

4.1 Regular Expressions to Automata

The regular expression is the notation of choice for describing lexical analyzers and other pattern-processing software. However, implementation of that software requires the simulation of a DFA, or perhaps simulation of an NFA. Because an NFA often has a choice of move on an input symbol, or even a choice of making a transition on ϵ : or on a real input symbol, its simulation is less straightforward than for a DFA. Thus often it is important to convert an NFA to a DFA that accepts the same language.

4.2 Conversion of an NFA to a DFA

For each NFA we can find a DFA accepting the same language. The number of states of the DFA could be exponential in the number of states of the NFA, but in practice this worst case occurs rarely.

Algorithm 4.1: The subset construction of a DFA from an NFA.

INPUT: An NFA N

OUTPUT: A DFA D accepting the same language as N .

METHOD: Each state of D is a set of states which N could be in after reading some sequence of input symbols. Thus D is able to simulate all possible moves N can make on a given input string. The initial state of D is the set consisting of S_0 , the initial state of N , together with all states of N that can be reached from S_0 by means of ϵ -transition only. The accepting states of D are those sets of states that contain at least one accepting states of N .

Let us define the function ϵ -CLOSURE (s) to be the set of states of N built by applying the following rules:-

- 1- S is added to ϵ -CLOSURE (s).
- 2- If t is in ϵ -CLOSURE (s), and there is an edge labeled ϵ from t to u , then u is added to ϵ -CLOSURE (s) if u is not already there. Rule 2 is repeated until no more states can be added to ϵ -CLOSURE (s).

Thus, ϵ -CLOSURE (s) is just the set of states that can be reached from S on ϵ -transition alone. If T is a set of states, then ϵ -CLOSURE (T) is just the union over all states S in T of ϵ -CLOSURE (S).

The computation of ϵ -CLOSURE (T) is a typical process of searching a graph for nodes reachable from a given set of nodes. In this case the nodes of T are the given set, and the graph consists of the ϵ - labeled edge of the transition diagram only. A simple algorithm to compute ϵ -CLOSURE (T) is shown in fig (12)

```

Begin
    Push all states in T onto STACK;
     $\epsilon$ -CLOSURE (T):=T;
    While STACK not empty do
        Pop S, the top element of STACK, off of stack;
        For each state t with an edge from S to t labeled  $\epsilon$  do
            If t is not in  $\epsilon$ -CLOSURE (T) then
                Begin
                    Add t to  $\epsilon$ -CLOSURE (T);
                    Push t onto STACK;
                End
            End
        End
    End
End

```

Fig (12) Computation of ϵ -CLOSURE

The states of D and their transition are constructed as follows. Initially, let ϵ -CLOSURE (S_0) be a state of D. This state is the start state of D. We

assume each state of D is initially "unmarked". Then perform the algorithm of Fig (13).

```
While there is an unmarked state  $x = (S_1, S_2 \dots S_n)$  of  $D$  do
  Begin
    Mark  $x$ ;
    For each input symbol  $a$  do
      Begin
        Let  $T$  be the set of states to which there is a
        transition on ' $a$ ' from some state  $S_i$  in  $x$ ;

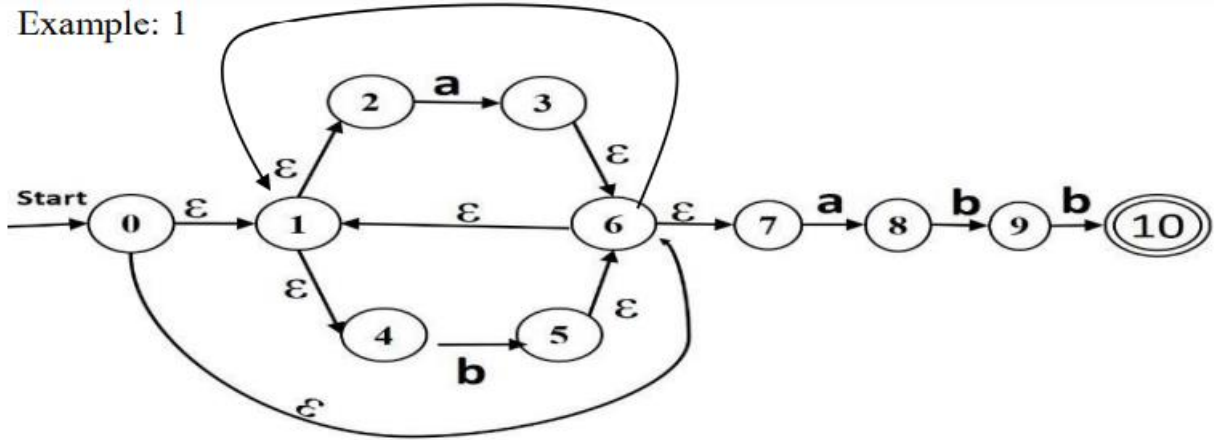
         $y = \epsilon$ -CLOSURE ( $T$ );

        If  $y$  has not yet been added to the set of states of
         $D$  then
          Make  $y$  an "unmarked" state of  $D$ ;

          Add a transition from  $x$  to  $y$  labeled  $a$  if not
          already present
      End
    End
  End
```

