Class 18: Predictive Parsing (2)

Held: Wednesday, 5 October 2011

Summary: Today we continue our exploration of predictive parsing by examining steps for automating the construction of predictive parsers.

Related Pages:

- EBoard.

Notes:

- There has been a request to move the exam to the Wednesday after fall break.
- I’m still working on the next phase of the project and anticipate releasing it tonight. Remember! The more code I write, the less code you have to write.
- Today is an early out for G-N Schools. I will have abbreviated office hours.

Overview:

- Predictive Parsing, Revisited.
- The Epsilon Problem.
- Helpful tables: First, Follow, and Nullable
- Building the First table.
- Building the Nullable table.
- Building the Follow table.
- Generalizing the Parser.

Predictive Parsing, Revisited

- Recall that we’re building a parser that does something like the following:

  When matching nonterminal \( n \), with current symbol \( S \)
  
  For each rule, \( n ::= rhs \)
  
  If \( S \) can begin \( rhs \),
  
  match the parts of \( rhs \)

- This strategy fails if \( S \) can begin multiple right-hand-sides.
- In order to use this strategy, we need to be able to figure out what can begin \( rhs \).

The Epsilon Problem

- We also need to figure out what to do about rules of the form
  
  \( n ::= \text{epsilon} \).
- Or, more generally, what to do about rules in which the RHS can derive epsilon.
Consider what to do when you see a \( B \) when matching an \( s \) in

\[
\begin{aligned}
s &: x \ B \\
& s : C \\
& x : X \ x \\
& | A \\
& | \text{epsilon} \\
\end{aligned}
\]

Analyzing the Grammar

- All of these analyses are aided by three tables (and corresponding functions) known as First (and first), Follow, and Nullable (and nullable).
- The tables and functions are mutually recursively constructed. That is, the functions may be defined in terms of the tables and the tables may be defined in terms of the current versions of the functions.
- **First** maps nonterminals to sets of tokens that can begin strings derived from those nonterminals.
  - **first** maps sequences of nonterminals and terminals to the symbols that can begin strings derived from those sequences
- **Follow** maps nonterminals to sets of tokens that can follow those nonterminals in sequences derivable in the grammar
  - There is no follow function as it is neither interesting nor useful.
- **Nullable** indicates which nonterminals can derive the empty string
  - **nullable** operates on sequences of symbols (nonterminals and terminals), returning true only when the sequence can derive epsilon

Building the First table and first function

- How do we define the first function?
  - We know that if a sequence begins with a token, then it can only derive strings that begin with that token.
- But what if the sequence begins with a nonterminal?
  - Then the sequence can derive strings that begin with any token that begins strings that the nonterminal can derive.
- What what if the nonterminal can derive the empty string?
  - Then we need to look further in the original sequence.
- Putting it all together

```plaintext
TokenSet first(SymbolList string) {
    Symbol sl = string.car();
    if (isToken(sl)) {
        return new TokenSet(sl);
    }
    else {
        tmp = First[sl];
        if (Nullable[sl]) {
            return new Union(tmp,
```
Note that this relies on the First table and the Nullable table.

How do we build the First table? Using the first function.

for each nonterminal N, First(N) = emptySet();
repeat
  for each production N ::= RHS
    First(N) = union(First(N), first(RHS))
  end for
until no changes are made

How do we know this terminates?

Building the Nullable table and the nullable function

The nullable function is relatively easy to define

boolean nullable(SymbolList string) {
  // The empty string can derive the empty string
  if (String.isEpsilon()) {
    return true;
  }
  Symbol s1 = string.car()
  // If we begin with a token, we’re not nullable
  if (isToken(s1)) {
    return false;
  }
  // Otherwise, we begin with a nonterminal. That nonterminal must
  // be nullable, as must the rest of the sequence.
  else {
    return Nullable[s1] &&
    nullable(string.cdr());
  }
} // nullable

And, once again, we can define the Nullable table in terms of the nullable function.

foreach nonterminal N, Nullable(N) = false
repeat
  for each production N ::= RHS
    if nullable(RHS) {
      Nullable[N] = true;
    }
  end for
until no changes are made

Why are we sure this loop terminates?
The Follow table is our final table for the predictive parser.

For every nonterminal, $n$, $\text{Follow}[n]$ is the set of terminals that can follow $n$ in a derived string.

- Optimally: In a string derived from the start symbol.
- Practically: In a string derived from some symbol.

The Follow table is used to tell us when we can apply a rule in which the rhs is nullable.

Why do we care?

- Sometimes it helps us catch errors earlier. (That is, you shouldn’t nullify something when you know it will eventually lead to a dead end.)
- More often, it helps us catch problems in the grammar.

### Building the Follow Table

- The easy part: Whenever a terminal, $T$, follows a nonterminal, $n$, in a right-hand side, add $T$ to $\text{Follow}[n]$.
- The moderate part: Whenever a nonterminal, $m$, follows a nonterminal, $n$, in a right-hand side, add $\text{First}[m]$ to $\text{Follow}[n]$.
- The hard part: What do you do when a nonterminal, $m$, follows a nonterminal, $n$ and $m$ is nullable?