Compilers: Principles, Techniques, & Tools

By Alfred V. Aho et all

First Edition

CONTENTS

Contents 1

1 Introduction 1

1 Language Processors . . . 1
2 The Structure of a Compiler . . . 1
2.1 Lexical Analysis . . . 1
2.2 Syntax Analysis . . . 2
2.3 Semantic Analysis . . . 2
2.4 Intermediate Code Generation . . . 2
2.5 The Grouping of Phases into Passes . . . 2
2.6 Compiler-Construction Tools . . . 2
3 The Evolution of Programming Language . . . 2
3.1 The Move to Higher-Level Languages . . . 2
4 Applications of Compiler Technology . . . 3
4.1 implementation of High-Level Programming Languages . . . 3
4.2 Optimizations for Computer Architectures . . . 3
5 Programming Language Basics . . . 3
5.1 The Static/Dynamic Distinction . . . 3
5.2 Environments and States . . . 3
5.3 Static Scope and Block Structure . . . 4
5.4 Explicit Access Control . . . 4
5.5 Dynamic Scope . . . 4
5.6 Parameter Passing Mechanisms . . . 4
5.7 Aliasing . . . 5

2 A Simple Syntax-Directed Translator 7

1 Introduction . . . 7
2 Syntax Definition . . . 8
2.1 Definition of Grammars . . . 8
2.2 Derivations . . . 8
2.3 Parse Trees . . . 8
2.4 Ambiguity . . . 9
2.5 Associativity of Operators . . . 9
2.6 Precedence of Operators . . . 9
3 Syntax-Directed Translation . . . 9
3.1 Postfix Notation . . . 9
3.2 Synthesized Attributes . . . 10
3.3 Tree Traversals . . . 10
3.4 Translation Schemes . . . 10
4 Parsing . . . 10
4.1 Top-Down Parsing . . . 10
4.2 Predictive Parsing . . . 10
4.3 Designing a Predictive Parser . . . 10
4.4 Left Recursion . . . 11
5 A Translator for Simple Expressions . . . 11
5.1 Abstract and Concrete Syntax . . . 11
5.2 Simplifying the Translator . . . 11
5.3 The Complete Program . . . 12

- 6 Lexical Analysis . . . 12
  - 6.1 Recognizing Keywords and Identifiers . . . 12
- 7 Symbol Tables . . . 13
  - 7.1 Symbol Table Per Scope . . . 13
- 8 Intermediate Code Generation . . . 13
  - 8.1 Two Kinds of Intermediate Representations . . . 13
  - 8.2 Construction of Syntax Trees . . . 13
  - 8.3 Static Checking . . . 14
  - 8.4 Three-Address Code . . . 14

3 Lexical Analysis **15**

- 1 The Role of the Lexical Analyzer . . . 15
  - 1.1 Tokens, Patterns, and Lexemes . . . 15

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

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

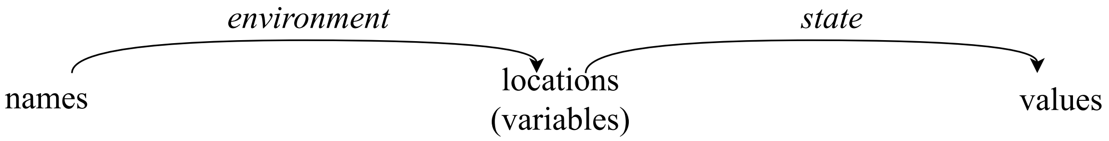


Figure 1.1: Two-stage mapping from names to values

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.

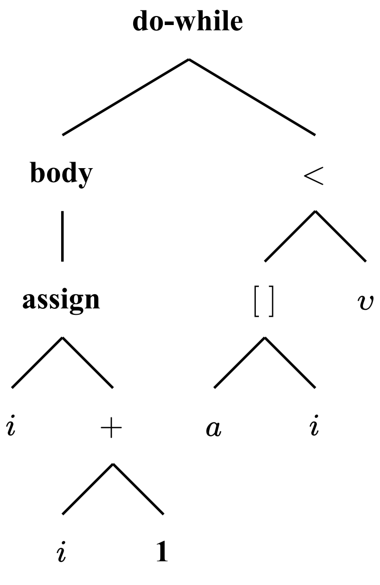
CHAPTER 2

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.



(a)

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

(b)

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 $A \rightarrow X_1 X_2 \cdots X_n$.

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:

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \epsilon \end{aligned}$$

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.

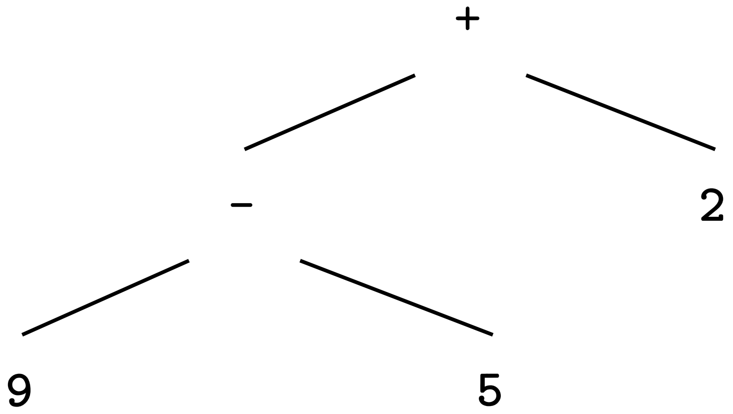


Figure 2.2: Syntax tree for `9-5+2`

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*.

Grammars routinely treat identifiers as terminals to simplify the parser, which can then expect the same terminal, say **id**, each time any identifier appears in the input. For example, on input

`count = count + increment;` (2.1)

the parser works with the terminal stream **id=id+id**.

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.

A program consists of blocks with optional declarations and "statements" consisting of single identifiers. Each such statement represents a use of the identifier. Here is a sample program in this language:

`{ int x; char y; { bool y; x; y; } x; y; }` (2.2)

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 Construction of Syntax Trees

Syntax Trees for Expressions

The table in Fig. 2.4 specifies the correspondence between the concrete and abstract syntax for several of the operators of Java.

The subscript *unary* in $-_{unary}$ is solely to distinguish a leading unary minus sign from a binary minus sign.

Concrete Syntax	Abstract Syntax
`=`	**assign**
`||`	**cond**
`&&`	**cond**
`== !=`	**rel**
`< <= >= >`	**rel**
`+ -`	**op**
`* / %`	**op**
`!`	**not**
$-_{unary}$	**minus**
`[]`	**access**

Figure 2.4: Concrete and abstract syntax for several Java operators

8.3 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.

8.4 Three-Address Code

Better Code for Expressions

We can avoid some copy instructions by modifying the translation functions to generate a partial instruction that computes, say `j+k`, but does not commit to where the result is to be placed, signified by **null** address for the result:

$$\textbf{null} = \texttt{j + k} \tag{2.3}$$

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 Input Buffering