Lexical Analysis - 3

- **Handling errors**
- **JLex - automating lexical analysis**
  - **Project 2**
  - **Example of input to JLex**
- **Review of context-free grammars**
- **Intro to bottom up parsing (shift-reduce)**

Error Handling

- **Panic mode recovery**
  - **Flush to the next well-formed token, if can**
  - **Often go to next statement delimeter (;)**
  - **When want to match something in case the listed patterns don’t match, don’t use .* because this pattern will always match the longest string in the input!**

Error Handling

- **Sophisticated alternatives (from spelling correction technology)**
  - **Delete an extra character and rescan**
  - **Insert a missing character**
  - **Replace incorrect character by correct character**
  - **Transpose 2 adjacent characters**

Error Handling

- **Empirical evidence**
  - **60% punctuation errors (;)**
  - **20% operator/operand errors (= instead of :=)**
  - **15% keyword errors (e.g., missing "end")**
- **Should report site where error is detected**
  - **Line number and character where error is recognized**
- **Stay simple in handling**
- **Sometimes put in error productions to catch likely errors and provide better messages**

JLex - a Scanner Generator

- **What is it?**
  - **A program that produces a Java program from a lexical specification**
    - **User defines each token and actions to be taken when recognized**
    - **Program produced can communicate with parser**
- **JLex is written to be like Lex (the original scanner generator for C - written in C)**
- **Warning: the error messages generated are pretty confusing!**

Using JLex

- **First, run *x.lex* file through Jlex to produce *x.lex.java*, a scanner for the tokens described in *x.lex***
- **Second, compile *x.lex.java* to byte code**
- **Third, write a data file with examples of the tokens in it**
- **Fourth, run *Parse.Main main* method that creates new *Yylex* object that can respond to *nextToken()* message and return next token**

JLex input files

- **Section 1 contains package declarations, any import statements and classes that may be used by the Java code in the rest of the file**
- **Section 2 contains RE abbreviations, state declarations, and directives to JLex (see manual), including Java code to be included in the scanner (*Yylex()*)**
- **Section 3 contains token REs and their corresponding actions**

Example

See Appel, Ch2; Also *myTiger.lex*

```
package Parse;
import ErrorMsg.ErrorMsg;
%%
%implements Lexer
%function nextToken
%type java_cup.runtime.Symbol
%char
%{
private void newline() {
  errorMsg.newline(yychar);
}
private java_cup.runtime.Symbol tok(int kind, Object value) {
   return new java_cup.runtime.Symbol(kind, yychar, yychar+yylength(), value);
…
```

Section 1: package defs and imports

Section 2: directives to Jlex

{%Java code to be included in scanner %} that is, in the Yylex class, unless it is a class itself

kind: token type; beginning and ending char position of token, semantic value

Example

See Appel, Ch2;
Also *myTiger.lex*

```
Yylex(java.io.InputStream s,ErrorMsg e) {
  this(s);
  errorMsg=e;
}
private void err(int pos, String s) {
  errorMsg.error(pos,s);
}
private void err(String s) {
  err(yychar,s);}
private ErrorMsg errorMsg;
%}
%eofval{
     {  return tok(sym.EOF, null);  }
%eofval}
```

definition of Yylex constructor

shows how to define an overloaded function, err
Java distinguishes between them by parameter types

end of Java code to be included

another Jlex directive; defines actions to be taken at end of input

Example

See Appel, Ch2; Also *myTiger.lex*

%% *section with REs and actions*

```
" "    { }
\n     {newline(); }
","    {return tok(sym.COMMA, null); }
[0-9]+  {return tok(sym.INT, new Integer(yytext())); }
[a-zA-Z]([a-zA-Z]|[0-9])* {return tok(sym.ID, yytext()); }
.  {System.out.println (yychar +" illegal character");}
```

sym class contains defined values for token types

Integer wrapper class

error match when all other patterns fail

NOTE: errors are usually traceable to some mistake in your REs or their associated actions; For example, one error we had was to put {} rather than { } for an empty action (the second set of braces is separated by a blank). JLex is picky so be fastidious!

