Parsing Techniques

CS152

Chris Pollett

Sep. 24, 2008.

Outline

- Top-down versus Bottom-up Parsing
- Recursive Descent Parsing
- Left Recursion Removal
- Left Factoring
- Predictive Parsing

Introduction

- We want to describe now how to write parsers for a grammar written in EBNF.
- At its most basic a parser needs to be able to recognizes programs in the programming language specified by the grammar.
- Most likely also want to be able to build a abstract syntax tree and attach some semantics to the program.

Top-down versus Bottom-up Parsing

- There are two approaches to parsing:
  - **Bottom-up**: Here we start from the program and try to match initial segments to left hand sides of rules. When we get a match, the right hand-side of a rule is replaced **(reduced)** with its left hand side on the stack. These parsers are sometimes called **shift/reduce parsers**, because one often shifts token onto the stack prior to deciding to do a reduction.
  - **Top-down**: One starts at the start symbol for the grammar, and replaces non-terminals with the right-hand-side of rules until one gets down to terminals which match the input. (Mentioned last-day.)
- Both methods can be automated. Yacc/Bison is a shift/reduce parser.

Recursive Descent Parsing

- One common way to write a top-down parser by hand is to rely on the run-time stack of the language you are writing the compiler in.
- That is, for each non-terminal you write a procedure. This procedure is supposed to be able to do parsing for that non-terminal. If that non-terminal is the right hand-side of a rule, the procedure will try to match any tokens in the rule to the input, and recursively call procedures for non-terminals on the right hand side of the rule.

Example

- Consider:

  sentence -> nounPhrase verbPhrase .

  nounPhrase -> article noun.

  article -> a | the .

- This might yield procedures such as:

```
void sentence (void) {nounPhrase();verbPhrase;}
void nounPhrase(void) {article(); noun();}
void article(void) {if (token==“a”) match(“a”);
  else if (token == “the”) match(“the”);
  else error(); }
```

- We imagine token is a global variable provided by the scanner/lexer.

What if a non-terminal has multiple things it goes to?

- We mentioned last day that one problem with ambiguous grammars was that we can't figure out what to put on the stack when doing top-down parsing. Isn't this the same problem?
- No. As long as the grammar is not ambiguous, we can take an approach like for article above. Consider S->AB | CD. We could write:

  ```
  void S() { A(); B(); if(parseError()) {rewind(); C(); D();
      }}
  ```

- rewind() returns the string to where we started parsing S. This is called *backtracking.*
- Ideally, we want to design our grammars so we don't need to do backtracking.

Left Recursion Removal

- Another issue with recursive descent is that it will tend to go into an infinite loop if you have a **left-recursive rule**. For example, a rule like *expr -> expr + term* | *term* where left hand side nonterminal is also the leftmost nonterminal on the right-hand side of the rule.
- This can be fixed by changing the above to *expr -> term + expr* | *term* , but note this makes + into a right associative operator.
- Code would look like:

```
void expr(void) {term(); if(token == "+"){match("+");
   expr();}}
```

Fixing Associativity

- Notice if we write the above in EBNF it becomes *expr -> term {+ term}*, a *term* followed by 0 or more + *term*'s. So we see this could be handled by using a loop rather than recursion in our procedure:
  - void expr() {term(); while(token == "+"){match("+"); term();}}
  - Just after the second call to term we can handle the associativity as we desire.

Left Factoring

- A right recursive rule like:

  <expr> -> <term> @ <expr> | <term>

  Can also be rewritten in EBNF as:

  <expr> -> <term> [@ <expr>]
- This is called **left factoring**.
- Consider:

  <if-statement> -> if(<expr>) <statement> |

  if(<expr>) <statement> else <statement>
- This cannot be directly translated into code as both rules begin with the same prefix, but we can “factor out” the prefix:

  <if-statement> -> if(<expr>) <statement> [else <statement>]
- This can be code viewing the [ ] as an if clause:

```
void ifStatement() {match(“if”); match(“(”); expression(); match(“)”);
    statement(); if(token==“else”){match(“else”); statement();}}
```

Predictive Parsing

- As we mentioned above, we would like to avoid backtracking.
- This means we need a way to predict which rule to select for a given nonterminal.
- For grammars which meet two conditions we now describe this can be done.
- The idea is that the parser will do a *single-symbol lookahead* ahead and use that to determine which rule to use.

More on Predictive Parsing

- Consider a rule of the form A -> α_1| α_2| α_3|…| α_n.
- The first condition is that the first symbols of each of the rules must be distinct.
- For example, for the grammar:

  <factor> ::= (<expr>)|<number>

  <number> ::= <digit> {<digit>}

  <digit> ::= 0|1|2|3|4|5|6|7|8|9
- We need First((<expr>)) and First(<number>) to be disjoint.
- First((<expr>)) = {(} and First (<number>) = First(<digit>) = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} so the condition holds.

Second Criteria for Predictive Parsing

- If we have rule of the form A->α[β]γ (I.e., we have an optional β), then the set of first tokens β can go to must be distinct from the set of tokens that could immediately follow β.
- For example

  A -> B [C] D.

  C-> aE | bF.

  D-> cG.

  Then First(C) = {a, b} and Follow(C) ={c}. So the criteria would hold.