CS 536 Announcements for Wednesday, February 21, 2024

Programming Assignment 2

- due Tuesday, February 20 – accepted until 11:59 pm Thursday
- see late policy on course website

Midterm 1

- Thursday, February 29, 7:30 – 9 pm
- S429 Chemistry
- bring your student ID

Last Time

- implementing ASTs

Today

- Java CUP

Next Week

- review for Midterm 1
- parsing

Parser generators

Tools that take an SDT spec and build an AST

- **YACC** Yet Another C Compiler – creates a parser in C
- **Java CUP** Constructor of Useful Parsers – creates a parser in Java

Conceptually similar to JLex:

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

$$\begin{tikzcd} \overset{\text{\_\_\_.cup}}{\textbf{parser specification}} \arrow[r] & \boxed{\textbf{Java CUP}} \arrow[r] & \textbf{parser source}\ \text{parser.java} \\ & & \textbf{symbols}\ \text{sym.java} \end{tikzcd}$$

Java CUP

`parser.java`

- constructor takes argument of type `Yylex` ← from scanner
- `parse` method
  - if input correct, returns `Symbol` whose `value` field contains translation of root nonterm
  - if input incorrect, quits on first syntax error
- uses output of JLex ← base.jlex
  - depends on scanner and `TokenVal` classes
  - `sym.java` defines the communication language = define token names used by both JLex & JavaCUP
- uses definitions of AST classes (in `ast.java`)

Parts of Java CUP specification

Grammar rules with ~~actions~~: ← not shown (yet)

```
expr ::= INTLITERAL
      |  ID
      |  expr PLUS expr
      |  expr TIMES expr
      |  LPAREN expr RPAREN
      ;
```

Terminal and nonterminal declarations:

```
terminal     INTLITERAL;
terminal     ID;
terminal     PLUS;
terminal     TIMES;
terminal     LPAREN;
terminal     RPAREN;

non terminal expr;
```

Precedence and associativity declarations:

```
precedence left PLUS;
precedence left TIMES;
```

↑ associativity

order (in __.cup) indicates precedence
low
↓
high

can do:

precedence nonassoc LESS;

Java CUP Example

defined in ast.java

Assume:

- Java class `ExpNode` with subclasses `IntLitNode`, `IdNode`, `PlusNode`, `TimesNode`
- `PlusNode` and `TimesNode` each have two children
- `IdNode` has a `String` field (for the identifier)
- `IntLitNode` has an `int` field (for the integer value)
- `INTLITERAL` token is represented by `IntLitTokenVal` class and has field `intVal`
- `ID` token is represented by `IdTokenVal` class and has field `idVal`

defined in base.jlex

Step 1: add types to terminals and nonterminals

```
/*
 * Terminal declarations
 */
terminal INTLITERAL;
terminal ID;
terminal PLUS;
terminal TIMES;
terminal LPAREN;
terminal RPAREN;
```

Need type if we want to use value associated with token

→ terminal IntLitTokenVal INTLITERAL;

→ terminal $\underbrace{\text{IdTokenVal}}_{\text{from scanner (base.jlex)}}$ ID;

```
/*
 * Nonterminal declarations
 */
non terminal expr;
```

Type required for all nonterms

non terminal $\underbrace{\text{ExpNode}}_{\text{from ast.java}}$ expr;

Step 2: add precedences and associativities

```
/*
 * Precedence and associativity declarations
 */
precedence left PLUS;
precedence left TIMES;
```

Java CUP Example (cont.)

