Compilers: Principles, Techniques, & Tools

By Alfred V. Aho et all

First Edition

CONTENTS

CHAPTER 1

INTRODUCTION

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*.

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.

An *interpreter* is another common kind of language processor.

The task of collecting the source program is sometimes entrusted to a separate program, called a *preprocessor*.

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.

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.

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. 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.

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

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

that is 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.

2.2 Syntax Analysis

The second phase of the compiler is *syntax analysis* or *parsing*. 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.

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.

An important part of semantic analysis is *type checking*, where the compiler checks that each operator has matching operands.

The language specification may permit some type conversions called *coercions*.

2.4 Intermediate Code Generation

We consider an intermediate form called *three-address code*, which consists of a sequence of assembly-like instructions with three operands per instruction.

2.5 The Grouping of Phases into Passes

In an implementation, activities from several phases may be grouped together into a *pass* that reads an input file and writes an output file.

2.6 Compiler-Construction Tools

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.

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

3 The Evolution of Programming Language

3.1 The Move to Higher-Level Languages

One classification is by generation. *First-generation languages* are the machine languages, *second-generation* the assembly languages, and *third-generation* the higher-level languages. *Fourth-generation languages* are languages designed for specific applications. The term *fifth-generation language* has been applied to logic- and constraint-based languages.

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.

The term *von Neumann language* is applied to programming languages whose computational model is based on the von Neumann computer architecture.

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.

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

4 Applications of Compiler Technology

4.1 implementation of High-Level Programming Languages

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.

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).

4.2 Optimizations for Computer Architectures

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 *in-*

struction level, where multiple operations are executed simultaneously and at the *processor level*, where different threads of the same application are run on different processors.

5 Programming Language Basics

5.1 The Static/Dynamic Distinction

If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a *static* policy or that the issue can be decided at *compile time*. On the other hand, a policy that only allows a decision to be made when we execute the program is said to be a *dynamic policy* or to require a decision at *run time*.

The *scope* of a declaration of x is the region of the program in which uses of x refer to this declaration. A language uses *static scope* or *lexical scope* if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses *dynamic scope*.

5.2 Environments and States

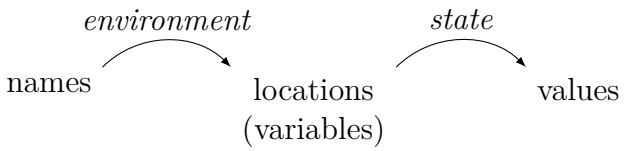


Figure 1.1: Two-stage mapping from names to values

The association of names with locations in memory (the *store*) and then with values can be described by two mappings that change as the program runs:

1. The *environment* is a mapping from names to locations in the store.
2. The *state* is a mapping from locations in store to their values.

The environment and state mappings in Fig. 1.1 are dynamic, but there are a few exceptions:

1. *Static versus dynamic binding* of names to locations.
2. *Static versus dynamic binding* of locations to values.

Names, Identifiers, and Variables

An *identifier* is a string of characters, typically letters or digits, that refers to (identifies) an entity. Composite names are called *qualified* names.

A *variable* refers to a particular location of the store.

5.3 Static Scope and Block Structure

The scope rules for C are based on program structure; the scope of a declaration is determined implicitly by where the declaration appears in the program. Later languages also provide explicit control over scopes through the use of keywords like **public**, **private** and **protected**.

A *block* is a grouping of declarations and statements. C uses braces `{` and `}` to delimit a block; the alternative use of **begin** and **end** for the same purpose dates back to Algol.

In C, the syntax of blocks is given by

1. One type of statement is a block. Blocks can appear anywhere that other types of statement can appear.

2. A block is a sequence of declarations followed by a sequence of statements, all surrounded by braces.

Note that this syntax allows blocks to be nested inside each other. This nesting property is referred to as *block structure*.

5.4 Explicit Access Control

Through the use of keywords like **public**, **private**, and **protected**, object-oriented languages provide explicit control over access to member names in a superclass. These keywords support *encapsulation* by restricting access.

5.5 Dynamic Scope

Technically, any scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes. The term *dynamic scope*, however, usually refers to the following policy: a use of a name x refers to the declaration of x in the most recently called procedure with such a declaration.

Declarations and Definitions

In C++, a method is declared in a class definition, by giving the types of the arguments and result of the method (often called the *signature* for the method).

5.6 Parameter Passing Mechanisms

Actual parameters (the parameters used in the call of a procedure) are associated with the *formal parameters* (those used in the procedure definition).

Call-by-Value

In *call-by-value*, the actual parameter is evaluated (if it is an expression) or copied (if it is a variable).

Call-by-Reference

In *call-by-reference*, the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter.

5.7 Aliasing

It is possible that two formal parameters can refer to the same location; such variables are said to be *aliases* of one another.

CHAPTER 2

A SIMPLE SYNTAX-DIRECTED TRANSLATOR

1 Introduction

The *syntax* of a programming language defines what its programs, while the *semantics* of the language defines what its program mean; that is, what each program does when it executes.

A lexical analyzer allows a translator to handle mutlicharacter constructs like identifiers, which are written as sequences of charactersm, but are treated as units called *tokens* during syntax analysis.

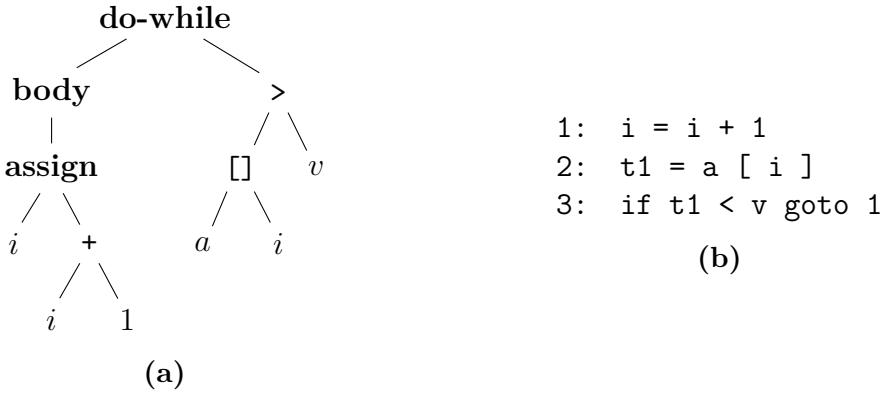


Figure 2.1: Intermediate code for "`do i=i+1; while(a[i]<v);`"

Two forms of intermediate code are illustrated in Fig. 2.1. One form, called *abstract syntax trees* or simply *syntax trees*, represents the hierarchical systematic structure of the source program.

2 Syntax Definition

A grammar naturally describes the hierarchical structure of most programming language constructs. For example, an if-else statement in Java can have the form

if (expression) statement **else** statement

Using the variable *expr* to denote an expression and the variable *stmt* to denote a statement, this structuring rule can be expressed as

$$stmt \rightarrow \textbf{if } (expr)\ stmt\ \textbf{else}\ stmt$$

in which the arrow may be read as "can have the form." Such a rule is called a *production*. In a production, lexical elements are called *terminals*. Variables like *expr* and *stmt* represent sequences of terminals and are called *nonterminals*.

2.1 Definition of Grammars

A *context-free grammar* has four components:

1. A set of *terminal* symbols, sometimes referred to as "tokens."
2. A set of *nonterminals*, sometimes called "syntactic variables."
3. A set of *productions*, where each production consists of a nonterminal, called the *head* or *left side* of the production, an arrow, and a sequence of terminals and/or nonterminals, called the *body* or *right side* of the production.
4. A designation of one of the nonterminals as the *start* symbol.

Tokens Versus Terminals

A token consists of two components, a token name and an attribute value. The token names are abstract symbols that are used by the parser for syntax analysis. Often, we shall call these token names *terminals*, since they appear as terminal symbols in the grammar for a programming language.

We say a production is *for* a nonterminal if the nonterminal is the head of the production. The string of zero terminals, written as ϵ, is called the *empty* string.

2.2 Derivations

The terminal strings that can be derived from the start symbol form the *language* defined by the grammar.

Parsing is the problem of taking a string of terminals and figuring out how to derive it from the start symbol of the grammar, and if it cannot be derived from the start symbol of the grammar, then reporting syntax errors within the string.

2.3 Parse Trees

A parse tree pictorially shows how the start symbol of a grammar derives a string in the language.

Formally, given a context-free grammar, a *parse tree* according to the grammar is a tree with the following properties:

1. The root is labeled by the start symbol.

2. Each leaf is labeled by a terminal or by ϵ.

3. Each interior node is labeled by a nonterminal.

4. If A is the nonterminal labeling some interior node and $X_1, X_2, \cdots, X_n$ are the labels of the children of that node from left to right, then there must be a production $A \to X_1 X_2 \cdots X_n$.

