Programming Language Concepts
Lexical and Syntactictic Analysis

Janyl Jumadinova
24 January, 2017
Most Important Steps in Compilation

- Optional Preprocessing
- Lexical analysis (scanning)
- Syntax analysis (parsing)
- Semantic analysis
- Intermediate code generation
- Optimization (usually machine-independent)
- Final code generation
- Optional final optimization
Lexical Analysis

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;unsigned int&gt;</td>
</tr>
<tr>
<td>1</td>
<td>'('</td>
</tr>
<tr>
<td>2</td>
<td>']'</td>
</tr>
<tr>
<td>3</td>
<td>'+'</td>
</tr>
<tr>
<td>4</td>
<td>'-'</td>
</tr>
<tr>
<td>5</td>
<td>&quot;for&quot;</td>
</tr>
<tr>
<td>6</td>
<td>&quot;while&quot;</td>
</tr>
<tr>
<td>7</td>
<td>&lt;identifier&gt; (not reserved)</td>
</tr>
<tr>
<td>8</td>
<td>';</td>
</tr>
<tr>
<td>9</td>
<td>'='</td>
</tr>
<tr>
<td>10</td>
<td>'=='</td>
</tr>
<tr>
<td>11</td>
<td>'&lt;='</td>
</tr>
<tr>
<td>12</td>
<td>'&gt;='</td>
</tr>
<tr>
<td>13</td>
<td>&lt;string literal&gt;</td>
</tr>
<tr>
<td>14</td>
<td>'&lt;'</td>
</tr>
</tbody>
</table>

A token is any component of a program that is generally treated as an indivisible piece, e.g., a variable name, an operator such as <=, a punctuation mark such as a semicolon, a string constant, etc.
For each token type, give a description:

- either a literal string (e.g., “≤” or “while” to describe an operator or reserved word),
Lexical Analysis

For each token type, give a description:

- either a literal string (e.g., “≤” or “while” to describe an operator or reserved word),
- or a <rule> (e.g., the rule <unsigned int> might stand for “a sequence of one or more digits”; the rule <identifier> might stand for “a letter followed by a sequence of zero or more letters or digits.”)
Lexical analysis produces a “token stream” in which the program is reduced to a sequence of token types, each with its identifying number and the actual string (in the program) corresponding to it.
Lexical Analysis

// see if 3 occurs
while x <= 10
  a = x+1
while (a == 3)
  found = 1
  a = f(x)

Program

Stream of Tokens
The syntax of a language is described by a grammar that specifies the legal combinations of tokens.
Syntactic Analysis

- The syntax of a language is described by a **grammar** that specifies the legal combinations of tokens.
- Grammars are often specified in BNF notation ("Backus Naur Form"): 
Syntactic Analysis

- The syntax of a language is described by a **grammar** that specifies the legal combinations of tokens.
- Grammars are often specified in BNF notation ("Backus Naur Form"): 
  
  \[
  \text{<item1>} ::= \text{valid replacements for <item1>}
  \]
  
  \[
  \text{<item2>} ::= \text{valid replacements for <item2>}
  \]
Example: an expression can be either a simple variable identifier; an integer; or an expression, followed by an operator, followed by another expression:

\[<expr> ::= <id> \mid <int> \mid <expr> \ <op> \ <expr>\]

Alternative notations:

\[
expr \rightarrow id \mid int \mid expr \ op \ expr
\]

\[
expr ::= id \mid int \mid expr \{op \ expr\}^*
\]

The symbol “\(|\)” means “or”

The “\(\{\ldots\}\)\)” means “zero or more repetitions of the items in \(\{\ldots\}\)”

Classic BNF notation

The book uses this notation (but as three separate rules)
Grammars (Context-free Grammars)

- Collection of VARIABLES (things that can be replaced by other things), also called NON-TERMINALS.
Grammars (Context-free Grammars)

- Collection of VARIABLES (things that can be replaced by other things), also called NON-TERMINALS.
- Collection of TERMINALS ("constants", strings that can’t be replaced)
Grammars (Context-free Grammars)

- Collection of VARIABLES (things that can be replaced by other things), also called NON-TERMINALS.
- Collection of TERMINALS ("constants", strings that can’t be replaced)
- One special variable called the START SYMBOL.
Grammars (Context-free Gramars)

- Collection of VARIABLES (things that can be replaced by other things), also called NON-TERMINALS.
- Collection of TERMINALS ("constants", strings that can’t be replaced)
- One special variable called the START SYMBOL.
- Collection of RULES, also called PRODUCTIONS.
Grammars (Context-free Grammars)

