CS 304 Compiler Design

Text books

1. Compilers – Principles, Techniques & Tools , Aho, Ravi Sethi, D. Ullman

- **Introduction to compilers and lexical analysis:- Analysis of the source program - Analysis and synthesis phases, Phases of a compiler.**
- **Compiler writing tools. Bootstrapping.**
- **Lexical Analysis - Role of Lexical Analyser, Input Buffering, Specification of Tokens, Recognition of Tokens**

Regular expressions

- Used to define precisely a language
- Eg : Pascal identifiers are letters followed by zero or more letters or digits

  letter (letter|digit)*

- A regular expression can be built up out of simpler regular expressions using a set of rules
- Each regular expression r denotes a language L(r)
- The defining rules specify how L(r) is formed by combining in various ways the languages denoted by the sub-expressions of r

- Here are the rules that define regular expressions over alphabet Σ
- Associated with each rule is a specification of the language denoted by the regular expression being defined

1. ε is a regular expression that denotes $\{\varepsilon\}$; the set containing the empty string
2. If a is symbol in Σ, then a is a regular expression that denotes $L(a)= \{a\}$, the set containing the string a

3. If r and s are regular expressions denoting the languages L(r) and L(s), then
   a. (r)|(s) is a regular expression denoting L(r) U L(s)
   b. (r)(s) is a regular expression denoting L(r) L(s)
   c. r* is a regular expression denoting (L(r))*
   d. (r) is a regular expression denoting $L(r)^2$

- A language denoted by a regular expression is said to be a regular set
- The specification of a regular expression is an example of a regular definition
- Rules 1 and 2 form the basis of the definition
- We use the term basic symbol to refer to ε or a symbol Σ appearing in a regular expression
- Rule 3 provide the inductive step

- Unnecessary parentheses can be avoided in regular expressions if we adopt the convention that

1. The unary operator* has the highest precedence and is left associative
2. Concatenation has the second highest precedence and is left associative
3. | has he lowest precedence and is left associative

- Eg : Let $\Sigma = \{a,b\}$

1. The regular expression a|b denotes the set {a,b}
2. The regular expression (a|b)(a|b) denotes {aa,ab,ba,bb}, the set of all strings of a's and b's of length two. Another regular expression for the same set is aa|ab|ba|bb
3. The regular expression a* denotes the set of all strings of zero or more a's ie {ε,a,aa,aaa,....}

4. The regular expression (a|b)* denotes the set of all strings containing zero or more instances of an a or b, ie, the set of all strings of a’s and b’s. another regular expression for this set is (a*b*)*
5. The regular expression a|a*b denotes the set containing the string a and all strings consisting of zero or more a’s followed by a b

AXIOM	DESCRIPTION
$r \mid s = s \mid r$	$\mid$ is commutative
$r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ is associative
$(rs)t = r(st)$	concatenation is associative
$r(s \mid t) = rs \mid rt$ $(s \mid t)r = sr \mid tr$	concatenation distributes over $\mid$
$\epsilon r = r$ $r\epsilon = r$	ϵ is the identity element for concatenation
$r^* = (r \mid \epsilon)^*$	relation between $*$ and ϵ
$r^{**} = r^*$	$*$ is idempotent

Fig. 3.9. Algebraic properties of regular expressions.

Regular definitions

- For notational convenience, names are given to regular expressions
- We can define regular expressions using these names as if they were symbols
- If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

………

$$d_n \rightarrow r_n$$

Where each d_i is a distinct name and each r_i is a regular expression over the symbols in $\Sigma \cup \{d_1, d_2, \ldots, d_{n-1}\}$, ie the basic symbols and the previously defined names letter -> A|B|..............|Z|a|b|.........|c

digit -> 0|1|.........|9

id -> letter (letter | digit)*

digits -> digit digit*

num -> digits

Regular definition for unsigned numbers in Pascal

- 5280, 39.37, 6.336E4, 1.894E-4

digit → 0|1|....|9

digits → digit digit*

optional _fraction →.digits |ε

optional_exponent → (E(+|-| ε)digits)| ε

num → digits optional_fraction optional_exponent

Notational shorthands

- Certain constructs occur frequently in regular expressions that it is convenient to introduce notational short hands for them

1. One or more instances
   - The unary postfix operator
   - If r is a regular expression that denotes the language L(r), then r+ is a regular expression that denotes the language (L(r))+
   - The regular expression a+ denotes the set of all strings of one or more a's
   - The operator + has the same precedence and associativity as the operator *
   - The algebraic identities $r^* = r^+|\varepsilon$ and $r^+ = rr^*$ relate Kleene and positive closures

2. Zero or one instance

- The unary ? Operator
- r? = r|ε
- If r is a regular expression, then (r)? is a regular expression that denotes the language L(r) U ε
- Using + and ? operators,we can rewrite the regular definition of num as follows
  - o digit → 0|1|....|9
  - o digits → digit digit*
  - o digits → digit+

- optional _fraction →.digits |ε
- optional _fraction →(.digits)?

