Compiler Design

Fatemeh Deldar

Isfahan University of Technology

1402-1403

Lexical Analysis

- 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
  1. **Stripping out comments and whitespace**
  2. **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

Lexical Analysis

- Sometimes, lexical analyzers are divided into a cascade of two processes:
  1. **Scanning** consists of the simple processes that do not require tokenization of the input, such as:
     - **Deletion of comments**
     - **Compaction of consecutive whitespace characters into one**
  2. **Lexical analysis** produces tokens from the output of the scanner

Lexical Analysis

- What does the lexical analyzer want to do?
- **Example**

```
if (i == j)
  Z = 0;
else
  Z = 1;
```

- The input is just a string of characters:
    - **\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;**
- **Goal:** Partition input string into substrings where the substrings are called **tokens**

Tokens

- **What's a Token?**
  - A syntactic category
    - **In English:**
      - **noun, verb, adjective, …**
    - **In a programming language:**
      - **Identifier, Integer, Keyword, Whitespace, …**
- A token class corresponds to a set of strings
  - **Examples**
    - **Identifier**: Strings of letters or digits, starting with a letter
    - **Integer**: A non-empty string of digits
    - **Keyword**: "else" or "if" or "for" or …
    - **Whitespace**: A non-empty sequence of blanks, newlines, and tabs
- **What are Tokens For?**
  - Classify program substrings according to role
    - An identifier is treated differently than a keyword

Tokens, Patterns, and Lexemes

- **A token** is a pair consisting of a **token name** and **an optional attribute value**
  - The token names are the input symbols that the parser processes
- **A pattern** is a description of the form that the lexemes of a token may take
  - **Example**
    - **In the case of a keyword as a token**, the pattern is the sequence of characters that form the keyword
    - **In the case of an identifier as a token**, 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

Tokens, Patterns, and Lexemes

- In many programming languages, the following classes cover most or all of the tokens:
  1. **One token for each keyword**
     - The pattern for a keyword is the same as the keyword itself
  2. **Tokens for the operators**, such as the token comparison
  3. **One token representing all identifiers**
  4. **One or more tokens representing constants**, such as numbers and literal strings
  5. **Tokens for each punctuation symbol**, such as left and right parentheses, comma, and semicolon

Tokens, Patterns, and Lexemes

- **Example**

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	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Attributes for Tokens

- We can assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information
- **Example**
  - Information about an **identifier** is kept in the symbol table, including:
    - Its lexeme
    - Its type
    - The location at which it is first found
  - **The appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier**

Attributes for Tokens

- **Example**
  - The token names and associated attribute values for the Fortran statement: 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>

  - *Note that in certain pairs, especially operators, punctuation, and keywords,* ***there is no need for an attribute value***

Specification of Tokens

- **Regular expressions** are an important notation for specifying lexeme patterns
- **Strings and Languages**
  - An **alphabet** is any finite set of symbols
    - Letters, digits, and punctuation
  - A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet
    - The terms "sentence" and "word" are often used as synonyms for "string"
  - A **language** is any countable set of strings over some fixed alphabet

Operations on Languages

- In lexical analysis, the most important operations on languages are **union**, **concatenation**, and **closure**

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} and let D be the set of digits {0, 1, …, 9}
    - $L \cup D$, LD, L^4, L^*, $L(L \cup D)^*$, D^+

Regular Expressions

- **Basis**
  - ϵ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$
  - If a is a symbol in $\sum$, then $\boldsymbol{a}$ is a regular expression, and $L(\boldsymbol{a}) = \{a\}$
- **Induction**
  1. $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$
  2. $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$
  3. $(r)^*$ is a regular expression denoting $(L(r))^*$
  4. (r) is a regular expression denoting $L(r)$

Regular Expressions

- **Precedences**
  - The unary operator * has highest precedence
  - Concatenation has second highest precedence
  - | has lowest precedence
- **Example**
  - $\sum = \{a, b\}$
  - $(a|b)(a|b) \rightarrow \{aa, ab, ba, bb\}$
- A language that can be defined by a regular expression is called a regular set

Regular Expressions

- Algebraic laws for regular expressions

LAW
$r\|s = s\|r$
$r\|(s\|t) = (r\|s)\|t$
$r(st) = (rs)t$
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$
$\epsilon r = r\epsilon = r$
$r^* = (r\|\epsilon)^*$
$r^{**} = r^*$

Regular Expressions

- **Example**
  - A regular definition for the language of C identifiers

$$
\begin{array}{rcl}
\mathit{letter\_} & \rightarrow & \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
\mathit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{id} & \rightarrow & \mathit{letter\_} \; (\; \mathit{letter\_} \mid \mathit{digit} \;)^*
\end{array}
$$

  - A regular definition for unsigned numbers (such as 5280, 0.034, 6.36E4, or 1.89E-4)

$$
\begin{array}{rcl}
\mathit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{digits} & \rightarrow & \mathit{digit} \; \mathit{digit}^* \\
\mathit{optionalFraction} & \rightarrow & \texttt{.} \; \mathit{digits} \mid \epsilon \\
\mathit{optionalExponent} & \rightarrow & (\; \texttt{E} \; (\; \texttt{+} \mid \texttt{-} \mid \epsilon \;) \; \mathit{digits} \;) \mid \epsilon \\
\mathit{number} & \rightarrow & \mathit{digits} \; \mathit{optionalFraction} \; \mathit{optionalExponent}
\end{array}
$$

Regular Expressions

- **Example**
  - Describe the languages denoted by the following regular expressions:

a) $\mathbf{a}(\mathbf{a}|\mathbf{b})^*\mathbf{a}$.

b) $((\epsilon|\mathbf{a})\mathbf{b}^*)^*$.

c) $(\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$.

d) $\mathbf{a}^*\mathbf{b}\mathbf{a}^*\mathbf{b}\mathbf{a}^*\mathbf{b}\mathbf{a}^*$.

e) $(\mathbf{aa}|\mathbf{bb})^*((\mathbf{ab}|\mathbf{ba})(\mathbf{aa}|\mathbf{bb})^*(\mathbf{ab}|\mathbf{ba})(\mathbf{aa}|\mathbf{bb})^*)^*$.

Recognition of Tokens

- **Example**
  - A grammar for branching statements (for Pascal language)

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

Recognition of Tokens

- **Example**
  - Patterns for tokens

$$
\begin{array}{rcl}
\textit{digit} & \to & [\texttt{0-9}] \\
\textit{digits} & \to & \textit{digit}^{+} \\
\textit{number} & \to & \textit{digits}\ (\texttt{.}\ \textit{digits})?\ (\ \texttt{E}\ [\texttt{+-}]?\ \textit{digits}\ )? \\
\textit{letter} & \to & [\texttt{A-Za-z}] \\
\textit{id} & \to & \textit{letter}\ (\ \textit{letter} \mid \textit{digit}\ )^{*} \\
\textit{if} & \to & \texttt{if} \\
\textit{then} & \to & \texttt{then} \\
\textit{else} & \to & \texttt{else} \\
\textit{relop} & \to & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}
$$

  - Assign the lexical analyzer the job of stripping out white-space

$$\boxed{\textit{ws} \to (\ \textbf{blank} \mid \textbf{tab} \mid \textbf{newline}\ )^{+}}$$

Recognition of Tokens

- **Example**
  - Tokens, their 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`