Introduction to Compiler Design

Lesson 10:

Parsers – Java CUP

Translating Lists

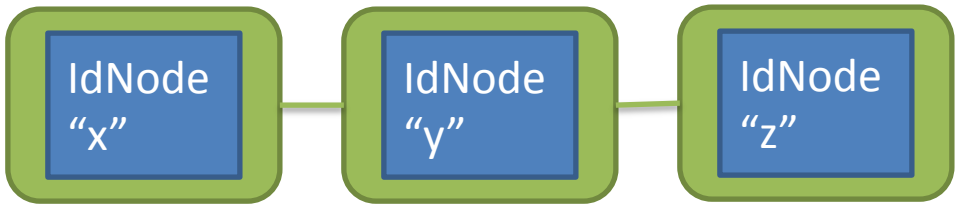
CFG

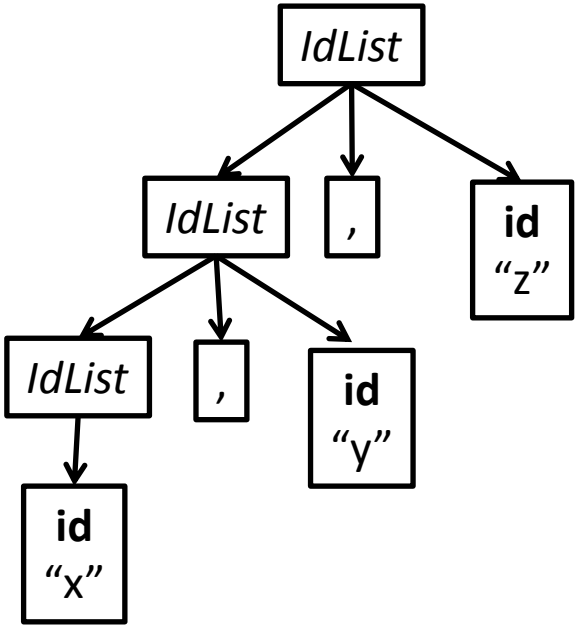
```
IdList -> id
        | IdList comma id
```

Input

```
x, y, z
```

AST





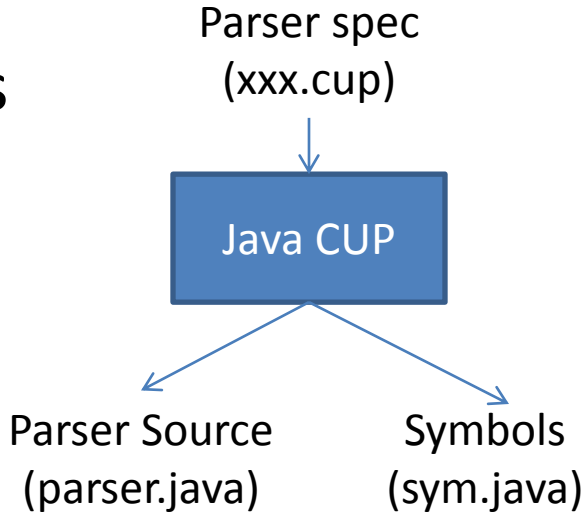
Parser Generators

Tools that take an SDT spec and build an AST

- YACC: Yet Another Compiler Compiler
- Java CUP: Constructor of Useful Parsers

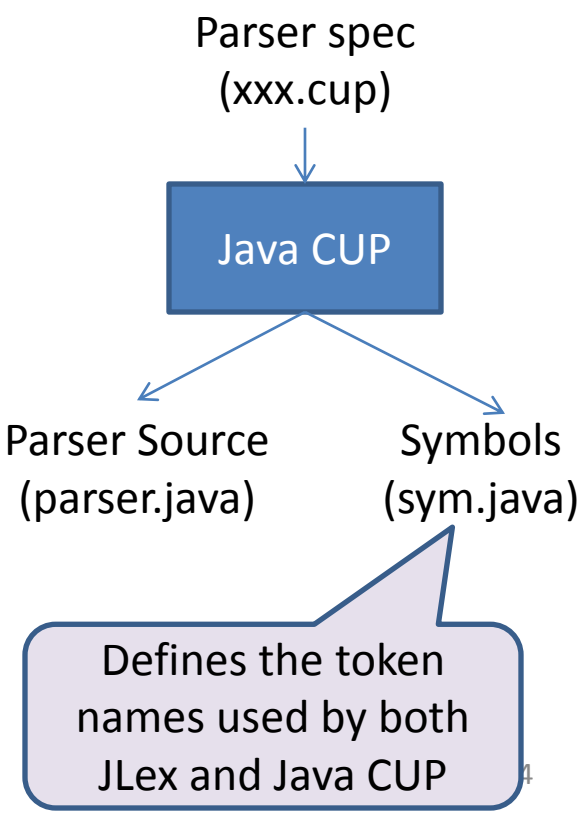
Conceptually similar to JLex

- Input: Language rules + actions
- Output: Java code



Java CUP

- Parser.java
  - Constructor takes arg of type Scanner (i.e., yylex)
  - Contains a parsing method
    - return: Symbol whose value contains translation of root nonterminal
  - Uses output of JLex
    - Depends on scanner and TokenVals
    - sym.java defines the communication language
  - Uses defs of AST classes
    - Also in xxx.cup



