The Role of Lexical Analyzer

Compiler Design Lexical Analysis
s.l. dr. ing. Ciprian-Bogdan Chirila
chirila@cs.upt.ro
http://www.cs.upt.ro/~chirila
Outline

- Lexical Analysis vs. Parsing
- Tokens, Patterns and Lexemes
- Attributes for Tokens
- Lexical Errors
Lexical Analysis

- **Manual approach – by hand**
  - To identify the occurrence of each lexeme
  - To return the information about the identified token

- **Automatic approach - lexical-analyzer generator**
  - Compiles lexeme patterns into code that functions as a lexical analyzer
  - e.g. Lex, Flex, …
  - **Steps**
    - Regular expressions - notation for lexeme patterns
    - Nondeterministic automata
    - Deterministic automata
    - Driver - code which simulates automata
The Role of the Lexical Analyzer

- Read input characters
- To group them into lexemes
- Produce as output a sequence of tokens
  - input for the syntactical analyzer
- Interact with the symbol table
  - Insert identifiers

Diagram: 
- Source program -> Lexical Analyzer
- Lexical Analyzer -> Parser
- Parser -> Semantic analysis
- Lexical Analyzer <-> Symbol Table
The Role of the Lexical Analyzer

- to strip out
  - comments
  - whitespaces: blank, newline, tab, …
  - other separators

- to correlate error messages generated by the compiler with the source program
  - to keep track of the number of newlines seen
  - to associate a line number with each error message
Lexical Analyzer Processes

- Scanning
  - to not require input tokenization
  - deletion of comments
  - compaction of consecutive white spaces into one

- Lexical analysis
  - to produce sequence of tokens as output
Lexical Analysis vs. Parsing

- **Simplicity of design**
  - Separation of lexical from syntactical analysis -> simplify at least one of the tasks
  - e.g. parser dealing with white spaces -> complex
  - Cleaner overall language design

- **Improved compiler efficiency**
  - Liberty to apply specialized techniques that serves only lexical tasks, not the whole parsing
  - Speedup reading input characters using specialized buffering techniques

- **Enhanced compiler portability**
  - Input device peculiarities are restricted to the lexical analyzer
Tokens, Patterns, Lexemes

- **Token** - pair of:
  - token name – abstract symbol representing a kind of lexical unit
    - keyword, identifier, …
  - optional attribute value

- **Pattern**
  - description of the form that the lexeme of a token may take
  - e.g.
    - for a keyword the pattern is the character sequence forming that keyword
    - for identifiers the pattern is a complex structure that is matched by many strings

- **Lexeme**
  - a sequence of characters in the source program matching a pattern for a token
# Examples of Tokens

<table>
<thead>
<tr>
<th>Token</th>
<th>Informal Description</th>
<th>Sample Lexemes</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>if</em></td>
<td>characters i, f</td>
<td>if</td>
</tr>
<tr>
<td><em>else</em></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><em>id</em></td>
<td>Letter followed by letters and digits</td>
<td>pi, score, D2</td>
</tr>
<tr>
<td>number</td>
<td>Any numeric constant</td>
<td>3.14159, 0, 02e23</td>
</tr>
<tr>
<td>literal</td>
<td>Anything but “, surrounded by“</td>
<td>“core dumped”</td>
</tr>
</tbody>
</table>
Examples of Tokens

- One token for each keyword
  - Keyword pattern = keyword itself
- Tokens for operators
  - Individually or in classes
- One token for all identifiers
- One or more tokens for constants
  - Numbers, literal strings
- Tokens for each punctuation symbol
  - ( ), ;
Attributes for Tokens

- more than one lexeme can match a pattern
- token **number** matches 0, 1, 100, 77,…
- lexical analyzer must return
  - Not only the token name
  - Also an attribute value describing the lexeme represented by the token
- token **id** may have associated information like
  - lexeme
  - type
  - location – in order to issue error messages
- token **id** attribute
  - pointer to the symbol table for that identifier
Tricky Problems in Token Recognition

- Fortran 90 example
  - assignment
    
    DO 5 I = 1.25
    DO51 = 1.25

  - do loop
    
    DO 5 I = 1,25
    DO5I = 1,25
Example of Attribute Values

- \( E = M \times C \times C \)
  - <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>
Lexical Errors

- can not be detected by the lexical analyzer alone
  - fi (a == f(x) ) …
- lexical analyzer is unable to proceed
  - none of the patterns matches any prefix of the remaining input
  - “panic mode” recovery strategy
    - delete one/successive characters from the remaining input
    - insert a missing character into the remaining input
    - replace a character
    - transpose two adjacent characters
Outline

- Input Buffering
  - Buffer Pairs
  - Sentinels
Input Buffering

- How to speed the reading of source program?
- to look one additional character ahead
- e.g.
  - to see the end of an identifier you must see a character
    - which is not a letter or a digit
    - not a part of the lexeme for id
  - in C
    - -, = , <
    - ->, ==, <=
- two buffer scheme that handles large lookaheads safely
- sentinels – improvement which saves time checking buffer ends
Buffer Pairs

- Buffer size N
- N = size of a disk block (4096)
- read N characters into a buffer
  - one system call
  - not one call per character
- read < N characters we encounter `eof`
- two pointers to the input are maintained
  - `lexemeBegin` – marks the beginning of the current lexeme
  - `forward` – scans ahead until a pattern match is found
Sentinels

- **forward pointer**
  - to test if it is at the end of the buffer
  - to determine what character is read (multiway branch)

- **sentinel**
  - added at each buffer end
  - can not be part of the source program
  - character `eof` is a natural choice
    - retains the role of entire input end
    - when appears other than at the end of a buffer it means that the input is at an end
Lookahead Code with Sentinels

switch(*forward++)
{
    case eof:
        if(forward is at the end of the first buffer)
        {
            reload second buffer;
            forward = beginning of the second buffer;
        }
        elseif(forward is at the end of the second buffer)
        {
            reload first buffer;
            forward = beginning the first buffer;
        }
    else
        /*eof within a buffer marks the end of input*/
        terminate lexical analysis;
        break;
    cases for the other characters
}
Potential Problems

- **usually**
  - lexemes are short
  - 1-2 characters lookahead are sufficient

- **problem: running out of buffer space**
  - when $N = 3,4,5 \times 1000$
  - long character strings $> N$

- **solution: concatenation of string components by grammar rules** (like in Java using the + operator to concatenate multiline strings)

- **long lookahead**
  - languages where keywords are not reserved
  - in PL/I:
    - DECLARE (ARG1, ARG2, …)
    - ambiguous identifier resolved by the parser (keyword or procedure name)
Bibliography