Lexical Analysis

Sukree Sinthupinyo[1]

[1]Department of Computer Engineering
Chulalongkorn University

14 July 2012

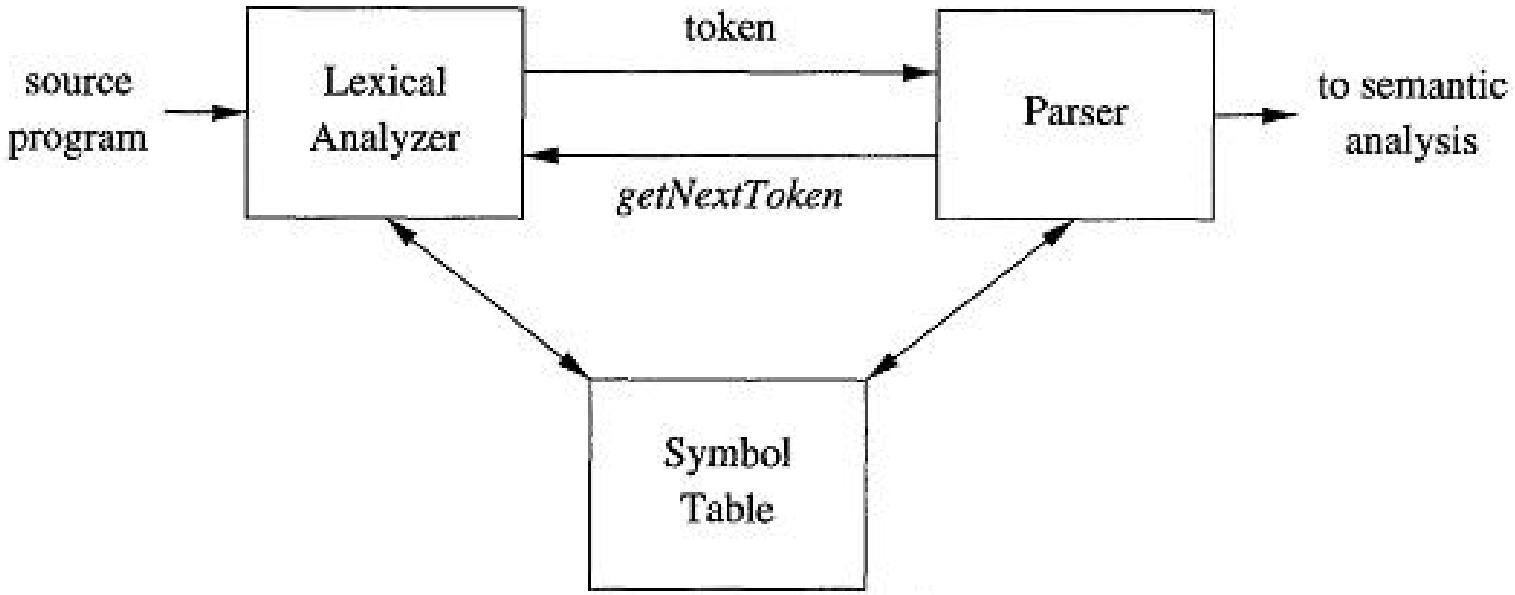
Outline

Learning Objectives

- Understand definition of *lexeme, token, etc.*
- Know a method which transforms string into token
- Know syntax of regular expression
- Know concept of transition diagram and code implemented from the diagram

First step

- The main task is to read the input characters of the source program and export a sequence of tokens.
- It also interacts with the symbol as well.



First step

- The lexical analyzer must
  - Strip out comments and whitespace.
  - Correlate error messages generated by the compiler with the source program

Tokens, Patterns, and Lexemes

- 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.
- A *pattern* is a description of the form that the lexemes of a token may take. For the keyword, 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.
- A *lexeme* is a sequence of characters in the source program that matches the pattern for a token.

Tokens, Patterns, and Lexemes

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

- `printf` and `score` are lexemes matching the pattern for token **id**
- `"Total = %d\n"` is a lexeme matching **literal**

Examples of tokens

Token	Informal Description	Sample Lexemes
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	< or > or <= or >= or == or !=	`<=,!=`
id	letter followed by letters and digits	`pi,score,D2`
number	any numeric constant	`3.14,6.02e23`
literal	anything but ", surrounded by "'s	`"core"`

General concept of tokens in many programming language

- One token for each keyword. The pattern for a keyword is the same as the keyword itself.
- Tokens for the operators
- One token representing all identifiers
- One or more tokens representing constants, such as numbers and literal strings.
- Tokens for each punctuation symbol, such as left and right parentheses, comma, and semi colon.

