Introduction

Programming languages are notations for describing computations to people and to machines.

The world as we know it **depends** on programming languages, because all the software running on all the computers was written in some programming language.

But, before a program can be run, it first must be translated into a form in which it can be executed by a computer. The **software systems** that do this translation are called **compilers**.

Language Processors

Simply stated, a compiler is a program that can read a program in one language — **the source language** — and translate it into an **equivalent** program in another language — **the target language**; see Fig. 1.1.

Note: An important role of the compiler is to **report any errors in the source program** that it detects during the translation process.

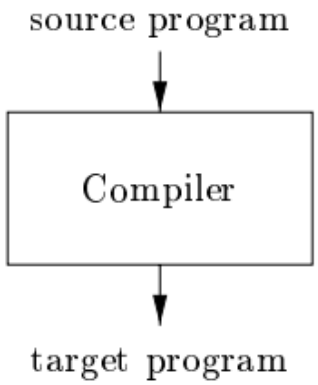


Figure 1.1: A compiler

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

Figure 1.2: Running the target program

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

Figure 1.3: An interpreter

The machine-language target program produced by a **compiler** is usually **much faster** than an **interpreter** at mapping inputs to outputs.

An interpreter, however, can usually give better error **diagnostics** than a compiler, because it executes the source program **statement by statement**.

Example 1.1: Java language processors combine **compilation** and **interpretation**, as shown in Fig. 1.4. A Java source program may first be **compiled into an intermediate** form called **bytecodes**.

The bytecodes are then **interpreted by a virtual machine**. **A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network**.

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

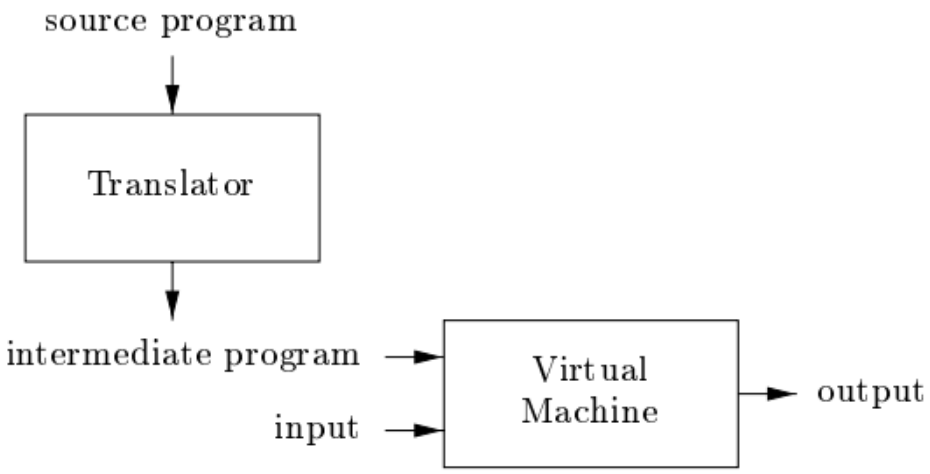


Figure 1.4: A hybrid compiler

In addition to a compiler, several other programs may be required to create an executable target program, as shown in Fig. 1.5.

A source program may be divided into **modules stored in separate files**. The task of collecting the source program is sometimes **entrusted** to a separate program, called a **preprocessor**.

The preprocessor may also **expand shorthand's, called macros**, into source language statements. The modified source program is then fed to a compiler.

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

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

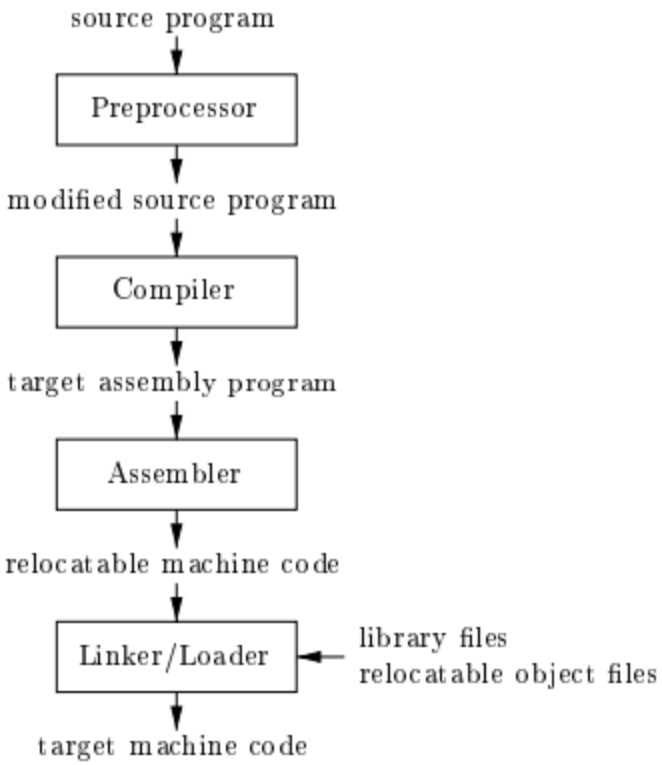


Figure 1.5: A language-processing system

1.2 The Structure of a Compiler

Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program.

If we open up this box a little, we see that there are two parts to this mapping: **analysis** and **synthesis**.

The analysis part **breaks up the source program into constituent pieces** and **imposes a grammatical structure on them**. It then uses this structure to create an intermediate representation of the source program.

If the analysis part detects that the source program is **either syntactically ill formed or semantically unsound**, then it must provide informative messages, so the user can take corrective action.

The analysis part also **collects information about the source program** and stores it in a data structure called a **symbol table**, which is passed along with the intermediate representation to the synthesis part.

The synthesis part **constructs** the **desired target program** from the intermediate representation and the information in the symbol table.

The **analysis** part is often called the **front end** of the compiler; the **synthesis** part is the **back end**.

If we examine the compilation process in more detail, we see that it operates as **a sequence of phases**, each of which **transforms one representation of the source program to another**.

A typical decomposition of a compiler into phases is shown in Fig. 1.6. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The **symbol table,** which stores information about the entire source program, is used by all phases of the compiler.

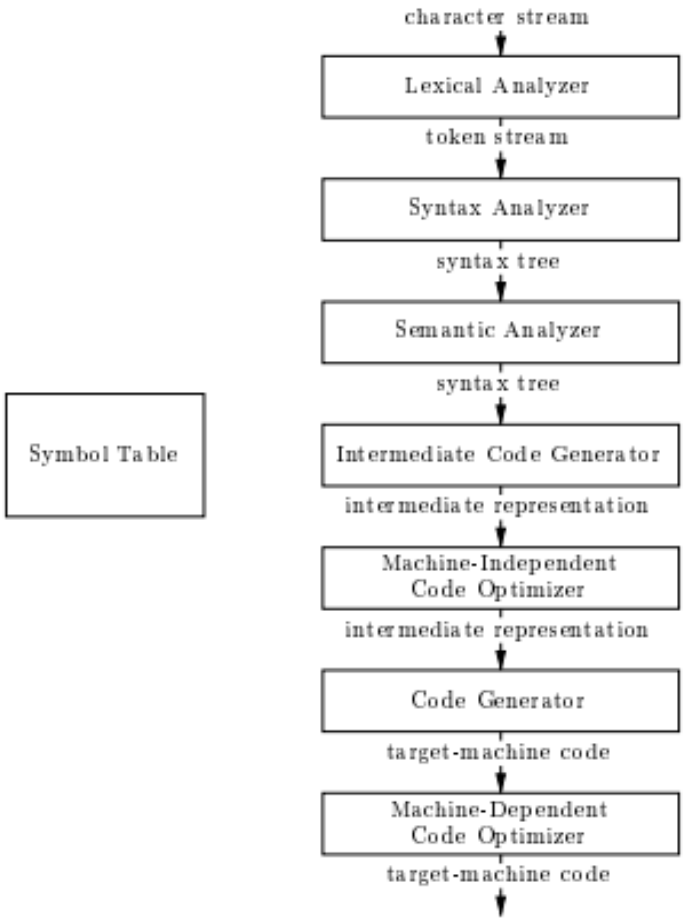


Figure 1.6: Phases of a compiler

Some compilers have a **machine-independent optimization** phase between **the front end** and **the back end**.

The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program since optimization is optional, one or the other of the two optimization phases shown in Fig. 1.6 may be missing.

1.2.1 Lexical Analysis

The first phase of a compiler is called **lexical analysis** or **scanning**. The lexical analyzer reads the stream of characters making up the source program and groups the characters into **meaningful sequences** called **lexemes**. For each lexeme, the lexical analyzer produces as output a token of the form:

(token-name, attribute-value)

That it passes on to the subsequent phase, syntax analysis.

In the token, the first component **token-name** is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the **symbol table** for this token. Information from the **symbol-table entry** is needed for **semantic analysis** and **code generation**. For example, suppose a source program contains the assignment statement

Position = initial + rate * 60 (1.1)

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

1. **Position** is a lexeme that would be mapped into a **token (id, 1)**, where id is an abstract symbol standing for **identifier** and 1 points to the symbol-table entry for position. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

2. The **assignment symbol** = is a lexeme that is mapped into the token **(=).** Since this token **needs no attribute-value**, we have omitted the second component.

3. **initial** is a lexeme that is mapped into the token **(id, 2)**, where 2 points to the symbol-table entry for initial.

4. **+** is a lexeme that is mapped into the token **(+)**.

5. **rate** is a lexeme that is mapped into the token **(id, 3)**, where 3 points to the symbol-table entry for rate.

6. ***** is a lexeme that is mapped into the token **(*)**.

7. **60** is a lexeme that is mapped into the token **(60)**.

