Class 21: Shift-Reduce Parsing (1)

Held: Wednesday, 12 October 2011

Summary: Today we consider an alternate form of parsing, shift-reduce parsing. We can use shift-reduce parsing for a larger set of grammars than we can use predictive parsing for.

Related Pages:
- EBoard.
- Reading: Aho et al. 4.5.

Notes:
- EC for tomorrow’s Thursday extra.

Overview:
- Introduction to shift-reduce parsing.
- A shift-reduce parser for a non-predictive language.
- A shift-reduce parser for a simple expression language.
- LR(0) parsers: Basic shift-reduce parsers.

Shift-Reduce Parsing

- Note also that recursive descent parsers are, in effect, top-down (you start with the start symbol and attempt to derive the string).
- We can gain some power by starting at the bottom and working our way up.
- The most common bottom-up parsers are the so-called shift-reduce parsers. These parsers examine the input tokens and either shift them onto a stack or reduce the top of the stack, replacing a right-hand side by a left-hand side.
- Think about the sample grammar for $a^n b^n | a^n c^n$, which we write as

  ```
  s : b
  s : c
  b : A b B
  b : epsilon
  c : A c C
  c : epsilon
  ```

- Note that this language is unlikely to be amenable to predictive parsing.
  - We’ll ignore the ambiguity of parsing the empty string for now.
  - Since both RHSs can begin with A, we factor it out
s : A s'
s' : b B
s' : c C

● Hmmm, both choices for s’ can begin with A, so we do this again

s' : B
s' : C
s' : A s''
s'' : b B B
s'' : c C C

● Whoops. The same problem all over again.

● Here’s a shift-reduce process for matching the language
  ○ If you see an A with nothing on the stack, shift A onto the stack.
  ○ If you see an A with A on the stack, shift A onto the stack.
  ○ If you see a B with an A on top of the stack, reduce the epsilon on top of the stack to b (which
    goes back on the stack)
  ○ If you see a B with [A b] on top of the stack, shift the B onto the stack.
  ○ If you see a B with [A b B] on the top of the stack, pop the top three elements of the stack and
    then push b. That is, reduce [A b B] to b.
  ○ If you see end of input with [A b B] on the top of the stack, pop the three elements of the stack
    and then push b.
  ○ If you see a C with an A on top of the stack, reduce the epsilon on top of the stack to c (which
    goes back on the stack)
  ○ If you see a C with [A c] on top of the stack, shift the C onto the stack.
  ○ If you see a C with [A c C] on top of the stack, pop the top three elements and then push c.
  ○ If you see end of input with [A c C] on top of the stack, pop the top three elements and then push
    c.
  ○ If you hit the end of the string with b on top of the stack, reduce b to s.
  ○ If you hit the end of the string with c on top of the stack, reduce c to s.
  ○ If you hit the end of the string with s the only value on the stack, accept.
  ○ In any other case, crash and burn.

● Here’s a set of helpful rules for expressions
  ○ (input: NUM, stack: anything) -> shift
  ○ (input: ADDOP, stack: NUM) -> reduce NUM to factor
  ○ (input: ADDOP, stack: term MULOP factor) -> reduce term MULOP factor to term
  ○ (input: ADDOP, stack: factor) -> reduce factor to term
  ○ (input: ADDOP, stack: expression ADDOP term) -> reduce expression ADDOP term to term
  ○ (input: ADDOP, stack: term) -> reduce term to expression
  ○ (input: ADDOP, stack: expression) -> shift
  ○ (input: MULOP, stack: NUM) -> reduce NUM to factor
  ○ (input: MULOP, stack: term MULOP factor) -> reduce term mulop factor to term
  ○ (input: MULOP, stack: factor) -> reduce factor to term
  ○ (input: MULOP, stack: term) -> shift
  ○ There’s some related stuff for end-of-input not shown.