- optional_exponent → (E(+|-| ε)digits)| ε
- optional_exponent → (E(+|-| ε)digits)?

- num → digits optional_fraction optional_exponent digit → 0|1|....|9

digits → digit+

optional _fraction →(.digits)?

optional_exponent → (E(+|-| ε)digits)?

num → digits optional_fraction optional_exponent

Character classes

- [abc] denotes a|b|c
- An abbreviated character class [a-z] denotes the regular expression a|b|....|z
- Using character classes, we can describe identifiers as being strings generated by regular expression

  [A-Za-z][A-Za-z0-9]*

Recognition of tokens

- Previous section considered how to specify tokens
- Here we address the question of how to recognize them
- We use the language generated by the following grammar as the example

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** stmt
	|	ε
expr	→	*term* **relop** *term*
	|	*term*
term	→	**id**
	|	**num**

Where the terminals **if, then, else, relop, id** and **num** generate sets of strings given by the following regular definitions

if	$\rightarrow$	if
then	$\rightarrow$	then
else	$\rightarrow$	else
relop	$\rightarrow$	$< \mid <= \mid = \mid <> \mid > \mid >=$
id	$\rightarrow$	$\textbf{letter(letter}\mid\textbf{digit)}^*$
num	$\rightarrow$	$\textbf{digit}^{+}\ (.\textbf{digit}^{+})?(E(+\mid-)?\textbf{digit}^{+})?$

Where letter and digit are as defined earlier

- For this language, the lexical analyzer will recognize the keywords if, then, else, as well as the lexemes denoted by relop, id and num
- In addition, lexemes are separated by white spaces, consisting of nonnull sequences of blanks, tabs and newlines
- Our lexical analyzer will strip out white space
- It do so by comparing a string against the regular definition **ws**

$$\textbf{delim} \quad \rightarrow \quad \textbf{blank} \mid \textbf{tab} \mid \textbf{newline}$$

$$\textbf{ws} \quad \rightarrow \quad \textbf{delim}^{+}$$

- If a match for ws is found, the lexical analyzer does not return a token to the parse, but it proceeds to find a token following the white space and returns that to the parser
- Our goal is to construct a lexical analyzer that will isolate the lexeme for the next token in the input buffer and produce as output a pair consisting of the appropriate token and attribute-value, using the translation table given
- The attribute-values for the relational operators are given by the symbolic constants LT, LE, EQ, NE, GT, GE

Regular Expression	Token	Attribute-Value
ws	-	-
if	if	-
then	then	-
else	else	-
id	id	Pointer to table entry
num	num	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Transition diagram

- Intermediate step in the construction of a lexical analyzer
- A stylized flowchart
- Depicts the actions that take place when a lexical analyzer is called by the parser to get the next token
- Suppose the input buffer is as in fig 3.3 and the lexeme beginning pointer points to the character following the last lexeme found
- We use a transition diagram to keep track of the information about characters that are seen as the forward pointer scans the input
- We do so by moving from position to position in the transition diagram as characters are read

- Positions in a transition diagram are drawn as circles – states
- States are connected by arrows – edges
- Edges leaving state s have labels indicating the input characters that can next appear after the transition diagram has reached state s
- The label “other” refers to any character that is not indicated by any of the other edges leaving s
- Transition diagrams are deterministic – ie, the labels of two edges leaving a state cannot match
-

- One state is labeled the start state – the initial state of the transition diagram where control resides when we begin to recognize a token
- Certain states may have actions that are executed when the flow control reaches that state
- On entering a state we read the next input character
- If there is an edge starting from the current state whose label matches this input character, we then go to the state pointed to by the edge, otherwise we indicate a failure