JLex

- **To use JLex, you will have to augment your CLASSPATH to access some packages (see project 2 webpage)**
- **JLex uses the *Symbol* class which is defined in the *java_cup.runtime* package**

Class Symbol

```
int sym; /*token type*/
int left, right; /*position in source file*/
Object value; /*semantic value*/
Symbol(int s,int l, int r, Object v){ /*constructor*/
    sym=s; left=l; right=r; value=v;}
```

JLex

- ***yytext()*** **always returns the string matched by the regular expression**
- ***yychar*** **returns the beginning position of that string (remember the 1st position is 0)**
- **You can use *System.out.println* statements liberally in your actions to try to see where your errors are occurring.**

makefile

```
JFLAGS=-g
Parse/Main.class: Parse/*.java Parse/Yylex.java
   javac ${JFLAGS} Parse/*.java

Parse/Yylex.java: Parse/Tiger.lex
   cd Parse; java JLex.Main Tiger.lex;
    mv Tiger.lex.java Yylex.java

ErrorMsg/ErrorMsg.class:  ErrorMsg/*.java
   javac ${JFLAGS} ErrorMsg/*.java

clean:
   rm Parse/*.class ErrorMsg/*.class Parse/Yylex.java
```

shows dependences between program parts
helps to build large systems

dependences shown
a:b a depends on b

main() in Parse.Main class

```
public static void main(String argv[]) throws java.io.IOException {
    String filename = argv[0];
    ErrorMsg errorMsg = new ErrorMsg(filename);
    java.io.InputStream inp=new java.io.FileInputStream(filename);
    Lexer lexer = new Ylex(inp,errorMsg);  create new scanner as Ylex object
    java_cup.runtime.Symbol tok;           with its own input stream and error
                                           handler
    do {
      tok=lexer.nextToken();
      System.out.println(symnames[tok.sym] + " " + tok.left);
    } while (tok.sym != sym.EOF);

    inp.close();
 }
```

New Java Features

- **Interfaces (e.g., *Lexer*)**
- **Envelope classes (e.g., *Integer*)**
  - **Needed because everything in Java is an object**
  - **A consistent way of integrating primitive types in an OOPL**
  - **A way of doing input cleanly, so every value read on input is a *String* which is then converted to, for example, *Integer* objects that then can have their *int* values accessed.**
  - **Envelope classes: *Integer*, *Double*, *Character*, *Boolean***

Integer Class

- **Interface (partial)**

  ***Integer (int value)*; //creates an Integer object**

  ***int IntValue()*;//obtains int value from Integer receiver**

  ***Integer valueOf(String s)*;//class method which converts a String object to an Integer object**

```
Integer Iobj = new Integer (5);
System.out.println(Iobj.intValue());

String item = nextToken();
(Integer.valueOf(item.trim())).intValue();
```

Integer method

String method

class method, class Integer

Class Methods and Variables

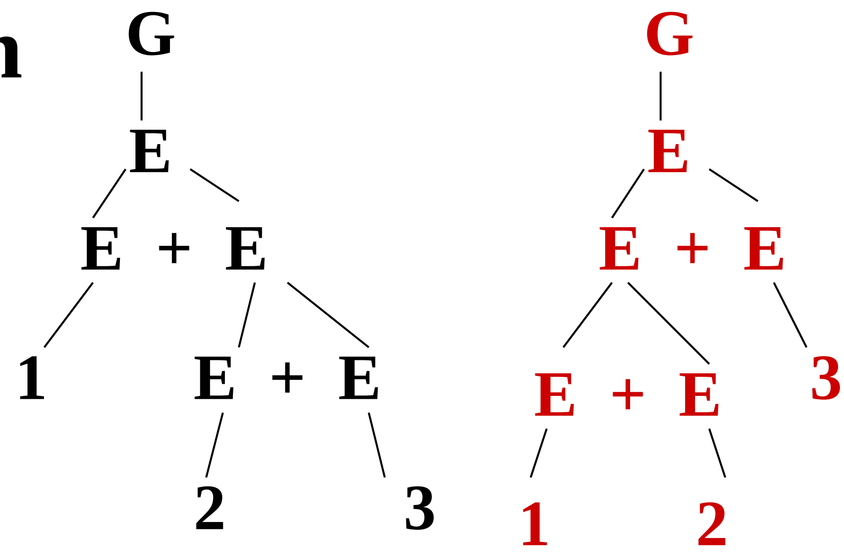
- ***Class methods*** **are something like utility procedures requiring no receiver object**
  - **Invoked by <class-name>. <method-name>**
  - **Defined by *static* keyword**
  - **Often used to change values of class variables**
- ***Class variables*** **are shared by all objects in the class (i.e., *static*)**
  - **Values can be changed only by class methods**
  - **Only one copy of each class variable for all objects in the class**

