From: John C. Mitchell Concepts in Programming Languages, Cambridge University Press, 2003

4 Fundamentals

In this chapter some background is provided on programming language implementation through brief discussions of syntax, parsing, and the steps used in conventional compilers. We also look at two foundational frameworks that are useful in programming language analysis and design: lambda calculus and denotational semantics. Lambda calculus is a good framework for defining syntactic concepts common to many programming languages and for studying symbolic evaluation. Denotational semantics shows that, in principle, programs can be reduced to functions.

A number of other theoretical frameworks are useful in the design and analysis of programming languages. These range from computability theory, which provides some insight into the power and limitations of programs, to type theory, which includes aspects of both syntax and semantics of programming languages. In spite of many years of theoretical research, the current programming language theory still does not provide answers to some important foundational questions. For example, we do not have a good mathematical theory that includes higher-order functions, state transformations, and concurrency. Nonetheless, theoretical frameworks have had an impact on the design of programming languages and can be used to identify problem areas in programming languages. To compare one aspect of theory and practice, we compare functional and imperative languages in Section 4.4.

4.1 COMPILERS AND SYNTAX

A program is a description of a dynamic process. The text of a program itself is called its syntax; the things a program does comprise its semantics. The function of a programming language implementation is to transform program syntax into machine instructions that can be executed to cause the correct sequence of actions to occur.

4.1.1 Structure of a Simple Compiler

Programming languages that are convenient for people to use are built around concepts and abstractions that may not correspond directly to features of the underlying machine. For this reason, a program must be translated into the basic instruction

JOHN BACKUS

An early pioneer, John Backus became a computer programmer at IBM in 1950. In the 1950s, Backus developed Fortran, the first high-level computer language, which became commercially available in 1957. The language is still widely used for numerical and scientific programming. In 1959, John Backus invented Backus naur form (BNF), the standard notation for defining the syntax of a programming language. In later years, he became an advocate of pure functional programming, devoting his 1977 ACM Turing Award lecture to this topic.

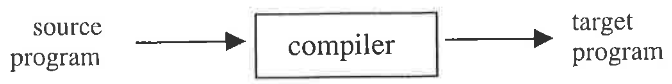
I met John Backus through IFIP WG 2.8, a working group of the International Federation of Information Processing on functional programming. Backus continued to work on functional programming at IBM Almaden through the 1980s, although his group was disbanded after his retirement. A mild-mannered and unpretentious individual, here is a quote that gives some sense of his independent, pioneering spirit:

"I really didn't know what the hell I wanted to do with my life. I decided that what I wanted was a good hi fi set because I liked music. In those days, they didn't really exist so I went to a radio technicians' school. I had a very nice teacher – the first good teacher I ever had – and he asked me to cooperate with him and compute the characteristics of some circuits for a magazine."

"I remember doing relatively simple calculations to get a few points on a curve for an amplifier. It was laborious and tedious and horrible, but it got me interested in math. The fact that it had an application – that interested me."

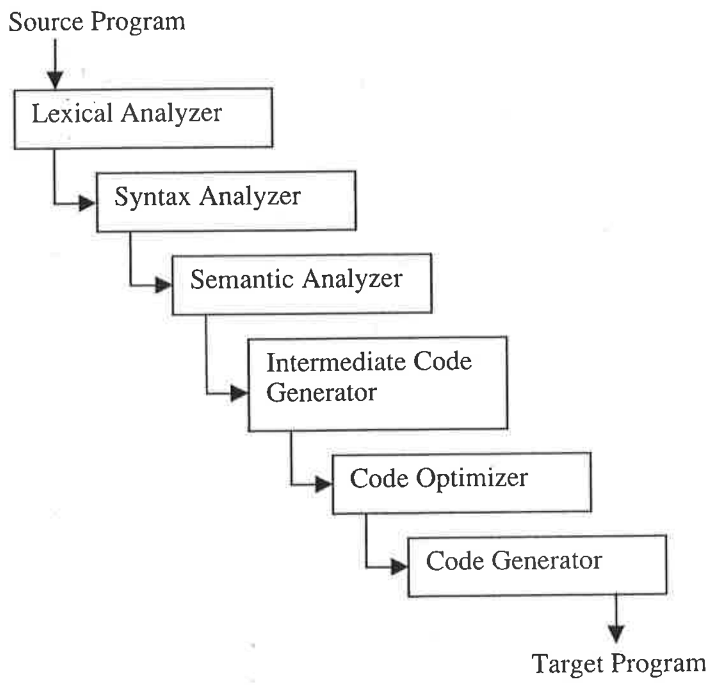
set of the machine before it can be executed. This can be done by a *compiler*, which translates the entire program into machine code before the program is run, or an *interpreter*, which combines translation and program execution. We discuss programming language implementation by using compilers, as this makes it easier to separate the main issues and to discuss them in order.