● Here’s an attempt to parse 1 - 2 - 3 * 4 - 5 with that set of rules
Starting state
  - Input: [1 - 2 - 3 * 4 - 5]
  - Stack: []
We shift 1 onto the stack:
  - Input: [- 2 - 3 * 4 - 5]
  - Stack: [NUM(1)]
We reduce the 1 on the stack to a factor
  - Input: [- 2 - 3 * 4 - 5]
  - Stack: [factor(1)]
We reduce the factor on the stack to a term
  - Input: [- 2 - 3 * 4 - 5]
  - Stack: [term(1)]
We reduce the term on the stack to an expression
  - Input: [- 2 - 3 * 4 - 5]
  - Stack: [expression(1)]
We shift the - onto the stack
  - Input: [2 - 3 * 4 - 5]
  - Stack: [expression(1) ADDOP(-)]
We shift 2 onto the stack:
  - Input: [- 3 * 4 - 5]
  - Stack: [expression(1) ADDOP(-) NUM(2)]
We reduce the 2 on the stack to a factor:
  - Input: [- 3 * 4 - 5]
  - Stack: [expression(1) ADDOP(-) factor(2)]
We reduce the factor on the stack to a term:
  - Input: [- 3 * 4 - 5]
  - Stack: [expression(1) ADDOP(-) term(2)]
We reduce the expression ADDOP term on the stack to an expression:
  - Input: [- 3 * 4 - 5]
  - Stack: [expression(1-2)]
We shift the minus onto the stack
  - Input: [3 * 4 - 5]
  - Stack: [expression(1-2) ADDOP(-)]
We shift the 3 onto the stack:
  - Input: [* 4 - 5]
  - Stack: [expression(1-2) ADDOP(-) NUM(3)]
We reduce the 3 to a factor:
  - Input: [* 4 - 5]
  - Stack: [expression(1-2) ADDOP(+)]
We reduce the factor to a term:
  - Input: [* 4 - 5]
  - Stack: [expression(1-2) ADDOP(-) term(3)]
We shift the multiplication symbol onto the stack:
Input: [4 - 5]
Stack: [expression(1-2) ADDOP(-) term(3) MULOP(*)]

- We shift the 4 onto the stack:
  - Input: [- 5]
  - Stack: [expression(1+2) ADDOP(-) term(3) MULOP(*) NUM(4)]
- We reduce the 4 to a factor:
  - Input: [- 5]
  - Stack: [expression(1-2) ADDOP(-) term(3) MULOP(*) factor(4)]
- We reduce the term MULOP factor to a term:
  - Input: [- 5]
  - Stack: [expression(1-2) ADDOP(-) term(3*4)]
- We reduce the expression ADDOP term to an expression:
  - Input: [- 5]
  - Stack: [expression((1-2)-(3*4))] as expression
- We shift the minus onto the stack:
  - Input: []
  - Stack: [expression((1-2)-(3*4)) ADDOP(-)]
- We shift the number onto the stack
  - Input: []
  - Stack: [expression((1-2)-(3*4)) ADDOP(-) NUM(5)]
- We reduce the 5 to a factor:
  - Input: []
  - Stack: [expression((1-2)+(3*4)) ADDOP(-) factor(5)]
- We reduce the factor to a term:
  - Input: []
  - Stack: [expression((1-2)+(3*4)) ADDOP(-) term(5)]
- We reduce the expression ADDOP term to an expression:
  - Input: []
  - Stack: [expression(((1-2)-(3*4))-5)]
- We reduce the expression to the start symbol.