A transition diagram for the token relop

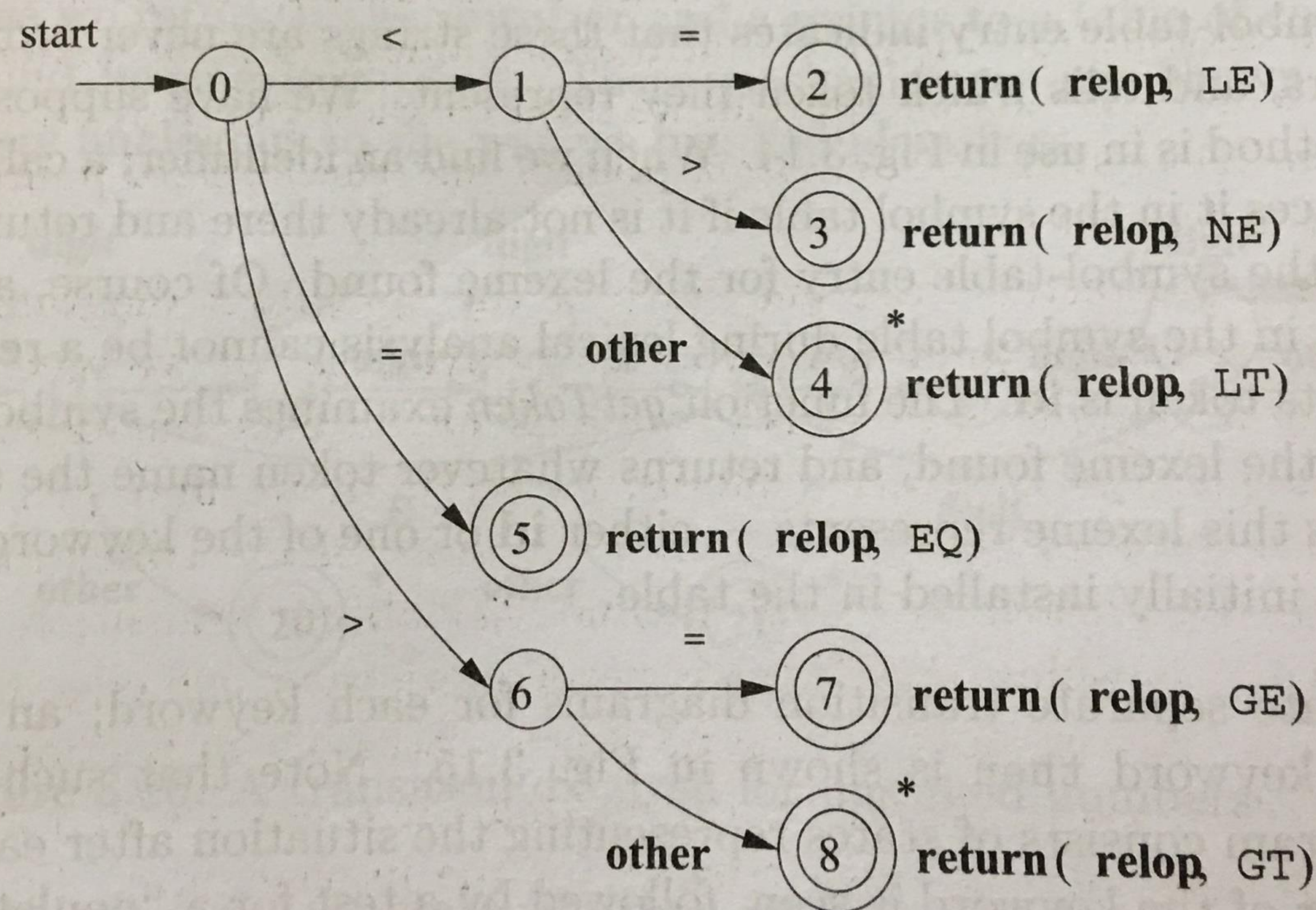


Figure 3.13: Transition diagram for **relop**

Transition diagram for identifiers and keywords

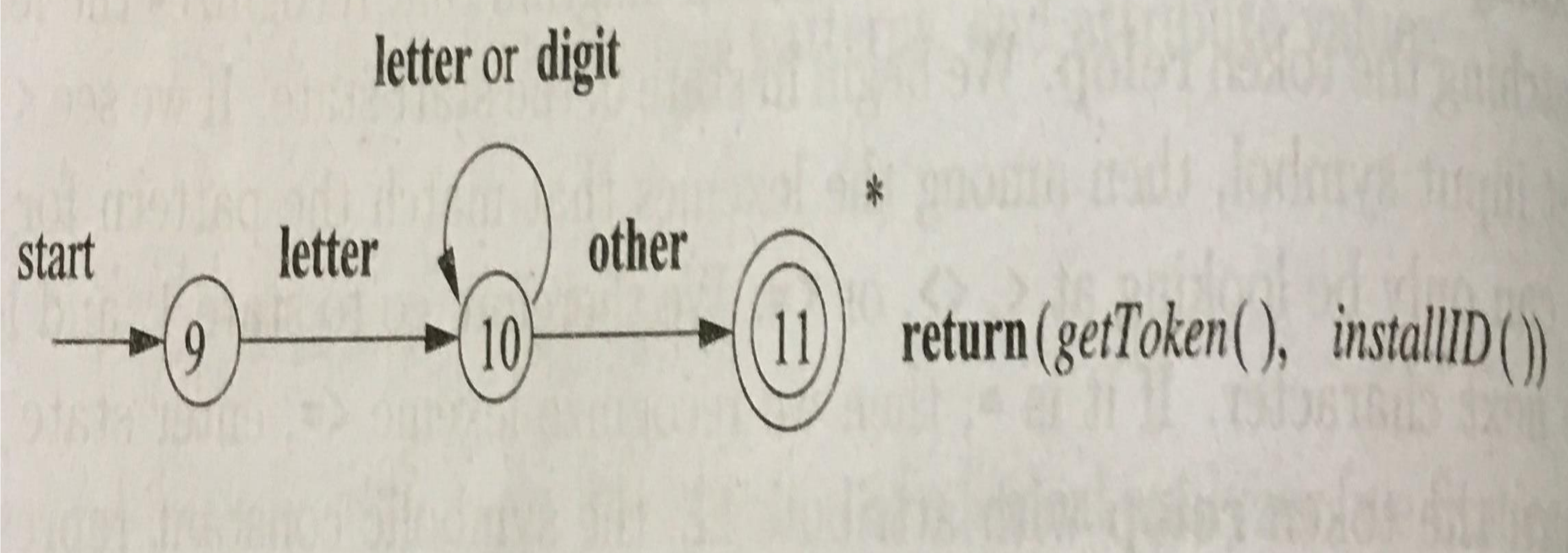


Figure 3.14: A transition diagram for **id**'s and keywords

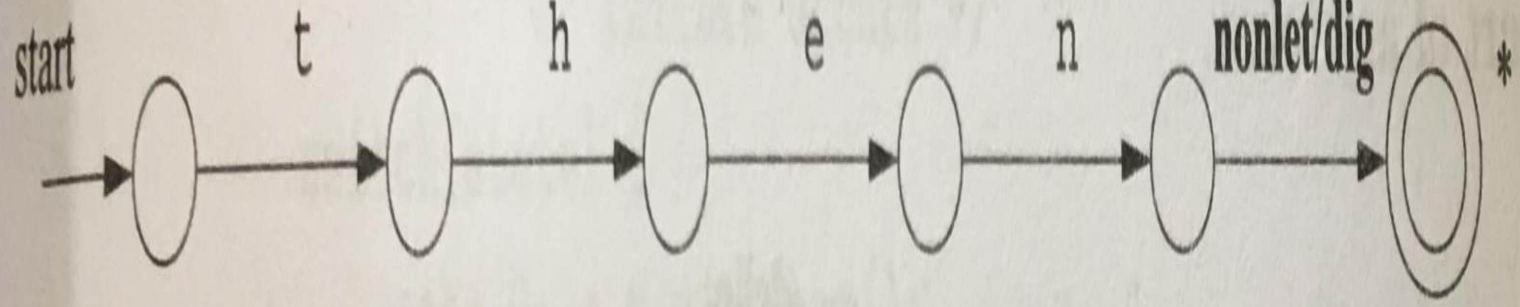


Figure 3.15: Hypothetical transition diagram for the keyword then

- gettoken()
  - Looks for the lexeme in the symbol table
  - If the lexeme is a keyword, the corresponding token is returned; otherwise the token **id** is returned
- Install_id()
  - Has access to the buffer, where the identifier lexeme has been located
  - The symbol table is examined and if the lexeme is found there marked as a keyword, install_id() returns 0
  - If the lexeme is found and is a program variable, install_id() returns a pointer to the symbol table entry
  - If the lexeme is not found in the symbol table, it is installed as a variable and a pointer to the newly created entry is returned