The main function of a compiler is illustrated in this simple diagram:



Given a program in some *source language*, the compiler produces a program in a *target language*, which is usually the instruction set, or machine language, of some machine.

Most compilers are structured as a series of phases, with each phase performing one step in the translation of source program to target program. A typical compiler might consist of the phases shown in the following diagram:



Each of these phases is discussed briefly. Our goal with this book is only to understand the parts of a compiler so that we can discuss how different programming language features might be implemented. We do not discuss how to build a compiler. That is the subject of many books on compiler construction, such as *Compilers: Principles, Techniques and Tools* by Aho, Sethi, and Ullman (Addison-Wesley, 1986), and *Modern Compiler Implementation in Java/ML/C* by Appel (Cambridge Univ. Press, 1998).

Lexical Analysis

The input symbols are scanned and grouped into meaningful units called *tokens*. For example, lexical analysis of the expression temp := x+1, which uses Algol-style notation := for assignment, would divide this sequence of symbols into five tokens: the identifier temp, the assignment "symbol" :=, the variable x, the addition symbol +, and the number 1. Lexical analysis can distinguish numbers from identifiers. However, because lexical analysis is based on a single left-to-right (and top-to-bottom) scan, lexical analysis does not distinguish between identifiers that are names of variables and identifiers that are names of constants. Because variables and constants are declared differently, variables and constants are distinguished in the semantic analysis phase.

Syntax Analysis

In this phase, tokens are grouped into syntactic units such as expressions, statements, and declarations that must conform to the grammatical rules of the programming language. The action performed during this phase, called *parsing*, is described in Subsection 4.1.2. The purpose of parsing is to produce a data structure called a parse tree, which represents the syntactic structure of the program in a way that is convenient for subsequent phases of the compiler. If a program does not meet the syntactic requirements to be a well-formed program, then the parsing phase will produce an error message and terminate the compiler.

Semantic Analysis

In this phase of a compiler, rules and procedures that depend on the context surrounding an expression are applied. For example, returning to our sample expression temp := x+1, we find that it is necessary to make sure that the types match. If this assignment occurs in a language in which integers are automatically converted to floats as needed, then there are several ways that types could be associated with parts of this expression. In standard semantic analysis, the types of temp and x would be determined from the declarations of these identifiers. If these are both integers, then the number 1 could be marked as an integer and + marked as integer addition, and the expression would be considered correct. If one of the identifiers, say x, is a float, then the number 1 would be marked as a float and + marked as a floating-point addition. Depending on whether temp is a float or an integer, it might also be necessary to insert a conversion around the subexpression x+1. The output of this phase is an augmented parse tree that represents the syntactic structure of the program and includes additional information such as the types of identifiers and the place in the program where each identifier is declared.

Although the phase following parsing is commonly called *semantic analysis*, this use of the word *semantic* is different from the standard use of the term for *meaning*. Some compiler writers use the word semantic because this phase relies on context information, and the kind of grammar used for syntactic analysis does not capture context information. However, in the rest of this book, the word semantics is used to refer to how a program executes, not the essentially syntactic properties that arise in the third phase of a compiler.

Intermediate Code Generation

Although it might be possible to generate a target program from the results of syntactic and semantic analysis, it is difficult to generate efficient code in one phase. Therefore, many compilers first produce an intermediate form of code and then optimize this code to produce a more efficient target program.

Because the last phase of the compiler can translate one set of instructions to another, the intermediate code does not need to be written with the actual instruction set of the target machine. It is important to use an intermediate representation that is easy to produce and easy to translate into the target language. The intermediate representation can be some form of generic low-level code that has properties common to several computers. When a single generic intermediate representation is used, it is possible to use essentially the same compiler to generate target programs for several different machines.