Step 3: add actions to CFG rules

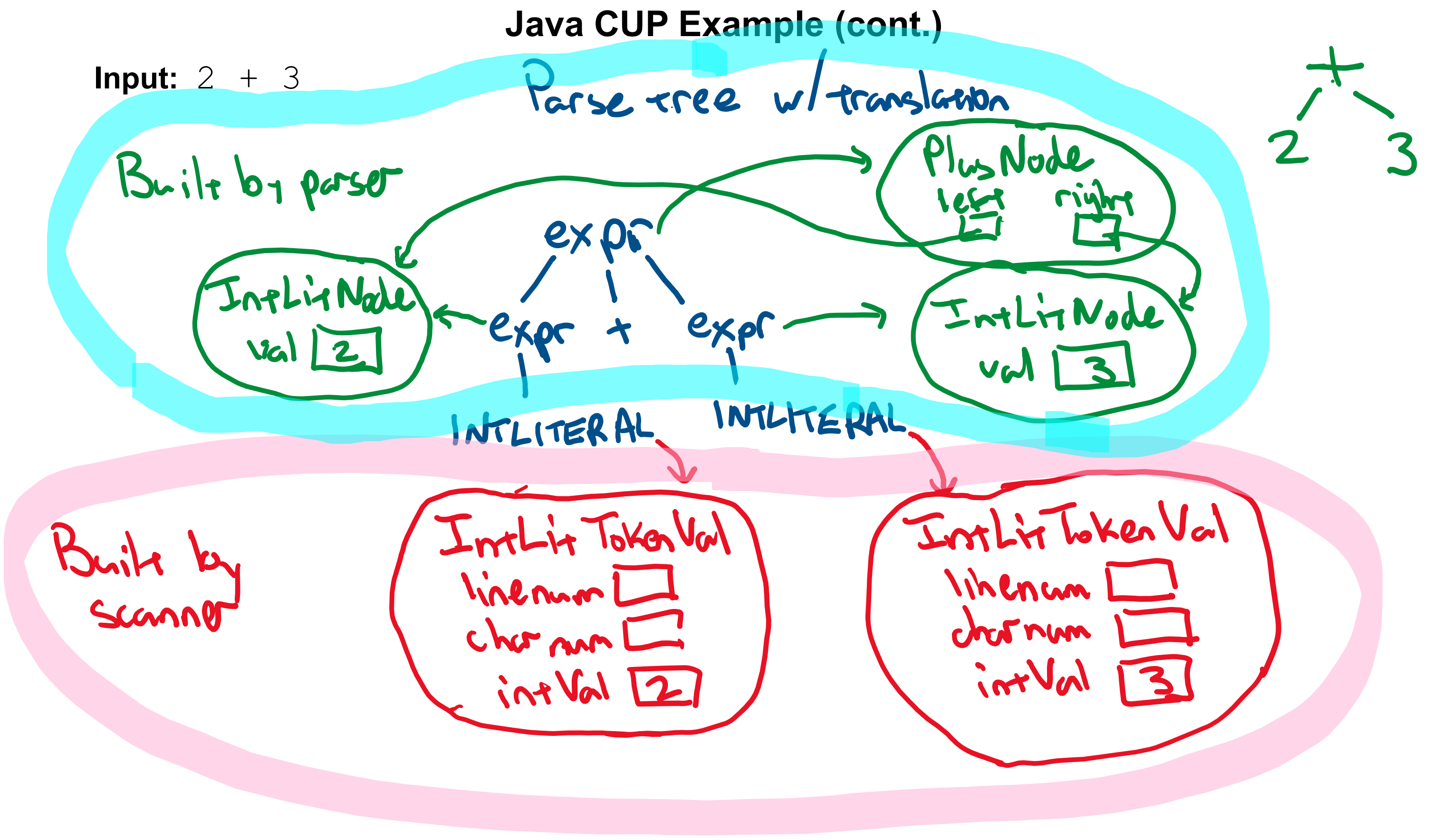
```
/*
 * Grammar rules with actions
 */
expr ::= INTLITERAL:i   <- type is IntLitTokenVal
         {:
            RESULT=new IntLitNode(i.intVal);
         :}
       | ID:i
         {:
            RESULT=new IdNode(i.idVal);              Subclasses
         :}                                               of
       | expr:e1   PLUS   expr:e2                      ExpNode
         {:
            RESULT= new PlusNode(e1, e2);
         :}
       | expr:e1   TIMES   expr:e2
         {:
            RESULT=new TimesNode(e1, e2);
         :}
       | LPAREN   expr:e   RPAREN
         {:
            RESULT=e;
         :}
       ;
```

General format

```
nonterm ::= rule1
            {: // action for rule1
               RESULT= ... ;
            :}
          | rule2
            {:
               RESULT= ... ;
            :}
          :
          ;
```

Java CUP Example (cont.)