Blanks separating the lexemes would be discarded by the lexical analyzer. Figure 1.7 shows the representation of the assignment statement (1.1) after lexical analysis as the sequence of tokens

(id,l) (=) (id, 2) (+) (id, 3) (*) (60) **(1.2)**

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

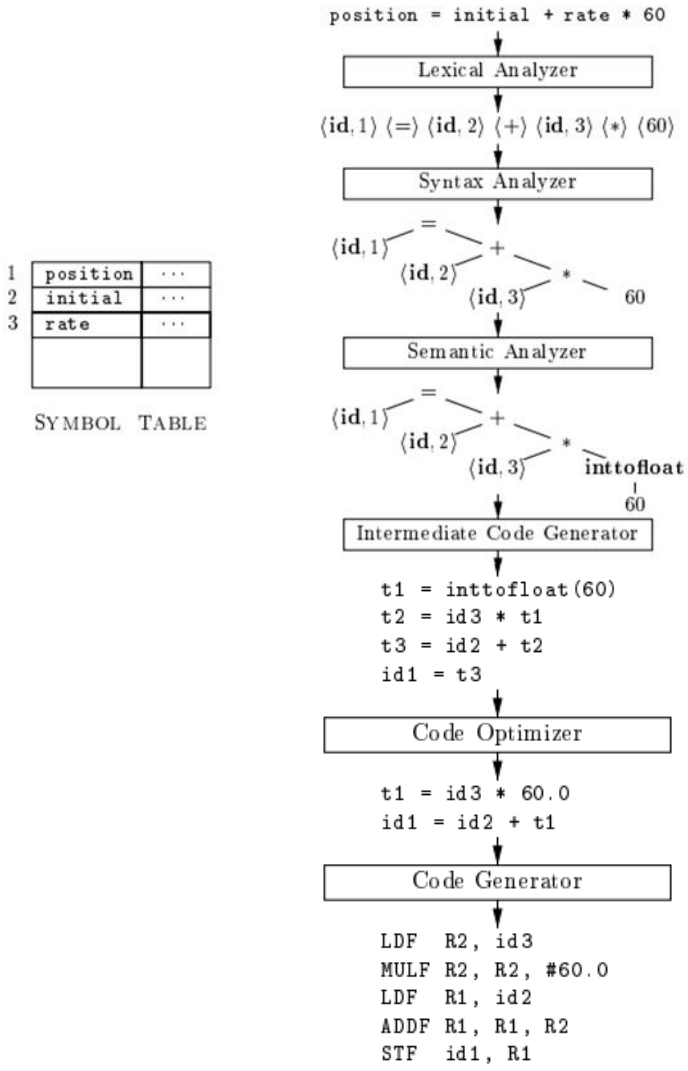


Figure 1.7: Translation of an assignment statement

1.2.2 Syntax Analysis

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

operation. A syntax tree for the token stream (1.2) is shown as the output of the syntactic analyzer in Fig. 1.7. This tree shows the order in which the operations in the assignment

position = initial + rate * 60

Are to be performed. The tree has an interior node labeled * with (id, 3) as its left child and the integer 60 as its right child. The node (id, 3) represents the identifier rate. The node labeled * makes it explicit that we must first multiply the value of rate by 60. The node labeled + indicates that we must add the result of this multiplication to the value of initial.

The root of the tree, **labeled =,** indicates that we must store the result of this addition into the location for the identifier position.

The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program.

1.2.3 Semantic Analysis

The semantic analyzer uses the syntax tree and the information in the symbol table **to check** the source program for semantic consistency with the language definition.

It also gathers type information and saves it in either **the syntax tree** or the **symbol table**, for subsequent use during intermediate-code generation. An **important part of semantic analysis is type checking**, where the compiler checks that each operator has matching operands. **For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array**. The language specification may permit some type conversions called **coercions**. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

Such a coercion appears in Fig. 1.7. **Suppose** that position, initial, and rate have been declared to be **floating-point numbers**, and that the lexeme 60 by itself

forms an **integer**. The type checker in the semantic analyzer in Fig. 1.7 discovers that the operator * is applied to a **floating-point number rate and an integer 60**. In this case, the integer may be converted into a floating-point number. In Fig. 1.7, notice that the output of the semantic analyzer has an extra node for the operator **inttofloat**, which explicitly converts its integer argument into a floating-point number.

1.2.4 Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. **Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis**. After syntax and semantic analysis of the source program, many compilers generate an **explicit** low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be **easy to produce** and it should be **easy to translate** into the target machine.

1.2.5 Code Optimization

The machine-independent code-optimization phase **attempts to improve** the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as **shorter code**, or target code that **consumes less power**. For example, a straight forward algorithm generates the intermediate code (1.3), using an instruction for each operator in the tree representation that comes from the semantic analyzer. A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code. The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the inttofloat operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, t3 is used only once to transmit its value to **idl** so the optimizer can transform (1.3) into the shorter sequence

$$t1=id3 * 60.0$$

$$id1=id2 + t1 \qquad (1.4)$$

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

1.2.6 Code Generation

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

Lexical Analysis (Scanner)

The Role of the Lexical Analyzer

As the first phase of a compiler, the main task of the lexical analyzer:

- Read the input characters of the source program,
- Group them into **lexemes**, and produce as output a sequence of tokens for each lexeme in the source program.

The stream of tokens is sent to **the parser for syntax analysis**. It is common for the lexical analyzer to interact with the **symbol table** as well.

When the lexical analyzer discovers a lexeme **constituting** an identifier, it needs to enter that lexeme into the symbol table.

In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser. These interactions are suggested in Fig. 3.1. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the getNextToken command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parse.

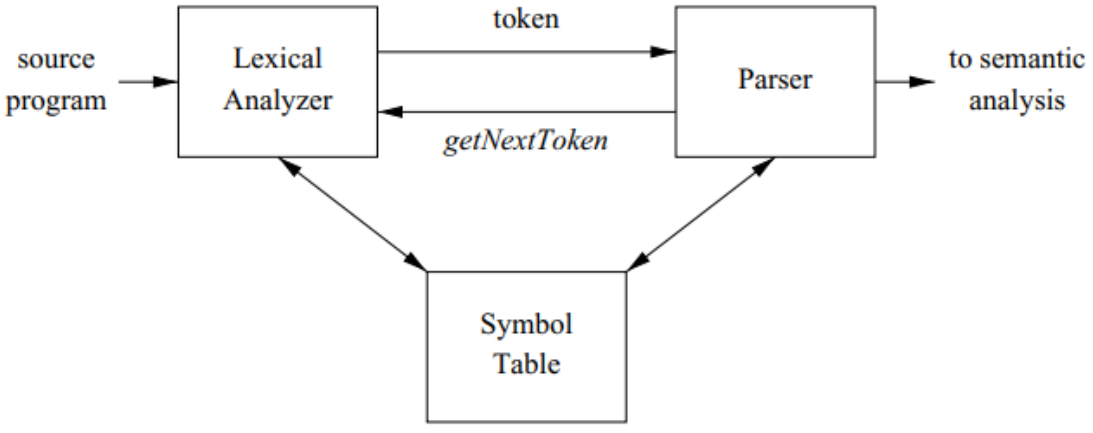


Figure 3.1: Interactions between the lexical analyzer and the parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides **identification of lexemes**. One such task

is **stripping** out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).

Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it **can associate a line number with each error message**.

In some compilers, the lexical analyzer makes a copy of the source program with the error messages **inserted at the appropriate positions**. Sometimes, lexical analyzers are divided into a cascade of two processes:

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

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

Lexical Analysis versus Parsing

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

1. **Simplicity** of design is the most important consideration:
   The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks.

   For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer. If we are designing a new language, separating lexical and syntactic concerns can lead to a cleaner overall language design.

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

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

Tokens, Patterns, and Lexemes

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

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

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

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

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

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

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

Example:
int a = 10; //Input Source code

Tokens
int (keyword), a (identifier), =(operator), 10(constant) and ;(punctuation-semicolon)
Answer – Total number of tokens = 5

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

$$E = M * C ** 2$$

are written below as a sequence of pairs.

<id, pointer to symbol-table entry for E>

<assign_op>

<id, pointer to symbol-table entry for M>

<mult_op>

<id, pointer to symbol-table entry for C>

<exp_op> <number, integer value 2>

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

Lexical Errors

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

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

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

Lexical errors are the errors thrown by your lexer when unable to continue. This means that there's no way to distinguish a lexeme as a valid token for your lexer.

Simple panic-mode error handling system requires that we return to a high-level parsing function when a parsing or lexical error is detected. Panic mode recovery includes:

- Delete one character from the remaining input (the successive character).

  Example: printf**f**---------- printf.

- Insert a missing character in to the remaining input.

  Ex: prite ------- pri**n**tf

- Replace an incorrect character with the correct character.

  Ex: Printf ---------- printf

- Transpose two adjacent characters.

  Ex: **rp**intf ---------- printf

Input Buffering

Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded.

This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme.

For **instance**, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for id. In C, single-character operators like -, =, or < **could also be the beginning of a two-character operator like** ->, ==, or <=. Thus, we shall introduce a two-buffer scheme that handles large lookaheads safely. We then consider an improvement involving "**sentinels**" that saves time checking for the ends of buffers.

Buffer Pairs

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the

amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded, as suggested in Fig. 3.3.

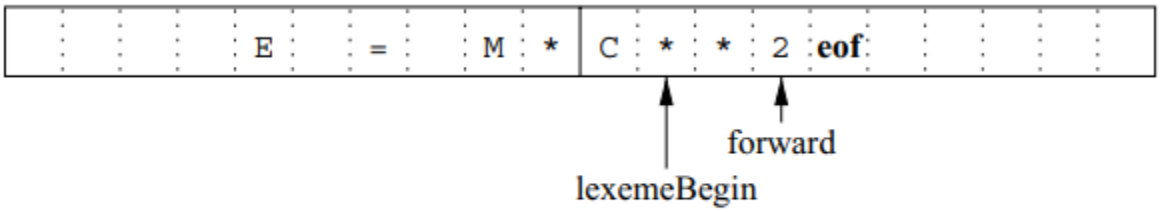


Figure 3.3: Using a pair of input buffers

Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters into a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by eof marks the end of the source file and is different from any possible character of the source program. Two pointers to the input are maintained:

1. Pointer **lexemeBegin**, marks the beginning of the current lexeme, whose extent **we are attempting to determine**.

2. **Pointer forward scans ahead until a pattern match is found**; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, lexemeBegin is set to the character immediately after the lexeme just found.

In Fig. 3.3, we see forward has passed the end of the next lexeme, ** (the Fortran exponentiation operator), and must be retracted one position to its left. Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the

distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

```
switch ( *forward++ ) {
    case eof:
        if (forward is at end of first buffer ) {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer ) {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    Cases for the other characters
}
```

Figure: Lookahead code with sentinels

Regular Expressions

The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules**.**

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols.