- Collection of VARIABLES (things that can be replaced by other things), also called NON-TERMINALS.
- Collection of TERMINALS ("constants", strings that can’t be replaced)
- One special variable called the START SYMBOL.
- Collection of RULES, also called PRODUCTIONS.

variable → rule1 | rule2 | rule3 | ... 
You can also write each rule on a separate line (as in the book)
A, B, and C are non-terminals.
0, 1, and 2 are terminals.
The start symbol is A.
The rules are:

- \( A \rightarrow 0A|1C|2B|0 \)
- \( B \rightarrow 0B|1A|2C|1 \)
- \( C \rightarrow 0C|1B|2A|2 \)
Grammars (Context-free Grammars)

Grammar

A, B, and C are non-terminals.
0, 1, and 2 are terminals.
The start symbol is A.
The rules are:

- A → 0A | 1C | 2B | 0
- B → 0B | 1A | 2C | 1
- C → 0C | 1B | 2A | 2

Can 2011020 can be parsed?
Grammars (Context-free Grammars)

**Grammar**

A, B, and C are non-terminals.
0, 1, and 2 are terminals.
The start symbol is A, the rules are:

- \( A \rightarrow 0A|1C|2B|0 \)
- \( B \rightarrow 0B|1A|2C|1 \)
- \( C \rightarrow 0C|1B|2A|2 \)

Can 1112202 can be parsed?
Can 00102 can be parsed?
Can 2120 can be parsed?
Grammars (Context-free Grammars)

Grammar

A, B, and C are non-terminals.
0, 1, and 2 are terminals.
The start symbol is A, the rules are:

- A → 0A|1C|2B|0
- B → 0B|1A|2C|1
- C → 0C|1B|2A|2

Can 1112202 can be parsed?
Can 00102 can be parsed?
Can 2120 can be parsed?
Syntactic Analysis

\[
\text{prog} \rightarrow \{\text{statement}\}^+ \\
\text{statement} \rightarrow \text{assignment} \mid \text{loop} \mid \text{io} \\
\text{assignment} \rightarrow \text{id} = \text{expression} \\
\text{loop} \rightarrow \text{while ( expression ) prog}
\]

“A program is one or more statements.”
“A statement is an assignment, a loop, or an input/output command.”
“An assignment is an identifier, followed by “=” , followed by an expression.”

The “\{\ldots\}^+” means “one or more repetitions of the items in \{\ldots\}”

In this example, “=”, “while”, “(" and ")” are terminals
The process of verifying that a token stream represents a valid application of the rules is called **parsing**.

Using the BNF rules we can construct a parse tree:
Sample Parse Tree (portion)

```
while (a == 3)
    found = 1
```
Sample Parse Tree (failed)

Parse halts after "while" -- unable to match the "(" in the rule with a "(" in the input. An error is reported by the compiler.
Grammar for Java (version 8)

- Overview of notation used:
  https://docs.oracle.com/javase/specs/jls/se8/html/jls-2.html

- The full syntax grammar:
Compiling

So far, we have looked at:

- the scanner (lexical analysis)—tokenizes input
- the parser (syntactic analysis)—validates structure

EXAMPLE:
In Java, `int i, i, i;` has the right structure for a declaration, but it's not legal to redeclare `i` within the same block of code.
So far, we have looked at:

- the scanner (lexical analysis)–tokenizes input
- the parser (syntactic analysis)–validates structure
- Next, we do **semantic analysis**: is the program meaningful?

EXAMPLE:

In Java,

```java
int i, i, i;
```

has the right structure for a declaration, but it’s not legal to redeclare `i` within the same block of code.
So far, we have looked at:

- the scanner (lexical analysis)–tokenizes input
- the parser (syntactic analysis)–validates structure
- Next, we do **semantic analysis**: is the program meaningful?

**EXAMPLE:**

In Java,

```java
int i, i, i;
```

has the right structure for a declaration, but it’s not legal to redeclare `i` within the same block of code.
During lexical analysis and parsing, as we process tokens we gather the user-defined names into a **symbol table**.
During lexical analysis and parsing, as we process tokens we gather the user-defined names into a **symbol table**.

Symbol table contains information such as:

- where the symbol first appeared (usually in a declaration)
- whether it has an initial value (parsing will tell us this)
- what its type is (parsing tells us), etc.
As the parser encounters names, it looks them up to see if they are already declared; if not, it creates a table entry. (Some names are pre-declared as part of the language.)
Symbol Table

Line# Program source

... ...

