Topic: Lexical Analysis

In Brief: Lexical analyzer, the first phase of the compilation process, converts an input program into a stream of tokens, and collects and stores their attributes.

Questions and issues to address:
- a word (token) is a smallest language unit above an alphabet character
- how to precisely and concisely describe tokens?
- how to use the description to identify tokens?

Scope:
- A overview of stages of compilation process.
- Language as a set of strings.
-Lexical analysis as a part of the compilations process.
- Theory and practice of lexical analysis; Finite State Automata, Regular Grammars, Regular Expressions.

Outline:
-------

Concepts and definitions:
Stages of the compilation process:
- **Front end:** lexical analysis (tokenizing), syntax analysis (parsing), semantic analysis (+type checking, intermediate code generation and optimization)
- **Back end:** machine code generation, machine code optimization.

Symbol table is used to collect information about variables (objects) related to the program, error handling is an additional module to help with errors.

Example: compilation of an arithmetical expression
(tokenizing, parsing, type checking, intermediate code)

The role of the Lexical Analyzer and its relation to other components of the compilation process.

Programming languages require precise definitions, and the best situation is when there is a standard defined and controlled by some interest group.

A complete description of a language is typically huge. For example, an ISO working draft for C++ from 2010 is 1325 pages long: [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf)

Lexical structure as an element of the programming language definition.
- lexical conventions
- groups of tokens
- token as a pair <token name, token value> (token value is optional)

A lexeme: the string that represents an instance of a token
A pattern: The set of all possible lexemes that can represent a token instance.

Every programming language provides lexical conventions and definitions of tokens. In C++: identifiers, keywords, literals, operators, punctuators and separators.
Keywords in C++ (cannot be used as identifiers):
alignof decltype goto reinterpret_cast try
asm default if return typedef
auto delete inline short typeid
debug double long sizeof union
case dynamic_cast mutable static unsigned
catch else namespace static_assert using
class enum new static_cast virtual
cast16_t explicit noexcept struct void
cast32_t export nullptr switch volatile
class extern operator template wchar_t
cast false private this while
constexpr float protected thread_local
cast_cast for public throw
continue friend register true

What is a sequence of tokens for the following fragment of C++ program?

```cpp
for ( int i = Dgts - 1; j > - 1; j-- ) //start sorting
{
  while (!container.empty()) //the rest of code removed
```
How to describe patterns? How to recognize if a string matches the pattern?

For simple languages, a narrative description of the pattern may be concise, clear and simple. For example, define binary strings corresponding to even numbers (numbers divisible by 2). Does the language of numbers divisible by 3 seem to be more complex than the language of numbers divisible by 2? Try to describe the set of binary strings corresponding to numbers divisible by 3. Is it easy? Is it possible?

Theory:
- Regular Expression
- Nondeterministic Finite Automata
- Deterministic Finite Automata

A recursive definition of a regular expression: (a finite alphabet is given)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>the language containing the empty string $L(\epsilon) = {\epsilon}$</td>
</tr>
<tr>
<td>$a$</td>
<td>the language containing symbol 'a': $L(a) = {a}$</td>
</tr>
<tr>
<td>$rs$</td>
<td>concatenation: $L(rs) = \text{the language of strings alpha beta with alpha in } L(r) \text{ and beta in } L(s)$</td>
</tr>
<tr>
<td>$r</td>
<td>s$</td>
</tr>
<tr>
<td>$r^*$</td>
<td>Kleene's closure – $L(r^*) = \text{the union of } {\epsilon} \text{ and } L(r)L(r)...L(r) \text{ for any finite number of concatenations}$</td>
</tr>
<tr>
<td>$r+$</td>
<td>like Kleene's closure but without ${\epsilon} – r \mid L(r^*)$</td>
</tr>
<tr>
<td>$[0-9]$</td>
<td>0</td>
</tr>
</tbody>
</table>

Examples:
Examples of regular expressions and regular definitions.

