Compiler Construction

Rationale

Language translation techniques play a vital role in many fields of Computer Technology. The need for language translators arises because the language in which one wants to express ones computational needs is significantly different from the language recognised by a real machine. Development of translators, therefore, have resulted in two important fallouts. On one hand it has a strong influence on the design of new programming languages, on the other the issue of efficient translation of programming languages is inherently linked with the design of new architectures for higher throughputs. Language translation also plays an important role in the design of command languages and user interfaces of software systems, which require knowledge of scanning and parsing techniques.

In the current context, it is essential for computer science undergraduate students to develop a thorough knowledge of the theory and techniques of language translation. This course is aimed at imparting this knowledge through a detailed treatment of the theory of language translation, and a familiarity with the use of currently available software tools for automatic generation of various modules of a language translator.

Objectives

At the end of this course on Compiler Construction, the student should be able to

1. Understand the two models of language processing, viz. compilers and interpreters, and realize their strengths and limitations.

2. Understand the influence of various features of programming languages and machine architectures on the complexity and efficiency of language translation.

3. Appreciate the role of lexical analysis in compilation, understand the theory behind design of lexical analysers and lexical analyser generators, and be able to use Lex to generate lexical analysers.

4. Be proficient in writing grammars to specify syntax, understand the theories behind different parsing strategies – their strengths and limitations, understand how generation of parsers can be automated, and be able to use Yacc to generate parsers.

5. Understand the error detection capabilities of a compiler and how to report them in a user friendly manner.

6. Understand the need for various static semantic analyses such as declaration processing, type checking, and intermediate code generation, and be able to perform such analysis in a syntax directed fashion through the use of attributed definitions.

7. Understand the issues involved in allocation of memory to data objects, and the relation of such issues with higher level language features such as data types, scope and nesting rules, and recursion.

8. Understand the key issues in the generation of efficient code for a given architecture.

9. Understand the role played by code optimisation.

Compiler Construction

Course Modules

Preface

In this module, we shall first discuss the role of lexical analysis and the interface of a lexical analyser with the rest of the compiler. We identify tokens in a typical programming language and discuss the actions to be performed on discovery of a token. The issues involved in handcoding a lexical analyser are discussed next. Following this, we discuss automatic generation of lexical analysers. Finally, we shall understand the need for an efficient representation of a DFA and see how it can be achieved by a particular representation using four arrays.

Contents

1 Introduction

The compiler sees an input program as a string of characters *without any structure.* **For example, the program shown below**

```
var I,J: integer;
begin
     I := I + 1;
     J := J + 1;
end.
```

is seen by the compiler as

var b I , J : b integer ; CR begin CR b b b b b I b := b I b + b 1 ; b b b b b J b := b J b + b 1 CR end .

Here b **and** CR **represent the blank and newline characters. The first task of the compiler is to identify the structure of the program. This is done in two steps:**

1. **The string is broken into 'words'–the smallest logical units. This is** *lexical analysis* **or** *scanning*. **During this process all white spaces (blanks, tabs and newline characters) are eliminated. The result of applying this step on the earlier program is shown below.**

var	I	,	J	:	integer	;	begin	I	:=	I	+	1	;	J	:=	J	+	1	end	.

2. **Discover the larger structures in the program. This is** *syntax analysis* **or** *parsing*, **and is illustrated in figure 1.**

Note that during parsing we do not consider any entity smaller than a word.

Definition: ***Lexical analysis*** **is the operation of reading the input program, and breaking it up into a sequence of** *lexemes* (*tokens*).

We shall distinguish between some terms with similar meanings that are used in connection with lexical analysis.

- *lexemes* – **smallest logical units** (words) **of a program, such as** `A`, `B`, `1.0`, `true`, `+`, `<=`.
- *tokens* – **classes of similar lexemes, such as** *identifier, constant, operator*

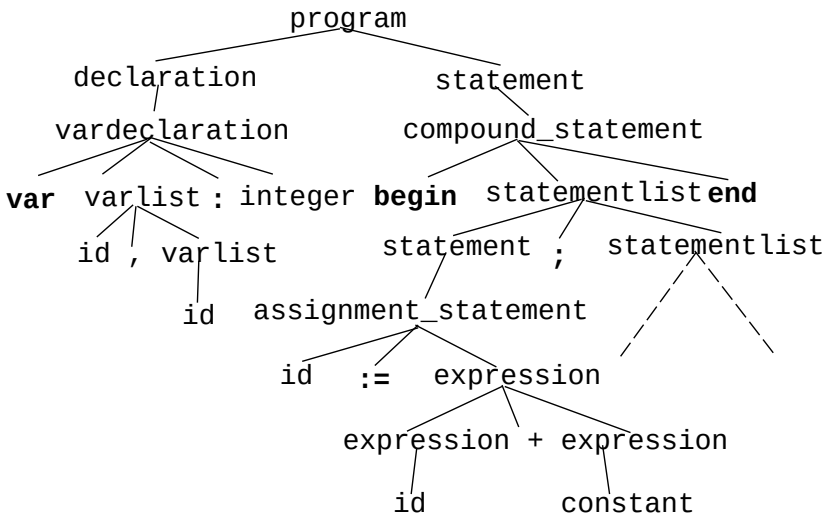


Figure 1: Syntactic structure of a program

- *pattern* – **an informal or formal description of a token, such as** *an identifier is a string of at most 8 characters, in which the first character is an alphabet, and the successive characters are either digits or alphabets.*

A pattern serves two purposes. It is a precise description or specification of tokens. And, as we shall see later, this description can be used to automatically generate a lexical analyser.

1.1 Context of a Lexical Analyser

How does a lexical analyser fit into the rest of the compiler? The lexical analyser is used primarily by the parser. When the parser needs the next token, it invokes the lexical analyser. Instead of analysing the entire input string, the lexical analyser sees enough of the input string to return a single token. Thus lexical analysis and parsing are not different passes of a compiler. They typically belong to the same pass, as shown in figure 2.

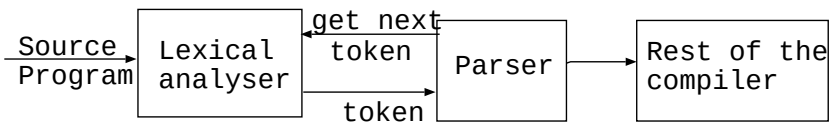


Figure 2: Context of a lexical analyser

This is the usual organization for compilers with a single pass front-end.

1.2 Types of Tokens

What are the different kinds of tokens in a typical language? To answer this question, consider the following function definition in Pascal.

```
function gcd (m, n:  integer):  integer;
begin
    if n = 0 then gcd := m
    else gcd := gcd(n,m mod n)
end; (* of gcd*)
```

Listed below are the tokens in the function definition and their corresponding lexemes.

tokens	lexemes
keyword	`function, begin, if, then, else, end`
identifier	`gcd, m, n, mod, integer`
constant	`0`
delimiter	`(, :, ), ;`
relop	`=`
assignop	`:=`
comment	`(*of gcd*)`

Some points should be noted here. The first is that `integer` is an indeed an identifier. It is a language defined type identifier. Also it is sometimes convenient to treat each keyword, each operator and each delimiter as a separate token. We shall, henceforth, assume such a scheme.