20 int i = 10;
21 double j = 3;
22 int k;
23 k = i*j;

<table>
<thead>
<tr>
<th>Entry</th>
<th>Symbol</th>
<th>Declared at</th>
<th>Type</th>
<th>Init Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>&lt;library&gt;.103</td>
<td>--</td>
<td>(TYPE)</td>
</tr>
<tr>
<td>6</td>
<td>double</td>
<td>&lt;library&gt;.142</td>
<td>--</td>
<td>(TYPE)</td>
</tr>
<tr>
<td>7</td>
<td>i</td>
<td>&lt;Lab3&gt;.20</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>j</td>
<td>&lt;Lab3&gt;.21</td>
<td>6</td>
<td>3.0</td>
</tr>
<tr>
<td>9</td>
<td>k</td>
<td>&lt;Lab3&gt;.22</td>
<td>5</td>
<td>uninit</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

test.java:23: error: possible loss of precision:
k = i*j;
  ^
required: int, found: double
Symbol Table

<table>
<thead>
<tr>
<th>Entry</th>
<th>Symbol</th>
<th>Declared at</th>
<th>Type</th>
<th>Init Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>&lt;library&gt;.103</td>
<td>--</td>
<td>(TYPE)</td>
</tr>
<tr>
<td>6</td>
<td>double</td>
<td>&lt;library&gt;.142</td>
<td>--</td>
<td>(TYPE)</td>
</tr>
<tr>
<td>7</td>
<td>i</td>
<td>&lt;Lab3&gt;.20</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>j</td>
<td>&lt;Lab3&gt;.21</td>
<td>6</td>
<td>3.0</td>
</tr>
<tr>
<td>9</td>
<td>k</td>
<td>&lt;Lab3&gt;.22</td>
<td>5</td>
<td>uninit</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

20 int i = 10;
21 double j = 3;
22 int k;
23 k = i*j;
24 int j;

test.java:24: error: variable j is already defined in method main (String[])
Symbol Table

Line#   Program source
20     int i = 10;
21     double j = 3;
22     int k;
23     k = i*j;
24     int j;
25     m = 17;

    test.java:25: error: cannot find symbol
    m = 17;
    ^

    symbol:   variable m
During this phase, the parsed program is converted into a simpler, step-by-step description in some intermediate language.
Intermediate Code Generation

- During this phase, the parsed program is converted into a simpler, step-by-step description in some intermediate language.
- Intermediate language may exist only as an internal representation within the compiler—it does not need be an “actual” language.
- A simple example is something called “three-address code.”
In “Three-address code”, everything is reduced to simple operations on two values, with the result stored in a (possibly temporary) variable.

**Source code:**

```c
double f = 3 + i * 7;
```

**Three-address code:**

```
temp1 = i
temp2 = 7
temp1 = temp1 * temp2
temp2 = 3
temp1 = temp2 + temp1
f = int-to-float(temp1)
```
Why Intermediate Code Generation?
Why Intermediate Code Generation?

Simple representation, easy to manipulate, close to assembly/machine language. E.g.,

```
temp1 = x
temp2 = 10
```

```
add $t0, $t0, $t1
lw  $t0, x
li  $t1, 10
```

Three-address code

MIPS assembly language
Optimization

Many levels, from very basic:

\[
\begin{align*}
\text{temp1} &= \text{temp1} + 1 \\
\text{temp1} &= \text{temp1} + x \\
\text{temp1} &= \text{temp1} + 5
\end{align*}
\]

\[
\begin{align*}
\text{temp1} &= \text{temp1} + x \\
\text{temp1} &= \text{temp1} + 6
\end{align*}
\]

“constant folding”
... to more complex:
\[
\begin{align*}
te\text{mp1} &= 1 \\
te\text{mp2} &= x \\
te\text{mp3} &= te\text{mp2}+9 \\
te\text{mp2} &= te\text{mp2}*te\text{mp3} \\
te\text{mp1} &= 8 \\
te\text{mp2} &= x \\
te\text{mp3} &= te\text{mp2}+9 \\
te\text{mp2} &= te\text{mp2}*te\text{mp3} \\
te\text{mp1} &= 8
\end{align*}
\]
“dead code removal”
Here is a list of the many, many kinds of optimizations:
http://www.compileroptimizations.com/

(The examples show the effects on source code, but the optimizations are usually made on the intermediate code.)
It is usually very straightforward to generate machine instructions from intermediate code, since the intermediate code is simple.

Some further machine-specific optimizations may take place during or after this stage.
The steps in compilation don’t need to be done in whole phases, one after the other, but can be “pipelined”:

- `int count = 1;`
  create tokens, pass to parser, generate some intermediate code

- `j = j + count;`
  create tokens, pass to parser, generate some intermediate code

... etc. ...