This member-only story is on us. Upgrade to access all of Medium.

✦ Member-only story

SOFTWARE ENGINEERING

Compiler vs. Interpreter: Know The Difference And When To Use Each Of Them

Types and use cases of compilers and interpreters

Rakia Ben Sassi · Follow

Published in Better Programming · 7 min read · Jan 19, 2021

350

Photo by Cookie the Pom on Unsplash

I still remember a discussion with a colleague of mine in which I said, “That’s the transpiler,” and he replied: “Trans…what?”

If you have never heard that name, you’re not alone. As developers, we all get used to writing code in a high-level language that humans can understand. However, computers can only understand a program written in a binary system known as machine code.

To speak to a computer in its non-human language, we came up with two solutions: interpreters and compilers. Ironically, most of us know very little about them, although they belong to our daily coding life.

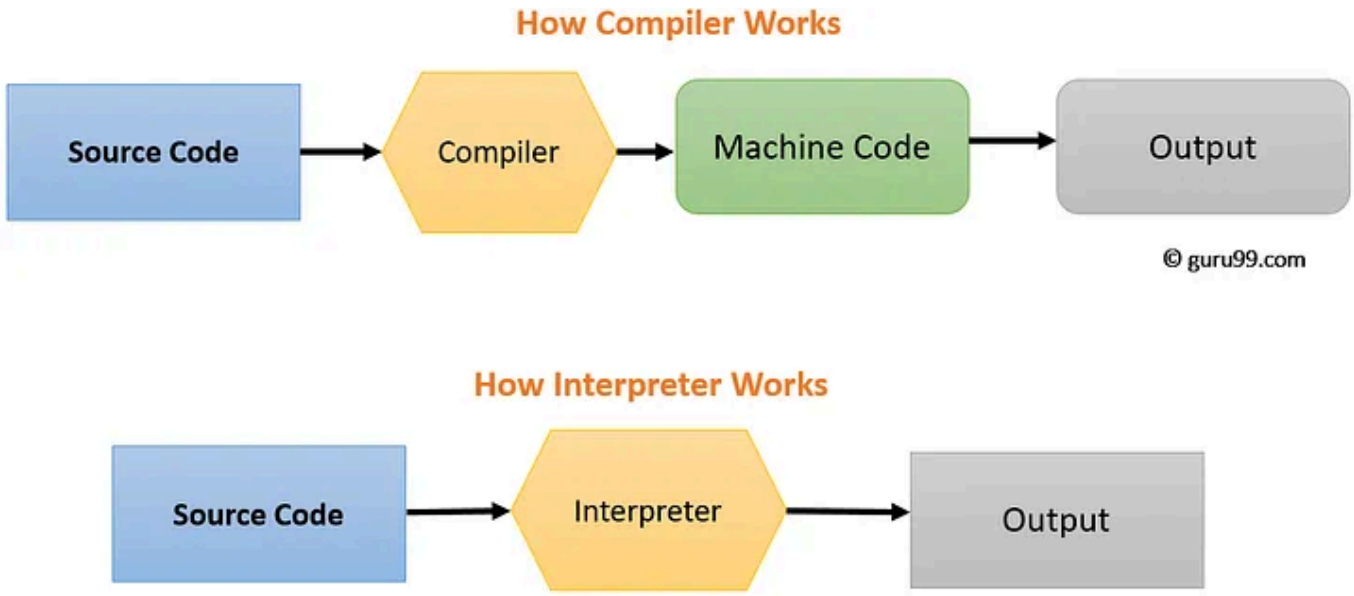
In this post, I’ll dive into the journey of translating a high-level language into a machine code ready for execution. I’ll focus on the inner working of the

two key players in this game — the compiler and the interpreter — and break down the related concepts.

The Journey From High- to Lower-Level Language

Compilers and interpreters have long been used as computer programs to transform code. But they work in different ways:

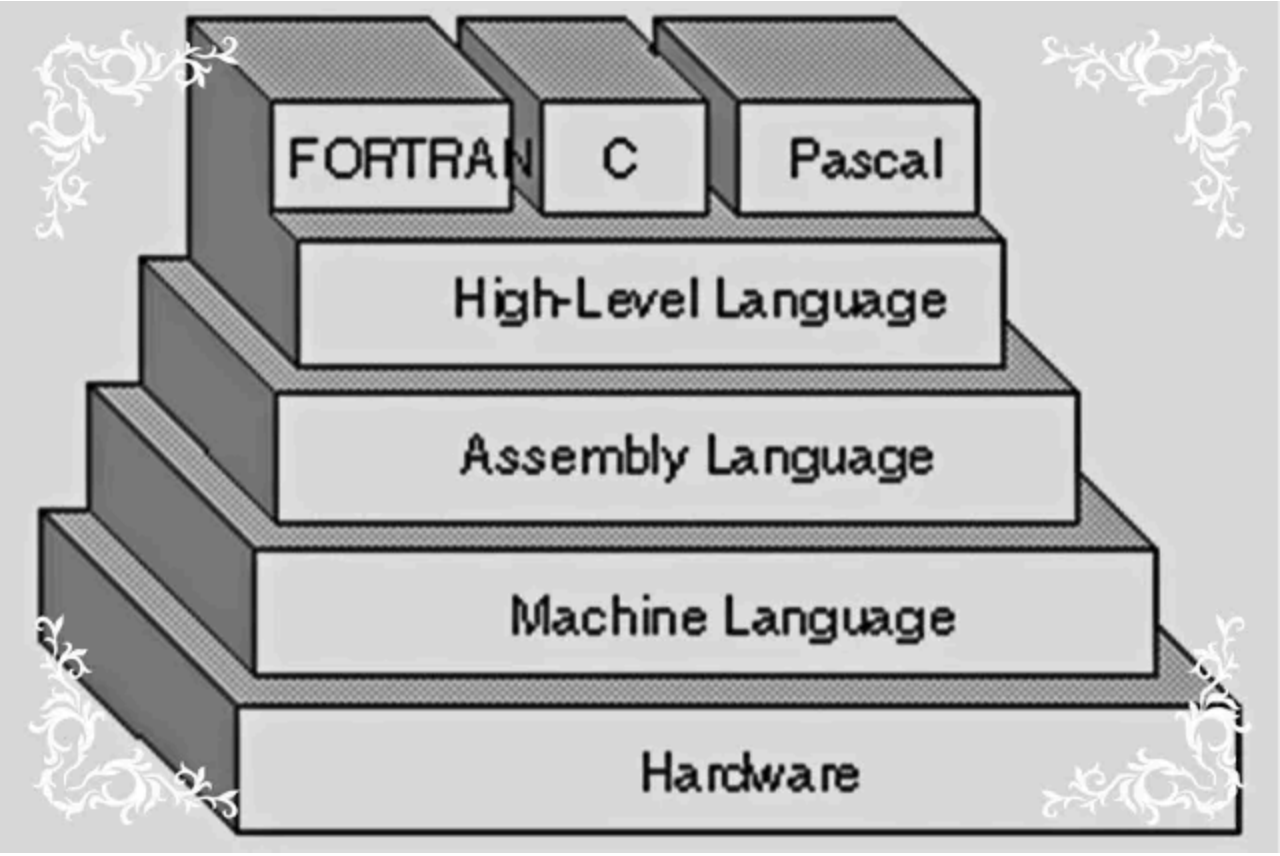
- A compiler translates a code written in a high-level programming language into *a lower-level language like assembly language, object code, and* machine code (binary 1 and 0 bits). It converts the code ahead of time before the program runs.
- An interpreter translates the code line by line when the program is running. You've likely used interpreters unknowingly at some point in your work career.



Compiler vs Interpreter

Both compilers and interpreters have pros and cons:

- A compiler takes an entire program and a lot of time to analyze the source code, whereas the interpreter takes a single line of code and very little time to analyze it.
- A compiled code runs faster while interpreted code runs slower.
- A compiler displays all errors after compilation. If your code has mistakes, it will not compile. But the interpreter displays errors of each line one by one.
- Interpretation does not replace compilation completely.
- Compilers can contain interpreters for optimization reasons like faster performance and smaller memory footprint.



Assembly Language

A high-level programming language is usually referred to as “compiled language” or “interpreted language.” However, in practice, they can have both compiled and interpreted implementations. C, for example, is called a compiled language, despite the existence of C interpreters. The first JavaScript engines were simple interpreters, but all modern engines use just-in-time (JIT) compilation for performance reasons.

Types of Interpreter

Interpreters were used as early as 1952 to ease programming and also used to translate between low-level machine languages. The first interpreted high-level language was Lisp. Python, Ruby, Perl, and PHP are other examples of programming languages that use interpreters.

Below is a non-exclusive list of interpreter’s types:

1. Bytecode interpreter

The trend toward bytecode interpretation and just-in-time compilation blurs the distinction between compilers and interpreters.

> *“In a bytecode interpreter each instruction starts with a byte, and therefore bytecode interpreters have up to 256 instructions, although not all may be used. Some bytecodes may take multiple bytes, and may be arbitrarily complicated.”*
>
> — *Wikipedia*

2. Threaded code interpreter

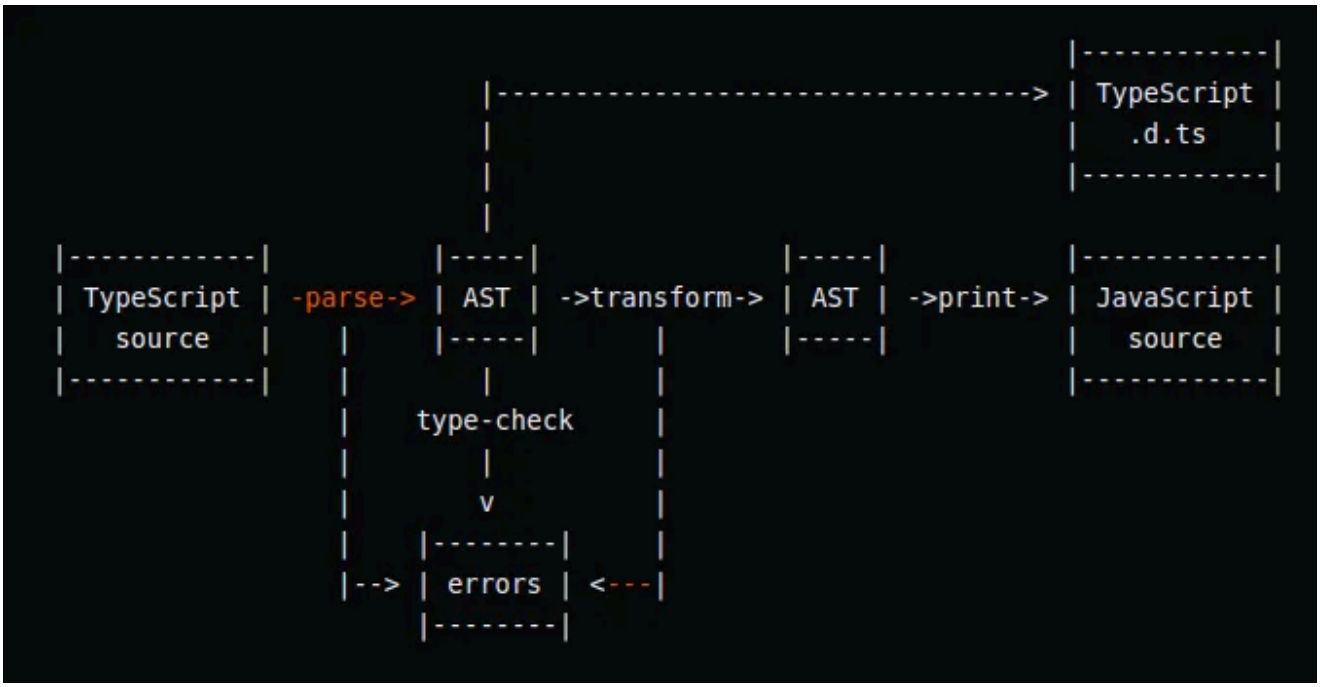
Unlike bytecode interpreters, threaded code interpreters use pointers instead of bytes. Each instruction is a word pointing to a function or an

instruction sequence, possibly followed by a parameter. The number of different instructions is limited by available memory and address space.

The Forth code — used in Open Firmware systems — is a classical example of threaded code. The source code is compiled into a bytecode known as "F code", then interpreted by a virtual machine.

3. Abstract syntax tree interpreter

If you're a TypeScript developer and you have some insights about the TypeScript architecture, you may have heard the acronym AST for Abstract Syntax Tree.



TypeScript Transpiler Architecture

AST is an approach to transform the source code into an optimized abstract

Open in app ↗

Medium Search Write

AST keeps the global program structure and relations between statements. This allows the system to perform better analysis during runtime and makes AST a better intermediate format for just-in-time compilers than bytecode representation.

However, for interpreters, AST causes more overhead. Interpreters walking the abstract syntax tree are slower than those generating bytecode.

4. Just-in-time compilation

Just-in-time compilation (JIT) is a technique in which the intermediate representation is compiled to native machine code at runtime.

Types of Compiler

1. Cross-compiler

A compiler running on a computer whose CPU or operating system differs from the one on which the code it produces will run.

2. Native compiler

A compiler producing an output that would run on the same type of computer and operating system as the compiler itself.

3. Bootstrap compiler

A compiler written in the language that it intends to compile.

4. Decompiler

A decompiler translates code from a low-level language to a higher level one.

5. Source-to-source compiler (Transpiler)

It's a program that translates between high-level languages. This type of compilers is also known as a transcompiler or transpiler.

Examples:

- Emscripten: transpiles C/C++ to JavaScript.
- Babel: transpiles JavaScript code from ES6+ to ES5.
- Cfront: the original compiler for C++ (from around 1983). It used C as its target language and created C code with no indent style and no pretty C intermediate code, since the generated code was usually not intended to be readable by humans.

6. A language rewriter

This is usually a program translating form of expressions without a change of language.

7. Bytecode compiler

A compiler that translates a high-level language into an intermediate simple language that can be interpreted by a bytecode interpreter or a virtual machine.

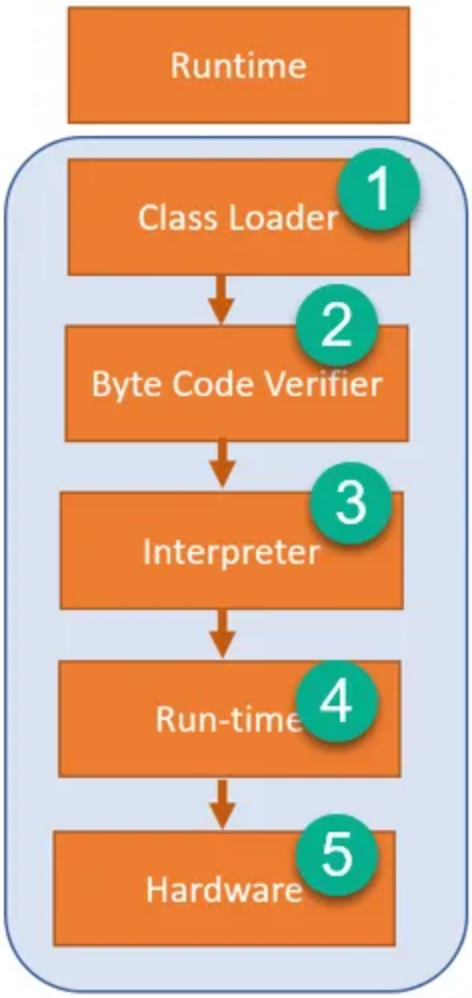
Examples: Bytecode compilers for Java and Python.

8. Just-in-time compiler (JIT Compiler)

JIT compiler defers compilation until runtime. It generally runs inside an interpreter.

Examples:

- The earliest published JIT compiler is attributed to *LISP* in 1960.
- The latter technique appeared in languages such as Smalltalk in the 1980s.
- Then JIT compilation has gained mainstream attention amongst modern language like Java, .NET Framework, Python, and most modern JavaScript implementations.



JRE (Java Runtime Environment) Functionality

In Java, source files are first compiled and converted into `.class` files which contain Java bytecode (highly optimized set of instructions), then a bytecode interpreter executes the bytecode, and later the JIT compiler translates the bytecode to machine code.

Java bytecode can either be interpreted at runtime by a virtual machine, or compiled at load time or runtime into native code. Modern JVM implementations use the compilation approach, so after the initial startup time the performance is equivalent to native code.

9. AOT Compilation

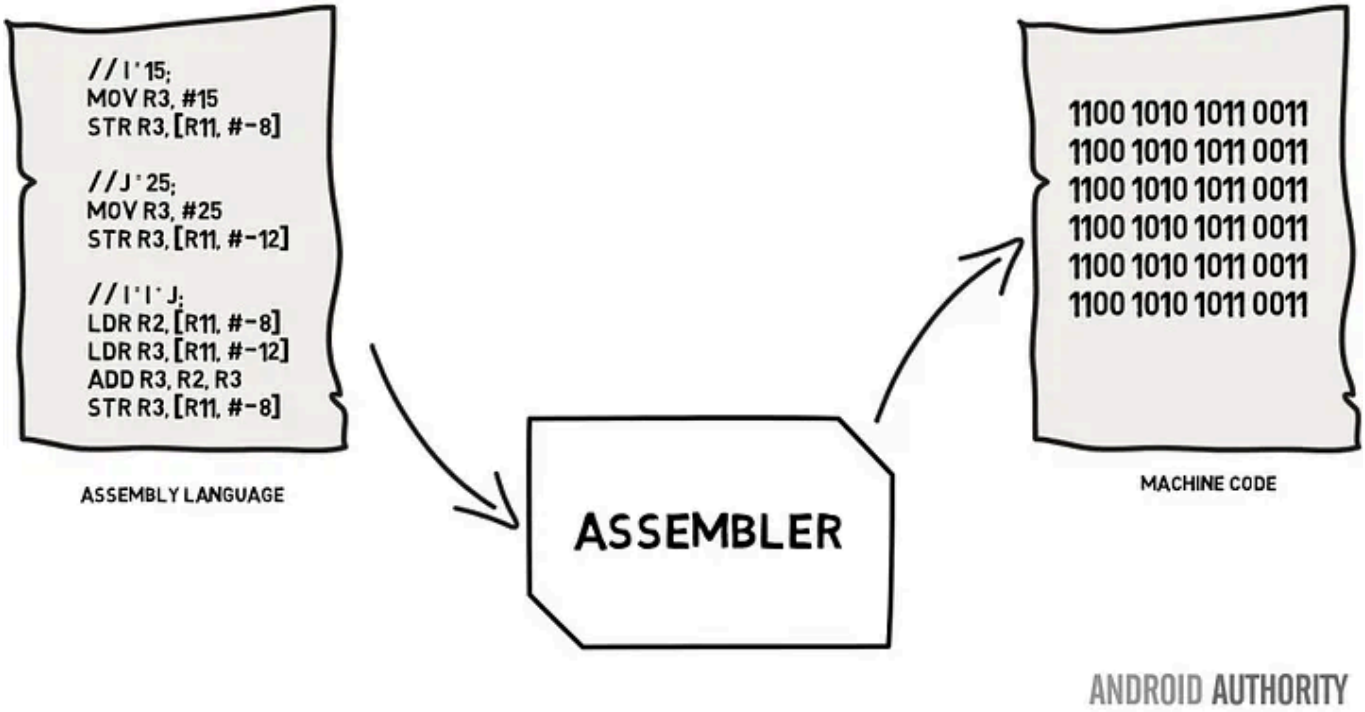
Ahead-of-time (AOT) compilation is the approach of compiling a higher-level programming language, or an intermediate representation such as Java bytecode, before the runtime.

Example:

The Angular framework uses an ahead-of-time (AOT) compiler to transform HTML and TypeScript code into JavaScript code during the build time to provide a faster rendering later on the browser when the code is running.

10. Assembler

An assembler translates human-readable assembly language into machine code. This compilation process is called assembly. The inverse program that converts machine code to assembly language is called a disassembler.



Assembler

An assembly language (abbreviated ASM) is a low-level programming language in which there is a dependence on the machine code instructions. That's why every assembly language is designed for exactly one specific computer architecture.

Takeaway

Both compilers and interpreters are computer programs that convert a code written in a high-level language into a lower-level or machine code understood by computers. However, there are differences in how they work and when to use them.

Even if you're not going to implement the next compiler or interpreter, these insights should help to improve your knowledge of the tools you use as a developer every day.

🧠💡 I write about engineering, technology, and leadership for a community of smart, curious people. **Join my free email newsletter for exclusive access** or sign up for Medium here.

*You can check my **video course** on Udemy: How to Identify, Diagnose, and Fix Memory Leaks in Web Apps.*

Programming JavaScript Technology Software Development Software Engineering

Written by Rakia Ben Sassi

6.3K Followers · Writer for Better Programming

Follow

Example 01

Prepared By: Sukarna Sarker

E -> TE’	First(E) ->
E’ -> +TE’ | ε	First(E’) ->
T -> FT’	First(T) -> {id , (}
T’-> +FT’ | ε	First(T’) -> {+, ε }
F -> id | (E)	First(F) -> {id , (}

Explanation:

First(F) -> {id , (}

There are two production rule where F is the production head:

For F -> id, the production body starts with a terminal **id**

For F -> (E), the production body starts with a terminal **(**

First(T’) -> {+, ε }

There are two production rule where T’ is the production head:

For T’ -> +FT’, the production body starts with a terminal **+**

For T’ -> ε, the production body is ε, whenever a nonterminal derives ε, we place ε in FIRST for that nonterminal.

First(T) -> {id, (}

There is one production rule where T is the production head:

T -> FT’, the production body starts with a non terminal F so everything FIRST(F) is surely in FIRST(T).

Prepared By: Sukarna Sarker

E -> TE’	First(E) -> {id , (}
E’ -> +TE’ | ε	First(E’) -> {+, ε}
T -> FT’	First(T) -> {id , (}
T’-> +FT’ | ε	First(T’) -> {+, ε }
F -> id | (E)	First(F) -> {id , (}

Explanation:

First(E’) -> {+, ε}

There are two production rule where E’ is the production head:

For E’ -> +TE’, the production body starts with a terminal **+**

For E’ -> ε, the production body is ε, whenever a nonterminal derives ε, we place ε in FIRST for that nonterminal

First(E) -> {id , (}

There is one production rule where E is the production head:

For E -> TE’, the production body starts with a non terminal T so everything FIRST(T) is surely in FIRST(E).

Prepared By: Sukarna Sarker

E -> TE’	Follow(E) -> {), $}
E’ -> +TE’ | ε	Follow(E’) -> {), $}
T -> FT’	Follow(T) ->
T’-> +FT’ | ε	Follow(T’) ->
F -> id | (E)	Follow(F) ->

Explanation:

Follow(E) -> {), $ }

There is one production rule where E is being followed.

For F -> (E), E is being followed by a terminal **)**

and as E is the start symbol there should be **$** in the Follow(E)

Follow(E’) -> {), $}

There are two production rules where E’ is being followed.

For E -> TE’, since E’ is at the end of the production, so FOLLOW(E') = FOLLOW(E).

For E’ -> +TE’, since E’ is at the end of the production,so FOLLOW(E') = FOLLOW(E’).

Prepared By: Sukarna Sarker

E -> TE’	Follow(E) -> {), $}
E’ -> +TE’ | ε	Follow(E’) -> {), $}
T -> FT’	Follow(T) -> {+,), $}
T’-> +FT’ | ε	Follow(T’) -> {+,), $}
F -> id | (E)	Follow(F) ->

Explanation:

Follow(T) -> {+,), $}

There are two production rules where T is being followed.

For E -> TE’, since E’ is following T, so FOLLOW(T) = FIRST(E’). But we can see that in the FIRST(T) there is ε, and we also know that ε can not be in a follow set. So, let’s see who is after E’. As we can see that there are no one after T’, so we will go to the head of the production rule, so FOLLOW(T) = FOLLOW(E).

For E’ -> +TE’, E’ is following T, so FOLLOW(T) = FIRST(E’). But ε can not be in the follow set and no one is following T anymore, we will go to the production head and will take followset of E’

Follow(T’) -> {+,), $}

There are two production rules where T’ is being followed.

For T -> FT’ since T’ is at the end of the production, so FOLLOW(T') = FOLLOW(T)

For T’-> +FT’, since T’ is at the end of the production, so FOLLOW(T') = FOLLOW(T)

Prepared By: Sukarna Sarker

E -> TE’	Follow(E) -> {), $}
E’ -> +TE’ | ε	Follow(E’) -> {), $}
T -> FT’	Follow(T) -> {+,), $}
T’-> +FT’ | ε	Follow(T’) -> {+,), $}
F -> id | (E)	Follow(F) -> {+,), $ }

Explanation:

Follow(F) -> {+,), $ }

There are two production rules where F is being followed.

For T -> FT’, T’ is following F so FOLLOW(F) = FIRST(T’). But we can see that in the FIRST(T’) there is ε, and we also know that ε can not be in a follow set. So, let’s see who is after T’. As we can see that there are no one after T’, so we will go to the head of the production rule, so FOLLOW(F) = FOLLOW(T).

For T’ -> +FT’, T’ is following F so FOLLOW(F) = FIRST(T’). But we can see that in the FIRST(T’) there is ε, and we also know that ε can not be in a follow set. So, let’s see who is after T’. As we can see that there are no one after T’, so we will go to the head of the production rule, so FOLLOW(F) = FOLLOW(T).

Prepared By: Sukarna Sarker

Example 02

Prepared By: Sukarna Sarker

S -> Aa
A -> BD
B -> b | ε
D -> d | ε

First(S) -> {b, d, a}
First(A) -> {b, d, ε }
First(B) -> {b, ε }
First(D) -> {d, ε }

Explanation:

First(D) -> {d, ε}
There are two production rule where D is the production head:
For D-> d, the production body starts with a terminal **d**
For F -> ε, the production body is ε, whenever a nonterminal derives ε, we place ε in FIRST for that nonterminal

First(B) -> {b, ε }
There are two production rule where B is the production head:
For B->b, the production body starts with a terminal **b**
For B -> ε, the production body is ε, whenever a nonterminal derives ε, we place ε in FIRST for that nonterminal

First(A) -> {b, d, ε}
There is one production rule where A is the production head:
For A-> BD, the production body starts with a non terminal B so everything in FIRST(B) is surely in FIRST(A). As, in First(B) there is ε, add the non ε symbols of FIRST(D),again in First(D) there is ε and as there are no nonterminal left add ε to First(A)

Prepared By: Sukarna Sarker

S -> Aa	First(S) -> {b, d, a}
A -> BD	First(A) -> {b, d, ε }
B -> b | ε	First(B) -> {b, ε }
D -> d | ε	First(D) -> {d, ε }

Explanation:

First(S) -> {b, d, a}

There is one production rule where S is the production head:

For S-> Aa, the production body starts with a non terminal A, so everything FIRST(A) is surely in FIRST(S). As, in First(A) there is ε, so add a in the first set.

Prepared By: Sukarna Sarker

S -> Aa
A -> BD
B -> b | ε
D -> d | ε

Follow(S) -> {$}
Follow(A) -> {a}
Follow(B) -> {d, a}
Follow(D) ->

Explanation:

Follow(S) -> {$}
S is not being followed anywhere. But as S is the start symbol there should be **$** in the Follow(S)

Follow(A) -> {a}
There are one production rule where A is being followed.
For S -> Aa, a is following A.

Follow(B) ->
There are one production rule where B is being followed.
For A-> BD, D is following B, So, Follow(B) = First(D). But we can see that in the FIRST(D) there is ε, and we also know that ε can not be in a follow set. So, let's see who is after D. As we can see that there are no one after D, so we will go to the head of the production rule, so FOLLOW(B) = FOLLOW(A).

Prepared By: Sukarna Sarker

S -> Aa	Follow(S) -> {$}
A -> BD	Follow(A) -> {a}
B -> b | ε	Follow(B) -> {d, a}
D -> d | ε	Follow(D) -> {a}

Explanation:

Follow(S]D) -> {$}

S is not being followed anywhere. But as S is the start symbol there should be **$** in the Follow(S)

Follow(D) -> {a}

There are one production rule where D is being followed.

For A-> BD, there are no one after D, so we will go to the head of the production rule, so FOLLOW(D) = FOLLOW(A).

Prepared By: Sukarna Sarker

Chapter 1

Introduction

Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer.

The software systems that do this translation are called *compilers.*

This book is about how to design and implement compilers. We shall discover that a few basic ideas can be used to construct translators for a wide variety of languages and machines. Besides compilers, the principles and techniques for compiler design are applicable to so many other domains that they are likely to be reused many times in the career of a computer scientist. The study of compiler writing touches upon programming languages, machine architecture, language theory, algorithms, and software engineering.

In this preliminary chapter, we introduce the different forms of language translators, give a high level overview of the structure of a typical compiler, and discuss the trends in programming languages and machine architecture that are shaping compilers. We include some observations on the relationship between compiler design and computer-science theory and an outline of the applications of compiler technology that go beyond compilation. We end with a brief outline of key programming-language concepts that will be needed for our study of compilers.

1.1 Language Processors

Simply stated, a compiler is a program that can read a program in one language — the *source* language — and translate it into an equivalent program in another language — the *target* language; see Fig. 1.1. An important role of the compiler is to report any errors in the source program that it detects during the translation process.

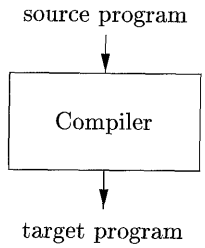


Figure 1.1: A compiler

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs; see Fig. 1.2.

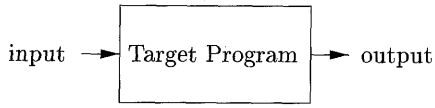


Figure 1.2: Running the target program

An *interpreter* is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user, as shown in Fig. 1.3.

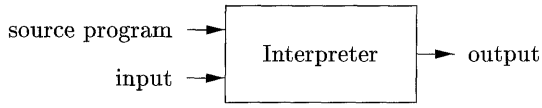


Figure 1.3: An interpreter

The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs . An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Example 1.1: Java language processors combine compilation and interpretation, as shown in Fig. 1.4. A Java source program may first be compiled into an intermediate form called *bytecodes*. The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.

In order to achieve faster processing of inputs to outputs, some Java compilers, called *just-in-time* compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input. □

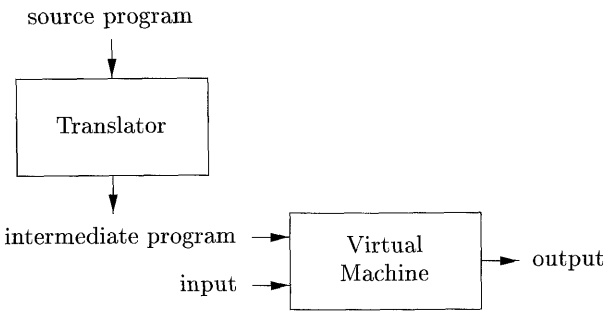


Figure 1.4: A hybrid compiler

In addition to a compiler, several other programs may be required to create an executable target program, as shown in Fig. 1.5. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a *preprocessor*. The preprocessor may also expand shorthands, called macros, into source language statements.

The modified source program is then fed to a compiler. The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug. The assembly language is then processed by a program called an *assembler* that produces relocatable machine code as its output.

Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine. The *linker* resolves external memory addresses, where the code in one file may refer to a location in another file. The *loader* then puts together all of the executable object files into memory for execution.

1.1.1 Exercises for Section 1.1

Exercise 1.1.1: What is the difference between a compiler and an interpreter?

Exercise 1.1.2: What are the advantages of (a) a compiler over an interpreter (b) an interpreter over a compiler?

Exercise 1.1.3: What advantages are there to a language-processing system in which the compiler produces assembly language rather than machine language?

Exercise 1.1.4: A compiler that translates a high-level language into another high-level language is called a *source-to-source* translator. What advantages are there to using C as a target language for a compiler?

Exercise 1.1.5: Describe some of the tasks that an assembler needs to perform.

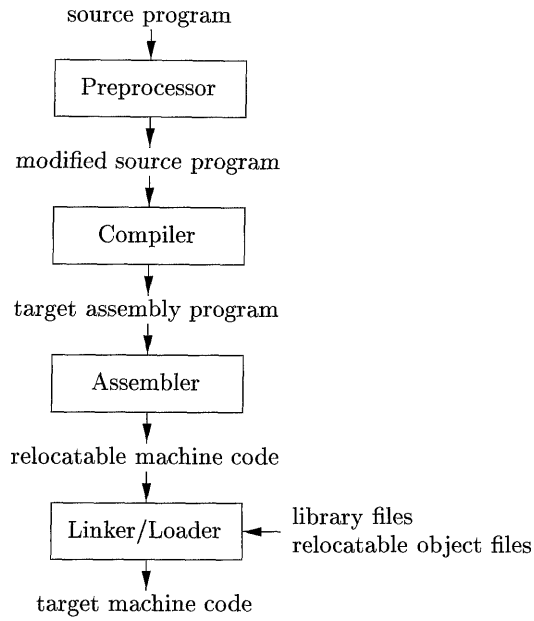


Figure 1.5: A language-processing system

1.2 The Structure of a Compiler

Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: analysis and synthesis.

The *analysis* part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a *symbol table*, which is passed along with the intermediate representation to the synthesis part.

The *synthesis* part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the *front end* of the compiler; the synthesis part is the *back end.*

If we examine the compilation process in more detail, we see that it operates as a sequence of *phases*, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in Fig. 1.6. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the

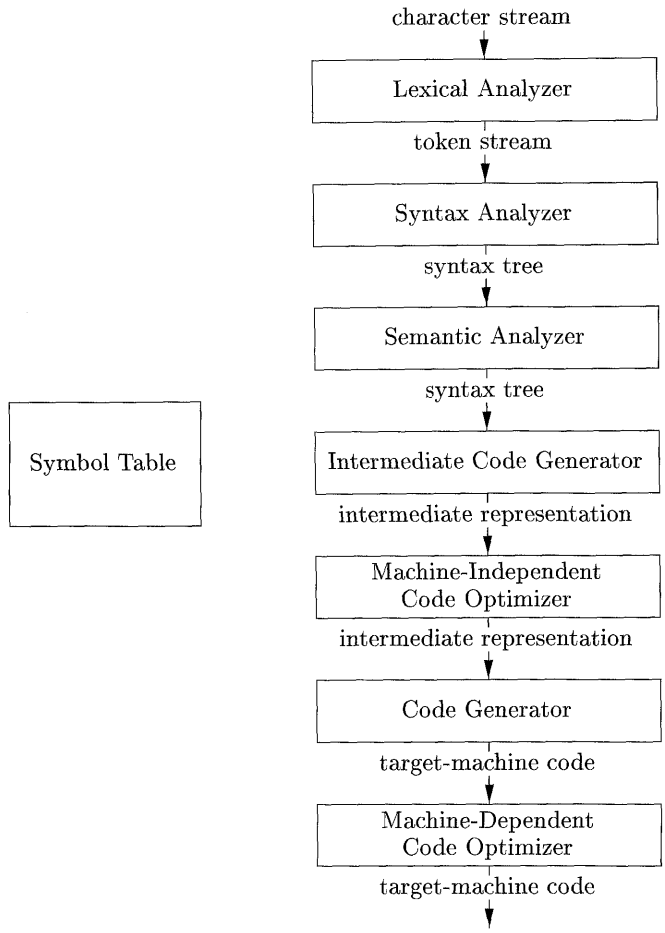


Figure 1.6: Phases of a compiler

entire source program, is used by all phases of the compiler.

Some compilers have a machine-independent optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program than it would have otherwise produced from an unoptimized intermediate representation. Since optimization is optional, one or the other of the two optimization phases shown in Fig. 1.6 may be missing.

1.2.1 Lexical Analysis

The first phase of a compiler is called *lexical analysis* or *scanning*. The lexical analyzer reads the stream of characters making up the source program

and groups the characters into meaningful sequences called *lexemes*. For each lexeme, the lexical analyzer produces as output a *token* of the form

$$\langle \textit{token-name}, \textit{attribute-value} \rangle$$

that it passes on to the subsequent phase, syntax analysis. In the token, the first component *token-name* is an abstract symbol that is used during syntax analysis, and the second component *attribute-value* points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation.

For example, suppose a source program contains the assignment statement

```
position = initial + rate * 60        (1.1)
```

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. `position` is a lexeme that would be mapped into a token $\langle \mathbf{id}, 1 \rangle$, where **id** is an abstract symbol standing for *identifier* and 1 points to the symbol-table entry for `position`. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

2. The assignment symbol `=` is a lexeme that is mapped into the token $\langle = \rangle$. Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.

3. `initial` is a lexeme that is mapped into the token $\langle \mathbf{id}, 2 \rangle$, where 2 points to the symbol-table entry for `initial`.

4. `+` is a lexeme that is mapped into the token $\langle + \rangle$.

5. `rate` is a lexeme that is mapped into the token $\langle \mathbf{id}, 3 \rangle$, where 3 points to the symbol-table entry for `rate`.

6. `*` is a lexeme that is mapped into the token $\langle * \rangle$.

7. `60` is a lexeme that is mapped into the token $\langle 60 \rangle$.[1]

Blanks separating the lexemes would be discarded by the lexical analyzer.

Figure 1.7 shows the representation of the assignment statement (1.1) after lexical analysis as the sequence of tokens

$$\langle \mathbf{id}, 1 \rangle \ \langle = \rangle \ \langle \mathbf{id}, 2 \rangle \ \langle + \rangle \ \langle \mathbf{id}, 3 \rangle \ \langle * \rangle \ \langle 60 \rangle \qquad (1.2)$$

In this representation, the token names $=$, $+$, and $*$ are abstract symbols for the assignment, addition, and multiplication operators, respectively.

[1] Technically speaking, for the lexeme `60` we should make up a token like $\langle \mathbf{number}, 4 \rangle$, where 4 points to the symbol table for the internal representation of integer 60 but we shall defer the discussion of tokens for numbers until Chapter 2. Chapter 3 discusses techniques for building lexical analyzers.

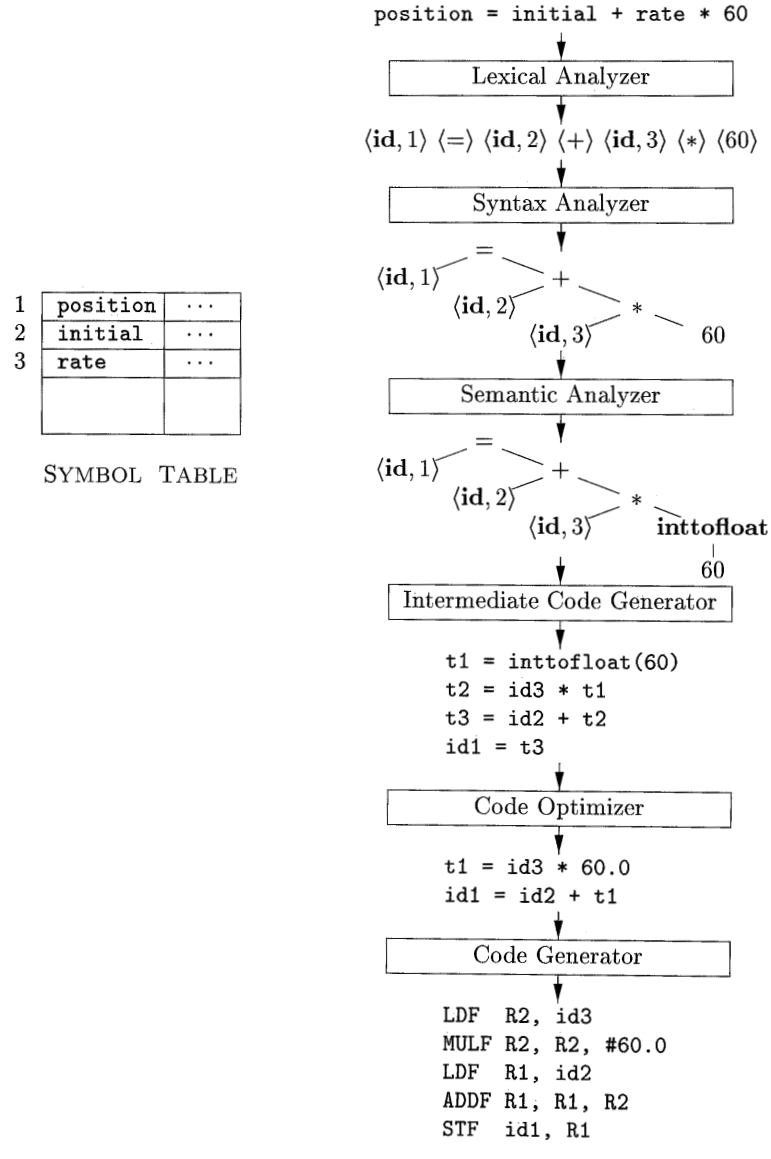


Figure 1.7: Translation of an assignment statement

1.2.2 Syntax Analysis

The second phase of the compiler is *syntax analysis* or *parsing*. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a *syntax tree* in which each interior node represents an operation and the children of the node represent the arguments of the operation. A syntax tree for the token stream (1.2) is shown as the output of the syntactic analyzer in Fig. 1.7.

This tree shows the order in which the operations in the assignment

```
position = initial + rate * 60
```

are to be performed. The tree has an interior node labeled $*$ with $\langle \mathbf{id}, 3 \rangle$ as its left child and the integer 60 as its right child. The node $\langle \mathbf{id}, 3 \rangle$ represents the identifier `rate`. The node labeled $*$ makes it explicit that we must first multiply the value of `rate` by 60. The node labeled $+$ indicates that we must add the result of this multiplication to the value of `initial`. The root of the tree, labeled $=$, indicates that we must store the result of this addition into the location for the identifier `position`. This ordering of operations is consistent with the usual conventions of arithmetic which tell us that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition.

The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program. In Chapter 4 we shall use context-free grammars to specify the grammatical structure of programming languages and discuss algorithms for constructing efficient syntax analyzers automatically from certain classes of grammars. In Chapters 2 and 5 we shall see that syntax-directed definitions can help specify the translation of programming language constructs.

1.2.3 Semantic Analysis

The *semantic analyzer* uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is *type checking*, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

The language specification may permit some type conversions called *coercions*. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

Such a coercion appears in Fig. 1.7. Suppose that `position`, `initial`, and `rate` have been declared to be floating-point numbers, and that the lexeme `60` by itself forms an integer. The type checker in the semantic analyzer in Fig. 1.7 discovers that the operator $*$ is applied to a floating-point number `rate` and an integer 60. In this case, the integer may be converted into a floating-point number. In Fig. 1.7, notice that the output of the semantic analyzer has an extra node for the operator **inttofloat**, which explicitly converts its integer argument into a floating-point number. Type checking and semantic analysis are discussed in Chapter 6.

1.2.4 Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

In Chapter 6, we consider an intermediate form called *three-address code*, which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register. The output of the intermediate code generator in Fig. 1.7 consists of the three-address code sequence

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```
(1.3)

There are several points worth noting about three-address instructions. First, each three-address assignment instruction has at most one operator on the right side. Thus, these instructions fix the order in which operations are to be done; the multiplication precedes the addition in the source program (1.1). Second, the compiler must generate a temporary name to hold the value computed by a three-address instruction. Third, some "three-address instructions" like the first and last in the sequence (1.3), above, have fewer than three operands.

In Chapter 6, we cover the principal intermediate representations used in compilers. Chapters 5 introduces techniques for syntax-directed translation that are applied in Chapter 6 to type checking and intermediate-code generation for typical programming language constructs such as expressions, flow-of-control constructs, and procedure calls.

1.2.5 Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power. For example, a straightforward algorithm generates the intermediate code (1.3), using an instruction for each operator in the tree representation that comes from the semantic analyzer.

A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code. The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the **inttofloat** operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, `t3` is used only once to transmit its value to `id1` so the optimizer can transform (1.3) into the shorter sequence

```
t1 = id3 * 60.0
id1 = id2 + t1
```
(1.4)

There is a great variation in the amount of code optimization different compilers perform. In those that do the most, the so-called "optimizing compilers," a significant amount of time is spent on this phase. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much. The chapters from 8 on discuss machine-independent and machine-dependent optimizations in detail.

1.2.6 Code Generation

The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables.

For example, using registers `R1` and `R2`, the intermediate code in (1.4) might get translated into the machine code

```
LDF  R2,  id3
MULF R2,  R2, #60.0
LDF  R1,  id2
ADDF R1,  R1, R2
STF  id1, R1
```
(1.5)

The first operand of each instruction specifies a destination. The `F` in each instruction tells us that it deals with floating-point numbers. The code in

(1.5) loads the contents of address `id3` into register `R2`, then multiplies it with floating-point constant 60.0. The `#` signifies that 60.0 is to be treated as an immediate constant. The third instruction moves `id2` into register `R1` and the fourth adds to it the value previously computed in register `R2`. Finally, the value in register `R1` is stored into the address of `id1`, so the code correctly implements the assignment statement (1.1). Chapter 8 covers code generation.

This discussion of code generation has ignored the important issue of storage allocation for the identifiers in the source program. As we shall see in Chapter 7, the organization of storage at run-time depends on the language being compiled. Storage-allocation decisions are made either during intermediate code generation or during code generation.

1.2.7 Symbol-Table Management

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly. Symbol tables are discussed in Chapter 2.

1.2.8 The Grouping of Phases into Passes

The discussion of phases deals with the logical organization of a compiler. In an implementation, activities from several phases may be grouped together into a *pass* that reads an input file and writes an output file. For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass. Code optimization might be an optional pass. Then there could be a back-end pass consisting of code generation for a particular target machine.

Some compiler collections have been created around carefully designed intermediate representations that allow the front end for a particular language to interface with the back end for a certain target machine. With these collections, we can produce compilers for different source languages for one target machine by combining different front ends with the back end for that target machine. Similarly, we can produce compilers for different target machines, by combining a front end with back ends for different target machines.

1.2.9 Compiler-Construction Tools

The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, test harnesses, and so on. In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler.

These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms. The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler. Some commonly used compiler-construction tools include

1. *Parser generators* that automatically produce syntax analyzers from a grammatical description of a programming language.

2. *Scanner generators* that produce lexical analyzers from a regular-expression description of the tokens of a language.

3. *Syntax-directed translation engines* that produce collections of routines for walking a parse tree and generating intermediate code.

4. *Code-generator generators* that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

5. *Data-flow analysis engines* that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.

6. *Compiler-construction toolkits* that provide an integrated set of routines for constructing various phases of a compiler.

We shall describe many of these tools throughout this book.

1.3 The Evolution of Programming Languages

The first electronic computers appeared in the 1940's and were programmed in machine language by sequences of 0's and 1's that explicitly told the computer what operations to execute and in what order. The operations themselves were very low level: move data from one location to another, add the contents of two registers, compare two values, and so on. Needless to say, this kind of programming was slow, tedious, and error prone. And once written, the programs were hard to understand and modify.

1.3.1 The Move to Higher-level Languages

The first step towards more people-friendly programming languages was the development of mnemonic assembly languages in the early 1950's. Initially, the instructions in an assembly language were just mnemonic representations of machine instructions. Later, macro instructions were added to assembly languages so that a programmer could define parameterized shorthands for frequently used sequences of machine instructions.

A major step towards higher-level languages was made in the latter half of the 1950's with the development of Fortran for scientific computation, Cobol for business data processing, and Lisp for symbolic computation. The philosophy behind these languages was to create higher-level notations with which programmers could more easily write numerical computations, business applications, and symbolic programs. These languages were so successful that they are still in use today.

In the following decades, many more languages were created with innovative features to help make programming easier, more natural, and more robust. Later in this chapter, we shall discuss some key features that are common to many modern programming languages.

Today, there are thousands of programming languages. They can be classified in a variety of ways. One classification is by generation. *First-generation languages* are the machine languages, *second-generation* the assembly languages, and *third-generation* the higher-level languages like Fortran, Cobol, Lisp, C, C++, C#, and Java. *Fourth-generation languages* are languages designed for specific applications like NOMAD for report generation, SQL for database queries, and Postscript for text formatting. The term *fifth-generation language* has been applied to logic- and constraint-based languages like Prolog and OPS5.

Another classification of languages uses the term *imperative* for languages in which a program specifies *how* a computation is to be done and *declarative* for languages in which a program specifies *what* computation is to be done. Languages such as C, C++, C#, and Java are imperative languages. In imperative languages there is a notion of program state and statements that change the state. Functional languages such as ML and Haskell and constraint logic languages such as Prolog are often considered to be declarative languages.

The term *von Neumann language* is applied to programming languages whose computational model is based on the von Neumann computer architecture. Many of today's languages, such as Fortran and C are von Neumann languages.

An *object-oriented language* is one that supports object-oriented programming, a programming style in which a program consists of a collection of objects that interact with one another. Simula 67 and Smalltalk are the earliest major object-oriented languages. Languages such as C++, C#, Java, and Ruby are more recent object-oriented languages.

Scripting languages are interpreted languages with high-level operators designed for "gluing together" computations. These computations were originally

called "scripts." Awk, JavaScript, Perl, PHP, Python, Ruby, and Tcl are popular examples of scripting languages. Programs written in scripting languages are often much shorter than equivalent programs written in languages like C.

1.3.2 Impacts on Compilers

Since the design of programming languages and compilers are intimately related, the advances in programming languages placed new demands on compiler writers. They had to devise algorithms and representations to translate and support the new language features. Since the 1940's, computer architecture has evolved as well. Not only did the compiler writers have to track new language features, they also had to devise translation algorithms that would take maximal advantage of the new hardware capabilities.

Compilers can help promote the use of high-level languages by minimizing the execution overhead of the programs written in these languages. Compilers are also critical in making high-performance computer architectures effective on users' applications. In fact, the performance of a computer system is so dependent on compiler technology that compilers are used as a tool in evaluating architectural concepts before a computer is built.

Compiler writing is challenging. A compiler by itself is a large program. Moreover, many modern language-processing systems handle several source languages and target machines within the same framework; that is, they serve as collections of compilers, possibly consisting of millions of lines of code. Consequently, good software-engineering techniques are essential for creating and evolving modern language processors.

A compiler must translate correctly the potentially infinite set of programs that could be written in the source language. The problem of generating the optimal target code from a source program is undecidable in general; thus, compiler writers must evaluate tradeoffs about what problems to tackle and what heuristics to use to approach the problem of generating efficient code.

A study of compilers is also a study of how theory meets practice, as we shall see in Section 1.4.

The purpose of this text is to teach the methodology and fundamental ideas used in compiler design. It is not the intention of this text to teach all the algorithms and techniques that could be used for building a state-of-the-art language-processing system. However, readers of this text will acquire the basic knowledge and understanding to learn how to build a compiler relatively easily.

1.3.3 Exercises for Section 1.3

Exercise 1.3.1: Indicate which of the following terms:

a) imperative	b) declarative	c) von Neumann
d) object-oriented	e) functional	f) third-generation
g) fourth-generation	h) scripting	

apply to which of the following languages:

1) C	2) C++	3) Cobol	4) Fortran	5) Java
6) Lisp	7) ML	8) Perl	9) Python	10) VB.

1.4 The Science of Building a Compiler

Compiler design is full of beautiful examples where complicated real-world problems are solved by abstracting the essence of the problem mathematically. These serve as excellent illustrations of how abstractions can be used to solve problems: take a problem, formulate a mathematical abstraction that captures the key characteristics, and solve it using mathematical techniques. The problem formulation must be grounded in a solid understanding of the characteristics of computer programs, and the solution must be validated and refined empirically.

A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled. Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

1.4.1 Modeling in Compiler Design and Implementation

The study of compilers is mainly a study of how we design the right mathematical models and choose the right algorithms, while balancing the need for generality and power against simplicity and efficiency.

Some of most fundamental models are finite-state machines and regular expressions, which we shall meet in Chapter 3. These models are useful for describing the lexical units of programs (keywords, identifiers, and such) and for describing the algorithms used by the compiler to recognize those units. Also among the most fundamental models are context-free grammars, used to describe the syntactic structure of programming languages such as the nesting of parentheses or control constructs. We shall study grammars in Chapter 4. Similarly, trees are an important model for representing the structure of programs and their translation into object code, as we shall see in Chapter 5.

1.4.2 The Science of Code Optimization

The term "optimization" in compiler design refers to the attempts that a compiler makes to produce code that is more efficient than the obvious code. "Optimization" is thus a misnomer, since there is no way that the code produced by a compiler can be guaranteed to be as fast or faster than any other code that performs the same task.

In modern times, the optimization of code that a compiler performs has become both more important and more complex. It is more complex because processor architectures have become more complex, yielding more opportunities to improve the way code executes. It is more important because massively parallel computers require substantial optimization, or their performance suffers by orders of magnitude. With the likely prevalence of multicore machines (computers with chips that have large numbers of processors on them), all compilers will have to face the problem of taking advantage of multiprocessor machines.

It is hard, if not impossible, to build a robust compiler out of "hacks." Thus, an extensive and useful theory has been built up around the problem of optimizing code. The use of a rigorous mathematical foundation allows us to show that an optimization is correct and that it produces the desirable effect for all possible inputs. We shall see, starting in Chapter 9, how models such as graphs, matrices, and linear programs are necessary if the compiler is to produce well optimized code.

On the other hand, pure theory alone is insufficient. Like many real-world problems, there are no perfect answers. In fact, most of the questions that we ask in compiler optimization are undecidable. One of the most important skills in compiler design is the ability to formulate the right problem to solve. We need a good understanding of the behavior of programs to start with and thorough experimentation and evaluation to validate our intuitions.

Compiler optimizations must meet the following design objectives:

- The optimization must be correct, that is, preserve the meaning of the compiled program,
- The optimization must improve the performance of many programs,
- The compilation time must be kept reasonable, and
- The engineering effort required must be manageable.

It is impossible to overemphasize the importance of correctness. It is trivial to write a compiler that generates fast code if the generated code need not be correct! Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free! Thus, the most important objective in writing a compiler is that it is correct.

The second goal is that the compiler must be effective in improving the performance of many input programs. Normally, performance means the speed of the program execution. Especially in embedded applications, we may also wish to minimize the size of the generated code. And in the case of mobile devices, it is also desirable that the code minimizes power consumption. Typically, the same optimizations that speed up execution time also conserve power. Besides performance, usability aspects such as error reporting and debugging are also important.

Third, we need to keep the compilation time short to support a rapid development and debugging cycle. This requirement has become easier to meet as

machines get faster. Often, a program is first developed and debugged without program optimizations. Not only is the compilation time reduced, but more importantly, unoptimized programs are easier to debug, because the optimizations introduced by a compiler often obscure the relationship between the source code and the object code. Turning on optimizations in the compiler sometimes exposes new problems in the source program; thus testing must again be performed on the optimized code. The need for additional testing sometimes deters the use of optimizations in applications, especially if their performance is not critical.

Finally, a compiler is a complex system; we must keep the system simple to assure that the engineering and maintenance costs of the compiler are manageable. There is an infinite number of program optimizations that we could implement, and it takes a nontrivial amount of effort to create a correct and effective optimization. We must prioritize the optimizations, implementing only those that lead to the greatest benefits on source programs encountered in practice.

Thus, in studying compilers, we learn not only how to build a compiler, but also the general methodology of solving complex and open-ended problems. The approach used in compiler development involves both theory and experimentation. We normally start by formulating the problem based on our intuitions on what the important issues are.

1.5 Applications of Compiler Technology

Compiler design is not only about compilers, and many people use the technology learned by studying compilers in school, yet have never, strictly speaking, written (even part of) a compiler for a major programming language. Compiler technology has other important uses as well. Additionally, compiler design impacts several other areas of computer science. In this section, we review the most important interactions and applications of the technology.

1.5.1 Implementation of High-Level Programming Languages

A high-level programming language defines a programming abstraction: the programmer expresses an algorithm using the language, and the compiler must translate that program to the target language. Generally, higher-level programming languages are easier to program in, but are less efficient, that is, the target programs run more slowly. Programmers using a low-level language have more control over a computation and can, in principle, produce more efficient code. Unfortunately, lower-level programs are harder to write and — worse still — less portable, more prone to errors, and harder to maintain. Optimizing compilers include techniques to improve the performance of generated code, thus offsetting the inefficiency introduced by high-level abstractions.

Example 1.2: The **register** keyword in the C programming language is an early example of the interaction between compiler technology and language evolution. When the C language was created in the mid 1970s, it was considered necessary to let a programmer control which program variables reside in registers. This control became unnecessary as effective register-allocation techniques were developed, and most modern programs no longer use this language feature.

In fact, programs that use the **register** keyword may lose efficiency, because programmers often are not the best judge of very low-level matters like register allocation. The optimal choice of register allocation depends greatly on the specifics of a machine architecture. Hardwiring low-level resource-management decisions like register allocation may in fact hurt performance, especially if the program is run on machines other than the one for which it was written. □

The many shifts in the popular choice of programming languages have been in the direction of increased levels of abstraction. C was the predominant systems programming language of the 80's; many of the new projects started in the 90's chose C++; Java, introduced in 1995, gained popularity quickly in the late 90's. The new programming-language features introduced in each round spurred new research in compiler optimization. In the following, we give an overview on the main language features that have stimulated significant advances in compiler technology.

Practically all common programming languages, including C, Fortran and Cobol, support user-defined aggregate data types, such as arrays and structures, and high-level control flow, such as loops and procedure invocations. If we just take each high-level construct or data-access operation and translate it directly to machine code, the result would be very inefficient. A body of compiler optimizations, known as *data-flow optimizations*, has been developed to analyze the flow of data through the program and removes redundancies across these constructs. They are effective in generating code that resembles code written by a skilled programmer at a lower level.

Object orientation was first introduced in Simula in 1967, and has been incorporated in languages such as Smalltalk, C++, C#, and Java. The key ideas behind object orientation are

1. Data abstraction and
2. Inheritance of properties,

both of which have been found to make programs more modular and easier to maintain. Object-oriented programs are different from those written in many other languages, in that they consist of many more, but smaller, procedures (called *methods* in object-oriented terms). Thus, compiler optimizations must be able to perform well across the procedural boundaries of the source program. Procedure inlining, which is the replacement of a procedure call by the body of the procedure, is particularly useful here. Optimizations to speed up virtual method dispatches have also been developed.

Java has many features that make programming easier, many of which have been introduced previously in other languages. The Java language is type-safe; that is, an object cannot be used as an object of an unrelated type. All array accesses are checked to ensure that they lie within the bounds of the array. Java has no pointers and does not allow pointer arithmetic. It has a built-in garbage-collection facility that automatically frees the memory of variables that are no longer in use. While all these features make programming easier, they incur a run-time overhead. Compiler optimizations have been developed to reduce the overhead, for example, by eliminating unnecessary range checks and by allocating objects that are not accessible beyond a procedure on the stack instead of the heap. Effective algorithms also have been developed to minimize the overhead of garbage collection.

In addition, Java is designed to support portable and mobile code. Programs are distributed as Java bytecode, which must either be interpreted or compiled into native code dynamically, that is, at run time. Dynamic compilation has also been studied in other contexts, where information is extracted dynamically at run time and used to produce better-optimized code. In dynamic optimization, it is important to minimize the compilation time as it is part of the execution overhead. A common technique used is to only compile and optimize those parts of the program that will be frequently executed.

1.5.2 Optimizations for Computer Architectures

The rapid evolution of computer architectures has also led to an insatiable demand for new compiler technology. Almost all high-performance systems take advantage of the same two basic techniques: *parallelism* and *memory hierarchies*. Parallelism can be found at several levels: at the *instruction level*, where multiple operations are executed simultaneously and at the *processor level*, where different threads of the same application are run on different processors. Memory hierarchies are a response to the basic limitation that we can build very fast storage or very large storage, but not storage that is both fast and large.

Parallelism

All modern microprocessors exploit instruction-level parallelism. However, this parallelism can be hidden from the programmer. Programs are written as if all instructions were executed in sequence; the hardware dynamically checks for dependencies in the sequential instruction stream and issues them in parallel when possible. In some cases, the machine includes a hardware scheduler that can change the instruction ordering to increase the parallelism in the program. Whether the hardware reorders the instructions or not, compilers can rearrange the instructions to make instruction-level parallelism more effective.

Instruction-level parallelism can also appear explicitly in the instruction set. VLIW (Very Long Instruction Word) machines have instructions that can issue

multiple operations in parallel. The Intel IA64 is a well-known example of such an architecture. All high-performance, general-purpose microprocessors also include instructions that can operate on a vector of data at the same time. Compiler techniques have been developed to generate code automatically for such machines from sequential programs.

Multiprocessors have also become prevalent; even personal computers often have multiple processors. Programmers can write multithreaded code for multiprocessors, or parallel code can be automatically generated by a compiler from conventional sequential programs. Such a compiler hides from the programmers the details of finding parallelism in a program, distributing the computation across the machine, and minimizing synchronization and communication among the processors. Many scientific-computing and engineering applications are computation-intensive and can benefit greatly from parallel processing. Parallelization techniques have been developed to translate automatically sequential scientific programs into multiprocessor code.

Memory Hierarchies

A memory hierarchy consists of several levels of storage with different speeds and sizes, with the level closest to the processor being the fastest but smallest. The average memory-access time of a program is reduced if most of its accesses are satisfied by the faster levels of the hierarchy. Both parallelism and the existence of a memory hierarchy improve the potential performance of a machine, but they must be harnessed effectively by the compiler to deliver real performance on an application.

Memory hierarchies are found in all machines. A processor usually has a small number of registers consisting of hundreds of bytes, several levels of caches containing kilobytes to megabytes, physical memory containing megabytes to gigabytes, and finally secondary storage that contains gigabytes and beyond. Correspondingly, the speed of accesses between adjacent levels of the hierarchy can differ by two or three orders of magnitude. The performance of a system is often limited not by the speed of the processor but by the performance of the memory subsystem. While compilers traditionally focus on optimizing the processor execution, more emphasis is now placed on making the memory hierarchy more effective.

Using registers effectively is probably the single most important problem in optimizing a program. Unlike registers that have to be managed explicitly in software, caches and physical memories are hidden from the instruction set and are managed by hardware. It has been found that cache-management policies implemented by hardware are not effective in some cases, especially in scientific code that has large data structures (arrays, typically). It is possible to improve the effectiveness of the memory hierarchy by changing the layout of the data, or changing the order of instructions accessing the data. We can also change the layout of code to improve the effectiveness of instruction caches.

1.5.3 Design of New Computer Architectures

In the early days of computer architecture design, compilers were developed after the machines were built. That has changed. Since programming in high-level languages is the norm, the performance of a computer system is determined not by its raw speed but also by how well compilers can exploit its features. Thus, in modern computer architecture development, compilers are developed in the processor-design stage, and compiled code, running on simulators, is used to evaluate the proposed architectural features.

RISC

One of the best known examples of how compilers influenced the design of computer architecture was the invention of the RISC (Reduced Instruction-Set Computer) architecture. Prior to this invention, the trend was to develop progressively complex instruction sets intended to make assembly programming easier; these architectures were known as CISC (Complex Instruction-Set Computer). For example, CISC instruction sets include complex memory-addressing modes to support data-structure accesses and procedure-invocation instructions that save registers and pass parameters on the stack.

Compiler optimizations often can reduce these instructions to a small number of simpler operations by eliminating the redundancies across complex instructions. Thus, it is desirable to build simple instruction sets; compilers can use them effectively and the hardware is much easier to optimize.

Most general-purpose processor architectures, including PowerPC, SPARC, MIPS, Alpha, and PA-RISC, are based on the RISC concept. Although the x86 architecture—the most popular microprocessor—has a CISC instruction set, many of the ideas developed for RISC machines are used in the implementation of the processor itself. Moreover, the most effective way to use a high-performance x86 machine is to use just its simple instructions.

Specialized Architectures

Over the last three decades, many architectural concepts have been proposed. They include data flow machines, vector machines, VLIW (Very Long Instruction Word) machines, SIMD (Single Instruction, Multiple Data) arrays of processors, systolic arrays, multiprocessors with shared memory, and multiprocessors with distributed memory. The development of each of these architectural concepts was accompanied by the research and development of corresponding compiler technology.

Some of these ideas have made their way into the designs of embedded machines. Since entire systems can fit on a single chip, processors need no longer be prepackaged commodity units, but can be tailored to achieve better cost-effectiveness for a particular application. Thus, in contrast to general-purpose processors, where economies of scale have led computer architectures

to converge, application-specific processors exhibit a diversity of computer architectures. Compiler technology is needed not only to support programming for these architectures, but also to evaluate proposed architectural designs.

1.5.4 Program Translations

While we normally think of compiling as a translation from a high-level language to the machine level, the same technology can be applied to translate between different kinds of languages. The following are some of the important applications of program-translation techniques.

Binary Translation

Compiler technology can be used to translate the binary code for one machine to that of another, allowing a machine to run programs originally compiled for another instruction set. Binary translation technology has been used by various computer companies to increase the availability of software for their machines. In particular, because of the domination of the x86 personal-computer market, most software titles are available as x86 code. Binary translators have been developed to convert x86 code into both Alpha and Sparc code. Binary translation was also used by Transmeta Inc. in their implementation of the x86 instruction set. Instead of executing the complex x86 instruction set directly in hardware, the Transmeta Crusoe processor is a VLIW processor that relies on binary translation to convert x86 code into native VLIW code.

Binary translation can also be used to provide backward compatibility. When the processor in the Apple Macintosh was changed from the Motorola MC 68040 to the PowerPC in 1994, binary translation was used to allow PowerPC processors run legacy MC 68040 code.

Hardware Synthesis

Not only is most software written in high-level languages; even hardware designs are mostly described in high-level hardware description languages like Verilog and VHDL (Very high-speed integrated circuit Hardware Description Language). Hardware designs are typically described at the register transfer level (RTL), where variables represent registers and expressions represent combinational logic. Hardware-synthesis tools translate RTL descriptions automatically into gates, which are then mapped to transistors and eventually to a physical layout. Unlike compilers for programming languages, these tools often take hours optimizing the circuit. Techniques to translate designs at higher levels, such as the behavior or functional level, also exist.

Database Query Interpreters

Besides specifying software and hardware, languages are useful in many other applications. For example, query languages, especially SQL (Structured Query

Language), are used to search databases. Database queries consist of predicates containing relational and boolean operators. They can be interpreted or compiled into commands to search a database for records satisfying that predicate.

Compiled Simulation

Simulation is a general technique used in many scientific and engineering disciplines to understand a phenomenon or to validate a design. Inputs to a simulator usually include the description of the design and specific input parameters for that particular simulation run. Simulations can be very expensive. We typically need to simulate many possible design alternatives on many different input sets, and each experiment may take days to complete on a high-performance machine. Instead of writing a simulator that interprets the design, it is faster to compile the design to produce machine code that simulates that particular design natively. Compiled simulation can run orders of magnitude faster than an interpreter-based approach. Compiled simulation is used in many state-of-the-art tools that simulate designs written in Verilog or VHDL.

1.5.5 Software Productivity Tools

Programs are arguably the most complicated engineering artifacts ever produced; they consist of many many details, every one of which must be correct before the program will work completely. As a result, errors are rampant in programs; errors may crash a system, produce wrong results, render a system vulnerable to security attacks, or even lead to catastrophic failures in critical systems. Testing is the primary technique for locating errors in programs.

An interesting and promising complementary approach is to use data-flow analysis to locate errors statically (that is, before the program is run). Data-flow analysis can find errors along all the possible execution paths, and not just those exercised by the input data sets, as in the case of program testing. Many of the data-flow-analysis techniques, originally developed for compiler optimizations, can be used to create tools that assist programmers in their software engineering tasks.

The problem of finding all program errors is undecidable. A data-flow analysis may be designed to warn the programmers of all possible statements violating a particular category of errors. But if most of these warnings are false alarms, users will not use the tool. Thus, practical error detectors are often neither sound nor complete. That is, they may not find all the errors in the program, and not all errors reported are guaranteed to be real errors. Nonetheless, various static analyses have been developed and shown to be effective in finding errors, such as dereferencing null or freed pointers, in real programs. The fact that error detectors may be unsound makes them significantly different from compiler optimizations. Optimizers must be conservative and cannot alter the semantics of the program under any circumstances.

In the balance of this section, we shall mention several ways in which program analysis, building upon techniques originally developed to optimize code in compilers, have improved software productivity. Of special importance are techniques that detect statically when a program might have a security vulnerability.

Type Checking

Type checking is an effective and well-established technique to catch inconsistencies in programs. It can be used to catch errors, for example, where an operation is applied to the wrong type of object, or if parameters passed to a procedure do not match the signature of the procedure. Program analysis can go beyond finding type errors by analyzing the flow of data through a program. For example, if a pointer is assigned `null` and then immediately dereferenced, the program is clearly in error.

The same technology can be used to catch a variety of security holes, in which an attacker supplies a string or other data that is used carelessly by the program. A user-supplied string can be labeled with a type "dangerous." If this string is not checked for proper format, then it remains "dangerous," and if a string of this type is able to influence the control-flow of the code at some point in the program, then there is a potential security flaw.

Bounds Checking

It is easier to make mistakes when programming in a lower-level language than a higher-level one. For example, many security breaches in systems are caused by buffer overflows in programs written in C. Because C does not have array-bounds checks, it is up to the user to ensure that the arrays are not accessed out of bounds. Failing to check that the data supplied by the user can overflow a buffer, the program may be tricked into storing user data outside of the buffer. An attacker can manipulate the input data that causes the program to misbehave and compromise the security of the system. Techniques have been developed to find buffer overflows in programs, but with limited success.

Had the program been written in a safe language that includes automatic range checking, this problem would not have occurred. The same data-flow analysis that is used to eliminate redundant range checks can also be used to locate buffer overflows. The major difference, however, is that failing to eliminate a range check would only result in a small run-time cost, while failing to identify a potential buffer overflow may compromise the security of the system. Thus, while it is adequate to use simple techniques to optimize range checks, sophisticated analyses, such as tracking the values of pointers across procedures, are needed to get high-quality results in error detection tools.

Memory-Management Tools

Garbage collection is another excellent example of the tradeoff between efficiency and a combination of ease of programming and software reliability. Automatic memory management obliterates all memory-management errors (e.g., "memory leaks"), which are a major source of problems in C and C++ programs. Various tools have been developed to help programmers find memory management errors. For example, Purify is a widely used tool that dynamically catches memory management errors as they occur. Tools that help identify some of these problems statically have also been developed.

Chapter 3

Lexical Analysis

In this chapter we show how to construct a lexical analyzer. To implement a lexical analyzer by hand, it helps to start with a diagram or other description for the lexemes of each token. We can then write code to identify each occurrence of each lexeme on the input and to return information about the token identified.

We can also produce a lexical analyzer automatically by specifying the lexeme patterns to a *lexical-analyzer generator* and compiling those patterns into code that functions as a lexical analyzer. This approach makes it easier to modify a lexical analyzer, since we have only to rewrite the affected patterns, not the entire program. It also speeds up the process of implementing the lexical analyzer, since the programmer specifies the software at the very high level of patterns and relies on the generator to produce the detailed code. We shall introduce in Section 3.5 a lexical-analyzer generator called *Lex* (or *Flex* in a more recent embodiment).

We begin the study of lexical-analyzer generators by introducing regular expressions, a convenient notation for specifying lexeme patterns. We show how this notation can be transformed, first into nondeterministic automata and then into deterministic automata. The latter two notations can be used as input to a "driver," that is, code which simulates these automata and uses them as a guide to determining the next token. This driver and the specification of the automaton form the nucleus of the lexical analyzer.

3.1 The Role of the Lexical Analyzer

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the

kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

These interactions are suggested in Fig. 3.1. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the *getNextToken* command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

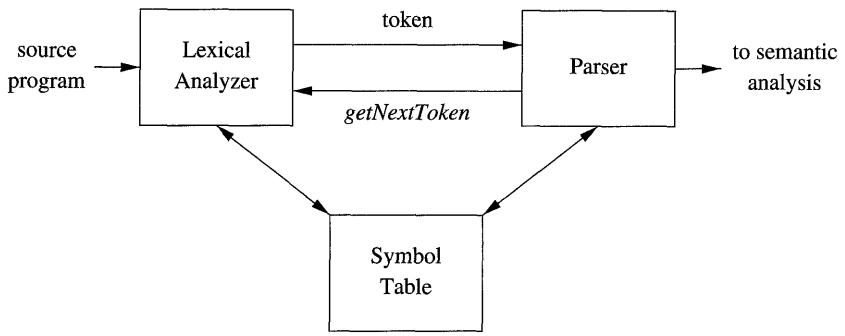


Figure 3.1: Interactions between the lexical analyzer and the parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is stripping out comments and *whitespace* (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input). Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message. In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

Sometimes, lexical analyzers are divided into a cascade of two processes:

a) *Scanning* consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

b) *Lexical analysis* proper is the more complex portion, where the scanner produces the sequence of tokens as output.

3.1.1 Lexical Analysis Versus Parsing

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

1. Simplicity of design is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer. If we are designing a new language, separating lexical and syntactic concerns can lead to a cleaner overall language design.

2. Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

3. Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical analyzer.

3.1.2 Tokens, Patterns, and Lexemes

When discussing lexical analysis, we use three related but distinct terms:

- A *token* is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

- A *pattern* is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is *matched* by many strings.

- A *lexeme* is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Example 3.1: Figure 3.2 gives some typical tokens, their informally described patterns, and some sample lexemes. To see how these concepts are used in practice, in the C statement

```
printf("Total = %d\n", score);
```

both `printf` and `score` are lexemes matching the pattern for token **id**, and `"Total = %d\n"` is a lexeme matching **literal**. □

In many programming languages, the following classes cover most or all of the tokens:

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Figure 3.2: Examples of tokens

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.

2. Tokens for the operators, either individually or in classes such as the token `comparison` mentioned in Fig. 3.2.

3. One token representing all identifiers.

4. One or more tokens representing constants, such as numbers and literal strings.

5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

3.1.3 Attributes for Tokens

When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched. For example, the pattern for token **number** matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program. Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token; the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.

We shall assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information. The most important example is the token **id**, where we need to associate with the token a great deal of information. Normally, information about an identifier — e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) — is kept in the symbol table. Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

Tricky Problems When Recognizing Tokens

Usually, given the pattern describing the lexemes of a token, it is relatively simple to recognize matching lexemes when they occur on the input. However, in some languages it is not immediately apparent when we have seen an instance of a lexeme corresponding to a token. The following example is taken from Fortran, in the fixed-format still allowed in Fortran 90. In the statement

```
DO 5 I = 1.25
```

it is not apparent that the first lexeme is `DO5I`, an instance of the identifier token, until we see the dot following the 1. Note that blanks in fixed-format Fortran are ignored (an archaic convention). Had we seen a comma instead of the dot, we would have had a do-statement

```
DO 5 I = 1,25
```

in which the first lexeme is the keyword `DO`.

Example 3.2: The token names and associated attribute values for the Fortran statement

```
E = M * C ** 2
```

are written below as a sequence of pairs.

<**id**, pointer to symbol-table entry for `E`>
<**assign_op**>
<**id**, pointer to symbol-table entry for `M`>
<**mult_op**>
<**id**, pointer to symbol-table entry for `C`>
<**exp_op**>
<**number**, integer value 2>

Note that in certain pairs, especially operators, punctuation, and keywords, there is no need for an attribute value. In this example, the token **number** has been given an integer-valued attribute. In practice, a typical compiler would instead store a character string representing the constant and use as an attribute value for **number** a pointer to that string. □

3.1.4 Lexical Errors

It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error. For instance, if the string `fi` is encountered for the first time in a C program in the context:

```
fi ( a == f(x)) ...
```

a lexical analyzer cannot tell whether `fi` is a misspelling of the keyword `if` or an undeclared function identifier. Since `fi` is a valid lexeme for the token **id**, the lexical analyzer must return the token **id** to the parser and let some other phase of the compiler — probably the parser in this case — handle an error due to transposition of the letters.

However, suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input. The simplest recovery strategy is "panic mode" recovery. We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

Other possible error-recovery actions are:

1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

Transformations like these may be tried in an attempt to repair the input. The simplest such strategy is to see whether a prefix of the remaining input can be transformed into a valid lexeme by a single transformation. This strategy makes sense, since in practice most lexical errors involve a single character. A more general correction strategy is to find the smallest number of transformations needed to convert the source program into one that consists only of valid lexemes, but this approach is considered too expensive in practice to be worth the effort.

3.1.5 Exercises for Section 3.1

Exercise 3.1.1: Divide the following C++ program:

```
float limitedSquare(x) float x {
    /* returns x-squared, but never more than 100 */
    return (x<=-10.0||x>=10.0)?100:x*x;
}
```

into appropriate lexemes, using the discussion of Section 3.1.2 as a guide. Which lexemes should get associated lexical values? What should those values be?

! **Exercise 3.1.2:** Tagged languages like HTML or XML are different from conventional programming languages in that the punctuation (tags) are either very numerous (as in HTML) or a user-definable set (as in XML). Further, tags can often have parameters. Suggest how to divide the following HTML document:

```
Here is a photo of <B>my house</B>:
<P><IMG SRC = "house.gif"><BR>
See <A HREF = "morePix.html">More Pictures</A> if you
liked that one.<P>
```

into appropriate lexemes. Which lexemes should get associated lexical values, and what should those values be?

Chapter 4

Syntax Analysis

This chapter is devoted to parsing methods that are typically used in compilers. We first present the basic concepts, then techniques suitable for hand implementation, and finally algorithms that have been used in automated tools. Since programs may contain syntactic errors, we discuss extensions of the parsing methods for recovery from common errors.

By design, every programming language has precise rules that prescribe the syntactic structure of well-formed programs. In C, for example, a program is made up of functions, a function out of declarations and statements, a statement out of expressions, and so on. The syntax of programming language constructs can be specified by context-free grammars or BNF (Backus-Naur Form) notation, introduced in Section 2.2. Grammars offer significant benefits for both language designers and compiler writers.

- A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language.

- From certain classes of grammars, we can construct automatically an efficient parser that determines the syntactic structure of a source program. As a side benefit, the parser-construction process can reveal syntactic ambiguities and trouble spots that might have slipped through the initial design phase of a language.

- The structure imparted to a language by a properly designed grammar is useful for translating source programs into correct object code and for detecting errors.

- A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks. These new constructs can be integrated more easily into an implementation that follows the grammatical structure of the language.

4.1 Introduction

In this section, we examine the way the parser fits into a typical compiler. We then look at typical grammars for arithmetic expressions. Grammars for expressions suffice for illustrating the essence of parsing, since parsing techniques for expressions carry over to most programming constructs. This section ends with a discussion of error handling, since the parser must respond gracefully to finding that its input cannot be generated by its grammar.

4.1.1 The Role of the Parser

In our compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in Fig. 4.1, and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing. In fact, the parse tree need not be constructed explicitly, since checking and translation actions can be interspersed with parsing, as we shall see. Thus, the parser and the rest of the front end could well be implemented by a single module.

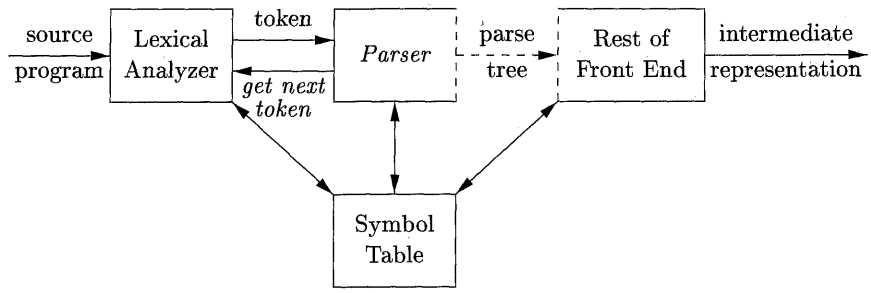


Figure 4.1: Position of parser in compiler model

There are three general types of parsers for grammars: universal, top-down, and bottom-up. Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar (see the bibliographic notes). These general methods are, however, too inefficient to use in production compilers.

The methods commonly used in compilers can be classified as being either top-down or bottom-up. As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves), while bottom-up methods start from the leaves and work their way up to the root. In either case, the input to the parser is scanned from left to right, one symbol at a time.

The most efficient top-down and bottom-up methods work only for subclasses of grammars, but several of these classes, particularly, LL and LR grammars, are expressive enough to describe most of the syntactic constructs in modern programming languages. Parsers implemented by hand often use LL grammars; for example, the predictive-parsing approach of Section 2.4.2 works for LL grammars. Parsers for the larger class of LR grammars are usually constructed using automated tools.

In this chapter, we assume that the output of the parser is some representation of the parse tree for the stream of tokens that comes from the lexical analyzer. In practice, there are a number of tasks that might be conducted during parsing, such as collecting information about various tokens into the symbol table, performing type checking and other kinds of semantic analysis, and generating intermediate code. We have lumped all of these activities into the "rest of the front end" box in Fig. 4.1. These activities will be covered in detail in subsequent chapters.

4.1.2 Representative Grammars

Some of the grammars that will be examined in this chapter are presented here for ease of reference. Constructs that begin with keywords like **while** or **int**, are relatively easy to parse, because the keyword guides the choice of the grammar production that must be applied to match the input. We therefore concentrate on expressions, which present more of challenge, because of the associativity and precedence of operators.

Associativity and precedence are captured in the following grammar, which is similar to ones used in Chapter 2 for describing expressions, terms, and factors. E represents expressions consisting of terms separated by + signs, T represents terms consisting of factors separated by * signs, and F represents factors that can be either parenthesized expressions or identifiers:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array} \tag{4.1}$$

Expression grammar (4.1) belongs to the class of LR grammars that are suitable for bottom-up parsing. This grammar can be adapted to handle additional operators and additional levels of precedence. However, it cannot be used for top-down parsing because it is left recursive.

The following non-left-recursive variant of the expression grammar (4.1) will be used for top-down parsing:

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array} \tag{4.2}$$

The following grammar treats + and * alike, so it is useful for illustrating techniques for handling ambiguities during parsing:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id} \tag{4.3}$$

Here, E represents expressions of all types. Grammar (4.3) permits more than one parse tree for expressions like $a + b * c$.

4.1.3 Syntax Error Handling

The remainder of this section considers the nature of syntactic errors and general strategies for error recovery. Two of these strategies, called panic-mode and phrase-level recovery, are discussed in more detail in connection with specific parsing methods.

If a compiler had to process only correct programs, its design and implementation would be simplified greatly. However, a compiler is expected to assist the programmer in locating and tracking down errors that inevitably creep into programs, despite the programmer's best efforts. Strikingly, few languages have been designed with error handling in mind, even though errors are so commonplace. Our civilization would be radically different if spoken languages had the same requirements for syntactic accuracy as computer languages. Most programming language specifications do not describe how a compiler should respond to errors; error handling is left to the compiler designer. Planning the error handling right from the start can both simplify the structure of a compiler and improve its handling of errors.

Common programming errors can occur at many different levels.

- *Lexical* errors include misspellings of identifiers, keywords, or operators — e.g., the use of an identifier `elipseSize` instead of `ellipseSize` — and missing quotes around text intended as a string.

- *Syntactic* errors include misplaced semicolons or extra or missing braces; that is, "{" or "}." As another example, in C or Java, the appearance of a `case` statement without an enclosing `switch` is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).

- *Semantic* errors include type mismatches between operators and operands. An example is a `return` statement in a Java method with result type `void`.

- *Logical* errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator `=` instead of the comparison operator `==`. The program containing `=` may be well formed; however, it may not reflect the programmer's intent.

The precision of parsing methods allows syntactic errors to be detected very efficiently. Several parsing methods, such as the LL and LR methods, detect

an error as soon as possible; that is, when the stream of tokens from the lexical analyzer cannot be parsed further according to the grammar for the language. More precisely, they have the *viable-prefix property*, meaning that they detect that an error has occurred as soon as they see a prefix of the input that cannot be completed to form a string in the language.

Another reason for emphasizing error recovery during parsing is that many errors appear syntactic, whatever their cause, and are exposed when parsing cannot continue. A few semantic errors, such as type mismatches, can also be detected efficiently; however, accurate detection of semantic and logical errors at compile time is in general a difficult task.

The error handler in a parser has goals that are simple to state but challenging to realize:

- Report the presence of errors clearly and accurately.
- Recover from each error quickly enough to detect subsequent errors.
- Add minimal overhead to the processing of correct programs.

Fortunately, common errors are simple ones, and a relatively straightforward error-handling mechanism often suffices.

How should an error handler report the presence of an error? At the very least, it must report the place in the source program where an error is detected, because there is a good chance that the actual error occurred within the previous few tokens. A common strategy is to print the offending line with a pointer to the position at which an error is detected.

4.1.4 Error-Recovery Strategies

Once an error is detected, how should the parser recover? Although no strategy has proven itself universally acceptable, a few methods have broad applicability. The simplest approach is for the parser to quit with an informative error message when it detects the first error. Additional errors are often uncovered if the parser can restore itself to a state where processing of the input can continue with reasonable hopes that the further processing will provide meaningful diagnostic information. If errors pile up, it is better for the compiler to give up after exceeding some error limit than to produce an annoying avalanche of "spurious" errors.

The balance of this section is devoted to the following recovery strategies: panic-mode, phrase-level, error-productions, and global-correction.

Panic-Mode Recovery

With this method, on discovering an error, the parser discards input symbols one at a time until one of a designated set of *synchronizing tokens* is found. The synchronizing tokens are usually delimiters, such as semicolon or }, whose role in the source program is clear and unambiguous. The compiler designer

must select the synchronizing tokens appropriate for the source language. While panic-mode correction often skips a considerable amount of input without checking it for additional errors, it has the advantage of simplicity, and, unlike some methods to be considered later, is guaranteed not to go into an infinite loop.

Phrase-Level Recovery

On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue. A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon. The choice of the local correction is left to the compiler designer. Of course, we must be careful to choose replacements that do not lead to infinite loops, as would be the case, for example, if we always inserted something on the input ahead of the current input symbol.

Phrase-level replacement has been used in several error-repairing compilers, as it can correct any input string. Its major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

Error Productions

By anticipating common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs. A parser constructed from a grammar augmented by these error productions detects the anticipated errors when an error production is used during parsing. The parser can then generate appropriate error diagnostics about the erroneous construct that has been recognized in the input.

Global Correction

Ideally, we would like a compiler to make as few changes as possible in processing an incorrect input string. There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction. Given an incorrect input string x and grammar G, these algorithms will find a parse tree for a related string y, such that the number of insertions, deletions, and changes of tokens required to transform x into y is as small as possible. Unfortunately, these methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest.

Do note that a closest correct program may not be what the programmer had in mind. Nevertheless, the notion of least-cost correction provides a yardstick for evaluating error-recovery techniques, and has been used for finding optimal replacement strings for phrase-level recovery.

4.5 Bottom-Up Parsing

A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). It is convenient to describe parsing as the process of building parse trees, although a front end may in fact carry out a translation directly without building an explicit tree. The sequence of tree snapshots in Fig. 4.25 illustrates

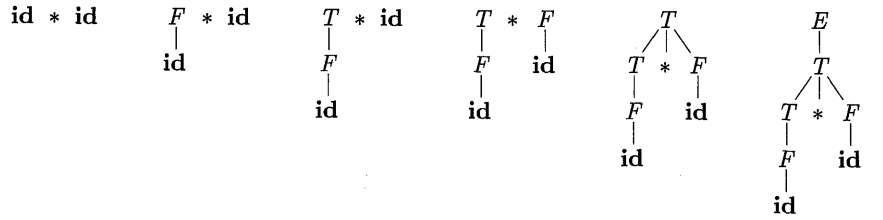


Figure 4.25: A bottom-up parse for **id** * **id**

a bottom-up parse of the token stream **id** * **id**, with respect to the expression grammar (4.1).

This section introduces a general style of bottom-up parsing known as shift-reduce parsing. The largest class of grammars for which shift-reduce parsers can be built, the LR grammars, will be discussed in Sections 4.6 and 4.7. Although it is too much work to build an LR parser by hand, tools called automatic parser generators make it easy to construct efficient LR parsers from suitable grammars. The concepts in this section are helpful for writing suitable grammars to make effective use of an LR parser generator. Algorithms for implementing parser generators appear in Section 4.7.

4.5.1 Reductions

We can think of bottom-up parsing as the process of "reducing" a string w to the start symbol of the grammar. At each *reduction* step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.

The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

Example 4.37: The snapshots in Fig. 4.25 illustrate a sequence of reductions; the grammar is the expression grammar (4.1). The reductions will be discussed in terms of the sequence of strings

$$\mathbf{id} * \mathbf{id}, \quad F * \mathbf{id}, \quad T * \mathbf{id}, \quad T * F, \quad T, \quad E$$

The strings in this sequence are formed from the roots of all the subtrees in the snapshots. The sequence starts with the input string **id***id. The first reduction produces $F * \mathbf{id}$ by reducing the leftmost **id** to F, using the production $F \rightarrow \mathbf{id}$. The second reduction produces $T * \mathbf{id}$ by reducing F to T.

Now, we have a choice between reducing the string T, which is the body of $E \rightarrow T$, and the string consisting of the second **id**, which is the body of $F \rightarrow \mathbf{id}$. Rather than reduce T to E, the second **id** is reduced to T, resulting in the string $T * F$. This string then reduces to T. The parse completes with the reduction of T to the start symbol E. □

By definition, a reduction is the reverse of a step in a derivation (recall that in a derivation, a nonterminal in a sentential form is replaced by the body of one of its productions). The goal of bottom-up parsing is therefore to construct a derivation in reverse. The following derivation corresponds to the parse in Fig. 4.25:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

This derivation is in fact a rightmost derivation.

4.5.2 Handle Pruning

Bottom-up parsing during a left-to-right scan of the input constructs a rightmost derivation in reverse. Informally, a "handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

For example, adding subscripts to the tokens **id** for clarity, the handles during the parse of $\mathbf{id}_1 * \mathbf{id}_2$ according to the expression grammar (4.1) are as in Fig. 4.26. Although T is the body of the production $E \rightarrow T$, the symbol T is not a handle in the sentential form $T * \mathbf{id}_2$. If T were indeed replaced by E, we would get the string $E * \mathbf{id}_2$, which cannot be derived from the start symbol E. Thus, the leftmost substring that matches the body of some production need not be a handle.

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

Figure 4.26: Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

Formally, if $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$, as in Fig. 4.27, then production $A \rightarrow \beta$ in the position following α is a *handle* of $\alpha\beta w$. Alternatively, a handle of a right-sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found, such that replacing β at that position by A produces the previous right-sentential form in a rightmost derivation of γ.

Notice that the string w to the right of the handle must contain only terminal symbols. For convenience, we refer to the body β rather than $A \rightarrow \beta$ as a handle. Note we say "a handle" rather than "the handle," because the grammar could be ambiguous, with more than one rightmost derivation of $\alpha\beta w$. If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

A rightmost derivation in reverse can be obtained by "handle pruning." That is, we start with a string of terminals w to be parsed. If w is a sentence

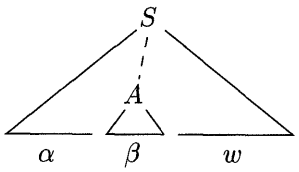


Figure 4.27: A handle $A \to \beta$ in the parse tree for $\alpha\beta w$

of the grammar at hand, then let $w = \gamma_n$, where γ_n is the nth right-sentential form of some as yet unknown rightmost derivation

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

To reconstruct this derivation in reverse order, we locate the handle β_n in γ_n and replace β_n by the head of the relevant production $A_n \to \beta_n$ to obtain the previous right-sentential form γ_{n-1}. Note that we do not yet know how handles are to be found, but we shall see methods of doing so shortly.

We then repeat this process. That is, we locate the handle β_{n-1} in γ_{n-1} and reduce this handle to obtain the right-sentential form γ_{n-2}. If by continuing this process we produce a right-sentential form consisting only of the start symbol S, then we halt and announce successful completion of parsing. The reverse of the sequence of productions used in the reductions is a rightmost derivation for the input string.

4.5.3 Shift-Reduce Parsing

Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. As we shall see, the handle always appears at the top of the stack just before it is identified as the handle.

We use \$ to mark the bottom of the stack and also the right end of the input. Conventionally, when discussing bottom-up parsing, we show the top of the stack on the right, rather than on the left as we did for top-down parsing. Initially, the stack is empty, and the string w is on the input, as follows:

STACK	INPUT
\$	w \$

During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack. It then reduces β to the head of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

STACK	INPUT
$\$ S$	$\$$

Upon entering this configuration, the parser halts and announces successful completion of parsing. Figure 4.28 steps through the actions a shift-reduce parser might take in parsing the input string $\mathbf{id}_1 * \mathbf{id}_2$ according to the expression grammar (4.1).

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \rightarrow \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift
$\$ T * \mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$ T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$ T$	$\$$	reduce by $E \rightarrow T$
$\$ E$	$\$$	accept

Figure 4.28: Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make: (1) shift, (2) reduce, (3) accept, and (4) error.

1. *Shift.* Shift the next input symbol onto the top of the stack.
2. *Reduce.* The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. *Accept.* Announce successful completion of parsing.
4. *Error.* Discover a syntax error and call an error recovery routine.

The use of a stack in shift-reduce parsing is justified by an important fact: the handle will always eventually appear on top of the stack, never inside. This fact can be shown by considering the possible forms of two successive steps in any rightmost derivation. Figure 4.29 illustrates the two possible cases. In case (1), A is replaced by βBy, and then the rightmost nonterminal B in the body βBy is replaced by γ. In case (2), A is again expanded first, but this time the body is a string y of terminals only. The next rightmost nonterminal B will be somewhere to the left of y.

In other words:

$$(1) \quad S \underset{rm}{\overset{*}{\Rightarrow}} \alpha Az \underset{rm}{\Rightarrow} \alpha\beta Byz \underset{rm}{\Rightarrow} \alpha\beta\gamma yz$$

$$(2) \quad S \underset{rm}{\overset{*}{\Rightarrow}} \alpha Bx Az \underset{rm}{\Rightarrow} \alpha Bxyz \underset{rm}{\Rightarrow} \alpha\gamma xyz$$

Figure 4.29: Cases for two successive steps of a rightmost derivation

Consider case (1) in reverse, where a shift-reduce parser has just reached the configuration

STACK	INPUT
$\$\alpha\beta\gamma$	$yz\$$

The parser reduces the handle γ to B to reach the configuration

$\$\alpha\beta B$	$yz\$$

The parser can now shift the string y onto the stack by a sequence of zero or more shift moves to reach the configuration

$\$\alpha\beta By$	$z\$$

with the handle βBy on top of the stack, and it gets reduced to A.

Now consider case (2). In configuration

$\$\alpha\gamma$	$xyz\$$

the handle γ is on top of the stack. After reducing the handle γ to B, the parser can shift the string xy to get the next handle y on top of the stack, ready to be reduced to A:

$\$\alpha Bxy$	$z\$$

In both cases, after making a reduction the parser had to shift zero or more symbols to get the next handle onto the stack. It never had to go into the stack to find the handle.

4.5.4 Conflicts During Shift-Reduce Parsing

There are context-free grammars for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next input symbol, cannot decide whether to shift or to reduce (a *shift/reduce conflict*), or cannot decide

which of several reductions to make (a *reduce/reduce conflict*). We now give some examples of syntactic constructs that give rise to such grammars. Technically, these grammars are not in the LR(k) class of grammars defined in Section 4.7; we refer to them as non-LR grammars. The k in LR(k) refers to the number of symbols of lookahead on the input. Grammars used in compiling usually fall in the LR(1) class, with one symbol of lookahead at most.

Example 4.38: An ambiguous grammar can never be LR. For example, consider the dangling-else grammar (4.14) of Section 4.3:

$$\begin{array}{lcl} \textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\ & | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\ & | & \textbf{other} \end{array}$$

If we have a shift-reduce parser in configuration

STACK	INPUT
$\cdots$ **if** *expr* **then** *stmt*	**else** $\cdots$ \$

we cannot tell whether **if** *expr* **then** *stmt* is the handle, no matter what appears below it on the stack. Here there is a shift/reduce conflict. Depending on what follows the **else** on the input, it might be correct to reduce **if** *expr* **then** *stmt* to *stmt*, or it might be correct to shift **else** and then to look for another *stmt* to complete the alternative **if** *expr* **then** *stmt* **else** *stmt*.

Note that shift-reduce parsing can be adapted to parse certain ambiguous grammars, such as the if-then-else grammar above. If we resolve the shift/reduce conflict on **else** in favor of shifting, the parser will behave as we expect, associating each **else** with the previous unmatched **then**. We discuss parsers for such ambiguous grammars in Section 4.8. □

Another common setting for conflicts occurs when we know we have a handle, but the stack contents and the next input symbol are insufficient to determine which production should be used in a reduction. The next example illustrates this situation.

Example 4.39: Suppose we have a lexical analyzer that returns the token name **id** for all names, regardless of their type. Suppose also that our language invokes procedures by giving their names, with parameters surrounded by parentheses, and that arrays are referenced by the same syntax. Since the translation of indices in array references and parameters in procedure calls are different, we want to use different productions to generate lists of actual parameters and indices. Our grammar might therefore have (among others) productions such as those in Fig. 4.30.

A statement beginning with `p(i,j)` would appear as the token stream **id**(**id**, **id**) to the parser. After shifting the first three tokens onto the stack, a shift-reduce parser would be in configuration

(1)	*stmt*	$\rightarrow$	**id** (*parameter_list*)
(2)	*stmt*	$\rightarrow$	*expr* := *expr*
(3)	*parameter_list*	$\rightarrow$	*parameter_list* , *parameter*
(4)	*parameter_list*	$\rightarrow$	*parameter*
(5)	*parameter*	$\rightarrow$	**id**
(6)	*expr*	$\rightarrow$	**id** (*expr_list*)
(7)	*expr*	$\rightarrow$	**id**
(8)	*expr_list*	$\rightarrow$	*expr_list* , *expr*
(9)	*expr_list*	$\rightarrow$	*expr*

Figure 4.30: Productions involving procedure calls and array references

STACK	INPUT
$\cdots$ **id (id**	**, id)** $\cdots$

It is evident that the **id** on top of the stack must be reduced, but by which production? The correct choice is production (5) if `p` is a procedure, but production (7) if `p` is an array. The stack does not tell which; information in the symbol table obtained from the declaration of `p` must be used.

One solution is to change the token **id** in production (1) to **procid** and to use a more sophisticated lexical analyzer that returns the token name **procid** when it recognizes a lexeme that is the name of a procedure. Doing so would require the lexical analyzer to consult the symbol table before returning a token.

If we made this modification, then on processing `p(i,j)` the parser would be either in the configuration

STACK	INPUT
$\cdots$ **procid (id**	**, id)** $\cdots$

or in the configuration above. In the former case, we choose reduction by production (5); in the latter case by production (7). Notice how the symbol third from the top of the stack determines the reduction to be made, even though it is not involved in the reduction. Shift-reduce parsing can utilize information far down in the stack to guide the parse. □

4.5.5 Exercises for Section 4.5

Exercise 4.5.1: For the grammar $S \rightarrow 0\ S\ 1 \mid 0\ 1$ of Exercise 4.2.2(a), indicate the handle in each of the following right-sentential forms:

a) 000111.

b) $00S11$.

Exercise 4.5.2: Repeat Exercise 4.5.1 for the grammar $S \rightarrow S\ S + \mid S\ S * \mid a$ of Exercise 4.2.1 and the following right-sentential forms:

a) $SSS + a * +$.

b) $SS + a * a +$.

c) $aaa * a + +$.

Exercise 4.5.3: Give bottom-up parses for the following input strings and grammars:

a) The input 000111 according to the grammar of Exercise 4.5.1.

b) The input $aaa * a + +$ according to the grammar of Exercise 4.5.2.

4.6 Introduction to LR Parsing: Simple LR

The most prevalent type of bottom-up parser today is based on a concept called LR(k) parsing; the "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions. The cases $k = 0$ or $k = 1$ are of practical interest, and we shall only consider LR parsers with $k \leq 1$ here. When (k) is omitted, k is assumed to be 1.

This section introduces the basic concepts of LR parsing and the easiest method for constructing shift-reduce parsers, called "simple LR" (or SLR, for short). Some familiarity with the basic concepts is helpful even if the LR parser itself is constructed using an automatic parser generator. We begin with "items" and "parser states;" the diagnostic output from an LR parser generator typically includes parser states, which can be used to isolate the sources of parsing conflicts.

Section 4.7 introduces two, more complex methods — canonical-LR and LALR — that are used in the majority of LR parsers.

4.6.1 Why LR Parsers?

LR parsers are table-driven, much like the nonrecursive LL parsers of Section 4.4.4. A grammar for which we can construct a parsing table using one of the methods in this section and the next is said to be an *LR grammar*. Intuitively, for a grammar to be LR it is sufficient that a left-to-right shift-reduce parser be able to recognize handles of right-sentential forms when they appear on top of the stack.

LR parsing is attractive for a variety of reasons:

- LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written. Non-LR context-free grammars exist, but these can generally be avoided for typical programming-language constructs.