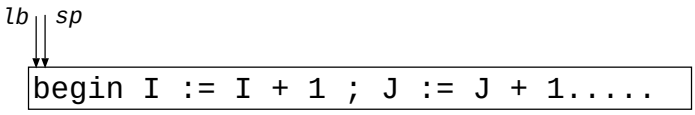
Apart from the token itself, the lexical analyser also passes other informations regarding the token. These items of information are called *token attributes.* **The table below lists some tokens and their attributes.**

lexeme	<token, token attribute>
`3`	< const, 3>
`A`	<identifier, `A`>
`if`	<if, –>
`:=`	<assignop, –>
`>`	<gt, –>
`;`	<semicolon, –>

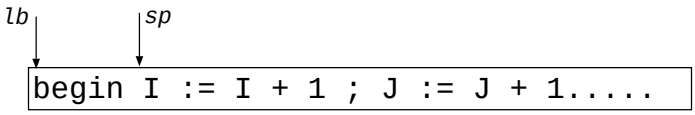
A question which can be raised is whether the attribute of a constant, say `3`, should be the lexeme `3` itself or the value 3. If the value of the constant is possibly used during compilation, then it would be more convenient to convert it into a value. For instance, if constants are used to specify the range of arrays, then the value of the constant is used to allocate memory for the array. For such a language the attribute of the constant should be its value.

1.3 Scanning the Input

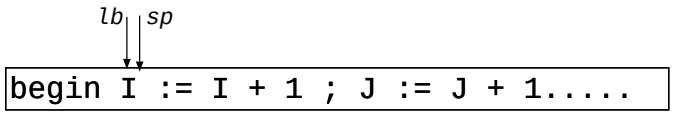
The lexical analyser uses two pointers (*lb:* lexeme_begining and *sp:* search_pointer) to keep track of the portion of the input string scanned.



Initially both pointers point to the beginning of a lexeme. *sp* then starts scanning forward to search for the end of the lexeme. The end of the lexeme, in this case, is indicated by the blank space after `begin`.

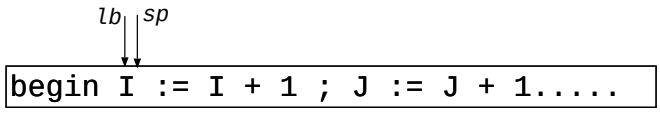


When the end of the lexeme is identified, the token and the attribute corresponding to this lexeme is returned. *lb* and *sp* are then made to point to the beginning of the next token.

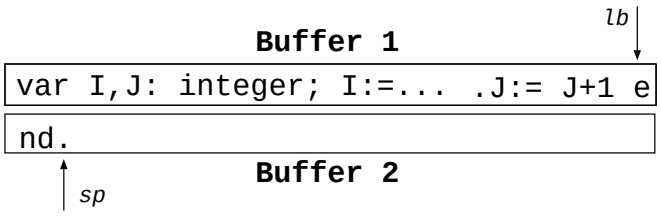


Reading the input character by character from the secondary storage is costly. A block of data is first read into a buffer, and then scanned by the lexical analyser. Buffering methods commonly used are:

1. ***One buffer scheme.*** However, there are problems if a lexeme crosses the buffer boundary. To scan the rest of the lexeme, the buffer has to be refilled, thereby overwriting the first part of the lexeme.



2. ***Two buffer scheme.*** **Here buffers 1 and 2 (shown in figure below) are scanned alternately. When one reaches the end of the current buffer, the other buffer is filled. Clearly the problem in the previous scheme is solved if no lexeme is longer than the length of a buffer.**



2 Construction of Lexical Analysers

There are two main approaches to the construction of lexical analysers:

1. *Hand code the lexical analyser:* **This involves actually writing a program to do lexical analysis.**

2. *Use a lexical analyser generator:* **This automatically generates the lexical analysis program from a formal description of the tokens of the language and their attributes.**

The lexical analyser generated by the first method is possibly more efficient, but the generation process is faster and is less prone to errors in the second case.

2.1 Handcoding a Lexical Analyser

To see the issues in coding a lexical analyser by hand, consider a very small language with the following tokens:

begin – **representing the lexeme** `begin`

end – **representing the lexeme** `end`

integer – **Examples:** `0`, `-5`, `250`

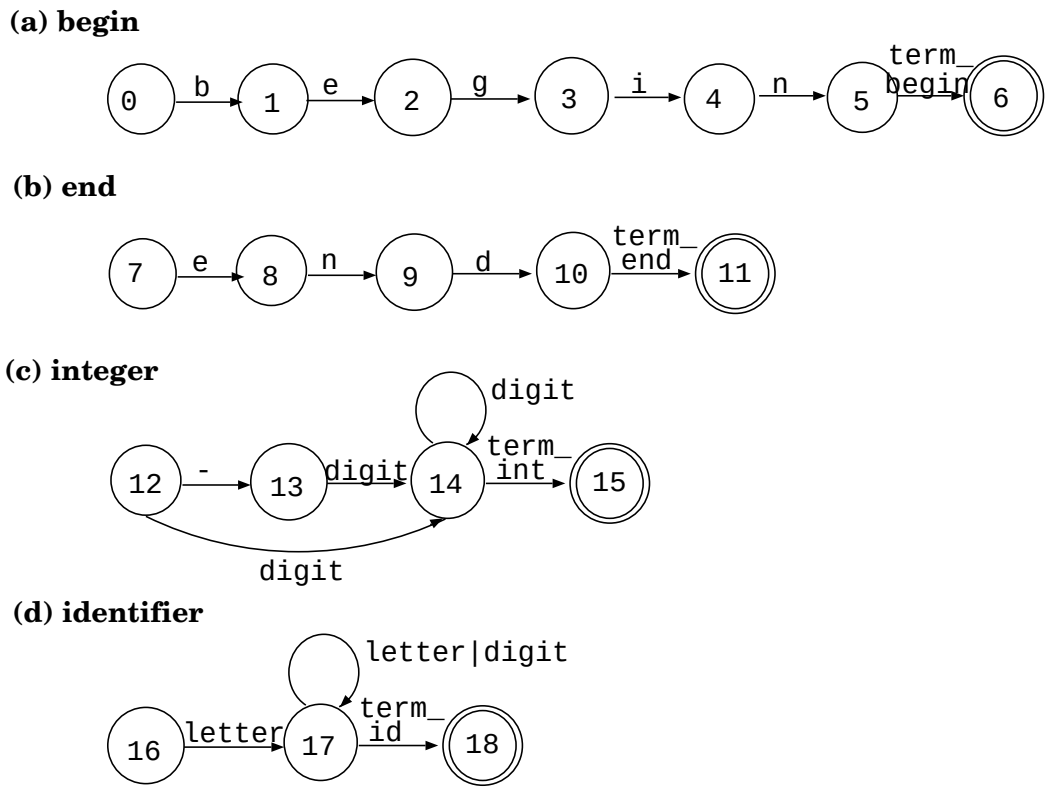


Figure 3: Transition diagrams

identifier – **Examples:** `a`, `A1`, `max`

Corresponding to each token, we draw a *transition diagram.* A transition diagram is a directed graph with nodes representing states, and edges representing transitions on input symbols. A state is a representation of the portion of input seen so far. For each transition diagram, there is a *start state* signifying anticipation of the corresponding token, and a *final state* signifying the end of the token.
Transition diagrams are useful in two ways. They serve as precise specification of tokens. They are also an aid in structuring the lexical analyser program. The transition diagrams for the four tokens are shown in figure 3. We shall explain the transitions on term_begin, term_end etc. later.