Input to java_cup

- optional package and import declarations
- optional user code
- terminal and nonterminal declarations
- optional precedence and associativity declarations
- grammar rules with associated actions

Output from java_cup

- parser.java

```
class parser {
  public parser(Yylex scanner) {…}
  public Symbol parse() {…}
…
}
```

returns a **Symbol** whose **value** field contains the translation of the root nonterminal

- sym.java

```
Class sym {
  public final static int TERMINAL=0;
  …
}
```

one **public final static int** for each terminal declared in the Java Cup specification

Java CUP Input Spec

- Terminal & nonterminal declarations
- Optional precedence and associativity declarations
- Grammar with rules and actions [no actions shown here]

Grammar rules

```
Expr ::= intliteral
       |  id
       |  Expr plus Expr
       |  Expr times Expr
       |  lparens Expr rparens
```

Terminal and Nonterminals

```
terminal intliteral;
terminal id;
terminal plus;
terminal minus;
terminal times;
terminal lparen;
terminal rparen;
non terminal Expr;
```

lowest precedence first

Precedence and Associativity

```
precedence left plus, minus;
precedence left times;
prededence nonassoc less;
```

Java CUP Example

- Assume ExpNode subclasses
  - `PlusNode`, `TimesNode` have 2 children for operands
  - `IdNode` has a String field
  - `IntLitNode` has an int field
- Assume Token classes
  - `IntLitTokenVal` with field intVal for the value of the integer literal
  - `IdTokenVal` with field idVal for the actual identifier

Step 1: Add types to terminals

```
terminal IntLitTokenVal intliteral;
terminal IdTokenVal id;
terminal plus;
terminal times;
terminal lparen;
terminal rparen;

non terminal ExpNode expr;
```

Java CUP Example

```
Expr ::= intliteral
         {:

         :}
     |   id
         {:

         :}
     |   Expr plus Expr
         {:

         :}
     |   Expr times Expr
         {:

         :}
     |   lparen Expr rparen
         {:

         :}
     ;
```

Java CUP Example

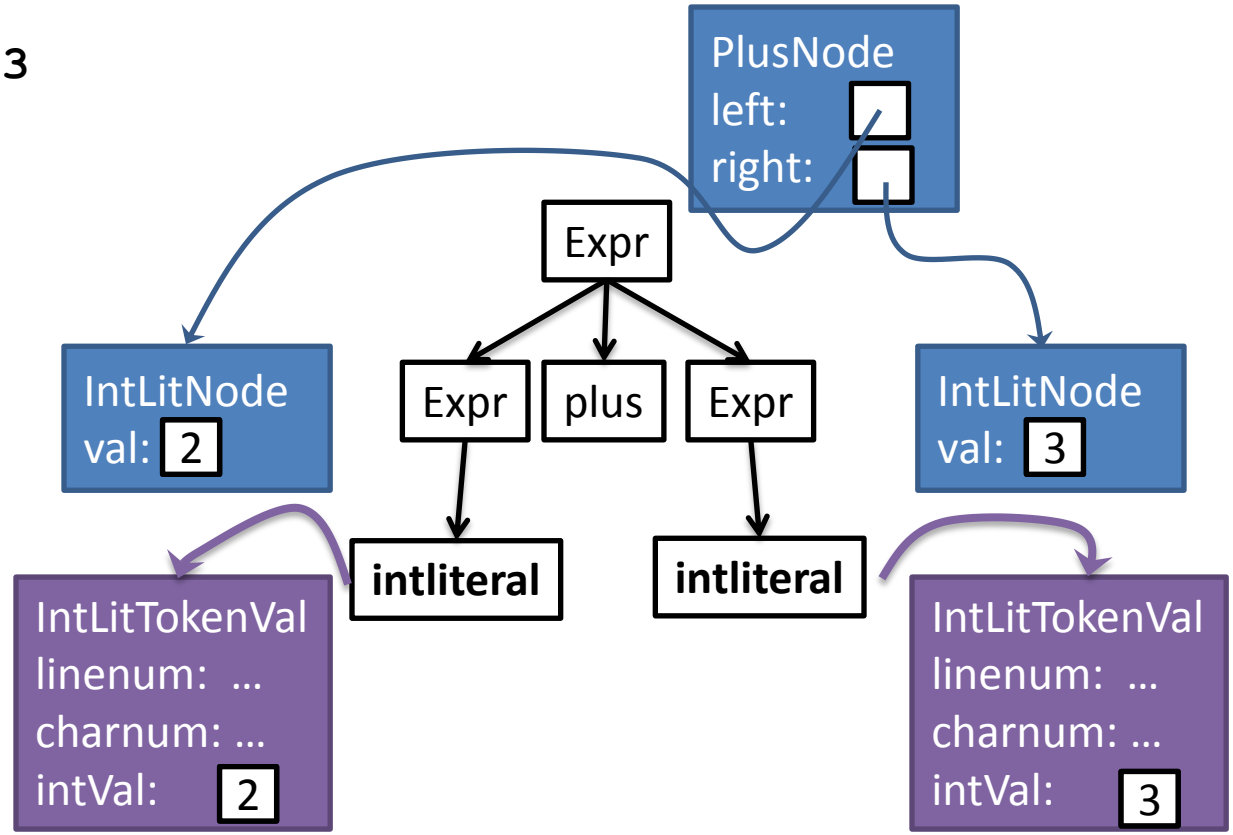
```
Expr ::= intliteral:i
         {:
            RESULT = new IntLitNode(i.intVal);
         :}
       |  id
         {:

         :}
       |  Expr plus Expr
         {:

         :}
       |  Expr times Expr
         {:

         :}
       |  lparen Expr rparen
         {:

         :}
       ;
```