Let: $N 

\begin{align*}
\text{digit} & \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \\
\text{digits} & \rightarrow \text{digit}+ \\
\text{optional_fraction} & \rightarrow (\ . \ digits)? \\
\text{optional_exponent} & \rightarrow (E(+-)?\text{digits})? \\
\text{Num} & \rightarrow \text{digits}\text{optional_fraction}\text{optional_exponent}
\end{align*}

It will be useful to consider abstract syntax trees (AST). An abstract syntax tree represents a structure of an expression with leaves corresponding to operands and nodes corresponding to operators. The concept of AST can be applied to programming languages, and not only expressions. Each node of the AST in that case is labeled with a construct, such as if-then-else, occurring in the source code. Note that a regex can be rendered as an AST. Let’s look at a few examples.
1: # http://www.chemie.fu-berlin.de/chemnet/use/info/gawk/gawk_17.html
2: # This program, ‘dupword.awk’, scans through a file one line
3: # at a time, and looks for adjacent occurrences of the same word.
4: # It also saves the last word on a line (in the variable prev)
5: # for comparison with the first word on the next line.
6: # The first two statements make sure that the line is all
7: # lower-case, so that, for example, "The" and "the" compare equal
8: # to each other.
9: # The second statement removes all non-alphanumeric
10: # and non-whitespace characters from the line, so that
11: # punctuation does not affect the comparison either.
12: # This sometimes leads to reports of duplicated words that
13: # really are different, but this is unusual.
14: #
15: # dupword -- find duplicate words in text
16: # Arnold Robbins, arnold@gnu.ai.mit.edu, Public Domain
17: # December 1991
18: {
19:   $0 = tolower($0)
20:   gsub(/[A-Za-z0-9 \t]/, " ");
21:   if ($1 == prev)
22:     printf("\%s: \%d: duplicate \%s\n", 
23:            FILENAME, FNR, $1)
24:   for (i = 2; i <= NF; i++)
25:     if ($i == $(i-1))
26:       printf("\%s: \%d: duplicate \%s\n", 
27:              FILENAME, FNR, $i)
28:   prev = $NF
29: }
30: #
31:
1: n=10;  ; k=4;
2: i=1;  p=1;
3: t=1;  b=1;
4: //hello
5: /* test */
6: /*
7: multiple
8: lines
9: */
10: */
11: while (i <= k){ // line
12:    t = t * ((n-k) +i);
13:    b = b * i;
14:    i = i + 1;
15: }
16: p = t / b;
17: print p;
18: 
19: 

Tokenized CALC3 program

Tokenized Java

2: public static void insertionsort(int[] numbers) {
3:     for (int i = 0; i < numbers.length; i++) {
4:         int copyNumber = numbers[i];
5:         int j = i;
6:         while (j > 0 && copyNumber < numbers[j-1]) {
7:             numbers[j] = numbers[j-1];
8:             j--;
9:         }
10:         numbers[j] = copyNumber;
11:     }
12: }
13: 
14:
Small exercise (abstract trees, regular expression)

A) What is a sequence of tokens for the following fragment of C++ program?

```cpp
for ( int i = Dgts - 1; j > - 1; j-- ) //start sorting
  { while (!container.empty()) //the rest of code removed
    }
```

A list of tokens:

B) Informal definition of a well-formed e-mail address

http://www.faqs.org/rfcs/rfc2822.html

C) Regular expression for binary strings corresponding to odd numbers? Even numbers? Numbers divisible by 4?

D) Convert the following arithmetical expressions into their postfix form:
   a) 2* 22/7 * (R + a)

E) (Zeller's eternal calendar) – convert to an Abstract Syntax Tree

```
day-of-week = K + floor(13 * (m + 1)/5) + D + floor(D/4) + floor(C/4) – 2*C
```

where floor is a unary operator.

K – day of the month, m – month (3 = March, 4 = April, 5 = May, ..., 14 = February). D – the last two digits of the year, C – the century.

(check the formula for a possible typo).