Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-15/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 5

• Lexical analysis: implementing a scanner
The Reason Why Lexical Analysis is a Separate Phase

• Simplifies the design of the compiler
  – LL(1) or LR(1) parsing with 1 token lookahead would not be possible (multiple characters/tokens to match)

• Provides efficient implementation
  – Systematic techniques to implement lexical analyzers by hand or automatically from specifications
  – Stream buffering methods to scan input

• Improves portability
  – Non-standard symbols and alternate character encodings can be normalized (e.g. UTF8, trigraphs)
Main goal of lexical analysis: tokenization

source code

\[ y := 31 + 28 \times x \]

Lexical analyzer
or
Scanner

Parser

(token attribute)

(token attribute)

(token lookahead)
Additional tasks of the Lexical Analyzer

• Remove comments and useless white spaces / tabs from the source code
• Correlate error messages of the parser with source code (e.g. keeping track of line numbers)
• Expansion of macros
Interaction of the Lexical Analyzer with the Parser

Source Program

Lexical Analyzer

Parser

Symbol Table

Get next token

Token, tokenval

error

error
Tokens, Patterns, and Lexemes

• A *token* is a pair `<token name, attribute>`
  – The token name (e.g. `id`, `num`, `div`, `geq`, ...) identifies the category of lexical units
  – The attribute is optional
  – **NOTE**: most often, one refers to a token using the *token name* only

• A *lexeme* is a character string that makes up a token
  – For example: `abc`, `123`, `\`, `>=`

• A *pattern* is a rule describing the set of lexemes belonging to a token
  – For example: “*letter followed by letters and digits*”, “*non-empty sequence of digits*”, “*character ‘\’*”, “*character ‘>’ followed by ‘=’*”

• The scanner reads characters from the input till when it recognizes a lexeme that matches the patterns for a token
## Example

<table>
<thead>
<tr>
<th>Token name</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>relation</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 &quot; surrounded by &quot;</td>
<td>&quot;core dumped&quot;</td>
</tr>
</tbody>
</table>
Attributes of tokens

- Needed when the pattern of a token matches different lexemes
- We assume single attribute, but can be structured
- Typically ignored by parsing, but used in subsequent compilation phases (static analysis, code generation, optimization)
- Kind of attribute depends on the token name
- Identifiers have several info associated (lexeme, type, position of definition,...)
  - Typically inserted as entries in a symbol table, and the attribute is a pointer to the simbol-table entry
Reading input characters

• Requires I/O operations: efficiency is crucial
• Lookahead can be necessary to identify a token
• Buffered input reduces I/O operations
• Naïve implementation makes two tests for each character
  – End of buffer?
  – Multiway branch on the character itself
• Use of “sentinels” encapsulate the end-of-buffer test into the multiway branch
Buffered input to Enhance Efficiency

### Lexeme Beginning

|     | E | = | M * | C * | * 2 | eof |     |

- **Buffered Input**: Enhances Efficiency

- **Current Token**

- **Forward**: Scans ahead to find pattern match

### Code

```pseudocode
if forward at end of first half then begin
    reload second half;
    forward := forward + 1
end
else if forward at end of second half then begin
    reload first half;
    move forward to beginning of first half
end
else forward := forward + 1;
```

- **Block I/O**: Executed for each input character
Algorithm: Buffered I/O with Sentinels

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

- **Current token**
- **Forward (scans ahead to find pattern match)**
- **Executed only is next character is eof**

```
forward := forward + 1;
if forward is at eof then begin
  if forward at end of first half then begin
    reload second half;
    forward := forward + 1
  end
  else if forward at end of second half then begin
    reload first half;
    move forward to beginning of first half
  end
  else /* eof within buffer signifying end of input */
  terminate lexical analysis
end
```

2nd eof ⇒ no more input!
Specification of Patterns for Tokens: Recalling some basic definitions

