Computer Language Engineering

• How to give instructions to a computer?  
  – Programming Languages.

• How to make the computer carry out the instructions efficiently? 
  – Compilers.

What is a Lexical Analyzer?

Source program text Tokens

• Examples of Token
  • Operators = + - ( { := == <=
  • Keywords if while for int double
  • Numeric literals 43 6.035 -3.6e10 0x13F3A
  • Character literals ‘a’ ‘c’ ‘\’
  • String literals “3.142” “Fall” “” = empty”

• Examples of non-token
  • White space space(‘ ’) tab(‘t’) end-of-line(‘\n’)
  • Comments /*this is not a token*/

Lexical Analyzer in Action

for ID(“var1”) eq_op Num(10) ID(“var1”) leq_op

• Partition input program text into sequence of tokens, attaching corresponding attributes.
  – E.g., C-lexeme “015” token NUM attribute 13
• Eliminate white space and comments
Syntax and Semantics of a programming language

- **Syntax**
  - What is the structure of the program?
  - Textual representation.
    - Formally defined using context-free grammars (Backus-Naur Formalism)
- **Semantics**
  - What is the meaning of a program?
  - Harder to give a mathematical definition.

---

**Example: A CFG for expressions**

- Simple arithmetic expressions with + and *
  - 8.2 + 35.6
  - 8.32 + 86 * 45.3
  - (6.001 + 6.004) * (6.035 * -(6.042 + 6.046))
- **Terminals (or tokens)**
  - `num` for all the numbers
  - `plus_op`, `minus_op`, `times_op`, `left_paren_op`, `right_paren_op`
- **What is the grammar for all possible expressions?**
Context-Free Grammars (CFGs)

- **Terminals**
  - Symbols for strings or tokens \{ `num`, (, ), +, *, - \}

- **Nonterminals**
  - Syntactic variables \{ `<expr>`, `<op>` \}

- **Start symbol**
  - A special nonterminal `<expr>`

- **Productions**
  - The manner in which terminals and nonterminals are combined to form strings.
  - A nonterminal in LHS and a string of terminals and nonterminals in RHS.
    - `<expr>` → `- <expr>`

---

English Language

- **Letters**: `a,b,c, ...`
- **Alphabet**: `{ a,b, ..., z } U { A, B, ..., Z }`
- **Tokens (Terminals)**: English words
- **Nonterminals**: `<Sentence>`, `<Subject>`, `<Clause>`, ...
- **Context-free aspect**: Replacing `<Noun>` by `<Proper Noun>` or `<Common Noun>`
- **Context-sensitive aspects**: Agreement among words w.r.t. number, gender, tense, etc.

---

Example Derivation

```
<expr>  ⇒  <expr> <op> <expr>
⇒  num <op> <expr>
⇒  num * <expr>
⇒  num * ( <expr> )
⇒  num * ( <expr> <op> <expr> )
⇒  num * ( num <op> <expr> )
⇒  num * ( num + <expr> )
⇒  num * ( num + num )
⇒  num * ( num + num )
```

Another Example Derivation

```
<expr>  ⇒  <expr> <op> <expr>
⇒  <expr> * <expr>
⇒  <expr> * ( <expr> )
⇒  <expr> * ( <expr> <op> <expr> )
⇒  <expr> * ( num <op> <expr> )
⇒  <expr> * ( num + <expr> )
⇒  <expr> * ( num + num )
⇒  num * ( num + num )
⇒  num * ( num + num )
```
Parse/Derivation Tree

- Graphical Representation of the parsed structure
- Shows the sequence of derivations performed
  - Internal nodes are non-terminals.
  - Leaves are terminals.
  - Each parent node is LHS and the children are RHS of a production.
- Abstracts the details of sequencing of the rule applications, but preserves decomposition of a non-terminal.

Expression –
One possible parse tree
num + num * num

Expression –
Another possible parse tree
num + num * num
Same string – Two distinct parse (derivation) trees

num + num * num

124 + (23.5 * 86) = 2145
(124 + 23.5) * 86 = 12685

Ambiguous Grammar

• Applying different derivation orders produces different parse trees.
  – This can lead to ambiguous/unexpected results.
  – Note that multiple derivations leading to the same parse tree is not a problem.
• A CFG is ambiguous if the same string can be associated with two distinct parse trees.
  – E.g., The expression grammar can be shown to be ambiguous using expressions containing binary infix operators and non-fully parenthesized expressions.