Regular expression is an important notation for specifying patterns. Each **pattern matches a set of strings**, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages.

There are a number of algebraic laws that are obeyed by regular expressions, which can be used to manipulate regular expressions into equivalent forms.

Operations on Languages

There are several important operations that can be applied to languages. For lexical Analysis the operations are:

1- **Union**. 2. **Concatenation**. 3. **Closure**.

Operation	Definition
Union L and M written $L \cup M$	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ in } M\}$
Concatenation of L and M written LM	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* denotes "zero or more concatenations of L.
Positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes "one or more concatenations of L.

Example: Let L and D be two languages where L = {a, b, c} and D= {0, 1} then

- Union: LUD = {a, b, c, 0,1}
- Concatenation: LD = {a0,a1, b0, b1, c0,c1}
- Exponentiation: L^2 = LL
- By definition: L^0 ={ Ɛ}

Examples: Let ∑= {a, b}; where ∑: indicates input set (i.e. input alphabet).

1- The RE a | b denotes the set {a, b}

2- The RE (a | b) (a | b) denotes {aa, ab, ba, bb}

3. The RE a* denotes { , a, aa, aaa, aaaa, …….}

4. The RE (a | b)* denotes { , a, b,ab,ba, bba, aaba, ababa, bb,...}
5. The RE a | ba* denotes the set of strings consisting of either signal a or b followed by zero or more a's.
6. The RE a*ba*ba*ba* denotes the set of strings consisting exactly three b's in total.
7. The RE (a | b)* a(a | b)* a(a | b)* a(a | b)* denotes the set of strings that have at least three a's in them.
8. The RE (a | b)* (aa | bb) denotes the set of strings that end in a double letter.
9. The RE Ɛ | a | b | (a | b) 3 (a | b)* denotes to all strings whose length is not two, could be zero, one, three…

[a-z] is all lower-case alphabets of English language.
[A-Z] is all upper-case alphabets of English language.
[0-9] is all natural digits used in mathematics.

Representing occurrence of symbols using regular expressions
letter = [a – z] or [A – Z]
digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]
sign = [+ | -]
Representing language tokens using regular expressions
Decimal = (**sign**)**?**(**digit**)$^{+}$
Identifier = (**letter**)(**letter** | **digit**)*

Digit ⟶ ***0|1|….|9***
Digits ⟶ ***Digit Digit******
Optional Fraction ⟶ ***.Digits| ε***
Optional Exponent ⟶ ***(E(+| -|ε)Digits)|ε***
Number ⟶ ***Digits Optional Fraction Optional Exponent***

By using shorthands it can be represented as:

Digit ⟶ ***[0 – 9]***
Digits ⟶ ***Digit**$^{+}$***
Number ⟶ ***Digits (.Digits)?(E[+-]? Digits)?***

Transition Diagrams

As an **intermediate step** in the construction of a lexical analyzer, we first convert patterns into **stylized flowcharts**, called "transition diagrams." In this section, we perform the conversion from regular-expression patterns to transition diagrams.

Transition diagrams have a collection of nodes or **circles**, called ***states***. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. We may think of a state as summarizing all we need to know about what characters we have seen between the *lexemeBegin* pointer and the *forward* pointer (as in the situation of Fig. 3.3). ***Edges* are directed from one state of the transition diagram to another**. Each edge is ***labeled* by a symbol or set of symbols**.

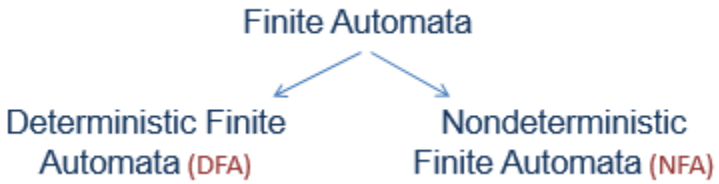
Finite Automata

Regular Expression ⟶ Finite Automata ⟶ Efficient Programs

A finite automaton is a machine that has a finite number of states and a finite number of transitions between these states. A transition between states is usually labeled by a character from the input alphabet.

A finite automaton can be used to decide if an input string is a member in some particular set of strings. To do this, we select one of the states of the automaton as the starting state. We start in this state and in each step, we can read a character from the input and follow a transition labeled by that character.

When all characters from the input are read, we see if the current state is marked as being accepting. If so, the string we have read from the input is in the language defined by the automaton.



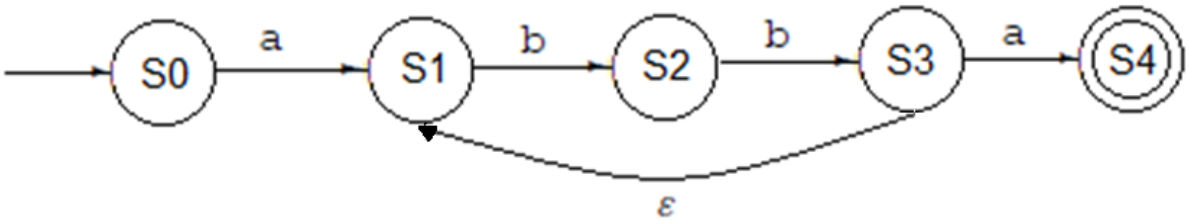
Nondeterministic Finite Automata (NFA)

- It includes transitions marked with *ε, epsilon transitions.*

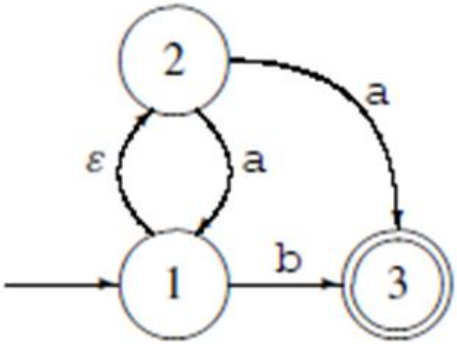
- There are several transitions go out from one state with the same symbol and we have to choose between these transitions. The choice of action is not determined easily by looking at the current state and input. It may be that some choices lead to an accepting state while others do not.

Example : A NFA that can recognize the set of strings that are described by the following regular expression :

1) a(bb)$^+$ a



2) a*(a|b)



A program that decides if a string is accepted by a given NFA will have to check all possible paths to see if any of these accepts the string. This requires either backtracking until a successful path found or simultaneously following all possible paths, both of which are too time-consuming.

Deterministic Finite Automata (DFA)

The deterministic finite automatons are **NFAs**, **but add a number of additional restrictions:**

• There are no **epsilon-transitions**.

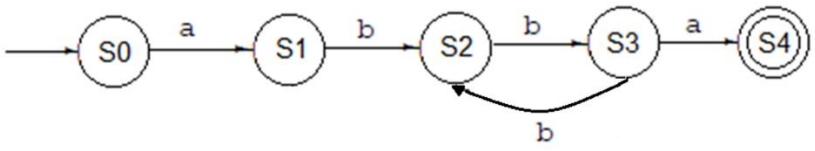
• There may not be two identically labeled transitions out of the same state.

This means that we never have a choice of several next-states: The state and the next input symbol uniquely determine the transition. This is why these automata are called deterministic.

Example:

1) A DFA that can recognize the set of strings that are described by the following regular expression :

$a(bb)^+ a$



Example

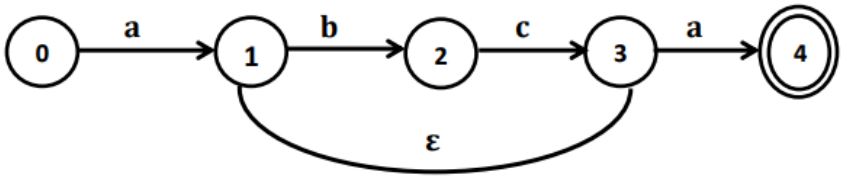
a (b c)⁺a

abca

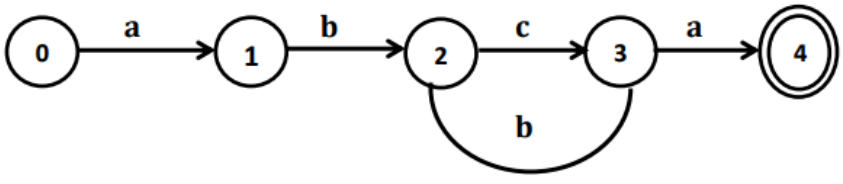
abcbca

abcbcbca

NFA

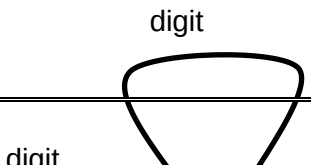


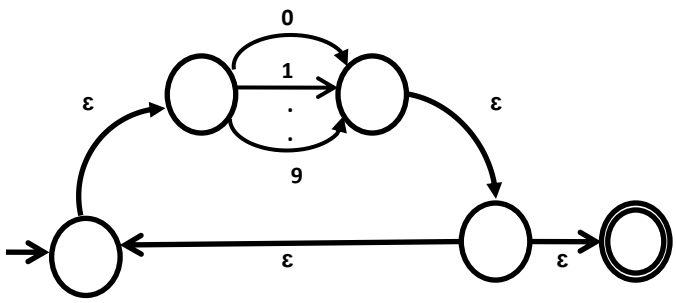
DFA



Numbers

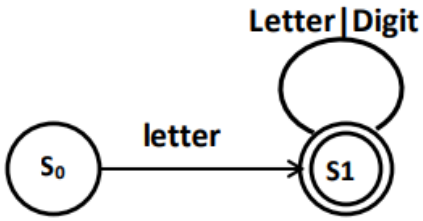
-Integers [0-9]+



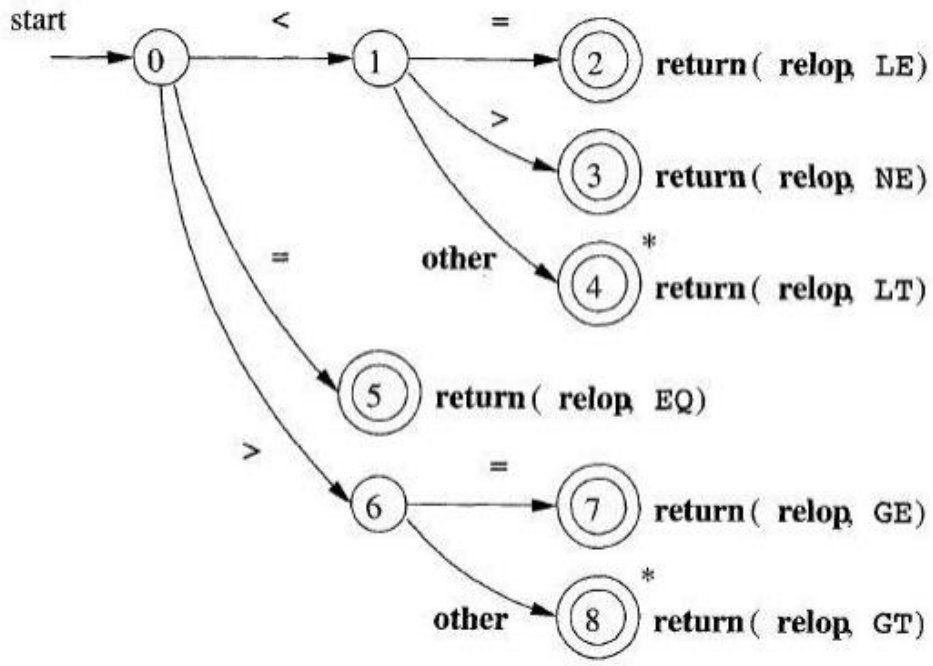


NFA

Transition Diagram for Identifier



Transition Diagram for Relop (Relational Operator)