Tree Terminology

Tree data structures figure prominently in compiling.

- A tree consists of one or more *nodes*. Nodes may have *labels*.
- Exactly one node is the *root*. All nodes except the root have a unique *parent*; the root has no parent.
- If node N is the parent of node M, then M is a *child* of N. The children of one node are called *siblings*. They have an order, *from the left*, and when we draw trees, we order the children
- A node with no children is called a *leaf*. Other nodes – those with one or more children – are *interior nodes*.
- A *descendant* of a node N is either N itself, a child of N, a child of a child of N, and so on, for any number of levels. We say node N is an *ancestor* of node M if M is a descendant of N.

From left to right, the leaves of a parse tree form the *yield* of the tree, which is the string *generated* or *derived* from the nonterminal at the root of the parse tree.

The process of finding a parse tree for a given string of terminals is called *parsing* that string.

2.4 Ambiguity

We have to be careful in talking about *the* structure of a string according to a grammar. A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be *ambiguous*.

2.5 Associativity of Operators

We say that the operator + *associates* to the left, because an operand with plus signs on both sides of it belongs to the operator to its left.

2.6 Precedence of Operators

We say that $*$ has *higher precedence* than + if $*$ takes its operands before + does.

3 Syntax-Directed Translation

This section introduces two concepts related to syntax-directed translation:

- *Attributes.* An *attribute* is any quantity associated with a programming construct.
- (*Syntax-directed*) *translation schemes.* A *translation scheme* is a notation for attaching program fragments to the productions of a grammar.

3.1 Postfix Notation

The *postfix notation* for an expression E can be defined inductively as follows:

1. If E is a variable or constant, then the postfix notation for E is E itself.
2. If E is an expression of the form E_1 **op** E_2, where **op** is any binary operator, then the postfix notation for E is E_1' E_2' **op**, where E_1' and E_2' are the postfix notations for E_1 and E_2, respectively.
3. If E is a parenthesized expression of the form (E_1), then the postfix notation for E is the same as the postfix notation for E_1.

No parentheses are needed in postfix notation, because the position and *arity* (number of arguments) of the operators permits only one decoding of a postfix expression.

3.2 Synthesized Attributes

A *syntax-directed definition* associates:

1. With each grammar symbol, a set of attributes, and
2. With each production, a set of *semantic rules* for computing the values of the attributes associated with the symbols appearing in the production.

A parse tree showing the attribute values at each node is called an *annotated* parse tree.

An attribute is said to be *synthesized* if its value at a parse-tree node N is determined from attribute values at the children of N and at N itself.

3.3 Tree Traversals

A *traversal* of a tree starts at the root and visits each node of the tree in some order.

A *depth-first* traversal starts at the root and recursively visits the children of each node in any order, not necessarily from left to right.

Synthesized attributes can be evaluated during any *bottom-up* traversal, that is, a traversal that evaluates attributes at a node after having evaluated attributes at its children.

3.4 Translation Schemes

Preorder and Postorder Traversals

Often, we traverse a tree to perform some particular action at each node. If the action is done when we first visit a node, then we may refer to the traversal as a *preorder traversal*. Similarly, if the action is done just before we leave a node for the last time, then we say it is a *postorder traversal* of the tree.

The *preorder* of a (sub)tree rooted at node N consists of N, followed by the preorders of the subtrees of each of its children, if any, from the left. The *postorder* of a (sub)tree rooted at N consists of the postorders of each of the subtrees for the children of N, if any, from the left, followed by N itself.

Program fragments embedded within production bodies are called *semantic actions*.

4 Parsing

Most parsing methods fall into one of two classes, called the *top-down* and *bottom-up* methods.

4.1 Top-Down Parsing

The current terminal being scanned in the input is frequently referred to as the *lookahead* symbol.

4.2 Predictive Parsing

Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input. Here, we consider a simple form of recursive-descent parsing, called *predictive parsing*, in which the lookahead symbol unambiguously determines the flow of control through the procedure body for each nonterminal.

4.3 Designing a Predictive Parser

Recall that a *predictive parser* is a program consisting of a procedure for every nonterminal.

4.4 Left Recursion

Consider a nonterminal A with tow productions

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

where α and β are sequences of terminals and nonterminals that do not start with A.

The nonterminal A and its production are said to be *left recursive*, because the production $A \rightarrow A\alpha$ has A itself as the leftmost symbol on the right side.

The same effect can be achieved by rewriting the productions for A in the following manner, using a new nonterminal R:

$$A \rightarrow \beta R$$
$$R \rightarrow \alpha R \mid \epsilon$$

Nonterminal R and its production $R \rightarrow \alpha R$ are *right recursive* because this production for R has R itself as the last symbol on the right side.

5 A Translator for Simple Expressions

5.1 Abstract and Concrete Syntax

In an *abstract syntax tree* for an expression, each interior node represents an operator; the children of the node represent the operands of the operator.

Abstract syntax trees, or simply *syntax trees*, resemble parse trees to an extent. Many nonterminals of a grammar represent programming constructs, but others are "helpers" of one sort of another. In the syntax tree, these helpers typically are not needed and are hence droped. To emphasize the contrast, a parse tree is sometimes called a *concrete syntax tree*, and the underlying grammar is called a *concrete syntax* for the language.

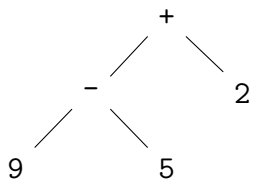


Figure 2.2: Syntax tree for `9-5+2`

In the syntax tree in Fig. 2.2, each interior node is associated with an operator, with no "helper" nodes for *single productions* (a production whose body consists of a single nonterminal, and nothing else) or for ϵ-productions.

5.2 Simplifying the Translator

When the last statement executed in a procedure body is a recursive call to the same procedure, the call is said to be *tail recursive*.

5.3 The Complete Program

```
import java.io.*
class Parser {
    static int lookahead;

    public Parser() throws IOException {
        lookahead = System.in.read();
    }

    void expr() throws IOException {
```

```
            term();
            while(true) {
                if (lookahead == '+') {
                    match('+'); term(); System.out.write('+');
                }
                else if (lookahead == '-') {
                    match('-'); term(); System.out.write('-');
                }
                else return;
            }
        }

        void term() throws IOException {
            if (Character.isDigit((char)lookahead)) {
                System.out.write((char)lookahead); match(lookahead);
            }
            else throw new Error("syntax error");
        }
    }

    public class Postfix {
        public static void main(String[] args) throws IOException {
            Parser parse = new Parser();
            parse.expr(); System.out.write('\n');
        }
    }
```

Figure 2.3: Java program to translate infix expressions into postfix form

The function `Parser`, with the same name as its class, is a *constructor*; it is called automatically when an object of the class is created.

The construction `(char)lookahead` *casts* or coerces `lookahead` to be a character.

6 Lexical Analysis

A sequence of input characters that comprises a single token is called a *lexeme*.

6.1 Recognizing Keywords and Identifiers

Most languages use fixed character strings as punctuation marks or to identify consturcts. Such character strings are called *keywords*.

Keywords generally satisfy the rules for forming identifiers, so a mechanism is needed f deciding when a lexeme forms a keyword and when it forms an identifier. The problem is easier to resolve if keywords are *reserved*; i.e., if they cannot be used as identifiers.

The lexical analyzer in this section solves two problems by using a table to hold character strings:

- *Single Representation.*
- *Reserved Words.*

In Java, a string table can be implemented as a hash table using a class called *Hashtable*.

7 Symbol Tables

Symbol tables are data structures that are used by compilers to hold information about source-program constructs.

7.1 Symbol Table Per Scope

The term *scope* by itself refers to a portion of a program that is the scope of one or more declarations.

The *most-closely nested* rule for blocks is that an identifier x is in the scope of the most-closely nested declaration of x; that is, the declaration of x found by examining blocks inside-out, starting with the block in which x appears.

8 Intermediate Code Generation

8.1 Two Kinds of Intermediate Representations

In addition to creating an intermediate representation, a compiler front end checks that the source program follows the syntactic and semantic rules of the source language. This checking is called *static checking*; in general "static" means "done by the compiler."

8.2 Static Checking

Static checking includes:

- *Syntactic Checking.*
- *Type Checking.*

L-values and R-values

The terms *l-value* and *r-value* refer to values that are appropriate on the left and right sides of an assignment, respectively.

Type Checking

Type checking assures that the type of a construct matches that expected by its context.

The idea of matching actual with expected types continues to apply, even in the following situations:

- *Coercions.* A *coercion* occurs if the type of an operand is automatically converted to the type expected by the operator.
- *Overloading.* A symbol is said to be *overloaded* if it has different meanings depending on its context.