• An alphabet \( \Sigma \) is a finite set of symbols (characters)
• A string \( s \) is a finite sequence of symbols from \( \Sigma \)
  – \( |s| \) denotes the length of string \( s \)
  – \( \varepsilon \) denotes the empty string, thus \( |\varepsilon| = 0 \)
  – \( \Sigma^* \) denotes the set of strings over \( \Sigma \)
• A language \( L \) over \( \Sigma \) is a set of strings over alphabet \( \Sigma \)
• Thus \( L \subseteq \Sigma^* \), or \( L \in 2^{\Sigma^*} \)
  – \( 2^X \) is the powerset of \( X \), i.e. the set of all subsets of \( X \)
• The concatenation of strings \( x \) and \( y \) is denoted by \( xy \)
• Exponentiation of a string \( s \): \( s^0 = \varepsilon \) \( s^i = s^{i-1}s \) for \( i > 0 \)
Operations on Languages

• Languages are sets (of strings) thus all operations on sets are defined over them
  – Eg. Union: \( L \cup M = \{ s \mid s \in L \text{ or } s \in M \} \)

• Additional operations lift to languages operations on strings
  – Concatenation \( LM = \{ xy \mid x \in L \text{ and } y \in M \} \)
  – Exponentiation \( L^0 = \{ \varepsilon \}; \quad L^i = L^{i-1}L \)
  – Kleene closure \( L^* = \bigcup_{i=0,\ldots,\infty} L^i \)
  – Positive closure \( L^+ = \bigcup_{i=1,\ldots,\infty} L^i \)
Language Operations: Examples

$L = \{a, b, ab, ba\}$ \hspace{1cm} $D = \{1, 2, ab, b\}$ \hspace{1cm} Assuming $\Sigma = \{a, b, 1, 2\}$

- $L \cup D = \{a, b, ab, ba, 1, 2\}$
- $LD = \{a1, a2, aab, ab, b1, b2, bab, bb, ab1, ab2, abab, abb, ba1, ba2, baab, bab\}$
- $L^2 = \{aa, ab, aab, aba, ba, bb, bab, bba, abb, abab, abba, baa, bab, baab, baba\}$
- $L^* = \{\varepsilon, 1, 2, ab, b, 11, 12, \ldots, 111, 112, \ldots, 1111, 1112, \ldots\}$
- $L^+ = L^* - \{\varepsilon\}$
Regular Expressions: syntax and semantics

• Given an alphabet $\Sigma$, a *regular expression over* $\Sigma$ denotes a language over $\Sigma$ and is defined as follows:

• Basis symbols:
  – $\varepsilon$ is a regular expression denoting language $\{\varepsilon\}$
  – $a$ is a regular expression denoting $\{a\}$, for each $a \in \Sigma$

• If $r$ and $s$ are regular expressions denoting languages $L(r)$ and $M(s)$ respectively, then
  – $(r) \mid (s)$ is a regular expression denoting $L(r) \cup M(s)$
  – $(r)(s)$ is a regular expression denoting $L(r)M(s)$
  – $(r)^*$ is a regular expression denoting $L(r)^*$
  – $(r)$ is a regular expression denoting $L(r)$

• A language defined by a regular expression is called a *regular language*
Regular Expressions: conventions and examples

• Syntactical conventions to avoid too many brackets:
  – Precedence of operators: \( (\_)^* > (\_)(\_ > (\_)|\(\_\)
  – Left-associativity of all operators
  – Example: \((a)\mid((b)^*(c))\) can be written as \(a\mid b^*c\)

• Examples of regular expressions (over \(\Sigma = \{a, b\}\)):
  – \(a\mid b\) denotes \(\{a, b\}\)
  – \((a\mid b)(a\mid b)\) denotes \(\{aa, ab, ba, bb\}\)
  – \(a^*\) denotes \(\{\epsilon, a, aa, aaa, aaaa, \ldots\}\)
  – \((a\mid b)^*\) denotes \(\{\epsilon, a, b, aa, ab, \ldots, aaa, aab, \ldots\} = \Sigma^*\)
  – \((a^*b^*)^*\) denotes ?