Syntax Analysis or Parsing

The parser obtains a string of tokens from the lexical analyzer and verifies that the string of **token names** can be generated by the grammar for the source language.

The parser constructs a parse tree and passes it to the rest of the compiler for further processing.

It does so by building a data structure, called a **Parse tree** or **Syntax tree**.

The main goal of syntax analysis is to create a parse tree.

It should also recover from commonly occurring errors so that it can continue processing the remainder of its input.

The methods commonly used in compilers are classified as being either **Top down** or **bottom up**. As indicated by their names, **Top down** parsers build parse trees from the **top (root)** to the **bottom (leaves)** and work up to the root. In both cases, the input to the parser is scanned from **left to right**, one symbol at time.

Features of syntax analysis:

Syntax Trees: Syntax analysis creates a syntax tree.

Context-Free Grammar (CFG): Syntax analysis uses context-free grammar to define the syntax of the programming language.

Top-Down and Bottom-Up Parsing: Syntax analysis can be performed using two main approaches: top-down parsing and bottom-up parsing.

Error Detection: Syntax analysis is responsible for **detecting syntax errors** in the code. If the code does not conform to the rules of the programming language, the parser will report an error and **halt** the compilation process.

Intermediate Code Generation: Syntax analysis generates an intermediate representation of the code. The intermediate representation is usually a more abstract form of the code, which is **easier** to work with than the original source code.

Optimization: Syntax analysis can perform basic optimizations on the code, such as **removing redundant** code and **simplifying expressions**.

Advantages of using syntax analysis in compiler design include:

Structural validation: Syntax analysis allows the compiler to check if the source code follows the grammatical rules of the programming language, which helps to detect and report errors in the source code.

Improved code generation: Syntax analysis can generate a parse tree or abstract syntax tree (AST) of the source code, which can be used in the code generation phase of the compiler design to generate more efficient and optimized code.

Easier semantic analysis: Once the parse tree is constructed, the compiler can perform semantic analysis more easily.

Context-Free Grammars (CFG)

The "context-free grammar," or "grammar" is a notation that is used to specify the syntax of a language, it describe their Hierarchical structure of most programming language constructs.

Formal Definition of a Context-Free Grammar

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

1- **Terminal symbols**, sometimes referred to as "tokens." The terminals are the elementary symbols of the language defined by the grammar. **Terminals** (or "**token**"). For if statement, the terminals are the keywords **if** and **else** and the symbols "(" and ") ."

2- **Nonterminals** are syntactic variables that denote sets of strings. For **if statement**, **stmt** and **expr** are **nonterminals**.

3- **Start Symbol**. 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. The productions for the start symbol are listed first.

4- **Productions**. The productions of **a grammar specify the manner in which the terminals and nonterminals can be combined to form strings**. Each production consists of:

- nonterminal called the **head** or **left side** of the production; this production defines some of the strings denoted by the head.
- The symbol. ⟶ Sometimes **::=** has been used in place of the arrow.

- A body or right side consisting of zero or more terminals and nonterminals.
-

Example : The following grammar defines simple arithmetic expressions.

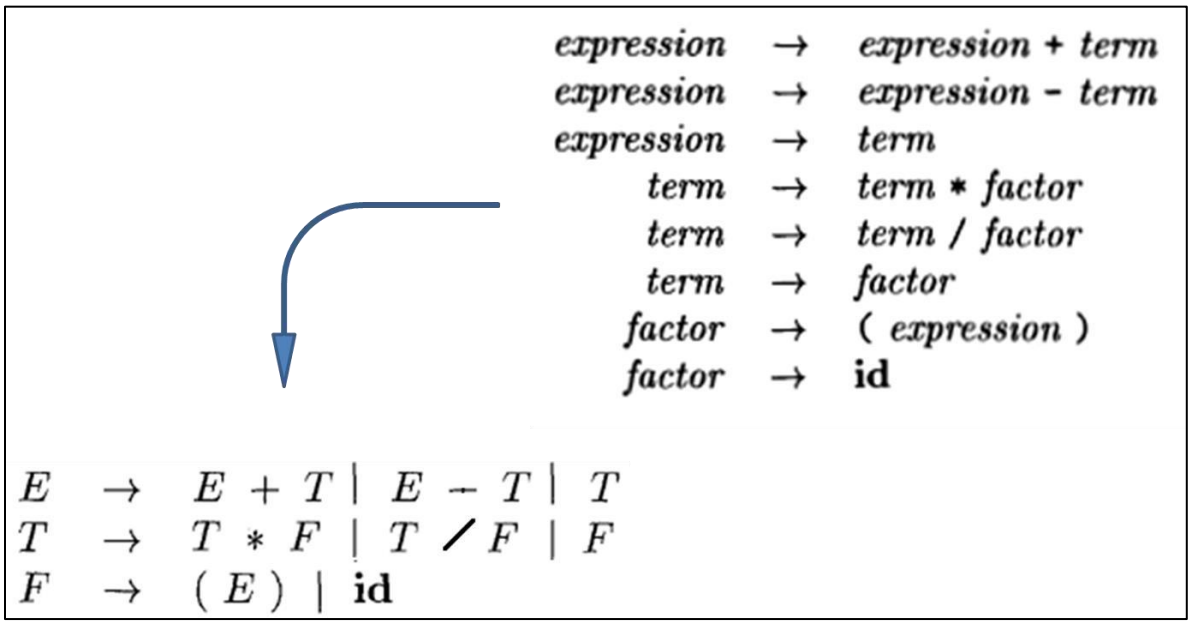
expression	→	*expression* + *term*
expression	→	*expression* - *term*
expression	→	*term*
term	→	*term* * *factor*
term	→	*term* / *factor*
term	→	*factor*
factor	→	(*expression*)
factor	→	**id**

- In this grammar, the terminal symbols are:

id + - * / ( )

- The nonterminal symbols are **expression**, **term** and **factor**,

- The start symbol is **expression**



Parse Tree 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.

Derivation

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.

Left-most Derivation

If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

Production rules:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

Input string: **id + id * id**

The left-most derivation is:

$E \rightarrow E * E$

$E \rightarrow E + E * E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$

Right-most Derivation

If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

The right-most derivation is:

E → E + E

E → E + E * E

E → E + E * id

E → E + id * id

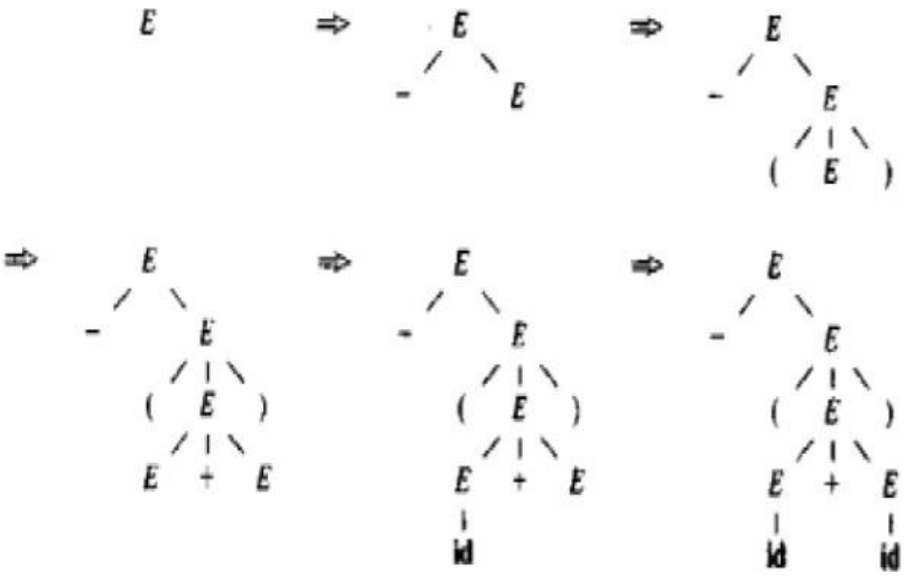
E → id + id * id

For example, the parse tree for -(id+id) implied previously is shown below, For the grammar:

E→ E+E | E-E | E*E | E/E | (E) | -E | id

Sol:-

E→ -E
E→ - (E)
E→ - (E+E)
E→ - (id+E)
E→ - (id+id)

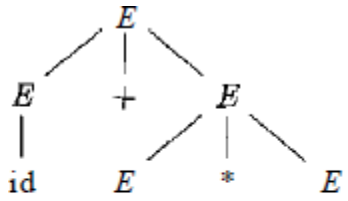


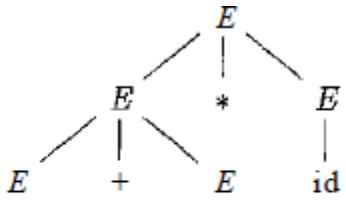
Example:- Consider the previous arithmetic expression grammar, the sentence id+id*id has the two distinct left most derivations:

$$
\begin{aligned}
E &\Rightarrow E+E \\
&\Rightarrow \mathbf{id}+E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
$$

$$
\begin{aligned}
E &\Rightarrow E*E \\
&\Rightarrow E+E*E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
$$