CHAPTER 3

LEXICAL ANALYSIS

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-analysis generator* and compiling those patterns into code that functions as a lexical analyzer.

1 The Role of the Lexical Analyzer

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).

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.

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

1.1 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.
- 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.

2 Specification of Tokens

Can We Run Out of Buffer Space

To avoid problems with long character strings, we can treat them as a concatenation of components, one from each line over which the string is written.

A more difficult problem occurs when arbitrarily long lookahead may be needed. For example, some languages like PL/I do not treat keywords as *reserved*; that is, you can use identifiers with the same name as a keyword.

2.1 Strings and Languages

An *alphabet* is any finite set of symbols. The set $\{0, 1\}$ is the *binary alphabet*.

A *string* over an alphabet is a finite sequence of symbols drawn from that alphabet. The *empty string*, denoted ϵ, is the string of length zero.

A *language* is any countable set of strings over some fixed alphabet. Abstract languages like $\emptyset$, the *empty set* are languages under this definition.

Terms for Parts of Strings

The following string-related terms are commonly used:

1. A *prefix* of string s is any string obtained by removing zero or more symbols from the end of s.
2. A *suffix* of string s is any string obtained by removing zero or more symbols from the beginning of s.
3. A *substring* of s is obtained by deleting any prefix and any suffix from s.
4. The *proper* prefixes, suffixes, and substrings of a string s are those, prefixes, suffixes, and substrings, respectively, of s that are not ϵ or not equal to s itself.
5. A *subsequence* of s is any string formed by deleting zero or more not necessarily consecutive positions of s.

If x and y are strings, then the *concatenation* of x and y, denoted xy, is the string formed by appending y to x.

2.2 Operations and Languages

The (*Kleene*) *closure* of a language L, denoted L^*, is the set of strings you get by concatenating L zero or more times.

2.3 Regular Expressions

A language that can be defined by a regular expression is called a *regular set*. If two regular expressions r and s denote the same regular set, we say they are *equivalent* and write $r = s$.

2.4 Regular Definitions

If Σ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

$$\begin{array}{lcl} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ & \cdots & \\ d_n & \rightarrow & r_n \end{array}$$

where:

1. Each d_i is a new symbol, not in Σ and not the same as any other of the d's, and
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

2.5 Extensions of Regular Expressions

Here we mention a few notational extensions that were first incorporated into Unix utilities that are particularly useful in the specification lexical analyzer.

1. *One or more instances.*
2. *Zero or one instance.*
3. *Character classes.*

3 Recognition of Tokens

3.1 Transition Diagrams

Transition diagrams have a collection of nodes or circles, called *states*.

Edges are directed from one state of the transition diagram to another. Each edge is *labeled* by a symbol or set of symbols. We shall assume that all our transition diagrams are *deterministic*, meaning that there is never more than one edge out of a given state with a given symbol among its labels. Some important conventions about transition diagrams are:

- Certain states are said to be *accepting*, or *final*.
- One state is designated the *start state*, or *initial state*; it is indicated by an edge, labeled "start," entering from nowhere.

3.2 Recognition of Reserved Words and Identifiers

Usually, keywords are reserved, so they are not identifiers even though they `look` like identifiers.

4 The Lexical-Analyzer Generator `Lex`

The input notation for the `Lex` tool is referred to as the *Lex language* and the tool itself is the *Lex compiler*.

4.1 Structure of `Lex` Programs

A `Lex` program has the following form:

declarations
%%
transition rules
%%
auxiliary functions

The declarations section includes declarations of variables, *manifest constants* (identifiers declared to stand for a constant).

5 Finite Automata

We shall now discover how `Lex` turns its input program into a lexical analyzer. At the heart of the transition is the formalism known as *finite automata*. These are essentially graphs, like transition diagrams, with a few differences:

1. Finite automata are *recognizers*; they simply say "yes" or "no" about each possible input string.
2. Finite automata come in two flavors:
   (a) *Nondeterministic finite automata* (NFA) have no restrictions on the labels of their edges.
   (b) *Deterministic finite automata* (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

Both deterministic and nondeterministic finite automata are capable of recognizing the same languages. In fact these languages are exactly the same languages, called the *regular languages*, the regular expressions can describe.[1]

5.1 Nondeterministic Finite Automata

A *nondeterministic finite automata* (NFA) consists of:

1. A finite set of state S.
2. A set of input symbols Σ, the *input alphabet*.

[1] There is a small lacuna: as we defined them, regular expressions cannot describe the empty language, since we would never want to use this pattern in practice. However, finite automata *can* define the empty language.

3. A *transition function* that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of *next states*.

4. A state s_0 from S that is distinguished as the *start state* (or *initial state*).

5. A set of states F, a subset of S, that is distinguished as the *accepting states* (or *final states*).

We can represent either an NFA or DFA by a *transition graph*, where the nodes are states and the labeled edges represent the transition function.

5.2 Transition Tables

We can also represent an NFA by a *transition table*, whose rows correspond to states, and whose columns correspond to the input symbols and ϵ.

5.3 Acceptance of Input Strings by Automata

An NFA *accepts* input string x if and only if there is some path in the transition graph from the start state to one of the accepting staets, such that the symbols along the path spell out x.

The *language defined* (or *accepted*) by an NFA is the set of strings labeling some path from the start to an accepting state.

5.4 Deterministic Finite Automata

A *deterministic finite automata* (DFA) is a special case of an NFA where:

1. There are no moves on input ϵ, and

2. For each state s and input symbol a, there is exactly one edge out of s labeled a.

Algorithm: Simulating a DFA.
INPUT: An input string x terminated by an end-of-file character **eof**. A DFA D with start state s_0, accepting states F, and transition function *move*.
OUTPUT: Answer "yes" if D accepts x; "no" otherwise.
METHOD: Apply the algorithm in Fig. 3.1 to the input string x. The function $move(s, c)$ gives the state to which there is an edge from state s on input c. The function *nextChar* returns the next character of the input string x.

```
s = s0;
c = nextChar();
while ( c != eof ) {
    s = move(s, c);
    c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```

Figure 3.1: Simulating a DFA

6 From Regular Expressions to Automata

6.1 Conversion of an NFA to a DFA

Algorithm: The *subset construction* of a DFA from an NFA.
INPUT: An NFA N.
OUTPUT: A DFA D accepting the same language as N.
METHOD: Our algorithm constructs a transition table $Dtran$ for D. Each state of D is a set of NFA states, and we construct $Dtran$ so D will simulate "in parallel" all possible moves N can make on a given input string. Our first problem is to deal with ϵ-transitions of N properly. In Fig. 3.2 we see the definitions of several functions that describe basic computations on the states of N that are needed in the algorithm. Note that s is a single state of N, while T is a set of states of N.

We must explore those sets of states that N can be in after seeing some input string. As a basis, before reading the first input symbol, N can be in any of the states of ϵ-$closure(s_0)$, where s_0 is its start state. For the induction, suppose that N can be in set of states T after reading input string x. If it next reads input a, then N can immediately go to any of the states in $move(T, a)$. However, after reading a, it may also make several ϵ-transitions; thus N could be in any state of ϵ-$closure(move(T, a))$ after reading input xa. Following these ideas, the construction of the set of D's states, $Dstates$, and its transition function $Dtran$, is shown in Fig. 3.3.

The start state of D is ϵ-$closure(s_0)$, and the accepting states of D are all those sets of N's states that include at least one accepting state of N. To complete our description of the subset construction, we need only to show how initially, ϵ-$closure(s_0)$ is the only state in $Dstates$, and it is unmarked; ϵ-$closure(T)$ is computed for any set of NFA states T. This process, shown in Fig. 3.4, is a straightforward search in graph from a set of states. In this case, imagine that only the ϵ-labeled edges are available in the graph.

Operation	Description
ϵ-$closure(s)$	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-$closure(T)$	Set of NFA states reachable from NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T} \epsilon$-$closure(s)$.
$move(T, a)$	Set of NFA states to which there is transition on input symbol a from some table s in T.

Figure 3.2: Operations on NFA states

```
while ( there is an unmarked state T in Dstates ) {
    mark T;
    for ( each input symbol a ) {
        U = ε-closure(move(T, a));
        if ( U is not in Dstates )
            add U as an unmarked state to Dstates;
        Dtran[T, a] = U;
    }
}
```

Figure 3.3: The subset construction

```
push all states of T onto stack;
initialize ε-closure(T) to T;
while ( stack is not empty ) {
    pop t, the top element, off stack;
    for ( each state u with an edge from t to u labeled ε )
        if ( u is not in ε-closure(T) ) {
            add u to ε-closure(T);
            push u onto stack;
        }
}
```

Figure 3.4: Computing ϵ-$closure(T)$

6.2 Simulation of an NFA

Algorithm: Simulating an NFA.
INPUT: An input string x terminated by an end-of-file character **eof**. An NFA N with start state s_0, accepting states F, and transition function $move$.
OUTPUT: Answer "yes" if M accepts x; "no" otherwise.
METHOD: The algorithm keeps a set of current states S, those that are reached from s_0 following a path labeled by the inputs read so far. If c is the next input character, read by function $nextChar()$, then we first compute $move(S, c)$ and then close that set using $\epsilon - closure()$. The algorithm is sketched in Fig 3.5.

