Class 22: Shift-Reduce Parsing (2)

Held: Friday, 14 October 2011

Summary: Today we continue our exploration of shift-reduce automata and the tables that represent them.

Related Pages:
- EBoard.

Notes:
- Have a great break!

Overview:
- Building an LR(0) automaton for expressions.
- One version of that automaton.
- Running the automaton.
- Building LR(0) automata.

LR(0) Automata

We started this construction in the previous class, and will continue today.

- 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.
  
  \[ s' ::= s \, s \, $ \]

- 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

```plaintext
start' ::= . expr $
expr ::= . expr ADDOP term
expr ::= . term
```

- 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).

```plaintext
start' ::= . expr $
expr ::= . expr ADDOP term
expr ::= . term
term ::= . term MULOP fact
term ::= . fact
fact ::= . id
fact ::= . num
fact ::= . ( expr )
```

- 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

```plaintext
start’ ::= expr . $
expr ::= expr . + term
expr ::= expr . - term
```

- If we see a plus sign in that state, we could only be making progress on one rule, giving us

```plaintext
expr ::= expr + . term
```

- But now, we’re ready to see a term, so we need to fill in all the items that say “ready to see a term”

```plaintext
expr ::= expr + . term
term ::= . term mulop fact
term ::= . fact
fact ::= . id
fact ::= . num
fact ::= . ( expr )
```

- If we do indeed see a term, we advance the “here mark” and get to

```plaintext
expr ::= expr + term .
term ::= term . mulop fact
```

- The first part suggests that we may be at the end of an expr. The second suggests that we may be in the midst of a term. How do we decide which it is? By context (and a little lookahead in some cases).
The Expression Automaton, Revisited

Our example may not match this precisely, but this is left in the outline for review purposes.

- **State: S00**
  - Rules:
    - `exp ::= . exp + term`
    - `exp ::= . exp - term`
    - `exp ::= . term`
    - `term ::= . term * factor`
    - `term ::= . term / factor`
    - `term ::= . factor`
    - `factor ::= . ID`
    - `factor ::= . NUM`
    - `factor ::= . ( exp )`
  - Edges:
    - `exp -> S01`
    - `term -> S02`
    - `factor -> S03`
    - `ID -> S04`
    - `NUM -> S05`
    - `(-) -> S06`

- **State: S01**
  - Rules:
    - `start ::= exp . EOI`
    - `exp ::= exp . + term`
    - `exp ::= exp . - term`
  - Edges:
    - `EOI -> ACCEPT`
    - `+ -> S08`
    - `- -> S09`

- **State: S02**
  - Rules:
    - `exp ::= term .`
    - `term ::= term . * factor`
    - `term ::= term . / factor`
  - Edges:
    - `* -> S10`
    - `/ -> S11`

- **State: S03**
  - Rules:
    - `term ::= factor .`

- **State: S04**
• Rules:
  • factor ::= ID .

• State: S05
  • Rules:
    • factor ::= NUM .

• State: S06
  • Rules
    • factor ::= ( . exp )
    • exp ::= . exp + term
    • exp ::= . exp - term
    • exp ::= . term
    • term ::= . term * factor
    • term ::= . term / factor
    • term ::= . factor
    • factor ::= . ID
    • factor ::= . NUM
    • factor ::= . ( exp )

  • Edges
    • exp > S14
    • term -> S02
    • factor -> S03
    • ID -> S04
    • NUM -> S05
    • ( -> S06

• State: S08
  • Rules:
    • exp ::= exp + . term
    • term ::= . term * factor
    • term ::= . term / factor
    • term ::= . factor
    • factor ::= . ID
    • factor ::= . NUM
    • factor ::= . ( exp )

  • Edges:
    • term -> S16
    • factor -> S03
    • ID -> S04
    • NUM -> S05
    • ( -> S06

• State: S09
  • Rules
    • exp ::= exp - . term
    • term ::= . term * factor
• term ::= . term / factor
• term ::= . factor
• factor ::= . ID
• factor ::= . NUM
• factor ::= . ( exp )
○ Edges:
  • term -> S17
  • factor -> S03
  • ID -> S04
  • NUM -> S05
  • ( -> S06

● State: S10
  ○ Rules:
    • term ::= term * . factor
    • factor ::= . ID
    • factor ::= . NUM
    • factor ::= . ( exp )
  ○ Edges:
    • factor -> S12
    • ID -> S04
    • NUM -> S05
    • ( -> S06

● State: S11
  ○ Rules:
    • term ::= term / . factor
    • factor ::= . ID
    • factor ::= . NUM
    • factor ::= . ( exp )
  ○ Edges:
    • factor -> S13
    • ID -> S04
    • NUM -> S05
    • ( -> S06

● State: S12
  ○ Rules:
    • term ::= term * factor

● State: S13
  ○ Rules:
    • term ::= term / factor

● State: S14
  ○ Rules
    • factor ::= ( exp )
    • exp ::= exp + term
• exp ::= exp . - term
  ○ Edges
    • ) -> S15
    • + -> S08
    • - -> S09
  ○ State: S15
    ○ Rules
      • factor ::= ( exp ) .
  ○ State: S16
    ○ Rules:
      • exp ::= exp + term .
      • term ::= term . * factor
      • term ::= term . / factor
    ○ Edges:
      • * -> S10
      • / -> S11

Using the Automaton

• The stack keeps track of the state of the automaton and the partially processed trees.
• When you see a symbol that corresponds to an edge, shift that symbol and push the new state.
• When a state contains a completed production and you see a symbol in Follow[lhs], pop the state/symbol pairs and reduce. If all goes well, you can then follow an edge based on the lhs.
• Example in class.

Constructing LR(0) Automata, Formalized

• Let us now formalize what we did.
• We need a method that “fills in the rest” whenever we advance the mark. We’ll call this closure

```
closure(State S)
  repeat
    for each item, n ::= alpha . m beta in S
      for each production M ::= gamma
        add m ::= . gamma to S
      end for each production
    end for each item
  until no changes are made to S
  return S
end closure
```

• We need a method that describes the edges in the automata. We’ll call this goto
goto(State S, Symbol sym)
  newS = {}
  for each item of the form (n ::= alpha . sym beta) in S
    newS = newS union { N ::= alpha sym . beta }
  end for
  return closure(newS)
end goto

- Finally, we can build the automaton as follows

  S0 = { s’ ::= . s $ }
  S0 = closure(S0);
  while there are unmarked states
    pick an unmarked state, S
    mark S
    for each symbol, sym, add state goto(S,sym) with edge labelled s
  end while