Fig (13) The Subset Construction

Example: 1



Regular Expression $R = (a|b)^*abb$

Solution:

E-closure (0) = {0, 1, 2, 4, 7} = A (new state in DFA)

// check every new state with input a, b

Move (A,a) = {3, 8}

E-closure(Move(A,a)) = {3, 6, 1, 2, 4, 7, 8}

// add state 6 because it the connect between (OR part and AND part)
= {1, 2, 3, 4, 6, 7, 8} = B

Move (A, b) = {5}

E-closure (Move(A, b)) = {5, 6, 1, 2, 4, 5, 7}

= {1, 2, 4, 5, 6, 7} = C

Move (B, a) = {3, 8}

E-closure (Move (B, a)) = {1, 2, 3, 4, 6, 7, 8} = B

Move (B, b) = {5, 9}

E-closure (Move (B, b)) = {1, 2, 4, 5, 6, 7, 9} = D

Move (C, a) = {3, 8}

E-closure (Move (C, a)) = {1, 2, 3, 4, 6, 7, 8} = B

Move (C, b) = {5}

E-closure (Move (C, b)) = {1, 2, 4, 5, 6, 7} = C

Move (D, a) = {3, 8}

E-closure (Move (D, a)) = {1, 2, 3, 4, 6, 7, 8} = B

Move (D, b) = {5, 10}

E-closure (Move (D, b)) = {1, 2, 4, 5, 6, 7, 10} = E

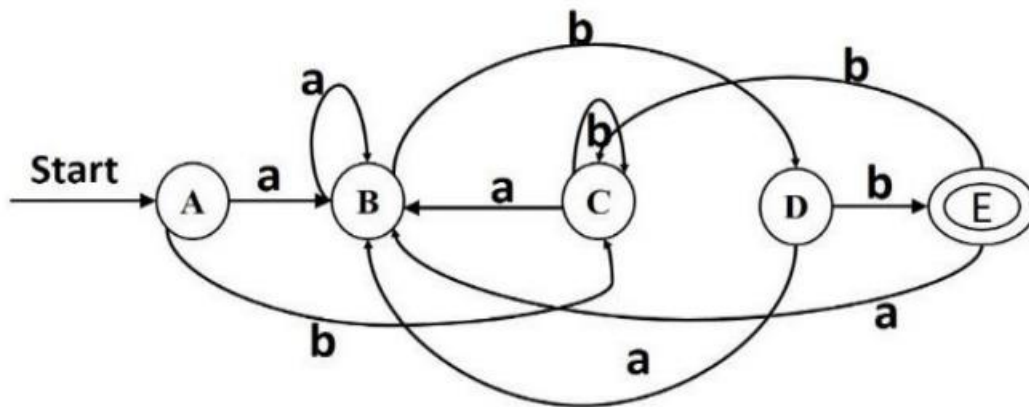
Move (E, a) = {3, 8} = B

Move (E, b) = {5} = C

Transition table:

States	Input System	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

DFA:



4.3 Types of Errors

Common programming errors can occur at many different levels.