```
1)  S = ε-closure(s0);
2)  c = nextChar();
3)  while ( c != eof ) {
4)      S = ε-closure(move(S, c));
5)      c = nextChar();
6)  }
7)  if ( S ∩ F != ∅ ) return "yes";
8)  else return "no";
```

Figure 3.5: Simulating an NFA

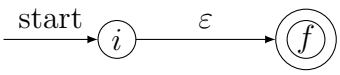
6.3 Efficiency of NFA Simulation

Big-Oh Notation

Technically, we say a function $f(n)$, perhaps the running time of some step of an algorithm, is $O(g(n))$ if there are constants c and n_0, such that whenever $n \geq n_0$, it is true that $f(n) \leq cg(n)$. The use of this *big-oh notation* enables us to avoid getting too far into the details of what we count as a unit of execution time, yet lets us express the rate at which the running time of an algorithm grows.

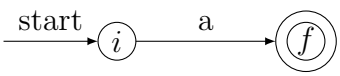
6.4 Construction of an NFA from a Regular Expression

Algorithm: The McNaughton-Yamada-Thompson algorithm to convert a regular expression to an NFA.
INPUT: A regular expression r over alphabet Σ.
OUTPUT: An NFA N accepting $L(r)$.
METHOD: Begin by parsing r into its constituent subexpressions. The rules for constructing an NFA consist of basic rules for handling subexpressions with no operators, and inductive rules for constructing larget NFA's from the NFA's for the immediate subexpressions of a given expression.
BASIS: For expression ϵ construct the NFA



Here, i is a new state, the start state of this NFA, and f is another new state, the accepting state for the NFA.

For any subexpression a in Σ, construct the NFA



where again i and f are new states, the start and accepting states, respectively. Note that in both of the basis constructions, we construct a distinct NFA, with new states, for every occurrence of ϵ or some a as a subexpression of r.
INDUCTION: Suppose $N(s)$ and $N(t)$ are NFA's for regular expression s and t, respectively.

a) Suppose $r = s|t$. Then $N(r)$, the NFA for r, is constructed as in Fig. 3.6. Here, i and f are new states, the start and accepting states of $N(r)$, respectively. There are ϵ-transitions from i to the start states of $N(s)$ and $N(t)$, and each of their accepting states have ϵ-transitions to the accepting state f. Note that the accepting states of $N(s)$ and $N(t)$ are not accepting in $N(r)$. Since any path from i to f must pass through either $N(s)$ or $N(t)$ exclusively, and since the label of that path is not changed by the ϵ's leaving i or entering f, we conclude that $N(r)$ accepts $L(s) \cup L(t)$, which is the same as $L(r)$. That is, Fig. 3.6 is a correct construction for the union operator.

b) Suppose $r = st$. Then construct $N(r)$ as in Fig. 3.7. The start state of $N(s)$ becomes the start state of $N(r)$, and the accepting state of $N(t)$ is the only accepting state of $N(r)$. The accepting state of $N(s)$ and the start state of $N(t)$ are merged into a single state, with all the transitions in or out of either state. A path from i to f in Fig. 3.7 must go first through $N(s)$, and therefore its label will begin with some string in $L(s)$. The path then continues through $N(t)$, so the path's label finishes with a string in $L(t)$. Accepting states never have edges out and start states never have edges in, so it is not possible for a path to re-enter $N(s)$ after leaving it. Thus $N(r)$ accepts exactly $L(s)L(t)$, and it is a correct NFA for $r = st$.

c) Suppose $r = s^*$. Then for r we construct the NFA $N(r)$ shown in Fig. 3.8. Here, i and f are new states, the start state and lone accepting state of $N(r)$. To get from i to f, we can either follow the introduced path labeled ϵ, which takes care of the one string in $L(s)^0$, or we can go to the start state of $N(s)$, through that NFA, then from its accepting state back to its start state zero or more times. These options allow $N(r)$ to accept all the strings in $L(s)^1$, $L(s)^2$, and so on, so the entire set of strings accepted by $N(r)$ is $L(s^*)$.

d) Finally, suppose $r = (s)$. Then $L(r) = L(s)$, and we can use the NFA $N(s)$ as $N(r)$.

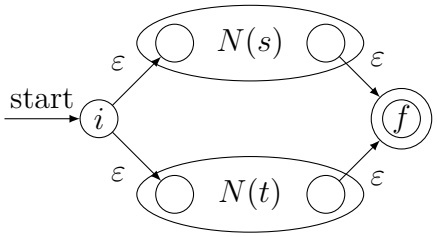


Figure 3.6: NFA for the union of two regular expressions

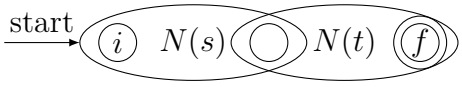


Figure 3.7: NFA for the concatenation of two regular expressions

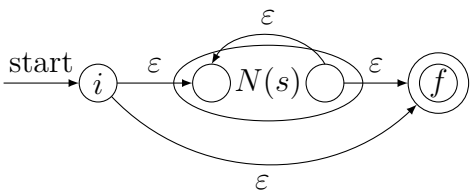


Figure 3.8: NFA for the closure of a regular expression

7 Design of a Lexical-Analyzer Generator

7.1 DFA's for Lexical Analyzers

We simulate the DFA until at some point there is no next state (or strictly speaking, the next state is $\varnothing$, the *dead state* corresponding to the empty set of NFA states).

8 Optimization of DFA-Based Pattern Matchers

8.1 Important States of an NFA

We call a state of an NFA *important* if it has a non-ϵ out-transition.

By concatenating a unique right endmarker # to a regular expression r, we give the accepting state for r a transition on #, making it an important state of the NFA for $(r)\#$. In other words, by using the *augmented* regular expression $(r)\#$, we can forget about accepting states as the subset construction proceeds; when the construction is complete, any state with a transition on # must be an accepting state.

It is useful to present the regular expression by its *syntax tree*, where the leaves correspond to operands and the interior ndes correspond to operators. An interior node is called a *cat-node*, *or-node*, or *star-node* if it is labeled by the concatenation operator (dot), union operator |, or star operator $*$, respectively.

Leaves in a syntax tree are labeled by ϵ or by an alphabet symbol. To each leaf not labeled ϵ, we attach a unique integer. We refer to this integer as the *position* of the leaf and also as a position of its symbol.

8.2 Converting a Regular Expression Directly to a DFA

Algorithm: Construction of a DFA from a regular expression r.
INPUT: A regular expression r.
OUTPUT: A DFA D that recognizes $L(r)$.
METHOD:

1. Construct a syntax tree T from the augmented regular expression $(r)\#$.
2. Compute *nullable*, *firstpos*, *lastpos*, and *followpos* for T.
3. Construct *Dstates*, the set of states of DFA D, and *Dtran*, the transition function for D, by the procedure of Fig. 3.9. The states of D are sets of positions in T. Initially, each state is "unmarked," and a state becomes "marked" just before we consider its out-transitions. The start state of D is $firstpos(n_0)$, where node n_0 is the root of T. The accepting states are those containing the position for the endmarker symbol #.