With the two corresponding parse trees shown below:





id id

(a)

id id

(b)

Example:- Parse and derivations, this sentence (id+id*id) by using this grammar:

E→ E+E | E-E | E*E | E/E | (E) | -E | id

Sol 1:-

E→ (E)

E→ (E+E)

E→ (E+E*E)

E→ (id+E*E)

E→ (id+id*E)

E→ (id+id*id)

Sol 2:-

E→ (E)

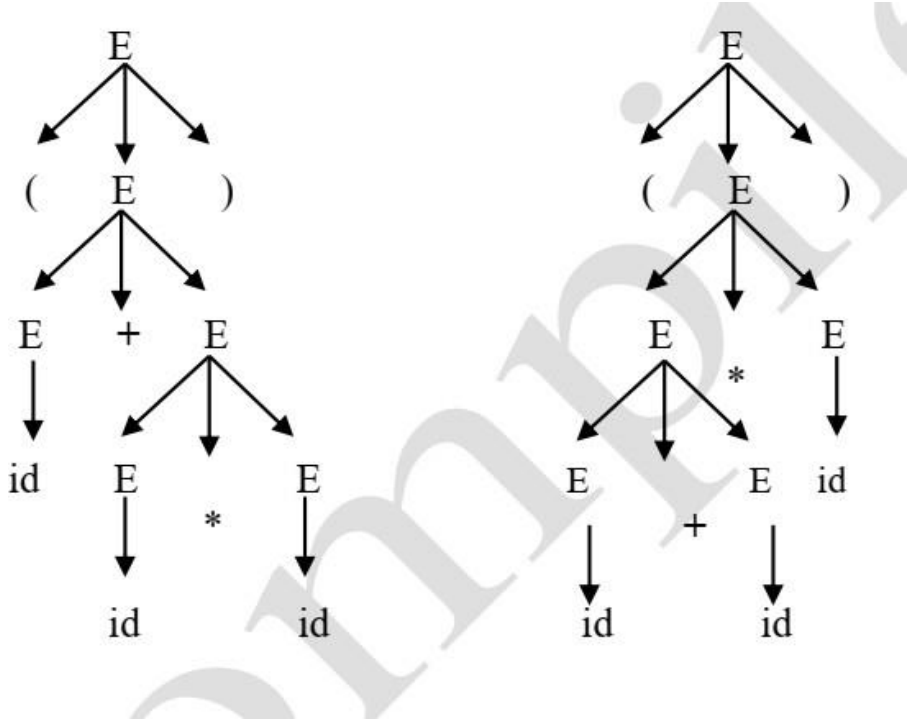
E→ (E*E)

E→ (E+E*E)

E→ (id+E*E)

E→ (id+id*E)

E→ (id+id*id)



H.W

E→ E+E | E-E | E*E | id
Parse this sentence: id+id*id

Ambiguity: (problems of grammar)

A grammar that produces more than one parse tree for some sentence is said to be ambiguous. An ambiguous grammar is one that produces more than one **left most** or more than one **right most derivation** for the same sentence. For certain types of 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.

Ex(1):-
S→ AB | aaB
A→ a | Aa
B→ b
W= aab

Sol (1):-

S→ aaB
S→ aab

Sol (2):-
S→ AB
S→ AaB
S→ aaB
S→aab

Ex (2):-
S→ A
A→ aA | Aa |a

Sol:-
S→ A
S→ aA
S→ aAa
S→ aaa
OR
S→ A
S→ Aa
S→ Aaa
S→ aaa

Sol(2):-
S→ A
S→ aA
S→ aaA
S→ aaa

H.W

Check this sentence Ambiguity or unambiguous id + id + id

E→ E+E | E-E | E*E | id

Left Recursion

A grammar is left recursive if it has a form :

$$A \rightarrow A\alpha$$

Where A is nonterminal and α is some string

Top-down parsing methods cannot handle left-recursive grammars because the parser will loop forever, so a transformation is needed to eliminate left recursion.

Elimination of Left Recursion

The following left-recursive pair of productions

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

could be replaced by the non-left-recursive productions:

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

Example : Grammar for simple arithmetic expressions

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

Non-left-recursive expression grammar is obtained by eliminating immediate left recursion from the expression grammar

$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E'$$
$$T \rightarrow F\ T'$$
$$T' \rightarrow *\ F\ T'$$
$$F \rightarrow (\ E\ )\ \mid\ \mathbf{id}$$

Example: Consider the following grammar for arithmetic expressions.

E→ E+T |T
T→ T*F|F
F→ (E) | id

Eliminating the immediate left recursion (productions of the form A→ Aα) to the production for E and then for T, we will get

E→ T$\bar{E}$

$\bar{E}$ → +T $\bar{E}$ | λ
T→ F$\bar{T}$

$\bar{T}$ → *F$\bar{T}$ | λ
F→ (E) | id

Example:-
A→ ABx | Aa| a

Sol:

A→ ABx | Aa | a
A→ a$\bar{A}$

$\bar{A}$ → Bx$\bar{A}$ | a$\bar{A}$ | λ

Example:-

A→ Ac | Aad | ba | c

Sol:

A→ Ac | Aad | ba | c
A→ bd$\bar{A}$ | c$\bar{A}$

$\bar{A} \rightarrow c\bar{A} \mid ad\bar{A} \mid \lambda$

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.

Example, if we have the two productions

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

on seeing the input **if**, we cannot immediately tell which production to choose to expand *stmt*. 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 $\alpha\beta_2$.

Example : The following grammar abstracts the "dangling-else" problem:

$$\begin{array}{l} S \rightarrow i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a \\ E \rightarrow b \end{array}$$

Here, i, t, and e stand for **if**, **then**, and **else**; E and S stand for "conditional expression" and "statement." Left-factored, this grammar becomes:

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

Example:-
E→ abc | ab | k

Sol:

E→ abc | ab | k

E→ abĒ | k
E→ c | λ

Example:-

S→ abMna | ckL | abMo

Sol:

S→ abMna | ckL | abMo
S → abM$\bar{S}$ | ckL
S→ na | o

Example:-
A→ xB | xY
Sol:
A→ xB | xY
A→ x$\bar{A}$

$\bar{A}$ → B | Y

Example:-
A→ aAB | aA | a

Sol:

A→ aAB | aA | a
A→ a$\bar{A}$

$\bar{A}$ → AB | A | λ

Note : Left factoring in grammars will cause Backtracking in parsers. And that means process the same input more than one time.

Non-recursive Predictive Parsing

In many cases, by carefully writing a grammar eliminating **left recursion** from it, and **left factoring** the resulting grammar, we can obtain a grammar that can be parsed by a non-backtracking predictive parser.

The parser simulate a **leftmost** derivation.

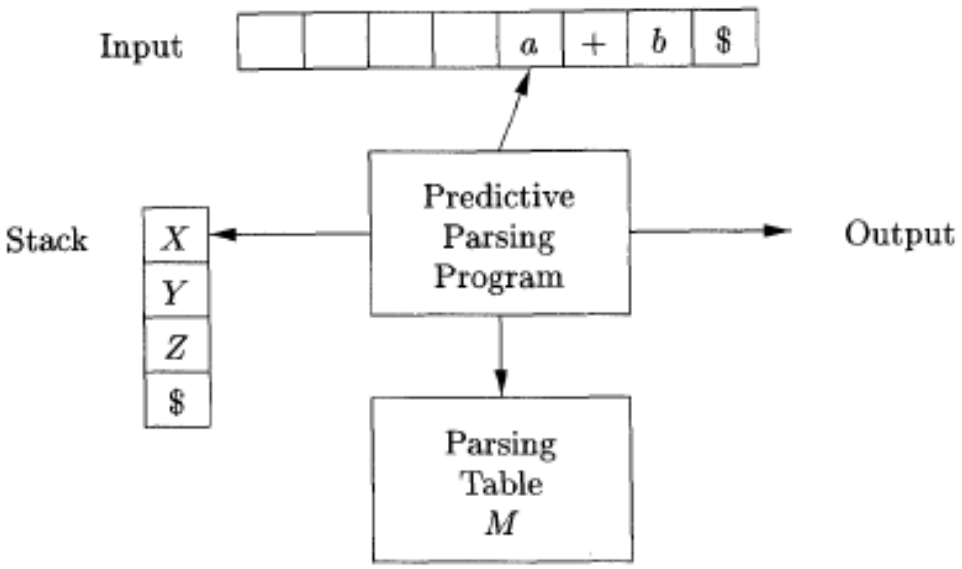
The main problem in predictive parser is to determine **what production will be applied for the non-terminal symbol**.

The non-recursive predictive parser lookup the **parsing table** to find the required production to be used.

A table-driven predictive parser has an **input buffer**, a **stack**, a **parsing table**, and an **output stream**.

The **input buffer** contains **the string to be parsed**, followed by **$**, (a symbol used as a right end marker to indicate the end of the input string).

The **stack** contains a **sequence of grammar symbols with $** on the bottom,(indicating the bottom of the stack). Initially, the stack contains the start symbol of the grammar on the top of $.



Model of a table-driven predictive parser

Parsing table, is a two dimensional array M[A,a] where **A** is non-terminal symbol and **a** is **terminal symbol** or $ and represent the current input symbol.