Removing Ambiguity

• Sometimes rewriting a grammar to reflect operator precedence with additional nonterminals will eliminate ambiguity.
  - * more binding than +.
  - && has precedence over ||.
• One can view the rewrite as “breaking the symmetry” through “stratification”.

Eliminating Ambiguity

<expr> → <term> + <expr>
<expr> → <term>
<term> → <unit> * <term>
<term> → <unit>
<unit> → num
<unit> → ( <expr> )

<expr> → <expr> <op> <expr>
<expr> → ( <expr> )
<expr> → num
<op> → +
<op> → *
Extended BNF (Backus Naur Formalism)

<expr> → <term> ( + <term> ) *
<term> → <unit> ( * <unit> ) *
<unit> → num | ( <expr> )

Expression Parser Fragment
(String => Parse Tree)

```java
Node expr() {
    // PRE: Expects lookahead token.
    // POST: Consume an Expression
    //        and update Lookahead token.
    Node temp = term();
    while ( inTok.ttype == '+' ) {
        inTok.nextToken();
        Node temp1 = term();
        temp = new OpNode(temp, '+', temp1);
    }
    return temp;
}
```

Arithmetic Expressions
(with variables)

<expr> → <expr> + <expr>
| <expr> * <expr>
| ( <expr> )
| <variable>
| <constant>

<variable> → x | y | z
<constant> → 0 | 1 | 2

Resolving Ambiguity in Expressions

- Different operators: precedence relation
- Same operator: associativity relation

Left Associative ((5-2)-3) | Right Associative (2**(3**4))
## C++ Operator Precedence and Associativity

<table>
<thead>
<tr>
<th>Level</th>
<th>Operator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>17R</td>
<td>::</td>
<td>global scope (unary)</td>
</tr>
<tr>
<td>17L</td>
<td>::</td>
<td>class scope (binary)</td>
</tr>
<tr>
<td>16L</td>
<td>-&gt; , .</td>
<td>member selectors</td>
</tr>
<tr>
<td>16L</td>
<td>[]</td>
<td>array index</td>
</tr>
<tr>
<td>16L</td>
<td>()</td>
<td>function call</td>
</tr>
<tr>
<td>16L</td>
<td>()</td>
<td>type construction</td>
</tr>
<tr>
<td>15R</td>
<td>sizeof</td>
<td>size in bytes</td>
</tr>
<tr>
<td>15R</td>
<td>++ , --</td>
<td>increment, decrement</td>
</tr>
<tr>
<td>15R</td>
<td>~</td>
<td>bitwise NOT</td>
</tr>
<tr>
<td>15R</td>
<td>!</td>
<td>logical NOT</td>
</tr>
<tr>
<td>15R</td>
<td>+ , -</td>
<td>unary minus, plus</td>
</tr>
<tr>
<td>15R</td>
<td>* , &amp;</td>
<td>dereference, address-of</td>
</tr>
<tr>
<td>15R</td>
<td>()</td>
<td>type conversion (cast)</td>
</tr>
<tr>
<td>14L</td>
<td>new , delete</td>
<td>free management</td>
</tr>
<tr>
<td>14L</td>
<td>.* , .*</td>
<td>member pointer select</td>
</tr>
<tr>
<td>13L</td>
<td>* , / , %</td>
<td>multiplicative operators</td>
</tr>
<tr>
<td>12L</td>
<td>+= , -=</td>
<td>arithmetic if</td>
</tr>
<tr>
<td>11L</td>
<td>&lt;&lt;= , &gt;&gt;=</td>
<td>bitwise shift</td>
</tr>
<tr>
<td>10L</td>
<td>&lt; , &lt;= , &gt; , &gt;=</td>
<td>relational operators</td>
</tr>
<tr>
<td>9L</td>
<td>== , !=</td>
<td>equality, inequality</td>
</tr>
<tr>
<td>8L</td>
<td>&amp; &amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>8L</td>
<td>^</td>
<td>bitwise XOR</td>
</tr>
<tr>
<td>8L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7L</td>
<td>&amp;&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td>7L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6L</td>
<td>? : ?</td>
<td>assignment operators</td>
</tr>
<tr>
<td>5L</td>
<td>,</td>
<td>comma operator</td>
</tr>
<tr>
<td>4L</td>
<td>,</td>
<td>comma operator</td>
</tr>
<tr>
<td>3L</td>
<td>,</td>
<td>comma operator</td>
</tr>
<tr>
<td>2L</td>
<td>,</td>
<td>comma operator</td>
</tr>
<tr>
<td>1L</td>
<td>,</td>
<td>comma operator</td>
</tr>
</tbody>
</table>