Transition diagram for unsigned numbers in Pascal

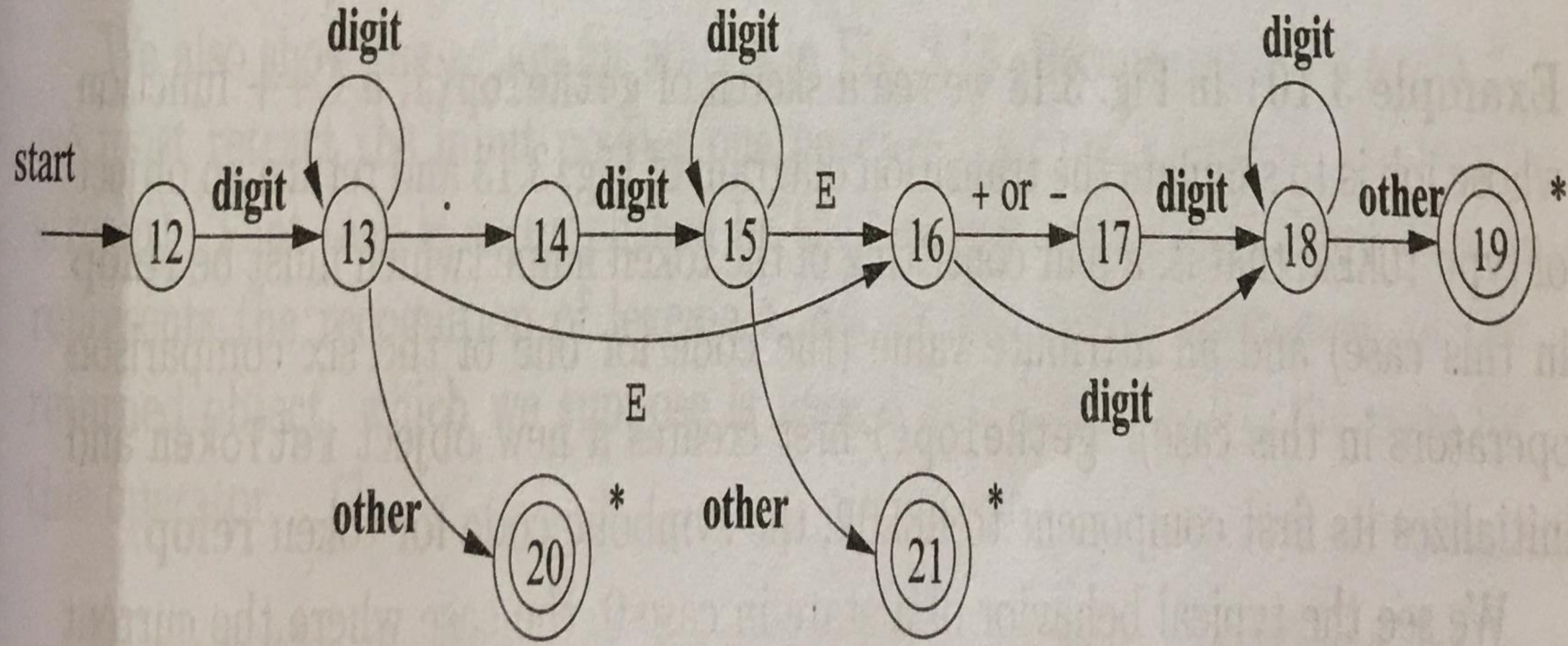


Figure 3.16: A transition diagram for unsigned numbers

- There are several ways in which the redundant matching in the transition diagram of fig 3.14 can be avoided

1. Rewrite the transition diagrams by combining them into one
2. Change the response to failure during the process of following a diagram

- A sequence of transition diagrams for all tokens of eg is obtained if we put together all the transition diagrams
- Lower-numbered states are to be attempted before higher numbered states

- The only remaining issue concerns white space
- Here nothing is returned to the parser when white space is found in the input
- Merely go back to the start state of the first transition diagram to look for another pattern

