Compiler Design ReferencePoint Suite

Table of Contents

Point 2: Principles of Compiler Design....1
The Components of a Compiler....1
The Lexical Analyzer....2
The Syntax Analyzer or Parser....4
The Intermediate Code Generator....6
The Code Optimizer....7
The Code Generator....8
Using Transition Diagrams....9
Defining Tokens Using Regular Expressions....9
Defining Languages Using Regular Expressions....10
Introducing Finite Automata....10
Nondeterministic Finite Automata....10
Deterministic Finite Automata....11
Back-End Analysis....12
Operator Precedence Parsing....12
Top-Down Parsing....13
Bottom-Up Parsing....14
Recursive-Descent Parsing....14
Semantic Analysis....15
The Symbol Table....16
Type Checking....16
Related Topics....17

Point 2: Principles of Compiler Design

Silvia Savitha George

A compiler is a program that converts source code written in high-level programming language into machine language. It analyzes the characters in the source code and generates optimized code.

A compiler works in phases, which consist of a set of logically related operations, such as lexical analysis, parsing, and translation. Each phase produces a different stage of the representation of an output.

This ReferencePoint explains how to build a compiler and describes the phases in the compilation process. In addition, it explains symbol table organization and the various intermediate representations used in back-end analysis by the compiler.

The Components of a Compiler

A compiler consists of a front end and a back end. The front end converts the source code to an intermediate representation by scanning, parsing, and type-checking the source code. The compiler uses scanning to read the tokens in the input source code from left to right. It uses parsing to return a parse tree of the tokens, and type checking to examine the identifiers in the source code for type compatibility. The compiler binds each identifier in the source code to its value and data type.

Note A token is a series of characters in the source code that can be logically grouped together. A parse tree is a tree structure into which tokens are converted and depicts the syntactic structure of an expression in the source code.

The front end is machine-independent and portable. The back end is dependent on the underlying source code. The back end generates machine code of the intermediate representation after code optimization.

Note A machine-dependent program cannot run on different types of computers. It can run only on the computer that it is compiled on. Machine-independent code can run on any computer irrespective of underlying hardware configuration.

There are five components of a compiler:

- Lexical analyzer
- Syntax analyzer
- Intermediate code generator
- Code optimizer
- Code generator

Figure 1-2-1 shows the components of a compiler:

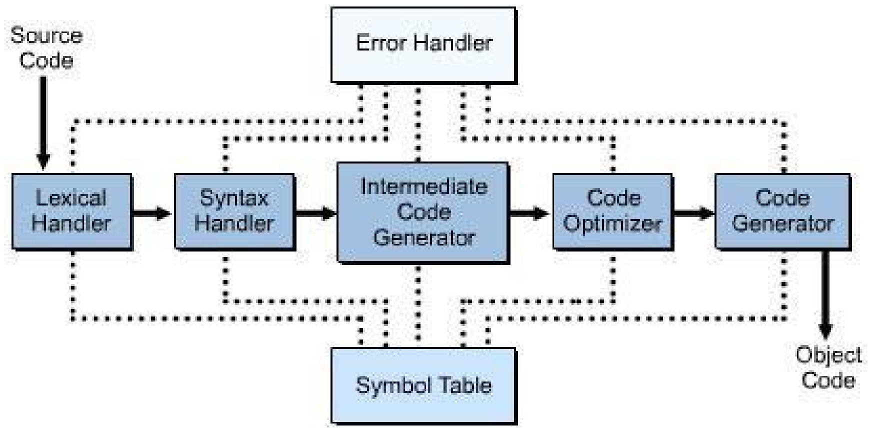


Figure 1-2-1: Components of a Compiler

Note Compilers that pass through the entire source code only once are called one-pass compilers. For efficient output, a compiler needs to execute multiple passes or traversals through the source code. Lexical analysis and the syntactical analysis are performed in the first pass of a compiler.

The Lexical Analyzer

The lexical analyzer separates the characters of the source code into logical groups called tokens. You can write a customized lexical analyzer or automatically generate it using the scanner generator software provided with definitions of all valid tokens in a language.

Tokens can be keywords such as DO, identifiers such as X or Num, operator symbols such as <=, and punctuation symbols such as !. The lexical analyzer matches every token against a list of legal keywords and operators and produces equivalent unique codes for every series of characters that it recognizes as a token. For example, DO might be represented by 1 and + by 2.