Attributes for Tokens

- Token must have an attribute associated with.
- For example, an **id** must associate with information about identifier; e.g., its lexeme, its type, and the location at which it is first found, is kept in the symbol table.

An Example of Attributes for Tokens

```
E = M * C ** 2
```

- <**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` >

String and Language

- A *string* over an alphabet is a finite sequence of symbols drawn from that alphabet. The length of string s is usually written $|s|$. The *empty string* is denoted ϵ.
- A *language* is any countable set of strings over some fixed alphabet.
- *Concatenation* of string x and y is the string formed by appending y to x. For example, if x = `dog` and y = `house`, then xy = `doghouse`.
- If we think of concatenation as a product, we can define the "exponentiation" of strings as follows. Define s^0 to be ϵ, and for all $i > 0$, define s^i to be $s^{i-1}s$. Since $\epsilon s = s$, it follows that $s^i = s$. Then $s^2 = ss$,$s^3 = sss$, and so on.

Operations on Languages

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Example

- Let L be the set of letters `A,B,...,Z,a,b,...,z`.
- D be the set of digits `0,1,...,9`.
  - $L \cup D$ is the set of letters and digits with 62 strings of length one.
  - LD is the set of 520 strings of length two.
  - L^4 is the set of all 4-letter strings.
  - L^* is the set of all strings of letter, including ϵ.
  - $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.
  - D^+ is the set of all strings of one or more digits.

Outline

Regular Expressions

- If we want to describe the set of valid *C* identifiers, we can use the language $L(L \cup D)$ with the underscore included among the letters.
- If *letter_* denotes any letter of the underscore, and *digit* stands for any digit, then we could describe the language of *C* identifiers by:

$$letter\_(letter\_|digit)^*$$

- where | denotes union, the parentheses are used to group subexpressions.

Regular Expressions

- Language $L(r)$ is defined recursively from the languages denoted by r's subexpressions using alphabet set $\sum$.
- **BASIS**: There are two rules that form the basis:
  1. ϵ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$, that is, the language whose sole member is the empty string.
  2. If a is a symbol in $\sum$, the **a** is a regular expression, and $L(\mathbf{a}) = \{a\}$, that is, the language with one string, of length one, with a in its one position.

Regular Expressions

- **INDUCTION**: The are four parts to the induction whereby larger expressions are built from the smaller one. Suppose r and s are regular expression denoting languages $L(r)$ and $L(s)$, respectively.
  1. $(r)|(s)$ denotes $L(r) \cup L(s)$.
  2. $(r)(s)$ denotes $L(r)L(s)$.
  3. $(r)^*$ denotes $L(r))^*$.
  4. (r) denotes $L(r)$.
- The precedence of operator is $*$, concatenation, and $|$.
- So (**a**)|((**b**)*(**c**)) can be written as **a**|**b*****c**

Regular Expressions

- Example
  - Let $\sum = \{a, b\}$
  - **a|b** denotes the language $\{a, b\}$
  - (**a|b**)(**a|b**) denotes $\{aa, ab, ba, bb\}$
  - **a*** denotes $\{a, aa, aaa, \ldots\}$.
  - (**a|b**)* denotes $\{\epsilon, a, b, aa, ab, ba, bb, aaa, ...\}$
  - **a|a*****b** denotes $\{a, b, ab, aab, aaab, ...\}$

LAW	DESCRIPTION
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over $\|$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	$*$ is idempotent

Definitions

Regular definition is a sequence of the form

$$
\begin{aligned}
d_1 &\rightarrow r_1 \\
d_2 &\rightarrow r_2 \\
&\dots \\
d_n &\rightarrow r_n
\end{aligned}
$$

Regular Definition Example

- *C* identifiers are strings of letters, digits, and underscore.

$$\begin{aligned} letter\_ &\rightarrow A|B|\ldots|Z|a|b|\ldots|z|\_ \\ digit &\rightarrow 0|1|\ldots|9 \\ id &\rightarrow letter\_(letter\_|digit)^* \end{aligned}$$

Extensions of Regular Expressions

- $+$: *One or more instances*
- ?: *Zero or one instances*
- $[a_1 a_2 \ldots a_n]$: $a_1|a_2|\ldots|a_n$ or $a_1 - a_n$

$$letter\_ \rightarrow [A - Za - z\_]$$
$$digit \rightarrow [0 - 9]$$
$$id \rightarrow letter\_(letter\_|digit)^*$$