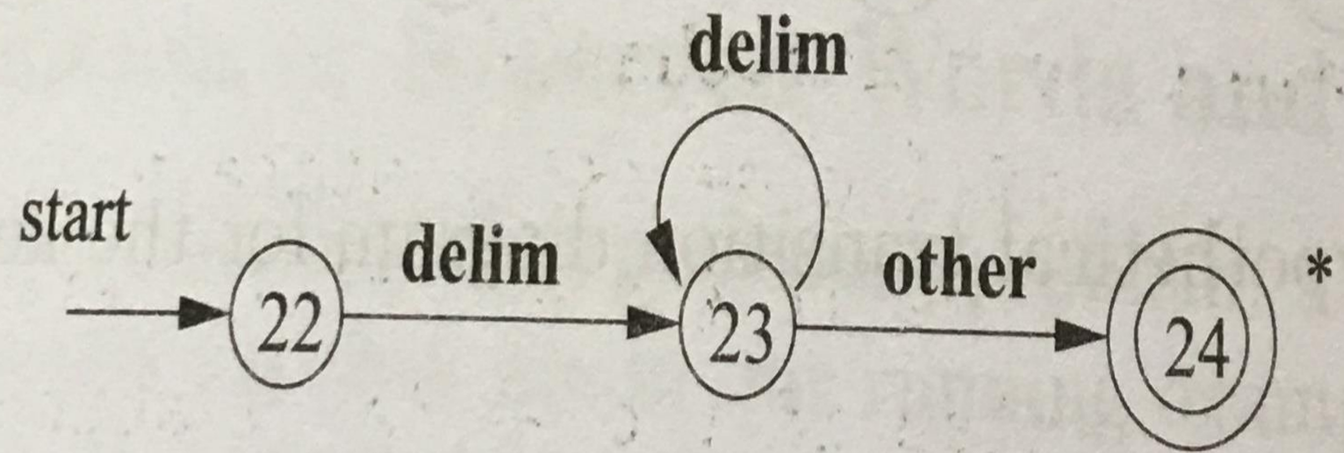


Figure 3.17: A transition diagram for whitespace

Implementing a transition diagram

- A sequence of transition diagrams can be converted into a program to look for the tokens specified by the diagrams
- Each state gets a segment of code
- If there are edges leaving a state, then its code reads a character and selects an edge to follow, if possible
- A function nextchar() is used to read the next character from the input buffer, advance the forward pointer at each call and return the character read
- If there is an edge labeled by the character read, or labeled by a character class containing the character read, then control is transferred to the code for the state pointed to by that edge

- If there is no such edge, and the current state is not one that indicates a token has been found, then a routine fail() is invoked to retract the forward pointer to the position of the beginning pointer and to initiate a search for a token specified by the next transition diagram
- If there are no other transition diagrams to try, fail() calls an error recovery routine
- To return tokens we use a global variable lexical_value which is assigned the pointers returned by the functions install_id() and install_num() when an identifier or number respectively is found
- The token class is returned by the main procedure of the lexical analyzer, called nexttoken()

- We use a case statement to find the start state of the next transition diagram
- Two variables state and start keep track of the present state and the starting state of the current transition diagram
- Edges in the transition diagrams are traced by repeatedly selecting the code fragment for a state and executing the code fragment to determine the next state

```
token nexttoken()
{    while(1) {
          switch (state) {
          case 0:    c = nextchar();
               /* c is lookahead character */
               if (c==blank || c==tab || c==newline) {
                    state = 0;
                    lexeme_beginning++;
                         /* advance beginning of lexeme */
               }
               else if (c == '<') state = 1;
               else if (c == '=') state = 5;
               else if (c == '>') state = 6;
               else state = fail();
               break;

               .../* cases 1-8 here */

          case 9:    c = nextchar();
               if (isletter(c)) state = 10;
               else state = fail();
               break;
```

```
        case 10:   c = nextchar();
            if (isletter(c)) state = 10;
            else if (isdigit(c)) state = 10;
            else state = 11;
            break;
        case 11:   retract(1); install_id();
            return ( gettoken() );

            .../* cases 12-24 here */

        case 25:   c = nextchar();
            if  (isdigit(c)) state = 26;
            else state = fail();
            break;
        case 26:   c = nextchar();
            if (isdigit(c)) state = 26;
            else state = 27;
            break;
        case 27:   retract(1); install_num();
            return ( NUM );
        }
    }
}
```

Fig. 3.16. C code for lexical analyzer.

```c
int state = 0, start = 0;
int lexical_value;
    /* to "return" second component of token */

int fail()
{
    forward = token_beginning;
    switch (start) {
        case 0:   start = 9; break;
        case 9:   start = 12; break;
        case 12:  start = 20; break;
        case 20:  start = 25; break;
        case 25:  recover(); break;
        default:   /* compiler error */
    }
    return start;
}
```

Fig. 3.15. C code to find next start state.