Code Optimization

There are a variety of techniques that may be used to improve the efficiency of a program. These techniques are usually applied to the intermediate representation. If several optimization techniques are written as transformations of the intermediate representation, then these techniques can be applied over and over until some termination condition is reached.

The following list describes some standard optimizations:

- *Common Subexpression Elimination:* If a program calculates the same value more than once and the compiler can detect this, then it may be possible to transform the program so that the value is calculated only once and stored for subsequent use.
- *Copy Propagation:* If a program contains an assignment such as x=y, then it may be possible to change subsequent statements to refer to y instead of to x and to eliminate the assignment.
- *Dead-Code Elimination:* If some sequence of instructions can never be reached, then it can be eliminated from the program.
- *Loop Optimizations:* There are several techniques that can be applied to remove instructions from loops. For example, if some expression appears inside a loop but has the same value on each pass through the loop, then the expression can be moved outside the loop.
- *In-Lining Function Calls:* If a program calls function f, it is possible to substitute the code for f into the place where f is called. This makes the target program more efficient, as the instructions associated with calling a function can be eliminated, but it also increases the size of the program. The most important consequence of in-lining function calls is usually that they allow other optimizations to be performed by removing jumps from the code.

Code Generation

The final phase of a standard compiler is to convert the intermediate code into a target machine code. This involves choosing a memory location, a register, or both, for each variable that appears in the program. There are a variety of register allocation algorithms that try to reuse registers efficiently. This is important because many machines have a fixed number of registers, and operations on registers are more efficient than transferring data into and out of memory.

4.1.2 Grammars and Parse Trees

We use grammars to describe various languages in this book. Although we usually are not too concerned about the pragmatics of parsing, we take a brief look in this subsection at the problem of producing a parse tree from a sequence of tokens.

Grammars

Grammars provide a convenient method for defining infinite sets of expressions. In addition, the structure imposed by a grammar gives us a systematic way of processing expressions.

A *grammar* consists of a start symbol, a set of nonterminals, a set of terminals, and a set of productions. The nonterminals are symbols that are used to write out the grammar, and the terminals are symbols that appear in the language generated by the grammar. In books on automata theory and related subjects, the productions of a grammar are written in the form $s \rightarrow tu$, with an arrow, meaning that in a string containing the symbol s, we can replace s with the symbols tu. However, here we use a more compact notation, commonly referred to as BNF.

The main ideas are illustrated by example. A simple language of numeric expressions is defined by the following grammar:

```
e ::= n | e+e | e-e
n ::= d | nd
d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

where e is the *start symbol*, symbols e, n, and d are nonterminals, and 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, and - are the terminals. The language defined by this grammar consists of all the sequences of terminals that we can produce by starting with the start symbol e and by replacing nonterminals according to the preceding productions. For example, the first preceding production means that we can replace an occurrence of e with the symbol n, the three symbols e+e, or the three symbols e-e. The process can be repeated with any of the preceding three lines.

Some expressions in the language given by this grammar are

```
0, 1 + 3 + 5, 2 + 4 - 6 - 8
```

Sequences of symbols that contain nonterminals, such as

```
e, e + e, e + 6 - e
```

are not expressions in the language given by the grammar. The purpose of nonterminals is to keep track of the form of an expression as it is being formed. All nonterminals must be replaced with terminals to produce a well-formed expression of the language.

Derivations

A sequence of replacement steps resulting in a string of terminals is called a *derivation*.

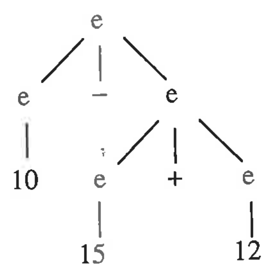
Here are two derivations in this grammar, the first given in full and the second with a few missing steps that can be filled in by the reader (be sure you understand how!):