2.1.1 Converting Transition Diagrams into Code

The lexical analyser is represented by the function *nexttoken* **in figure 4. This function returns the next token and its attributes. It tries the transition diagrams successively starting with the diagram beginning in state 0. On failing**

```
function fail;
  begin
    sp := lb;
    case start of
      0: start := 7; return := 7;
      7: start := 12; return := 12;
      12: start := 16; return := 16;
      other: error
    end
  end;

function nexttoken;
  begin
    state := 0; start := 0;
    while true do
       case state of
         0: getnonblank; lb := sp;
            if c = 'b' then state := 1 else state = fail;
         ...
         5: c:= nextchar;
            if letter(c) or digit(c) then state := fail else state :=6;
         6: sp := sp -1; return(<begin, ->)
         ...
         16: c := nextchar;
            if letter(c) then state := 17 else state := fail;
         17: c := nextchar;
            if letter(c) or digit(c) then state := 17 else state := 18;
         18: sp := sp - 1; return (<identifier, input[lb,sp]>)
       end /*case*/
  end; /*nexttoken*/
```

Figure 4: The functions *nexttoken* and $fail$.

to traverse the current transition diagram, it backtracks to the beginning of the lexeme and tries the next diagram specified by the function $fail$. Notice that it is easy to understand the code for *nexttoken* and $fail$ because they are straightforward translations of the transition diagrams.

The following functions have been used in the code fragment.

1. *error*: **an unspecified error routine.**
2. *getnonblank*: **returns the next non-blank character.**
3. *nextchar*: **returns the next character.**
4. *letter*(c)/*digit*(c): **checks whether** c **is a letter/digit.**

Note the following points:

1. **Some tokens like begin and end have fixed lengths, whereas others, like those corresponding to numbers, have variable lengths.**

2. **Associated with each token, is a set of characters called its terminators. A terminator indicates the end of a token. For example:**

term_num	any character other than a digit.
term_id	any character other than a letter or a digit.
term_begin	any character other than a letter or a digit.
term_>	any character other than =

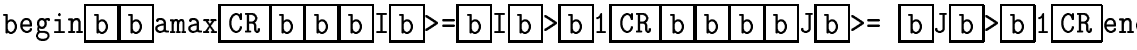
3. **A lookahead is necessary for a token, only if it is a prefix of a larger token. For example lookaheads are necessary for the keywords because each keyword is a prefix of an identifier.**

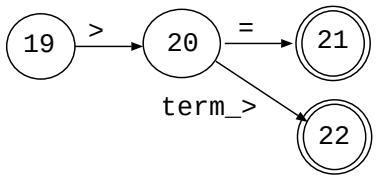
4. **The order in which transitions are tried is important. For example, between identifiers and keywords, the transition diagrams for keywords must be tried first, failing which the transition diagram for the identifier must be tried.**

5. **Keywords cause a proliferation in the number of states (and hence increases the code size). To prevent this we can adopt the following method. Right at the beginning of lexical analysis, we can enter all keywords in a keyword table. During lexical analysis we detect a keyword as an identifier. We then look up the keyword table for the identifier. If it is found, we declare it as a keyword, Otherwise it is declared as an identifier.**

Try the following as exercises:

1. **Add the token relop consisting of the lexemes > and >= as shown in the figure below, and try to scan the example input string:**





2. **Suppose you wanted to extend the function *nexttoken* by adding a token for reals. How would you do it?**

2.1.2 Combining Transition Diagrams

Transition diagrams can be combined to yield two benefits. The number of states of the combined transition diagram can be made smaller than the number of states of the original transition diagrams. As a consequence, the code size of the lexical analyser, which depends on the number of states, reduces. Also, by combining transition diagrams we can eliminate backtracking, the execution time of the lexical analyser goes down.

Notice that in the transition diagram shown in figure 5, there is a transition from each state marked # to state 14. A deterministic transition diagram with a single starting state as shown above is also called a DFA.

The size of the combined transition diagram can be brought down even further by eliminating the transition on terminators. This can be done if we adopt the following traversal strategy. Starting at the initial state state 0, we traverse the transition diagram on the input string as long as possible. If the last state on which there is no transition on the next input character is a final state, we return the token corresponding to the final state. Otherwise, we announce an error. The diagram obtained after eliminating the transitions on terminators is shown in the figure 6. Note once again that there is a transition from each state marked # to state 11.

It is not hard to see that this strategy is based on the assumption that the end of a token can be detected by examining *a single extra character (a lookahead of one).* Are there examples of tokens which require more than one lookaheads?

2.2 Automatic Generation of Lexical Analysers

To generate a lexical analyser, a generator will need two inputs. Firstly it will need a precise specification of the tokens of the language. Moreover, since this specification will be read in by a program, it should be *linguistic*. Second,

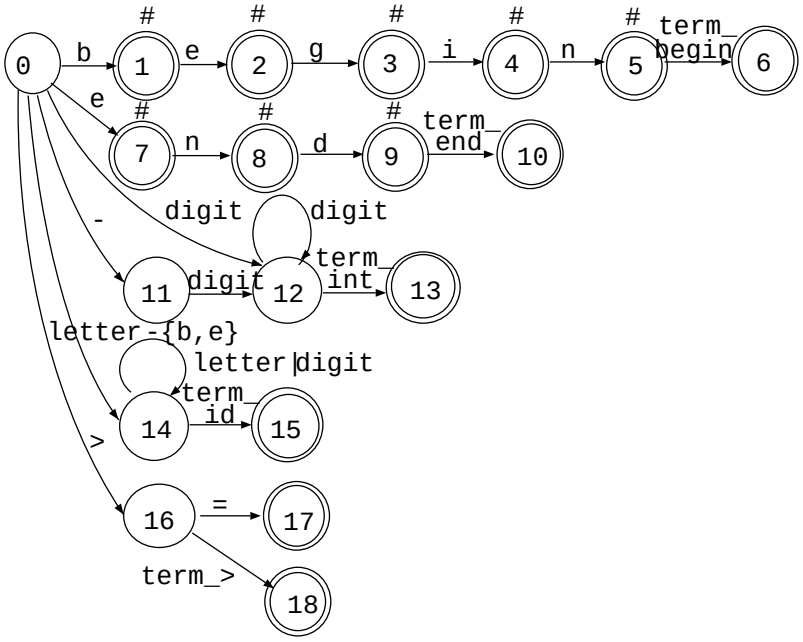


Figure 5: Combining transition diagrams

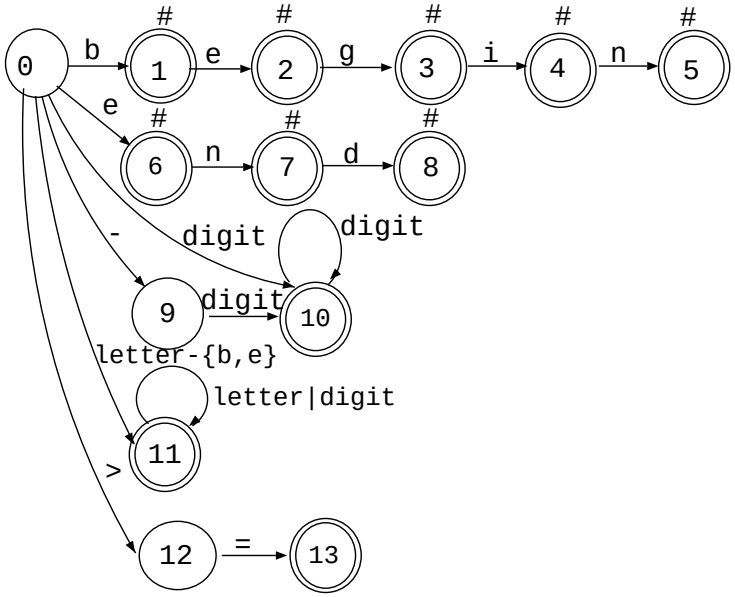


Figure 6: Eliminating transitions on terminators.

it will need a specification of the action to be performed, on identifying each token. The action will have to be specified through a program fragment in some programming language. From these two inputs, the generator will generate a lexical analyser, which can be incorporated into a compiler as

shown in figure 7. The UNIX tool lex is such a generator.