Finite Automata

- A recognizer for a language is a program that takes as input a string x and answers "yes" if x is a sentence of the language and "no"otherwise
- We compile a regular expression into a recognizer by constructing a generalized transition diagram called a finite automaton
- A finite automaton can be deterministic or non-deterministic
- Here we discuss the methods for converting regular expressions into both kinds of finite automata

Nondeterministic Finite Automata

- A non deterministic finite automaton (NFA) is a 5-tuple that consists of

1. A set of states S
2. A set of input symbols Σ (the input alphabet)
3. A transition function δ that maps state-symbols pairs
4. A state so distinguished as the start state
5. A set of states F distinguished as accepting (or final) states

Deterministic Finite Automata

- A DFA is a special case of an NFA in which

1. No state has an ε-transition
2. For each state s and input symbol a, there is atmost one edge labeled a leaving s

- The following algorithm show how to simulate the behavior of a DFA on an input string

Simulating a DFA

Input : An input string x terminated by an end-of-file character eof. A DFA D with start state so and set of accepting states F

Output : The answer “yes” if D accepts x; “no” otherwise

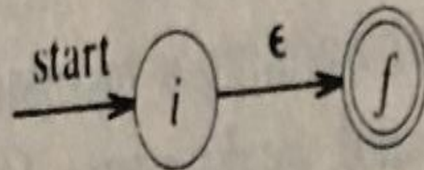
Method : Apply the algorithm following to the input string x. The function move(s,c) gives the state to which there is a transition from state s on input character c. The function nextchar returns the next character of the input string x

```
s := so;
c := nextchar;
while c ≠ eof do
        s := move(s,c);
        c := nextchar();
end;
if s is in F then
        return "yes";
else return "no";
```

From regular expressions to an NFA

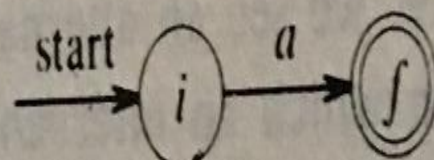
- Algorithm : (Thompson's construction), An NFA from a regular expression
- Input : A regular expression r over an alphabet Σ
- Output : An NFA N accepting L(r)
- Method :

1. For ϵ, construct the NFA



Here i is a new start state and f a new accepting state. Clearly, this NFA recognizes $\{\epsilon\}$.

