COMP 520 - Compilers

Lecture 9 (Tue Feb 9, 2016)

Wrap-up: Parsing and ASTs

• Reading for Thu Feb 11
  – Finish Chapter 4
    • section 4.6 – Syntactic Analysis in Triangle (pp 124 – 128)
Topics

• **PA1 testing and grading**
  – 99 tests, 129 points (average 91, median 108)
  – test report and score in your submission directory
  – walk-through of tests and regression tester

• **AST construction and visitor classes**
  – AST construction and traversal example
    • simpleAST example walkthrough – code on web site

• **Precedence parsing using a bottom-up parser**
  – how does a bottom up parser work?
  – yacc & lex
    • precedence parsing
PA1 Testing and grading

- Tricky parts in PA1 scanning and parsing
  - Left factoring the miniJava grammar
  - Comments as whitespace
  - Unclosed comments
  - Operators with valid prefixes

- Testing
  - Special accommodation for small errors with large consequences
    - resubmission
    - oblivious parsers

- Tests
  - pa1_tests and the Checkpoint1 tester
  - run all tests or debug individual tests
AST full example

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

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

- **Example**

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

```java
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) { ... }
}
```
The Visitor design pattern

- Visitor interface requires a `visitX` method for every (non-abstract) AST class `X`:

  ```java
  public interface Visitor {
    visitBinaryExpr(BinaryExpr e);
    visitNumExpr(NumExpr e);
  }
  ```

- Each AST class is augmented with a single `visit` method:

  ```java
  class NumExpr extends Expr {
    public int val;
    public NumExpr(int v) { . . . }

    public void visit(Visitor v) { v.visitNumExpr(this); }
  }
  ```

- All AST traversals use the same “`visit`” method in each node type:
  - the method “connects” a specific visitor `v` to `this` specific node.
Bottom-up parsing

- **Example (not unlike a part of the miniJava grammar)**

  CFG $G$ has $N = \{S, A, B, D\}$, $T = \{a, b, d, \$\}$
  
  \[
  \begin{align*}
  S &::= A \$
  
  A &::= B \mid D \\
  B &::= a B \mid b \\
  D &::= a D \mid d
  \end{align*}
  \]

  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 there exists some $u \in NT^*$ such that $S \Rightarrow^* vu$
      
      - So there is some possible extension of $v \in L(G)$
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 ε
  - repeat until no choice available
    - SHIFT terminal b onto parse stack
    - REDUCE α at top of parse stack to A
      - “predicting” correct rule A ::= α

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

CFG

\[
\begin{align*}
S &::= A \$ \\
A &::= B \mid D \\
B &::= aB \mid b \\
D &::= aD \mid d
\end{align*}
\]
Bottom-up parser

- **Most powerful linear-time parser**
  - Parses largest class of grammars with given lookahead
    - LR(0), LR(1), …
  - But cannot 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
  – 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 * eval uat or */
%
#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 */
%
token T_NUM T_PLUS T_M NUS T_TI MES T_DIV V T_LPAREN T_RPAREN
%
eft T_PLUS T_M NUS eft T_TI MES T_DIV V right NEG
%
S : Expr {printf("Ans = %d\n",$1);} ;

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