Compiler Construction
Winter 2020

Recitation 9:
Lexical Analysis

Yotam Feldman

Based on slides by Guy Golan-Gueta and the Technion compilers class’ staff
Lexing & Parsing

Java code → Lexical analysis → Token Sequence → Parsing → Abstract Syntax Tree (AST)

Today

Next week
Lexical Analysis

- Tokens are strings with an “identified meaning”

<table>
<thead>
<tr>
<th>src string</th>
<th>tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = size + 29 ; \n</td>
<td>id(“x”) assign id(“size”) plus num(29) sep</td>
</tr>
</tbody>
</table>

No associated token!
Lexical Analysis

- Tokens are strings with an “identified meaning”
- The **terminals** in the grammar

```
src string →
tokens →

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>=</td>
<td>size</td>
<td>+</td>
<td>29</td>
<td>;</td>
<td>\n</td>
</tr>
<tr>
<td>id(“x”)</td>
<td>assign</td>
<td>id(“size”)</td>
<td>plus</td>
<td>num(29)</td>
<td>sep</td>
<td></td>
</tr>
</tbody>
</table>
```

```
S → id assign id plus num sep

S → ID WS ‘=’ WS ID WS ‘+’ WS NUM WS ‘;’ WS
WS → ‘\n’ WS | ‘ ’ WS | ...
ID → ‘a’ ID | ‘b’ ID | ...
NUM → ‘1’ NUM0 | ‘2’ NUM0 | ...
```
Lexical Analysis

• Tokens are strings with an “identified meaning”
• The **terminals** in the grammar

<table>
<thead>
<tr>
<th>src string</th>
<th>tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = size + 29 ; \n</td>
<td>id(“x”) assign id(“size”) plus num(29) sep</td>
</tr>
</tbody>
</table>

**No associated token!**

• The compiler “**forgets**” the original strings
Token nextToken() {
    char c;
    while (true) {
        c = getchar();
        switch (c) {
            case `
            case `;`: return Semicolon;
            case `+`:  c = getchar();
                switch (c) {
                    case `+`: return PlusPlus;
                    case `=`: return PlusEqual;
                    default: ungetc(c);
                        return Plus;
                } ... 
        }
    }
}
Lexical Analysis Generator

- Off the shelf lexical analysis generator
- Input
  - scanner specification file
- Output
  - Lexical analyzer written in Java

```
MiniJava.lex → JFlex → Lexer.java → javac → Lexical analyzer
```

Minijava text → tokens
JFlex

• Simple
• Good for reuse
• Easy to understand
• Many developers and users debugged the generators

"+" { return new symbol (sym.PLUS); }
"boolean" { return new symbol (sym.BOOLEAN); }
"int" { return new symbol (sym.INT); }
"null" {return new symbol (sym.NULL);}
"while" {return new symbol (sym.WHILE);}
"=" {return new symbol (sym.ASSIGN);}
...

JFlex Spec File

User code
  Copied directly to Java file

%%

JFlex directives
  Define macros, state names

%%

Lexical analysis rules
  How to break input to tokens
  Action when token matched

DIGIT= [0-9]
LETTER= [a-zA-Z]
YYINITIAL

{LETTER}
  (({LETTER}|{DIGIT}))*
## Regular Expressions’ Notation

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>any character except the newline</td>
<td></td>
</tr>
<tr>
<td>&quot;...&quot;</td>
<td>string</td>
<td>&quot;int&quot;</td>
</tr>
<tr>
<td>a–b</td>
<td>range of characters</td>
<td>[0-9]</td>
</tr>
<tr>
<td>[...]</td>
<td>class of characters - any one character enclosed in brackets</td>
<td>[xyz] [a-zA-Z_]</td>
</tr>
<tr>
<td>[^...]</td>
<td>negated class – any one not enclosed in brackets</td>
<td>[^\t\r\n]</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>match a or b</td>
</tr>
<tr>
<td>*</td>
<td>zero or more repetitions</td>
<td>.*</td>
</tr>
<tr>
<td>+</td>
<td>one or more repetitions</td>
<td>[0-9]+</td>
</tr>
<tr>
<td>?</td>
<td>zero or one repetitions</td>
<td></td>
</tr>
<tr>
<td>(...)</td>
<td>grouping within regular expressions</td>
<td></td>
</tr>
<tr>
<td>{name}</td>
<td>macro expansion</td>
<td></td>
</tr>
</tbody>
</table>
Rules – Action