Example: Consider the following grammar G and the following parsing table M

Grammar G:

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

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

Parsing table M

On input **id + id * id**, the non-recursive predictive parser makes the following sequence of moves. These moves correspond to a leftmost derivation

Stack	Input	Output
$ E	id + id * id $	
$ E' T	id + id * id $	E → T E'
$ E' T' F	id + id * id $	T → F T'
$ E' T' id	id + id * id $	F → id
$ E' T'	+ id * id $	
$ E'	+ id * id $	T' → ε
$ E' T +	+ id * id $	E' → + T E'
$ E' T	id * id $	
$ E' T' F	id * id $	T → F T'
$ E' T' id	id * id $	F → id
$ E' T'	* id $	
$ E' T' F *	* id $	T' → * F T'
$ E' T' F	id $	
$ E' T' id	id $	F → id
$ E' T'	$	
$ E'	$	T' → ε
$	$	E' → ε

FIRST Rules

1- **First (Terminal) → {Terminal}.**
2- **If A→ aα , a is Terminal and α is set of (Terminals / Non-Terminals) so First (A) → {a}.**
3- **If A→ βα and (β ≠ ε) : First (A)→First(β).**
   If A→ βα and (β →ε) : First(A)→{ First(β) - ε} U {First(α)}

Example1

S→AB

A→ a B → b, Find First set for this Grammar

Solution

First(A) ={a} First(B) ={b}
First(S) → =First(AB) = First(A) ={a}

Example2

S→AB

A→a | € B→b, Find First set for this Grammar Solution

Solution

First(A) ={a , €} First(B)={b}

First(S) → =First (AB) = (First(A)-€)U First(B) = {a,b}

Example3

A → abc | def | ghi

A→ abc

A→ def

A→ ghi

Sol:

First A = a,d,g

Example4

S→ AB | b | c

A→ a

First A = a

First S = b,c,a

Example5

E → T$\overline{E}$

$\overline{E}$ → +T$\overline{E}$ | λ

T → F$\overline{T}$

$\overline{T}$ → *F$\overline{T}$ | λ

F → (E) | id

First (E)= { (, id }

First ($\overline{E}$)= { + , € }

First (T)= { (, id }

First ($\overline{T}$)= { * , € }

First (F)= {(, id }

Follow Rules

1- Follow(S)={$}

2- If A→αB , then Follow(B)=Follow(A)

3- If A→αBβ , where (β≠€) … Follow(B)=First(β)

4- If A→αBβ , where (β=€) … Follow(B)={First(β)-€}U Follow(A)

Ex1:-

S1→ S#

S → qABC
A → a | bbD
B → a | €
C → b | €
D → c | €

First **follow**

Sol :-
S1→ { q }
S → { q }
A → { a , b }
B → { a , € }
C → { b , € }
D → { c , € }

S1→ { $ }
S → { # }
A → { a , # }
B → { b , # }
C → { # }
D → { a , # }

Ex2:

	First	**Follow**
E → TE'		
E' → +TE' | ε	First (E)= { (, id }	Follow (E) = { $,) }
T → FT'	First ($\overline{E}$)= { + , € }	Follow ($\overline{E}$) = { $,) }
T' → *FT' | ε	First (T)= { (, id }	Follow (T) = { + , $,) }
F → (E) | id	First ($\overline{T}$)= { * , € }	Follow ($\overline{T}$) = { + , $,) }
	First (F)= {(, id }	Follow (F) = {* , + , $,) }

Example:-

S→ aSb | X
X→ cXb | b
X→ bXZ | X
Z→ n

➢ First	Follow
First (S) = a , c , b	Follow (S) = {$, b }
First (X) = c , b	Follow (X) = {$, b , n }
First (Z) = n	Follow (Z) = {$, b , n }

Ex:-

S → bXY
X → b | c
Y → b | ε

	First	Follow
S	b	$
X	b , c	b , $
Y	b , ε	$

Ex:

S → ABb | bc
A → ε | abAB
B → bc | cBS

	First	
S	**b , a**	**$, b , c , a**
A	**ε , a**	**b , c**
B	**b , c**	**b , c , a**

ε

Constructing parsing Table

1- Fill the entries with productions depending on First Set of Non-Terminals
2- If First(Non-Terminal) contains ε , Then fill the entries with ε depending on Follow Set of Non-Terminals.

Ex:

E→ E+T | T
T → T*F | F
F → (E) | id

Parse the input id * id + id by using predictive parsing:

1- We must solve the **left recursion** and **left factoring** if it founded in the grammar

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

2- We must find the first and follow to the grammar:

	First	Follow
E	(, id	$,)
E'	+ , ε	$,)
T	(, id	+ , $,)
T'	* , ε	+, $,)
F	(, id	+ , *, $,)

3- We must find or construct now the predictive parsing table

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

LL (1) grammars:

The previous algorithm can be applied to any grammar G to produce a parsing **table M**. For some grammars, M may have some entries that are **multiply defined**. If G is left recursive or ambiguous, then M will have at least one **multiply-defined entry**.

Ex:

First(S) = {i, a} follow(S) = {$, e}
First($\overline{S}$) = {e, ε} follow($\overline{S}$) = { $, e }
First (E) = {b} follow (E) = {t}

So the parsing table for our grammar is:

NONTER-MINAL	INPUT SYMBOL					
	a	*b*	*e*	*i*	*t*	$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

The entry for M[S',e] contains both S'→ eS and S'→ ε, since FOLLOW(S') ={e, $}. The grammar is ambiguous and the ambiguity is manifested by a choice in what production to use when an e (else) is seen. We can resolve the ambiguity if we choose S'→ eS. Note that the choice S'→ ε would prevent e from ever being put on the stack or removed from the input, and is therefore surely wrong.

A grammar whose parsing table has no multiply-defined entries is said to be LL(1). The first "L" in LL(1) indicates the reading direction (left-to-right), the second "L" indicates the derivation order (left), and the "1" indicates that there is a one-symbol or lookahead at each step to make parsing action decisions.

Error Detection and Reporting

An error is detected during predictive parsing when **the terminal on top of the stack does not match the next input symbol** or **when non-terminal A is on top of the stack, a is the next input symbol, and the parsing table entry M[A,a] is empty**.

Error recovery is based on the idea of **skipping symbols** on the input until a token in selected set of synchronizing tokens appears. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice.

We can place all symbols in Follow (A) into the synchronizing set for non-terminal A. If we skip tokens until an element of Follow (A) is seen and pop A from the stack, it is likely that parsing can continue.

Example:
Using Follow symbols as synchronizing tokens works reasonably well when expressions are parsed according to the grammar:

$E \rightarrow T\overline{E}$

$\overline{E} \rightarrow +T\ \overline{E} \mid \lambda$
$T \rightarrow F\overline{T}$

$\overline{T} \rightarrow *F\overline{T} \mid \lambda$
$F \rightarrow (E) \mid id$

The parsing table for this grammar is repeated with synchronizing tokens If the parser looks up entry M[A,a] and finds that it is blank, then the input symbol a is skipped. If the entry is synchronize, then the nonterminal on top of the stack is popped in an attempt to resume parsing.

Non-terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→∈	E'→∈
T	T→FT'			T→FT'		
T'		T'→∈	T'→*FT'		T'→∈	T'→∈
F	F→id			F→(E)		

From Follow sets. Pop top of stack NT

"synch" action

Skip input symbol

STACK	INPUT	Remark
$E	+ id * + id$	error, skip +
$E	id * + id$	
$E'T	id * + id$	
$E'T'F	id * + id$	
$E'T' id	id * + id$	
$E'T'	* + id$	
$E'T'F *	* + id$	
$E'T'F	+ id$	error, M[F,+] = synch
$E'T'	+ id$	F has been popped
$E'	+ id$	
$E'T+	+ id$	
$E'T	id$	
$E'T'F	id$	
$E'T' id	id$	
$E'T'	$	
$E'	$	
$	$	

Possible Error Msg: "Misplaced + I am skipping it"

Possible Error Msg: "Missing Term"

What should a parser do to handle the error?

The parser design should be able to provide an error message (an error message which depicts as much possible information as it can). It should be recovering from that error case, and it should be able to continue parsing with the rest of the input.

Error Recovery Techniques:

Panic-Mode Error Recovery: In Panic-Mode Error Recovery the technique is skipping the input symbols until a synchronizing token is found.

- If the parser looks up entry M[A,a] and finds that it is blank, the input symbol a is skipped.
- If the entry is **synch**, the the non-terminal on top of the stack is popped.
- If a token on top of the stack does not match the input symbol, then we pop the token from the stack

Phrase-Level Error Recovery: Each empty entry in the parsing table is filled with a pointer to a specific error routing take care of that error case.

Example | let the following CFG

$\boldsymbol{S \rightarrow AbS|e|\epsilon}$

$\boldsymbol{A \rightarrow a|cAd}$

Synchronization tokens are b and d , follow(A)={b , d}

Non-Terminal	Input symbol					
	a	b	c	d	e	$
S	AbS		AbS		e	ε
A	a	Synch	cAd	Synch		

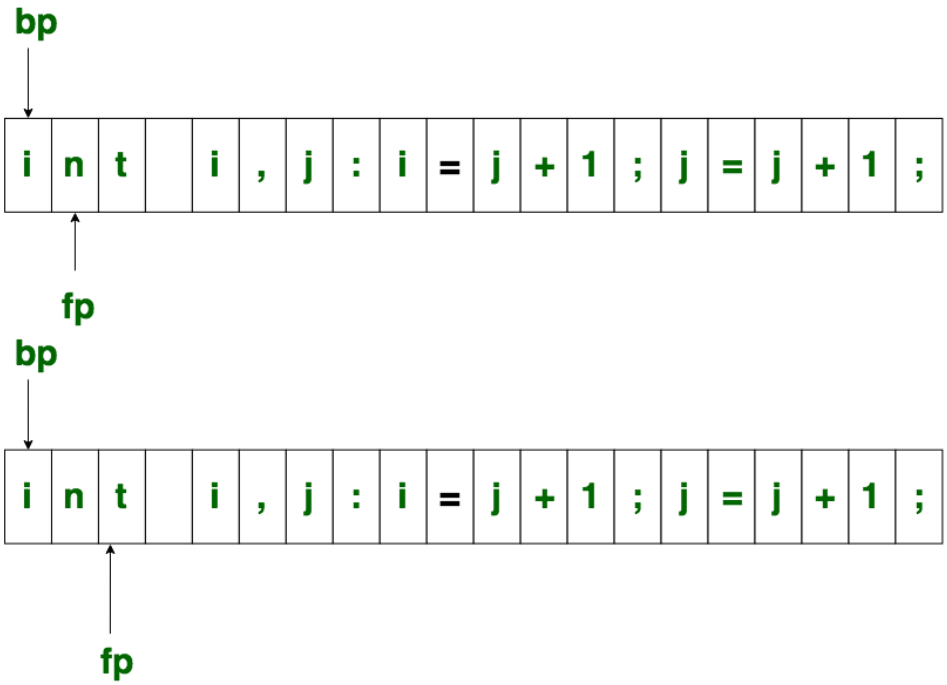
Let us check the input (aab)