- Note that it seems that we made different decisions in the same context.
  - That is, when we saw an operation, sometimes we shifted the operation and sometimes we reduced the top of the stack.
  - Similarly, when we had a term on top of the stack, sometimes we reduced the term and sometimes we shifted something else onto the stack.
  - But the context is not the same. Decisions are based on what is on the stack and what the next input tokens are.
- In some ways, these parsers can be viewed as finite automata that use a stack (also known as push-down automata).
- Shift reduce parsing is traditionally done with LR(k) parsers. The first L stands for “left-to-right traversal of the input”, the next R stands for “rightmost derivation” and the k stands for “number of characters of lookahead”.
  - You should be able to understand the traversal and lookahead.
  - The rightmost refers to the types of derivations that the grammar represents. In a rightmost
top-down derivation from the start symbol, you always replace the rightmost nonterminal.
● While LR parsers are bottom up, they simulate rightmost top-down derivations

**LR(0) Automata**

● The simplest LR parsers are based on LR automata with no lookahead.
  ○ “Wait!” You may say, “How can we build a parser with no lookahead?”
  ○ It turns out the 0 refers to the lookahead used in constructing the automata, not (in this case) to the lookahead in running them.
● The states of all LR parsers are sets of extended productions (also called items), representing the states of all possible parses of the input (more or less)
  ○ Each production is extended with a position (where in the parse we may be)
  ○ Each production may be extended with lookahead symbols (none in LR(0) automata).
● We begin building an LR(0) parser by augmenting the grammar with a simple rule in which we add an “end of input” ($) to the start symbol.

\[ s' ::= s \\]

● The initial state of an LR(0) parser begins with

\[ s' ::= . s \\]

● This means “when we begin parsing, we are ready to match an S and the end-of-input symbol”
● We then augment this with the other things that we might match on our way to building an S and end of input. What are those things? Anything that might build us an S. What have we seen of those things? Nothing, yet.
● For our expression grammar, we might write

\[
\begin{align*}
\text{start'} & ::= . \text{expr} \\$
\text{expr} & ::= . \text{expr ADDOP term}
\text{expr} & ::= . \text{term}
\text{term} & ::= . \text{term MULOP fact}
\text{term} & ::= . \text{fact}
\text{fact} & ::= . \text{id}
\text{fact} & ::= . \text{num}
\text{fact} & ::= . ( \text{expr} )
\end{align*}
\]
● Of course, in this case, if we’re waiting to see a term, then we also need to add the term rules (and then the fact rules).

\[
\begin{align*}
\text{start'} & ::= . \text{expr} \\$
\text{expr} & ::= . \text{expr ADDOP term}
\text{expr} & ::= . \text{term}
\text{term} & ::= . \text{term MULOP fact}
\text{term} & ::= . \text{fact}
\text{fact} & ::= . \text{id}
\text{fact} & ::= . \text{num}
\text{fact} & ::= . ( \text{expr} )
\end{align*}
\]
● We make new states in the automaton by choosing a symbol (terminal or nonterminal) and advancing the “here mark” (period) over that symbol in all rules, and then filling in the rest.
● For example, if we see an expr in state 0, we could be
\[ \text{start'} ::= \text{expr} . \$ \]
\[ \text{expr} ::= \text{expr} . + \text{term} \]
\[ \text{expr} ::= \text{expr} . - \text{term} \]

- If we see a plus sign in that state, we could only be making progress on one rule, giving us
  \[ \text{expr} ::= \text{expr} + . \text{term} \]
- But now, we’re ready to see a \text{term}, so we need to fill in all the items that say “ready to see a term”
  \[ \text{expr} ::= \text{expr} + . \text{term} \]
  \[ \text{term} ::= . \text{term} \text{mulop} \text{fact} \]
  \[ \text{term} ::= . \text{fact} \]
  \[ \text{fact} ::= . \text{id} \]
  \[ \text{fact} ::= . \text{num} \]
  \[ \text{fact} ::= . ( \text{expr} ) \]
- If we do indeed see a term, we advance the “here mark” and get to
  \[ \text{expr} ::= \text{expr} + \text{term} . \]
  \[ \text{term} ::= \text{term} . \text{mulop} \text{fact} \]
- The first part suggests that we may be at the end of an \text{expr}. The second suggests that we may be in
  the midst of a \text{term}. How do we decide which it is? By context (and a little lookahead in some cases).