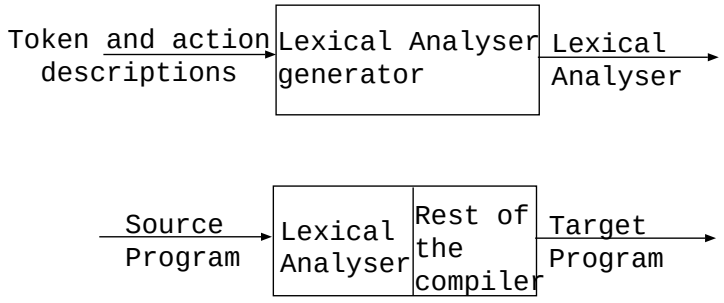


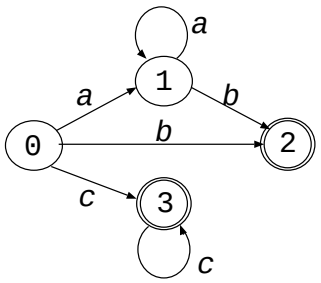
Figure 7: Generating a lexical analyser

How is the lexical analyser generated? Given the token and action specifications, the generator puts together (i) A transition diagram constructed from the token specification, (ii) A code fragment which can traverse *any transition diagram* (a driver routine), and (iii) the code specifying actions to be taken on recognition of tokens. As we shall shortly see, the resulting entity will be a lexical analyser for the language.

Consider an example language with two tokens. Token 1 is the set of strings consisting of zero or more a's and ending with a single b. And Token 2 is the set of strings consisting of one or more c's. We shall formally describe the two tokens as:

a^*b **{print('token_1 found')}**
c^+ **{print('token_2 found')}**

This formalism that we have used to describe tokens is called *regular expressions.* As we shall see, it describes the set of strings constituting a token. From the description, we can construct a structure called a deterministic finite automaton (DFA). Indeed, this conversion from a regular expression to a DFA is the key step in automatically generating a lexical analyser.



Given an input string, there is a notion of *acceptance* of the string by a DFA. This is similar to what we have been discussing for transition diagrams, i.e. starting from the initial state, the DFA is traversed using the input string as long as possible. If the last state to be traversed is a final state, the string is accepted by the DFA.

Now consider the following three together:

1. **The same DFA stored in a tabular form in an array called** $nextstate$ **as shown in figure 8. Given a state *s* and a symbol *a*,** $nextstate[s, a]$ **will give the state that** s **transits to on input** a**. This table will also contain information regarding which states are final. In the figure we have shown this by encircling the final states.**

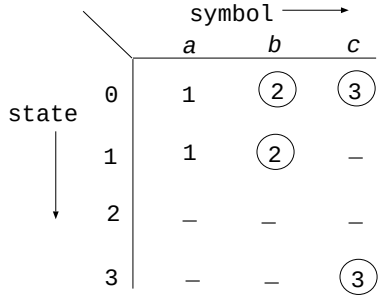


Figure 8: DFA in tabular form.

2. **A driver routine which can traverse** *any* **given DFA. Such a driver routine is shown below.**

```
function nexttoken;
begin
    state := 0; c := nextchar;
    while nextstate[state,c] <> _ do
            begin
                state := nextstate[state,c]; c := nextchar
            end
    if not final(state) then
            begin error; return end
    else begin
              unput(c); action; return
         end
    end;
```

3. The action part of the token specification, captured in a function *action*. See that the action specified are inserted as such in the function.

```
function action;
begin
    case state of
        2: print('token_1 found');
        3: print('token_2 found');
    end
end;
```

See that the DFA, the driver routine, and the action routines, taken together, constitute the lexical analyser for the small two token language.

Suppose now we want to generate a lexical analyser for a different language with a different set of tokens and actions. The set of tokens, represented as regular expressions and the actions represented as program fragments are once again fed to the generator. The generator creates a new DFA, creates a new action function and adds these to the same driver routine. This becomes a lexical analyser for the new language.

Since the driver routine is common to all generated lexical analysers, and creation of the action routine is simple in the sense that it consists of taking the actions specified and inserting them in the function *action*, the only non-trivial part of a generator is the conversion of a regular expression to a DFA. We shall discuss this next.

2.2.1 Specifying Tokens

To specify tokens, we use regular expressions. Roughly, a regular expression denotes a set of strings, as does a token. A set of strings is also called a *language*. We talk about a token in the context of lexical analysis, whereas the concept of regular expression is used in much more general contexts.

We shall now first formally define the notion of a language. Then we shall define the syntax of regular expressions. Following this we specify the language that each regular expression denotes. Finally, we shall see how to specify tokens through regular expressions.

2.2.2 Languages

An *alphabet* is any finite set of symbols. For example, $L = \{A, \ldots, Z, a, \ldots, z\}$, $D = \{0, 1, \ldots, 9\}$, $L \bigcup D$ are languages. A *string* over an alphabet is a finite (possibly zero) sequence of symbols drawn from the alphabet. *begin*, *max*1, 123, ϵ (the empty string), are strings over $L \bigcup D$. We can construct bigger strings from smaller ones by *concatenation*. If s and r are strings, then the *concatenation* of s *and* r, written sr is the string formed by appending r to s. Further $s\epsilon = \epsilon s = s$. A string, exponentiated i times, is defined inductively as $\mathrm{s}^0 = \epsilon$, $\mathrm{s}^i = \mathrm{ss}^{i-1}$. Finally, a *language* is any set of strings over some fixed alphabet. Some examples of languages are {max, max1, A123}, {1, 123, -87}, {begin}, {end}, {ϵ}, ϕ (the empty set).

2.2.3 Operations on Languages

Languages can be combined to give bigger languages using the operations shown below. If L and M are languages, then:

1. **Union**
$$L \bigcup M = \{s | s \in L \text{ or } s \in M\}$$
2. **Intersection**
$$L \bigcap M = \{s | s \in L \text{ and } s \in M\}$$
3. **Concatenation**
$$LM = \{sr | s \in L \text{ and } r \in M\}$$
4. **Exponentiation**
$$L^i = LL^{i-1}$$
5. **Kleene Closure (zero or more concatenations)**
$$L^* = \bigcup_{i=0}^{\infty} L^i$$
6. **Positive Closure (one or more concatenations)**
$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

How does all this help us in defining the token identifier, which, as we know is a language consisting of strings of alphanumeric characters, in which the first characters is a letter. Firstly we have $L = \{A, \ldots, Z, a, \ldots, z\}$, the language of strings consisting of a single letter, and $D = \{0, 1, \ldots, 9\}$, the language of strings consisting of a single digit. Therefore $L \bigcup D$ is the language

of single letters and digits. This gives $(L \bigcup D)^*$ as the language of all strings of letters and digits, including the null string ϵ. We know that this is not enough because ϵ can never be an identifiee and that an identifier must start with a letter. An identifier, therefore is the language $L(L \bigcup D)^*$. In some languages, the maximum length if identifiers is restricted. Verify that an identifier of length at most three can be expressed as $L \bigcup L(L \bigcup D)^1 \bigcup L(L \bigcup D)^2$.

