Class 20: Predictive Parsing (3)

Held: Monday, 10 October 2011

Summary: Today we conclude our exploration of predictive parsing by considering the construction of the primary tables and functions.

Related Pages:

- EBoard.

Notes:

- EC for the Thursday extra, by a visitor from UNL.
- EC for the Thursday’s panel on the future of the Book, 4:15 in the gallery.
- If you have not done so already, start reading section 4.5.

Overview:

- The Follow Table.
- Building the Follow Table.
- Using Predictive Parsing Tables.
- Potential Problems with Predictive Parsing.
- Reflecting on Predictive Parsing.
- Looking Ahead: Predictive Parsing.

Building the Follow Table

- What we did last class:

  For all rules of the form \( m := \text{Stuff} \ n \ \text{MoreStuff} \)
  \[
  \text{Follow}[n] := \text{Follow}[n] \cup \text{first}(\text{MoreStuff})
  \]

- Our leftover question:
  - Consider the rule \( m := \text{Stuff} \ n \)
  - What does this tell us about \( \text{Follow}[m] \) or \( \text{Follow}[n] \)?
- A solution: Anything that follows the nonterminal on the left can follow the nonterminal on the right.
  - That is, for rules of this form, we should say

    \[
    \text{Follow}[n] := \text{Follow}[n] \cup \text{Follow}[m]
    \]

  - Sketch of proof:
So, anything that comes after m can come after m.

More generally, what if we the stuff that follows m is nullable?

m := Stuff n MoreStuff
    // nullable (MoreStuff) \rightarrow; TRUE
    // Stuff and MoreStuff are sequences of nonterminals and
    //    terminals

A solution: Anything that follows the nonterminal on the left can follow the nonterminal on the right.

Sketch of proof:

Building the Predictive Parsing Table

We’re going to build a table that tells you, for every [nonterminal,terminal] pair, what rule to apply next.

If a table cell is empty, it’s an error.

If a table cell contains a RHS, we match each element of the rhs side.

If a table cell contains epsilon, we reduce immediately to epsilon

Recall that the predictive parsing table tells you, for every nonterminal/terminal pair, what rule to apply next.

for each production, p_i of the form n ::= rhs
    for each terminal, T, in first(rhs)
        set Parse[n,T] to p_i
    if nullable(rhs)
        for each terminal, T, in Follow[n]
            set Parse[n,T] to p_i
    If you ever try to set an already-set position
    report an error

Some Grammars that Cause Problems

As you may have noted in your own work (or your analysis of the code above), some grammars seem to be unamenable to predictive parsing.

In particular, grammars in which two RHS’s can begin with the same token cannot be predictively parsed.
This problem can usually be solved by *left factoring* the grammar:

- Identify the common initial sequence.
- Rewrite the grammar so that there is only one rule for the nonterminal that uses that common initial sequence which is then followed by a new nonterminal.
- That new nonterminal then represents all possible remainders.

```
conditional : _IF test _THEN statement optional_else ;
optional_else : /* epsilon */
              | _ELSE statement ;
```

It turns out that *left recursive* grammars (grammars in which a nonterminal can derive a sequence that starts with that nonterminal) also cannot be predictively parsed.

```
lines : lines line
       | line ;

lines : line lines
       | line ;

lines : line morelines
       ;
morelines : lines
         | /* epsilon */ ;
```

Warning! We won’t be doing this with Yacc/Bison, since Yacc/Bison use a different approach.

### Predictive Problems, Revisited

- Many grammars are not amenable to predictive parsing.
- Some are for languages not amenable to predictive parsing.
- Some must be mangled to permit predictive parsing.
- Neither situation is useful.
- With careful programming, we can handle use grammars to generate unmangled parse trees.
- With more lookahead, we can often do better (at keeping the original grammar and at accepting a greater range of languages).
- However, it is also instructive to look for other techniques, which we will start in our next class.
Question: What do you see as the potential difficulties in writing a predictive parser for Pascal?

Looking Ahead to Shift-Reduce Parsing

- The main problem we have right now is that when we see a symbol, we might not be sure which rule applies until we read some more stuff.
  - And we might need to read a lot more stuff.
- Predictive parsing insists that we make the decision immediately upon seeing a symbol.
- An alternative strategy, shift-reduce parsing, lets us delay reduction.
- In shift-reduce parsing, we keep a stack of unprocessed stuff.
  - The stuff may include terminals that we’ve put aside until we know which rule the participate in.
  - The stuff may also include nonterminals. And, again, we may not know what rule they participate in.
- When we see a token, we have two basic choices:
  - We can shift it onto the stack, noting that we are not yet sure what role it serves.
  - We can reduce what is on top of the stack, applying a production from right to left. (That is, converting a sequence of terminals and nonterminals into a nonterminal.)