Stack	input	Action
$S\$$	$aab\ \$$	AbS
$AbS\$$	$aab\$$	a
$abS\$$	$aab\ \$$	$match\ a$
$bS\$$	$ab\$$	Error missing b
$S\$$	$ab\$$	AbS
$Abs\$$	$ab\$$	a
$abS\$$	$ab\$$	$match\ a$
$bS\$$	$b\$$	$match\ b$
$S\$$	$\$$	ϵ
$\$$	$\$$	$Accept$

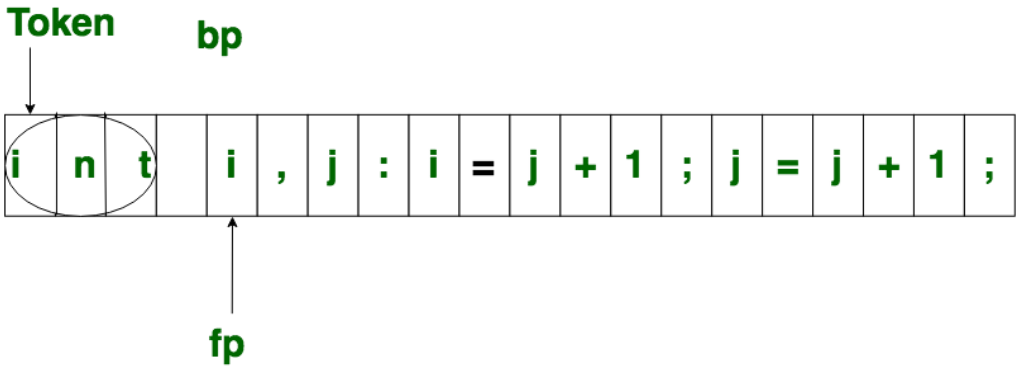
Now let us check the input (ceadb)

Stack	input	Action
$S\$$	$ceadb\ \$$	AbS
$AbS\$$	$ceadb\$$	cAd
$cAdbS\$$	$ceadb\$$	$match\ c$
$AdbS\$$	$eadb\ \$$	*Error unexpected e , skip symbols till b or d , pop A*
$dbS\$$	$db\$$	$match\ d$
$bS\$$	$b\$$	$match\ b$
$S\$$	$\$$	ϵ
$\$$	$\$$	$Accept$

Initially both the pointers point to the first character of the input string as shown



Input Buffering



Input buffering

The forward **ptr** moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as ptr (**fp**) encounters a blank space the lexeme “int” is identified. The fp will be moved ahead at white space, when fp encounters white space, it ignore and moves ahead. then both the begin ptr(bp) and forward ptr(fp) are set at next token. The input character is thus read from secondary storage, but reading in this way from

secondary storage is costly. hence buffering technique is used. A block of data is first read into a buffer, and then second by lexical analyzer

Examples of Regular Expression

Example 1:

Write the regular expression for the language accepting all the string which are **starting** with 1 and **ending** with **0**, over $\sum = \{0, 1\}$.

Ans\\

In a regular expression, the first symbol should be 1, and the last symbol should be 0. The r.e. is as follows:

$$\mathbf{R = 1\ (0|1)^*\ 0}$$

Example 2:

Write the regular expression for the language **starting** and **ending** with **a** and having any having any combination of **b's** in between.

Ans\\

The regular expression will be:

$$\mathbf{R = a\ b^*\ a}$$

Example 3:

Write the regular expression for the language starting with a but not having consecutive b's.

Ans\\: The regular expression has to be built for the language:

L = {a, aba, aab, aba, aaa, abab, ...}

The regular expression for the above language is:

$$\mathbf{R = a^+\ (\ ba\ )^*((ba)^*|b)}$$

Example 4:

Write the regular expression for the language accepting all the string in which any number of **a's** is followed by any number of **b's** is followed by any number of **c's**.

Ans\\:

R = a* b* c*

Example 5:

Write the regular expression for the language over ∑ = {0} having even length of the string.

Ans\\

The regular expression has to be built for the language:

L = {ε, 00, 0000, 000000, ...}

The regular expression for the above language is:

R = (00)*

Example 6:

Write the regular expression for the language having a string which **should have atleast one 0 and atleast one 1.**

Ans\\

The regular expression will be:

R = [(0 | 1)* 0 (0 | 1)* 1 (0 + 1)*] | [(0 + 1)* 1 (0 + 1)* 0 (0 + 1)*]

Example 7:

Describe the language denoted by following regular expression

R = (b* (aaa)* b*)*

Ans\\:

The language can be predicted from the regular expression by finding the meaning of it. We will first split the regular expression as:

Re = (any combination of b's) (aaa)* (any combination of b's)

L = {The language consists of the string in which a's appear triples, there is no restriction on the number of b's}

Example 8:

Write the regular expression for the language L over $\sum = \{0, 1\}$ such that all the string do not contain the substring 01.

Solution:

The Language is as follows:

L = {ε, 0, 1, 00, 11, 10, 100, ...}

The regular expression for the above language is as follows:

R = (1* 0*)

Example 9:

Write the regular expression for the language containing the string over {0, 1} in which there are at least two occurrences of 1's between any two occurrences of 1's between any two occurrences of 0's.

Ans\\ At least two 1's between two occurrences of 0's can be denoted by **(0111*0)*.**

Similarly, if there is no occurrence of 0's, then any number of 1's are also allowed. Hence the r.e. for required language is:

R = (1 | (0111*0))*

Example 10:

Write the regular expression for the language containing the string in which every 0 is immediately followed by 11.

Ans\\

The regular expectation will be:

R = (011 | 1)*

NFA (Non-Deterministic finite automata)

It is easy to construct an NFA than DFA for a given regular language.

Every NFA is not DFA, but each NFA can be translated into DFA.

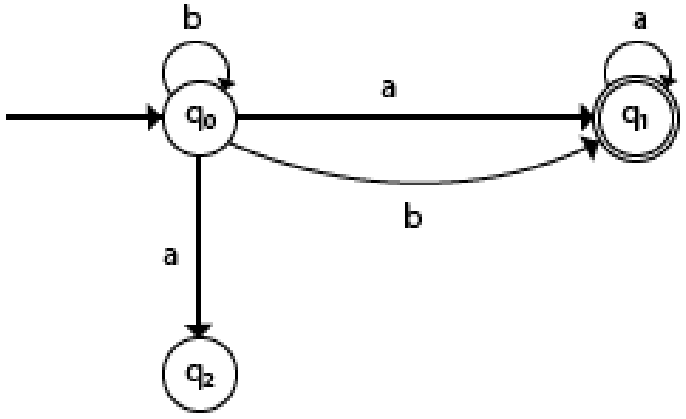


Fig:- NDFA

NFA also has five states same as DFA, but with different transition function, as shown follows:

Q: finite set of states

∑: finite set of the input symbol

q0: initial state

F: final state

Graphical Representation of an NFA

An NFA can be represented by digraphs called state diagram. In which:

The state is represented by vertices.

The arc labeled with an input character show the transitions.

The initial state is marked with an arrow.

The final state is denoted by the double circle.

Example 1:

Q = {q0, q1, q2}

∑ = {0, 1}

q0 = {q0}

F = {q2}

Solution:

Transition diagram:

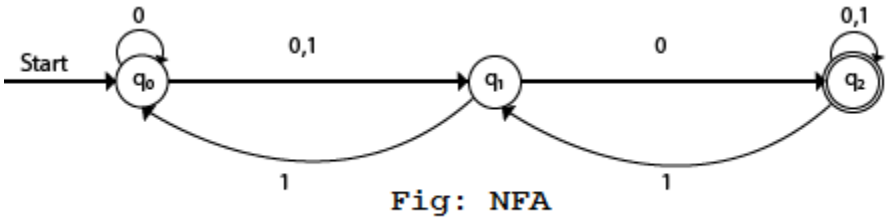


Fig: NFA

Transition Table:

Present State	Next state for Input 0	Next State of Input 1
→**q0**	q0, q1	q1
q1	q2	q0
q2	q2	q1, q2

In the above diagram, we can see that when the current state is q0, on input 0, the next state will be q0 or q1, and on 1 input the next state will be q1. When the current state is q1, on input 0 the next state will be q2 and on 1 input, the next state will be q0. When the current state is q2, on 0 input the next state is q2, and on 1 input the next state will be q1 or q2.

Example 2:

NFA with ∑ = {0, 1} accepts all strings with 01.

Ans\\

Fig: NFA

Transition Table:

Present State	Next state for Input 0	Next State of Input 1
→**q0**	q1	ε
q1	ε	q2
q2	q2	q2

Example 3:

NFA with ∑ = {0, 1} and accept all string of length atleast 2.

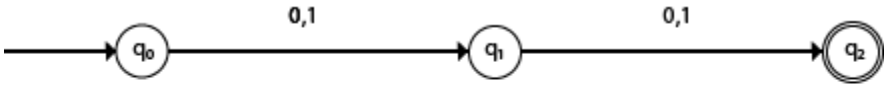


Fig: NFA

Transition Table:

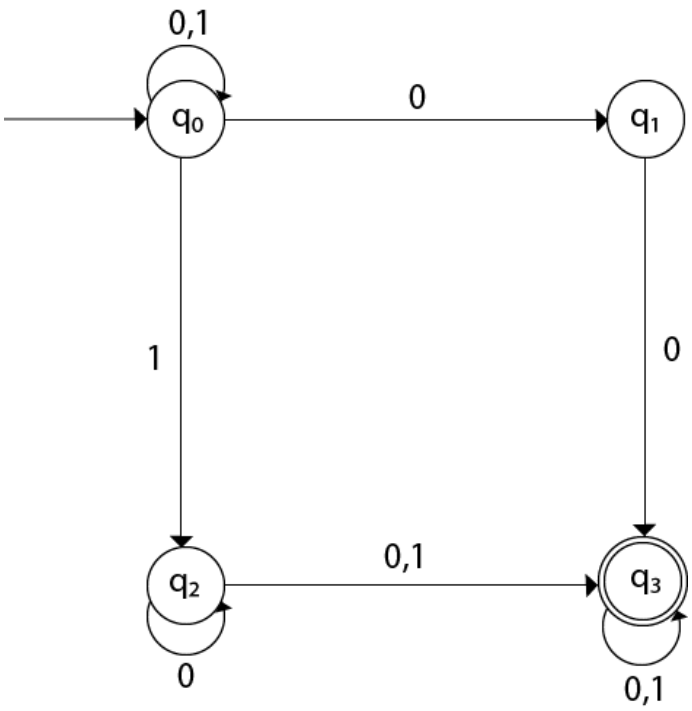
Present State	Next state for Input 0	Next State of Input 1
→**q0**	q1	q1
q1	q2	q2
***q2**	ε	ε

Example 1:

Design a NFA for the transition table as given below:

Present State	0	1
→q0	q0, q1	q0, q2
q1	q3	ε
q2	q2, q3	q3
→q3	q3	q3

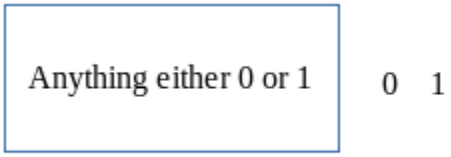
The transition diagram can be drawn by using the mapping function as given in the table.