2.2.4 Regular Expressions

After having defined the concept of languages formally, let us now define the syntax of regular expressions and the languages they denote. Regular expressions over an alphabet Σ are constructed using the following rules.

1. ϵ is a regular expression
2. if a is a symbol in Σ, then a is a regular expression.
3. if r and s are regular expressions, then $r|s$ is a regular expression.
4. if r and s are regular expressions, then rs is a regular expression.
5. if r is a regular expression, then r^* is a regular expression.
6. if r is a regular expression, then (r) is a regular expression.

Examples of regular expressions over the alphabet $\{a, b\}$ are ϵ, a, $a|b$, $(a|b)^**$. The reader is encouraged to identify the rules that were used in the construction of these regular expressions.

We now specify the language that a regular expression denotes. This is tabulated below.

regular expression	language
ϵ	$\{\epsilon\}$
$a \in \Sigma$	$\{a\}$
$r\|s$	$L(r) \bigcup L(s)$
rs	$L(r)L(s)$
r^*	$L(r)^*$
(r)	$L(r)$

Let us see what regular expression will denote the languge corresponding to the token identifier. The language L is captured by the regular expression $l = A|B|\ldots|Z|a|b|\ldots|z$. Similarly, D is represented by the regular expression

$d = 0|1|\ldots|9$. **Therefore** $l|d$ **denotes** $(L \bigcup D)$ **and** $l(l|d)*$ **denotes** $(L \bigcup D)^*$**, the language of identifiers.**

A language defined by a regular expression is called a *regular language.*

2.2.5 Extended Regular Expressions

The language provided by Lex to describe patterns is an extension of regular expressions that we have seen earlier. We call this language *extended regular expressions*. **The extension serves two purposes:**

1. **It is more concise than the usual regular expressions. For example, instead of writing** `A | B | C | D | E`**, we can write** `[A-E]`.

2. **It provides for machine readability. The symbols of the language whose tokens are being specified and the metalanguage in which the regular expressions are being expressed are clearly distinguished. Thus** `|` **is the metalinguistic character denoting 'alternate', Whereas** `\|` **is the character** `|` **itself.**

We describe *extended regular expressions* **with examples.**

Expression	Describes	Example
c	any non-metalinguistic character c	a
\c	character c literally	*
''s''	string s literally	''**''
.	any character but newline	a.*b
^	beginning of a line	^abc
$	end of line	abc$
[s]	any character in s	[abc]
[^s]	any character not in s	[^abc]
$r*$	zero or more r's	a*
$r+$	one or more r's	a+
$r?$	zero or one r	a?
r_1r_2	r_1 then r_2	ab
$r_1\|r_2$	r_1 or r_2	a|b
(r)	r	(a|b)
r_1/r_2	r_1 when followed by r_2	abc/123

2.2.6 Converting Regular Expression to DFA

Consider the regular expression $(a|b)^*bba|c^+$. We can represent this regular expression in the form of a tree as shown in figure 9.

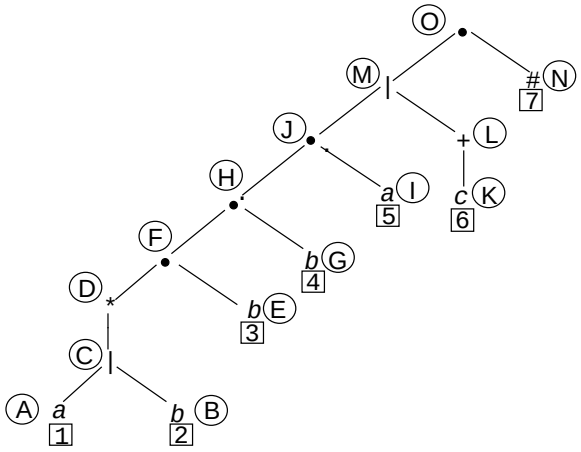


Figure 9: Tree representation of regular expression

The procedure for converting a regular expression to a tree is self-explanatory. However note that leaves have been labeled both with positions (1,2 etc.) and alphabetic labels (A,B etc.), whereas interior nodes are labeled with alphabetic labels only. The subtree rooted at a any node represents some subexpression of the regular expression. Also, concatenation has been represented by a $\bullet$. Finally, to signify the end of a token, we use # as a end marker.

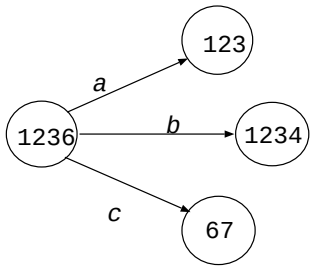
The key idea is to identify a state with a set of positions in the regular expression. For example, the starting state of the DFA for $(a|b)^*bba|c^+$ could expect a a (from position 1 of the corresponding tree), a b (from position 2 or 3) or a c (from position 6). Therefore, we shall identify this state with positions 1,2,3 and 6. These are the positions from where the first symbol a or b could be generated.

1236

Now observe that if the first symbol is a b coming from position 1, then, because of the $*$ at node D, the next symbol could be a a coming from position 1 or a b coming from position 2 or a b coming from 3. This is what the second row of the table below represents. Similarly, as shown in the third row, if the first symbol is a b coming from position 3, the next symbol can only be a b coming from position 4. figured out similarly.

if we are in	then on symbol	we could go to
position 1	a	1,2,3
position 2	b	1,2,3
position 3	b	4
position 6	c	6,7

Therefore, if we follow our idea of identifying states with a set of positions where the next symbol could come from, then, from the staring state 1236, we shall go to the state 1234 on the symbol b. The transitions on a, and c could be figured out similarly. This gives us three new states as shown below:



We can now reason similarly with each of the three new states. For every state we could ask where the transition will be for each of the three symbols a, b and c in the alphabet. In each case we shall either transit to a new state or one of the states cretaed earlier. Continuing in this manner, the final DFA will be as shown in figure 10.

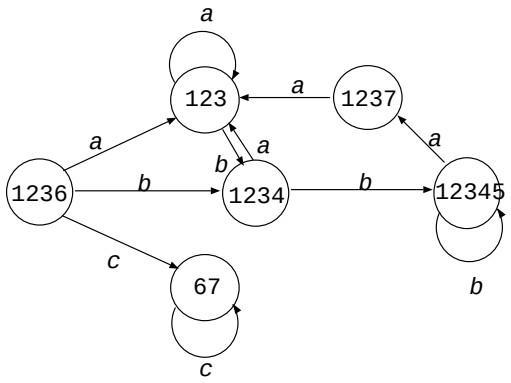


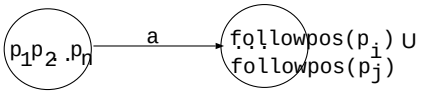
Figure 10: DFA for $(a|b)^*bba|c^+$

It is easy to see that the final states are those which contain position 7, i.e. the position expecting #.