• Two regular expressions are equivalent if they denote the same language. Eg: \((a\mid b)^* = (a^*b^*)^*\)
Some Algebraic Properties of Regular Expressions

<table>
<thead>
<tr>
<th>LAW</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r \</td>
<td>\ s = s \</td>
</tr>
<tr>
<td>$r \</td>
<td>\ (s \</td>
</tr>
<tr>
<td>$(r \ s) \ t = r (s \ t)$</td>
<td>concatenation is associative</td>
</tr>
<tr>
<td>$r (s \</td>
<td>\ t) = rs \</td>
</tr>
<tr>
<td>$(s \</td>
<td>\ t) r = sr \</td>
</tr>
<tr>
<td>$\varepsilon r = r$</td>
<td>$\varepsilon$ is the identity element for concatenation</td>
</tr>
<tr>
<td>$r \varepsilon = r$</td>
<td></td>
</tr>
<tr>
<td>$r^* = (r \</td>
<td>\ \varepsilon)^*$</td>
</tr>
<tr>
<td>$r^{**} = r^*$</td>
<td>* is idempotent</td>
</tr>
</tbody>
</table>

- Equivalence of regular expressions is decidable
- There exist complete axiomatizations
Regular Definitions

• Provide a convenient syntax, similar to BNF, for regular expressions, introducing name-to-regular-expression bindings.

• A regular definition has the form

\[
\begin{align*}
  d_1 & \rightarrow r_1 \\
  d_2 & \rightarrow r_2 \\
  \vdots \\
  d_n & \rightarrow r_n
\end{align*}
\]

where each \( r_i \) is a regular expression over \( \Sigma \cup \{d_1, \ldots, d_{i-1}\} \)

• Recursion is forbidden! \( digits \rightarrow digit \mid digit\ digits \) wrong!

• Iteratively replacing names with the corresponding definition yields a single regular expression for \( d_n \)
Extensions of Regular Expressions

- Several operators on regular expressions have been proposed, improving expressivity and conciseness
- Modern scripting languages are very rich
- Clearly, each new operator must be definable with a regular expression
- Here are some common conventions

  - \([xyz]\) match one character \(x, y,\) or \(z\)
  - \[^{xyz}\] match any character except \(x, y,\) and \(z\)
  - \([a-z]\) match one of \(a\) to \(z\)
  - \(r^+\) positive closure (match one or more occurrences)
  - \(r^?\) optional (match zero or one occurrence)
Recognizing Tokens

• We described how to specify patterns of tokens using regular expressions/definitions
• Let’s show how to write code for recognizing tokens
• Recall: in the CFG of a language, terminal symbols correspond to the tokens the parser will use.

• Running example CFG:

• The tokens are:

if, then, else, relop, id, num

\[
\begin{align*}
stmt & \rightarrow \textit{if expr then stmt} \\
& \quad \mid \textit{if expr then stmt else stmt} \\
& \quad \mid \epsilon \\
expr & \rightarrow \textit{term relop term} \\
& \quad \mid \textit{term} \\
term & \rightarrow \textit{id} \\
& \quad \mid \textit{num}
\end{align*}
\]
## Running example: Informal specification of tokens and their attributes

<table>
<thead>
<tr>
<th>Pattern of lexeme</th>
<th>Token</th>
<th>Attribute-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any ws</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>if</td>
<td>if</td>
<td>-</td>
</tr>
<tr>
<td>then</td>
<td>then</td>
<td>-</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
<td>-</td>
</tr>
<tr>
<td>Any id</td>
<td>id</td>
<td>pointer to table entry</td>
</tr>
<tr>
<td>Any num</td>
<td>num</td>
<td>pointer to table entry</td>
</tr>
<tr>
<td>&lt;</td>
<td>relop</td>
<td>LT</td>
</tr>
<tr>
<td>&lt;=</td>
<td>relop</td>
<td>LE</td>
</tr>
<tr>
<td>=</td>
<td>relop</td>
<td>EQ</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>relop</td>
<td>NE</td>
</tr>
<tr>
<td>&gt;</td>
<td>relop</td>
<td>GT</td>
</tr>
<tr>
<td>&gt;=</td>
<td>relop</td>
<td>GE</td>
</tr>
</tbody>
</table>
Regular Definitions for tokens