- *Lexical* errors include misspellings of identifiers, keywords, or operators — e.g., the use of an identifier `elipseSize` instead of `ellipseSize` — and missing quotes around text intended as a string.
- *Syntactic* errors include misplaced semicolons or extra or missing braces; that is, “{” or “}.” As another example, in C or Java, the appearance of a `case` statement without an enclosing `switch` is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).
- *Semantic* errors include type mismatches between operators and operands. An example is a `return` statement in a Java method with result type `void`.
- *Logical* errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator `=` instead of the comparison operator `==`. The program containing `=` may be well formed; however, it may not reflect the programmer’s intent.

Syntax Analysis

4.4 Introduction

the parser obtains a string of tokens from the lexical analyzer, as shown in Fig. 4.1, and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program.

Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing. In fact, the parse tree need not be constructed explicitly, since checking and translation actions can be interspersed with parsing, as we shall see. Thus, the parser and the rest of the front end could well be implemented by a single module.

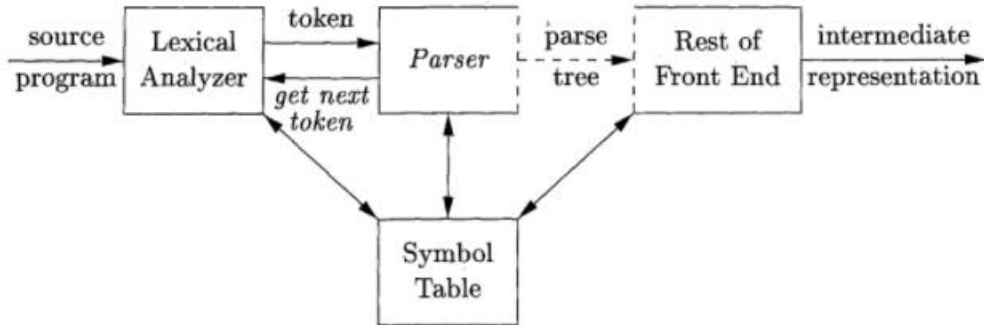


Figure 4.1: Position of parser in compiler model

Due to the limitations of regular expressions the lexical analyzer cannot check the syntax of a given sentence. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

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

4.5 Context-Free Grammars

Grammars systematically describe the syntax of programming language constructs like expressions and statements. 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. The term “token name” is a synonym for “terminal” and frequently we will use the word “token” for terminal when it is clear that we are talking about just the token name. We assume that the terminals are the first components of the tokens output by the lexical analyzer. In (4.4), the terminals are the keywords **if** and **else** and the symbols “(” and “).”
2. *Nonterminals* are syntactic variables that denote sets of strings. In (4.4), *stmt* and *expr* are nonterminals. The sets of strings denoted by nonterminals help define the language generated by the grammar. Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.
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. Conventionally, the productions for the start symbol are listed first.
4. The productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings.

Example:

$$\begin{aligned}
 E &\rightarrow E + T \mid E - T \mid T \\
 T &\rightarrow T * F \mid T / F \mid F \\
 F &\rightarrow (E) \mid \text{id}
 \end{aligned}$$

The notational conventions tell us that E, T, and F are nonterminals, with E the start symbol. The remaining symbols are terminals.

Example: The grammar in Fig. 4.2 defines simple arithmetic expressions.

In this grammar, the terminal symbols are

$$\text{id} + - * / ()$$

The nonterminal symbols are *expression*, *term* and *factor*, and *expression* is the start symbol □

$$\begin{aligned}
 \textit{expression} &\rightarrow \textit{expression} + \textit{term} \\
 \textit{expression} &\rightarrow \textit{expression} - \textit{term} \\
 \textit{expression} &\rightarrow \textit{term} \\
 \textit{term} &\rightarrow \textit{term} * \textit{factor} \\
 \textit{term} &\rightarrow \textit{term} / \textit{factor} \\
 \textit{term} &\rightarrow \textit{factor} \\
 \textit{factor} &\rightarrow (\textit{expression}) \\
 \textit{factor} &\rightarrow \textit{id}
 \end{aligned}$$

Figure 4.2: Grammar for simple arithmetic expressions

4.6 Derivations

Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions. This derivational view corresponds to the top-down construction of a parse tree, but the precision afforded by derivations will be especially helpful when bottom-up parsing is discussed. As we shall see, 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:

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \textit{id} \quad (4.7)$$

The string $-(\textit{id} + \textit{id})$ is a sentence of grammar (4.7) because there is a derivation

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\textit{id} + E) \Rightarrow -(\textit{id} + \textit{id}) \quad (4.8)$$

The strings $E, -E, -(E), \dots, -(\textit{id} + \textit{id})$ are all sentential forms of this grammar. We write $E \stackrel{*}{\Rightarrow} -(\textit{id} + \textit{id})$ to indicate that $-(\textit{id} + \textit{id})$ can be derived from E .