```
initialize Dstates to contain only the unmarked state firstpos(n0),
    where n0 is the root of syntax tree T for (r)#;
while ( there is an unmarked state S in Dstates ) {
    mark S;
    for ( each input symbol a ) {
        let U be the union of followpos(p) for all p
            in S that correspond to a;
        if ( U is not in Dstates )
            add U as an unmarked state to Dstates;
        Dtran[S, a] = U;        }
}
```

Figure 3.9: Construction of a DFA directly from a regular expression

8.3 Minimizing the Number of States of a DFA

We shall say that two automata are *the same up to the state names* if one can be transformed into the other by doing nothing more than changing the names of states.

We say that string x *distinguishes* state s from state t if exactly one of the states reached from s and t by following the path with label x is an accepting state. State s is *distinguishable* from state t if there is some string that distinguishes them.

Algorithm: Minimizing the number of states of a DFA.
INPUT: A DFA D with set of states S, input alphabet Σ, start state s_0, and set of accepting states F.
OUTPUT: A DFA D' accepting the same language as D and having as few states as possible.
METHOD:

1. Start with an initial partition Π with two groups, F and $S - F$, the accepting and nonaccepting states of D.
2. Apply the procedure of Fig. 3.10 to construct a new partition Π_{new}.
3. If $\Pi_{\text{new}} = \Pi$, let $\Pi_{\text{final}} = \Pi$ and continue with step (4). Otherwise, repeat step (2) with Π_{new} in place of Π.
4. Choose one state in each group of Π_{final} as the *representative* for that group. The representatives will be the states of the minimum-state DFA D'. The other components of D' are constructed as follows:
   (a) The start state of D' is the representative of the group containing the start state of D.
   (b) The accepting states of D' are the representatives of those groups that contain an accepting state of D. Note that each group contains either only accepting states, or only nonaccepting states, because we started by separating those two classes of states, and the procedure of Fig. 3.10 always forms new groups that are subgroups of previously constructed groups.

(c) Let s be the representative of some group G of Π_{final}, and let the transition of D from s on input a be to state t. Let r be the representative of t's group H. Then in D', there is a transition from s to r on input a. Note that in D, every state in group G must go to some state of group H on input a, or else, group G would have been split according to Fig. 3.10.

initially, let $\Pi_{\text{new}} = \Pi$;
for (each group G of Π) {
    partition G into subgroups such that two states s and t
        are in the same subgroup if and only if for all input symbols a, states s and t have transitio
        to states in the same group of Π;
    /* at worst, a state will be in a subgroup by itself */
    replace G in Π_{new} by the set of all subgroups formed;
}

Figure 3.10: Construction of Π_{new}

CHAPTER 4

SYNTAX ANALYSIS

1 Introduction

1.1 Representative Grammars

Associativity and precedence are captured in the following grammar. 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:

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

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

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

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

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

1.2 Syntax Error Handling

Common programming errors can occur at many different levels.

- *Lexical* errors include misspelling of identifiers, keywords, or operators and missing quotes around text intended as a string.
- *Syntactic* errors include misplaced semicolons or extra missing braces; that is, "{" or "}".
- *Semantic* errors include type mismatches between operators and operands.
- *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 ==.

Several parsing 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.

1.3 Error-Recovery Strategies

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.

2 Context-Free Grammars

2.1 The Formal Definition of a Context-Free Grammar

A context-free grammar (grammar for short) consists of terminals, nonterminals, a start symbol, and productions.

1. *Terminals* are the basic symbols from which strings are formed.
2. *Nonterminals* are syntactic variables that denote sets of strings.
3. In a grammar, one nonterminal is distinguished as the *start symbol*, and the set of strings it denotes is the language generated by the grammar.
4. Each *production* consists of:
   (a) A nonterminal called the *head* or *left side* of the production; this production defines some of the strings denoted by the head.
   (b) The symbol $\rightarrow$.
   (c) A *body* or *right side* consisting of zero or more terminals and nonterminals.

2.2 Derivations

Bottom-up parsing is related to a class of derivations known as "rightmost" derivations, in which the rightmost nonterminal is rewritten at each step.

For example, consider the following grammar, with a single nonterminal E, which adds a production $E \rightarrow -E$ to the grammar (4.3):

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid \mathbf{id} \tag{4.4}$$

We can take a single E and repeatedly apply productions in any order to get a sequence of replacements. For example,

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

We call such a sequence of replacements a *derivation* of $-(\mathbf{id})$ from E.

When a sequence of derivation steps $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ rewrites α_1 to α_n, we say α_1 *derives* α_n.

If $S \stackrel{*}{\Rightarrow} \alpha$, where S is the start symbol of a grammar G, we say that α is a *sentential form* of G. A *sentence* of G is a sentential form with no nonterminals. The *language generated by* a grammar is its set of sentences. A language that can be generated by a grammar is said to be a *context-free language*. If two grammars generate the same language, the grammars are said to be *equivalent*.

To understand how parsers work, we shall consider derivations in which the nonterminal to be replaced at each step is chosen as follows:

1. In *leftmost* derivations, the leftmost nonterminal in each sentential is always chosen.

2. In *rightmost* derivations, the rightmost nonterminal is always chosen; we write $\alpha \underset{rm}{\Rightarrow} \beta$ in this case.

If $S \underset{lm}{\stackrel{*}{\Rightarrow}} \alpha$, then we say that α is a *left-sentential form* of the grammar at hand.

Rightmost derivations are sometimes called *canonical* derivations.

2.3 Parse Trees and Derivations

The leaves of a parse tree are labeled by nonterminals or terminals and, read from left to right, constitute a sentential form, called the *yield* or *frontier* of the tree.

2.4 Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be *ambiguous*.

For most parsers, it is desirable that the grammar be made unambiguous, for if it is not, we cannot uniquely determine which parse tree to select for a sentence. In other cases, it is convenient to use carefully chosen ambiguous grammars, together with *disambiguating rules* that "throw away" undesirable parse trees, leaving only one tree for each sentence.

3 Writing a Grammar

3.1 Elimination of Left Recursion

A grammar is *left recursive* if it has a nonterminal A such that there is a derivation $A \stackrel{+}{\Rightarrow} A\alpha$ for some string α.

> **Algorithm:** Eliminating left recursion.
> **INPUT:** Grammar G with no cycles or ϵ-productions.
> **OUTPUT:** An equivalent grammar with no left recursion.
> **METHOD:** Apply the algorithm in Fig. 4.1 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. ■

```
1)   arrange the nonterminals in some order A_1, A_2, ..., A_n.
2)   for ( each i from 1 to n ) {
3)       for ( each j from 1 to i - 1 ) {
4)           replace each production of the form A_i → A_j γ by the
               production A_i → δ_1 γ | δ_2 γ | ··· | δ_k γ, where
               A_j → δ_1 | δ_2 | ··· | δ_k are all current A_j-productions
5)       }
6)       eliminate the immediate left recursion among the A_i-productions
7)   }
```

Figure 4.1: Algorithm to eliminate left recursion from a grammar

3.2 Left Factoring

Algorithm: Left factoring a grammar.
INPUT: Grammar G.
OUTPUT: An equivalent left-factored grammar.
METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ – i.e., there is a nontrivial common prefix – replace all of the A-productions $A \to \alpha\beta_1 | \alpha\beta_2 | \cdots | \alpha\beta_n | \gamma$, where γ represents all alternatives that do not begin with α, by

$$A \to \alpha A' \mid \gamma$$
$$A' \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

3.3 LL(1) Grammars

Algorithm: Construction of a predictive parsing table.
INPUT: Grammar G.
OUTPUT: Parsing table M.
METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(A), add $A \to \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table).

3.4 Nonrecursive Predictive Parsing

The behavior of the parser can be described in terms of its *configurations*, which give the stack contents and the remaining input. The next algorithm describes how configurations are manipulated.

Algorithm: Table-driven predictive parsing.
INPUT: A string w and a parsing table M for grammar G.
OUTPUT: If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.
METHOD: Initially, the parser is in a configuration with $w\$$ in the input buffer and the startt symbol S of G on top of the stack, above \$. The program in Fig. 4.2 uses the predictive parsing table M to produce a predictive parse for the input. ■

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
    if ( X is a ) pop the stack and advance ip;
    else if ( X is a terminal ) error();
    else if ( M[X, a] is an error entry ) error();
    else if ( M[X, a] = X → Y1Y2 ··· Yk ) {
        output the production X → Y1Y2 ··· Yk;
        pop the stack;
        push Yk, Yk−1, ..., Y1 onto the stack, with Y1 on top;
    }
    set X to the top stack symbol;
}
```

Figure 4.2: Predictive parsing algorithm

4 Bottom-Up Parsing

4.1 Reductions

At each *reduction* step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.

Formally, if $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$, then production $A \to \beta$ in the position following α is a *handle* of $\alpha\beta w$.

4.2 Shift-Reduce Parsing

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.

4.3 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*).

5 Introduction to LR Parsing: Simple LR

5.1 Items and the LR(0) Automaton

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.

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*.

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 $S' \to S$.

Closure of Item Sets

We divide all the sets of items of interest into two classes:

1. *Kernel items*: the initial term, $S' \to \cdot S$, 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 $S' \to \cdot S$.

5.2 The LR-Parsing Algortihm

Where a shift-reduce parser would shift a symbol, an LR parser shifts a *state*.

LR-Parser Configurations

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.