We shall now attempt to formalise the method. Suppose we are able to define a function $followpos$ which takes a position as an argument and returns a set of positions as result. $followpos(i)$ answers the following question: Suppose the current symbol of a string generated from the regular expression is from position i of the syntax tree, then from which positions can the next symbol of the string come from? Given the function $followpos$, we can write the earlier table in terms of $followpos$ as,

if we are in	then on symbol	we could go to
position 1	a	followpos(1)
position 2	b	followpos(2)
position 3	b	followpos(3)
position 6	c	followpos(6)

Assume that the current state under consideration consists of positions $p_1, \ldots, p_n$, and of these $p_i, \ldots, p_j$ are the only positions associated with a certain symbol a. Then the next state on symbol a can be described in terms of $followpos$ as:



Clearly, if we manage to define the function $followpos$ for a regular expression, then the DFA for this regular expression can be constructed.

To find $followpos$, we have to first define some additional functions:

1. $nullable(n)$: is a function which takes a node of the tree as argument and returns a boolean value as result. $nullable(n)$ is true if the subtexpression represented by the node n can generate a null string. For example, the node represented by D is $(a|b)^*$, which can generate the null string. Similarly the node represented by F is $(a|b)^*b$, which cannot generate the empty string. Therefore we have:

   $nullable$(D) = true;
   $nullable$(F) = false;

2. $firstpos(n)$: $firstpos$ is a function from a node to a set of positions. $firstpos$ of a node n is the set of positions from which the first symbol of some string derivable from the subexpression represented by n could come.

$firstpos$(D) = {1, 2}
$firstpos$(F) = {1, 2, 3}

This is because the first symbol a string derivable from $(a|b)^*$ at node D could either be a a coming from 1, or a b coming from 2. Similarly, the first symbol of a string derivable from $(a|b)^*b$ at node F could either come from 1 (a), 2 (b), or from 3 (b).

3. $lastpos(n)$: $lastpos$ is a function from a node to a set of positions. $lastpos$ of a node n is the set of positions from which the last symbol of some string derivable from the subexpression represented by n could come.

$lastpos$(D) = {1,2}
$lastpos$(F) = {3}

We shall now state rules for constructing the functions $nullable$, $firstpos$ and $lastpos$. These rules are tabulated below.

node n	$nullable(n)$
n is a leaf labeled ϵ	true
n is a leaf labeled with symbol a	false
n is $c_1\|c_2$	$nullable(c_1)$ or $nullable(c_2)$
n is $c_1 \bullet c_2$	$nullable(c_1)$ and $nullable(c_2)$
n is c^*	true
n is c^+	nullable(c)

node n	$firstpos(n)$
n is a leaf labeled ϵ	ϕ
n is a leaf labeled a at position i	$\{i\}$
n is $c_1\|c_2$	$firstpos(c_1) \bigcup firstpos(c_2)$
n is $c_1 \bullet c_2$	if $nullable(c_1)$ then $firstpos(c_1) \bigcup firstpos(c_2)$ else $firstpos(c_1)$
n is c^*	$firstpos(c)$
n is c^+	$firstpos(c)$

node n	$lastpos(n)$
n **is a leaf labeled** ϵ	ϕ
n **is a leaf labeled** a **at position** i	$\{i\}$
n **is** $c_1 \mid c_2$	$lastpos(c_1) \bigcup lastpos(c_2)$
n **is** $c_1 \bullet c_2$	**if** $nullable(c_2)$ **then** $lastpos(c_1) \bigcup lastpos(c_2)$ **else** $lastpos(c_2)$
n **is** c^*	$lastpos(c)$
n **is** c^+	$lastpos(c)$

Given $nullable$,$firstpos$ **and** $lastpos$, $followpos$ **can be found out by repeated application of the three rules given below.**

1. $c_1 \bullet c_2$**: If** i **is a position in** $lastpos(c_1)$**, then everything in** $firstpos(c_2)$ **is in followpos(i).**

2. c^***: If** i **is a position in** $lastpos(c)$**, then every position in** $firstpos(c)$ **is in** $followpos(i)$**.**

3. c^+**: If** i **is a position in** $lastpos(c)$**, then every position in** $firstpos(c)$ **is in** $followpos(i)$**.**

For the example that we have been considering, the functions $nullable$,$firstpos$, $lastpos$ **and** $followpos$ **are shown in figure 11. Notice that** $followpos$ **is defined only for leaf nodes.**

Let us see how $followpos(\mathsf{A})$ **(or** $followpos(1)$**, since** A **corresponds to position 1 in the syntax tree) was calculated. We shall use nodes in the syntax tree and the regular expressions denoted by them interchangeably. Consider the regular expression** D•F**. Since 1 is in** $lastpos(\mathsf{D})$**, from rule 1 it follows that 3, which is in** $firstpos(\mathsf{F})$ **will be in** $followpos(1)$**. Similary, by considering the subexpression** C^***, we see that the positions 1 and 2, which are in** $firstpos(\mathsf{C})$ **will be in** $followpos(1)$**. So** $followpos(1) = \{1, 2, 3\}$**. No other rule generates any more elements in** $followpos(1)$**.**

Now given these functions, the algorithm for constructing the entire DFA is shown in 12. The algorithm is a straightforward translation of the construction method discussed earlier. It creates states in a depth first manner. The only comment that we shall make is that a state is put in stack only when it is created for the first time. Whereas a marked state is one which has not only been created but transitions from which have also been been computed.

node	$firstpos$	$lastpos$	$followpos$
A	{1}	{1}	{1,2,3}
B	{2}	{2}	{1,2,3}
C	{1,2}	{1,2}	
D	{1,2}	{1,2}	
E	{3}	{3}	{4}
F	{1,2,3}	{3}	
G	{4}	{4}	{5}
H	{1,2,3}	{4}	
I	{5}	{5}	{7}
J	{1,2,3}	{5}	
K	{6}	{6}	{6,7}
L	{6}	{6}	
M	{1,2,3,6}	{5,6}	
N	{7}	{7}	
O	{1,2,3,6}	{7}	

Figure 11: $firstpos$, $lastpos$ and $followpos$ for the example

3 Lexical Errors

There are primarily two kinds of lexical errors:

1. **Lexemes whose length exceed the bound specified by the language. As an example, in Fortran, an identifier whose length is more than 7 characters is a lexical error. Similarly, most languages have a bound on the precision of numeric constants. A constant whose length exceeds this bound is a lexical error.**

2. **Illegal characters appearing in the input. For example, the characters `~`, `&` and `@` occuring in a Pascal program (outside a string or a comment) are lexical errors.**

Lexical errors can easily be detected by lexical analyser. Apart from issuing an error message, the lexical analyser can possibly take other actions on detection of an error. For an error of the first type, the entire lexeme can be read and then truncated to the specified length. In case of an error of the second kind, a simple strategy would be to skip illegal characters till one detects a character which can begin a token. Another alternative would be to

1. Construct the tree for $r\#$.

2. Construct functions $nullable$, $firstpos$, $lastpos$ and $followpos$.

3. Let $firstpos(root)$ be the first state. Push it on top of stack;
   while stack not empty
   do begin
   pop the top state U off the stack; mark U;
   for each input symbol a do
   begin
   let $p_1, \ldots, p_k$ be the positions in U
   corresponding to the symbol a;
   let $V = followpos(p_1) \bigcup \ldots \bigcup followpos(p_k)$;
   put V in stack if not marked and not
   already in stack;
   make a transition from U to V labeled a
   end
   end

4. Final states are the states containing the position corresponding to $\#$.

Figure 12: Algorithm for constructing DFA

pass the character to the parser and depend on the error handling capability of the parser. (see module on Error Analysis). This might better in certain situations because the parser has better knowledge of the context in which error has occurred. This helps the parser to recover from the error by not only deletion (like the strategy mentioned above) but also by insertion.