The lexical analyzer scans the tokens so that the syntax analyzer can receive a stream of input tokens. For example, the following statement computes var * 6 and returns the result:

```
return var * 6;
```

The above expression has five tokens: the keyword return, the identifier var, the operator *, the constant 6, and the punctuation symbol, a semicolon.

The compiler invokes the lexical analyzer once for every token in the source code. To locate each token, the lexical analyzer scans every character and may need to scan several characters beyond a token before determining that a token has been found. For example, while analyzing the expression in the following statement, the lexical analyzer marks the left parenthesis as the first token:

```
WHILE (NUM.GE.87) Jump L1
```

Note Only the expression is being handled in the above statement. The lexical analyzer works in tandem with the syntax analyzer. The lexical analyzer positions a pointer at each character, processes the statements, and forms tokens. The collection of tokens is called the token stream. In the token stream, each identifier and constant is associated with a token value.

In the expression, WHILE (NUM.GE.87) Jump L1, when the syntax analyzer asks for the next token, the lexical analyzer reads all the characters between NUM and 87 to determine that the next token is the identifier NUM. The lexical analyzer needs to scan until the character NUM is found because it is unsure if it has read the complete constant. After determining that the next token is the constant NUM, the lexical analyzer repositions the pointer at the dot (.) after NUM. The lexical analyzer returns the token type as a constant to the syntax analyzer. The value associated with this token value for 87 could be the numerical value 87 or a pointer to the string 87, depending on the compiler design . After the lexical analyzer processes the statement completely, the token stream is:

```
WHILE ([id, 330] GE [const, 467])
```

In the above expression, the pairs of tokens in parentheses have analogous values. The presentation indicates whether the token is an identifier or a number. In the above expression, id denotes an identifier and const denotes a constant.

The second component in a pair is an index to the symbol table, which maintains information about constants, variables, and labels.

Note A symbol table stores information about the names that appear in the source program. A symbol table is a collection of bindings, which is a key-value pair. A key is used to uniquely identify the binding and the value is data that is linked to its key. A symbol table allows the compiler to insert new bindings, to retrieve the values of bindings with specified keys, and to remove bindings with specified keys.

Figure 1-2-2 shows the entries of the symbol table for the expression (NUM.GE.87) in a while statement:

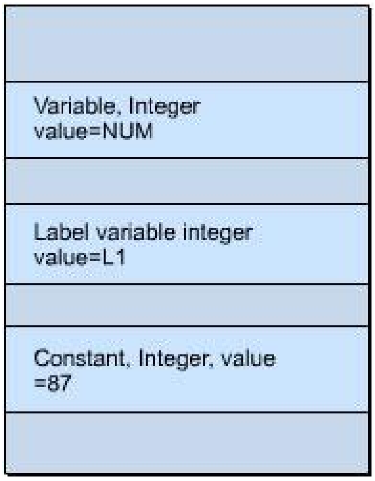


Figure 1-2-2: The Symbol Table for the Expression WHILE (NUM.GE.87) Jump L1
A lexical analyzer has five functions:

- Identifies keywords or reserved words of the language, such as the word, struct, in a C program.
- Identifies an individual special character or a group of special characters, such as those in a variable name.
- Identifies identifiers and constants.
- Identifies preprocessor directives and macros.
- Eliminates comments and white spaces, such as tabs and line feeds, from the program.

The Syntax Analyzer or Parser

The parser and the lexical analyzer are distinct components of the compiler. You can write a custom parser or automatically generate one using the parser generator software.

Note The parser generator is a part of the syntax analyzer. It has definitions of all valid syntax in a language.

The lexical analyzer identifies the tokens in source code. The parser groups the tokens in declarations, statements, and control statements. The parser determines whether or not the sequence of tokens extracted by the lexical analyzer is in the correct order. The parsing process depends on the programming language. There are four types of programming languages:

- Unrestricted: Natural languages, such as English, are unrestricted programming languages.
- Context-sensitive: Context-sensitive programming languages are modular in nature and can be parsed with a system stack. Older programming languages, such as FORTRAN, are context-sensitive. All context-free languages are context-sensitive. An example of a context-sensitive language that is not context-free is all numbers of the form, a^n, where n is a prime number.
  Note The system stack is used to store pointers to subroutines or vectors.
- Context-free: Context-free languages can be parsed using a state machine and system stack. All regular languages are always context-free but a context-free language is not necessarily a regular language. An example of a context-free language, which is not regular is, numbers a^nb^n, where n is not 0. Context-free grammar is normally used to specify the syntactic structure of a language.
- Regular: A regular language can be parsed with a state machine and a parser.