Input: 2 + 3



Translating lists

Example left recursive

idList → idList COMMA ID | ID

Left-recursion or right-recursion?

- for top-down parsers must use right recursion
  left-recursion leads to infinite loop
- for Java CUP use left recursion
  → bottom-up parser

Example

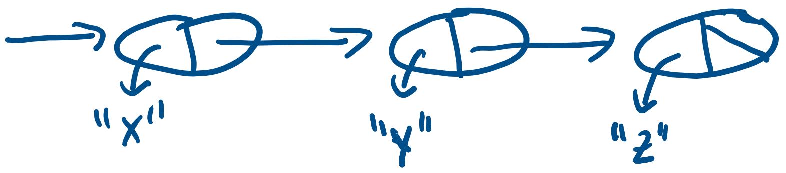
CFG: idList → idList COMMA ID | ID

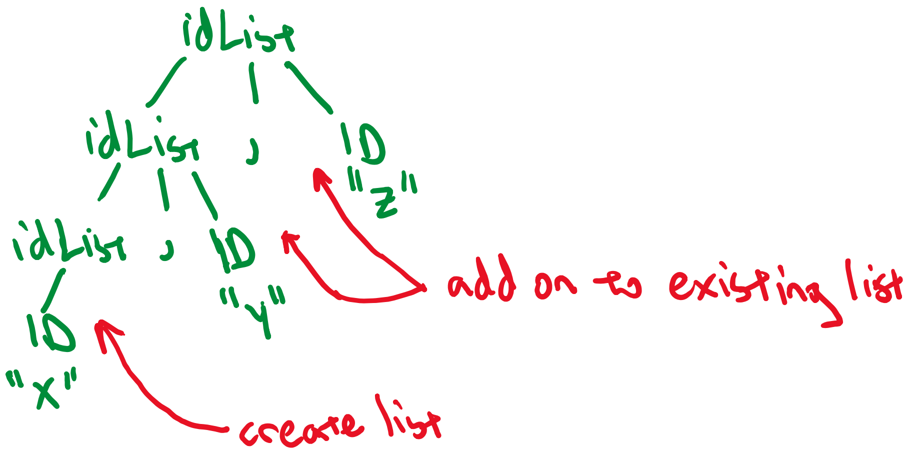
Goal: the translation of an `idList` is a `LinkedList` of `String`s

Example

Input: `x , y , z`

Output:





idList → idList COMMA ID
| ID

Example (cont.)

Java CUP specification for this syntax-directed translation

Terminal and nonterminal declarations:

```
terminal IdTokenVal ID;
terminal            COMMA;

non terminal LinkedList<String> idList;
```

Grammar rules and actions:

```
   idList ::= idList:L    COMMA       ID:i
              {:  L.addLast(i.idVal);
                  RESULT= L;
              :}
           |  ID:i
              {:
                 LinkedList<String> L = new LinkedList<String>( );
                 L.add(i.idVal);
                 RESULT=L;
              :}
           ;
```

Handling unary minus

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

binary minus has lowest precedence

"phony" token (never returned by scanner)

unary minus has highest precedence

```
/*
 * grammar rules
 */
exp  ::= . . .
     |    MINUS exp:e
       {: RESULT = new UnaryMinusNode(e);
       :} %prec UMINUS
     |    exp:e1 PLUS exp:e2
       {: RESULT = new PlusNode(e1, e2);
       :}
     |    exp:e1 MINUS exp:e2
       {: RESULT = new MinusNode(e1, e2);
       :}
        . . .
     ;
```

Precedence of a rule is that of the *last* token of the rule, *unless* assigned a specific precedence via %prec