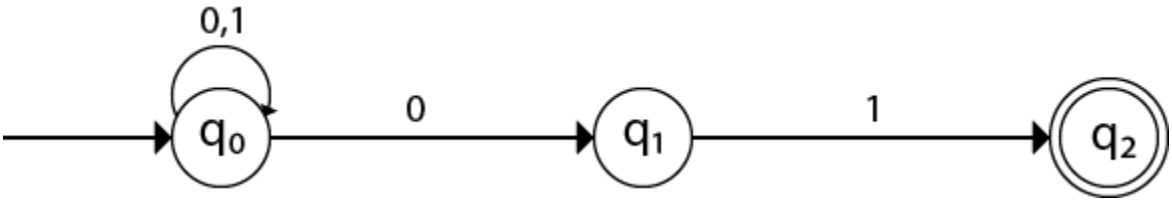


Example 2:

Design an NFA with ∑ = {0, 1} accepts all string ending with 01.

Anx



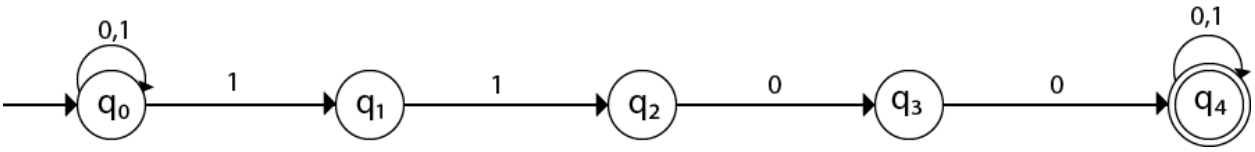


Example 3:

Design an NFA with ∑ = {0, 1} in which double '1' is followed by double '0'.

Ans

The FA with double 1 is as follows:

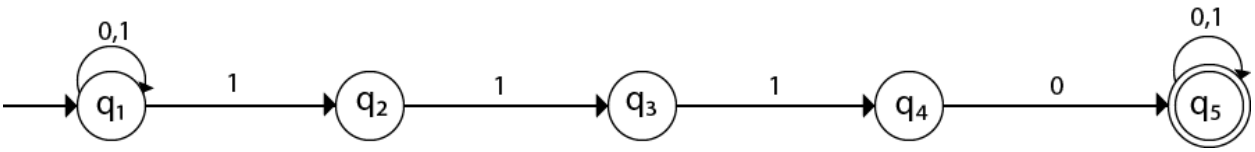


Now considering the string 01100011

q0 → q1 → q2 → q3 → q4 → q4 → q4 → q4

Example 4:

Design an NFA in which all the string contain a substring 1110.



Transition table for the above transition diagram can be given below:

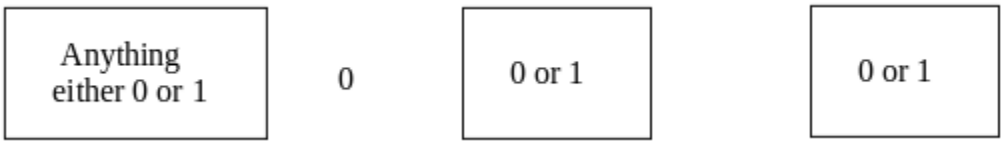
Present State	0	1
→**q1**	q1	q1, q2
q2		q3
q3		q4
q4	q5	
q5	q5	q5

Consider a string 111010,

Example 5:

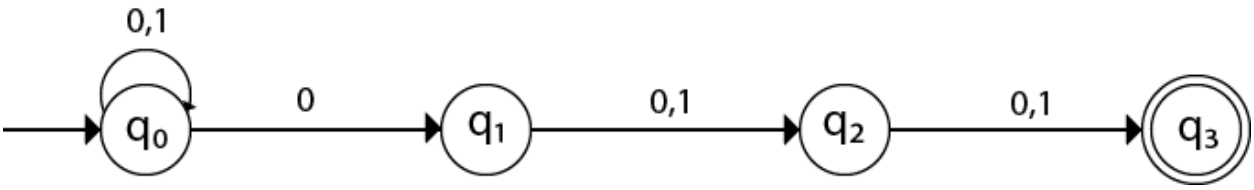
Design an NFA with ∑ = {0, 1} accepts all string in which the third symbol from the right end is always 0.

Ans



Examples of NFA

Thus we get the third symbol from the right end as '0' always. The NFA can be:



DFA (Deterministic finite automata)

In DFA, there is only one path for specific input from the current state to the next state.

DFA does not accept the null move, i.e., the DFA cannot change state without any input character.

DFA can contain multiple final states. It is used in Lexical Analysis in Compiler.

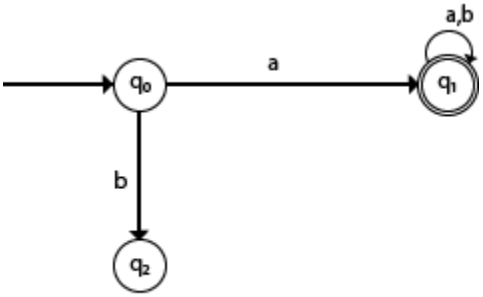


Fig:- DFA \

In the following diagram, we can see that from state q0 for input a, there is only one path which is going to q1. Similarly, from q0, there is only one path for input b going to q2.

Formal Definition of DFA

A DFA is a collection of 5-tuples same as we described in the definition of FA.

Q: finite set of states

∑: finite set of the input symbol

q0: initial state

F: final state

Example 1:

Q = {q0, q1, q2}

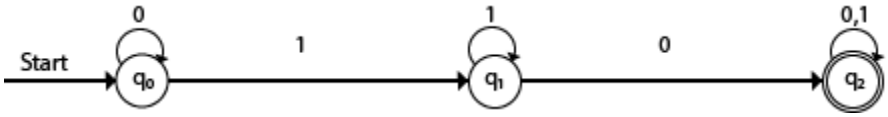
∑ = {0, 1}

q0 = {q0}

F = {q2}

Ans

Transition Diagram:

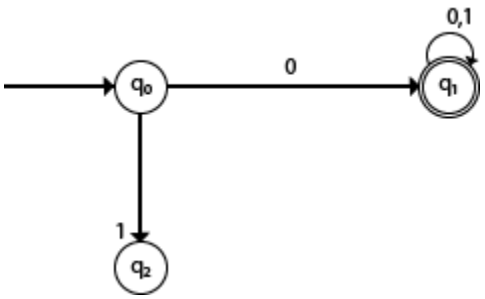


Transition Table:

Present State	Next state for Input 0	Next State of Input 1
→**q0**	q0	q1
q1	q2	q1
q2	q2	q2

Example 2:

DFA with ∑ = {0, 1} accepts all starting with 0.



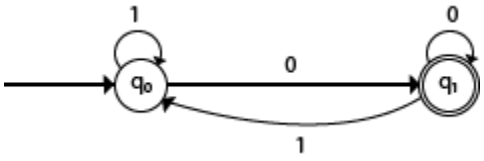
Explanation:

In the above diagram, we can see that on given 0 as input to DFA in state q0 the DFA changes state to q1 and always go to final state q1 on starting input 0. It can accept 00, 01, 000, 001....etc. It can't accept any string which starts with 1, because it will never go to final state on a string starting with 1.

Example 3:

DFA with ∑ = {0, 1} accepts all ending with 0.

Ans\\



Explanation:

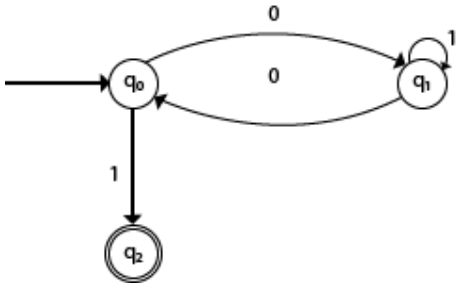
In the above diagram, we can see that on given 0 as input to DFA in state q0, the DFA changes state to q1. It can accept any string which ends with 0 like 00, 10, 110, 100....etc. It can't accept any string which ends with 1, because it will never go to the final state q1 on 1 input, so the string ending with 1, will not be accepted or will be rejected.

Example 4:

Design a FA with ∑ = {0, 1} accepts the strings with an even number of 0's followed by single 1.

Ans\\

The DFA can be shown by a transition diagram as:

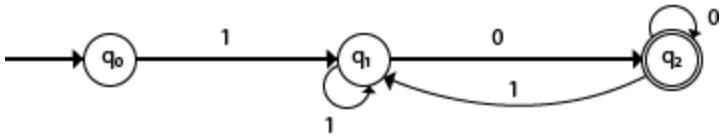


Example 5:

Design a FA with ∑ = {0, 1} accepts those string which starts with 1 and ends with 0.

Ans\\

The FA will have a start state q0 from which only the edge with input 1 will go to the next state.



In state q1, if we read 1, we will be in state q1, but if we read 0 at state q1, we will reach to state q2 which is the final state. In state q2, if we read either 0 or 1, we will go to q2 state or q1 state respectively. Note that if the input ends with 0, it will be in the final state.

Example 6:

Design FA with $\sum = \{0, 1\}$ accepts the set of all strings with three consecutive 0's.

Ans\\

The strings that will be generated for this particular languages are 000, 0001, 1000, 10001, ... in which 0 always appears in a clump of 3. The transition graph is as follows: