Regular Definitions

These will look like the productions of a context free grammar we saw previously, but there are differences. Let Σ be an alphabet, then a *regular definition* is a sequence of definitions

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
...
$d_n \rightarrow r_n$

where the d's are unique and not in Σ and
r_i is a regular expressions over $\Sigma \cup \{d_1,\ldots,d_{i-1}\}$.

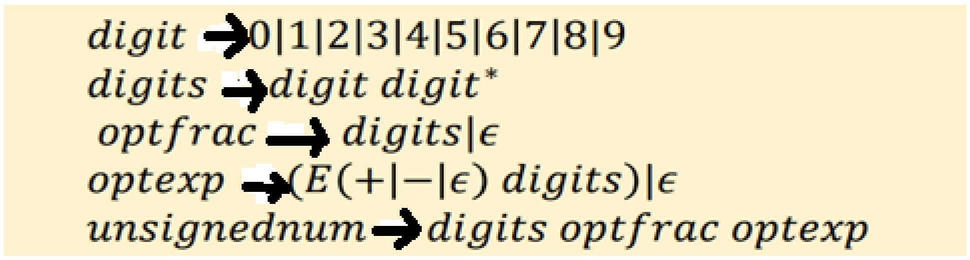
Note that each d_i can depend on all the previous d's.

Example: C identifiers can be described by the following regular definition

letter_ → A | B | ... | Z | a | b | ... | z | _
digit → 0 | 1 | ... | 9
CId → letter_ (letter_ | digit)*

Example : Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4.

The regular definition



$$\begin{aligned}
&\textit{digit} \rightarrow 0|1|2|3|4|5|6|7|8|9 \\
&\textit{digits} \rightarrow \textit{digit digit}^* \\
&\textit{optfrac} \rightarrow \textit{digits}|\epsilon \\
&\textit{optexp} \rightarrow (E(+|-|\epsilon)\ \textit{digits})|\epsilon \\
&\textit{unsignednum} \rightarrow \textit{digits optfrac optexp}
\end{aligned}$$

Extensions of Regular Expressions:

1. One or more instances. The unary, postfix operator + represents the positive closure of a regular expression and its language. Tha t is, if r is a regular expression, then (r) + denotes the language (L(r)) + . The operator + has the same precedence and associativity as the operator *.
   Two useful algebraic laws, r* = r + e and r = rr* = r*r relate the Kleene closure and positive closure.
2. Zero or one instance. The unary postfix operator ? means "zero or one occurrence." That is, r? is equivalent to r|e, or put another way, L(r?) = L(r) U {e}. The ? operator has the same precedence and associativity as * and +.

3. Character classes. A regular expression qi,, where the a^s are each symbols of the alphabet, can be replaced by the shorthand [a_1,a_2,….. a_n]. More importantly, when a_1,a_2,….. a_n in a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by a_1-a_n , that is, just the first and last separated by a hyphen. Thus, [abc] is shorthand for a|b|c, and [a-z] is shorthand for a|b|.--|z

Write a We can Rewrite Regular definition for unsigned number using above extension

$$
\begin{aligned}
&digits \Rightarrow [0-9]\\
&digits \Rightarrow digit^{+}\\
&unsignednum \Rightarrow digits\ (.\ digits)?\ (E[+-]?\ digits)?
\end{aligned}
$$

Extensions of Regular Expressions:

1. One or more instances. The unary, postfix operator + represents the positive closure of a regular expression and its language. Tha t is, if r is a regular expression, then (r) + denotes the language (L(r)) + . The operator + has the same precedence and associativity as the operator *. Two useful algebraic laws, r* = r + e and r = rr* = r*r relate the Kleene closure and positive closure.

2. Zero or one instance. The unary postfix operator ? means "zero or one occurrence."

Tha t is, r? is equivalent to r|e, or put another way, L(r?) = L(r) U {e}.

The ? operator has the same precedence and associativity as * and +. 3.

Character classes. A regular expression qi,, where the a^s are each symbols of the alphabet, can be replaced by the shorthand [aia,2 • • - an]. More importantly, when 01,02,.. . ,a n f° r m a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by oi-a n , tha t is, just the first and last separated by a hyphen. Thus, [abc] is shorthand for a|b|c, and [a-z] is shorthand for a|b|.--|z

$$
\begin{aligned}
&digits = [0-9]\\
&digits = digit^{+}\\
&unsignednum = digits\ (.\ digits)?\ (E[+-]?\ digits)?
\end{aligned}
$$

3.4 Recognition of Tokens

we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.

• Patterns are denoted with regular expressions, and recognized with finite state automata

• Regular definitions, a mechanism based on regular expressions, are popular for specification of tokens

• Transition diagrams, a variant of finite state automata, are used to implement regular definitions and to recognize tokens • Usually used to model LA before translating them to executable programs.

Assume the following grammar for the following discussion

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

where the terminals if, then, else, relop, id and num generates sets of strings given by following regular definitions

- For this language, the lexical analyzer will recognize the keywords if, then, and else, as well as lexemes that match the patterns for relop, id, and number.
- We know that keywords are also reserved words: that is they cannot be used as identifiers.
- The num represents the unsigned integer and real numbers of Pascal.
- In addition, we assume lexemes are separated by white space, consisting of nonnull sequences of blanks, tabs, and newlines.
- Our lexical analyzer will strip out white space. It will do so by comparing a string against the regular definition ws, below.

Delim → blank | tab | newline
ws → delim

- If a match for ws is found, the lexical analyzer does not return a token to the parser.
- It is the following token that gets returned to the parser.

Tha following table shows, for each lexeme or family of lexemes, which token name is returned to the parser and what attribute value.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	-	-
if	**if**	–
then	**then**	-
else	**else**	-
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	

Transition Diagrams

It is a directed labeled graph consisting of nodes and edges. Nodes represent states, while edges represent state transitions.

Components of Transition Diagram

One state is labelled the Start State. It is the initial state of transition diagram where control resides when we begin to recognize a token.

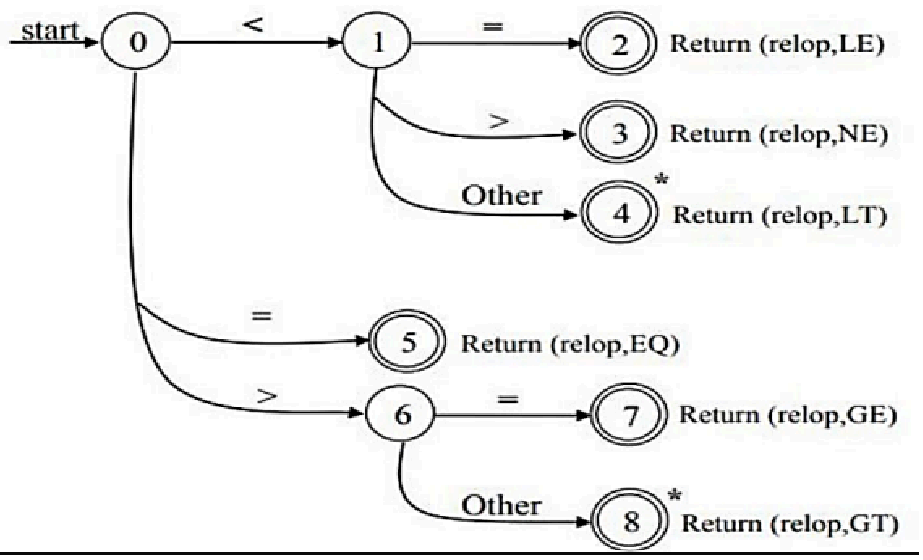
Position is a transition diagram are drawn as circles and are called states.

The states are connected by Arrows called edges. Labels on edges are indicating the input characters

Zero or more final states or Accepting states are represented by double circle in which the tokens has been found.

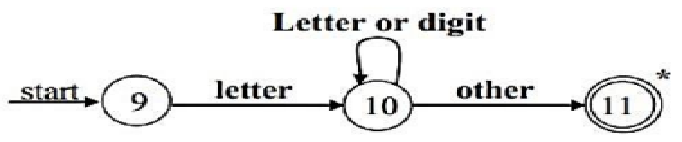
Here is the transition diagram of Finite Automata that recognizes the lexemes matching the token relop.

