Compiler Design

IT 423

Lecture - 3
Bootstrapping is an important concept in building new compiler. For constructing any new compiler, we require three language.

- Source Language (S) – which actually compiles
- Target Language (T) – which generate code for target language
- Implementation Language (I) – the language in which it is written

This can be shown in the form of T-diagram.

represents a compiler for Source S, Target T, implemented in I
A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.

For example, a compiler that runs on a Windows 7 PC but generates code that runs on Android smartphone is a cross compiler.
Lexical analysis

- Role of lexical analyzer
- Specification of tokens
- Recognition of tokens
- Lexical analyzer generator
- Finite automata
- Design of lexical analyzer generator
The lexical analysis is the first phase of a compiler. Its main task is to read the input characters and produces as output a sequence of token that the parser uses for syntax analysis. Interaction of lexical analysis is summarized as below.
Role of lexical analyzer

Issues in design of lexical analysis

✔️ Several reasons for separating the analysis phases of compiling into lexical analysis and parsing

- Simpler design is perhaps the most important consideration. The separation of lexical analysis from syntax often allows us to simplify one or the other of these phases.

- Compiler efficiency is improved. A separate lexical analysis allows us to construct a specialized and potentially more efficient processor for the task.
Role of lexical analyzer

Processes in lexical analyzers

- Scanning
  - Pre-processing
    - Strip out comments and white space
    - Macro functions
  - Correlating error messages from compiler with source program
    - A line number can be associated with an error message
- Lexical analysis
Role of lexical analyzer

Terms of the lexical analyzer

- **Token**
  - Types of words in source program
  - Keywords, operators, identifiers, constants, literal strings, punctuation symbols (such as commas, semicolons)

- **Lexeme**
  - Actual words in source program

- **Pattern**
  - A rule describing the set of lexemes that can represent a particular token in source program

- **Relation** \{ <, <=, >, >=, ==, <> \}
### Role of lexical analyzer

#### Example

<table>
<thead>
<tr>
<th>Token</th>
<th>Informal description</th>
<th>Sample lexemes</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>Characters i, f</td>
<td>if</td>
</tr>
<tr>
<td>else</td>
<td>Characters e, l, s, e</td>
<td>else</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt; or &gt; or &lt;= or &gt;= or == or !=</td>
<td>&lt;=, !=</td>
</tr>
<tr>
<td>id</td>
<td>Letter followed by letter and digits</td>
<td>pi, score, D2</td>
</tr>
<tr>
<td>number</td>
<td>Any numeric constant</td>
<td>3.14159, 0, 6.02e23</td>
</tr>
<tr>
<td>literal</td>
<td>Anything but “ surrounded by “</td>
<td>“core dumped”</td>
</tr>
</tbody>
</table>
Role of lexical analyzer

Attributes for Tokens

- A pointer to the symbol-table entry in which the information about the token is kept

E.g  \( E = M \times C^{**2} \)

\(<\text{id}, \text{pointer to symbol-table entry for } E>\)
\(<\text{assign\_op,}>\)
\(<\text{id}, \text{pointer to symbol-table entry for } M>\)
\(<\text{multi\_op,}>\)
\(<\text{id}, \text{pointer to symbol-table entry for } C>\)
\(<\text{exp\_op,}>\)
\(<\text{num, integer value 2}>\)
Role of lexical analyzer

Lexical Errors

- Deleting an extraneous character
- Inserting a missing character
- Replacing an incorrect character by a correct character
- Transposing two adjacent characters (such as , fi=>if)
- Pre-scanning
Role of lexical analyzer

Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters
Role of lexical analyzer

Input Buffering

➢ Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
  • In C language: we need to look after -, = or < to decide what token to return
  • In Fortran: DO 5 I = 1.25
➢ We need to introduce a two buffer scheme to handle large look-aheads safely

\[ E = M \times C^{* * 2} \text{eof} \]
Approach for implementation of a lexical analyzer

- Three general approach to the implementation of a lexical analyzer.
  - Use a lexical analyzer generation, such as LEX compiler to produce the lexical analyzer from a regular expression based specification. In this case, the generator provides routines for reading and buffering the input.
  - Write the lexical analyzer in a conventional system programming language, using the I/O facilities of that language to read the input.
  - Write the lexical analyzer in assembly language and explicitly manage the reading of input.
Alphabet

- An alphabet, is any finite set of symbols. Typical examples of symbols are letters, digits, and punctuation. E.g. \{a,b,c\}, \{0,1\}.
- The set \{0,1\} is the binary alphabet.
- ASCII is an important example of an alphabet.
String

- A string over some alphabet is a finite sequence of symbols drawn from that alphabet.
- In language theory, the terms sentence and word are often used as synonyms for ‘string’.
- The length of string $S$, usually written $|S|$, is the no. of occurrence of symbol in $S$.
- For ex. ‘banana’ is a string of length six.
- The empty string, denoted $\varepsilon$, is the string of length zero.
A language is any countable set of strings over some fixed alphabet.