4 Minimizing the Number of States

For the regular expression $(b|\epsilon)(a|b)^*abb$, the DFA constructed by the earlier method is shown in figure 13.

It can be verified that there is another DFA, with lesser number of states, which accepts the same language. This is shown in figure 14.

For a typical language, the number of states of the DFA is in order of hundreds. Therefore we should try to minimize the number of states, if possible.

The key idea is that certain states can possibly be merged in a DFA

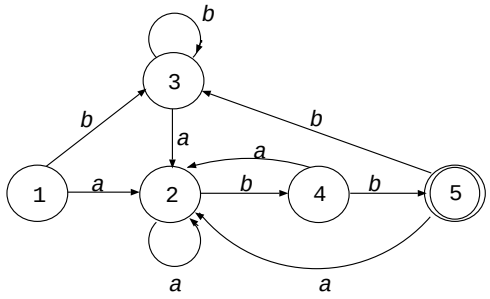


Figure 13: A DFA for $(b|\epsilon)(a|b)^*abb$

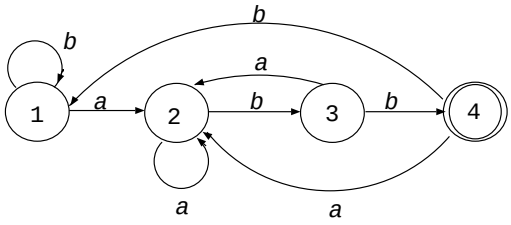


Figure 14: The minimized DFA for $(b|\epsilon)(a|b)^*abb$

without affecting acceptability. As example, note that *any string takes 1 to a final state, iff it takes 3 to a final state.* **The states 1 and 3 are therefore, as far acceptance is concerned,** *indistinguishable.* **States 2 and 4, on the other hand, are** *distinguishable*. **The string** b **takes 4 to the final state 5, but takes 2 to 4.** b **is called a *distinguishing string* for states 2 and 4. So 1 and 3 can be merged, but 2 and 4 have to be kept separate.**

Two states are *distinguishable* **if there exists a string** s **which takes one of the states to a final state, and the other to a non-final state.** s **is then called a** *distinguishing string* **for the two states.**

How does one find distinguishable states? Firstly, note that the set of final states are distinguishable from the set of non-final states. This is true because the null-string ϵ **distinguishes them.**

Assume initially that the set of states have been partitioned into two groups, one consisting of the set of final states, and the other the set of non-final states. The states within a group are deemed to be indistinguishable. Therefore,

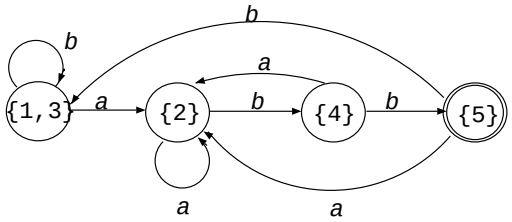
Π_1 **= {{1,2,3,4},{5}}**

Now consider states 3 and 4. The string b **takes 3 to itself and 4 to 5. Since 3 and 5 are already distinguishable on the basis of** ϵ**, states 3 and 4 become distinguishable on the basis of the string** $b\epsilon$**. The new partition** Π_2

and the next partition Π_3 are shown in the table below.

Partition n	Distinguishing string
Π_1 = {{1,2,3,4},{5}}	ϵ
Π_2 = {{1,2,3},{4},{5}}	$b\epsilon$
Π_3 = {{1,3},{2},{4},{5}}	$bb\epsilon$

In the minimized DFA, there will be a state corresponding to each group of the partition. If a group has more than one state of the original DFA, then, on a particular symbol, they will all transit to states of the same group. The resulting DFA is



Note that in the context of lexical analyser generators, all the final states should not merged in the initial partition. This is because two different final states may recognise two different tokens, and therefore should be distinguished. So, in the initial partition, only those final states which recognise the same token are merged in a group. The algorithm for minimizing a DFA is shown in figure 15.

5 Efficient Representation of DFA

A naive method to represent a DFA is to use a two dimensional array indexed by state and input symbol, whose entries are, in turn, states. For a typical language, the number of DFA states is in the order of hundreds (1111 in the case of a particular Fortran-77 compiler), and the number of input symbols is roughly 100. The size of the resulting DFA array is extremely large. Therefore it is necessary to find a representation which is smaller in size without compromising on the time required to retrieve the nextstate information.

5.1 The Four Arrays Scheme

A key observation leading to a efficient representation is the fact that for a DFA like the one shown in 16, the states marked with # behave like state 11

1. Construct an initial partition $\Pi = \{S - F, F\}$, where F is thed set of final states and $S - F$ is the state of non-final states.

2. for each group G of Π do
   partition G into subgroups such that two states
   s and t of G are in the same subgroup if and only if
   for all input symbols a, states s and t have transitions
   onto states in the same group of Π;
   replace G in Π_{new} by the set of all subgroups formed

3. If $\Pi_{new} = \Pi$, let $\Pi_{final} := \Pi$ and continue with step 2. Otherwise repeat step 4. with $\Pi := \Pi_{new}$.

4. Construct one state in each group of the partition Π_{final} as the representative for that group. Let s be a representative state. If, in the old DFA, there is a transition from s to t on a, then in the new DFA, there is a transition from s to the representative of the group containing t in Π_{final}.

5. Remove any dead states.

Figure 15: Algorithm for minimizing states of DFA

on all symbols *except for one symbol*. **Therefore information about state 11 can also be used for these states. So 11 is a defualt state for the states marked with #.**

This scheme uses four one dimensional arrays called $DEFAULT$, $BASE$, $NEXT$ and $CHECK$. If s is a state and a is the numeric representation of a symbol, then

1. **$BASE[s]$ gives the base location for the information stored about state s in the array $NEXT$. This is required because the transitions of all the states are being stored in the same array $NEXT$.**

2. **$NEXT[BASE[s]+a]$ gives the next state for s and symbol a. However, this information is valid only if $CHECK[BASE[s]+a] = s$.**

3. **If $CHECK[BASE[s]+a] \neq s$, it means that the transition for state s on symbol a has not been stored explicitly. So we have to look up the information associated with the dafult of s to find this transition. $DEFAULT[s]$ gives the default state for the state s.**

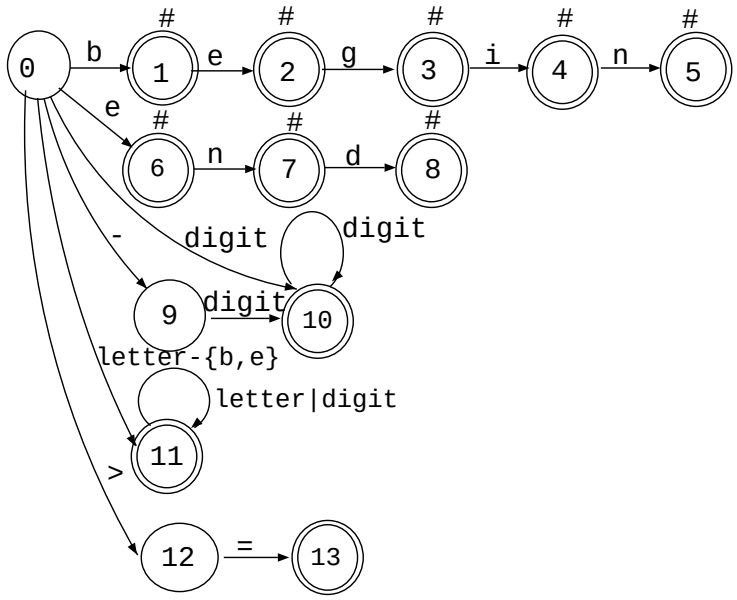


Figure 16: Example DFA

The function which gives the next state can be described as:

function *nextstate*(s,a);
begin
  if $CHECK[BASE[s] + a] = s$ **then** $NEXT[BASE[s]+a]$
  else return(*nextstate*($DEFAULT[s]$,a))
end