- The LR-parsing method is the most general nonbacktracking shift-reduce parsing method known, yet it can be implemented as efficiently as other, more primitive shift-reduce methods (see the bibliographic notes).
- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.
- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive or LL methods. For a grammar to be LR(k), we must be able to recognize the occurrence of the right side of a production in a right-sentential form, with k input symbols of lookahead. This requirement is far less stringent than that for LL(k) grammars where we must be able to recognize the use of a production seeing only the first k symbols of what its right side derives. Thus, it should not be surprising that LR grammars can describe more languages than LL grammars.

The principal drawback of the LR method is that it is too much work to construct an LR parser by hand for a typical programming-language grammar. A specialized tool, an LR parser generator, is needed. Fortunately, many such generators are available, and we shall discuss one of the most commonly used ones, `Yacc`, in Section 4.9. Such a generator takes a context-free grammar and automatically produces a parser for that grammar. If the grammar contains ambiguities or other constructs that are difficult to parse in a left-to-right scan of the input, then the parser generator locates these constructs and provides detailed diagnostic messages.

4.6.2 Items and the LR(0) Automaton

How does a shift-reduce parser know when to shift and when to reduce? For example, with stack contents $\$T$ and next input symbol $*$ in Fig. 4.28, how does the parser know that T on the top of the stack is not a handle, so the appropriate action is to shift and not to reduce T to E?

An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse. States represent sets of "items." An *LR(0) item* (*item* for short) of a grammar G is a production of G with a dot at some position of the body. Thus, production $A \rightarrow XYZ$ yields the four items

$$
\begin{aligned}
A &\rightarrow \cdot XYZ \\
A &\rightarrow X \cdot YZ \\
A &\rightarrow XY \cdot Z \\
A &\rightarrow XYZ \cdot
\end{aligned}
$$

The production $A \rightarrow \epsilon$ generates only one item, $A \rightarrow \cdot$.

Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process. For example, the item $A \rightarrow \cdot XYZ$ indicates that we hope to see a string derivable from XYZ next on the input. Item

Representing Item Sets

A parser generator that produces a bottom-up parser may need to represent items and sets of items conveniently. Note that an item can be represented by a pair of integers, the first of which is the number of one of the productions of the underlying grammar, and the second of which is the position of the dot. Sets of items can be represented by a list of these pairs. However, as we shall see, the necessary sets of items often include "closure" items, where the dot is at the beginning of the body. These can always be reconstructed from the other items in the set, and we do not have to include them in the list.

$A \to X \cdot YZ$ indicates that we have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ. Item $A \to XYZ \cdot$ indicates that we have seen the body XYZ and that it may be time to reduce XYZ to A.

One collection of sets of LR(0) items, called the *canonical* LR(0) collection, provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions. Such an automaton is called an *LR(0) automaton.*[3] In particular, each state of the LR(0) automaton represents a set of items in the canonical LR(0) collection. The automaton for the expression grammar (4.1), shown in Fig. 4.31, will serve as the running example for discussing the canonical LR(0) collection for a grammar.

To construct the canonical LR(0) collection for a grammar, we define an augmented grammar and two functions, CLOSURE and GOTO. If G is a grammar with start symbol S, then G', the *augmented grammar* for G, is G with a new start symbol S' and production $S' \to S$. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input. That is, acceptance occurs when and only when the parser is about to reduce by $S' \to S$.

Closure of Item Sets

If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by the two rules:

1. Initially, add every item in I to CLOSURE(I).
2. If $A \to \alpha \cdot B\beta$ is in CLOSURE(I) and $B \to \gamma$ is a production, then add the item $B \to \cdot\gamma$ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

[3]Technically, the automaton misses being deterministic according to the definition of Section 3.6.4, because we do not have a dead state, corresponding to the empty set of items. As a result, there are some state-input pairs for which no next state exists.

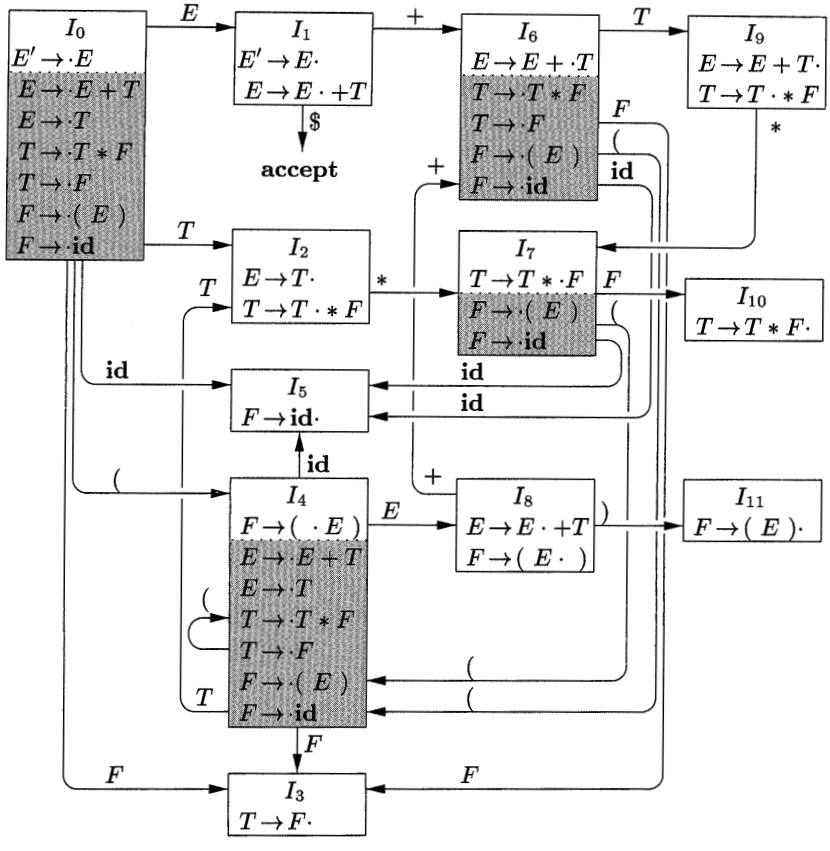


Figure 4.31: LR(0) automaton for the expression grammar (4.1)

Intuitively, $A \to \alpha \cdot B\beta$ in CLOSURE(I) indicates that, at some point in the parsing process, we think we might next see a substring derivable from $B\beta$ as input. The substring derivable from $B\beta$ will have a prefix derivable from B by applying one of the B-productions. We therefore add items for all the B-productions; that is, if $B \to \gamma$ is a production, we also include $B \to \cdot\gamma$ in CLOSURE(I).

Example 4.40: Consider the augmented expression grammar:

$$\begin{array}{lcl} E' & \to & E \\ E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ E & \to & (E) \mid \mathbf{id} \end{array}$$

If I is the set of one item $\{[E' \to \cdot E]\}$, then CLOSURE(I) contains the set of items I_0 in Fig. 4.31.

To see how the closure is computed, $E' \rightarrow \cdot E$ is put in CLOSURE(I) by rule (1). Since there is an E immediately to the right of a dot, we add the E-productions with dots at the left ends: $E \rightarrow \cdot E + T$ and $E \rightarrow \cdot T$. Now there is a T immediately to the right of a dot in the latter item, so we add $T \rightarrow \cdot T * F$ and $T \rightarrow \cdot F$. Next, the F to the right of a dot forces us to add $F \rightarrow \cdot(E)$ and $F \rightarrow \cdot$**id**, but no other items need to be added. □

The closure can be computed as in Fig. 4.32. A convenient way to implement the function *closure* is to keep a boolean array *added*, indexed by the nonterminals of G, such that *added*[B] is set to **true** if and when we add the item $B \rightarrow \cdot\gamma$ for each B-production $B \rightarrow \gamma$.

```
SetOfItems CLOSURE(I) {
        J = I;
        repeat
                for ( each item A → α·Bβ in J )
                        for ( each production B → γ of G )
                                if ( B → ·γ is not in J )
                                        add B → ·γ to J;
        until no more items are added to J on one round;
        return J;
}
```

Figure 4.32: Computation of CLOSURE

Note that if one B-production is added to the closure of I with the dot at the left end, then all B-productions will be similarly added to the closure. Hence, it is not necessary in some circumstances actually to list the items $B \rightarrow \cdot\gamma$ added to I by CLOSURE. A list of the nonterminals B whose productions were so added will suffice. We divide all the sets of items of interest into two classes:

1. *Kernel items*: the initial item, $S' \rightarrow \cdot S$, and all items whose dots are not at the left end.

2. *Nonkernel items*: all items with their dots at the left end, except for $S' \rightarrow \cdot S$.

Moreover, each set of items of interest is formed by taking the closure of a set of kernel items; the items added in the closure can never be kernel items, of course. Thus, we can represent the sets of items we are really interested in with very little storage if we throw away all nonkernel items, knowing that they could be regenerated by the closure process. In Fig. 4.31, nonkernel items are in the shaded part of the box for a state.

The Function GOTO

The second useful function is GOTO(I, X) where I is a set of items and X is a grammar symbol. GOTO(I, X) is defined to be the closure of the set of all items $[A \to \alpha X \cdot \beta]$ such that $[A \to \alpha \cdot X\beta]$ is in I. Intuitively, the GOTO function is used to define the transitions in the LR(0) automaton for a grammar. The states of the automaton correspond to sets of items, and GOTO(I, X) specifies the transition from the state for I under input X.

Example 4.41: If I is the set of two items $\{[E' \to E\cdot],\ [E \to E\cdot + T]\}$, then GOTO$(I, +)$ contains the items

$$
\begin{aligned}
E &\to E + \cdot T \\
T &\to \cdot T * F \\
T &\to \cdot F \\
F &\to \cdot(E) \\
F &\to \cdot \mathbf{id}
\end{aligned}
$$

We computed GOTO$(I, +)$ by examining I for items with $+$ immediately to the right of the dot. $E' \to E\cdot$ is not such an item, but $E \to E\cdot + T$ is. We moved the dot over the $+$ to get $E \to E + \cdot T$ and then took the closure of this singleton set. □

We are now ready for the algorithm to construct C, the canonical collection of sets of LR(0) items for an augmented grammar G' — the algorithm is shown in Fig. 4.33.

```
void items(G') {
    C = CLOSURE({[S' → ·S]});
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C on a round;
}
```

Figure 4.33: Computation of the canonical collection of sets of LR(0) items

Example 4.42: The canonical collection of sets of LR(0) items for grammar (4.1) and the GOTO function are shown in Fig. 4.31. GOTO is encoded by the transitions in the figure. □

Use of the LR(0) Automaton

The central idea behind "Simple LR," or SLR, parsing is the construction from the grammar of the LR(0) automaton. The states of this automaton are the sets of items from the canonical LR(0) collection, and the transitions are given by the GOTO function. The LR(0) automaton for the expression grammar (4.1) appeared earlier in Fig. 4.31.

The start state of the LR(0) automaton is CLOSURE($\{[S' \rightarrow \cdot S]\}$), where S' is the start symbol of the augmented grammar. All states are accepting states. We say "state j" to refer to the state corresponding to the set of items I_j.

How can LR(0) automata help with shift-reduce decisions? Shift-reduce decisions can be made as follows. Suppose that the string γ of grammar symbols takes the LR(0) automaton from the start state 0 to some state j. Then, shift on next input symbol a if state j has a transition on a. Otherwise, we choose to reduce; the items in state j will tell us which production to use.

The LR-parsing algorithm to be introduced in Section 4.6.3 uses its stack to keep track of states as well as grammar symbols; in fact, the grammar symbol can be recovered from the state, so the stack holds states. The next example gives a preview of how an LR(0) automaton and a stack of states can be used to make shift-reduce parsing decisions.

Example 4.43: Figure 4.34 illustrates the actions of a shift-reduce parser on input **id** $*$ **id**, using the LR(0) automaton in Fig. 4.31. We use a stack to hold states; for clarity, the grammar symbols corresponding to the states on the stack appear in column SYMBOLS. At line (1), the stack holds the start state 0 of the automaton; the corresponding symbol is the bottom-of-stack marker \$.

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	**id** $*$ **id** \$	shift to 5
(2)	0 5	\$ **id**	$*$ **id** \$	reduce by $F \rightarrow$ **id**
(3)	0 3	\$ F	$*$ **id** \$	reduce by $T \rightarrow F$
(4)	0 2	\$ T	$*$ **id** \$	shift to 7
(5)	0 2 7	\$ $T*$	**id** \$	shift to 5
(6)	0 2 7 5	\$ $T*$ **id**	\$	reduce by $F \rightarrow$ **id**
(7)	0 2 7 10	\$ $T*F$	\$	reduce by $T \rightarrow T*F$
(8)	0 2	\$ T	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ E	\$	accept

Figure 4.34: The parse of **id** $*$ **id**

The next input symbol is **id** and state 0 has a transition on **id** to state 5. We therefore shift. At line (2), state 5 (symbol **id**) has been pushed onto the stack. There is no transition from state 5 on input $*$, so we reduce. From item $[F \rightarrow$ **id**$\cdot]$ in state 5, the reduction is by production $F \rightarrow$ **id**.

With symbols, a reduction is implemented by popping the body of the production from the stack (on line (2), the body is **id**) and pushing the head of the production (in this case, F). With states, we pop state 5 for symbol **id**, which brings state 0 to the top and look for a transition on F, the head of the production. In Fig. 4.31, state 0 has a transition on F to state 3, so we push state 3, with corresponding symbol F; see line (3).

As another example, consider line (5), with state 7 (symbol $*$) on top of the stack. This state has a transition to state 5 on input **id**, so we push state 5 (symbol **id**). State 5 has no transitions, so we reduce by $F \rightarrow$ **id**. When we pop state 5 for the body **id**, state 7 comes to the top of the stack. Since state 7 has a transition on F to state 10, we push state 10 (symbol F). $\square$

4.4.2 FIRST and FOLLOW

The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G. During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol. During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

Define *FIRST*(α), where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α. If $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then ϵ is also in FIRST(α). For example, in Fig. 4.15, $A \stackrel{*}{\Rightarrow} c\gamma$, so c is in FIRST(A).

For a preview of how FIRST can be used during predictive parsing, consider two A-productions $A \rightarrow \alpha \mid \beta$, where FIRST($\alpha$) and FIRST($\beta$) are disjoint sets. We can then choose between these A-productions by looking at the next input

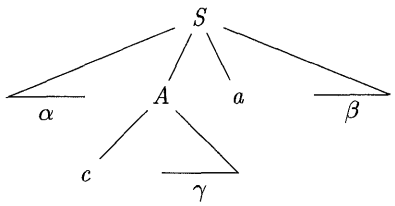


Figure 4.15: Terminal c is in FIRST(A) and a is in FOLLOW(A)

symbol a, since a can be in at most one of FIRST(α) and FIRST(β), not both. For instance, if a is in FIRST(β) choose the production $A \to \beta$. This idea will be explored when LL(1) grammars are defined in Section 4.4.3.

Define *FOLLOW*(A), for nonterminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form; that is, the set of terminals a such that there exists a derivation of the form $S \stackrel{*}{\Rightarrow} \alpha A a \beta$, for some α and β, as in Fig. 4.15. Note that there may have been symbols between A and a, at some time during the derivation, but if so, they derived ϵ and disappeared. In addition, if A can be the rightmost symbol in some sentential form, then \$ is in FOLLOW(A); recall that \$ is a special "endmarker" symbol that is assumed not to be a symbol of any grammar.

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) = $\{X\}$.

2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), ... , FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$), and so on.

3. If $X \to \epsilon$ is a production, then add ϵ to FIRST(X).

Now, we can compute FIRST for any string $X_1 X_2 \cdots X_n$ as follows. Add to FIRST($X_1 X_2 \cdots X_n$) all non-ϵ symbols of FIRST(X_1). Also add the non-ϵ symbols of FIRST(X_2), if ϵ is in FIRST(X_1); the non-ϵ symbols of FIRST(X_3), if ϵ is in FIRST(X_1) and FIRST(X_2); and so on. Finally, add ϵ to FIRST($X_1 X_2 \cdots X_n$) if, for all i, ϵ is in FIRST(X_i).

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.

2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).

3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Example 4.30: Consider again the non-left-recursive grammar (4.28). Then:

1. FIRST(F) = FIRST(T) = FIRST(E) = $\{(, \mathbf{id}\}$. To see why, note that the two productions for F have bodies that start with these two terminal symbols, **id** and the left parenthesis. T has only one production, and its body starts with F. Since F does not derive ϵ, FIRST(T) must be the same as FIRST(F). The same argument covers FIRST(E).

2. FIRST(E') = $\{+, \epsilon\}$. The reason is that one of the two productions for E' has a body that begins with terminal $+$, and the other's body is ϵ. Whenever a nonterminal derives ϵ, we place ϵ in FIRST for that nonterminal.

3. FIRST(T') = $\{*, \epsilon\}$. The reasoning is analogous to that for FIRST(E').

4. FOLLOW(E) = FOLLOW(E') = $\{), \$\}$. Since E is the start symbol, FOLLOW(E) must contain \$. The production body (E) explains why the right parenthesis is in FOLLOW(E). For E', note that this nonterminal appears only at the ends of bodies of E-productions. Thus, FOLLOW(E') must be the same as FOLLOW(E).

5. FOLLOW(T) = FOLLOW(T') = $\{+,), \$\}$. Notice that T appears in bodies only followed by E'. Thus, everything except ϵ that is in FIRST(E') must be in FOLLOW(T); that explains the symbol $+$. However, since FIRST(E') contains ϵ (i.e., $E' \stackrel{*}{\Rightarrow} \epsilon$), and E' is the entire string following T in the bodies of the E-productions, everything in FOLLOW(E) must also be in FOLLOW(T). That explains the symbols \$ and the right parenthesis. As for T', since it appears only at the ends of the T-productions, it must be that FOLLOW(T') = FOLLOW(T).

6. FOLLOW(F) = $\{+, *,), \$\}$. The reasoning is analogous to that for T in point (5).

□

4.6.3 The LR-Parsing Algorithm

A schematic of an LR parser is shown in Fig. 4.35. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (ACTION and GOTO). The driver program is the same for all LR parsers; only the parsing table changes from one parser to another. The parsing program reads characters from an input buffer one at a time. Where a shift-reduce parser would shift a symbol, an LR parser shifts a *state*. Each state summarizes the information contained in the stack below it.

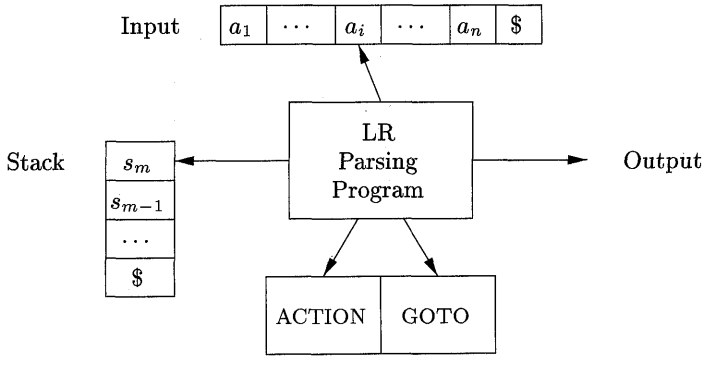


Figure 4.35: Model of an LR parser

The stack holds a sequence of states, $s_0 s_1 \cdots s_m$, where s_m is on top. In the SLR method, the stack holds states from the LR(0) automaton; the canonical-LR and LALR methods are similar. By construction, each state has a corresponding grammar symbol. Recall that states correspond to sets of items, and that there is a transition from state i to state j if GOTO$(I_i, X) = I_j$. All transitions to state j must be for the same grammar symbol X. Thus, each state, except the start state 0, has a unique grammar symbol associated with it.[4]

[4]The converse need not hold; that is, more than one state may have the same grammar

Structure of the LR Parsing Table

The parsing table consists of two parts: a parsing-action function ACTION and a goto function GOTO.

1. The ACTION function takes as arguments a state i and a terminal a (or \$, the input endmarker). The value of ACTION$[i, a]$ can have one of four forms:
   (a) Shift j, where j is a state. The action taken by the parser effectively shifts input a to the stack, but uses state j to represent a.
   (b) Reduce $A \to \beta$. The action of the parser effectively reduces β on the top of the stack to head A.
   (c) Accept. The parser accepts the input and finishes parsing.
   (d) Error. The parser discovers an error in its input and takes some corrective action. We shall have more to say about how such error-recovery routines work in Sections 4.8.3 and 4.9.4.
2. We extend the GOTO function, defined on sets of items, to states: if GOTO$[I_i, A] = I_j$, then GOTO also maps a state i and a nonterminal A to state j.

4.6.3 The LR-Parsing Algorithm

A schematic of an LR parser is shown in Fig. 4.35. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (ACTION and GOTO). The driver program is the same for all LR parsers; only the parsing table changes from one parser to another. The parsing program reads characters from an input buffer one at a time. Where a shift-reduce parser would shift a symbol, an LR parser shifts a *state*. Each state summarizes the information contained in the stack below it.

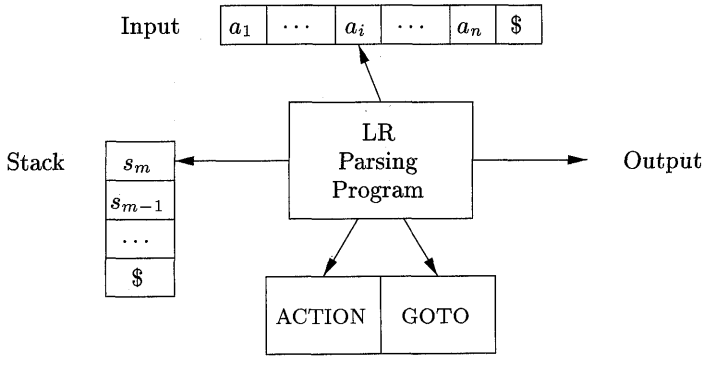


Figure 4.35: Model of an LR parser

The stack holds a sequence of states, $s_0 s_1 \cdots s_m$, where s_m is on top. In the SLR method, the stack holds states from the LR(0) automaton; the canonical-LR and LALR methods are similar. By construction, each state has a corresponding grammar symbol. Recall that states correspond to sets of items, and that there is a transition from state i to state j if GOTO$(I_i, X) = I_j$. All transitions to state j must be for the same grammar symbol X. Thus, each state, except the start state 0, has a unique grammar symbol associated with it.[4]

[4]The converse need not hold; that is, more than one state may have the same grammar

Structure of the LR Parsing Table

The parsing table consists of two parts: a parsing-action function ACTION and a goto function GOTO.

1. The ACTION function takes as arguments a state i and a terminal a (or \$, the input endmarker). The value of ACTION$[i, a]$ can have one of four forms:
   (a) Shift j, where j is a state. The action taken by the parser effectively shifts input a to the stack, but uses state j to represent a.
   (b) Reduce $A \to \beta$. The action of the parser effectively reduces β on the top of the stack to head A.
   (c) Accept. The parser accepts the input and finishes parsing.
   (d) Error. The parser discovers an error in its input and takes some corrective action. We shall have more to say about how such error-recovery routines work in Sections 4.8.3 and 4.9.4.
2. We extend the GOTO function, defined on sets of items, to states: if GOTO$[I_i, A] = I_j$, then GOTO also maps a state i and a nonterminal A to state j.

LR-Parser Configurations

To describe the behavior of an LR parser, it helps to have a notation representing the complete state of the parser: its stack and the remaining input. A *configuration* of an LR parser is a pair:

$$(s_0 s_1 \cdots s_m,\ a_i a_{i+1} \cdots a_n\$)$$

where the first component is the stack contents (top on the right), and the second component is the remaining input. This configuration represents the right-sentential form

$$X_1 X_2 \cdots X_m a_i a_{i+1} \cdots a_n$$

in essentially the same way as a shift-reduce parser would; the only difference is that instead of grammar symbols, the stack holds states from which grammar symbols can be recovered. That is, X_i is the grammar symbol represented by state s_i. Note that s_0, the start state of the parser, does not represent a grammar symbol, and serves as a bottom-of-stack marker, as well as playing an important role in the parse.

---

symbol. See for example states 1 and 8 in the LR(0) automaton in Fig. 4.31, which are both entered by transitions on E, or states 2 and 9, which are both entered by transitions on T.

Behavior of the LR Parser

The next move of the parser from the configuration above is determined by reading a_i, the current input symbol, and s_m, the state on top of the stack, and then consulting the entry ACTION$[s_m, a_i]$ in the parsing action table. The configurations resulting after each of the four types of move are as follows

1. If ACTION$[s_m, a_i]$ = shift s, the parser executes a shift move; it shifts the next state s onto the stack, entering the configuration

$$(s_0 s_1 \cdots s_m s,\ a_{i+1} \cdots a_n \$)$$

   The symbol a_i need not be held on the stack, since it can be recovered from s, if needed (which in practice it never is). The current input symbol is now a_{i+1}.

2. If ACTION$[s_m, a_i]$ = reduce $A \rightarrow \beta$, then the parser executes a reduce move, entering the configuration

$$(s_0 s_1 \cdots s_{m-r} s,\ a_i a_{i+1} \cdots a_n \$)$$

   where r is the length of β, and s = GOTO$[s_{m-r}, A]$. Here the parser first popped r state symbols off the stack, exposing state s_{m-r}. The parser then pushed s, the entry for GOTO$[s_{m-r}, A]$, onto the stack. The current input symbol is not changed in a reduce move. For the LR parsers we shall construct, $X_{m-r+1} \cdots X_m$, the sequence of grammar symbols corresponding to the states popped off the stack, will always match β, the right side of the reducing production.

   The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production. For the time being, we shall assume the output consists of just printing the reducing production.

