• Please pick up from back of class
  – Written assignment 3
    • Useful exercise to help you transform your PA1 grammar to construct a precedence parser for PA2
    • Due next Tuesday, Feb 14
Topics

- PA1 testing and grading
  - Still in progress …

- AST construction and visitor classes
  - AST construction and traversal example
    - simpleAST example walkthrough – code on web site
  - miniJava ASTs
    - miniJava AbstractSyntaxClasses walkthrough
    - ASTDisplay

- Precedence parsing using a bottom-up parser
  - how does a bottom up parser work?
  - yacc & lex
    - precedence parsing
AST full example

- Check simpleAST miniArith in Examples section online
  - ex: simple arithmetic expressions
    \[
    \text{Expr} ::= \text{Expr} \ \text{Op} \ \text{Expr} \quad \text{(BinExpr)}
    \]
    \[
    \quad | \qquad \text{Num} \quad \text{(NumExpr)}
    \]

- Abstract syntax tree for \(2 + (3 * 4)\)

```
Expr
  /\  
Expr  Op  Expr
    /\     
Num  +  Expr
      |     
  2   3   Num
    |     |
    *   |
    4
```
AST representation

- Example

```
Expr ::= Expr Oper Expr (BinExpr)
  | Num (NumExpr)
```

```
abstract public class AST {}

abstract public class Expr extends AST {}

public class BinExpr extends Expr {
    public Token op;
    public Expr left, right;
    public BinExpr(Expr left, Terminal oper, Expr right) { ... }
}

public class NumExpr extends Expr {
    public Token num;
    public NumExpr(Terminal num) { ... }
}
```
Bottom-up parsing

- Example (not unlike a part of the miniJava grammar)
  CFG G has $N = \{S, A, B, D\}$, $T = \{a, b, d, $\}$
  
  $S ::= A \; \$$
  $A ::= B \mid D$
  $B ::= a \; B \mid b$
  $D ::= a \; D \mid d$

  Why not LL(1)?  Can we left-factor?

- CFG can be parsed “as is” using a bottom up parser
  - BU parser works by procrastination!
  - To parse w
    - parser delays its decision which rule to use as long as possible
  - It maintains the *viable prefix* property
    - If $v$ is the prefix of $w$ that has been read so far, then there must exist some $u \in NT^*$ such that $S \Rightarrow^* vu$. This means $vu \in L(G)$
    - How to ensure the viable prefix property?
Bottom-up parsing

- How do we recognize sentences using a BU parser?
  - Simulate a derivation
    - BU parser simulates a *rightmost* derivation in reverse!

- Bottom-up parser operation
  - input is read from left to right
  - parse stack initialized to $\varepsilon$
  - repeat until no choice available

  - SHIFT terminal $b$ onto parse stack
  OR
  - REDUCE $\alpha$ at top of parse stack to $A$
    - “predicting” correct rule $A ::= \alpha$

  - $w \in L(G)$ iff parse stack = $S$ when input is consumed
Canonical FSM for grammar

CFG

S ::= A $
A ::= B | D
B ::= a B | b
D ::= a D | d

S => .A$
S => A$
A => B.
A => D.
B => aB.
B => .aB
B => .b
D => aD.
D => .aD
D => .d
B => b.
D => d.

FSM tracks possible derivation states to decide whether to shift or reduce
Bottom-up parser

- Most powerful linear-time parser
  - parses largest class of grammars with given lookahead
    - LR(0), LR(1), ...
  - but can not parse all CFGs

- Uses a pair of stacks and a single table State x terminal -> State
  - symbol stack
    - symbol stack concatenated with remaining input is a sentential form in a rightmost derivation
  - state stack
    - determines which right hand side of a rule appears at stack top

- Works “backwards”
  - shifts input onto empty stack and replaces right hand sides of rules at stack top with left-hand nonterminal

- not amenable to direct implementation using recursive procedures
  - grammar analysis can generate a table-driven parser
  - typically use simplified lookahead LALR(1) to keep table size small
So why don’t we use a bottom-up parser?

- Canonical FSM doesn’t always yield conflict-free decisions in each state!
  - shift-reduce conflict
  - reduce-reduce conflict

- How does this get presented to the parser programmer?
  - shift-reduce conflict in state 285
  - reduce-reduce conflict in state 317, 424, 111
  - requires considerable expertise to resolve

- Bottom-up parsing
  - left recursion is very easy and generally doesn’t require any lookahead!
  - right recursion requires stacking input and lookahead
  - But …/ there exist simple grammars that are not LR(k) for any k>0
BU parser easily incorporates precedence and evaluation

- **Scanner (flex)**

```c
/* scanner for integer expression evaluation
 * evaluators */

%{
#include "y.tab.h"
}%

Num [0-9]+   
WS [\t\n]*

"+"  return (T_PLUS);
"-"  return (T_MINUS);
"*"  return (T_TIMES);
"/"  return (T_DIV);
"("  return (T_LPAREN);
")"  return (T_RPAREN);
{Num}  return (T_NUM);
{WS}   
.  printf("lexical error");
exit(4);
```

- **Parser (yacc)**

```c
/* parser for integer expression evaluation with precedence */

%{
extern char yytext[]; /* token spelling */
}%

%token T_NUM T_PLUS T_MINUS T_TIMES T_DIV T_LPAREN T_RPAREN
%
S    : Expr {printf("Ans = %d\n",$1);} 

Expr : Expr T_PLUS Expr {$$ = $1 + $3;} |
Expr T_MINUS Expr {$$ = $1 - $3;} |
Expr T_TIMES Expr {$$ = $1 * $3;} |
Expr T_DIV Expr {$$ = $1 / $3;} |
Expr T_LPAREN Expr T_RPAREN {$$ = $2;} |
T_MINUS Expr %prec NEG {$$ = -$2;} |
T_NUM   {$$ = atoi(yytext);}
```