<u>**Example:**</u> A Transition Diagram for the token relation operators **"relop"** is shown in Figure below:



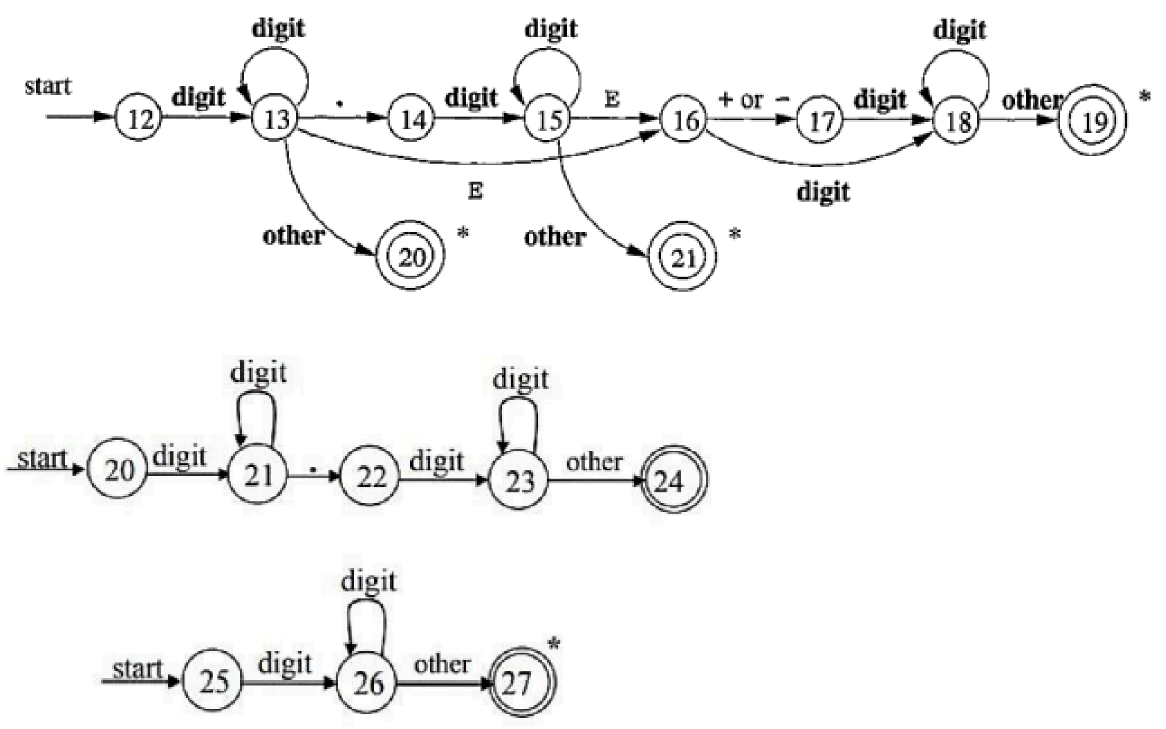
Here is the Finite Automata Transition Diagram for the Identifiers and Keywords.

<u>**Example:**</u> A Transition Diagram for the **identifiers and keywords**



Here is the Finite Automata Transition Diagram for the Unsigned Numbers

<u>**Example:**</u> A Transition Diagram for **Unsigned Numbers** in Pascal



num $\longrightarrow$ $digit^+(.\ digit^+ | \epsilon)(E(+|-|\epsilon)digit^+|\epsilon)$

Transition Diagram for whitespace:

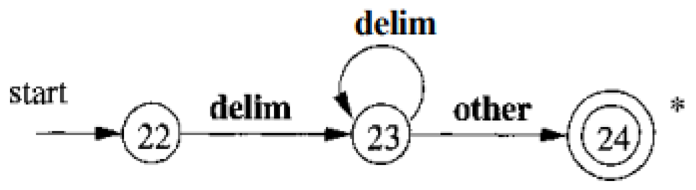


Figure 3.17: A transition diagram for whitespace