LL and Recursive-Descent Parsing

Hal Perkins
Winter 2009

Agenda
- Top-Down Parsing
- Predictive Parsers
- LL(k) Grammars
- Recursive Descent
- Grammar Hacking
  - Left recursion removal
  - Factoring

Basic Parsing Strategies (1)
- 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 (2)
- 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
- Situation: have completed part of a derivation
  \[ S \Rightarrow^* wA \Rightarrow^* wy \]
- Basic Step: Pick some production
  \[ A \colon= \beta_1 \beta_2 \ldots \beta_n \]
  that will properly expand \( A \) to match the input
  - Want this to be deterministic

Predictive Parsing
- If we are located at some non-terminal \( A \), and there are two or more possible productions
  \[ A \colon= \alpha \]
  \[ A \colon= \beta \]
  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(k) 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) = \emptyset`
- 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 grammars
  - LL(k) for `k>1` is very rare in practice

Table-Driven LL(k) Parsers

- As with LR(k), a table-driven parser can be constructed from the grammar
- Example
  
  1. `S ::= ( S ) S`
  2. `S ::= [ S ] S`
  3. `S ::= ε`

- Table
  
<table>
<thead>
<tr>
<th></th>
<th>[ ]</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>S</code></td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

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)

- LR(1) is more powerful than LL(1)
  - Includes a larger set of grammars
- (editorial opinion) If you’re going to use a tool-generated parser, might as well use LR
  - But there are some very good LL parser tools out there (ANTLR, JavaCC, …) that might win for non-LL vs LR reasons
Recursive-Descent Parsers

- An advantage of top-down parsing is that it is easy to implement by hand
- 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

Example: Statements

- Grammar
  - stmt ::= id = 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 ("
    getNextToken();
    getNextToken();
    // parse condition
    exp();
    // skip ")"
    getNextToken();
    // parse stmt
    stmt();
  }

Invariant for 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 (and LL(1)) parsers
  - Left recursion (e.g., $E ::= E + T | ...$)
  - Common prefixes on the right hand side of productions

Left Recursion Problem

- Grammar rule
  - expr ::= expr + term
  - | term

- Code
  - // parse expr ::= ...
  - void expr() {
    expr();
    if (current token is PLUS) {
      getNextToken();
      term();
    }
  }

And the bug is????
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?

Another Way to Look at This

- Observe that
  \[ expr ::= expr + term | term \]
  generates the sequence
  \[ term + term + term + \ldots + term \]
- We can sugar the original rule to reflect this
  \[ expr ::= term \{ + term \}^* \]
- This leads directly to parser code

Code for Expressions (1)

```c
// parse
void expr() {
  term();
  while (next symbol is PLUS) {
    getNextToken();
    term();
  }
}
```

Code for Expressions (2)

```c
// parse
void factor() {
  switch (nextToken) {
    case ID:
      process identifier;
      getNextToken();
      break;
    case INT:
      process int constant;
      getNextToken();
      break;
    case LPAREN:
      getNextToken();
      expr();
      getNextToken();
      break;
    case PLUS:
      getNextToken();
      expr();
      break;
  }
}
```

What About Indirect Left Recursion?

- A grammar might have a derivation that leads to a left recursion
  \[ A \Rightarrow \beta_1 \Rightarrow \ast \beta_n \Rightarrow A \gamma \]
- There are systematic ways to factor such grammars
  - See any good compiler book
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

Original grammar

```plaintext
ifStmt ::= if ( expr ) stmt
     | if ( expr ) stmt else stmt
```

Factored grammar

```plaintext
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
  ```c
  // parse if ( expr ) stmt [ else stmt ]
  void ifStmt() {
    getNextToken(); // expr
    getNextToken(); // (
    expr();
    getNextToken(); // )
    stmt();
    if (next symbol is ELSE) {
      getNextToken(); // else
      stmt();
    }
  }
  ```

Another Lookahead Problem

- In languages like FORTRAN, parentheses are used for array subscripts
- A FORTRAN grammar includes something like
  ```plaintext
  factor ::= id ( commaSeparatedList ) | 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
  ```plaintext
  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
- If you need to write a quick-n-dirty parser, recursive descent is often the method of choice
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...