CSE 401 – Compilers

Lecture 11: LL and Recursive-Descent Parsing

Michael Ringenburg

Winter 2013

Agenda

- Finish Discussing Visitor Patterns
- Top-Down Parsing (as much as we can)
  - LL(k) Grammars
  - Recursive Descent
  - Grammar Hacking
    - Left recursion removal
    - Factoring

Dynamic Dispatch

- Why do we need accept methods? Why can't we just call `visit(node)`, and have it dynamically select the correct `visit` method based on the runtime type of `node`?
- Because Java (and C++, and many other OO languages) use *single dispatch*.
  - In other words, dynamic dispatch in Java is based on the run-time type of a single object, namely the *receiver*, i.e, `x` in `x.foo(y, z)`

Two Code Samples

```
class WhileNd extends StmtNd {
  ExpNd cond;
}

void visit(WhileNd w) {
  visit(w.cond);
}
```

- Selects visit based on static type of w.cond (ExpNd).
- Typically an error (usually no "visit(ExpNd e)").

```
class WhileNd extends StmtNd {
  ExpNd cond;
  void accept(Visitor v) {v.visit(this);}
}
void visit(WhileNd w) {
  w.cond.accept(this);
}
```

- Selects accept based on dynamic type of w.cond (e.g., LessThan).
- Accept statically selects correct visit method.

How do we traverse AST?

```
public void visit(WhileNode p) {
  p.expr.accept(this);
  p.stmt.accept(this);
  …
}
```

- Visitors often control the traversal
  - The visitor knows what kind of traversal it needs – may be different for different operations/nodes.
  - E.g., imagine an optimization eliminates code that will never execute (this is called "Dead Code Elimination").
    - No need to traverse portions of the tree that you've proved won't execute.
  - Or, imagine a pass that records all variable declarations
    - No need traverse parts of the AST where declarations can't occur.

What Do Visitors Do?

- A visitor function has a reference to the node it is visiting (the parameter)
  - Will often access and manipulate subtrees directly
- Visitor object can also include local data (state) shared by the visitor methods

```
public class TypeCheckVisitor extends NodeVisitor {
    public void visit(WhileNode s) { … }
    public void visit(IfNode s) { … }
    …
    private <local state>;  // all methods can read/write this
}
```

A Brief Word About Encapsulation

- A visitor object often needs to be able to access state in the AST nodes
  - Thus, may need to expose more node state than we might do to otherwise
  - Overall a good tradeoff – better modularity
    - (plus, the nodes are relatively simple data objects anyway – not hiding much of anything)

References

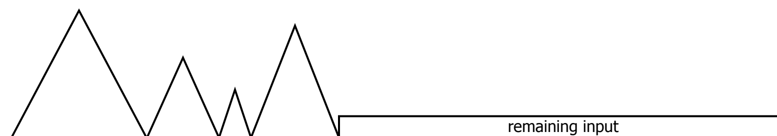
- For Visitor pattern (and many others)
  - *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson, and Vlissides, Addison-Wesley, 1995 (the classic, uses C++, Smalltalk)
  - *Object-Oriented Design & Patterns*, Horstmann, A-W, 2nd ed, 2006 (uses Java)
- Specific information for MiniJava AST and visitors in Appel textbook & online

Agenda

- Finish Discussing Visitor Patterns
- Top-Down Parsing (as much as we can)
  - LL(k) Grammars
  - Recursive Descent
  - Grammar Hacking
    - Left recursion removal
    - Factoring

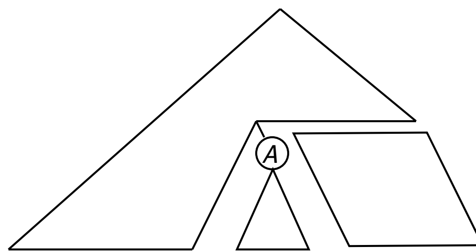
Basic Parsing Strategies

- Bottom-up
  - Build up tree from leaves
    - Shift next input or reduce a handle
    - Accept when all input read and reduced to start symbol of the grammar
  - LR(k) and subsets (SLR(k), LALR(k), …)



Basic Parsing Strategies

- Top-Down
  - Begin at root with start symbol of grammar
  - Repeatedly pick a non-terminal and expand
  - Success when expanded tree matches input
  - LL(k)