3. If ACTION$[s_m, a_i]$ = accept, parsing is completed.

4. If ACTION$[s_m, a_i]$ = error, the parser has discovered an error and calls an error recovery routine.

The LR-parsing algorithm is summarized below. All LR parsers behave in this fashion; the only difference between one LR parser and another is the information in the ACTION and GOTO fields of the parsing table.

Algorithm 4.44: LR-parsing algorithm.

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.36. □

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Figure 4.36: LR-parsing program

Example 4.45: Figure 4.37 shows the ACTION and GOTO functions of an LR-parsing table for the expression grammar (4.1), repeated here with the productions numbered:

(1)	$E \to E + T$	(4)	$T \to F$
(2)	$E \to T$	(5)	$F \to (E)$
(3)	$T \to T * F$	(6)	$F \to \mathbf{id}$

The codes for the actions are:

1. si means shift and stack state i,
2. rj means reduce by the production numbered j,
3. acc means accept,
4. blank means error.

Note that the value of GOTO[s, a] for terminal a is found in the ACTION field connected with the shift action on input a for state s. The GOTO field gives GOTO[s, A] for nonterminals A. Although we have not yet explained how the entries for Fig. 4.37 were selected, we shall deal with this issue shortly.

STATE	ACTION						GOTO		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Figure 4.37: Parsing table for expression grammar

On input **id** * **id** + **id**, the sequence of stack and input contents is shown in Fig. 4.38. Also shown for clarity, are the sequences of grammar symbols corresponding to the states held on the stack. For example, at line (1) the LR parser is in state 0, the initial state with no grammar symbol, and with **id** the first input symbol. The action in row 0 and column **id** of the action field of Fig. 4.37 is s5, meaning shift by pushing state 5. That is what has happened at line (2): the state symbol 5 has been pushed onto the stack, and **id** has been removed from the input.

Then, * becomes the current input symbol, and the action of state 5 on input * is to reduce by $F \rightarrow$ **id**. One state symbol is popped off the stack. State 0 is then exposed. Since the goto of state 0 on F is 3, state 3 is pushed onto the stack. We now have the configuration in line (3). Each of the remaining moves is determined similarly. □

5.4 Syntax-Directed Translation Schemes

Syntax-directed translation schemes are a complementary notation to syntax-directed definitions. All of the applications of syntax-directed definitions in Section 5.3 can be implemented using syntax-directed translation schemes.

From Section 2.3.5, a *syntax-directed translation scheme* (SDT) is a context-free grammar with program fragments embedded within production bodies. The program fragments are called *semantic actions* and can appear at any position within a production body. By convention, we place curly braces around actions; if braces are needed as grammar symbols, then we quote them.

Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order; that is, during a preorder traversal. An example appears in Section 5.4.3.

Typically, SDT's are implemented during parsing, without building a parse tree. In this section, we focus on the use of SDT's to implement two important classes of SDD's:

1. The underlying grammar is LR-parsable, and the SDD is S-attributed.

2. The underlying grammar is LL-parsable, and the SDD is L-attributed.

We shall see how, in both these cases, the semantic rules in an SDD can be converted into an SDT with actions that are executed at the right time. During parsing, an action in a production body is executed as soon as all the grammar symbols to the left of the action have been matched.

SDT's that can be implemented during parsing can be characterized by introducing distinct *marker nonterminals* in place of each embedded action; each marker M has only one production, $M \to \epsilon$. If the grammar with marker nonterminals can be parsed by a given method, then the SDT can be implemented during parsing.

5.4.1 Postfix Translation Schemes

By far the simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDD is S-attributed. In that case, we can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production. SDT's with all actions at the right ends of the production bodies are called *postfix SDT's*.

Example 5.14: The postfix SDT in Fig. 5.18 implements the desk calculator SDD of Fig. 5.1, with one change: the action for the first production prints a value. The remaining actions are exact counterparts of the semantic rules. Since the underlying grammar is LR, and the SDD is S-attributed, these actions can be correctly performed along with the reduction steps of the parser. □

$$
\begin{array}{lcll}
L & \rightarrow & E\ \mathbf{n} & \{\ \text{print}(E.val);\ \} \\
E & \rightarrow & E_1 + T & \{\ E.val = E_1.val + T.val;\ \} \\
E & \rightarrow & T & \{\ E.val = T.val;\ \} \\
T & \rightarrow & T_1 * F & \{\ T.val = T_1.val \times F.val;\ \} \\
T & \rightarrow & F & \{\ T.val = F.val;\ \} \\
F & \rightarrow & (\ E\ ) & \{\ F.val = E.val;\ \} \\
F & \rightarrow & \mathbf{digit} & \{\ F.val = \mathbf{digit}.lexval;\ \}
\end{array}
$$

Figure 5.18: Postfix SDT implementing the desk calculator

5.4.2 Parser-Stack Implementation of Postfix SDT's

Postfix SDT's can be implemented during LR parsing by executing the actions when reductions occur. The attribute(s) of each grammar symbol can be put on the stack in a place where they can be found during the reduction. The best plan is to place the attributes along with the grammar symbols (or the LR states that represent these symbols) in records on the stack itself.

In Fig. 5.19, the parser stack contains records with a field for a grammar symbol (or parser state) and, below it, a field for an attribute. The three grammar symbols XYZ are on top of the stack; perhaps they are about to be reduced according to a production like $A \rightarrow XYZ$. Here, we show $X.x$ as the one attribute of X, and so on. In general, we can allow for more attributes, either by making the records large enough or by putting pointers to records on the stack. With small attributes, it may be simpler to make the records large enough, even if some fields go unused some of the time. However, if one or more attributes are of unbounded size — say, they are character strings — then it would be better to put a pointer to the attribute's value in the stack record and store the actual value in some larger, shared storage area that is not part of the stack.

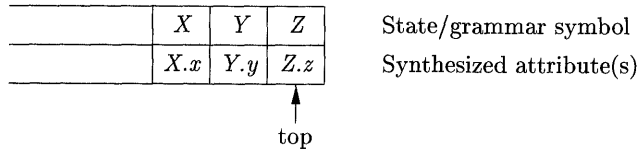


Figure 5.19: Parser stack with a field for synthesized attributes

If the attributes are all synthesized, and the actions occur at the ends of the productions, then we can compute the attributes for the head when we reduce the body to the head. If we reduce by a production such as $A \rightarrow XYZ$, then we have all the attributes of X, Y, and Z available, at known positions on the stack, as in Fig. 5.19. After the action, A and its attributes are at the top of the stack, in the position of the record for X.

Example 5.15: Let us rewrite the actions of the desk-calculator SDT of Ex-

ample 5.14 so that they manipulate the parser stack explicitly. Such stack manipulation is usually done automatically by the parser.

PRODUCTION	ACTIONS
$L \rightarrow E$ **n**	{ print($stack[top-1].val$); $top = top - 1$; }
$E \rightarrow E_1 + T$	{ $stack[top-2].val = stack[top-2].val + stack[top].val$; $top = top - 2$; }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ $stack[top-2].val = stack[top-2].val \times stack[top].val$; $top = top - 2$; }
$T \rightarrow F$	
$F \rightarrow (E)$	{ $stack[top-2].val = stack[top-1].val$; $top = top - 2$; }
$F \rightarrow$ **digit**	

Figure 5.20: Implementing the desk calculator on a bottom-up parsing stack

Suppose that the stack is kept in an array of records called *stack*, with *top* a cursor to the top of the stack. Thus, $stack[top]$ refers to the top record on the stack, $stack[top-1]$ to the record below that, and so on. Also, we assume that each record has a field called *val*, which holds the attribute of whatever grammar symbol is represented in that record. Thus, we may refer to the attribute $E.val$ that appears at the third position on the stack as $stack[top-2].val$. The entire SDT is shown in Fig. 5.20.

For instance, in the second production, $E \rightarrow E_1 + T$, we go two positions below the top to get the value of E_1, and we find the value of T at the top. The resulting sum is placed where the head E will appear after the reduction, that is, two positions below the current top. The reason is that after the reduction, the three topmost stack symbols are replaced by one. After computing $E.val$, we pop two symbols off the top of the stack, so the record where we placed $E.val$ will now be at the top of the stack.

In the third production, $E \rightarrow T$, no action is necessary, because the length of the stack does not change, and the value of $T.val$ at the stack top will simply become the value of $E.val$. The same observation applies to the productions $T \rightarrow F$ and $F \rightarrow$ **digit**. Production $F \rightarrow (E)$ is slightly different. Although the value does not change, two positions are removed from the stack during the reduction, so the value has to move to the position after the reduction.

Note that we have omitted the steps that manipulate the first field of the stack records — the field that gives the LR state or otherwise represents the grammar symbol. If we are performing an LR parse, the parsing table tells us what the new state is every time we reduce; see Algorithm 4.44. Thus, we may

simply place that state in the record for the new top of stack. □

5.4.3 SDT's With Actions Inside Productions

An action may be placed at any position within the body of a production. It is performed immediately after all symbols to its left are processed. Thus, if we have a production $B \rightarrow X \ \{a\} \ Y$, the action a is done after we have recognized X (if X is a terminal) or all the terminals derived from X (if X is a nonterminal). More precisely,

- If the parse is bottom-up, then we perform action a as soon as this occurrence of X appears on the top of the parsing stack.
- If the parse is top-down, we perform a just before we attempt to expand this occurrence of Y (if Y a nonterminal) or check for Y on the input (if Y is a terminal).

SDT's that can be implemented during parsing include postfix SDT's and a class of SDT's considered in Section 5.5 that implements L-attributed definitions. Not all SDT's can be implemented during parsing, as we shall see in the next example.

Example 5.16: As an extreme example of a problematic SDT, suppose that we turn our desk-calculator running example into an SDT that prints the prefix form of an expression, rather than evaluating the expression. The productions and actions are shown in Fig. 5.21.

1)	L	$\rightarrow$	E **n**
2)	E	$\rightarrow$	{ print('+'); } $E_1 + T$
3)	E	$\rightarrow$	T
4)	T	$\rightarrow$	{ print('*'); } $T_1 * F$
5)	T	$\rightarrow$	F
6)	F	$\rightarrow$	(E)
7)	F	$\rightarrow$	**digit** { print(**digit**.*lexval*); }

Figure 5.21: Problematic SDT for infix-to-prefix translation during parsing

Unfortunately, it is impossible to implement this SDT during either top-down or bottom-up parsing, because the parser would have to perform critical actions, like printing instances of $*$ or $+$, long before it knows whether these symbols will appear in its input.

Using marker nonterminals M_2 and M_4 for the actions in productions 2 and 4, respectively, on input 3, a shift-reduce parser (see Section 4.5.3) has conflicts between reducing by $M_2 \rightarrow \epsilon$, reducing by $M_4 \rightarrow \epsilon$, and shifting the digit. □

Any SDT can be implemented as follows:

1. Ignoring the actions, parse the input and produce a parse tree as a result.
2. Then, examine each interior node N, say one for production $A \to \alpha$. Add additional children to N for the actions in α, so the children of N from left to right have exactly the symbols and actions of α.
3. Perform a preorder traversal (see Section 2.3.4) of the tree, and as soon as a node labeled by an action is visited, perform that action.

For instance, Fig. 5.22 shows the parse tree for expression $3 * 5 + 4$ with actions inserted. If we visit the nodes in preorder, we get the prefix form of the expression: $+ * 3\ 5\ 4$.

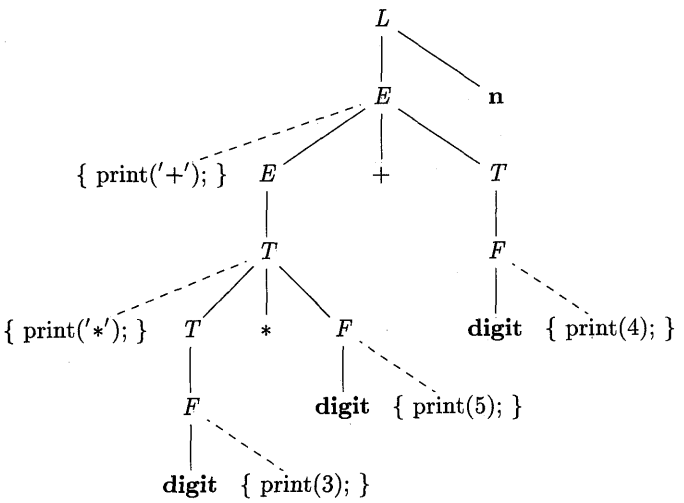


Figure 5.22: Parse tree with actions embedded

5.4.4 Eliminating Left Recursion From SDT's

Since no grammar with left recursion can be parsed deterministically top-down, we examined left-recursion elimination in Section 4.3.3. When the grammar is part of an SDT, we also need to worry about how the actions are handled.

First, consider the simple case, in which the only thing we care about is the order in which the actions in an SDT are performed. For example, if each action simply prints a string, we care only about the order in which the strings are printed. In this case, the following principle can guide us:

- When transforming the grammar, treat the actions as if they were terminal symbols.

This principle is based on the idea that the grammar transformation preserves the order of the terminals in the generated string. The actions are therefore executed in the same order in any left-to-right parse, top-down or bottom-up.

The "trick" for eliminating left recursion is to take two productions

$$A \rightarrow A\alpha \mid \beta$$

that generate strings consisting of a β and any number of α's, and replace them by productions that generate the same strings using a new nonterminal R (for "remainder") of the first production:

$$\begin{array}{l} A \rightarrow \beta R \\ R \rightarrow \alpha R \mid \epsilon \end{array}$$

If β does not begin with A, then A no longer has a left-recursive production. In regular-definition terms, with both sets of productions, A is defined by $\beta(\alpha)^*$. See Section 4.3.3 for the handling of situations where A has more recursive or nonrecursive productions.

Example 5.17: Consider the following E-productions from an SDT for translating infix expressions into postfix notation:

$$\begin{array}{lcll} E & \rightarrow & E_1 + T & \{\ \text{print}('+');\ \} \\ E & \rightarrow & T & \end{array}$$

If we apply the standard transformation to E, the remainder of the left-recursive production is

$$\alpha \ = \ + \ T \ \{\ \text{print}('+');\ \}$$

and β, the body of the other production is T. If we introduce R for the remainder of E, we get the set of productions:

$$\begin{array}{lcl} E & \rightarrow & T\ R \\ R & \rightarrow & +\ T\ \{\ \text{print}('+');\ \}\ R \\ R & \rightarrow & \epsilon \end{array}$$

□

When the actions of an SDD compute attributes rather than merely printing output, we must be more careful about how we eliminate left recursion from a grammar. However, if the SDD is S-attributed, then we can always construct an SDT by placing attribute-computing actions at appropriate positions in the new productions.

We shall give a general schema for the case of a single recursive production, a single nonrecursive production, and a single attribute of the left-recursive nonterminal; the generalization to many productions of each type is not hard, but is notationally cumbersome. Suppose that the two productions are

$$\begin{array}{lcl} A & \rightarrow & A_1\ Y\ \{A.a = g(A_1.a, Y.y)\} \\ A & \rightarrow & X\ \{A.a = f(X.x)\} \end{array}$$

Here, $A.a$ is the synthesized attribute of left-recursive nonterminal A, and X and Y are single grammar symbols with synthesized attributes $X.x$ and $Y.y$, respectively. These could represent a string of several grammar symbols, each with its own attribute(s), since the schema has an arbitrary function g computing $A.a$ in the recursive production and an arbitrary function f computing $A.a$ in the second production. In each case, f and g take as arguments whatever attributes they are allowed to access if the SDD is S-attributed.

We want to turn the underlying grammar into

$$\begin{array}{lcl} A & \rightarrow & X\ R \\ R & \rightarrow & Y\ R \mid \epsilon \end{array}$$

Figure 5.23 suggests what the SDT on the new grammar must do. In (a) we see the effect of the postfix SDT on the original grammar. We apply f once, corresponding to the use of production $A \rightarrow X$, and then apply g as many times as we use the production $A \rightarrow AY$. Since R generates a "remainder" of Y's, its translation depends on the string to its left, a string of the form $XYY\cdots Y$. Each use of the production $R \rightarrow YR$ results in an application of g. For R, we use an inherited attribute $R.i$ to accumulate the result of successively applying g, starting with the value of $A.a$.

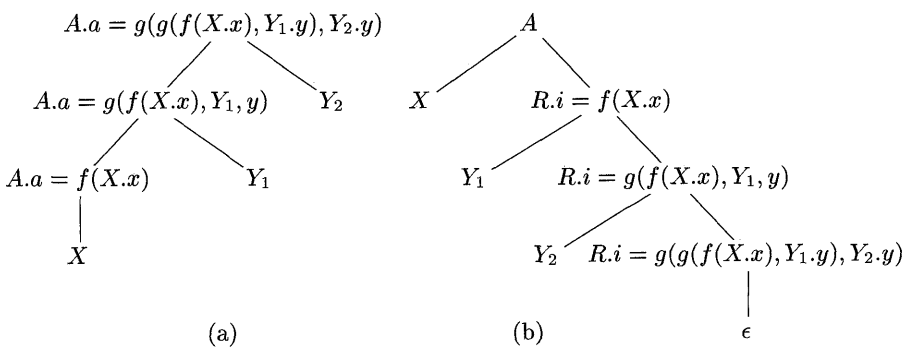


Figure 5.23: Eliminating left recursion from a postfix SDT

In addition, R has a synthesized attribute $R.s$, not shown in Fig. 5.23. This attribute is first computed when R ends its generation of Y symbols, as signaled by the use of production $R \rightarrow \epsilon$. $R.s$ is then copied up the tree, so it can become the value of $A.a$ for the entire expression $XYY\cdots Y$. The case where A generates XYY is shown in Fig. 5.23, and we see that the value of $A.a$ at the root of (a) has two uses of g. So does $R.i$ at the bottom of tree (b), and it is this value of $R.s$ that gets copied up that tree.

To accomplish this translation, we use the following SDT:

$$\begin{array}{lcl} A & \rightarrow & X \ \{R.i = f(X.x)\} \ R \ \{A.a = R.s\} \\ R & \rightarrow & Y \ \{R_1.i = g(R.i, Y.y)\} \ R_1 \ \{R.s = R_1.s\} \\ R & \rightarrow & \epsilon \ \{R.s = R.i\} \end{array}$$

Notice that the inherited attribute $R.i$ is evaluated immediately before a use of R in the body, while the synthesized attributes $A.a$ and $R.s$ are evaluated at the ends of the productions. Thus, whatever values are needed to compute these attributes will be available from what has been computed to the left.

5.4.5 SDT's for L-Attributed Definitions

In Section 5.4.1, we converted S-attributed SDD's into postfix SDT's, with actions at the right ends of productions. As long as the underlying grammar is LR, postfix SDT's can be parsed and translated bottom-up.

Now, we consider the more general case of an L-attributed SDD. We shall assume that the underlying grammar can be parsed top-down, for if not it is frequently impossible to perform the translation in connection with either an LL or an LR parser. With any grammar, the technique below can be implemented by attaching actions to a parse tree and executing them during preorder traversal of the tree.

The rules for turning an L-attributed SDD into an SDT are as follows:

1. Embed the action that computes the inherited attributes for a nonterminal A immediately before that occurrence of A in the body of the production. If several inherited attributes for A depend on one another in an acyclic fashion, order the evaluation of attributes so that those needed first are computed first.

2. Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production.

We shall illustrate these principles with two extended examples. The first involves typesetting. It illustrates how the techniques of compiling can be used in language processing for applications other than what we normally think of as programming languages. The second example is about the generation of intermediate code for a typical programming-language construct: a form of while-statement.

Example 5.18: This example is motivated by languages for typesetting mathematical formulas. `Eqn` is an early example of such a language; ideas from `Eqn` are still found in the TEX typesetting system, which was used to produce this book.

We shall concentrate on only the capability to define subscripts, subscripts of subscripts, and so on, ignoring superscripts, built-up fractions, and all other mathematical features. In the `Eqn` language, one writes `a sub i sub j` to set the expression a_{i_j}. A simple grammar for *boxes* (elements of text bounded by a rectangle) is

$$B \rightarrow B_1\ B_2 \mid B_1\ \textbf{sub}\ B_2 \mid (\ B_1\ ) \mid \textbf{text}$$

Corresponding to these four productions, a box can be either

1. Two boxes, juxtaposed, with the first, B_1, to the left of the other, B_2.
2. A box and a subscript box. The second box appears in a smaller size, lower, and to the right of the first box.
3. A parenthesized box, for grouping of boxes and subscripts. Eqn and TEX both use curly braces for grouping, but we shall use ordinary, round parentheses to avoid confusion with the braces that surround actions in SDT's.
4. A text string, that is, any string of characters.

This grammar is ambiguous, but we can still use it to parse bottom-up if we make subscripting and juxtaposition right associative, with **sub** taking precedence over juxtaposition.

Expressions will be typeset by constructing larger boxes out of smaller ones. In Fig. 5.24, the boxes for E_1 and *.height* are about to be juxtaposed to form the box for E_1*.height*. The left box for E_1 is itself constructed from the box for E and the subscript 1. The subscript 1 is handled by shrinking its box by about 30%, lowering it, and placing it after the box for E. Although we shall treat *.height* as a text string, the rectangles within its box show how it can be constructed from boxes for the individual letters.

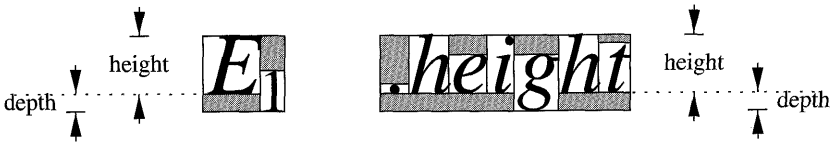


Figure 5.24: Constructing larger boxes from smaller ones

In this example, we concentrate on the vertical geometry of boxes only. The horizontal geometry — the widths of boxes — is also interesting, especially when different characters have different widths. It may not be readily apparent, but each of the distinct characters in Fig. 5.24 has a different width.

The values associated with the vertical geometry of boxes are as follows:

a) The *point size* is used to set text within a box. We shall assume that characters not in subscripts are set in 10 point type, the size of type in this book. Further, we assume that if a box has point size p, then its subscript box has the smaller point size $0.7p$. Inherited attribute $B.ps$ will represent the point size of block B. This attribute must be inherited, because the context determines by how much a given box needs to be shrunk, due to the number of levels of subscripting.

b) Each box has a *baseline*, which is a vertical position that corresponds to the bottoms of lines of text, not counting any letters, like "g" that extend below the normal baseline. In Fig. 5.24, the dotted line represents the baseline for the boxes E, *.height*, and the entire expression. The baseline for the box containing the subscript 1 is adjusted to lower the subscript.

c) A box has a *height*, which is the distance from the top of the box to the baseline. Synthesized attribute $B.ht$ gives the height of box B.

d) A box has a *depth*, which is the distance from the baseline to the bottom of the box. Synthesized attribute $B.dp$ gives the depth of box B.

The SDD in Fig. 5.25 gives rules for computing point sizes, heights, and depths. Production 1 is used to assign $B.ps$ the initial value 10.

	PRODUCTION	SEMANTIC RULES
1)	$S \rightarrow B$	$B.ps = 10$
2)	$B \rightarrow B_1\ B_2$	$B_1.ps = B.ps$ $B_2.ps = B.ps$ $B.ht = \max(B_1.ht, B_2.ht)$ $B.dp = \max(B_1.dp, B_2.dp)$
3)	$B \rightarrow B_1$ **sub** B_2	$B_1.ps = B.ps$ $B_2.ps = 0.7 \times B.ps$ $B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps)$ $B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps)$
4)	$B \rightarrow (\ B_1\ )$	$B_1.ps = B.ps$ $B.ht = B_1.ht$ $B.dp = B_1.dp$
5)	$B \rightarrow$ **text**	$B.ht = getHt(B.ps, \textbf{text}.lexval)$ $B.dp = getDp(B.ps, \textbf{text}.lexval)$

Figure 5.25: SDD for typesetting boxes

Production 2 handles juxtaposition. Point sizes are copied down the parse tree; that is, two sub-boxes of a box inherit the same point size from the larger box. Heights and depths are computed up the tree by taking the maximum. That is, the height of the larger box is the maximum of the heights of its two components, and similarly for the depth.

Production 3 handles subscripting and is the most subtle. In this greatly simplified example, we assume that the point size of a subscripted box is 70% of the point size of its parent. Reality is much more complex, since subscripts cannot shrink indefinitely; in practice, after a few levels, the sizes of subscripts

shrink hardly at all. Further, we assume that the baseline of a subscript box drops by 25% of the parent's point size; again, reality is more complex.

Production 4 copies attributes appropriately when parentheses are used. Finally, production 5 handles the leaves that represent text boxes. In this matter too, the true situation is complicated, so we merely show two unspecified functions *getHt* and *getDp* that examine tables created with each font to determine the maximum height and maximum depth of any characters in the text string. The string itself is presumed to be provided as the attribute *lexval* of terminal **text**.