Context-free Grammars

- **Grammar consists of**
  - **Terminal symbols**
  - **Nonterminal symbols**
  - **Rules for forming nonterminals from sequences of terminals and nonterminals**
  - **Distinguished symbol**
- **If rules are of form *nonterminal* alone on left hand side, grammar is *context-free***

Definitions to Review

- **Canonical derivation**
- **Parse tree**
- **Ambiguity**
- **Precedence**



G ::= E
E ::= E + E | E * E | F
F ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
G-->E-->E+E-->1+E-->1+E+E-->1+2+E-->1+2+3
G-->E-->E+E-->E+E+E-->1+E+E-->1+2+E-->1+2+3

Parsing

- **Is reverse of doing a derivation**
- **By looking at the terminal string, effectively try to build the parse tree from the bottom up**
- **Finding which sequences of terminals and nonterminals form the right hand side of production and *reducing* them to the left hand side nonterminal**

Shift-reduce Parsing

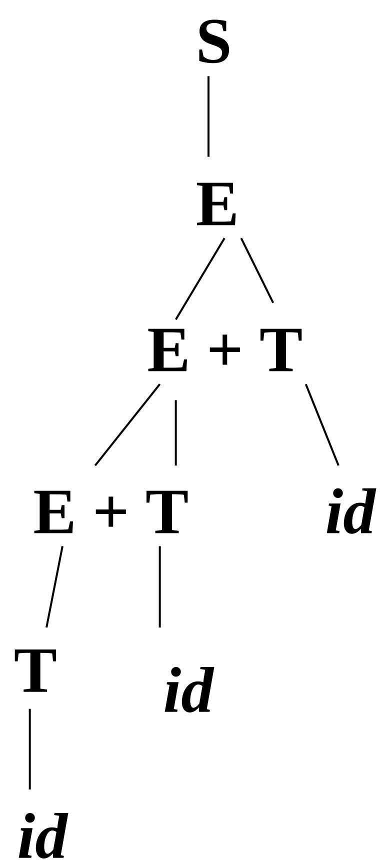
- ***Handle*- substring which is right hand side of some production; corresponds to the last expansion in a *rightmost derivation***
- **Replacement of handle by its corresponding nonterminal left hand side, results in reduction to the distinguished nonterminal by a *reverse rightmost derivation***
- **Parse works by shifting symbols onto the stack until have *handle* on top; then reduce; then continue**

Example

S → E		(1)
E → E + T		(2)
E → T		(3)
T → *id*		(4)

Rightmost derivation of a+b+c, handles in red

S → E
→ E + T
→ E + ***id***
→ E + T + ***id***
→ E + ***id*** + ***id***
→ **T** + ***id*** + ***id***
→ ***id*** + ***id*** + ***id***



Shift-Reduce Parser, Example

Actions: shift, reduce, accept, error

Stack	Input	Action
\$	id1 + id2 + id3 \$	shift
\$ id1	+ id2 + id3 \$	reduce (4)
\$ T	+ id2 + id3 \$	reduce (3)
\$ E	+ id2 + id3 \$	shift
\$ E +	id2 + id3 \$	shift
\$ E + id2	+ id3 \$	reduce(4)
\$ E + T	+ id3 \$	reduce (2)
\$ E	+ id3 \$	shift
\$ E +	id3 \$	shift
\$ E + id3	\$	reduce (4)
\$ E + T	\$	reduce(2)
\$ E	\$	reduce (1)
\$ S	\$	accept

S → E (1)
E → E + T (2)
E → T (3)
T → *id* (4)

Possible Problems

- **Can get into conflicts where one rule implies *shift* while another implies *reduce***

S → if E then S | if E then S else S

On stack: if E then S

Input: else

Should *shift* trying for 2nd rule or *reduce* by first rule?

Possible Problems

- **Can have two grammar rules with same right hand side which leads to *reduce-reduce* conflicts**

  A → α and B → α both in grammar

  When α on stack, how know which production choose? That is, whether to reduce to A or B?