Top-Down Parsing

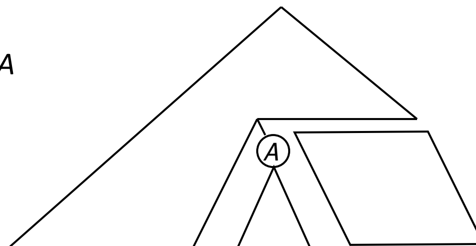
- General Idea: At any point, have completed N (possibly 0) steps of a leftmost derivation

  S =>* w$A\alpha$ =>* wxy (w,x,y are strings of terminals, α is string of any symbols, A is a nonterminal)

- Basic Step: Pick some production as step N+1

  $A ::= \beta_1 \beta_2 \dots \beta_n$

  that will properly expand A to match the input
  - Want this to be deterministic.



Predictive Parsing

- Ideally, if we are located at some non-terminal *A*, and there are two or more possible productions

  A ::= α

  A ::= β

  we want to make the correct choice by looking at just the next input symbol
- If we can do this, we can build a *predictive parser* that can perform a top-down parse without backtracking

Example

- Programming language grammars are often suitable for predictive parsing
- Typical example

  stmt ::= *id* = *exp* ; | return *exp* ;
  | if (*exp*) *stmt* | while (*exp*) *stmt*

  If the next part of the input begins with the tokens

  IF LPAREN ID(x) …

  we should expand *stmt* to an if-statement

LL(1) Property

- A grammar has the LL(1) property if, for all non-terminals *A*, if productions *A* ::= α and *A* ::= β both appear in the grammar, then it is the case that FIRST(α) $\cap$ FIRST(β) = Ø
- If a grammar has the LL(1) property, we can build a predictive parser for it that uses 1-symbol lookahead

LL(k) Parsers

- An LL(k) parser
  - Scans the input Left to right
  - Constructs a Leftmost derivation
  - Looking ahead at most k symbols
- 1-symbol lookahead is enough for many practical programming language features

Table-Driven LL(k) Parsers

- As with LR(k), a table-driven parser can be constructed from the grammar
- Rows are nonterminals that we want to expand, columns are terminals from the input, entries are productions
- Let's see how this would be done with:
    1. *S* ::= (*S*) *S*
    2. *S* ::= [*S*] *S*
    3. *S* ::= ε

Table-Driven LL(k) Parsers

Bad: (]

Good: ([ ] ( ) ([ ]))

1. *S* ::= (*S*) *S*
2. *S* ::= [*S*] *S*
3. *S* ::= *ε*

LL vs LR (1)

- Table-driven parsers for both LL and LR can be automatically generated by tools
- LL(1) has to make a decision based on a single non-terminal and the next input symbol
- LR(1) can base the decision on the entire left context (i.e., contents of the stack) as well as the next input symbol

LL vs LR (2)

- Thus, LR(1) is more powerful than LL(1)
  - Includes a larger set grammars
  - Some non-LL grammars can be converted to LL grammars, but this requires modifications that make the grammar harder to understand.
  - However – some of the better automatic tools (e.g., ANTLR) can do some of this conversion for you. Latest version of ANTLR claims it can do *all* the work for you in many cases (but caveats if you read the fine print).

Recursive-Descent Parsers

- Main advantage of top-down parsing is that it is easy to implement by hand
  - And automatic LL generators may produce easier to follow/debug code.
- Key idea: write a function (procedure, method) corresponding to each non-terminal in the grammar
  - Each of these functions is responsible for matching its non-terminal with the next part of the input
  - Recall this is top-down – so at each step we are looking at the leftmost non-terminal trying to figure out what to do.

Example: Statements

- Grammar

stmt ::= *id* = *exp* ;
| return *exp* ;
| if (*exp*) *stmt*
| while (*exp*) *stmt*

- Method for this grammar rule

```
// parse stmt ::= id=exp; | …
void stmt( ) {
 switch(nextToken) {
   RETURN: returnStmt(); break;
   IF:  ifStmt(); break;
   WHILE: whileStmt(); break;
   ID: assignStmt(); break;
 }
}
```