• Action
  – Java code
  – Can use special methods and vars
    • yyline
    • yytext()
  – Return a token when found

<YYINITIAL> {ID}
  {return symbol(sym.ID, new String(yytext()));}
  – “Eat” chars for non tokens (whitespaces, comments)
Rules – “Eating”

• Whitespaces:

```c
<YYINITIAL> " " | "\t" | "\r" | "\n"
    { /* just skip what was found, do nothing */ }  
```

• Single line comments?
• Multiline comments?
<YYINITIAL> "//" { yybegin(COMMENTS); }

<COMMENTS> [^\n] { }

<COMMENTS> [\n] { yybegin(YYINITIAL); }

Rules – State
import java_cup.runtime.Symbol;

%%
%cup
%
{
    private int lineCounter = 0;
}%

%eofval{
    System.out.println("line number=" + lineCounter);
    return new Symbol(sym.EOF);
}%eofval

NEWLINE=\n
%%
<YYINITIAL>{NEWLINE} {
    lineCounter++;
}
<YYINITIAL>[^{NEWLINE}] { }
Lexer Conflicts

• When the same input matches several tokenizations.

• **Example**: Suppose that the following tokens are defined:
  – for: “for”
  – id: [a-z]+

• Conflicts:

<table>
<thead>
<tr>
<th>Input</th>
<th>Possible tokenizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>id(“a”)</td>
</tr>
<tr>
<td></td>
<td>id(“ab”)</td>
</tr>
<tr>
<td></td>
<td>id(“abc”)</td>
</tr>
<tr>
<td>ford</td>
<td>id(“f”)</td>
</tr>
<tr>
<td></td>
<td>id(“fo”)</td>
</tr>
<tr>
<td></td>
<td>id(“for”)</td>
</tr>
<tr>
<td></td>
<td>id(“ford”)</td>
</tr>
<tr>
<td></td>
<td>for</td>
</tr>
<tr>
<td>for</td>
<td>id(“for”)</td>
</tr>
<tr>
<td></td>
<td>for</td>
</tr>
</tbody>
</table>
Solving Conflicts

• Maximal munch
  – Add more characters as long as still match some token

• Priorities
  – If still conflicted, the first matching rule in the lexer specification

Can such conflict resolving be performed directly in the parser?

<table>
<thead>
<tr>
<th>Input</th>
<th>Possible tokenizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>id(“a”)</td>
</tr>
<tr>
<td></td>
<td>id(“ab”)</td>
</tr>
<tr>
<td></td>
<td>id(“abc”)</td>
</tr>
<tr>
<td>ford</td>
<td>id(“f”)</td>
</tr>
<tr>
<td></td>
<td>id(“fo”)</td>
</tr>
<tr>
<td></td>
<td>id(“for”)</td>
</tr>
<tr>
<td></td>
<td>id(“ford”)</td>
</tr>
<tr>
<td></td>
<td>for</td>
</tr>
<tr>
<td>for</td>
<td>id(“for”)</td>
</tr>
<tr>
<td></td>
<td>for</td>
</tr>
</tbody>
</table>
Lexer Errors

• When the current input sequence cannot be matched to a token
• One form of syntactic error
  – More in the parser
Regex Debugger

- https://regex101.com/
Demo: Lexer for Simple Expressions
Summary

• Lexical analysis
• JFlex
• Maximal munch
• Rule precedence