Java CUP Example

```
Expr ::= intliteral:i
         {:
            RESULT = new IntLitNode(i.intVal);
         :}
       |  id:i
         {:
             RESULT = new IdNode(i.idVal);
         :}
       |  Expr:e1 plus Expr:e2
         {:
             RESULT = new PlusNode(e1,e2);
         :}
       |  Expr:e1 times Expr:e2
         {:
             RESULT = new TimesNode(e1,e2);
         :}
       |  lparen Expr:e rparen
         {:
             RESULT = e;
         :}
       ;
```

Java CUP Example

Input: 2 + 3



Purple = Terminal Token (Built by Scanner)
Blue = Symbol (Built by Parser)

Handling Lists in Java CUP

```
stmtList ::= stmtList:sl stmt:s
         {:  sl.addToEnd(s);
             RESULT = sl;
         :}
         |   /* epsilon */
         {:  RESULT = new Sequence();
         :}
         ;
```

Another issue: left-recursion (as above) or right-recursion?

- For top-down parsers, must use right-recursion
  - Left-recursion causes an infinite loop
- With Java CUP, <u>use left-recursion</u>!
  - Java CUP is a bottom-up parser (LALR(1))
  - Left-recursion allows a bottom-up parser to recognize a list s1, s2, s3, s4 with no extra stack space:

    recognize instance of “stmtList ::= epsilon” (current nonterminal stmtList)
    recognize instance of “stmtList ::= stmtList:current stmt:s1” [s1]
    recognize instance of “stmtList ::= stmtList:current stmt:s2” [s1, s2]
    recognize instance of “stmtList ::= stmtList:current stmt:s3” [s1, s2, s3]
    recognize instance of “stmtList ::= stmtList:current stmt:s4” [s1, s2, s3, s4]

Handling Unary Minus

```
/* precedences and associativities of operators */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
precedence nonassoc UMINUS;
```

UMINUS is a phony token never returned by the scanner. UMINUS is solely for the purpose of being used in "%prec UMINUS"

```
exp  ::=  . . .
     |   MINUS exp:e
     {:  RESULT = new UnaryMinusNode(e);
     :}    %prec UMINUS  /* artificially elevate the precedence to that of UMINUS
*/
     |   exp:e1 PLUS exp:e2
     {:  RESULT = new PlusNode(e1, e2);
     :}
     |   exp:e1 MINUS exp:e2
     {:  RESULT = new MinusNode(e1, e2);
     . . .
     ;
```

The precedence of a rule is that of the <u>last</u> token of the rule, unless assigned a specific precedence via "%prec <TOKEN>"

Grammar Rules

- Declare start non-terminal (otherwise uses head of first production rule)

```
start with program;
```

- Declare terminals and non-terminals:

```
terminal SEMICOLON;
terminal INT;
terminal IdTokenVal ID;
non terminal VarDeclNode varDecl;
non terminal TypeNode type;
non terminal IdNode id;
```

Grammar Rules

- Finally create production rules and actions:

```
varDecl ::= type:t id:i SEMICOLON {: RESULT = new
  VarDeclNode(t, i); :} ;
type ::= INT {: RESULT = new IntNode(); :} ;
id ::= ID:i {: RESULT = new IdNode(i.idVal); :} ;
```

::= divides head from body of production rule

:x used to give variable name to terminal/non terminals in body

{: :} separate out the action (java code)

RESULT special variable holding translation of head non-terminal

each production rules ends with semicolon

Running java_cup

Run the Main class

```
java java_cup.Main < xxx.cup
```

Rename the parser class (default is `parser`)

```
java java-cup.Main –parser CmmParser < cmm.cup
```