Example (cont)

```
// parse while (exp) stmt
void whileStmt() {
    // skip "while" "("
    // real parser would verify "("
    getNextToken(); getNextToken();

    // parse condition
    exp();

    // skip ")"
    getNextToken();

    // parse stmt
    stmt();
}
```

```
// parse return exp ;
void returnStmt() {
    // skip "return"
    getNextToken();

    // parse expression
    exp();

    // skip ";"
    getNextToken();
}
```

Invariant for Parser Functions

- The parser functions need to agree on where they are in the input
- Useful invariant: When a parser function is called, the current token (next unprocessed piece of the input) is the token that begins the expanded non-terminal being parsed
  - Corollary: when a parser function is done, it must have completely consumed input correspond to that non-terminal

Possible Problems

- Two common problems for recursive-descent parsers (and LL(1) parsers in general).
  - Left recursion (e.g., $E ::= E + T \mid \ldots$)
  - Common prefixes on the right side of productions

Left Recursion Problem

- Grammar rule

expr ::= *expr* + *term*
    | *term*

- And the bug is????

- Code

```
// parse expr ::= …
void expr() {
   expr();
   if (current token is
                 PLUS) {
    getNextToken();
    term();
   }
}
```

Left Recursion Problem

- If we code up a left-recursive rule as-is, we get an infinite recursion
- Non-solution: replace with a right-recursive rule

  expr ::= *term* + *expr* | *term*

  – Why isn't this the right thing to do?

One Left Recursion Solution

- Rewrite using right recursion and a new non-terminal
- Original: *expr* ::= *expr* + *term* | *term*
- New

  expr ::= *term exprtail*

  exprtail ::= + *term exprtail* | ε
- Properties
  – No infinite recursion if coded up directly
  – Maintains left associatively (*if you interpret the parse tree the right way, i.e., use the correct semantic actions)*
- Let's demonstrate that this really does generate the equivalent language.

Derivations

1. *expr* ::= *term exprtail*
2. *exprtail* ::= + *term exprtail*
3. *exprtail* :: = ε
4. *term* ::= x

Another Way to Look at This

- Observe that

  expr ::= *expr* + *term* | *term*

  generates the sequence

  (…((*term* + *term*) + *term*) + …) + *term*
- We can sugar the original rule to reflect this

  expr ::= *term* { + *term* }*
- This leads directly to parser code
  - Just be sure to do the correct thing to handle associativity as the terms are parsed

Code for Expressions (1)

```
// parse
//    expr ::=  term { + term }*
ExprNode expr() {
     ExprNode op1, op2;
     op1 = term();
     while (next symbol is PLUS) {
      getNextToken();
      op2 = term();
      op1 = CreatePlusNode(op1, op2);
     }
     return op1;
}
```

```
// parse
//     term ::= factor { * factor }*
ExprNode term() {
     ExprNode op1, op2;
     op1 = factor();
     while (next symbol is  TIMES) {
      getNextToken();
      op2 = factor();
      op1 = CreateMultNode(op1, op2);
     }
     return op1;
}
```

Code for Expressions (2)

```
// parse
//    factor ::= int | id | ( expr )
ExprNode factor() {
  ExprNode res;
  switch(nextToken) {

     case INT:
       res = processIntConstant();
       break;
     …
     case ID:
       res = processIdentifier();
       break;
     case LPAREN:
       getNextToken();
       res = expr();
       getNextToken();
     }
     return res;
}
```

Example (Showing Left Associativity)

4 + 5 + x

```
// parse
//    expr ::=  term { + term }*
ExprNode expr() {
    ExprNode op1, op2;
    op1 = term();
    while (next symbol is PLUS) {
     getNextToken();
     op2 = term();
     op1 = CreatePlusNode(op1, op2);
    }
    return op1;
}
```

Example (Showing Left Associativity)

4 + 5 + x

Op1: Int(4)

```
// parse
//    expr ::=  term { + term }*
ExprNode expr() {
    ExprNode op1, op2;
    op1 = term();
    while (next symbol is PLUS) {
     getNextToken();
     op2 = term();
     op1 = CreatePlusNode(op1, op2);
    }
    return op1;
}
```

Example (Showing Left Associativity)

4 + 5 + x

Op1: Int(4)
Op2: Int(5)

```
// parse
//    expr ::=  term { + term }*
ExprNode expr() {
     ExprNode op1, op2;
     op1 = term();
     while (next symbol is PLUS) {
       getNextToken();
       op2 = term();
       op1 = CreatePlusNode(op1, op2);
     }
     return op1;
}
```