If 'x' and 'y' are two strings, then concatenation of 'x' and 'y' written as 'xy' is the string formed by appending 'y' to 'y'.

For ex. If 'x'='dog' and 'y'='house' then 'xy'='dog house' and 'yx'='house dog'

xy ≠ yx
## Alphabet, Strings and Languages

### Terms for parts of strings

<table>
<thead>
<tr>
<th>Terms</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix of S</td>
<td>A string obtained by removing zero or more trailing symbols of string S, e.g. ‘ban’ is a prefix of ‘banana’</td>
</tr>
<tr>
<td>Suffix of S</td>
<td>A string formed by deleting zero or more of the leading symbols of S, e.g. ‘nana’ is suffix of ‘banana’</td>
</tr>
<tr>
<td>Substring of S</td>
<td>A string obtained by deleting a prefix and a suffix from S. e.g. ‘nan’ is a substring of ‘banana’. Every prefix and every suffix of S is a substring of S is a prefix or a suffix of S. for every string S, both S and ( \varepsilon ) are prefixes, suffixes, and substring of S.</td>
</tr>
<tr>
<td>Proper prefix, suffix or substring of S</td>
<td>Any nonempty string ( x ) that is, respectively a prefix, suffix or substring of S, such that ( S \neq x )</td>
</tr>
<tr>
<td>Subspace of x</td>
<td>Any string formed by deleting zero or more not necessarily consecutive positions of S. for ex. ‘baan’ is a subsequence of ‘banana’</td>
</tr>
</tbody>
</table>
Operation on language

➢ In lexical analysis, the most important operations on languages are union, concatenation, and closure.

Union of two language

➢ Let two language are Land M, then union of these two language is LUM, and defined as \[ L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \} \]

Concatenation of two language

➢ Let two language are L and M, then concatenation of these two language is LM, and defined as \[ LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \} \]
Klene Closure of Language

Let $L$ be a language, then Klene closure of this language is written as $L^*$

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

Positive Closure of Language

Let $L$ be a language, then Klene closure of this language is written as $L^+$

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$
Specification of Tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
  - Letter_ (letter_ | digit)*
- Each regular expression is a pattern specifying the form of strings
Regular Expression

- ε is a regular expression, L(ε) = {ε}
- If a is a symbol in Σ then a is a regular expression, L(a) = {a}
- (r) | (s) is a regular expression denoting the language L(r) ∪ L(s)
- (r)(s) is a regular expression denoting the language L(r)L(s)
- (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denoting L(r)
Regular Expression

Algebraic laws of regular expressions

1) $\alpha|\beta = \beta|\alpha$

2) $\alpha|(|\beta|\gamma|) = (\alpha|\beta)|\gamma \quad \alpha(\beta\gamma) = (\alpha \beta)\gamma$

3) $\alpha(\beta|\gamma) = \alpha\beta | \alpha\gamma \quad (\alpha|\beta)\gamma = \alpha\gamma| \beta\gamma$

4) $\varepsilon\alpha = \alpha\varepsilon = \alpha$

5) $(\alpha^*)^* = \alpha^*$

6) $\alpha^* = \alpha^+ | \varepsilon \quad \alpha^+ = \alpha \quad \alpha^* = \alpha^*\alpha$

7) $(\alpha|\beta)^* = (\alpha^* |\beta^*)^* = (\alpha^* \beta^*)^*$
Regular Expression

Algebraic laws of regular expressions

8) If $\varepsilon \notin L(\gamma)$, then

$$\alpha = \beta | \gamma \alpha \quad \alpha = \gamma^* \beta$$

$$\alpha = \beta | \alpha \gamma \quad \alpha = \beta \gamma^*$$

Notes: We assume that the precedence of * is the highest, the precedence of | is the lowest and they are left associative.
Regular Definition

d1 -> r1
d2 -> r2
...
dn -> rn

Each di is a new symbol, not in $\Sigma$ and not the same as any other of the di’s

Example:

letter_ -> A | B | … | Z | a | b | … | Z | _
digit -> 0 | 1 | … | 9
id -> letter_ (letter_ | digit)*
- Regular Expression
- Transition Diagram
- Context free Grammar
- Finite Automata
  - Deterministic Finite Automata
  - Non Deterministic Finite Automata
- Conversion for DFA to NDFA and vice versa
Recognition of tokens

Task of recognition of token in a lexical analyzer

- Isolate the lexeme for the next token in the input buffer
- Produce as output a pair consisting of the appropriate token and attribute-value, such as `<id,pointer to table entry>`, using the translation table given in the Fig in next page

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Token</th>
<th>Attribute-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>if</td>
<td>-</td>
</tr>
<tr>
<td>id</td>
<td>id</td>
<td>Pointer to table entry</td>
</tr>
<tr>
<td>&lt;</td>
<td>relop</td>
<td>LT</td>
</tr>
</tbody>
</table>