Our last task is to turn this SDD into an SDT, following the rules for an L-attributed SDD, which Fig. 5.25 is. The appropriate SDT is shown in Fig. 5.26. For readability, since production bodies become long, we split them across lines and line up the actions. Production bodies therefore consist of the contents of all lines up to the head of the next production. □

	PRODUCTION	ACTIONS
1)	$S \rightarrow B$	$\{\ B.ps = 10;\ \}$
2)	$B \rightarrow$	$\{ B_1.ps = B.ps;\ \}$
	B_1	$\{ B_2.ps = B.ps;\ \}$
	B_2	$\{\ B.ht = \max(B_1.ht, B_2.ht);$
		$\quad B.dp = \max(B_1.dp, B_2.dp);\ \}$
3)	$B \rightarrow$	$\{ B_1.ps = B.ps;\ \}$
	B_1 **sub**	$\{ B_2.ps = 0.7 \times B.ps;\ \}$
	B_2	$\{\ B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps);$
		$\quad B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps);\ \}$
4)	$B \rightarrow ($	$\{ B_1.ps = B.ps;\ \}$
	$B_1\ )$	$\{\ B.ht = B_1.ht;$
		$\quad B.dp = B_1.dp;\ \}$
5)	$B \rightarrow$ **text**	$\{\ B.ht = getHt(B.ps, \textbf{text}.lexval);$
		$\quad B.dp = getDp(B.ps, \textbf{text}.lexval);\ \}$

Figure 5.26: SDT for typesetting boxes

Our next example concentrates on a simple while-statement and the generation of intermediate code for this type of statement. Intermediate code will be treated as a string-valued attribute. Later, we shall explore techniques that involve the writing of pieces of a string-valued attribute as we parse, thus avoiding the copying of long strings to build even longer strings. The technique was introduced in Example 5.17, where we generated the postfix form of an infix

expression "on-the-fly," rather than computing it as an attribute. However, in our first formulation, we create a string-valued attribute by concatenation.

Example 5.19: For this example, we only need one production:

$$S \rightarrow \textbf{while} \ (\ C \ ) \ S_1$$

Here, S is the nonterminal that generates all kinds of statements, presumably including if-statements, assignment statements, and others. In this example, C stands for a conditional expression — a boolean expression that evaluates to true or false.

In this flow-of-control example, the only things we ever generate are labels. All the other intermediate-code instructions are assumed to be generated by parts of the SDT that are not shown. Specifically, we generate explicit instructions of the form **label** L, where L is an identifier, to indicate that L is the label of the instruction that follows. We assume that the intermediate code is like that introduced in Section 2.8.4.

The meaning of our while-statement is that the conditional C is evaluated. If it is true, control goes to the beginning of the code for S_1. If false, then control goes to the code that follows the while-statement's code. The code for S_1 must be designed to jump to the beginning of the code for the while-statement when finished; the jump to the beginning of the code that evaluates C is not shown in Fig. 5.27.

We use the following attributes to generate the proper intermediate code:

1. The inherited attribute $S.next$ labels the beginning of the code that must be executed after S is finished.

2. The synthesized attribute $S.code$ is the sequence of intermediate-code steps that implements a statement S and ends with a jump to $S.next$.

3. The inherited attribute $C.true$ labels the beginning of the code that must be executed if C is true.

4. The inherited attribute $C.false$ labels the beginning of the code that must be executed if C is false.

5. The synthesized attribute $C.code$ is the sequence of intermediate-code steps that implements the condition C and jumps either to $C.true$ or to $C.false$, depending on whether C is true or false.

The SDD that computes these attributes for the while-statement is shown in Fig. 5.27. A number of points merit explanation:

- The function *new* generates new labels.

- The variables $L1$ and $L2$ hold labels that we need in the code. $L1$ is the beginning of the code for the while-statement, and we need to arrange

$S \rightarrow$ **while** (C) S_1	$L1 = new();$
	$L2 = new();$
	$S_1.next = L1;$
	$C.false = S.next;$
	$C.true = L2;$
	$S.code =$ **label** $\|\|$ $L1$ $\|\|$ $C.code$ $\|\|$ **label** $\|\|$ $L2$ $\|\|$ $S_1.code$

Figure 5.27: SDD for while-statements

that S_1 jumps there after it finishes. That is why we set $S_1.next$ to $L1$. $L2$ is the beginning of the code for S_1, and it becomes the value of $C.true$, because we branch there when C is true.

- Notice that $C.false$ is set to $S.next$, because when the condition is false, we execute whatever code must follow the code for S.
- We use $||$ as the symbol for concatenation of intermediate-code fragments. The value of $S.code$ thus begins with the label $L1$, then the code for condition C, another label $L2$, and the code for S_1.

This SDD is L-attributed. When we convert it into an SDT, the only remaining issue is how to handle the labels $L1$ and $L2$, which are variables, and not attributes. If we treat actions as dummy nonterminals, then such variables can be treated as the synthesized attributes of dummy nonterminals. Since $L1$ and $L2$ do not depend on any other attributes, they can be assigned to the first action in the production. The resulting SDT with embedded actions that implements this L-attributed definition is shown in Fig. 5.28. □

$S \rightarrow$ **while** (	{ $L1 = new();$ $L2 = new();$ $C.false = S.next;$ $C.true = L2;$ }
C)	{ $S_1.next = L1;$ }
S_1	{ $S.code =$ **label** $\|\|$ $L1$ $\|\|$ $C.code$ $\|\|$ **label** $\|\|$ $L2$ $\|\|$ $S_1.code;$ }

Figure 5.28: SDT for while-statements

5.4.6 Exercises for Section 5.4

Exercise 5.4.1: We mentioned in Section 5.4.2 that it is possible to deduce, from the LR state on the parsing stack, what grammar symbol is represented by the state. How would we discover this information?

Exercise 5.4.2: Rewrite the following SDT:

$$A \rightarrow A \ \{a\} \ B \mid A \ B \ \{b\} \mid 0$$
$$B \rightarrow B \ \{c\} \ A \mid B \ A \ \{d\} \mid 1$$

so that the underlying grammar becomes non-left-recursive. Here, a, b, c, and d are actions, and 0 and 1 are terminals.

! **Exercise 5.4.3:** The following SDT computes the value of a string of 0's and 1's interpreted as a positive, binary integer.

$$\begin{array}{rcl} B & \rightarrow & B_1\ 0\ \{B.val = 2 \times B_1.val\} \\ & | & B_1\ 1\ \{B.val = 2 \times B_1.val + 1\} \\ & | & 1\ \{B.val = 1\} \end{array}$$

Rewrite this SDT so the underlying grammar is not left recursive, and yet the same value of $B.val$ is computed for the entire input string.

! **Exercise 5.4.4:** Write L-attributed SDD's analogous to that of Example 5.19 for the following productions, each of which represents a familiar flow-of-control construct, as in the programming language C. You may need to generate a three-address statement to jump to a particular label L, in which case you should generate **goto** L.

a) $S \rightarrow$ **if** (C) S_1 **else** S_2

b) $S \rightarrow$ **do** S_1 **while** (C)

c) $S \rightarrow$ '{' L '}'; $L \rightarrow L\ S \mid \epsilon$

Note that any statement in the list can have a jump from its middle to the next statement, so it is not sufficient simply to generate code for each statement in order.

Exercise 5.4.5: Convert each of your SDD's from Exercise 5.4.4 to an SDT in the manner of Example 5.19.

Exercise 5.4.6: Modify the SDD of Fig. 5.25 to include a synthesized attribute $B.le$, the length of a box. The length of the concatenation of two boxes is the sum of the lengths of each. Then add your new rules to the proper positions in the SDT of Fig. 5.26

Exercise 5.4.7: Modify the SDD of Fig. 5.25 to include superscripts denoted by operator **sup** between boxes. If box B_2 is a superscript of box B_1, then position the baseline of B_2 0.6 times the point size of B_1 above the baseline of B_1. Add the new production and rules to the SDT of Fig. 5.26.

Regular Expressions

Regular Expressions are so cool. Knowledge of regexes will allow you to save the day.

CONTENTS

Definitions

In formal language theory, a **regular expression** (a.k.a. regex, regexp, or r.e.), is a string that represents a regular (type-3) language.

Huh??

Okay, in many programming languages, a **regular expression** is a **pattern** that **matches** strings or pieces of strings. *The set of strings they are capable of matching goes way beyond what regular expressions from language theory can describe.*

Basic Examples

Rather than start with technical details, we'll start with a bunch of examples.

Regex	Matches any string that
`hello`	contains {hello}
`gray|grey`	contains {gray, grey}
`gr(a|e)y`	contains {gray, grey}
`gr[ae]y`	contains {gray, grey}
`b[aeiou]bble`	contains {babble, bebble, bibble, bobble, bubble}
`[b-chm-pP]at|ot`	contains {bat, cat, hat, mat, nat, oat, pat, Pat, ot}
`colou?r`	contains {color, colour}
`rege(x(es)?|xps?)`	contains {regex, regexes, regexp, regexps}
`go*gle`	contains {ggle, gogle, google, gooogle, goooogle, ...}
`go+gle`	contains {gogle, google, gooogle, goooogle, ...}
`g(oog)+le`	contains {google, googoogle, googoogoogle, googoogoogoogle, ...}
`z{3}`	contains {zzz}
`z{3,6}`	contains {zzz, zzzz, zzzzz, zzzzzz}
`z{3,}`	contains {zzz, zzzz, zzzzz, ...}
`[Bb]rainf\*\*k`	contains {Brainf**k, brainf**k}
`\d`	contains {0,1,2,3,4,5,6,7,8,9}
`\d{5}(-\d{4})?`	contains a United States zip code
`1\d{10}`	contains an 11-digit string starting with a 1
`[2-9]|[12]\d|3[0-6]`	contains an integer in the range 2..36 inclusive
`Hello\nworld`	contains Hello followed by a newline followed by world
`mi.....ft`	contains a nine-character (sub)string beginning with mi and ending with ft (Note: depending on context, the dot stands either for “any character at all” or “any character except a newline”.) Each dot is allowed to match a different character, so both microsoft and minecraft will match.
`\d+(\.\d\d)?`	contains a positive integer or a floating point number with exactly two characters after the decimal point.

`[^i*&2@]`	contains any character other than an i, asterisk, ampersand, 2, or at-sign.
`//[^\r\n]*[\r\n]`	contains a Java or C# slash-slash comment
`^dog`	begins with "dog"
`dog$`	ends with "dog"
`^dog$`	is exactly "dog"

Notation

There are many different syntaxes for regular expressions, but in general you will see that:

- Most characters stand for themselves
- Certain characters, called metacharacters, have special meaning and **must be escaped** (usually with `\`) if you want to use them as characters. In most syntaxes the metacharacters are:

  ```
  (  )  [  ]  {  }  ^  $  .  \  ?  *  +  |
  ```

- Within square brackets, you only have to escape (1) an initial `^`, (2) a non-initial or non-final `-`, (3) a non-initial `]`, and (4) a `\`.

Using Regular Expressions

Many languages allow programmers to define regexes and then use them to:

- **Validate** that a piece of text (or a portion of that text) matches some pattern
- **Find** fragments of some text that match some pattern
- **Extract** fragments of some text
- **Replace** fragments of text with other text

Generally a regex is first **compiled** into some internal form that can be used for super fast validation, extraction, and replacing. Sometimes there is an explicit `compile` function or method, and sometimes special syntax is used to compile, such as the very common form `/.../`.

Validation

Example: find `"color"` or `"colour"` in a given string.

```
// Java
Pattern p = Pattern.compile("colou?r");
Matcher m = p.matcher("The color green");
m.find();                           // returns true
m.start();                          // returns 4
m.end();                            // returns 9
m = p.matcher("abc");
m.find();                           // returns false
```

```
# Perl
$p = /colou?r/;
"The color green" =~ $p;            # returns 1 (cuz no Perl true)
"abc" =~ $p;                        # returns 0 (cuz no Perl false)
```

```
# Ruby
p = /colou?r/
"The color green" =~ p              # returns 4
"abc" =~ p                          # returns nil
```

```
# Python
p = re.compile("colou?r")
m = p.search("The color green")
m.start()                           # returns 4
m = p.search("abc")                 # returns None
```

```
// JavaScript
const p = /colou?r/;
```

```
"The color green".search(p);        // returns 4
"abc".search(p);                    // returns -1
```

If you want to know if an entire string matches a pattern, define the pattern with `^` and `$`, or with `\A` and `\Z`. In Java, you can call `matches()` instead of `find()`.

Extraction

After doing a match against a pattern, most regex engines will return you a bundle of information, including such things as:

- the part of the text that matched the pattern
- the index within the string where the match begins
- each part of the text matching the parenthesized portions within the pattern
- (sometimes) the text before the matched text
- (sometimes) the text after the matched text

Example in Ruby:

```
>> pattern = /weighs (\d+(\.\d+)?) (\w+)/
=> /weighs (\d+(\.\d+)?) (\w+)/
>> pattern =~ 'The thing weighs 2.5 kilograms or so.'
=> 10
>> $&
=> "weighs 2.5 kilograms"
>> $1
=> "2.5"
>> $2
=> ".5"
>> $3
=> "kilograms"
>> $`
=> "The thing "
>> $'
=> " or so."
```

The same thing in JavaScript:

```
> const pattern = /weighs (\d+(\.\d+)?) (\w+)/
> pattern.exec('The thing weighs 2.5 kilograms or so.')
[ 'weighs 2.5 kilograms',
  '2.5',
  '.5',
  'kilograms',
```

```
  index: 10,
  input: 'The thing weighs 2.5 kilograms or so.' ]
```

Note how in JavaScript, the match result object looks like an array and an object.

The so-called *group numbers* are found by counting the left-parentheses in the pattern:

TODO PICTURE GOES HERE

Sometimes you need parentheses only for precedence purposes and you don't want to incur the cost of extracting a group. We have **non-capturing groups** for this purpose.

Ruby:

```
>> phone = /((\d{3})(?:\.|-))?(\d{3})(?:\.|-)(\d{4})/
=> /((\d{3})(?:\.|-))?(\d{3})(?:\.|-)(\d{4})/
>> phone =~ 'Call 555-1212 for info'
=> 5
>> [$`, $&, $', $1, $2, $3, $4, $5]
=> ["Call ", "555-1212", " for info", nil, nil, "555", "1212", nil]
>> phone =~ '800.221.9989'
=> 0
>> [$`, $&, $', $1, $2, $3, $4, $5]
=> ["", "800.221.9989", "", "800.", "800", "221", "9989", nil]
>> phone =~ '1800.221.9989'
=> 1
>> [$`, $&, $', $1, $2, $3, $4, $5]
=> ["1", "800.221.9989", "", "800.", "800", "221", "9989", nil]
```

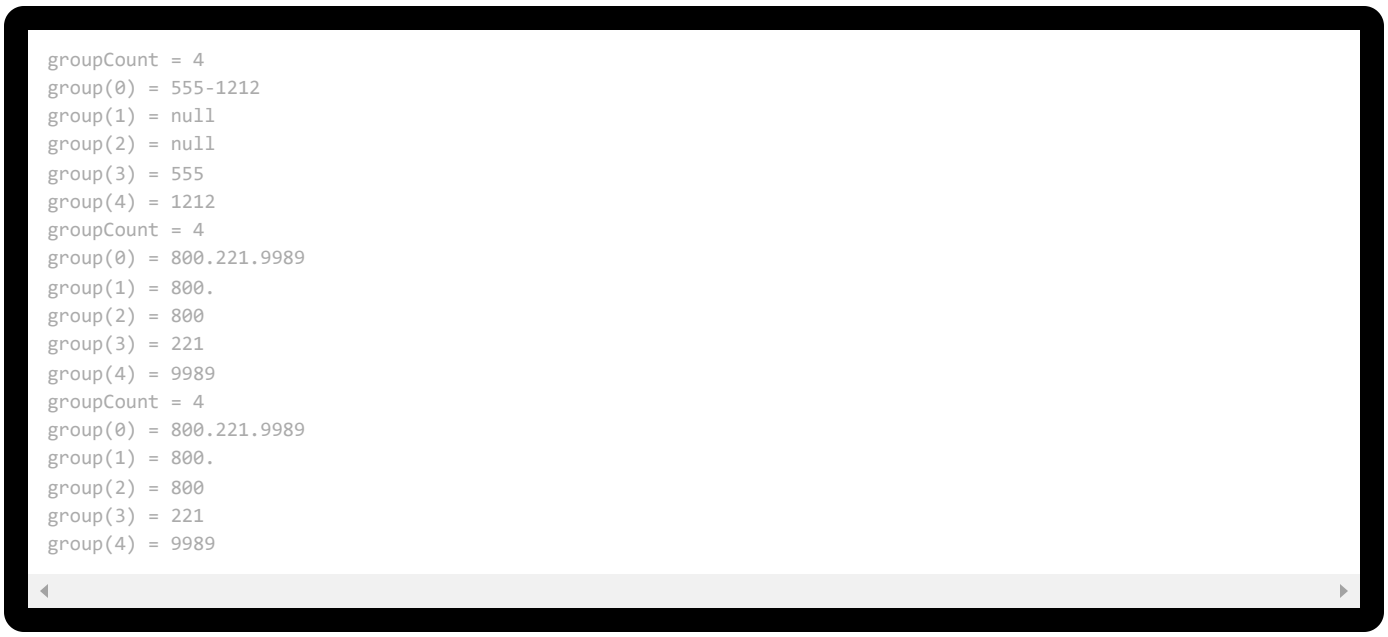
JavaScript:

```
> const r = /((\d{3})(?:\.|-))?(\d{3})(?:\.|-)(\d{4})/g;
> const m = r.exec("Call 1.800.555-1212 for info");
> m.index
7
> JSON.stringify(m);
["800.555-1212","800.","800","555","1212"]
```

Java

```
Pattern phone = Pattern.compile("((\\d{3})(?:\\.|-))?(\\d{3})(?:\\.|-)(\\d{4})");
String[] tests = {"Call 555-1212 for info", "800.221.9989", "1800.221.9989"};
for (String s : tests) {
    Matcher m = phone.matcher(s);
    m.find();
    System.out.println("groupCount = " + m.groupCount());
```

```
        System.out.println("group(0) = " + m.group(0));
        System.out.println("group(1) = " + m.group(1));
        System.out.println("group(2) = " + m.group(2));
        System.out.println("group(3) = " + m.group(3));
        System.out.println("group(4) = " + m.group(4));
    }
```



```
groupCount = 4
group(0) = 555-1212
group(1) = null
group(2) = null
group(3) = 555
group(4) = 1212
groupCount = 4
group(0) = 800.221.9989
group(1) = 800.
group(2) = 800
group(3) = 221
group(4) = 9989
groupCount = 4
group(0) = 800.221.9989
group(1) = 800.
group(2) = 800
group(3) = 221
group(4) = 9989
```

Substitution

Many languages have `replace` or `replaceAll` methods that replace the parts of a string that match a regex. Sometimes you will see a `g` flag on a regex instead of a `replaceAll` function.

```
alert("Rascally Rabbit".replace(/[RrLl]/g, "w"));
```

Components of Regexes

Character Classes

- Square brackets `[ ]` — means exactly one character
- A leading `^` negates, a non-leading, non-terminal `-` defines a range:

```
[abc]          a or b or c
[^abc]         any character _except_ a, b, or c (negation)
[a-zA-Z]       a through z or A through Z, inclusive (range)
```

- If you have a `]` in your set, put it first. Use `\` to escape.
- Java allows crazy extensions:

```
[a-d[m-p]]     [a-dm-p] (union, Java only, I think)
[a-e&&[def]]   [de] (intersection, Java only, I think)
[a-r&&[^bq-z]] [ac-p] (subtraction, Java only, I think)
```

- Other ways to say exactly one character from a set are:

```
\d             [0-9]
\D             [^\d]
\s             [ \t\n\x0B\f\r]
\S             [^\s]
\w             [a-zA-Z0-9_]
\W             [^\w]
.              any character at all, except maybe not a line termina
```

Groups

Defined above, in the section on extraction.

Quantifiers

Generally, 18 types:

	Eager	Reluctant	Possessive
Zero or one	`?`	`??`	`?+`
Zero or more	`*`	`*?`	`*+`
One or more	`+`	`+?`	`++`
m times	`{m}`	`{m}?`	`{m}+`
At least m times	`{m,}`	`{m,}?`	`{m,}+`
At least m, at most n times	`{m,n}`	`{m,n}?`	`{m,n}+`

Eager (Greedy and Generous) — match as much as possible, but give back

```
\w+\d\d\w+  // matches abcdef42skjhfskjfhsjdfs
            // but inefficiently
```

Possessive — match as much as possible, but do NOT give back

```
\w++\d\d\w+  // does not match abcdef42skjhfskjfhsjdfs
             // but is efficient
```

Reluctant — match as little as possible

```
\w+?\d\d\w+  // matches abcdef42skjhfskjfhsjdfs
             // efficiently, yay!
```

Backreferences

Things captured can be used later:

```
<(\w+)>[^<]*</\1>
```

Anchors, Boundaries, Delimiters

Some regex tokens do not consume characters! They just assert the matching engine is at a particular place, so to speak.

- `^`: Beginning of string (or line, depending on the mode)
- `$`: End of string (or line, depending on the mode)
- `\A`: Beginning of string
- `\z`: End of string
- `\Z`: Varies a lot depending on the engine, so be careful with it
- `\b`: Word boundary
- `\B`: Not a word boundary

Read more about these at Rexegg.

Also, the lookarounds (up next!) don't consume any characters either!

Lookarounds

- Lookarounds do not consume anything
- Even though they have parens, they do not capture
- **Positive Lookahead**: Matches only if followed by something

```
Hillary(?=\s+Clinton)
```

  matches the Hillary in Hillary Clinton but not the Hillary in Hillary Makasa.

- **Negative Lookahead**: Matches only if not followed by something

```
q(?!u)
\b(?:[a-eg-z]|f(?!oo))\w*\b      // Word not starting with foo
\b(?:[a-eg-z]|f(?!oo\b))\w*\b    // Any word except foo
((?!foo).*)
```

- **Positive Lookbehind**: Matches only if preceded by something

```
(?<=-)\p{L}+              // a word following a hyphen
(?<=http://)\S+           // URL not including the http:// part
```

- **Negative Lookbehind**: Matches only if not preceded by something

```
(?<![-+\d])(\d+)      // Digits not preceded by a digit, +, or -
```

- Lookarounds show up in search and replace applications

```
Pattern p = Pattern.compile("Hillary(?=\\s+Clinton)");
String text = "Once Hillary Clinton was talking about Sir\n" +
    "Edmund Hillary to Hillary Makasa and then Hillary\n" +
    "Clinton had to run off on important business.";
Matcher m = p.matcher(text);
System.out.println(m.replaceAll("Secretary"));
```

Note: Read this awesome article on lookarounds.

Modifiers

A modifier affects the way the rest of the regex is interpreted. Not every language supports all of the modifiers below. For example, JavaScript (officially) supports only i, g, and m.

Modifier	Meaning
`g`	global
`i`	ignore case
`m`	multiple line
`s`	single line (DOTALL): Means that the dot matches any character at all. Without this modifier, the dot matches any character except a newline.
`x`	ignore whitespace in the pattern
`d`	Unix line mode: Considers only U+000A as a line separator, rather than U+000D or the U+000D/U+000A combo or even U+2028.
`u`	Unicode case: in this mode the case-insensitive modifier respects Unicode cases; outside of this mode that modifier only consolidates cases of US-ASCII

	characters.

Performance Pitfalls

You should know some things about how your *regex engine* works since two "equivalent" regexes can have drastic differences in processing speed.

- It is possible to write regexes that take exponential time to match, but you pretty much have to TRY to make one (they're pathological)
- It is more common to accidentally create regexes that run in quadratic time
- Main types of problems
  - Recompilation (from forgetting to compile regexes used multiple times)

```
// Java shortcut, should not be used in most cirumstances
s.matches("colou?r");
```

  - Dot-star in the Middle (which causes backtracking)
    - Solution 1: Use negated character class
    - Solution 2: Use reluctant quantifiers
  - Nested Repetition

Performance Tips

Always do the following:

- Use non-capturing groups when you need parentheses but not capture.
- If the regex is very complex, do a quick spot-check before attempting a match, e.g.
  - Does an email address contain '`@`'?
- Present the most likely alternative(s) first, e.g.
  - `black|white|blue|red|green|metallic seaweed`
- Reduce the amount of looping the engine has to do
  - `\d\d\d\d\d` is faster than `\d{5}`

    - `aaaa+` is faster than `a{4,}`
- Avoid obvious backtracking, e.g.
    - `Mr|Ms|Mrs` should be `M(?:rs?|s)`
    - `Good morning|Good evening` should be `Good (?:morning|evening)`

Miscellaneous Language-Specific Notes

A few things that are good to know:

- Java's built-in support for regexes exceeds that of many languages
- Especially good for Unicode and for character classes (has more than Perl)
- Syntax is more cumbersome (string literal support weak, no operator for matching...) — live with it!
- Perl has nice regex, too, even allows you can even embed code inside them.
- Great Perl documentation at the perldoc pages perlrequick, perlretut, perlre. and perlreref.
- JavaScript seems to less extensive support than other languages, but I think this is changing.
- Python puts regex functions in a module.
- Python docs are here.

Study and Practice

Here are some good sources:

- The Premier Site for Regexes
- Maybe this is a better premier site
- Regex 101—Develop regexes online
- RegExr (Awesome online tool for Java regexes)
- Rubular (Ruby Online Regex Tester)
- Perl Regular Expressions Tutorial

- Perl Regular Expressions (Manual)
- Perl Regular Expressions Quick Reference
- Ruby Regexp class documentation
- Java Regex Tutorial
- Java Regex Optimization Article
- Java Pattern class API docs
- JavaScript Regular Expressions (at Mozilla Developer Center)
- Python Regular Expressions

Chapter 5

Syntax-Directed Translation

This chapter develops the theme of Section 2.3: the translation of languages guided by context-free grammars. The translation techniques in this chapter will be applied in Chapter 6 to type checking and intermediate-code generation. The techniques are also useful for implementing little languages for specialized tasks; this chapter includes an example from typesetting.

We associate information with a language construct by attaching *attributes* to the grammar symbol(s) representing the construct, as discussed in Section 2.3.2. A syntax-directed definition specifies the values of attributes by associating semantic rules with the grammar productions. For example, an infix-to-postfix translator might have a production and rule

PRODUCTION	SEMANTIC RULE	(5.1)
$E \rightarrow E_1 + T$	$E.code = E_1.code \parallel T.code \parallel \text{'+'}$	

This production has two nonterminals, E and T; the subscript in E_1 distinguishes the occurrence of E in the production body from the occurrence of E as the head. Both E and T have a string-valued attribute *code*. The semantic rule specifies that the string $E.code$ is formed by concatenating $E_1.code$, $T.code$, and the character '+'. While the rule makes it explicit that the translation of E is built up from the translations of E_1, T, and '+', it may be inefficient to implement the translation directly by manipulating strings.

From Section 2.3.5, a syntax-directed translation scheme embeds program fragments called semantic actions within production bodies, as in

$$E \rightarrow E_1 + T \quad \{ \text{ print '+' } \} \tag{5.2}$$

By convention, semantic actions are enclosed within curly braces. (If curly braces occur as grammar symbols, we enclose them within single quotes, as in

'{' and '}'.) The position of a semantic action in a production body determines the order in which the action is executed. In production (5.2), the action occurs at the end, after all the grammar symbols; in general, semantic actions may occur at any position in a production body.

Between the two notations, syntax-directed definitions can be more readable, and hence more useful for specifications. However, translation schemes can be more efficient, and hence more useful for implementations.

The most general approach to syntax-directed translation is to construct a parse tree or a syntax tree, and then to compute the values of attributes at the nodes of the tree by visiting the nodes of the tree. In many cases, translation can be done during parsing, without building an explicit tree. We shall therefore study a class of syntax-directed translations called "L-attributed translations" (L for left-to-right), which encompass virtually all translations that can be performed during parsing. We also study a smaller class, called "S-attributed translations" (S for synthesized), which can be performed easily in connection with a bottom-up parse.

5.1 Syntax-Directed Definitions

A *syntax-directed definition* (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. If X is a symbol and a is one of its attributes, then we write $X.a$ to denote the value of a at a particular parse-tree node labeled X. If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X. Attributes may be of any kind: numbers, types, table references, or strings, for instance. The strings may even be long sequences of code, say code in the intermediate language used by a compiler.

5.1.1 Inherited and Synthesized Attributes

We shall deal with two kinds of attributes for nonterminals:

1. A *synthesized attribute* for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head. A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

2. An *inherited attribute* for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body. An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

An Alternative Definition of Inherited Attributes

No additional translations are enabled if we allow an inherited attribute $B.c$ at a node N to be defined in terms of attribute values at the children of N, as well as at N itself, at its parent, and at its siblings. Such rules can be "simulated" by creating additional attributes of B, say $B.c_1, B.c_2, \ldots$. These are synthesized attributes that copy the needed attributes of the children of the node labeled B. We then compute $B.c$ as an inherited attribute, using the attributes $B.c_1, B.c_2, \ldots$ in place of attributes at the children. Such attributes are rarely needed in practice.

While we do not allow an inherited attribute at node N to be defined in terms of attribute values at the children of node N, we do allow a synthesized attribute at node N to be defined in terms of inherited attribute values at node N itself.

Terminals can have synthesized attributes, but not inherited attributes. Attributes for terminals have lexical values that are supplied by the lexical analyzer; there are no semantic rules in the SDD itself for computing the value of an attribute for a terminal.

Example 5.1: The SDD in Fig. 5.1 is based on our familiar grammar for arithmetic expressions with operators + and $*$. It evaluates expressions terminated by an endmarker **n**. In the SDD, each of the nonterminals has a single synthesized attribute, called *val*. We also suppose that the terminal **digit** has a synthesized attribute *lexval*, which is an integer value returned by the lexical analyzer.

	PRODUCTION	SEMANTIC RULES
1)	$L \to E\ \mathbf{n}$	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}$.lexval

Figure 5.1: Syntax-directed definition of a simple desk calculator

The rule for production 1, $L \to E\ \mathbf{n}$, sets $L.val$ to $E.val$, which we shall see is the numerical value of the entire expression.

Production 2, $E \to E_1 + T$, also has one rule, which computes the *val* attribute for the head E as the sum of the values at E_1 and T. At any parse-

tree node N labeled E, the value of *val* for E is the sum of the values of *val* at the children of node N labeled E and T.

Production 3, $E \rightarrow T$, has a single rule that defines the value of *val* for E to be the same as the value of *val* at the child for T. Production 4 is similar to the second production; its rule multiplies the values at the children instead of adding them. The rules for productions 5 and 6 copy values at a child, like that for the third production. Production 7 gives $F.val$ the value of a digit, that is, the numerical value of the token **digit** that the lexical analyzer returned. □

An SDD that involves only synthesized attributes is called *S-attributed*; the SDD in Fig. 5.1 has this property. In an S-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.

For simplicity, the examples in this section have semantic rules without side effects. In practice, it is convenient to allow SDD's to have limited side effects, such as printing the result computed by a desk calculator or interacting with a symbol table. Once the order of evaluation of attributes is discussed in Section 5.2, we shall allow semantic rules to compute arbitrary functions, possibly involving side effects.

An S-attributed SDD can be implemented naturally in conjunction with an LR parser. In fact, the SDD in Fig. 5.1 mirrors the Yacc program of Fig. 4.58, which illustrates translation during LR parsing. The difference is that, in the rule for production 1, the Yacc program prints the value $E.val$ as a side effect, instead of defining the attribute $L.val$.

An SDD without side effects is sometimes called an *attribute grammar*. The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

5.1.2 Evaluating an SDD at the Nodes of a Parse Tree

To visualize the translation specified by an SDD, it helps to work with parse trees, even though a translator need not actually build a parse tree. Imagine therefore that the rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree. A parse tree, showing the value(s) of its attribute(s) is called an *annotated parse tree*.

How do we construct an annotated parse tree? In what order do we evaluate attributes? Before we can evaluate an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends. For example, if all attributes are synthesized, as in Example 5.1, then we must evaluate the *val* attributes at all of the children of a node before we can evaluate the *val* attribute at the node itself.

With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree; the evaluation of S-attributed definitions is discussed in Section 5.2.3.

For SDD's with both inherited and synthesized attributes, there is no guarantee that there is even one order in which to evaluate attributes at nodes. For instance, consider nonterminals A and B, with synthesized and inherited attributes $A.s$ and $B.i$, respectively, along with the production and rules

PRODUCTION	SEMANTIC RULES
$A \rightarrow B$	$A.s = B.i;$
	$B.i = A.s + 1$

These rules are circular; it is impossible to evaluate either $A.s$ at a node N or $B.i$ at the child of N without first evaluating the other. The circular dependency of $A.s$ and $B.i$ at some pair of nodes in a parse tree is suggested by Fig. 5.2.

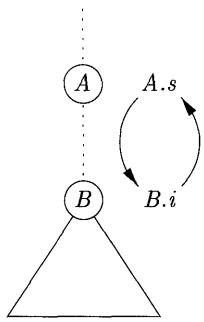


Figure 5.2: The circular dependency of $A.s$ and $B.i$ on one another

It is computationally difficult to determine whether or not there exist any circularities in any of the parse trees that a given SDD could have to translate.[1] Fortunately, there are useful subclasses of SDD's that are sufficient to guarantee that an order of evaluation exists, as we shall see in Section 5.2.

Example 5.2: Figure 5.3 shows an annotated parse tree for the input string $3 * 5 + 4$ **n**, constructed using the grammar and rules of Fig. 5.1. The values of *lexval* are presumed supplied by the lexical analyzer. Each of the nodes for the nonterminals has attribute *val* computed in a bottom-up order, and we see the resulting values associated with each node. For instance, at the node with a child labeled $*$, after computing $T.val = 3$ and $F.val = 5$ at its first and third children, we apply the rule that says $T.val$ is the product of these two values, or 15. □

Inherited attributes are useful when the structure of a parse tree does not "match" the abstract syntax of the source code. The next example shows how inherited attributes can be used to overcome such a mismatch due to a grammar designed for parsing rather than translation.

[1] Without going into details, while the problem is decidable, it cannot be solved by a polynomial-time algorithm, even if $\mathcal{P} = \mathcal{NP}$, since it has exponential time complexity.

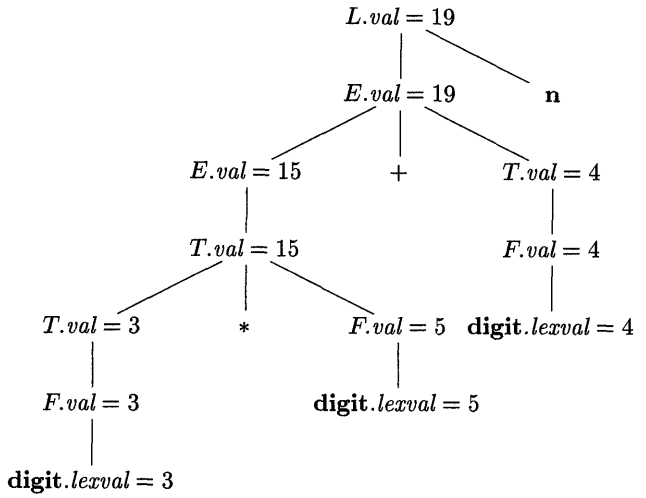


Figure 5.3: Annotated parse tree for $3 * 5 + 4$ **n**

Symbol Tables and Static Checks

Contents

Introduction

The parser ensures that the input program is syntactically correct, but there are other kinds of correctness that it cannot (or usually does not) enforce. For example:

- A variable should not be declared more than once in the same scope.
- A variable should not be used before being declared.
- The type of the left-hand side of an assignment should match the type of the right-hand side.
- Methods should be called with the right number and types of arguments.

The next phase of the compiler after the parser, sometimes called the **static semantic analyzer** is in charge of checking for these kinds of errors. The checks can be done in two phases, each of which involves traversing the abstract-syntax tree created by the parser:

1. For each scope in the program: Process the declarations, adding new entries to the symbol table and reporting any variables that are multiply declared; process the statements, finding uses of undeclared variables, and updating the "ID" nodes of the abstract-syntax tree to point to the appropriate symbol-table entry.
2. Process all of the statements in the program again, using the symbol-table information to determine the type of each expression, and finding type errors.

Below, we will consider how to build symbol tables and how to use them to find multiply-declared and undeclared variables. We will then consider type checking.

Symbol Tables

The purpose of the symbol table is to keep track of names declared in the program. This includes names of classes, fields, methods, and variables. Each symbol table entry associates a set of attributes with one name; for example:

- which kind of name it is

- what is its type
- what is its nesting level
- where will it be found at runtime.

One factor that will influence the design of the symbol table is what **scoping rules** are defined for the language being compiled. Let's consider some different kinds of scoping rules before continuing our discussion of symbol tables.

Scoping

In most languages, the same name can be declared multiple times if the declarations occur in different scopes, and/or involve different kinds of names. For example, in Java you can use the same name for a class, a field of the class, a method of the class, and a local variable of the method (this is not recommended, but it is legal):

```
class Test {
  int Test;

  void Test(  ) {
      double Test;  // could also be declared int
  }
}
```

In both Java and C++ (but not in Pascal or C), you can use the same name for more than one method as long as the number and/or types of parameters are unique.

In Java you *cannot* declare a variable x in a method if there is also a parameter named x, or another variable named x declared in an enclosing block or *for* loop. However, such declarations *are* allowed in C++. For example, the following is a legal C++ function, but not a legal Java method:

```
void f( int k ) {   // k is a parameter
   int k = 0;       // also a local variable

   while (...) {
      int k = 1;   // and another local variable, inside the loop
      ...
   }
}
```

In general, the scope rules of a language determine which declaration of a named object corresponds to each use. C++ and Java use what is called **static scoping**; that means that the correspondence between uses and declarations is made at compile time. C++ uses the "most closely nested" rule to match nested declarations to their uses: a use of variable x matches the declaration in the most closely enclosing scope such that the declaration precedes the use. In C++, there is one, outermost scope that includes all function names and the names of the global variables (the variables that are declared outside the functions). Each function has two or more scopes: one for the parameters, one for the function body, and possibly additional scopes for each *for* loop and each nested block (delimited by curly braces) in the function.

In the example given above, the outermost scope includes just the name "f", and function f itself has three (nested) scopes:

1. The outer scope for f just includes parameter k.
2. The next scope is for the body of f, and includes the variable k that is initialized to 0.
3. The innermost scope is for the body of the while loop, and includes the variable k that is initialized to 1.

So a use of variable k inside the while loop matches the declaration in the loop (has the value 1), while a use of k outside the loop (either before or after the loop) matches the declaration at the beginning of the function (has the value 0).

TEST YOURSELF #1

Question 1: Consider the names declared in the following code. For each, determine whether it is legal according to the rules used in Java.

```
class animal {
  // methods
  void attack(int animal) {
     for (int animal=0; animal<10; animal++) {
         int attack;
     }
  }

  int attack(int x) {
     for (int attack=0; attack<10; attack++) {
        int animal;
     }
  }

  void animal() { }

  // fields
  double attack;
  int attack;
  int animal;
}
```

Question 2: Consider the following C++ code. For each use of a name, determine which declaration it corresponds to (or whether it is a use of an undeclared name).

```
int k=10, x=20;

void foo(int k) {
    int a = x;
    int x = k;
    int b = x;
    while (...) {
        int x;
        if (x == k) {
           int k, y;
           k = y = x;
        }
        if (x == k) {
           int x = y;
        }
    }
}
```

Not all languages use static scoping. Lisp, APL, and Snobol use what is called **dynamic** scoping. A use of a variable that has no corresponding declaration in the same function corresponds to the declaration in the **most-recently-called still active** function. For example, consider the following code:

```
void main() {
  f1();
  f2();
}

void f1() {
  int x = 10;
  g();
}
```

```
void f2() {
  String x = "hello";
  f3();
  g();
}

void f3() {
  double x = 30.5;
}

void g() {
  print(x);
}
```

Under dynamic scoping this program outputs "10 hello". The first call to g comes from f1, whose copy of x has value 10. The next call to g comes from f2. Although f3 is called by f2 before it calls g, the call to f3 is not active when g is called; therefore, the use of x in g matches the declaration in f2, and "hello" is printed.

TEST YOURSELF #2

Assuming that dynamic scoping is used, what is output by the following program?

```
void main() {
  int x = 0;
  f1();
  g();
  f2();
}

void f1() {
  int x = 10;
  g();
}

void f2() {
  int x = 20;
  f1();
  g();
}

void g() {
  print(x);
}
```

It is generally agreed that dynamic scoping is a bad idea; it can make a program very difficult to understand, because a single use of a variable can correspond to many different declarations (with different types)! The languages that use dynamic scoping are all old languages; recently designed languages all use static scoping.

Another issue that is handled differently by different languages is whether names can be used before they are defined. For example, in Java, a method or field name *can* be used before the definition appears, but this is *not* true for a variable:

```
class Test {
  void f() {
    val = 0;   // field val has not yet been declared -- OK
    g();       // method g has not yet been declared  -- OK
    x = 1;     // variable x has not yet been declared -- ERROR!
    int x;
  }
```

```
  void g() {}
  int val;
}
```

In what follows, we will assume that we are dealing with a language that:

- uses static scoping
- requires that *all* names be declared before they are used
- does not allow multiple declarations of a name in the same scope (even for different kinds of names)
- *does* allow the same name to be declared in multiple nested scopes (but only once per scope)
- uses the same scope for a method's parameters and for the local variables declared at the beginning of the method

Symbol Table Implementations

In addition to the assumptions listed at the end of the previous section, we will assume that:

- The symbol table will be used to answer two questions:
    1. Given a declaration of a name, is there already a declaration of the same name in the current scope (i.e., is it multiply declared)?
    2. Given a use of a name, to which declaration does it correspond (using the "most closely nested" rule), or is it undeclared?
    3. The symbol table is only needed to answer those two questions (i.e., once all declarations have been processed to build the symbol table, and all uses have been processed to link each ID node in the abstract-syntax tree with the corresponding symbol-table entry, the symbol table itself is no longer needed).

Given these assumptions, the symbol-table operations we will need are:

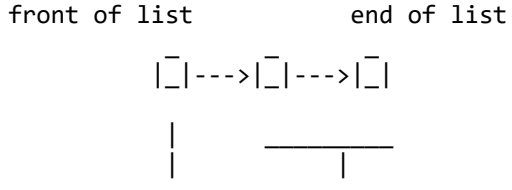
1. Look up a name in the current scope only (to check if it is multiply declared).
2. Look up a name in the current and enclosing scopes (to check for a use of an undeclared name, and to link a use with the corresponding symbol-table entry).
3. Insert a new name into the symbol table with its attributes.
4. Do what must be done when a new scope is entered.
5. Do what must be done when a scope is exited.

We will look at two ways to design a symbol table: a list of tables, and a table of lists. For each approach, we will consider what must be done when entering and exiting a scope, when processing a declaration, and when processing a use. To keep things simple, we will assume that each symbol-table entry includes only:

- the symbol name
- its type
- the nesting level of its declaration

Method 1: List of Hashtables

The idea behind this approach is that the symbol table consists of a list of hashtables, one for each currently visible scope. When processing a scope S, the structure of the symbol table is:



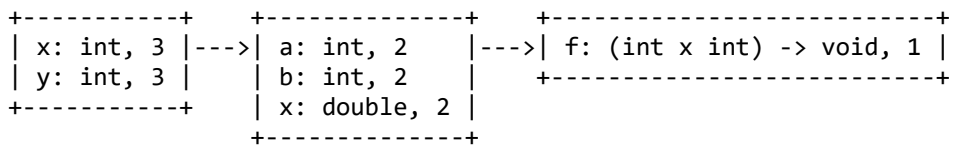
```
declarations   |
made in S      declarations made in scopes that enclose S; each hashtable
               in the list corresponds to one scope (i.e., contains all
               of the declarations for that scope)
```

For example, given this code:

```
void f(int a, int b) {
   double x;
   while (...) {
      int x, y;
      ...
   }

void g() {
   f();
}
```

After processing the declarations inside the while loop, the symbol table looks like this:



The declaration of method g has not yet been processed, so it has no symbol-table entry yet. Note that because f is a method, its type includes the types of its parameters (int x int), and its return type (void).

Here are the operations that need to be performed on scope entry/exit, and to process a declaration/use:

1. On scope entry: increment the current level number and add a new empty hashtable to the front of the list.
2. To process a declaration of x: look up x in the first table in the list. If it is there, then issue a "multiply declared variable" error; otherwise, add x to the first table in the list.
3. To process a use of x: look up x starting in the first table in the list; if it is not there, then look up x in each successive table in the list. If it is not in *any* table then issue an "undeclared variable" error.
4. On scope exit, remove the first table from the list and decrement the current level number.

Remember that method names need to go into the hashtable for the outermost scope (not into the same table as the method's variables). For example, in the picture above, method name f is in the symbol table for the outermost scope; name f is *not* in the same scope as parameters a and b, and variable x. This is so that when the use of name f in method g is processed, the name is found in an enclosing scope's table.

Here are the times required for each operation:

1. **Scope entry**: time to initialize a new, empty hashtable; this is probably proportional to the size of the hashtable.
2. **Process a declaration:** using hashing, constant expected time (O(1)).
3. **Process a use**: using hashing to do the lookup in each table in the list, the worst-case time is O(depth of nesting), when every table in the list must be examined.
4. **Scope exit**: time to remove a table from the list, which should be O(1) if garbage collection is ignored.

TEST YOURSELF #3

For all three questions below, assume that the symbol table is implemented using a list of hashtables.

Question 1: Recall that Java does not allow the same name to be used for a local variable of a method, and for another local variable declared inside a nested scope in the method body. Even with this restriction, it is not a

good idea to put *all* of a method's local variables (whether they are declared at the beginning of the method, or in some nested scope within the method body) in the *same* table. Why not?

Question 2: C++ does not use exactly the scoping rules that we have been assuming. In particular, C++ **does** allow a function to have both a parameter and a local variable with the same name (and any uses of the name refer to the local variable).

Consider the following code. Draw the symbol table as it would be after processing the declarations in the body of f under:

- the scoping rules we have been assuming
- C++ scoping rules

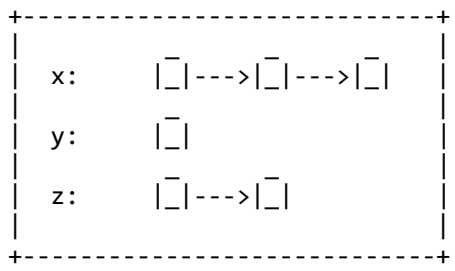
```
void g(int x, int a) { }

void f(int x, int y, int z) {
  int a, b, x;
  ...
}
```

Question 3: Which of the four operations (scope entry, process a declaration, process a use, scope exit) described above would change (and how would it change) if Java rules for name reuse were used instead of C++ rules (i.e., if the same name can be used within one scope as long as the uses are for different kinds of names, and if the same name *cannot* be used for more than one variable declaration in nested scopes)?

Method 2: Hashtable of Lists

The idea behind this approach is that when processing a scope S, the structure of the symbol table is:



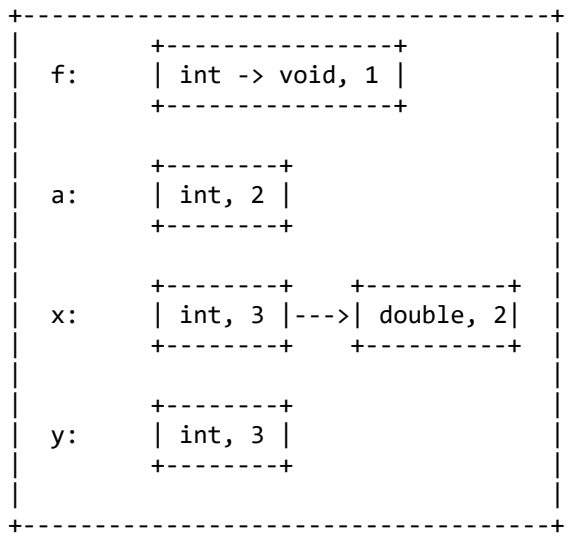
There is just one big hashtable, containing an entry for each variable for which there is some declaration in scope S or in a scope that encloses S. Associated with each variable is a list of symbol-table entries. The first list item corresponds to the most closely enclosing declaration; the other list items correspond to declarations in enclosing scopes.

For example, given this code:

```
void f(int a) {
   double x;
   while (...) {
      int x, y;
      ...
   }

void g() {
   f();
}
```

After processing the declarations inside the while loop, the symbol table looks like this:



Note that the level-number attribute stored in each list item enables us to determine whether the most closely enclosing declaration was made in the current scope or in an enclosing scope.

Here are the operations that need to be performed on scope entry/exit, and to process a declaration/use:

1. On scope entry: increment the current level number.
2. To process a declaration of x: look up x in the symbol table. If x is there, fetch the level number from the first list item. If that level number = the current level then issue a "multiply declared variable" error; otherwise, add a new item to the front of the list with the appropriate type and the current level number.
3. To process a use of x: look up x in the symbol table. If it is not there, then issue an "undeclared variable" error.
4. On scope exit, scan all entries in the symbol table, looking at the first item on each list. If that item's level number = the current level number, then remove it from its list (and if the list becomes empty, remove the entire symbol-table entry). Finally, decrement the current level number.

The required times for each operation are:

1. **Scope entry**: time to increment the level number, O(1).
2. **Process a declaration**: using hashing, constant expected time (O(1)).
3. **Process a use**: using hashing, constant expected time (O(1)).
4. **Scope exit**: time proportional to the number of names in the symbol table (or perhaps even the size of the hashtable if no auxiliary information is maintained to allow iteration through the non-empty hashtable buckets).

TEST YOURSELF #4

Assume that the symbol table is implemented using a hashtable of lists. Draw pictures to show how the symbol table changes as each declaration in the following code is processed.

```
void g(int x, int a) {
   double d;
   while (...) {
      int d, w;
      double x, b;
      if (...) {
         int a,b,c;
      }
   }
```

```
        while (...) {
           int x,y,z;
        }
     }
```

Type Checking

As mentioned in the Introduction, the job of the type-checking phase is to:

- Determine the type of each expression in the program (each node in the AST that corresponds to an expression).
- Find type errors.

The **type rules** of a language define how to determine expression types, and what is considered to be an error. The type rules specify, for every operator (including assignment), what types the operands can have, and what is the type of the result. For example, both C++ and Java allow the addition of an int and a double, and the result is of type double. However, while C++ also allows a value of type double to be assigned to a variable of type int, Java considers that an error.

TEST YOURSELF #5

List as many of the operators that can be used in a Java program as you can think of (don't forget to think about the logical and relational operators as well as the arithmetic ones). For each operator, say what types the operands may have, and what is the type of the result.

In addition to finding type errors caused by operators being applied to operands of the wrong type, the type checker must also find type errors having to do with expressions that, because of their **context** must be boolean, and type errors having to do with method calls. Examples of the first kind of error include:

- the condition of an *if* statement
- the condition of a *while* loop
- the termination condition part of a *for* loop

and examples of the second kind of error include:

- calling something that is not a method
- calling a method with the wrong number of arguments
- calling a method with arguments of the wrong types