The syntax analyzer or parser performs two functions:

- Verifying that the tokens, which are the output of the lexical analyzer, occur in patterns as defined by the language specifications. For example, a C program contains the following expression:

  ```
  X * - Y
  ```

  After the lexical analysis, the expression may appear to the syntax analyzer as a token stream, as in:

  ```
  identifier * - identifier
  ```

  In the above expression, there are two operators, * and -. If the syntax analyzer discovers the - symbol in the token stream, it detects an error because the specifications of the C language do not permit these two operators in succession.

- Converting the tokens into a tree structure called the parse tree by defining the parts of the token stream that need to be in a single group.

You can interpret the expression, X * Y - Z, in two ways:

- Multiply X and Y and subtract Z from the result.
- Subtract Z from Y and multiply the result by X.

You can represent these interpretations in a parse tree diagram, which depicts the syntactic structure of the expression. Figure 1-2-3 shows the parse tree for the interpretation, multiply X and Y and subtract Z from the product:

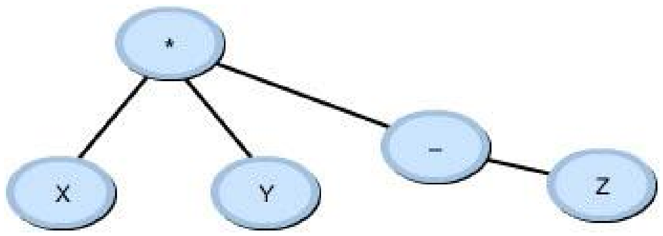


Figure 1-2-3: The Parse Tree Representing the Multiplication of X and Y and the Subtraction of Z from the Product

Figure 1-2-4 shows the parse tree that represents the interpretation, subtract Z from Y and multiply the result by X:

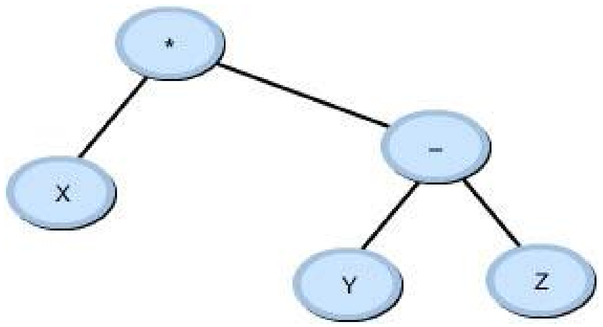


Figure 1-2-4: The Parse Tree Representing the Subtraction of Z from Y and the Multiplication of the Result by X

The language specification must indicate which of these interpretations the language uses. The language specification or grammar rules form the syntactic specification of the source language.

After a lexical analyzer analyzes the tokens in the source code, the parser builds a parse tree using the tokens. For example, after the lexical analyzer parses the statement, WHILE (NUM.GE.87) Jump L1, the parser constructs the parse tree for the statement. Figure 1-2-5 shows the tree structure for the token stream of the statement, WHILE (NUM.GE.87) Jump L1:

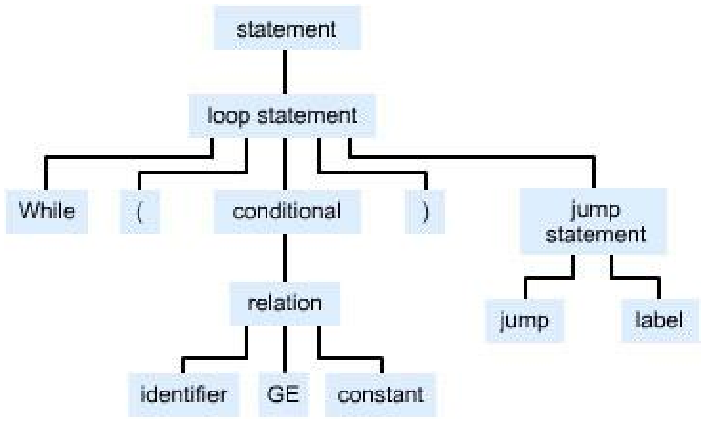


Figure 1-2-5: Token Stream Structure

The Intermediate Code Generator

After the syntax analysis is complete, the compiler can generate the object code for each construct. The compiler creates an intermediate code before generating the object code. One form of intermediate code is a parse tree, which stores variables as nodes.

Note A construct is part of a statement. It is a collection of syntactically legal keywords and identifiers.

The intermediate code generator produces a representation of the parse tree in intermediate language, which consists of unconditional and simple conditional branching statements.

Note Conditional branching statements test only one relation to determine whether or not to generate a branch.

A popular type of intermediate language is the three-address code, which is also known as quadruples or postfix code. The general form of the three-address code is:

```
X := Y bin-opertr Z
```

In the above code, X, Y, and Z are operands and bin-opertr is a binary operator.

The intermediate code generator translates higher-level flow of control statements, such as the while-do statements or the if-then-else statements, into lower level conditional three-address codes. For example, a while-do expression is:

```
Do while X <= Y && X<100
X=X-2;
```

The intermediate code generator translates the above expression into its corresponding token stream:

```
Do while {identifer,i1} <= [identifer,i2] && [identifier,i1]< [constant,i3]
[identifer,i1] = [identifer,i1] - [constant,i4] ;
```

In the above code, i1, i2, i3, and i4 are pointers to the entries in the symbol table for X, Y, 100, and 2. Figure 1-2-6 shows the parse tree for the while-do expression:

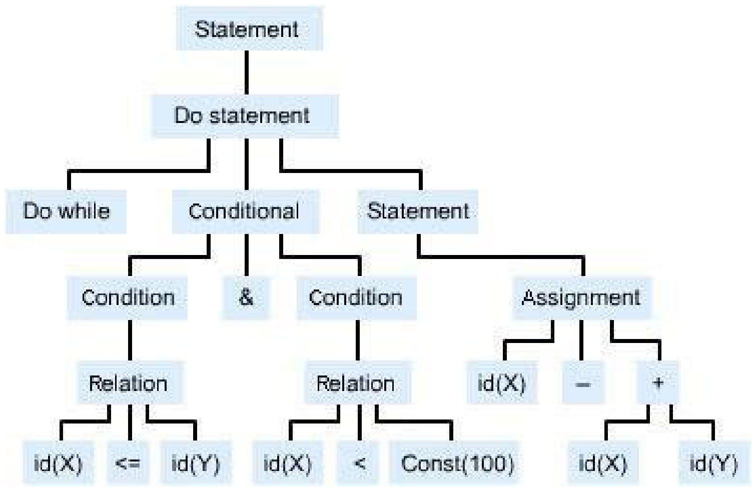


Figure 1-2-6: The Parse Tree for the while-do X <= Y && X<100 X=X+2
Listing 1-2-1 shows the intermediate code for the above parse tree:

Listing 1-2-1: Intermediate Code for the while-do Expression

```
        Label1: if X <= Y goto Label2
                Goto Label3
        Label2: if X < 100 goto Label42
                Goto Label3
                T2 := T1 - 5
        Label4: X := X + 2
                Goto Label1
        Label3:
```

The above listing shows the parse tree for the sample while-do expression converted to an intermediate form. The code uses conditions and jump statements to compare X and Y with each other and other values, and performs actions depending on the result of the condition.

The Code Optimizer

The code optimizer optimizes the intermediate code by applying mathematical transformations to the code. When optimizing the code, the compiler needs to ensure that the object programs that are frequently executed are small and have short execution times.

The code optimizer uses an interference graph to optimize the code. Using an interference graph, you can store temporary variables in registers. A pair of variables can be stored in the same register if they do not overlap or interoperate, that is, if their operations do not share memory or CPU usage.

An efficient compiler can optimize a target program for greater execution speeds. A code optimizer can apply mathematical transformations locally to the intermediate code. For example, in the following intermediate code, there are two jump statements:

```
if Max > input_var jump Label2
                jump Label3
L2:
```

You can replace the above code using the following statement:

```
If Max <= input_var jump Label3
```

An example of statements with redundant subexpressions is:

```
MAX := NUM1 + NUM2 + NUM3
NUM := NUM1 + NUM2 + NUM4
```

In the above code, if NUM1 is not an alias for NUM2 or NUM3, the assignments can be evaluated as:

```
TEMP  := NUM1  + NUM2
MAX  := TEMP  + NUM3
NUM  := TEMP  + NUM4
```

Alternative notations, such as postfix, also help in code optimization. Postfix notations enable you to build representations without parentheses for expressions and to facilitate compact storage.

You can apply code optimization extensively to loops and remove computations that produce the same result in all iterations. You need to place the computation just before the code enters the loop. An example of a non-optimized code is:

A Non-Optimized Code

```
While (num.GE.7)
        X:=2
        Y:=X+J/78.3
        Print Y
Endwhile
```

In the above code, the X:=2 statement does not need to be included in the while loop.

The code below shows the same code after optimization:

An Optimized Code

```
X:=2
While (num.GE.7)
        Y:=X+J/78.3
        Print Y
Endwhile
```

The above code places the X:=2 statement outside the loop. As a result, system resources are not wasted in assigning 2 to X for every iteration.

The Code Generator

The code generator converts intermediate code into the final object code. The code generator scans the parse tree and assigns nodes to memory locations. It then rescans the tree and generates the object code, which is a sequence of machine instructions.

Figure 1-2-7 shows a sample parse tree for the expression (U-V)+((W+X)+(Y*Z)):

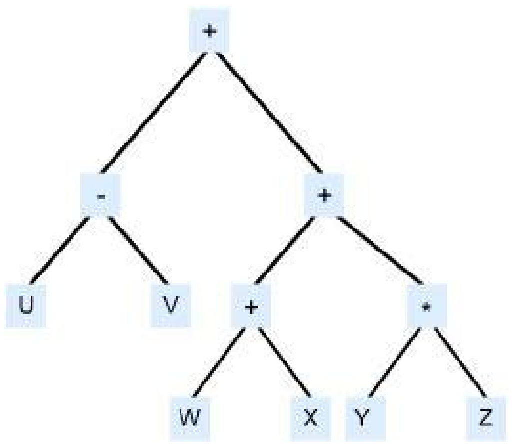


Figure 1-2-7: Parse Tree for the Expression (U -V)+((W+X)+(Y*Z))
Equivalent code generated for this parse tree using only two registers is:

```
load R1, W
add R1, X
```

```
load R2, Y
mult R2, Z
add R1, R2
load R2, U
sub R2, V
add R2, R1
```

Many computers have a limited number of high-speed registers intended for high-speed computations. An efficient code generator utilizes these registers efficiently. This aspect of code generation, called register allocation, is difficult to optimize.

Using Transition Diagrams

Transition diagrams are representations of regular expressions and are specialized flowcharts for lexical analyzers that enable you to manually simulate the process of lexical analysis. Transition diagrams recognize tokens in the input token stream. In a transition diagram, states are represented as circled numbers, which are connected by directed edges. A transition diagram represents a finite state machine, which is a set of states, input actions, output actions, and a transition procedure.

Figure 1-2-8 shows a transition diagram for a constant in the C language:

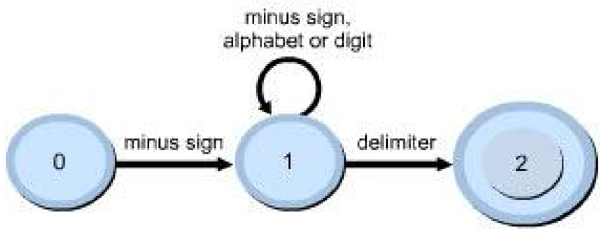


Figure 1-2-8: Transition Diagram for a Constant
Figure 1-2-8 shows a transition diagram for a constant, which is defined as a minus sign (-) or a letter followed by any number of alphanumeric characters or a minus sign. The starting state of the transition diagram is state 0. The edge from state 0 indicates that the first input character must be a letter or a minus sign.

To perform lexical analysis of the constant, enter state 1, and observe the next input character. If it is an alphanumeric character or a minus sign, reenter state 1, and observe the next input character. Continue to read alphanumeric character and minus signs. Make transitions from state 1 to itself until the next input character is a delimiter for a constant. If the input character is the delimiter, enter state 2.

Defining Tokens Using Regular Expressions

Every token is associated with a token class such as an identifier or a constant. If you assume every token class to be a language, you can use regular expressions to describe the tokens.

If an identifier is defined as a letter followed by letters or numbers, it can be written in the form of a regular expression, such as:

```
Identifier = alphabet (alphabet | number) *
```

The pipe symbol, |, denotes union. An expression grouped in parentheses is a subexpression. The asterisk is the closure operator indicating zero or more instances.

For example, the tokens for keywords, identifiers, constants, and relational operators can be described in the form of regular expressions, such as:

```
keyword := case | endcase | begin | end | if | then | else
identifier := alphabet (alphabet | number ) *
constant := number +
relational_opr := < | <= | = | <> | > | >=
```

In the above notation, alphabet represents A | B | C.....| Z and number represents 0 | 1 | 2 | ....| 9.

Defining Languages Using Regular Expressions

You can define languages using regular expressions, which is a defined finite set of symbols. For example, the set [0,1] is a class with two symbols, 0 and 1. This set is often called the binary alphabet. Two examples of alphabets are the ASCII and the EBCDIC character sets.

A string, such as 100, is a finite sequence of symbols. The length of a string y, usually denoted as |y|, is the number of symbols in y. For example, the string 010110 has a length of 6.

A special string is an empty string denoted by . Concatenated strings can be represented by a . symbol. For example, if a and b are strings, the concatenated string can be represented as a.b or ab.

You can apply concatenation to languages. If P and Q are languages, P.Q or PQ is the language that consists of all the strings ab that are formed by selecting string a from P, string b from Q, and concatenating them in that order:

```
PQ = { pq | a is  in P and b is in Q }
```

Introducing Finite Automata

Finite automata are transition diagrams that you can use to recognize the programming language in which the source program is coded.

A finite automaton has a finite set of states and transitions. The set of states in a finite automaton has an initial state and final states. A recognizer for a language P is a program that takes a string, a, as input, and answers yes if a is a sentence of P and no if it is not. To check if a string is a sentence of a language P, divide the string into a sequence of substrings denoted by the primitive subexpressions in P.

Nondeterministic Finite Automata

To recognize a programming language, you construct a generalized transition diagram from the expression called a Nondeterministic Finite Automata (NFA). NFA is a labeled directed graph and is equivalent to a transition diagram.

An NFA has four components:

- An identifier that enters various states
- A start state
- A scanning mechanism that reads from left to right
- An accepting or final state

Figure 1-2-9 shows the NFA for the language (x|y|z) * xyz:

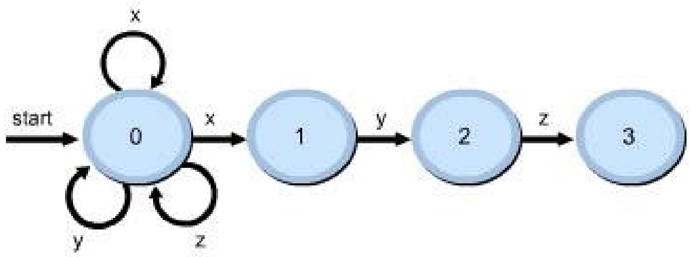
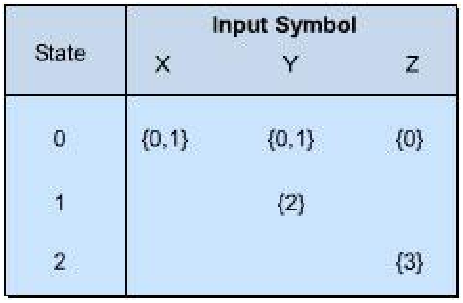


Figure 1-2-9: Transition Diagram NFA for the Language (x|y|z) * xyz
In an NFA, the nodes are called states and the edges are called transitions. You need to label both nodes and edges. In addition, you can label two or more transitions with the same character. You can distinguish one state as the start state and one or more states are distinguished as the accepting or final states.

You can represent the transitions of an NFA in a transition table, as shown in Figure 1-2-10. In the table in this figure, there is a row for each state and a column for each admissible input symbol. The entry for row I and the symbol x are the set of next states that are possible for state I on input x.



State	Input Symbol		
	X	Y	Z
0	{0,1}	{0,1}	{0}
1		{2}	
2			{3}

Figure 1-2-10: Possible Next States
Note Input symbols are unprocessed symbols in the source code.

Deterministic Finite Automata

An input variable can enter one of multiple states in an NFA. For example, there may be a transition from state 0 for the input x, indicating that x may go to either state 0 or state 1. These multivalued transition functions are difficult to implement.

Deterministic Finite Automata (DFA) is easier than NFA to simulate because it contains a maximum of one path from the start state that is labeled.

Figure 1-2-11 shows a DFA that accepts the language (x|y|z) * xyz:

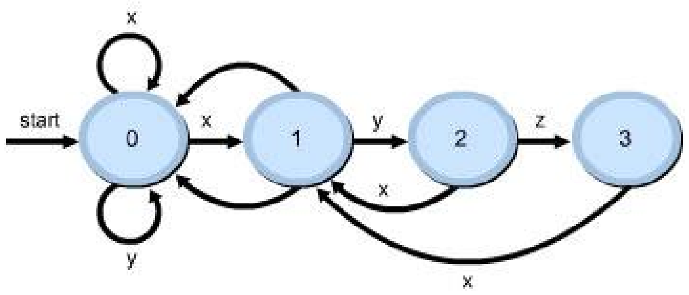


Figure 1-2-11: Deterministic Finite Automata

Back-End Analysis

Back-end analysis is the process of analyzing the syntactic structures in a language. Parsing is used for back-end analysis. There are four types of parsers:

- Operator precedence: Is used for expressions. You can use operator precedence for parsing expressions because it can use information about the precedence and the association of operators to guide the parsing.

- Top-down: Form the root and then fix the leaves of the parse tree. For a specified string, a top-down parser builds a tree starting from the nodes at the bottom of the tree and proceeds to the root node. This process transforms a given string into a grammar start symbol.

- Bottom-up: Involves shifting input symbols onto a stack until the right side of a production appears on top of the stack. The parser then replaces the symbols on the right side by the symbol on the left side of the production, and the process iterates. The bottom-up parsing method is called shift-reduce parsing.

- Recursive descent: Is used for all syntactic structures, other than expressions, in a language. Recursive descent uses a set of mutually recursive routines to analyze the syntactic structures in a language.
  Note Mutually recursive routines are two routines that invoke each other.

Operator Precedence Parsing

Operator precedence parsing is an easy-to-implement parsing technique. It manipulates tokens without any reference to the grammar that defines those tokens.

The only drawback of this method is that it is difficult to handle tokens, such as the minus sign. This is because minus has two precedence rules. In addition, only a small class of grammars can be parsed using operator precedence techniques.

Note The minus sign can be a unary operator, such as -1. It can also be used as a binary operator, as in a-b.

To set the precedence of operators, the operator precedence parsing method uses the following precedence priorities: <., =., and >.. For example:

- In the Exp1 <. Exp2 expression, Exp1 has less precedence than Exp2.
- In the Exp1 =. Exp2 expression, Exp1 and Exp2 have the same precedence
- In the Exp1 >. Exp2 expression, Exp1 has greater precedence than Exp2

These precedence relations, <. , = ., and >., enable you to determine what precedence relation occurs between any two operands. For example, if the / operator has a higher precedence than -, you define the precedence as follows:

```
-/ <. /
/ >. -.
```

Instead of using the above notation, you can build an unambiguous grammar for the language, which reflects the correct precedence of the operators.

Top-Down Parsing

Top-down parsing begins at the start symbol and derives the entire input string from the grammar.

The top-down parsing strategy involves backtracking, which involves repeated scans of the input. For example, in the following grammar, if you use the derivation Input string = Input string - component, instead of Input string = Input string + component, you need to backtrack because the derived symbols do not match the tokens in the input string:

```
Input string = Input string + component
Input string = Input string - component
Input string = component
Component = component * factor
Component = component / factor
Component = factor
Factor = digit or identifier
```

Top-down parsing obtains the preorder representation for an input string. It constructs a parse tree for the input by beginning from the root node and creating other nodes or leaves of the parse tree, first the root node, next the left node, and finally the right node.

For example, for the input token u = xpy, the productions are:

```
        G -> xpy
        P -> pq | p
```

To construct a parse tree for this sentence using top-down parsing, construct a tree that consists of a single node labeled G.

The first production in the derivation expands G to xpy. Figure 1-2-12 shows the parse tree for G -> xpy:

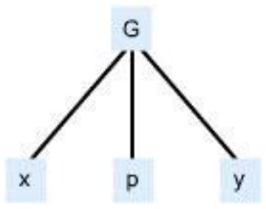


Figure 1-2-12: Parse Tree for G -> xpy

Start from x and you find that it cannot be expanded further. Continuing from left to right, you find P, which has another production defining its behavior. P has two syntactic structures or alternates, pq and p. Figure 1-2-13 shows the first alternate for P:

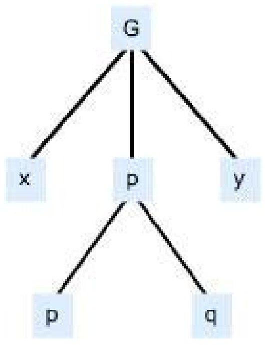


Figure 1-2-13: Expanding P with the First Alternate, pq

Next, you need to build the parse tree for the second alternate for P. For this, you need to backtrack and reset the current node to P. Figure 1-2-14 shows the tree for the second alternate:

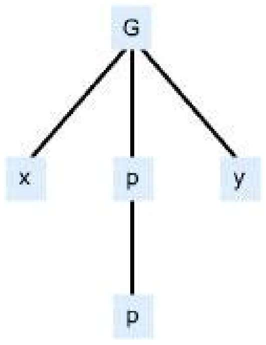


Figure 1-2-14: The Second Alternate for P

G branches to x, p, and y. The branch p is further expanded to another p. The leaf P matches the second symbol of u, and the leaf y matches the third symbol. The compiler has now produced the parse tree for u, and the parsing is successfully complete.

Bottom-Up Parsing

This type of parser processes grammar productions from right to left and uses a technique called reduction. The parser matches the first few symbols with the right sides of the productions in the grammar. If the parser finds a match, the left side of the equations replaces these symbols in the input string, which is stored in a stack. For example, if the input string is - Y * Y and if there is a production Y = Y * Y, the input string becomes - Y.

If the parser finds no matches between the rules and the input string, the symbols in the input string are shifted and the parser reads the next token.

Recursive-Descent Parsing

Recursive-descent parsing eliminates the backtracking required in top-down parsing. This type of parser uses a set of recursive procedures to recognize its input.

Recursive-descent parsing is a special case of top-down parsing. This method checks the syntax of an input stream for validity because the parser reads the stream from left to right. It matches the tokens from the input stream with terminals from the grammar. The recursive descent parser advances the pointer only when the parser finds a match.

In a production of the form X-> x1 | x2 | .....| xn, if the current input symbol x and the nonterminal X need to be expanded, you need to determine the unique alternate of this production that derives the string beginning with x. You can determine the required alternate by examining the first symbol the alternate derives. For example, you can detect control constructs with distinguishing keywords by determining the unique alternate of the production.

A sample production is:

```
Statement: if condition then statement else statement
                | while condition do statement
                 | begin statement-list end
```

In the above code, the keywords if, while, and begin indicate the only alternate that could succeed if you find a statement.

Semantic Analysis

Semantic analysis performs type checking using the symbol table. To check if the object code has valid data types, the compiler uses syntax or parse trees. The compiler needs to ensure the consistent use of variables and functions. When it detects an inconsistency, it needs to generate an error.

Listing 1-2-2 shows a sample program in C that triggers semantic errors:

Listing 1-2-2: A Semantically Incorrect Program

```
#include <stdio.h>
void main()
{
        int a;
        a = a + 1; /* valid */
        a[1] = 5; /* invalid */
}
```

In the above listing, a is defined as an integer. The statement, a = a + 1, is correct, but the statement a[1] = 5 generates an error.

Incorrect usage of functions and function calls also lead to semantic errors. Listing 1-2-3 shows a sample program that contains a semantic error because of an incorrect function call:

Listing 1-2-3: A Sample Program with Incorrect Function Call

```
#include <stdio.h>
void main()
{
        char *u;
        int j;
        int myfun(int a, int *g);
        myfun(5, &j); /* valid */
        myfun(u, 3); /* invalid */
}
```

In the above listing, the myfun() function takes two parameters of types integer and integer pointer. The second call to the myfun() function passes a character pointer instead of an integer and an integer in place of a pointer. The semantic analyzer needs to identify this and generate an error.

The Symbol Table

To record declarations, the compiler uses the symbol table. The compiler collects and uses information about the names in the symbol table that appear in the source program. The information collected about a name includes the string of characters by which it is denoted, its type, form, and location in memory; and other attributes that depend on the language.

Each entry in the symbol table is a pair of the form:

```
          name, information
```

Every time the compiler encounters a name, it searches the symbol table to see whether or not that name previously was observed. If the name is new, the compiler enters it into the symbol table.

The symbol table can:

- Determine whether or not a given name is in the table.
- Add a new name to the table.
- Access information about a given name.
- Add new information a given name.
- Delete a name or a group of names from the table.

The data structures that can be used to construct the symbol table are lists, self-organizing lists, search trees, or hash tables.

Type Checking

Type checking is the basic run-time support feature that a compiler provides. A compiler needs to examine identifiers for type compatibility. For example, in the following arithmetic expression, the compiler needs to perform four type checks:

```
x = g / h;
```

First, the compiler needs to compare g and h for compatibility. Then it needs to compare g and h individually with / and finally, it needs to compare the result of the expression with x.

Compilers apply type checking in three ways:

- Priori: Checks whether all types always are compatible with each other in compilers that have only one kind of data type. For example, if a compiler implements only the character data type, all variables are guaranteed to be type-compatible.
- Structural: Checks if two types are compatible with each other. They are compatible if they are identical in order and structure. For example, in C, two structures, both containing a char and float data type each, are type-compatible.

- Named: Checks whether or not two types are compatible with each other. They are compatible if they are explicitly declared to be so.

Related Topics

For related information on this topic, you can refer to:

- Introducing Compiler Design
- *Implementing Context-Free Grammar in Natural Language*
- Introducing Optimization in Compiler Design