```
e ⟶ n ⟶ nd ⟶ dd ⟶ 2d ⟶ 25
e ⟶ e - e ⟶ e - e+e ⟶ ··· ⟶ n-n+n ⟶ ··· ⟶ 10-15+12
```

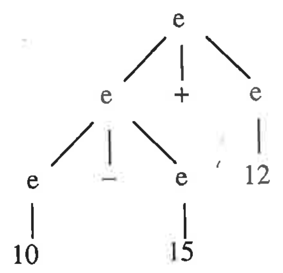
Parse Trees and Ambiguity

It is often convenient to represent a derivation by a tree. This tree, called the *parse tree* of a derivation, or *derivation tree*, is constructed with the start symbol as the root of the tree. If a step in the derivation is to replace s with $x_1, \ldots, x_n$, then the children of s in the tree will be nodes labeled $x_1, \ldots, x_n$.

The parse tree for the derivation of 10-15+12 in the preceding subsection has some useful structure. Specifically, because the first step yields e-e, the parse tree has the form



where we have contracted the subtrees for each two-digit number to a single node. This tree is different from



which is another parse tree for the same expression. An important fact about parse trees is that each corresponds to a unique parenthesization of the expression. Specifically, the first tree corresponds to 10-(15+12) whereas the second corresponds to (10-15)+12. As this example illustrates, the value of an expression may depend on how it is parsed or parenthesized.

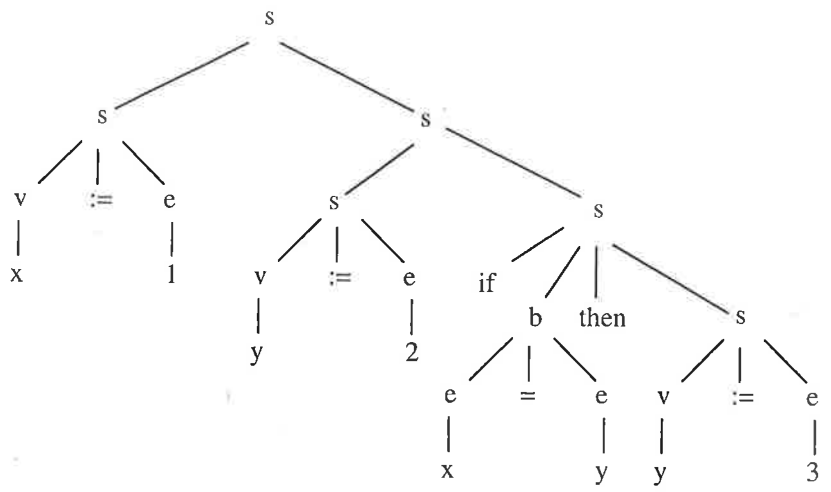
A grammar is *ambiguous* if some expression has more than one parse tree. If every expression has at most one parse tree, the grammar is *unambiguous*.

Example 4.1

There is an interesting ambiguity involving if-then-else. This can be illustrated by the following simple grammar:

```
s ::= v := e | s;s | if b then s | if b then s else s
v ::= x | y | z
e ::= v | 0 | 1 | 2 | 3 | 4
b ::= e=e
```

where s is the start symbol, s, v, e, and b are nonterminals, and the other symbols are terminals. The letters s, v, e, and b stand for statement, variable, expression, and Boolean test, respectively. We call the expressions of the language generated by this grammar *statements*. Here is an example of a well-formed statement and one of its parse trees:



```
x := 1; y := 2; if x=y then y := 3
```

This statement also has another parse tree, which we obtain by putting two assignments to the left of the root and the if-then statement to the right. However, the difference between these two parse trees will not affect the behavior of code generated by an ordinary compiler. The reason is that $s_1;s_2$ is normally compiled to the code for s_1 followed by the code for s_2. As a result, the same code would be generated whether we consider $s_1;s_2;s_3$ as $(s_1;s_2);s_3$ or $s_1;(s_2;s_3)$.

A more complicated situation arises when if-then is combined with if-then-else in the following way:

```
if b₁ then if b₂ then s₁ else s₂
```

What should happen if b_1 is true and b_2 is false? Should s_2 be executed or not? As you can see, this depends on how the statement is parsed. A grammar that allows this combination of conditionals is ambiguous, with two possible meanings for statements of this form.

4.1.3 Parsing and Precedence

Parsing is the process of constructing parse trees for sequences of symbols. Suppose we define a language L by writing out a grammar G. Then, given a sequence of symbols s, we would like to determine if s is in the language L. If so, then we would like to compile or interpret s, and for this purpose we would like to find a parse tree for s. An algorithm that decides whether s is in L, and constructs a parse tree if it is, is called a *parsing algorithm* for G.

There are many methods for building parsing algorithms from grammars. Many of these work for only particular forms of grammars. Because parsing is an important part of compiling programming languages, parsing is usually covered in courses and textbooks on compilers. For most programming languages you might consider, it is either straightforward to parse the language or there are some changes in syntax that do not change the structure of the language very much but make it possible to parse the language efficiently.

Two issues we consider briefly are the syntactic conventions of precedence and right or left associativity. These are illustrated briefly in the following example.

Example 4.2
A programming language designer might decide that expressions should include addition, subtraction, and multiplication and write the following grammar:

```
e ::= 0 | 1 | e+e | e-e | e*e
```

This grammar is ambiguous, as many expressions have more than one parse tree. For expressions such as 1-1+1, the value of the expression will depend on the way it is parsed. One solution to this problem is to require complete parenthesization. In other words, we could change the grammar to

```
e ::= 0 | 1 | (e+e) | (e-e) | (e*e)
```

so that it is no longer ambiguous. However, as you know, it can be awkward to write a lot of parentheses. In addition, for many expressions, such as 1+2+3+4, the value of the expression does not depend on the way it is parsed. Therefore, it is unnecessarily cumbersome to require parentheses for every operation.

The standard solution to this problem is to adopt parsing conventions that specify a single parse tree for every expression. These are called *precedence* and *associativity*. For this specific grammar, a natural precedence convention is that multiplication ($*$) has a higher precedence than addition ($+$) and subtraction ($-$). We incorporate precedence into parsing by treating an unparenthesized expression e op_1 e op_2 e as if parentheses are inserted around the operator of higher precedence. With this rule in effect, the expression 5*4-3 will be parsed as if it were written as (5*4)-3. This coincides with the way that most of us would ordinarily think about the expression 5*4-3. Because there is no standard way that most readers would parse 1+1-1, we might give addition and subtraction equal precedence. In this case, a compiler could issue an error message requiring the programmer to parenthesize 1+1-1. Alternatively, an expression like this could be disambiguated by use of an additional convention.

Associativity comes into play when two operators of equal precedence appear next to each other. Under left associativity, an expression e op_1 e op_2 e would be parsed as (e op_1 e) op_2 e, if the two operators have equal precedence. If we adopted a right-associativity convention instead, e op_1 e op_2 e would be parsed as e op_1 (e op_2 e).

Expression	Precedence	Left Associativity	Right Associativity
5∗4-3	(5∗4)-3	(no change)	(no change)
1+1-1	(no change)	(1+1)-1	1+(1-1)
2+3-4∗5+2	2+3-(4∗5)+2	((2+3)-(4*5))+2	2+(3-((4*5)+2))

4.2 LAMBDA CALCULUS

Lambda calculus is a mathematical system that illustrates some important programming language concepts in a simple, pure form. Traditional lambda calculus has three main parts: a notation for defining functions, a proof system for proving equations between expressions, and a set of calculation rules called reduction. The first word in the name lambda calculus comes from the use of the Greek letter lambda (λ) in function expressions. (There is no significance to the letter λ.) The second word comes from the way that reduction may be used to *calculate* the result of applying a function to one or more arguments. This calculation is a form of symbolic evaluation of expressions. Lambda calculus provides a convenient notation for describing programming languages and may be regarded as the basis for many language constructs. In particular, lambda calculus provides fundamental forms of parameterization (by means of function expressions) and binding (by means of declarations). These are basic concepts that are common to almost all modern programming languages. It is therefore useful to become familiar enough with lambda calculus to regard expressions in your favorite programming language as essentially a form of lambda expression. For simplicity, the untyped lambda calculus is discussed; there are also typed versions of lambda calculus. In typed lambda calculus, there are additional type-checking constraints that rule out certain forms of expressions. However, the basic concepts and calculation rules remain essentially the same.

4.2.1 Functions and Function Expressions

Intuitively, a function is a rule for determining a value from an argument. This view of functions is used informally in most mathematics. (See Subsection 2.1.2 for a discussion of functions in mathematics.) Some examples of functions studied in mathematics are

$$f(x) = x^2 + 3,$$

$$g(x + y) = \sqrt{x^2 + y^2}.$$

In the simplest, pure form of lambda calculus, there are no domain-specific operators such as addition and exponentiation, only function definition and application. This allows us to write functions such as

$$h(x) = f(g(x))$$

because h is defined solely in terms of function application and other functions that we assume are already defined. It is possible to add operations such as addition