Behavior of the LR Parser

Algorithm: 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 parset 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. 4.3. ■

```
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 4.3: LR-parsing program

5.3 Constructing SLR-Parsing Tables

Algorithm: Constructing an SLR-parsing table.
INPUT: An augmented grammar G'.
OUTPUT: The SLR-parsing table functions ACTION and GOTO for G'.
METHOD:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G'.
2. State i is constructed from I_i. The parsing actions for state i are determined as follows:
   (a) If $[A \to \alpha \cdot a\beta]$ is in I_i and GOTO$(I_i, a) = I_j$, then set ACTION$[i, a]$ to "shift j." Here a must be a terminal.
   (b) If $[A \to \alpha\cdot]$ is in I_i, then set ACTION$[i, a]$ to "reduce $A \to \alpha$" for all a in FOLLOW(A); here A may not be S'.
   (c) If $[S' \to S\cdot]$ is in I_i, then set ACTION$[i, \$]$ to "accept."

   If any conflicting actions result from the above rules, we asy the grammar is not SLR(1). The algorithm fails to produce a parser in this case.
3. The goto transitions for state i are constructed for all nonterminals A using the rule: If GOTO$(I_i, A) = I_j$, then GOTO$[i, A] = j$.
4. All entries not defined by rules (2) and (3) are made "error."
5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S]$.

The parsing table consisting of the ACTION and GOTO functions determined by Algorithm 5.3 is called the *SLR(1) table for G*. An LR parser using the SLR(1) table for

G is called the SLR(1) parser for G, and a grammar having an SLR(1) parsing table is said to be *SLR(1)*.

5.4 Viable Prefixes

The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called *viable prefixes*.

We say item $A \to \beta_1 \cdot \beta_2$ is *valid* for a viable prefix $\alpha\beta_1$ is there is a derivation $S' \underset{rm}{\overset{*}{\Rightarrow}} \alpha a w \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 w$.

6 More Powerful LR Parsers

6.1 Canonical LR(1) Items

Recall that in the SLR method, state i calls for reduction by $A \to \alpha$ if the set of items I_i contains item $[A \to \alpha\cdot]$ and a is in FOLLOW(A).

It is possible to carry more information in the state that will allow us to rule out some of these invalid reductions by $A \to \alpha$.

The extra information is incorporated into the state by redefining items to include a terminal symbol as a second component. The general form of an item becomes $[A \to \alpha \cdot \beta, a]$, where $A \to \alpha\beta$ is a production and a is a terminal or the right endmarker \$. We call such an object an *LR(1) item*. The 1 refers to the length of the second component, called the *lookahead* of the item.

Formally, we say LR(1) item $[A \to \alpha \cdot \beta, a]$ is *valid* for a viable prefix γ if there is a derivation $S \underset{rm}{\overset{*}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta\alpha\beta w$, where

1. $\gamma = \delta\alpha$, and

2. Either a is the first symbol of w, or w is ϵ and a is \$.

6.2 Constructing LR(1) Sets of Items

Algorithm: Construction of the sets of LR(1) items.
INPUT: An augmented grammar G'.
OUTPUT: The sets of LR(1) items that are the set of items valid for one or more viable prefixes of G'.
METHOD: The procedures CLOSURE and GOTO and the main routine *items* for constructing the sets of items were shown in Fig. 4.4. ■

```
SetOfItems CLOSURE(I) {
    repeat
        for ( each item [A → α · Bβ, a] in I )
            for ( each production B → γ in G′ )
                for ( each terminal b in FIRST(βa) )
                    add [B → ·γ, b] to set I;
    until no more items are added to I;
    return I;
}

SetOfItems GOTO(I, X) {
    initialize J to be the empty set;
    for ( each item [A → α · Xβ, a] in I )
        add item [A → αX · β, a] to set J;
    return CLOSURE(J);
}

void items(G′) {
    initialize C to 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;
}
```

Figure 4.4: Sets-of-LR(1)-items construction for grammar G'

6.3 Canonical LR(1) Parsing Tables

We now give the rules for constructing the LR(1) ACTION and GOTO functions from the sets of LR(1) items.

Algorithm: Construction of canonical-LR parsing tables.
INPUT: An augmented grammar G'.
OUTPUT: The canonical-LR parsing table functions ACTION and GOTO for G'.
METHOD:

1. Construct $C' = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items for G'.
2. State i of the parser is constructed from I_i. The parsing action for state i is determined as follows.
   (a) If $[A \to \alpha \cdot a\beta, b]$ is in I_i and GOTO$(I_i, a) = I_j$, then set ACTION$[i, a]$ to "shift j." Here a must be a terminal.
   (b) If $[A \to \alpha\cdot, a]$ is in I_i, $A \neq S'$, then set ACTION$[i, a]$ to "reduce $A \to \alpha$."
   (c) If $[S' \to S\cdot, \$]$ is in I_i, then set ACTION$[i, \$]$ to "accept."

If any conflicting actions result from the above rules, we say the grammar is not LR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If GOTO$(I_i, A) = I_j$, then GOTO$([i, A]) = j$.

4. All entries not defined by rules (2) and (3) are made "error."

5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S, \$]$.

The table formed from the parsing action and goto functions produced by the above algorithm is called the *canonical* LR(1) parsing table. If the parsing action function has no multiply defined entries, then the given grammar is called an *LR(1) grammar*.

6.4 Constructing LALR Parsing Tables

We now introduce our last parser construction method, the LALR (*lookahead*-LR) technique.

Algorithm: An easy, but space-comsuming LALR table construction.
INPUT: An augmented grammar G'.
OUTPUT: The LALR parsing-table functions ACTION and GOTO for G'.
METHOD:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items.

2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.

3. Let $C' = \{J_0, J_1, \ldots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner as in Algorithm 4.56. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).

4. The GOTO table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is, $J = I_1 \cup I_2 \cup \cdots \cup I_k$, then the cores of GOTO$(I_1, X)$, GOTO$(I_2, X)$,..., GOTO$(I_k, X)$ are the same, since $I_1, I_2, \ldots, I_k$ all have the same core. Let K be the union of all sets of items having the same core as GOTO(I_1, X). Then GOTO(J, X)= K.

The table produced by Algorithm 4.59 is called the *LALR parsing table* for G. If there are no parsing action conflicts, then the given grammar is said to be an *LALR(1) grammar*. The collection of sets of items constructed in step (3) is called the *LALR(1) collection*.

6.5 Efficient Construction of LALR Parsing Tables

There are two ways a lookahead b can get attached to an LR(0) item $B \to \gamma \cdot \delta$ in some set of LALR(1) items J:

1. There is a set of items I, with a kernel item $A \to \alpha \cdot \beta, a$, and $J = \text{GOTO}(I, X)$, and the construction of

$$\text{GOTO}(\text{CLOSURE}(\{[A \to \alpha \cdot \beta, a]\}), X)$$

as given in Fig. 4.4, contains $[B \to \gamma \cdot \delta, b]$, regardless of a. Such a lookahead b is said to be generated *spontaneously* for $B \to \gamma \cdot \delta$.

2. As a special case, lookahead \$ is generated spontaneously for the item $S' \to \cdot S$ in the initial set of items.

3. All is as in (1), but $a = b$, and $\text{GOTO}(\text{CLOSURE}(\{[A \to \alpha \cdot \beta, b]\}), X)$, as given in Fig. 4.4, contains $[B \to \gamma \cdot \delta, b]$ only because $A \to \alpha \cdot \beta$ has b as one of its associated lookaheads. In such a case, we say that lookaheads *propagate* from $A \to \alpha \cdot \beta$ in the kernel of I to $B \to \gamma \cdot \delta$ in the kernel of J. Note that propagation does not depend on the particular lookahead symbol; either all lookaheads propagate from one item to another, or none do.

Let # be a symbol not in the grammar at hand. Let $A \to \alpha \cdot \beta$ be a kernel LR(0) item in set I. Compute, for each X, $J = \text{GOTO}(\text{CLOSURE}(\{[A \to \alpha \cdot \beta, \#]\}), X)$. For each kernel item in J, we examine its set of lookaheads. If # is a lookahead, then lookaheads propagate to that item from $A \to \alpha \cdot \beta$. Any other lookahead is spontaneously generated. These ideas are made precise in the following algorithm, which also makes use of the fact that the only kernel items in J must have X immediately to the left of the dot; that is, they must be of the form $B \to \gamma X \cdot \delta$.

Algorithm: Determining lookaheads.
INPUT: The kernel K of a set of LR(0) items I and a grammar symbol X.
OUTPUT: The lookaheads spontaneously generated by items in I for kernel items in $\text{GOTO}(I, X)$ and the items in I from which lookaheads are propagated to kernel items in $\text{GOTO}(I, X)$.
METHOD: The algorithm is given in Fig. 4.5. ■