Example (Showing Left Associativity)

4 + 5 + x

Op1: (4 + 5)
Op2: Int(5)

```
// parse
//    expr ::=  term { + term }*
ExprNode expr() {
     ExprNode op1, op2;
     op1 = term();
     while (next symbol is PLUS) {
       getNextToken();
       op2 = term();
       op1 = CreatePlusNode(op1, op2);
     }
     return op1;
}
```

Example (Showing Left Associativity)

4 + 5 + x

Op1: (4 + 5)
Op2: Id(x)

```
// parse
//    expr ::=  term { + term }*
ExprNode expr() {
     ExprNode op1, op2;
     op1 = term();
     while (next symbol is PLUS) {
       getNextToken();
       op2 = term();
       op1 = CreatePlusNode(op1, op2);
     }
     return op1;
}
```

Example (Showing Left Associativity)

4 + 5 + x

Op1: ((4 + 5) + x)
Op2: Id(x)

```
// parse
//    expr ::=  term { + term }*
ExprNode expr() {
     ExprNode op1, op2;
     op1 = term();
     while (next symbol is PLUS) {
       getNextToken();
       op2 = term();
       op1 = CreatePlusNode(op1, op2);
     }
     return op1;
}
```

Example (Showing Left Associativity)

4 + 5 + x

Op1: ((4 + 5) + x)
Op2: Id(x)
Returns ((4 + 5) + x)

```
// parse
//   expr ::= term { + term }*
ExprNode expr() {
    ExprNode op1, op2;
    op1 = term();
    while (next symbol is PLUS) {
      getNextToken();
      op2 = term();
      op1 = CreatePlusNode(op1, op2);
    }
    return op1;
}
```

What About Indirect Left Recursion?

- A grammar might have a derivation that leads to a left recursion

  $A \Rightarrow \beta_1 \Rightarrow^* \beta_n \Rightarrow A\ \gamma$

- There are systematic ways to remove this from a grammar
  - See any compiler or formal language book

Interesecting FIRST sets: Left Factoring

- If two rules for a non-terminal have right hand sides that begin with the same symbol, we can't predict which one to use
- Solution: Factor the common prefix into a separate production

Left Factoring Example

- Original grammar

  ifStmt ::= if (*expr*) *stmt*

  | if (*expr*) *stmt* else *stmt*

- Factored grammar

  ifStmt ::= if (*expr*) *stmt* *ifTail*

  ifTail ::= else *stmt* | ε

Parsing if Statements

- But it's easiest to just code up the "else matches closest if" rule directly.
- If you squint hard enough, this is really just left factoring with the two productions combined in a single routine.

```
// parse:  if (expr) stmt [ else stmt ]
StmtNode ifStmt() {
    getNextToken();     // ifStmt
    getNextToken();
    ExprNode cond = expr();
    getNextToken();
    StmtNode thenStmt = stmt();
    StmtNode elseStmt = null;     //ifTail
    if (next symbol is ELSE) {
      getNextToken();
      elseStmt = stmt();
    }
    return ifStmt(cond, thenStmt, elseStmt);
}
```

Another Lookahead Problem

- In languages like FORTRAN, parentheses are used for array subscripts
- A FORTRAN grammar includes something like

    factor ::= *id* (*subscripts*) | *id* (*arguments*) | …
- When the parser sees "*id* (", how can it decide whether this begins an array element reference or a function call?

Two Ways to Handle *id* (?)

- Use the type of *id* to decide
  - Requires declare-before-use restriction if we want to parse in 1 pass
- Use a covering grammar

  factor ::= *id* (*commaSeparatedList*) | …

  and fix/check later when more information is available (e.g., types)

Top-Down Parsing Concluded

- Works with a smaller set of grammars than bottom-up, but can be done for most sensible programming language constructs
  - With some possible grammar refactoring.
- If you need to write a quick-n-dirty parser, recursive descent is often the method of choice
  - Also, some sophisticated hand-written parsers for real languages (e.g., C++) are "based on" LL parsing, but highly customized.

Parsing Concluded

- That's it!
- On to the rest of the compiler
- Coming attractions
  - Intermediate representations (ASTs etc.)
  - Semantic analysis (including type checking)
  - Symbol tables
  - & more…