Example

$$
\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\
 & | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \textbf{ else } \mathit{stmt} \\
 & | & \epsilon \\
\mathit{expr} & \rightarrow & \mathit{term} \textbf{ relop } \mathit{term} \\
 & | & \mathit{term} \\
\mathit{term} & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}
$$

Example

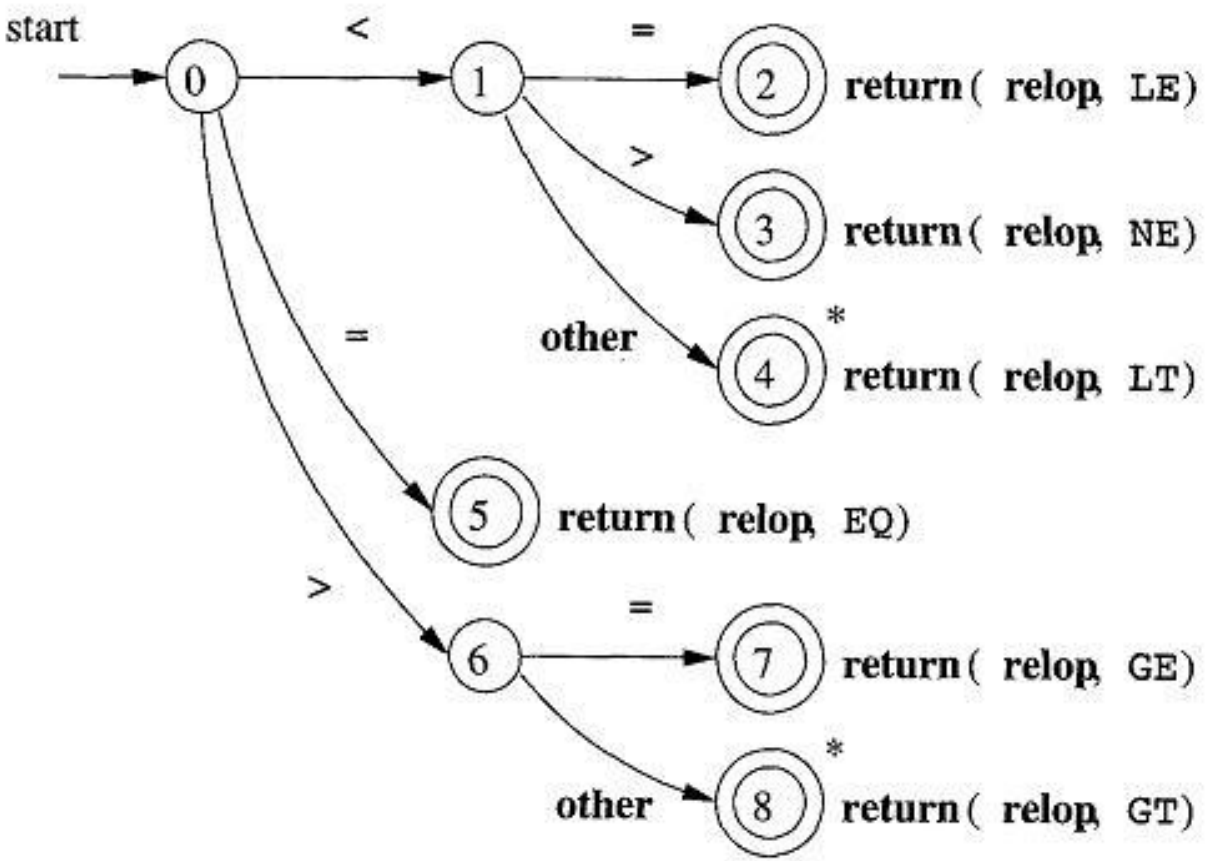
digit	$\rightarrow$	`[0-9]`
digits	$\rightarrow$	*digit*$^+$
number	$\rightarrow$	*digits* (`.` *digits*)? (`E` `[+-]`? *digits*)?
letter	$\rightarrow$	`[A-Za-z]`
id	$\rightarrow$	*letter* (*letter* | *digit*)*
if	$\rightarrow$	`if`
then	$\rightarrow$	`then`
else	$\rightarrow$	`else`
relop	$\rightarrow$	`<` | `>` | `<=` | `>=` | `=` | `<>`

Tokens, Patterns, and Attribute Values

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**	GE

Outline

Transition Diagram for **relop**



Example Code for **relop**

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```