Java CUP
Last Time

What do we want?
  – An AST

When do we want it?
  – Now!
This Time

A little review of ASTs

The philosophy and use of a *Parser Generator*
Translating Lists

CFG

\[ \text{IdList} \rightarrow \text{id} \]
\[ \quad | \quad \text{IdList} \text{ comma } \text{id} \]

Input

\[ x, y, z \]

AST

- IdNode “x”
- IdNode “y”
- IdNode “z”
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
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;
```

**Precedence and Associativity**

```
prededence nonassoc less;
```

lowest precedence first

```
predecence left plus, minus;
predecence left times;
```
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

```java
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, use left-recursion!
  - 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;  // Also used for precedence of unary minus

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);
. . .
;

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

The precedence of a rule is that of the last token of the rule, unless assigned a specific precedence via “%prec <TOKEN>”
Java CUP Demo