At each step in a derivation, there are two choices to be made. We need to choose which nonterminal to replace, and having made this choice, we must pick a production with that nonterminal as head. For example, the following alternative derivation of $-(\textit{id} + \textit{id})$ differs from derivation (4.8) in the last two steps:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + \textit{id}) \Rightarrow -(\textit{id} + \textit{id}) \quad (4.9)$$

From above we can see that there are two types of derivatives:

1. In *leftmost* derivations, the leftmost nonterminal in each sentential is always chosen. If $\alpha \Rightarrow \beta$ is a step in which the leftmost nonterminal in α is replaced, we write $\alpha \xRightarrow{lm} \beta$.
2. In *rightmost* derivations, the rightmost nonterminal is always chosen; we write $\alpha \xRightarrow{rm} \beta$ in this case.

Derivation (4.8) is **leftmost**, while (4.9) is a **rightmost derivation**.

4.7 Parse Trees and Derivations

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Let us see this by an example, The given string is, $a + b * c$. The given Grammar is $E \rightarrow E * E / E + E / id$.

The left-most derivation is:

$$E \rightarrow E * E$$

$$E \rightarrow E + E * E$$

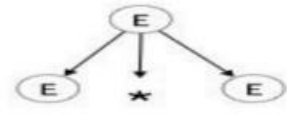
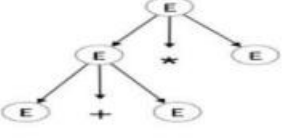
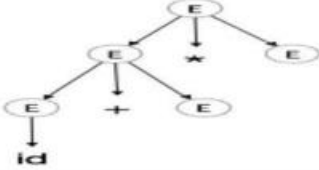
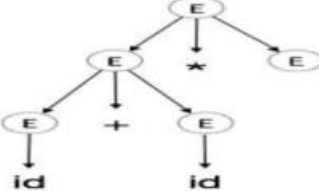
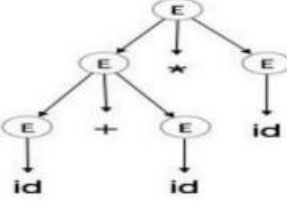
$$E \rightarrow id + E * E$$

$$E \rightarrow id + id * E$$

$$E \rightarrow id + id * id$$

In a parse tree:

1. All leaf nodes are terminals.
2. All interior nodes are non-terminals.
3. In-order traversal gives original input string.

Steps	Parse Tree
Step 1 : $E \rightarrow E * E$	
Step 2: $E \rightarrow E + E * E$	
Step 3: $E \rightarrow id + E * E$	
Step 4: $E \rightarrow id + id * E$	
Step 5: $E \rightarrow id + id * id$	

4.8 Ambiguity

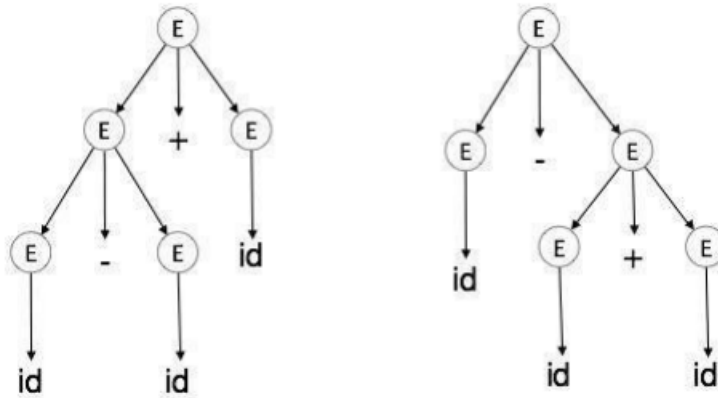
A grammar G is said to be an ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

Example: -

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow id$



For the string $id + id - id$, the above grammar generates two parse trees:

When the non-terminal on the right side of given production depends on the non-terminal on the left side of the same production, the grammar thus formed is called inherently Ambiguous. From the above example, the language generated by an ambiguous grammar is said to be inherently ambiguous. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following associativity and precedence constraints.