```
for ( each item A → α · β in K ) {
    J := CLOSURE({[A → α · β, #]});
    if ( [B → γ · Xδ, a] is in J, and a is not # )
        conclude that lookahead a is generated spontaneously for item
            B → γX · δ in GOTO(I, X);
    if ( [B → γ · Xδ, #] is in J )
        conclude that lookaheads propagate from A → α · β in I to
            B → γX · δ in GOTO(I, X);
}
```

Figure 4.5: Discovering propagated and spontaneous lookaheads

The next algorithm describes one technique to propagate lookaheads to all items.

Algorithm: Efficient computation of the kernels of the LALR(1) collection of sets of items.
INPUT: An augmented grammar G'.

OUTPUT: The kernels of the LALR(1) collection of sets of items for G'.
METHOD:

1. Construct the kernels of the sets of LR(0) items for G. If space is not at a premium, the simplest way is to construct the LR(0) sets of items and then remove the nonkernel items. If space is severely constrained, we may wish instead to store only the kernel items for each set, and compute GOTO for a set of items I by first computing the closure of I.

2. Apply Algorithm 4.62 to the kernel of each set of LR(0) items and grammar symbol X to determine which lookaheads are spontaneously generated for kernel items in GOTO(I, X), and from which items in I lookaheads are propagated to kernel items in GOTO(I, X).

3. Initialize a table that gives, for each kernel item in each set of items, the associated lookaheads. Initially, each item has associated with it only those lookaheads that we determined in step (2) were generated spontaneously.

4. Make repeated passes over the kernel items in all sets. When we visit an item i, we look up the kernel items to which i propagates its lookaheads, using information tabulated in step (2). The current set of lookaheads for i is added to those already associated with each of the items to which i propagates its lookaheads. We continue making passes over the kernel items until no more new lookaheads are propagated.

7 Using Ambiguous Grammars

7.1 Precedence and Associativity to Resolve Conflicts

The parser for the ambiguous grammar (4.3) will not waste time reducing by these *single* productions (productions whose body consists of a single nonterminal).

CHAPTER 5

SYNTAX-DIRECTED TRANSLATION

We associate information with a language construct by attaching *attributes* to the grammar symbol(s) representing the construct.

1 Syntax-Directed Definitions

A *syntax-directed definition* (SDD) is a context-free grammar together with attributes and rules.

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.

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.

An SDD that involves only synthesized attributes is called *S-attributed*.
An SDD without side effects is sometimes called an *attribute grammar*.

1.2 Evaluating an SDD at the Nodes of a Parse Tree

A parse tree, showing the value(s) of its attribute(s) is called an *annotated parse tree*.

2 Evaluation Orders for SDD's

2.1 Dependency Graphs

A *dependency graph* depicts the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second.

2.2 Ordering the Evaluation of Attributes

If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N. Thus, the only allowable orders of evaluation are those sequences of nodes $N_1, N_2, \ldots, N_k$ such that if there is an edge of the dependency garph from N_i to N_j, then $i < j$. Such an ordering embeds a directed graph into a linear order, and is called a *topological sort* of the graph.

2.3 S-Attributed Definitions

In practice, translations can be implemented using classes of SDD's that guarantee an evaluation order, since they do not permit dependency graphs with cycles.

The first class is defined as follows:

- An SDD is *S-attributed* if every attribute is synthesized.

2.4 L-attributed Definitions

The second class of SDD's is called *L-attributed definitions*.

3 Syntax-Directed Translation Schemes

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.

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 $M \to \epsilon$.

3.1 Postfix Translation Schemes

SDT's with all actions at the right ends of the production bodies are called *postfix SDT's*.

4 Implementing L-Attributed SDD's

The following methods do translation by traversing a parse tree:

1. *Build the parse tree and annotate.*
2. *Build the parse tree, add actions, and execute the actions in preorder.*

In this section, we discuss the following methods for translation during parsing:

3. *Use a recursive-descent parser* with one function for each nonterminal.
4. *Generate code on the fly*, using a recursive-descent parser.
5. *Implement an SDT in conjunction with an LL-parser.*
6. *Implement an SDT in conjunction with an LR-parser.*

4.1 On-The-Fly Code Generation

The construction of long strings of code that are attribute values is undesirable for several reasons, including the time it could take to copy or move long strings. In common cases such as our running code-generation example, we can instead incrementally generate pieces of the code into an array or output file by executing actions in an SDT. The elements we need to make this technique work are:

1. There is, for one or more nonterminals, a *main* attribute.
2. The main attributes are synthesized.
3. The rules that evaluate the main attribute(s) ensure that
   (a) The main attribute in the concatenation of main attributes of nonterminals appearing in the body of the production involved, perhaps with other elements that are not main attributes.
   (b) The main attributes of nonterminals appear in the rule in the same order as the nonterminals themselves appear in the production body.

4.2 L-Attributed SDD's and LL Parsing

In addition to records representing terminals and nonterminals, the parser stack will hold *action-records* representing actions to be executed and *synthesized-records* to hold the synthesized attributes for nonterminals.

CHAPTER 6

INTERMEDIATE-CODE GENERATION

Static checking includes *type checking*, which ensures that operators are applied to compatible operands.

1 Variants of Syntax Trees

A directed acyclic graph (hereafter called a *DAG*) for an expression identifies the *common subexpressions* (subexpressions that occur more than once) of the expression.

1.1 The Value-Number Method for Constructing DAG's

Often, the nodes of a syntax tree or DAG are stored in an array of records.

In this array, we refer to nodes by giving the integer index of the record for that node within the array. This integer historically has been called the *value number* for the node or for the expression represented by the node.

Let the *signature* of an interior node be the triple $\langle op, l, r \rangle$, where op is the label, l its left child's value number, and r its right child's value number.

> **Algorithm:** The value-number method for constructing the nodes of a DAG.
> **INPUT**: Label op, node l, and node r.
> **OUTPUT**: The value number of a node in the array with signature $\langle op, l, r \rangle$.
> **METHOD**: Search the array for a node M with label op, left child l, and right child r. If there is such a node, return the value number of M. If not, create in the array a new node N with label op, left child l, and right child r, and return its value number.

The hash table is one of several data structures that support *dictionaries* efficiently.

To construct a hash table for the nodes of a DAG, we need a *hash function* h that computes the index of the bucket for a signature $\langle op, l, r \rangle$, in a way that distributes the signatures across buckets, so that it is unlikely that any one bucket will get much more than a fair share of the nodes.

An array, indexed by hash value, holds the *bucket headers*, each of which points to the first cell of the list.

2 Three-Address Code

2.1 Addresses and Instructions

An address can be one of the following:

- A *name.*
- A *constant.*
- A *compiler-generated temporary.*

Here is a list of the common three-address instruction forms:

1. Assignment instructions of the form $x = y\ op\ z$, where *op* is a binary arithmetic or logical operation, and x, y, and z are addresses.
2. Assignments of the form $x = op\ y$, where *op* is a unary operation.
3. *Copy instructions* of the form $x = y$, where x is assigned the value of y.
4. An unconditional jump `goto` L.
5. Conditional jumps of the form `if` x `goto` L and `ifFalse` x `goto` L.
6. Conditional jumps which apply a relational operator to x and y, and execute the instruction with label L next if x stands in relation *relop* to y.
7. Procedure calls and returns are implemented using the following instructions: `param` x for parameters; `call` p, n and y `= call` p, n for procedure and function calls, respectively; and `return` y.
8. Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$.
9. Address and pointer assignments of the form $x = \&y, x = *y$, and $*x = y$.

2.2 Quadruples

A *quadruple* (or just "*quad*") has four fields, which we call op, arg_1, arg_2, and the *result*.

2.3 Triples

A *triple* has only three fields, which we call op, arg_1, and arg_2.

Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves.

2.4 Static Single-Assignment Form

Two distinctive aspects distinguish SSA from three-address code. The first is that all assignments in SSA are to variables with distinct names; hence the term *static single-assignment.*

3 Types and Declarations

The applications of types can be grouped under checking and translation:

- *Type checking* uses logical rules to reason about the behavior of a program at run time.
- *Transition Applications.*

The actual storage for a procedure call or an object is allocated at run time, when the procedure is called or the object is created. As we examine local declarations at compile time, we can, however, lay out *relative addresses*, where the relative address of a name or a component of a data structure is an offset from the start of a data area.

3.1 Type Expressions

Types have structure, which we shall represent using *type expressions*: a type expression is either a basic type or is formed by applying an operator called a *type constructor* to a type expression.

We shall use the following definition of type expressions:

- A basic type is a type expression. Typical basic types for a language include *boolean*, *char*, *integer*, *float*, and *void*; the latter denotes "the absence of a value".
- A type name is a type expression.
- A type expression can be formed by applying the *array* type constructor to a number and a type expression.
- A record is a data structure with named fields. A type expression can be formed by applying the *record* type constructor to the field names and their types.
- A type expression can be formed by using the type constructor $\rightarrow$ for function types.
