LL and Recursive-Descent Parsing

CSE 413
Autumn 2007

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 using a production
    - 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
  \[ S \Rightarrow \star wA \Rightarrow \star w \]
- Basic Step: Pick some production
  \[ A ::= \beta_1 \beta_2 ... \beta_n \]
  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
  \[ A ::= \alpha \]
  \[ A ::= \beta \]
  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 first part of the unparsed 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
  ```
  A ::= α
  A ::= β
  ```
  both appear in the grammar, then:
  ```
  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
  ```
  S ::= < S > S
  ```
  1. `S ::= ( S ) S`
  2. `S ::= ε`
  Table
  ```
  $ 1 2 3 3$
  ```

Table-Driven LL(k) Parsers
- As with LR(k), a table-driven parser can be constructed from the grammar
  ```
  S ::= < S > S
  ```
  1. `S ::= ( S ) S`
  2. `S ::= ε`
  Table
  ```
  $ 1 2 3 3 3$
  ```

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 as well as the next input symbol

LL vs LR (2)
- LR(1) is more powerful than LL(1)
  ```
  ::= LR(1) is more powerful than LL(1)
  ```
  Includes a larger set of grammars
  But
  ```
  But
  ```
  It is easier to write a LL(1) parser by hand
  There are some very good LL parser tools out there (ANTLR, JavaCC, ...)
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 |
        return exp |
        if (exp) stmt |
        while (exp) stmt
  ```

- **Method for this grammar rule**
  
  ```
  void stmt(){
    switch(nextToken){
      returnStmt(); break;
      ifStmt(); break;
      whileStmt(); break;
      assignStmt(); break;
    }
  }
  ```

Example (cont)

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

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 corresponding 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**
  
  ```
  void 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?

Left Recursion Solution
- Rewrite using right recursion and a new non-terminal
  - **Original:** \( expr ::= expr + term | term \)
  - **New:**
    \[ expr ::= term expr \]
    \[ expr \] = + term expr \| \epsilon \]
- Properties
  - No infinite recursion if coded up directly
  - Maintains left associatively (required)

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 show this
  \[ expr ::= term \{ + term \}^* \]
- This leads directly to parser code

Code for Expressions (1)
```c
void expr() {
  term();
  while (next symbol is PLUS) {
    getNextToken();
    term();
  }
}
```
```c
void term() {
  factor();
  while (next symbol is TIMES) {
    getNextToken();
    factor();
  }
}
```

Code for Expressions (2)
```c
void factor() {
  case INT:
    process identifier;
    break;
  case LPAREN:
    getNextToken();
    while (next symbol is TIMES) {
      getNextToken();
      factor();
    }
    getNextToken();
    break;
  default:
    process identifier;
    break;
  }
}
```

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

```
// parse
//   if (expr) stmt [ else stmt ]

void ifStmt() {
    getNextToken();
    stmt();
    if (nextSymbol == ELSE) {
        getNextToken();
        stmt();
    }
}
```

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