As an example, consider the alphabet for the example DFA with its symbols numbered as shown below:

a–z	**0–25**
0–9	**26–35**
>	**36**
=	**37**
-	**38**

Figure 17 shows how information about states 1, 7, and 11 of the earlier DFA would be stored under the four arrays scheme. Note the following points:

1. **All the entries for state 11 have been stored in the array** $NEXT$**. The** $CHECK$ **array shows that the entries are valid for state 11. However, for each of states 7 and 11, only one transition has been stored.**

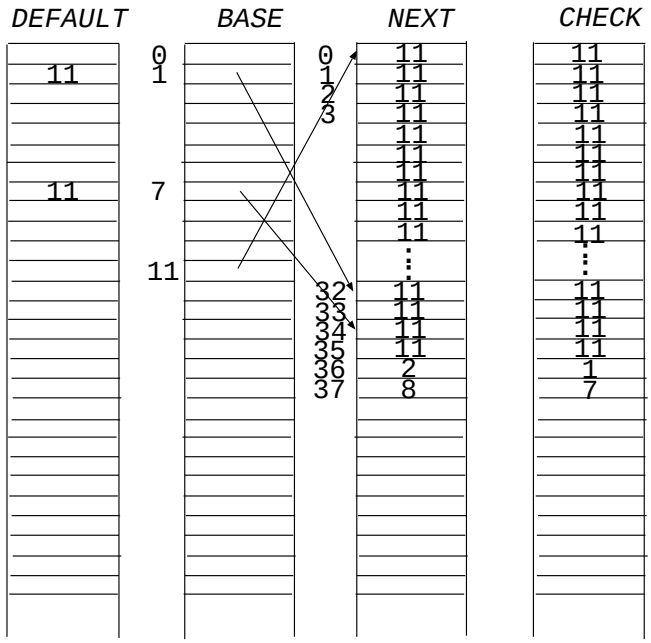


Figure 17: DFA stored under four arrays scheme

2. **State 1 has a transition on e(numbered 4), which is different from the corresponding transition on state 11. This differing entry is stored in $NEXT$[36]. Therefore $BASE$[1] is set to 36 − 4 = 32. By a similar reasoning $BASE$[7] is set to 34.**

3. **To find nextstate[1,0], we first refer to $NEXT$[32+0], But since $CHECK$[32+0] is not 1 we have to refer to $DEFAULT$[1] which is 11. So the correct next state is found from $NEXT[BASE[11]+0] = 11$.**

4. **How does one fill up the arrays so that not much space is wasted? As seen earlier, for a given state, the transitions on some symbols are stored explicitly, the transition on others are found through the default state. One heuristic, which works well in practice, is to find for a given state, the lowest $BASE$, so that the explicitly stored transitions can be entered without conflicting with entries already existing in the table.**

6 Graded Exercises

1. **Identify the lexemes that make up the tokens in the following programs. Give reasonable attribute values for the tokens.**

   a. **Pascal**

```
function max ( i, j : integer ) : integer;
{ return maximum of integers i and j }
begin
    if i > j then max := i
    else max := j
end;
```

   b. **Fortran 77**

```
      FUNCTION MAX ( I, J )
C     RETURN MAXIMUM OF INTEGERS I AND J
          IF (I .GT. J) THEN
              MAX = I
          ELSE
              MAX = J
          END IF
      RETURN
```

2. **Write a regular expression for the language consisting of all strings of 0's and 1's, in which the number of 1's is a multiple of 3.**

3. **Draw a DFA to accept binary representations of all numbers which are multiples of 3. Repeat the exercise for the sets $\{k \mid k \ mod\ 3 \ = \ 1 \}$ and $\{k \mid k \ mod\ 3 \ = \ 2\}$.**

4. **Describe the sets denoted by the following regular expressions:**

   a. **(11|0)*(00|1)***

   b. **(1|01|001)*(ϵ |0|00)**

   c. **(00|11|(01|10)(00|11)*(01|10))***

5. **Define a comment as any sequence of characters enclosed between (* and *), not containing the sequence *). Comments may extend over more than one lines.**

Write an extended regular expression to specify a comment.

6. **Repeat the above exercise for a string, defined as a sequence of characters enclosed between ' and '. If the character ' itself is a part of the string, it should be represented as ''.**

7. **Using the direct method, construct minimum DFAs for the following regular expressions:**

   a. **10|(0|11)0*1**

   b. **01(((10)*|111)*|0)*1**

   c. **((0|1)(0|1))*|((0|1)(0|1)(0|1))***

8. **Consider the following LEX-like specification:**

   AUXILLIARY DEFINITIONS

```
letter = A |.. | Z
digit  = 0 |.. | 9
sign   = + | -
```

   TRANSLATION RULES

```
END                 {return 1}
FOR                 {return 2}
GOSUB               {return 3}
GOTO                {return 4}
IF                  {return 5}
LET                 {return 6}
REM                 {return 7}
RETURN              {return 8}
letter(digit)?      {return 9}
sign digit digit*   {return 10}
```

   a. **Construct the minimum DFA which would recognize the tokens specified by the program.**

   b. **Construct a four array representation for this DFA.**

7 References

1. Aho, A.V., R. Sethi and J.D. Ullman (1986) : ***Compilers – Principles, Techniques and Tools***, Addison-Wesley, Reading.

2. Barrett, W.A. and J.D. Couch (1977): ***Compiler Construction***, Science Research Associates, Pennsylvania.

3. Brown, P.J. (1979): ***Writing Interactive Compilers and Interpreters***, Wiley, New York.

4. Calingaert, P. (1979): ***Assemblers, Compilers and Program Translation***, Computer Science Press, Maryland.

5. Dhamdhere, D.M. (1983): ***Compiler Construction – Principles and Practice***, Macmillan India, New Delhi.

6. Fischer, C.N. and R.J. LeBlanc (1988) : ***Crafting a compiler***, Benjamin/Cummings, Menlo Park, California.

7. Gries, D. (1971): ***Compiler Construction for Digital Computers***, Wiley, New York.

8. Holub, A.I. (1993): ***Compiler Design in C***, Prentice-Hall, Englewood-Cliffs.

9. Levine, J.R., T. Mason and D. Brown (1990): ***Lex and Yacc***, 2^{nd} edition, O'Reilly & Associates, Sebastopol.

10. Schreiner, A.T., H.G. Friedman Jr. (1985): ***Introduction to Compiler Construction with UNIX*** Prentice-Hall, Englewood Cliffs.

11. Tremblay, J.P. and P.G. Sorenson (1985): ***The Theory and Practice of Compiler Writing***, McGraw-Hill, New York.

12. Watson, D. (1989): ***High Level Languages and their Compilers***, Addison-Wesley, Reading.

13. Weingarten, F.W. (1973): ***Translation of Computer Languages***, Holden-Day, San Francisco.