- If s and t are type expressions, then their Cartesian product $s \times t$ is a type expression.
- Type expressions may contain variables whose values are type expressions.

3.2 Type Equivalence

When type expressions are represented by graphs, two types are *structurally equivalent* if and only if one of the following conditions is true:

- They are the same basic type.
- They are formed by applying the same constructor to structurally equivalent types.
- One is a type name that denotes the other.

If type names are treated as standing for themselves, then the first two conditions in the above definition lead to *name equivalence* of type expressions.

3.3 Storage Layout for Local Names

Address Alignment

The storage layout for data objects is strongly influenced by the addressing constraints of the target machine. For example, instructions to add integers may expect integers to be *aligned*, that is, placed at certain positions in memory. Space left unused due to alignment considerations is referred to as *padding*. When space is at a premium, a compiler may *pack* data so that no padding is left; additional instructions may then need to be executed at run time to position packed data so that it can be operated on as if it were properly aligned.

The *width* of a type is the number of storage units needed for objects of that type.

4 Translation of Expressions

4.1 Addressing Array Elements

A two-dimensional array is normally stored in one of two forms, either *row-major* (row-by-row) or *column-major* (column-by-column).

5 Type Checking

To do *type checking* a compiler needs to assign a type expression to each component of the source program. The compiler must then determine that these type expressions conform to a collection of logical rules that is called the *type system* for the source language.

A *sound* type system eliminates the need for dynamic checking for type errors, because it allows us to determine statically that these errors cannot occur when the target program runs. An implementation of a language is *strongly typed* if a compiler guarantees that the programs it accepts will run without type errors.

5.1 Rules for Type Checking

Type synthesis builds up the type of an expression from the types of its subexpressions.

Type inference determines the type of a language construct from the way it is used.

5.2 Type Conversions

The rules for Java distinguish between *widening* conversions, which are intended to preserve information, and *narrowing* conversions, which can lose information.

Conversion from one type to another is said to be *implicit* if it is done automatically by the compiler. Implicit type conversions, also called *coercions*, are limited in many languages to widening conversions. Conversion is said to be *explicit* if the programmer must write something to cause the conversion. Explicit converions are also called *casts*.

5.3 Overloading of Functions and Operators

An *overloaded* symbol has different meanings depending on its context. Overloading is *resolved* when a unique meaning is determined for each occurrence of a name.

5.4 Type Inference and Polymorphic Functions

In this section, we consider *parametric polymorphism*, where the polymorphism is characterized by parameters or type variables.

The $\forall$ symbol is the *universal quantifier*, and the type variable to which it is applied is said to be *bound* by it.

Substitutions, Instances, and Unification

If t is a type expression, and S is a substitution (a mapping from type variables to type expressions), then we write $S(t)$ for the result of consistently replacing all occurences of each type variable α in t by $S(\alpha)$. $S(t)$ is called an *instance* of t.

Substitution S is a *unifier* of type expressions t_1 and t_2 if $S(t_1) = S(t_2)$. S is the *most general unifier* of t_1 and t_2 if for any other unifier of t_1 and t_2, say S', it is the case that for any t, $S'(t)$ is an instance of $S(t)$.

A *substitution* is a mapping from type variables to type expressions.

Two type expressions t_1 and t_2 *unify* if there exists some substitution S such that $S(t_1) = S(t_2)$.

Algorithm: Type inference for polymorphic functions.
INPUT: A program consisting of a sequence of function definitions followed by an expression to be evaluated. An expression is made up of function applications and names, where names can have predefined polymorphic types.
OUTPUT: Inferred types for the names in the program.
METHOD: For simplicity, we shall deal with unary functions only. The type of a function $f(x_1, x_2)$ with two parameters can be represented by a type expression $s_1 \times s_2 \to t$, where s_1 and s_2 are the types of x_1 and x_2, respectively, and t is the type of the result $f(x_1, x_2)$. An expression $f(a, b)$ can be checked by matching the type of a with s_1 and the type of b with s_2.

Check the function definitions and the expression in the input sequence. Use the inferred type of a function if it is subsequently used in an expression.

- For a function definition **fun** $\mathbf{id}_1(\mathbf{id}_2) = E$, create fresh type variables α and β. Associate the type $\alpha \to \beta$ with the function $\mathbf{id}_1$, and the type α with the parameter $\mathbf{id}_2$. Then, infer a type for expression E. Suppose α denotes types s and β denotes type t after inference for E. The inferred type of function $\mathbf{id}_1$ is $s \to t$. Bind any type variables that remain unconstrained in $s \to t$ by $\forall$ quantifiers.
- For a function application $E_1(E_2)$, infer types for E_1 and E_2. Since E_1 is used as a function, its type must have the form $s \to s'$. (Technically, the type of E_1 must unify with $\beta \to \gamma$, where β and γ are new type variables). Let t be the inferred type of E_1. Unify s and t. If unification fails, the expression has a type

error. Otherwise, the inferred type of $E_1(E_2)$ is s'.

- For each occurrence of a polymorphic function, replace the bound variables in its type by distinct fresh variables and remove the $\forall$ quantifiers. The resulting type expression is the inferred type of this occurrence.
- For a name that is encountered for the first time, introduce a fresh variable for its type.

5.5 An Algorithm for Unification

Algorithm: Unification of a pair of nodes in a type graph.
INPUT: A graph representing a type and a pair of nodes m and n to be unified.
OUTPUT: Boolean value true if the expressions represented by the nodes m and n unify; false, otherwise.
METHOD: A node is implemented by a record with fields for a binary operator and pointers to the left and right children. The sets of equivalent nodes are maintained using the *set* field. One node in each equivalence class is chosen to be the unique representative of the equivalence class by making its *set* field contain a null pointer. The *set* fields of the remaining nodes in the equivalent class will point (possibly indirectly through other nodes in the set) to the representative. Initially, each node n is in an equivalence class by itself, with n as its own representative node.

The unification algorithm, shown in Fig. 6.32, uses the following two operations on nodes:

- $find(n)$ returns the representative node of the equivalence class currently containing node n.
- $union(m, n)$ merges the equivalence classes containing nodes m and n. If one of the representatives for the equivalence classes of m and n is a nonvariable node, *union* makes that nonvariable node be the representative for the merged equivalence class; otherwise, *union* makes one or the other of the original representatives be the new representative. This asymmetry in the specification of *union* is important because a variable cannot be used as the representative for an equivalence class for an expression containing a type constructor or basic type. Otherwise, two inequivalent expressions may be unified through that variable.

The *union* operation on sets is implemented by simply changing the *set* field of the representative of one equivalence class so that it points to the representative of the other. To find the equivalence class that a node belongs to, we follow the *set* pointers of nodes until the representative (the node with a null pointer in the set field) is reached.

Note that the algorithm in Fig. 6.1 uses $s = find(m)$ and $t = find(n)$ rather than m and n, respectively. The representative nodes s and t are equal if m and n are in the same equivalence class. If s and t represent the same basic type, the call $unify(m, n)$ returns true. If s and t are both interior nodes for a binary type constructor, we merge their equivalence classes on speculation and recursively check that their respective children are equivalent. By merging first, we decrease the number of equivalence classes before recursively checking the children, so the algorithm terminates.

The substitution of an expression for a variable is implemented by adding the leaf for the variable to the equivalence class containing the node for that expression. Suppose

either m or n is a leaf for a variable. Suppose also that this leaf has been put into an equivalence class with a node representing an expression with a type constructor or a basic type. Then $find$ will return a representative that reflects that type constructor or basic type, so that a variable cannot be unified with two different expressions. ■

```
boolean unify(Node m,Node n) {
    s = find(m); t = find(n);
    if (s = t) return true;
    else if (nodes s and t represent the same basic type) return true;
    else if (s is an op-node with children s1 and s2 and
             t is an op-node with children t1 and t2) {
        union(s, t);
        return unify(s1, t1) and unify(s2, t2);
    }
    else if s or t represents a variable {
        union(s, t);
        return true;
    }
    else return false;
}
```

Figure 6.1: Unification algorithm.

6 Control Flow

In programming languages, boolean expressions are often used to

1. *Alter the flow of control.*
2. *Compute logical values.*

6.1 Short-Circuit Code

In *short-circuit* (or *jumping*) code, the boolean operators &&, ||, and ! translate into jumps.

6.2 Boolean Values and Jumping Code

The focus in this section has been on the use of boolean expressions to alter the flow of control in statements. A boolean expression may also be evaluated for its value, as in assignment statements.

A clean way of handling both roles of boolean expressions is to first build a syntax tree for expressions, using either of the following approaches:

1. *Use two passes.*
2. *Use one pass for statements, but two passes for expressions.*

7 Backpatching

This section takes a complementary approach, called *backpatching*, in which lists of jumps are passed as synthesized attributes.