4.8.1 Associativity

If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators. If the operation is left-associative, then the operand will be taken by the left operator; or if the operation is right-associative; the right operator will take the operand.

Example

Operations such as Addition, Multiplication, Subtraction, and Division are left associative. If the expression contains:

$id\ op\ id\ op\ id$ it will be evaluated as: $(id\ op\ id)\ op\ id$

For example, $(id + id) + id$

Operations like Exponentiation are right associative, i.e., the order of evaluation in the same expression will be:

$id\ op\ (id\ op\ id)$

For example, $id \wedge (id \wedge id)$

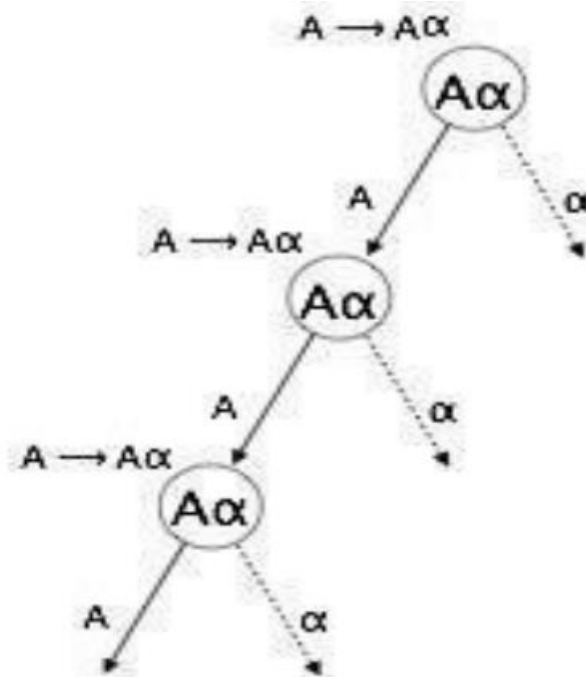
4.8.2 Precedence

If two different operators share a common operand, the precedence of operators decides which will take the operand. That is, $2+3*4$ can have two different parse trees, one corresponding to $(2+3)*4$ and another corresponding to $2+(3*4)$. By setting precedence among operators, this problem can be easily removed. As in the previous example, mathematically $*$ (multiplication) has precedence over $+$ (addition), so the expression $2+3*4$ will always be interpreted as: $2 + (3 * 4)$ These methods decrease the chances of ambiguity in a language or its grammar.

4.9 Elimination of Left Recursion

A grammar is *left recursive* if it has a nonterminal A such that there is a derivation $A \xRightarrow{+} A\alpha$ for some string α . Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.

As shown below:



we can showed how the left-recursive pair of productions $A \rightarrow A\alpha \mid \beta$ could be replaced by the non-left-recursive productions:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

Example

Production Rule No	With Left Recursion If the production is $A \rightarrow A\alpha \mid \beta$	Without Left Recursion $A \rightarrow \beta A' \quad A' \Rightarrow \alpha A' \mid \epsilon$
1	$E \rightarrow E+T \mid T$	$E \rightarrow TE'$ $E' \rightarrow +TE' \mid \epsilon$
2	$T \rightarrow T*F \mid F$	$T \rightarrow FT''$ $T'' \rightarrow *FT'' \mid \epsilon$
3	$F \rightarrow (E) \mid id$	No need for Left Recursion.

4.10 Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

In general, if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two A-productions, and the input begins with a nonempty string derived from α , we do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$. However, we may defer the decision by expanding A to $\alpha A'$. Then, after seeing the input derived from α , we expand A' to β_1 or to β_2 . That is, left-factored, the original productions become

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

Example: if we have the following grammar

$$S \rightarrow i E t S \mid i E t S e S \mid a$$

$$E \rightarrow b$$

Left-factored, this grammar becomes:

$$\begin{aligned} S &\rightarrow i E t S S' \mid a \\ S' &\rightarrow e S \mid \epsilon \\ E &\rightarrow b \end{aligned}$$