• The specification of the patterns for the tokens is provided with regular definitions

\[
\begin{align*}
\text{letter} & \rightarrow [A-Za-z] \\
\text{digit} & \rightarrow [0-9] \\
\text{digits} & \rightarrow \text{digit}^+ \\
\text{if} & \rightarrow \text{if} \\
\text{then} & \rightarrow \text{then} \\
\text{else} & \rightarrow \text{else} \\
\text{relop} & \rightarrow < \mid \leq \mid \lt \mid > \mid \geq \mid = \\
\text{id} & \rightarrow \text{letter} (\text{letter} \mid \text{digit})^* \\
\text{num} & \rightarrow \text{digits} (\text{.digits})? (\text{E} (\text{+} \mid \text{-})? \text{digits})?
\end{align*}
\]
From Regular Definitions to code

• From the regular definitions we first extract a transition diagram, and next the code of the scanner.

• In the example the lexemes are recognized either when they are completed, or at the next character. In real situations a longer lookahead might be necessary.

• The diagrams guarantee that the longest lexeme is identified.
Coding Regular Definitions in Transition Diagrams

\[
\text{rel} \rightarrow < \mid \leq \mid \lessgtr \mid > \mid \geq \mid =
\]

\[
\text{id} \rightarrow \text{letter} ( \text{letter} \mid \text{digit} )^* \quad \text{letter or digit}
\]

\[
\text{return}(\text{relop}, \text{LE}) \quad \text{return}(\text{relop}, \text{NE}) \quad \text{return}(\text{relop}, \text{LT}) \quad \text{return}(\text{relop}, \text{EQ}) \quad \text{return}(\text{relop}, \text{GE}) \quad \text{return}(\text{relop}, \text{GT})
\]

\[
\text{return} (\text{gettoken}(), \text{install_id}())
\]
Coding Regular Definitions in *Transition Diagrams* (cont.)

Transition diagram for unsigned numbers

\[ \text{num} \rightarrow \text{digit}^+ (\cdot \text{digit}^+) ? (\ E (+ | -) ? \text{digit}^+ ) ? \]
From Individual Transition Diagrams to Code

• Easy to convert each Transition Diagram into code
• Loop with multiway branch (switch/case) based on the current state to reach the instructions for that state
• Each state is a multiway branch based on the next input channel
Coding the Transition Diagrams for Relational Operators

```
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);
        }
    }
    return(retToken);
}
```
Putting the code together

```
token nexttoken()
{ while (1) {
    switch (state) {
    case 0: c = nextchar();
      if (c==blank || c==tab || c==newline) {
          state = 0;
          lexeme_beginning++;
        }
      else if (c=='<') state = 1;
      else if (c=='=') state = 5;
      else if (c=='>') state = 6;
      else state = fail();
      break;
    case 1: ...
    case 9: c = nextchar();
      if (isletter(c)) state = 10;
      else state = fail();
      break;
    case 10: c = nextchar();
      if (isletter(c)) state = 10;
      else if (isdigit(c)) state = 10;
      else state = 11;
      break;
    ...
}
}
```

```
int fail()
{ forward = token_beginning;
    switch (state) {
    case 0: start = 9; break;
    case 9: start = 12; break;
    case 12: start = 20; break;
    case 20: start = 25; break;
    case 25: recover(); break;
    default: /* error */
    }
    return start;
}
```

The transition diagrams for the various tokens can be tried sequentially: on failure, we re-scan the input trying another diagram.
Putting the code together: Alternative solutions

• The diagrams can be checked in parallel
• The diagrams can be merged into a single one, typically *non-deterministic*: this is the approach we will study in depth.
Lexical errors

• Some errors are out of power of lexical analyzer to recognize:

  fi (a == f(x)) …

• However, it may be able to recognize errors like:

  d = 2r

• Such errors are recognized when no pattern for tokens matches a character sequence
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
- Minimal Distance