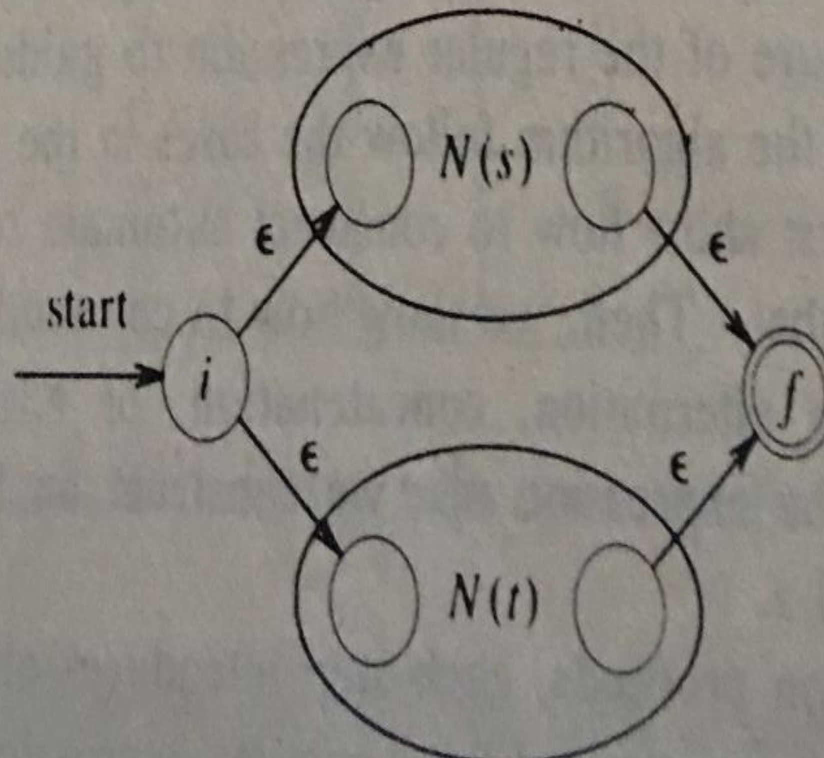
2. For a in Σ, construct the NFA



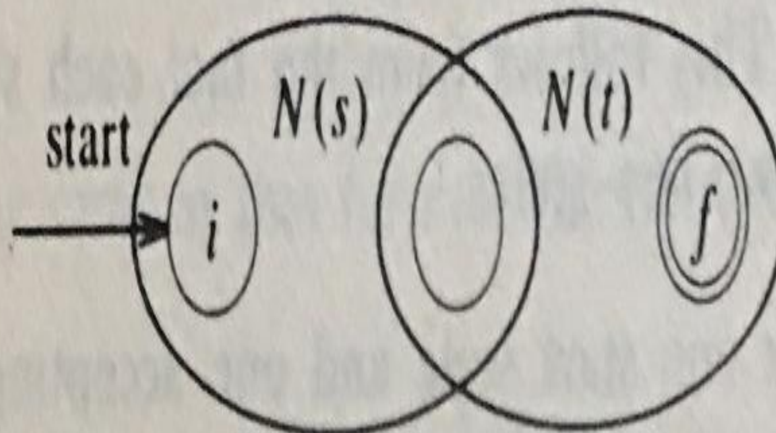
Again i is a new start state and f a new accepting state. This machine recognizes $\{a\}$

3. Suppose $N(s)$ and $N(t)$ are NFA's for regular expressions s and t.

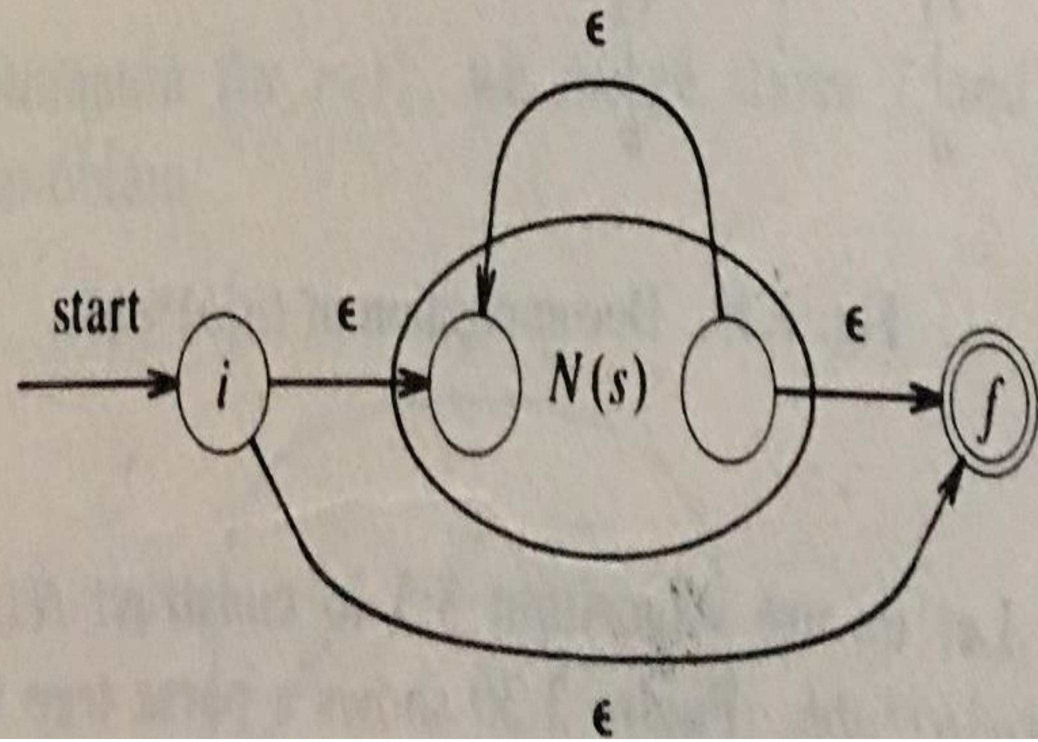
a) For the regular expression $s|t$, construct the following composite NFA $N(s|t)$:



b) For the regular expression *st*, construct the composite NFA $N(st)$:



c) For the regular expression s^*, construct the composite NFA $N(s^*)$:



- D. For the parenthesized regular expression (s), use N(s) itself as the NFA

- Eg : (a| b)*abb