

```
module functionreturns;

start main : void → void
{
  int a;
  a = myfunction();
}

myfunction : void → int
{
  int b = 2;
}
```

□

To make sure all *non-void function* return, we check for the *return* keyword which should be in the function *code block*. Like with most *semantic checks* we go through the *AST* pre-order and search all function *code block* for the *return* keyword. When a function has a *return statement* in an *if-then-else statement* both *then* and *else* block should contain the *return* keyword because the *code blocks* are executed conditionally. The same is for a *switch block*, all *case block* should contain a *return statement*. All *non-void function* without *return* keyword will generate a warning.

11.4 Duplicate Cases

Generally a *switch statement* has one or more *case blocks*. It is syntactically correct to define multiple *code blocks* with the same *case value*, so-called *duplicate case values*. If *duplicate case values* occur it might not be clear which *code*

block is executed, this is a choice which you should make as a compiler builder. The *semantic check* in Inger generates a warning when a *duplicate case value* is found and generates code for the first *case* code block. We choose to generate a *warning* instead of an *error* message because the multi-value construction still allows us to go to the next phase in compilation; *code generation*. Example program 11.9, will have the output

```
This is the first code block
```

because we choose to generate code for the first *code block* definition for *duplicate case value* 0.

Example 11.9 (Duplicate Case Values)

```
/* Duplicate cases
 * A program with duplicate case values
 */
module duplicate_cases;

start main : void → void
{
  int a = 0;

  switch( a )
  {
    case 0
    {
      printf ( "This is the first case block" );

    }
    case 0
    {
      printf ( "This is the second case block" );

    }
    default
    {
      printf ( "This is the default case" );

    }
  }
}
```

□

The algorithm that checks for *duplicate case values* is pretty simple and works recursively down the AST. It starts at the *root node* of the AST and searches for NODE_SWITCH *nodes*. For each *switch node* found we search for duplicate *children* in the cases block. If any *duplicates* were found, generate a proper *warning*, else continue until the complete AST was searched. In the end this check will detect all *duplicate values* and report them.

11.5 Goto Labels

In the Inger language we implemented the *goto statement* although use of this statement is often considered harmful. Why exactly *is* goto considered harmful? As the late Edsger Dijkstra ([3]) stated:

> *The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program*

Despite its possible harmfulness we decided to implement it. Why? Because it *is* a very cool feature. For the unaware Inger programmer we added a subtle reminder to the keyword *goto* and implemented it as goto_considered_harmful.

As with using *variables*, *goto labels* should be declared before using them. Since this pre-condition cannot be forced using grammar rules (syntax) it should be checked in the semantic stage. Due to a lack of time we did *not* implement this semantic check and therefore programmers and users of the Inger compiler should be aware that jumping to undeclared *goto labels* may result in inexplicable and possibly undesired program behaviour. Example code 11.10 shows the correct way to use the goto keyword.

Example 11.10 (Goto Usage)

```
int n = 10;
label here;
printstr ( n );
n = n - 1;
if ( n > 0 )
{
    goto_considered_harmful here;
}
```

□

A good implementation for this check would be, to store the *label* declarations in the *symbol table* and walk through the *AST* and search for *goto statements*. The identifier in a *goto statement* like

```
goto_considered_harmful labelJumpHere
```

will be looked up in the *symbol table*. If the *goto label* is not found, an *error message* will be generated. Although *goto* is a very cool feature, be careful using it.

Bibliography

[1] A. Hunt, D. Thomas: *The Pragmatic Programmer*, Addison Wesley, 2002

[2] Thomas A. Sudkamp: *Languages And Machines* Second Edition, Addison Wesley, 1997

[3] Edsger W. Dijkstra: *Go To Statement Considered Harmful* http://www.acm.org/classics/oct95/

Part IV

Code Generation

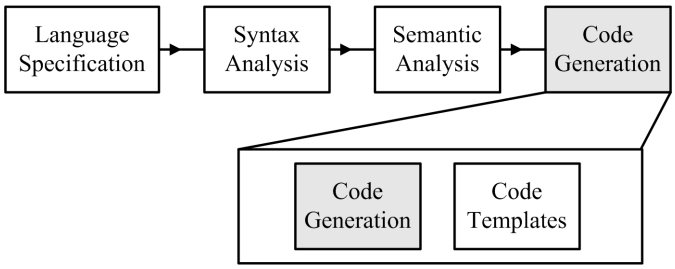
Code generation is the final step in building a compiler. After the semantic analysis there should be no more errors in the source code. If there are still errors then the code generation will almost certainly fail.

This part of the book contains descriptions of how the assembly output will be generated from the Inger source code. The subjects covered in this part include implementation (assembly code) of every operator supported by Inger, storage of data types, calculation of array offsets and function calls, with regard to stack frames and return values.

In the next chapter, code generation is explained at an abstract level. In the final chapter of this book, *code templates*, we present assembly code templates for each operation in Inger. Using templates, we can guarantee that operations can be chained together in any order desired by the programmer, including orders we did not expect.

Chapter 12

Code Generation



12.1 Introduction

Code generation is the least discussed and therefore the most mystical aspect of compiler construction in the literature. It is also not extremely difficult, but requires great attention to detail. The approach using in the Inger compiler is to write a *template* for each operation. For instance, there is a template for addition (the code+ operation), a template for multiplication, dereferencing, function calls and array indexing. All of these templates may be chained together in any order. We can assume that the order is valid, since if the compiler gets to code generation, the input code has passed the syntax analysis and semantic analysis phases. Let's take a look at a small example of using templates.

```
int a = *(b + 0x20);
```

Generating code for this line of Inger code involves the use of four templates. The order of the templates required is determined by the order in which the expression is evaluated, i.e. the order in which the tree nodes in the abstract syntax tree are linked together. By traversing the tree post-order, the first template applied is the template for addition, since the result of `b + 0x20` must be known before anything else can be evaluated. This leads to the following ordering of templates:

1. Addition: calculate the result of `b + 0x20`
2. Dereferencing: find the memory location that the number between braces

points to. This number was, of course, calculated by the previous (inner) template.

3. Declaration: the variable a is declared as an integer, either on the stack (if it is a local variable) or on the heap (if it is a global variable).

4. Assignment: the value delivered by the dereferencing template is stored in the location returned by the declaration template.

If the templates are written carefully enough (and tested well enough), we can create a compiler that suppports and ordering of templates. The question, then, is how templates can be linked together. The answer lies in assigning one register (in casu, eax), as the result register. Every template stores its result in eax, whether it is a value or a pointer. The meaning of the value stored in eax is determined by the template that stored the value.

12.2 Boilerplate Code

Since the Inger compiler generates assembly code, it is necessary to wrap up this code in a format that the assembler expects. We use the GNU AT&T assembler, which uses the AT&T assembly language syntax (a syntax very similar to Intel assembly, but with some peculiar quirks. Take a look at the following assembly instruction, first in Intel assembly syntax:

```
MOV EAX, EBX
```

This instruction copies the value stored in the EBX register into the EAX register. In GNU AT&T syntax:

```
movl %ebx, %eax
```

We note several differences:

1. Register names are written lowercase, and prefixed with a percent (%) sign to indicate that they are registers, not global variable names;

2. The order of the operands is reversed. This is a most irritating property of the AT&T assembly language syntax which is a major source of errors. You have been warned.

3. The instruction mnemonic mov is prefixed with the size of its operands (4 bytes, long). This is similar to Intel's BYTE PTR, WORD PTR and DWORD PTR keywords.

There are other differences, some more subtle than others, regarding dereferencing and indexing. For complete details, please refer to the *GNU As Manual*[10].

The GNU Assembler specifies a defailt syntax for the assembly files, at file level. Every file has at least one data segment (designated with .data), and one code segment (designated with .text). The data segment contains global

```
    .data
    .globl  a
        .align  4
        .type   a,@object
5       .size   a,4
a:
        .long   0
```

Listing 12.1: Global Variable Declaration

variables and string constants, while the code segment holds the actual code. The code segment may never be written to, while the data segment is modifiable. Global variables are declared by specifying their size, and optionally a type and alignment. Global variables are always of type `@object` (as opposed to type `@function` for functions). The code in listing 12.1 declares the variable `a`.

It is also required to declare at least one function (the main function) as a global label. This function is used as the program entry point. Its type is always `@function`.

12.3 Globals

The assembly code for an Inger program is generated by traversing the tree multiple times. The first pass is necessary to find all global declarations. As the tree is traversed, the code generation module checks for declaration nodes. When it finds a declaration node, the symbol that belongs to the declaration is retrieved from the symbol table. If this symbol is a global, the type information is retrieved and the assembly code to declare this global variable is generated (see listing 12.1). Local variables and function parameters are skipped during this pass.

12.4 Resource Calculation

During the second pass when the real code is generated, the implementations for functions are also created. Before the code of a function can be generated, the code generation module must know the location of all function parameters and local variables on the stack. This is done by quickly scanning the body of the function for local declarations. Whenever a declaration is found its position on the stack is determined and this is stored in the symbol itself, in the symbol table. This way references to local variables and parameters can easily be converted to stack locations when generating code for the function implementation. The size and location of each symbol play an important role in creating the layout for function stack frames, later on.

12.5 Intermediate Results of Expressions

The code generation module in Inger is implemented in a very simple and straightforward way. There is no real register allocation involved, all intermediate values and results of expressions are stored in the EAX register. Even though this will lead to extremely unoptimized code – both in speed and size – it is also very easy to write. Consider the following simple program:

```
/*
* simple.i
* Simple example program to demonstrate code generation.
*/
module simple;

extern printInt : int i → void;

int a, b;

start main : void → void
{
    a = 16;
    b = 32;
    printInt ( a * b );
}
```

This little program translates to the following x86 assembly code wich shows how the intermediate values and results of expressions are kept in the EAX register:

```
.data
.globl a
    .align 4
    .type   a,@object
    .size   a,4
a:
    .long   0
.globl b
    .align 4
    .type   b,@object
    .size   b,4
b:
    .long   0
.text
    .align 4
.globl main
    .type   main,@function
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $0, %esp
    movl    $16, %eax
    movl    %eax, a
```

```
        movl    $32, %eax
25      movl    %eax, b
        movl    a, %eax
        movl    %eax, %ebx
        movl    b, %eax
        imul    %ebx
30      pushl   %eax
        call    printInt
        addl    $4, %esp
        leave
        ret
```

The eax register may contain either values or references, depending on the code template that placed a value in eax. If the code template was, for instance, addition, eax will contain a numeric value (either floating point or integer). If the code template was the template for the address operator (&), then eax will contain an address (a pointer).

Since the code generation module can assume that the input code is both syntactically and semantically correct, the meaning of the value in eax does not really matter. All the code generation module needs to to is make sure that the value in eax is passed between templates correctly, and if possible, efficiently.

12.6 Function calls

Function calls are executed using the Intel assembly call statement. Since the GNU assembler is reasonable high-level assembler, it is sufficient to supply the *name* of the function being called; the linker (ld) will take care of the job of filling in the correct address, assuming the function exists, but once again – we can assume that the input Inger code is semantically and syntactically correct. If a function really does not exist, and the linker complains about it, it is because there is an error in a header file. The syntax of a basic function call is:

```
call printInt
```

Of course, most interesting functions take parameters. Parameters to functions are always passed to the function using the stack. For this, Inger using the same paradigm that the C language uses: the caller is responsible for both placing parameters on the stack and for removing them after the function call has completed. The reason that Inger is so compatible with C is a practical one: this way Inger can call C functions in operating system libraries, and we do not need to supply wrapper libraries that call these functions. This makes the life of Inger programmers and compiler writers a little bit easier.

Apart from parameters, functions also have local variables and these live on the stack too. All in all the stack is rapidly becoming complex and we call the order in which parameters and local variables are placed on the stack the *stack frame*. As stated earlier, Inger adheres to the *calling convention* popularized by the C programming language, and therefore the stack frames of the two languages are identifical.

The function being called uses the ESP register to point to the top of the stack. The EBP register is the base pointer to the stack frame. As in C, parameters are pushed on the stack from right to left (the last argument is pushed first). Return values of 4 bytes or less are stored in the EAX register. For return values with more than 4 bytes, the caller passes an extra first argument to the *callee* (the function being called). This extra argument is the address of the location where the return value must be stored (this extra argument is the first argument, so it is the last argument to be pushed on the stack). To illustrate this point, we give an example in C:

Example 12.1 (Stack Frame)

```
/* vec3 is a structure of
 * 3 floats (12 bytes). */
struct vec3
{
    int x, y, z;
};

/* f is a function that returns
 * a vec3 struct: */
vec3 f( int a, int b, int c );
```

Since the return value of the function `f` is more than 4 bytes, an extra first argument must be placed on the stack, containing the address of the `vec3` structure that the function returns. This means the call:

```
v = f ( 1, 0, 3 );
```

is transformed into:

```
f( &v , 1, 0, 3 );
```

□

It should be noted that Inger does not support structures at this time, and all data types can be handled using either return values of 4 bytes or less, which fit in `eax`, or using pointers (which are also 4 bytes and therefore fit in `eax`). For future extensions of Inger, we have decided to support the extra return value function argument.

Since functions have a stack frame of their own, the contents of the stack frame occupied by the caller are quite safe. However, the registers used by the caller will be overwritten by the callee, so the caller must take care to push any values it needs later onto the stack. If the caller wants to save the `eax`, `ecx` and `edx` registers, it has to push them on the stack first. After that, it pushes the arguments (from right to left), and when the call instruction is called, the `eip` register is pushed onto the stack too (implicitly, by the `call` instruction), which means the return address is on top of the stack.

Although the caller does most of the work creating the stack frame (pushing parameters on the stack), the callee still has to do several things. The stack frame is not yet finished, because the callee must create space for local variables (and set them to their initial values, if any). Furthermore, the callee must set save the contents of ebx, esi and edi as needed and set esp and ebp to point to the top and bottom of the stack, respectively. Initially, the EBP register points to a location in the caller's stack frame. This value must be preserved, so it must be pushed onto the stack. The contents of esp (the bottom of the current stack frame) are then copied into esp, so that esp is free to do other things and to allow arguments to be referenced as an offset from ebp. This gives us the stack frame depicted in figure 12.1.

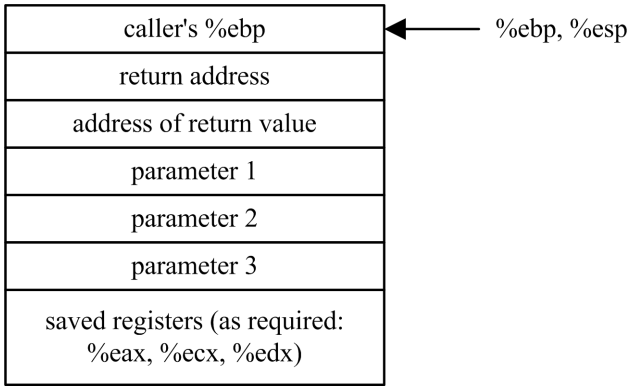


Figure 12.1: Stack Frame Without Local Variables

To allocate space for local variables and temporary storage, the callee just subtracts the number of bytes required for the allocation from esp. Finally, it pushes ebx, esi and edi on the stack, if the function overwrites them. Of course, this depends on the templates used in the function, so for every template, its effects on ebx, esi and edi must be known.

The stack frame now has the form shown in figure 12.2.

During the execution of the function, the stack pointer esp might go up and down, but the ebp register is fixed, so the function can always refer to the first argument as [ebp+8]. The second argument is located at [ebp+12] (decimal offset), the third argument is at [ebp+16] and so on, assuming all argument are 4 bytes in size.

The callee is not done yet, because when execution of the function body is complete, it must perform some cleanup operations. Of course, the caller is responsible for cleaning up function parameters it pushed onto the stack (just like in C), but the remainer of the cleanup is the callee's job. The callee must:

- Store the return value in eax, or in the extra parameter;
- Restore the ebx, esi and edi registers as needed.

Restoration of the values of the ebx, esi and edi registers is performed by popping them from the stack, where they had been stored for safekeeping earlier.

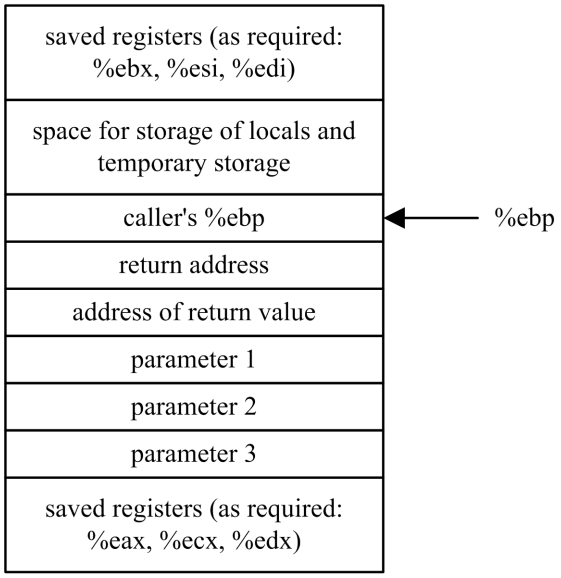


Figure 12.2: Stack Frame With Local Variables

Of course, it is important to only pop the registers that were pushed onto the stack in the first place: some functions save ebx, esi and edi, while others do not.

The last thing to do is taking down the stack frame. This is done by moving the contents from ebp to esp (thus effectively discarding the stack frame) and popping the original ebp from the stack.[1] The return (ret) instruction can now be executed, wich pops the return address of the stack and places it in the eip register.

Since the stack is now exactly the same as it was before making the function call, the arguments (and return value when larger than 4 bytes) are still on the stack. The esp can be restored by adding the number of bytes the arguments use to esp.

Finally, if there were any saved registers (eax, ecx and edx) they must be popped from the stack as well.

12.7 Control Flow Structures

The code generation module handles if/then/else structures by generating comparison code and conditional jumps. The jumps go to the labels that are generated before the then and else blocks.

Loops are also implemented in a very straight forward manner. First it generates a label to jump back to every iteration. After that the comparison code is generated. This is done in exactly the same way as it is done with if expressions. After this the code block of the loop is generated followed by a jump to the label right before the comparison code. The loop is concluded

[1] The i386 instruction set has an instruction leave which does this exact thing.

with a final label where the comparison code can jump to if the result of the expression is false.

12.8 Conclusion

This concludes the description of the inner workings of the code generation module for the Inger language.

Bibliography

[1] O. Andrew and S. Talbott: *Managing projects with Make*, OReilly & associates, inc., December 1991.

[2] B. Brey: *8086/8088, 80286, 80386, and 80486 Assembly Language Programming*, Macmillan Publishing Company, 1994.

[3] G. Chapell: *DOS Internals*, Addison-Wesley, 1994.

[4] J. Duntemann: *Assembly Language Step-by-Step*, John Wiley & Sons, Inc., 1992.

[5] T. Hogan: *The Programmers PC Sourcebook: Charts and Tables for the IBM PC Compatibles, and the MS-DOS Operating System, including the new IBM Personal System/2 computers*, Microsoft Press, Redmond, Washington, 1988.

[6] K. Irvine: *Assembly Language for Intel-based Computers*, Prentice-Hall, Upper Saddle River, NJ, 1999.

[7] M. L. Scott: *Porgramming Language Pragmatics*, Morgan Kaufmann Publishers, 2000.

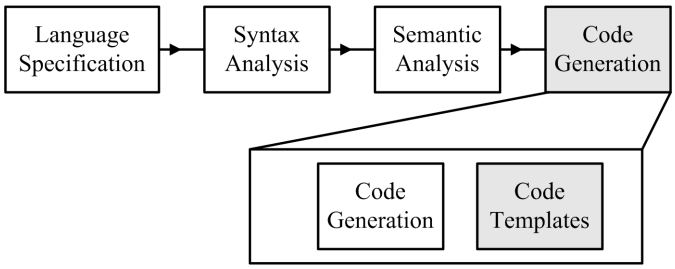
[8] I. Sommerville: *Software Engineering (sixth edition)*, Addison-Wesley, 2001.

[9] W. Stallings: *Operating Systems: achtergronden, werking en ontwerp*, Academic Service, Schoonhoven, 1999.

[10] R. Stallman: *GNU As Manual*,
http://www.cs.utah.edu/dept/old/texinfo/as/as.html

Chapter 13

Code Templates



This final chapter of the book serves as a repository of code templates. These templates are used by the compiler to generate code for common (sub) expressions. Every template has a name and will be treated on the page ahead, each template on a page of its own.

Addition

Inger

```
expr + expr
```

Example

```
3 + 5
```

Assembler

1. The left expression is evaluated and stored in `eax`.
2. **`movl %eax, %ebx`**
3. The right expression is evaluated and stored in `eax`.
4. **`addl %ebx, %eax`**

Description

The result of the left expression is added to the result of the right expression and the result of the addition is stored in `eax`.

Subtraction

Inger

```
expr − expr
```

Example

```
8 − 3
```

Assembler

1. Left side of expression is evaluated and stored in eax.
2. **movl %eax, %ebx**
3. Right side of expression is evaluated and stored in eax.
4. **subl %ebx, %eax**

Description

The result of the right expression is subtracted from the result of the left expression and the result of the subtraction is stored in eax.

Multiplication

Inger

expr ∗ expr

Example

12 − 4

Assembler

1. Left side of expression is evaluated and stored in eax.
2. **movl %eax, %ebx**
3. Right side of expression is evaluated and stored in eax.
4. **imul %ebx**

Description

The result of the left expression is multiplied with the result of the right expression and the result of the multiplication is stored in eax.

Division

Inger

```
expr / expr
```

Example

```
32 / 8
```

Assembler

1. Left side of expression is evaluated and stored in eax.
2. **movl %eax, %ebx**
3. Right side of expression is evaluated and stored in eax.
4. **xchgl %eax, %ebx**
5. **xorl %edx, %edx**
6. **idiv %ebx**

Description

The result of the left expression is divided by the result of the right expression and the result of the division is stored in eax.

Modulus

Inger

```
expr % expr
```

Example

```
14 % 3
```

Assembler

1. Left side of expression is evaluated and stored in eax.
2. **movl %eax, %ebx**
3. Right side of expression is evaluated and stored in eax.
4. **xchgl %eax, %ebx**
5. **xorl %edx, %edx**
6. **idiv %ebx**
7. **movl %edx, %eax**

Description

The result of the left expression is divided by the result of the right expression and the remainder of the division is stored in eax.

Negation

Inger

```
−expr
```

Example

```
−10
```

Assembler

1. Expression is evaluated and stored in `eax`.
2. **`neg %eax`**

Description

The result of the expression is negated and stored in the `eax` register.

Left Bitshift

Inger

```
expr << expr
```

Example

```
256 << 2
```

Assembler

1. Left side of expression is evaluated and stored in `eax`.
2. **movl %eax, %ecx**
3. Right side of expression is evaluated and stored in `eax`.
4. **xchgl %eax, %ecx**
5. **sall %cl, %eax**

Description

The result of the left expression is shifted n bits to the left, where n is the result of the right expression. The result is stored in the `eax` register.

Right Bitshift

Inger

```
expr >> expr
```

Example

```
16 >> 2
```

Assembler

1. Left side of expression is evaluated and stored in eax.
2. **movl %eax, %ecx**
3. Right side of expression is evaluated and stored in eax.
4. **xchgl %eax, %ecx**
5. **sarl %cl, %eax**

Description

The result of the left expression is shifted n bits to the right, where n is the result of the right expression. The result is stored in the eax register.

Bitwise And

Inger

```
expr & expr
```

Example

```
255 & 15
```

Assembler

1. Left side of expression is evaluated and stored in `eax`.
2. **`movl %eax, %ebx`**
3. Right side of expression is evaluated and stored in `eax`.
4. **`andl %ebx, %eax`**

Description

The result of an expression is subject to a *bitwise and* operation with the result of another expression and this is stored in the `eax` register.

Bitwise Or

Inger

```
expr | expr
```

Example

```
13 | 22
```

Assembler

1. Left side of expression is evaluated and stored in eax.
2. **movl %eax, %ebx**
3. Right side of expression is evaluated and stored in eax.
4. **orl %ebx, %eax**

Description

The result of an expression is subject to a *bitwise or* operation with the result of another expression and this is stored in the eax register.

Bitwise Xor

Inger

```
expr ^ expr
```

Example

```
63 ^ 31
```

Assembler

1. Left side of expression is evaluated and stored in `eax`.
2. **`movl %eax, %ebx`**
3. Right side of expression is evaluated and stored in `eax`.
4. **`andl %ebx, %eax`**

Description

The result of an expression is subject to a *bitwise xor* operation with the result of another expression and this is stored in the `eax` register.

If-Then-Else

Inger

```
if ( expr )
{
    // Code block
}
// The following part is optional
else
{
   // Code block
}
```

Example

```
int a = 2;
if ( a == 1 )
{
    a = 5;
}
else
{
    a = a - 1;
}
```

Assembler

When there is only a *then* block:

1. Expression is evaluated and stored in eax.
2. **cmpl** $0, %**eax**
3. **je** .LABEL0
4. *Then* code block is generated.
5. .LABEL0:

When there is an *else* block:

1. Expression is evaluated and stored in eax.
2. **cmpl** $0, %**eax**
3. **je** .LABEL0
4. *Then* code block is generated.
5. **jmp** .LABEL1
6. .LABEL0:
7. *Else* code block is generated.
8. .LABEL1:

Description

This template describes an *if-then-else* construction. The conditional code execution is realized with conditional jumps to labels. Different templates are used for *if-then* and *if-then-else* constructions.

While Loop

Inger

```
while( expr ) do
{
    // Code block
}
```

Example

```
int i = 5;
while( i > 0 ) do
{
    i = i - 1;
}
```

Assembler

1. Expression is evaluated and stored in eax.
2. .LABEL0:
3. **cmpl** $0, %**eax**
4. **je** .LABEL1
5. Code block is generated
6. **jmp** .LABEL0

Description

This template describes a *while* loop. The expression is evaluated and while the result of the expression is true the code block is executed.

Function Application

Inger

```
func( arg1, arg2, argN );
```

Example

```
printInt ( 4 );
```

Assembler

1. The expression of each argument is evaluated, stored in eax, and pushed on the stack.
2. **movl %ebp, %ecx**
3. The location on the stack is determined.
4. **call** printInt (in this example the function name is printInt)
5. The number of bytes used for the arguments is calculated.
6. **addl** $4, **%esp** (in this example the number of bytes is 4)

Description

This template describes the application of a function. The arguments are pushed on the stack according to the C style function call convention.

Function Implementation

Inger

```
func: type ident1, type ident2, type identN → returntype
{
    // Implementation
}
```

Example

```
square: int i → int
{
    return( i * i );
}
```

Assembler

1. **.globl** square (in this example the function name is square)
2. **.type** square, @function
3. square:
4. **pushl** %ebp
5. **movl** %esp, %ebp
6. The number of bytes needed for the parameters are counted here.
7. **subl** $4, %esp (in this example the number of bytes needed is 4)
8. The implementation code is generated here.
9. **leave**
10. **ret**

Description

This template describes implementation of a function. The number of bytes needed for the parameters is calculated and subtracted from the esp register to allocate space on the stack.

Identifier

Inger

identifier

Example

i

Assembler

For a global variable:

1. Expression is evaluated and stored in eax
2. **movl** i, **%eax** (in this example the name of the identifier is i)

For a local variable:

1. **movl %ebp**, **%ecx**
2. The location on the stack is determined
3. **addl** $4, **%ecx** (in this example the stack offset is 4)
4. **movl** (**%ecx**), **%eax**

Description

This template describes the use of a variable. When a global variable is used it is easy to generate the assembly because we can just use the name of the identifier. For locals its position on the stack has to be determined.

Assignment

Inger

```
identifier = expr;
```

Example

```
i = 12;
```

Assembler

For a global variable:

1. The expression is evaluated and stored in `eax`.
2. **`movl %eax`**, `i` (in this example the name of the identifier is i)

For a local variable:

1. The expression is evaluated and stored in `eax`.
2. The location on the stack is determined
3. **`movl %eax`**, `4`**`(%ebp)`** (in this example the offset on the stack is 4)

Description

This template describes an assignment of a variable. Global and local variables must be handled differently.

Global Variable Declaration

Inger

```
type identifier = initializer ;
```

Example

```
int i = 5;
```

Assembler

For a global variable:

1. **.data**
2. **.globl** i (in this example the name of the identifier is i)
3. **.type** i,@object)
4. **.size** i,4 (in this example the type is 4 bytes in size)
5. a:
6. **.long** 5 (in this example the initializer is 5)

Description

This template describes the declaration of a global variable. When no initializer is specified, the variable is initialized to zero.

Equal

Inger

expr == expr

Example

i == 3

Assembler

1. The left expression is evaluated and stored in eax.
2. **movl %eax, %ebx**
3. The right expression is evaluated and stored in eax.
4. **cmpl %eax, %ebx**
5. **movl** $0, **%ebx**
6. **movl** $1, **%ecx**
7. **cmovne %ebx, %eax**
8. **cmove %ecx, %eax**

Description

This template describes the == operator. The two expressions are evaluated and the results are compared. When the results are the same, 1 is loaded in eax. When the results are not the same, 0 is loaded in eax.

Not Equal

Inger

```
expr != expr
```

Example

```
i != 5
```

Assembler

1. The left expression is evaluated and stored in eax.
2. **movl %eax, %ebx**
3. The right expression is evaluated and stored in eax.
4. **cmpl %eax, %ebx**
5. **movl** $0, **%ebx**
6. **movl** $1, **%ecx**
7. **cmove %ebx, %eax**
8. **cmovne %ecx, %eax**

Description

This template describes the $\neq$ operator. The two expressions are evaluated and the results are compared. When the results are not the same, 1 is loaded in eax. When the results are the same, 0 is loaded in eax.

Less

Inger

expr < expr

Example

i < 18

Assembler

1. The left expression is evaluated and stored in eax.
2. **movl %eax, %ebx**
3. The right expression is evaluated and stored in eax.
4. **cmpl %eax, %ebx**
5. **movl** $0, **%ebx**
6. **movl** $1, **%ecx**
7. **cmovnl %ebx, %eax**
8. **cmovl %ecx, %eax**

Description

This template describes the < operator. The two expressions are evaluated and the results are compared. When the left result is less than the right result, 1 is loaded in eax. When the left result is not smaller than the right result, 0 is loaded in eax.

Less Or Equal

Inger

expr <= expr

Example

i <= 44

Assembler

1. The left expression is evaluated and stored in eax.
2. **movl %eax, %ebx**
3. The right expression is evaluated and stored in eax.
4. **cmpl %eax, %ebx**
5. **movl** $0, **%ebx**
6. **movl** $1, **%ecx**
7. **cmovnle %ebx, %eax**
8. **cmovle %ecx, %eax**

Description

This template describes the $\leq$ operator. The two expressions are evaluated and the results are compared. When the left result is less than or equals the right result, 1 is loaded in eax. When the left result is not smaller than and does not equal the right result, 0 is loaded in eax.

Greater

Inger

expr > expr

Example

i > 57

Assembler

1. The left expression is evaluated and stored in eax.
2. **movl %eax, %ebx**
3. The right expression is evaluated and stored in eax.
4. **cmpl %eax, %ebx**
5. **movl** $0, **%ebx**
6. **movl** $1, **%ecx**
7. **cmovng %ebx, %eax**
8. **cmovg %ecx, %eax**

Description

This template describes the > operator. The two expressions are evaluated and the results are compared. When the left result is greater than the right result, 1 is loaded in eax. When the left result is not greater than the right result, 0 is loaded in eax.

Greater Or Equal

Inger

```
expr >= expr
```

Example

```
i >= 26
```

Assembler

1. The left expression is evaluated and stored in eax.
2. **movl %eax, %ebx**
3. The right expression is evaluated and stored in eax.
4. **cmpl %eax, %ebx**
5. **movl** $0, **%ebx**
6. **movl** $1, **%ecx**
7. **cmovnge %ebx, %eax**
8. **cmovge %ecx, %eax**

Description

This template describes the $\geq$ operator. The two expressions are evaluated and the results are compared. When the left result is greater than or equals the right result, 1 is loaded in eax. When the left result is not greater than and does not equal the right result, 0 is loaded in eax.

Chapter 14

Bootstrapping

A subject which has not been discussed so far is *bootstrapping*. Bootstrapping means building a compiler in its own language. So for the Inger language it would mean that we build the compiler in the Inger language as well. To discuss the practical application of this theory is beyond the scope of this book. Below we explain it in theory.

Developing a new language is mostly done to improve some aspects compared to other, existing languages. What we would prefer, is to compile the compiler in its own language, but how can that be done when there is no compiler to compile the compiler? To visualize this problem we use so-called *T-diagrams*, to illustrate the process of bootstrapping. To get familiar with T-diagrams we present a few examples.

Example 14.1 (T-Diagrams)

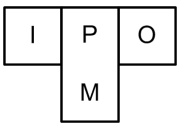


Figure 14.1: Program P can work on machine with language M. I = input, O = output.

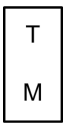


Figure 14.2: Interpreter for language T, is able to work on a machine with language M.

□

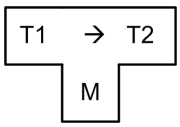


Figure 14.3: Compiler for language T1 to language T2, is able to work on a machine with language M.

Figure 14.4: Machine for language M.

The bootstrapping problem can be resolved in the following way:

1. Build two versions of the complier. One version is the *optimal-compiler* and the other version is the *bootstrap-compiler*. The optimal compiler written for the new language T, complete with all optimisation is written in language T itself. The bootstrap-compiler is written in an existing language m. Because this compiler is not optimized and therefore slower in use, the m is written in lowercase instead of uppercase.

2. Translate the optimal-compiler with the bootstrap-compiler. The result is the optimal-compiler, which can run on the target machine M. However, this version of the compiler is not optimized yet (slow, and a lot of memory usage). We call this the temporary-compiler.

3. Now we must the compile the optimal-compiler again, only this time we use the temporary-compiler to compile it with. The result will be the final, optimal compiler able to run on machine M. This compiler will be fast and produce optimized output.

4. The result:

   It is a long way before you have a bootstrap compiler, but remember, this is the ultimate compiler!

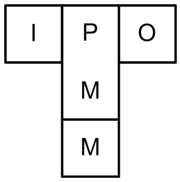


Figure 14.5: Program P runs on machine M.

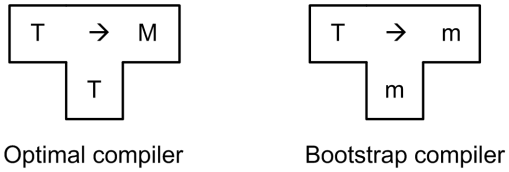


Figure 14.6: The two compilers.

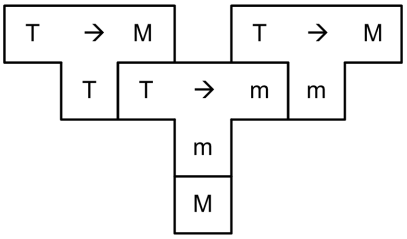


Figure 14.7: temporary-compiler.

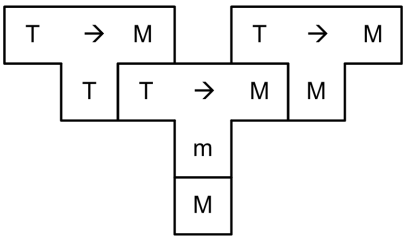


Figure 14.8: compile process.

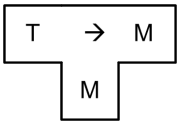


Figure 14.9: final-compiler.

Chapter 15

Conclusion

All parts how to build a compiler, from the setup of a language to the code generation, have now been discussed. Using this book as a reference, it should be possible to build your own compiler.

The compiler we build in this book is not innovating. Lots of this type of compiler (for imperative languages) compilers already exist, only the language differs: examples include compilers for C or Pascal.

Because the Inger compiler is a low-level compiler, it is extremely suitable for system programming (building operating systems). The same applies to game programming and programming command line applications (such as UNIX filter tools).

We hope you will be able to put the theory and practical examples we described in this book to use, in order to build your own compliler. It is up to you now!

Appendix A

Requirements

A.1 Introduction

This chapter specifies the software necessary to either use (run) Inger or to develop for Inger. The version numbers supplied in this text are the version numbers of software packages we used. You may well be able to use older versions of this software, but it is not guaranteed that this will work. You can always (except in rare cases) use newer versions of the software discussed below.

A.2 Running Inger

Inger was designed for the Linux operating system. It is perfectly possible to run Inger on other platforms like Windows, and even do development work for Inger on non-Linux platforms. However, this section discusses the software required to run Inger on Linux.

The Linux distribution we used is RedHat,[1] but other advanced Linux distributions like SuSE[2] or Mandrake[3] will do fine.

The most elementary package you need to run Inger is naturally Inger itself. It can be downloaded from its repository at Source Forge.[4] There are two packages available: the user package, which contains only the compiler binary and the user manual, and the development package, which contains the compiler source as well. Be sure to download the user version, not the developer version.

As the Inger compiler compiles to GNU AT&T assembly code, the GNU assembler as is required to convert the assembly code to object code. The GNU assembler is part of the binutils package.[5] You may use any other assembler, provided it supports the AT&T assembly syntax; as is the only assembler that supports it that we are currently aware of. A linker is also required — we use the GNU linker (which is also part of the binutils package).

Some of the code for Inger is generated using scripts written in the Perl scripting language. You will need the perl[6] interpreter to execute these scripts.

[1] RedHat Linux 7.2, http://www.redhat.com/apps/download/
[2] SuSE Linux 8.0, http://www.suse.com/us/private/download/index.html
[3] Mandrake Linux 8.0, http://www.mandrake.com
[4] Inger 0.x, http://www.sourceforge.net/projects/inger
[5] Binutils 2.11.90.0.8, http://www.gnu.org/directory/GNU/binutils.html
[6] Perl 6, http://www.perl.com

If you use a Windows port of Inger, you can also use the GNU ports of as and ld that come with DJGPP.[7] DJGPP is a free port of (most of) the GNU tools.

It can be advantageous to be able to view this documentation in digital form (as a Portable Document File), which is possible with the Acrobat Reader.[8] The Inger website may also offer this documentation in other forms, such as HTML.

Editing Inger source code in Linux can be done with the free editor vi, which is included with virtually all Linux distributions. You can use any editor you want, though. An Inger syntax highlighting template for Ultra Edit is available from the Inger archive at Source Forge.

If a Windows binary of the Inger compiler is not available or not usable, and you need to run Inger on a Windows platform, you may be able to use the Linux emulator for the Windows platform, Cygwin,[9] to execute the Linux binary.

A.3 Inger Development

For development of the Inger language, rather than development *with* the Inger language, some additional software packages are required. For development purposes we strongly recommend that you work on a Linux platform, since we cannot guarantee that all development tools are available on Windows platforms or work the same way as they do on Linux.

The Inger binary is built from source using automake[10] and autoconf[11], both of which are free GNU software. These packages allow the developer to generate makefiles that target the user platform, i.e. use available C compiler and lexical scanner generator versions, and warn if no suitable software is available. To execute the generated makefiles, GNU make, which is part of the binutils package, is also required. Most Linux installations should have this software already installed.

C sources are compiled using the GNU Compiler Collection (gcc). [12] We used the lexical analyzer generator GNU flex [13] to generate a lexical scanner.

All Inger code is stored in a Concurrent Versioning System repository on a server at Source Forge, which may be accessed using the cvs package. [14] Note that you must be registered as an Inger developer to be able to change the contents of the CVS repository. Registration can be done through Source Forge.

All documentation was written using the LaTeX 2_ε typesetting package, [15] which is also available for Windows as the MikTeX[16] system. Editors that come in handy when working with TeX sources are Ultra Edit, [17] which supports TeX syntax highlighting, and TexnicCenter [18] which is a full-fledged TeX editor with

[7] DJGPP 2.03, http://www.delorie.com or http://www.simtel.net/pub/djgpp
[8] Adobe Acrobat 5.0, http://www.adobe.com/products/acrobat/readstep.html
[9] Cygwin 1.11.1p1, http://www.cygwin.com
[10] Automake 1.4-p5, http://www.gnu.org/software/automake
[11] Autoconf 2.13, http://www.gnu.org/software/autoconf/autoconf.html
[12] GCC 2.96, http://www.gnu.org/software/gcc/gcc.html
[13] Flex 2.5.4, http://www.gnu.org/software/flex
[14] CVS 1.11.1p1, http://www.cvshome.org
[15] LaTeX 2_ε, http://www.latex-project.org
[16] MikTeX 2.2, http://www.miktex.org
[17] Ultra Edit 9.20, http://www.ultraedit.com
[18] TexnicCenter, http://www.toolscenter.org/products/texniccenter

many options (although no direct visual feedback — it is a *what you see is what you mean* (WYSIWYM) tool).

The Inger development package comes with a project definition file for KDevelop, an open source clone of Microsoft Visual Studio. If you have a Linux distribution that has the X window system with KDE (K Desktop Environment) installed, then you can do development work for Inger in a graphical environment.

A.4 Required Development Skills

Development on Inger requires the following skills:

- A working knowledge of the C programming language;
- A basic knowlege of the Perl scripting language;
- Experience with GNU assembler (specifically, the AT&T assembly syntax).

The rest of the skills needed, including working with the lexical analyzer generator flex and writing tree data structures can be acquired from this book. Use the bibliography at the end of this chapter to find additional literature that will help you master all the tools discussed in the preceding sections.

Bibliography

[1] M. Bar: *Open Source Development with CVS*, Coriolis Group, 2^{nd} edition, 2001

[2] D. Elsner: *Using As: The Gnu Assembler*, iUniverse.com, 2001

[3] M. Goossens: *The Latex Companion*, Addison-Wesley Publishing, 1993

[4] A. Griffith: *GCC: the Complete Reference*, McGraw-Hill Osborne Media, 1^{st} edition, 2002

[5] E. Harlow: *Developing Linux Applications*, New Riders Publishing, 1999.

[6] J. Levine: *Lex & Yacc*, O'Reilly & Associates, 1992

[7] M. Kosta Loukides: *Programming with GNU Software*, O'Reilly & Associates, 1996

[8] C. Negus: *Red Hat Linux 8 Bible*, John Wiley & Sons, 2002

[9] Oetiker, T.: *The Not So Short Introduction to LaTeX 2_ε*, version 3.16, 2000

[10] A. Oram: *Managing Projects with Make*, O'Reilly & Associates, 2^{nd} edition, 1991

[11] G. Purdy: *CVS Pocket Reference*, O'Reilly & Associates, 2000

[12] R. Stallman: *Debugging with GDB: The GNU Source-Level Debugger*, Free Software Foundation, 2002

[13] G.V. Vaughan: *GNU Autoconf, Automake, and Libtool*, New Riders Publishing, 1^{st} edition, 2000

[14] L. Wall: *Programming Perl*, O'Reilly & Associates, 3^rd edition, 2000

[15] M. Welsch: *Running Linux*, O'Reilly & Associates, 3^rd edition, 1999

Appendix B

Software Packages

This appendix lists the locations software packages that are required or recommended in order to use Inger or do development work for Inger. Note that the locations (URLs) of these packages are subject to change and may not be correct.

Package	Description and location
RedHat Linux 7.2	Operating system http://www.redhat.com
SuSE Linux 8.0	Operating system http://www.suse.com
Mandrake 8.0	Operating system http://www.mandrake.com
GNU Assembler 2.11.90.0.8	AT&T syntax assembler http://www.gnu.org/directory/GNU/binutils.html
GNU Linker 2.11.90.0.8	COFF file linker http://www.gnu.org/directory/GNU/binutils.html
DJGPP 2.03	GNU tools port http://www.delorie.com
Cygwin 1.2	GNU Linux emulator for Windows http://www.cygwin.com

Package	Description and location
CVS 1.11.1p1	Concurrent Versioning System http://www.cvshome.org
Automake 1.4-p5	Makefile generator http://www.gnu.org/software/automake
Autoconf 2.13	Makefile generator support http://www.gnu.org/software/autoconf/autoconf.html
Make 2.11.90.0.8	Makefile processor http://www.gnu.org/software/make/make.html
Flex 2.5.4	Lexical analyzer generator http://www.gnu.org/software/flex
LaTeX 2_ε	Typesetting system http://www.latex-project.org
MikTeX 2.2	Typesetting system http://www.miktex.org
TexnicCenter	TeX editor http://www.toolscenter.org/products/texniccenter
Ultra Edit 9.20	TeX editor http://www.ultraedit.com
Perl 6	Scripting language http://www.perl.com

Appendix C

Summary of Operations

C.1 Operator Precedence Table

Operator	Priority	Associatity	Description
`()`	1	L	function application
`[]`	1	L	array indexing
`!`	2	R	logical negation
`-`	2	R	unary minus
`+`	2	R	unary plus
`~`	3	R	bitwise complement
`*`	3	R	indirection
`&`	3	R	referencing
`*`	4	L	multiplication
`/`	4	L	division
`%`	4	L	modulus
`+`	5	L	addition
`-`	5	L	subtraction
`>>`	6	L	bitwise shift right
`<<`	6	L	bitwise shift left
`<`	7	L	less than
`<=`	7	L	less than or equal
`>`	7	L	greater than
`>=`	7	L	greater than or equal
`==`	8	L	equality
`!=`	8	L	inequality
`&`	9	L	bitwise and
`^`	10	L	bitwise xor
`|`	11	L	bitwise or
`&&`	12	L	logical and
`||`	12	L	logical or
`?:`	13	R	ternary if
`=`	14	R	assignment

C.2 Operand and Result Types

Operator	Operation	Operands	Result
`()`	function application	any	any
`[]`	array indexing	int	none
`!`	logical negation	bool	bool
`-`	unary minus	int	int
`+`	unary plus	int	int
`~`	bitwise complement	int, char	int, char
`*`	indirection	any	any pointer
`&`	referencing	any pointer	any
`*`	multiplication	int, float	int, float
`/`	division	int, float	int, float
`%`	modulus	int, char	int, char
`+`	addition	int, float, char	int, float, char
`-`	subtraction	int, float, char	int, float, char
`>>`	bitwise shift right	int, char	int, char
`<<`	bitwise shift left	int, char	int, char
`<`	less than	int, float, char	int, float, char
`<=`	less than or equal	int, float, char	int, float, char
`>`	greater than	int, float, char	int, float, char
`>=`	greater than or equal	int, float, char	int, float, char
`==`	equality	int, float, char	int, float, char
`!=`	inequality	int, float, char	int, float, char
`&`	bitwise and	int, char	int, char
`^`	bitwise xor	int, char	int, char
`|`	bitwise or	int, char	int, char
`&&`	logical and	bool	bool
`||`	logical or	bool	bool
`?:`	ternary if	bool	(2x) any
`=`	assignment	any	any

Appendix D

Backus-Naur Form

module:	**module** <identifier> **;** globals.
globals :	ϵ .
globals :	global globals .
globals :	**extern** global globals.
global :	function .
global :	declaration .
function :	functionheader functionrest .
functionheader:	modifiers < identifier > **:** paramlist $->$ returntype.
functionrest :	**;** .
functionrest :	block .
modifiers :	ϵ .
modifiers :	**start**.
paramlist :	**void**.
paramlist :	paramblock moreparamblocks.
moreparamblocks:	ϵ .
moreparamblocks:	**;** paramblock moreparamblocks.
paramblock:	type param moreparams.
moreparams:	ϵ .
moreparams:	**,** param moreparams.
param:	reference < identifier > dimensionblock.

returntype: type reference dimensionblock.

reference : ϵ .
reference : * reference .

dimensionblock: ϵ .
dimensionblock: [ ] dimensionblock.

block: { code } .

code: ϵ .
code: block code
code: statement codeϵ.

statement: **label** < identifier > **;**
statement: **;**
statement: **break ;**
statement: **continue ;**
statement: expression **;**
statement: declarationblock **;**
statement: **if** (expression) block elseblock
statement: **goto** <identifier> **;**
statement: **while** (expression) **do** block
statement: **do** block **while** (expression) **;**
statement: **switch** (expression) {
switchcases **default** block }
statement: **return** (expression) **;**.

elseblock : ϵ .
elseblock : **else** block.

switchcases : ϵ .
switchcases : **case** < intliteral > block swithcases.

declarationblock : type declaration restdeclarations .

restlocals : ϵ .
restlocals : **,** declaration restdeclarations .

local : reference < identifier > indexblock
initializer .

indexblock: ϵ .
indexblock: [< intliteral >] indexblock.

initializer : ϵ .
initializer : = expression.

expression : logicalor restexpression .

restexpression : ϵ .
restexpression : = logicalor restexpression .

logicalor : logicaland restlogicalor .

restlogicalor : ϵ .
restlogicalor : || logicaland restlogicalor .

logicaland : bitwiseor restlogicaland .

restlogicaland : ϵ .
restlogicaland : && bitwiseor restlogicaland.

bitwiseor : bitwisexor restbitwiseor .

restbitwiseor : ϵ .
restbitwiseor : | bitwisexor restbitwiseor .

bitwisexor : bitwiseand restbitwisexor .

restbitwisexor : ϵ .
restbitwisexor : ^ bitwiseand restbitwisexor .

bitwiseand: equality restbitwiseand.

restbitwiseand : ϵ .
restbitwiseand : & equality restbitwiseand.

equality : relation restequality .

restequality : ϵ .
restequality : equalityoperator relation
restequality .

equalityoperator : ==.
equalityoperator : !=.

relation : shift restrelation .

restrelation : ϵ .
restrelation : relationoperator shift restrelation .

relationoperator : <.
relationoperator : <=.
relationoperator : >.
relationoperator : >=.

shift : addition restshift .

restshift : ϵ .
restshift : shiftoperator addition restshift .

shiftoperator : $<<$.
shiftoperator : $>>$.

addition : multiplication restaddition .

restaddition : ϵ .
restaddition : additionoperator multiplication restaddition .

additionoperator : $+$.
additionoperator : $-$.

multiplication : unary3 restmultiplication .

restmultiplication : ϵ .
restmultiplication : multiplicationoperator unary3 restmultiplication .

multiplicationoperator : $*$.
multiplicationoperator : $/$.
multiplicationoperator : $\%$.

unary3: unary2
unary3: unary3operator unary3.

unary3operator: &.
unary3operator: $*$.
unary3operator: ~.

unary2: factor .
unary2: unary2operator unary2.

unary2operator: $+$.
unary2operator: $-$.
unary2operator: !.

factor : $<$ identifier $>$ application.
factor : immediateϵ.
factor : (expression) .

application : ϵ .
application : [expression] application .
application : (expression moreexpressions) .

moreexpressions: ϵ .
moreexpressions: , expression morexpressions.

type: **bool**.
type: **char**.
type: **float**.
type: **int**.
type: **untyped**.

immediate: <booleanliteral>.
immediate: <charliteral>.
immediate: < floatliteral >.
immediate: < intliteral >.
immediate: < stringliteral >.

Appendix E

Syntax Diagrams

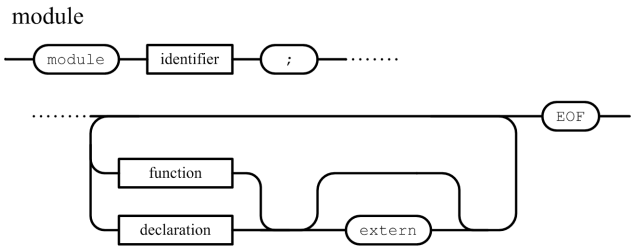


Figure E.1: Module

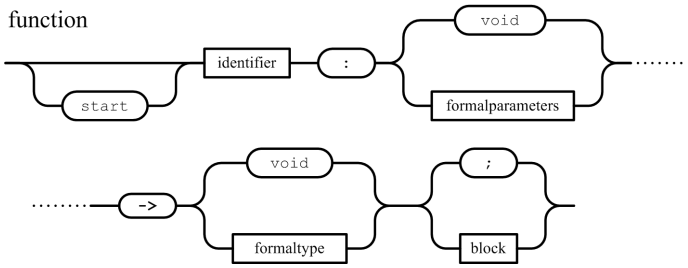


Figure E.2: Function

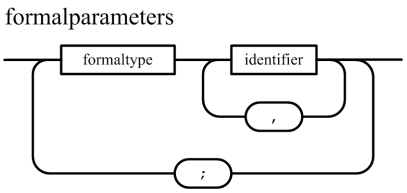


Figure E.3: Formal function parameters

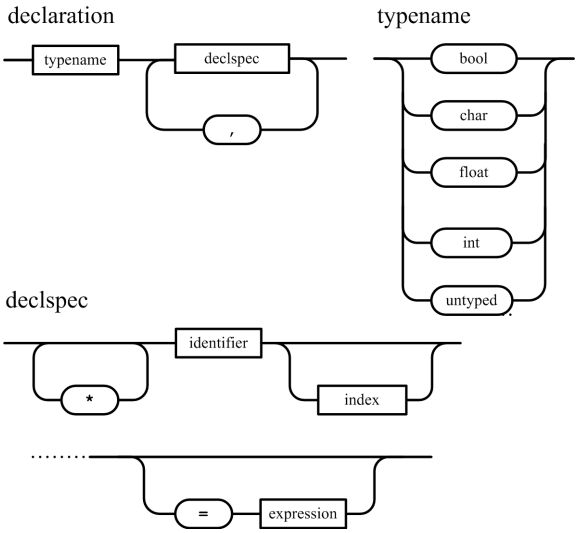


Figure E.4: Data declaration

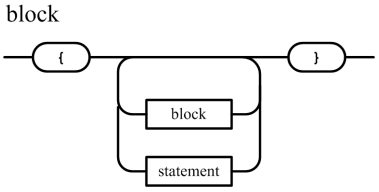


Figure E.5: Code block

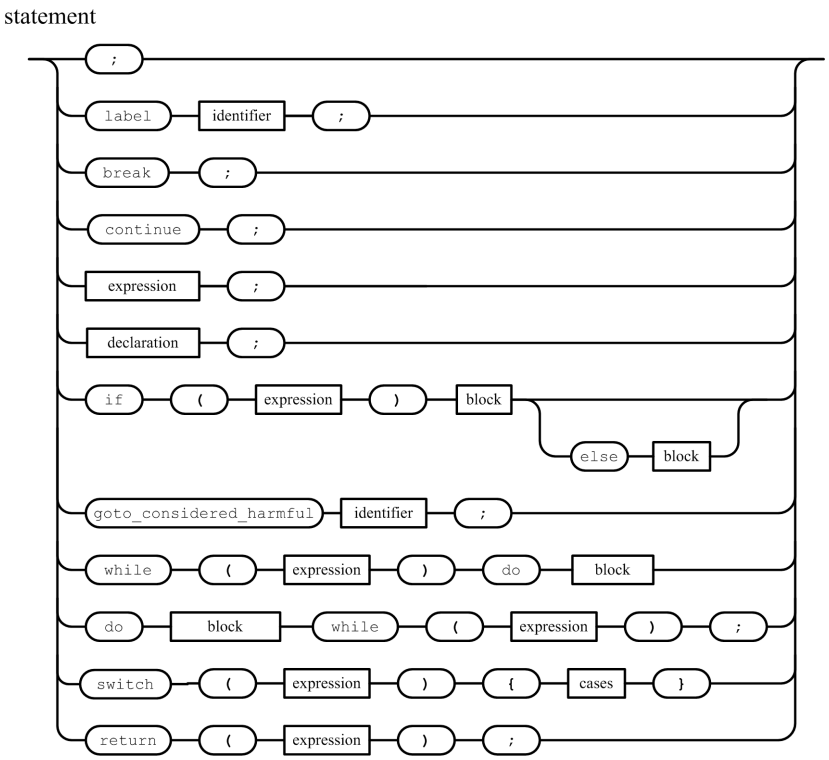


Figure E.6: Statement

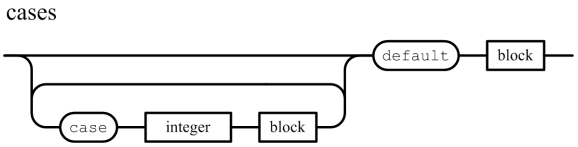


Figure E.7: Switch cases

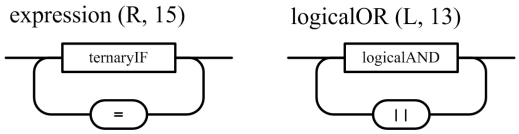


Figure E.8: Assignment, Logical OR Operators

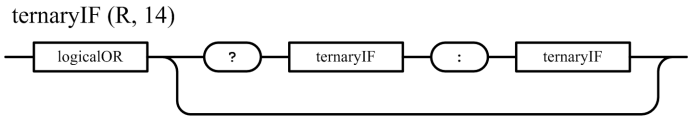


Figure E.9: Ternary IF

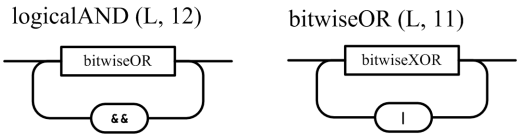


Figure E.10: Logical AND and Bitwise OR Operators

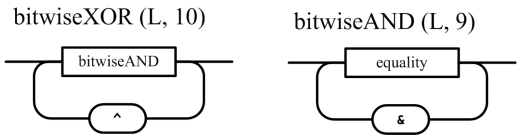


Figure E.11: Bitwise XOR and Bitwise AND Operators

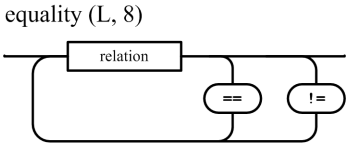


Figure E.12: Equality Operators

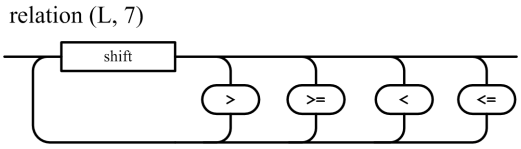


Figure E.13: Relational Operators

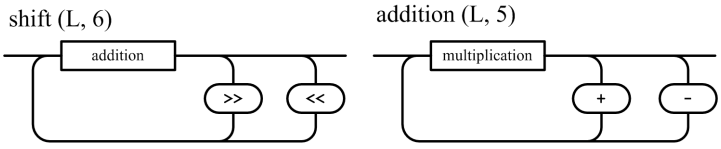


Figure E.14: Bitwise Shift, Addition and Subtraction Operators

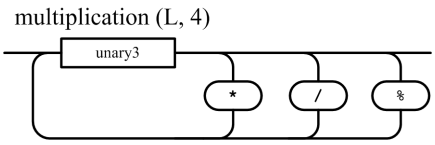


Figure E.15: Multiplication and Division Operators

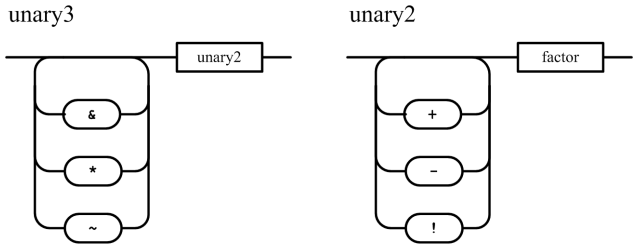


Figure E.16: Unary Operators

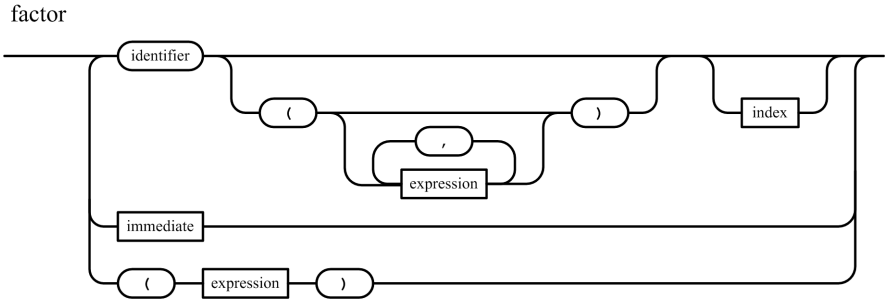


Figure E.17: Factor (Variable, Immediate or Expression)

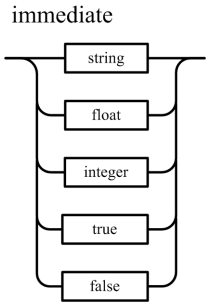


Figure E.18: Immediate (Literal) Value

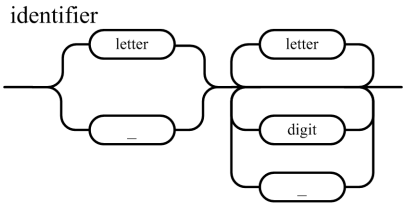


Figure E.19: Literal identifier

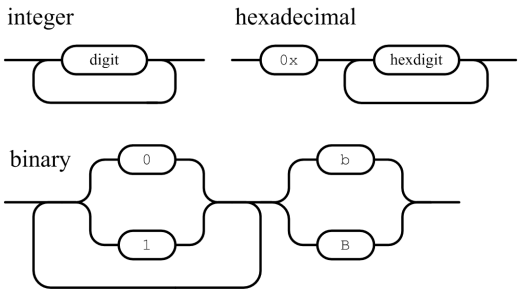


Figure E.20: Literal integer number

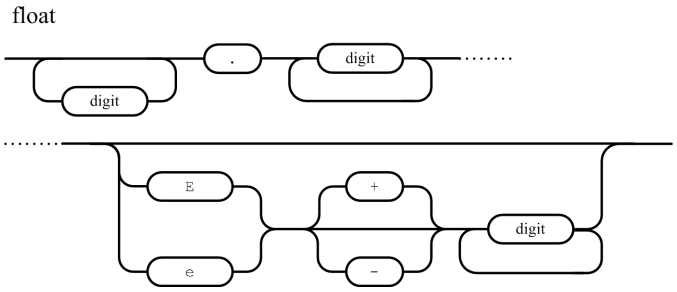


Figure E.21: Literal float number

Appendix F

Inger Lexical Analyzer Source

F.1 tokens.h

```
#ifndef TOKENS_H
#define TOKENS_H

#include "defs.h"
/* #include "type.h" */
#include "tokenvalue.h"
#include "ast.h"

/*
 *
 *  MACROS
 *
 */

/* Define where a line starts (at position 1)
 */
#define LINECOUNTBASE 1
/* Define the position of a first character of a line.
 */
#define CHARPOSBASE 1
/* Define the block size with which strings are allocated.
 */
#define STRING_BLOCK 100

/*
 *
 *  TYPES
 *
 */

/* This enum contains all the keywords and operators
```

```
 * used in the language.
 */
enum
{
    /* Keywords */
    KW_BREAK        = 1000, /* "break" keyword */
    KW_CASE,                /* "case" keyword */
    KW_CONTINUE,            /* "continue" keyword */
    KW_DEFAULT,             /* "default" keyword */
    KW_DO,                  /* "do" keyword */
    KW_ELSE,                /* "else" keyword */
    KW_EXTERN,              /* "extern" keyword */
    KW_GOTO,                /* "goto" keyword */
    KW_IF,                  /* "if" keyword */
    KW_LABEL,               /* "label" keyword */
    KW_MODULE,              /* "module" keyword */
    KW_RETURN,              /* "return"keyword */
    KW_START,               /* "start" keyword */
    KW_SWITCH,              /* "switch" keyword */
    KW_WHILE,               /* "while" keyword */

    /* Type identifiers */
    KW_BOOL,                /* "bool" identifier */
    KW_CHAR,                /* "char" identifier */
    KW_FLOAT,               /* "float" identifier */
    KW_INT,                 /* "int" identifier */
    KW_UNTYPED,             /* "untyped" identifier */
    KW_VOID,                /* "void" identifier */

    /* Variable lexer tokens */
    LIT_BOOL,               /* bool constant */
    LIT_CHAR,               /* character constant */
    LIT_FLOAT,              /* floating point constant */
    LIT_INT,                /* integer constant */
    LIT_STRING,             /* string constant */
    IDENTIFIER,             /* identifier */

    /* Operators */
    OP_ADD,                 /* "+" */
    OP_ASSIGN,              /* "=" */
    OP_BITWISE_AND,         /* "&" */
    OP_BITWISE_COMPLEMENT, /* "~" */
    OP_BITWISE_LSHIFT,      /* "<<" */
    OP_BITWISE_OR,          /* "|" */
    OP_BITWISE_RSHIFT,      /* ">>" */
    OP_BITWISE_XOR,         /* "^" */
    OP_DIVIDE,              /* "/" */
    OP_EQUAL,               /* "==" */
    OP_GREATER,             /* ">" */
    OP_GREATEREQUAL,        /* ">=" */
    OP_LESS,                /* "<" */
    OP_LESSEQUAL,           /* "<=" */
    OP_LOGICAL_AND,         /* "&&" */
    OP_LOGICAL_OR,          /* "||" */
```

```
    OP_MODULUS,             /* "%" */
    OP_MULTIPLY,            /* "*" */
    OP_NOT,                 /* "!" */
    OP_NOTEQUAL,            /* "!=" */
    OP_SUBTRACT,            /* "-" */
    OP_TERNARY_IF,          /* "?" */

    /* Delimiters */
    ARROW,                  /* "->" */
    LBRACE,                 /* "{" */
    RBRACE,                 /* "}" */
    LBRACKET,               /* "[" */
    RBRACKET,               /* "]" */
    COLON,                  /* ":" */
    COMMA,                  /* "," */
    LPAREN,                 /* "(" */
    RPAREN,                 /* ")" */
    SEMICOLON               /* ";" */
}
tokens;

/*
 *
 *  FUNCTION DECLARATIONS
 *
 */

TreeNode *Parse();

/*
 *
 *  GLOBALS
 *
 */

extern Tokenvalue tokenvalue;

#endif
```

F.2 lexer.l

```
%{

/* Include stdlib for string to number conversion routines. */
#include <stdlib.h>
/* Include errno for errno system variable. */
#include <errno.h>
/* Include string.h to use strtoul(). */
#include <string.h>
/* include assert.h for assert macro. */
#include <assert.h>
/* Include global definitions. */
```

```
#include "defs.h"
/* The token #defines are defined in tokens.h. */
#include "tokens.h"
/* Include error/warning reporting module. */
#include "errors.h"
/* Include option.h to access command line option. */
#include "options.h"

/*
 *
 * MACROS
 *
 */
#define INCPOS charPos += yyleng;


/*
 *
 * FORWARD DECLARATIONS
 *
 */
char SlashToChar( char str [] );
void AddToString( char c );

/*
 *
 * GLOBALS
 *
 */

/*
 * Tokenvalue (declared in tokens.h) is used to pass
 * literal token values to the parser.
 */
Tokenvalue tokenvalue;

/*
 * lineCount keeps track of the current line number
 * in the source input file.
 */
int lineCount;

/*
 * charPos keeps track of the current character
 * position on the current source input line.
 */
int charPos;

/*
 * Counters used for string reading
 */
static int stringSize , stringPos;

/*
```

```
 * commentsLevel keeps track of the current
 * comment nesting level, in order to ignore nested
 * comments properly.
 */
static int commentsLevel = 0;


%}

 /*
  *
  * LEXER STATES
  *
  */

/* Exclusive state in which the lexer ignores all input
   until a nested comment ends. */
%x STATE_COMMENTS
/* Exclusive state in which the lexer returns all input
   until a string terminates with a double quote. */
%x STATE_STRING


%pointer

 /*
  *
  * REGULAR EXPRESSIONS
  *
  */
%%

 /*
  *
  * KEYWORDS
  *
  */

 start                  { INCPOS; return KW_START; }

 bool                   { INCPOS; return KW_BOOL; }
 char                   { INCPOS; return KW_CHAR; }
 float                  { INCPOS; return KW_FLOAT; }
 int                    { INCPOS; return KW_INT; }
 untyped                { INCPOS; return KW_UNTYPED; }
 void                   { INCPOS; return KW_VOID; }

 break                  { INCPOS; return KW_BREAK; }
 case                   { INCPOS; return KW_CASE; }
 default                { INCPOS; return KW_DEFAULT; }
 do                     { INCPOS; return KW_DO; }
 else                   { INCPOS; return KW_ELSE; }
 extern                 { INCPOS; return KW_EXTERN; }
 goto_considered_harmful { INCPOS; return KW_GOTO; }
```

```
120 if                  { INCPOS; return KW_IF; }
    label               { INCPOS; return KW_LABEL; }
    module              { INCPOS; return KW_MODULE; }
    return              { INCPOS; return KW_RETURN; }
    switch              { INCPOS; return KW_SWITCH; }
125 while               { INCPOS; return KW_WHILE; }


     /*
      *
130   * OPERATORS
      *
      */

    "->"                { INCPOS; return ARROW; }
135 "=="                { INCPOS; return OP_EQUAL; }
    "!="                { INCPOS; return OP_NOTEQUAL; }
    "&&"                { INCPOS; return OP_LOGICAL_AND; }
    "||"                { INCPOS; return OP_LOGICAL_OR; }
    ">="                { INCPOS; return OP_GREATEREQUAL; }
140 "<="                { INCPOS; return OP_LESSEQUAL; }
    "<<"                { INCPOS; return OP_BITWISE_LSHIFT; }
    ">>"                { INCPOS; return OP_BITWISE_RSHIFT; }
    "+"                 { INCPOS; return OP_ADD; }
    "-"                 { INCPOS; return OP_SUBTRACT; }
145 "*"                 { INCPOS; return OP_MULTIPLY; }

    "/"                 { INCPOS; return OP_DIVIDE; }
    "!"                 { INCPOS; return OP_NOT; }
    "~"                 { INCPOS; return OP_BITWISE_COMPLEMENT; }
150 "%"                 { INCPOS; return OP_MODULUS; }
    "="                 { INCPOS; return OP_ASSIGN; }

    ">"                 { INCPOS; return OP_GREATER; }
    "<"                 { INCPOS; return OP_LESS; }
155 "&"                 { INCPOS; return OP_BITWISE_AND; }
    "|"                 { INCPOS; return OP_BITWISE_OR; }

    "^"                 { INCPOS; return OP_BITWISE_XOR; }
    "?"                 { INCPOS; return OP_TERNARY_IF; }
160


     /*
      *
      * DELIMITERS
165   *
      */

    "("                 { INCPOS; return LPAREN; }
    ")"                 { INCPOS; return RPAREN; }
170 "["                 { INCPOS; return LBRACKET; }
    "]"                 { INCPOS; return RBRACKET; }
    ":"                 { INCPOS; return COLON; }
    ";"                 { INCPOS; return SEMICOLON; }
```

```
"{"                     { INCPOS; return LBRACE; }
"}"                     { INCPOS; return RBRACE; }
","                     { INCPOS; return COMMA; }


 /*
  *
  * VALUE TOKENS
  *
  */

true        { /* boolean constant */
              INCPOS;
              tokenvalue.boolvalue = TRUE;
              return( LIT_BOOL );
            }

false       { /* boolean constant */
              INCPOS;
              tokenvalue.boolvalue = FALSE;
              return( LIT_BOOL );
            }

[0-9]+      { /* decimal integer constant */
              INCPOS;
              tokenvalue.uintvalue = strtoul( yytext, NULL, 10 );
              if( tokenvalue.uintvalue == -1 )
              {
                  tokenvalue.uintvalue = 0;
                  AddPosWarning( "integer literal value "
                        "too large. Zero used",
                        lineCount, charPos );
              }
              return( LIT_INT );
            }

"0x"[0-9A-Fa-f]+ {
              /* hexidecimal integer constant */
              INCPOS;
              tokenvalue.uintvalue = strtoul( yytext, NULL, 16 );
              if( tokenvalue.uintvalue == -1 )
              {
                  tokenvalue.uintvalue = 0;
                  AddPosWarning( "hexadecimal integer literal value "
                                 "too large. Zero used",
                                 lineCount, charPos );
              }
              return( LIT_INT );
            }

[0-1]+[Bb] { /* binary integer constant */
              INCPOS;
              tokenvalue.uintvalue = strtoul( yytext, NULL, 2 );
              if( tokenvalue.uintvalue == -1 )
```

```
                {
                    tokenvalue.uintvalue = 0;
                    AddPosWarning( "binary integer literal value too "
                                   "large. Zero used", 
                                   lineCount, charPos );
                }
                return( LIT_INT );
            }

[_A-Za-z]+[_A-Za-z0-9]* {
            /* identifier */
                INCPOS;
                tokenvalue.identifier = strdup( yytext );
                return( IDENTIFIER );
                }

[0-9]*\.[0-9]+([Ee][+-]?[0-9]+)? {
                /* floating point number */
                INCPOS;
                if ( sscanf( yytext, "%f",
                     &tokenvalue.floatvalue ) == 0 )
                {
                  tokenvalue.floatvalue = 0;
                  AddPosWarning( "floating point literal value too "
                                 "large. Zero used",
                                 lineCount, charPos );
                }
                return( LIT_FLOAT );
                }



 /*
  *
  * CHARACTERS
  *
  */

\'\\[\'\" abfnrtv]\' {
                    INCPOS;
                    yytext[strlen(yytext)-1] = '\0';
                    tokenvalue.charvalue =
                        SlashToChar( yytext+1 );
                    return( LIT_CHAR );
                    }

\'\\B[0-1][0-1][0-1][0-1][0-1][0-1][0-1][0-1]\' {
                    /* \B escape sequence. */
                    INCPOS;
                    yytext[strlen(yytext)-1] = '\0';
                    tokenvalue.charvalue =
                        SlashToChar( yytext+1 );
                    return( LIT_CHAR );
                    }
```

```
\'\\o[0-7][0-7][0-7]\' {
                    /* \o escape sequence. */
                    INCPOS;
                    yytext[ strlen(yytext)-1] = '\0';
                    tokenvalue.charvalue =
                        SlashToChar( yytext+1 );
                    return( LIT_CHAR );
                }

\'\\x[0-9A-Fa-f][0-9A-Fa-f]\' {
                    /* \x escape sequence. */
                    INCPOS;
                    yytext[ strlen(yytext)-1] = '\0';
                    tokenvalue.charvalue =
                        SlashToChar( yytext+1 );
                    return( LIT_CHAR );
                }

\'.\'           {
                    /* Single character. */
                    INCPOS;
                    tokenvalue.charvalue = yytext[1];
                    return( LIT_CHAR );
                }



 /*
  *
  * STRINGS
  *
  */

\"              { INCPOS;
                  tokenvalue.stringvalue =
                  (char*) malloc( STRING_BLOCK );
                  memset( tokenvalue.stringvalue,
                      0, STRING_BLOCK );
                  stringSize = STRING_BLOCK;
                  stringPos = 0;
                  BEGIN STATE_STRING; /* begin of string */
                }

<STATE_STRING>\" {
                    INCPOS;
                    BEGIN 0;
                    /* Do not include terminating " in string */
                    return( LIT_STRING ); /* end of string */
                }

<STATE_STRING>\n {
                    INCPOS;
                    AddPosWarning( "strings cannot span multiple "
```

```
                                    " lines ", lineCount , charPos );
                  AddToString( '\n' );
                }

<STATE_STRING>\\[\'\"abfnrtv] {
                  /* Escape sequences in  string . */
                  INCPOS;
                  AddToString( SlashToChar( yytext ) );
                }

<STATE_STRING>\\B[0-1][0-1][0-1][0-1][0-1][0-1][0-1][0-1] {
                  /* \B escape sequence. */
                  INCPOS;
                  AddToString( SlashToChar( yytext ) );
                }

<STATE_STRING>\\o[0-7][0-7][0-7] {
                  /* \o escape sequence. */
                  INCPOS;
                  AddToString( SlashToChar( yytext ) );
                }

<STATE_STRING>\\x[0-9A-Fa-f][0-9A-Fa-f] {
                  /* \x escape sequence. */
                  INCPOS;
                  AddToString( SlashToChar( yytext ) );
                }

<STATE_STRING>. {
                  /* Any other character */
                  INCPOS;
                  AddToString( yytext [0] );
                }


 /*
  *
  * LINE COMMENTS
  *
  */

"//"[^\n]*      { ++lineCount; /* ignore comment lines */ }


 /*
  *
  * BLOCK COMMENTS
  *
  */

"/*"            { INCPOS;
                  ++commentsLevel;
                  BEGIN STATE_COMMENTS;
                  /* start  of comments */
```

```
390                 }

    <STATE_COMMENTS>"/*" {
                    INCPOS;
                    ++commentsLevel;
395                 /* begin of deeper nested
                       comments */
                }

    <STATE_COMMENTS>. { INCPOS; /* ignore all characters */ }
400
    <STATE_COMMENTS>\n {
                    charPos = 0;
                    ++lineCount; /* ignore newlines */
                }
405
    <STATE_COMMENTS>"*/" {
                    INCPOS;
                    if ( --commentsLevel == 0 )
                      BEGIN 0; /* end of comments*/
410                 }


     /*
      *
415   * WHITESPACE
      *
      */

    [\t ]           { ++charPos; /* ignore whitespaces */ }
420
    \n              { ++lineCount;
                    charPos = 0; /* ignored  newlines */
                    }

425                 /* unmatched character */
    .               { INCPOS; return yytext[0]; }

    %%


430
     /*
      *
      * ADDITIONAL VERBATIM C CODE
      *
435   */

    /*
     * Convert slashed  character  (e.g. \n, \r  etc.)  to  a
     * char  value.
440  * The argument is a  string  that  start  with  a  backslash,
     * e.g. \x2e, \o056, \n, \b11011101
     *
     * Pre:  (for \x, \B and \o): strlen(yytext)  is  large
```

```
 *        enough. The regexps in the lexer take care
 *        of this.
 */
char SlashToChar( char str[] )
{
    static char strPart[20];

    memset( strPart, 0, 20 );

    switch( str[1] )
    {
    case '\\':
        return( '\\' );
    case '\"':
        return( '\"' );
    case '\'':
        return( '\'' );
    case 'a':
        return( '\a' );
    case 'b':
        return( '\b' );
    case 'B':

        strncpy( strPart, str+2, 8 );
        return( strtoul( yytext+2, NULL, 2 ) );
    case 'f':
        return( '\f' );
    case 'n':
        return( '\n' );
    case 'o':
        strncpy( strPart, str+2, 3 );
        return( strtoul( strPart, NULL, 8 ) );
    case 't':
        return( '\t' );
    case 'r':
        return( '\r' );
    case 'v':
        return( '\v' );
    case 'x':
        strncpy( strPart, str+2, 2 );
        return( strtoul( strPart, NULL, 16 ) );
    default:
        /* Control should never get here! */
        assert( 0 );
    }
}


/*
 * For string reading (which happens on a
 * character-by-character basis), add a character to
 * the global lexer string 'tokenvalue.stringvalue'.
 */
void AddToString( char c )
```

```
    {
        if ( tokenvalue.stringvalue == NULL )
500     {
            /* Some previous realloc () already went wrong.
             * Silently abort.
             */
            return;
505     }

        if ( stringPos >= stringSize - 1 )

        {
510         stringSize += STRING_BLOCK;
            DEBUG( "resizing string memory +%d, now %d bytes",
                    STRING_BLOCK, stringSize );

            tokenvalue.stringvalue =
515             (char*) realloc ( tokenvalue.stringvalue ,
                 stringSize );
            if ( tokenvalue.stringvalue == NULL )
            {
                AddPosWarning( "Unable to claim enough memory "
520                             "for string storage",
                                lineCount, charPos );
                return;
            }
            memset( tokenvalue.stringvalue + stringSize
525                 - STRING_BLOCK, 0, STRING_BLOCK );
        }

        tokenvalue.stringvalue[stringPos] = c;
        stringPos++;
530 }
```

Appendix G

Logic Language Parser Source

G.1 Lexical Analyzer

```
%{

#include "lexer.h"

unsigned int nr = 0;

%}

%%

\n              {
    nr = 0;
}

[ \t]+      {
    nr += yyleng;
}

"->"            {
    nr += yyleng;
    return (RIMPL);
}

"<-"            {
    nr += yyleng;
    return (LIMPL);
}

"<->"           {
    nr += yyleng;
    return (EQUIV);
```

```
}

[A-Z]{1}    {
    nr += yyleng;
    return (IDENT);
}

"RESULT"    {
    nr += yyleng;
    return (RESULT);
}

"PRINT" {
    nr += yyleng;
    return (PRINT);
}

.                       {
    nr += yyleng;
    return (yytext[0]);
}


%%
```

G.2 Parser Header

```
#ifndef _LEXER_H
#define _LEXER_H 1

enum
{
    LIMPL = 300,
    RIMPL,
    EQUIV,
    RESULT,
    PRINT,
    IDENT
};

#endif
```

G.3 Parser Source

```
#include <stdio.h>
#include <stdlib.h>

#include "lexer.h"

```

```
#ifdef DEBUG
#   define debug(args ...)   printf (args)
#else
#   define debug (...)
#endif

extern unsigned char* yytext;
extern unsigned int nr;
extern int yylex (void);

unsigned int token;

// who needs complex datastructures anyway?
unsigned int acVariables [26];

void gettoken (void)
{
    token = yylex ();
    debug("new token: %s\n", yytext);
}

void error (char * e)
{
     fprintf (stderr, "ERROR(%d:%c): %s\n",
       nr, yytext [0], e);
    exit (1);
}

void statement (void);
int negation (void);
int restnegation (int);
int conjunction (void);
int restconjunction (int);
int implication (void);
int restimplication (int);
int factor (void);

void statement (void)
{
    int res = 0, i, var = 0;
    debug("statement()\n");

    if (token == IDENT)
    {
        var = yytext[0] - 65;
        gettoken ();
        if (token == '=')
        {
            gettoken();
            res = implication ();
            acVariables[var] = res;
            if (token != ';')
                error ("; expected");
            gettoken();
```

```
60              } else {
                    error ("= expected");
                }
            } else {
                error ("This shouldn't have happened.");
65          }
            for (i = 0; i < 26; i++)
                debug ("%d", acVariables[i]);
            debug ("\n");
        }
70
        int implication (void)
        {
            int res = 0;
            debug("implication()\n");
75          res = conjunction();
            res = restimplication (res);
            return (res);
        }

80      int restimplication (int val)
        {
            int res = val;
            int operator;
            debug(" restimplication ()\n");
85
            if (token == EQUIV || token == RIMPL || token == LIMPL)
            {
                operator = token;
                gettoken();
90              res = conjunction();
                switch (operator)
                {
                case RIMPL:
                    res = (val == 0) || (res == 1) ? 1 : 0;
95                  break;
                case LIMPL:
                    res = (val == 1) || (res == 0) ? 1 : 0;
                    break;
                case EQUIV:
100                 res = (res == val) ? 1 : 0;
                    break;
                }
                res = restimplication (res);
            }
105         return (res);
        }

        int conjunction (void)
        {
110         int res = 0;
            debug("conjunction()\n");
            res = negation();
            res = restconjunction (res);
```

```
    return (res);
}

int restconjunction (int val)
{
    int res = val, operator;
    debug("restconjunction()\n");
    if (token == '&' || token == '|')
    {
        operator = token;
        gettoken();
        res = negation();
        if (operator == '&')
        {
            res = ((res == 1) && (val == 1)) ? 1 : 0;
        } else { /* '|' */
            res = ((res == 1) || (val == 1)) ? 1 : 0;
        }
        res = restconjunction(res);
    }
    return (res);
}

int negation (void)
{
    int res = 0;
    debug("negation()\n");
    if (token == '~')
    {
        gettoken();
        res = negation() == 0 ? 1 : 0;
    } else {
        res = factor();
    }
    return (res);
}

int factor (void)
{
    int res = 0;
    debug("factor()\n");
    switch (token)
    {
        case '(':
            gettoken();
            res = implication();
            if (token != ')')
                error ("missing ')'");
            break;
        case '1':
            res = 1;
            break;
        case '0':
```

```
                res = 0;
                break;
        case IDENT:
                debug("'%s' processed\n", yytext);
                res = acVariables[yytext[0] - 65];
                break;
        default:
                error ("(, 1, 0 or identifier expected");
    }
    debug ("factor is returning %d\n", res);
    gettoken();
    return (res);
}

void program (void)
{
    while (token == IDENT || token == PRINT)
    {
        if (token == IDENT)
        {
            statement();
        }
        else if (token == PRINT)
        {
            gettoken();
            printf ("%d\n", implication());
            if (token != ';')
                error (";expected");
            gettoken();
        }
    }
}

int main (void)
{
    int i = 0;
    for (i = 0; i < 26; i++)
        acVariables[i] = 0;

    /* start off */
    gettoken();
    program ();
    return (0);
}
```

Listings

Index