Class 23: Shift-Reduce Parsing (3)

Held: Monday, 24 October 2011

Summary: We conclude our discussion of shift-reduce parsing.

Related Pages:

- EBoard.

Notes:

- The mid-semester examination will be held on Wednesday.
- No EC for this week’s Thursday extra: Discrete Structures and the future of math requirements in CS.
- Due Friday: Phase 2 of your project.
- Reading for Friday: 5.1-5.3.
- EC for any of the Grinnell College Young Innovator for Social Justice events (Awards ceremony Tuesday night; Coffee break on Wednesday at 2:30; Symposium on Wednesday at 4:15; Symposium Wednesday at 8:00 p.m.; Morris Dees Convo).

Overview:

- Detour: About the Mid-Semester Examination.
- Refresher: Shift-Reduce Parsers.
- Some Potential Problems with LR(0) Parsers.
- Other Techniques for Computing Shift-Reduce Automata.
- Parser Geerators.

Detour: The Mid-Semester Examination

- The exam has four questions.
  - It takes me under ten minutes to answer those questions.
- You may bring one hand-written, double-side 8.5x11 inch page of notes to the exam. The exam is otherwise closed book and closed computer.
- The exam is on all of the things we’ve done to date. That includes:
  - The phases of a compiler.
  - Lexical Analysis
    - Regular expressions.
    - Grep and lex.
    - Finite automata.
    - Turning regular expressions into NFA’s.
    - Turning NFA’s into DFA’s.
    - Optimizing DFA’s.
Syntactic Analysis
- BNF grammars
- Hand-coded predictive parsers
- Helpful tables (First, Follow, Nullable)
- Building predictive parsers
- LR(0) parsers

Tools
- Grep
- f?lex
- Bison/Yacc

Pascal
- Tokens, etc.

Miscellaneous
- Languages not amenable to lexical analysis

Some kinds of questions
- End of section questions from Aho et al.
- Write a regular expression for this language or explain why you cannot write such a regular expression.
- Write a BNF grammar for this language
- What language does this regular expression describe?
- What language does this grammar describe?
- This regular expression is supposed to describe this language. Does it? Why or why not?
- This grammar is supposed to describe this language. Does it? Why or why not?
- Translate from this form to this form
  - Regular expression to NFA
  - NFA to DFA
  - DFA to optimal DFA
  - Regular expression to BNF grammar
  - Grammar to unambiguous grammar
  - Grammar to predictive-parsable grammar

Build this table for this grammar
- First, Follow, Nullable
- Predictive parsing table
- LR(0) parsing table

Shift-Reduce Automata, Reviewed
- A bottom-up technique.
- We keep a stack of symbols (nonterminals and terminals) and states.
- We use a table indexed by (state,token) to determine what to do next.
  - Shift the token onto the stack and enter a new state
  - Reduce the RHS on the top of the stack (and then use the nonterminal to shift into a new state)
- We currently know one technique for building shift-reduce tables: LR(0).
Conflicts in LR Automata

- At times, there are conflicts in LR automata. What kinds of conflicts?
- A state may include a “final” item (one in which the position marker is at the end) and a nonfinal item.
  - This is called a shift-reduce conflict
- A state may include two different “final” items.
  - This is called a reduce-reduce conflict
  - Can we have reduce-reduce conflicts in unambiguous grammars?
- Can we have a shift-shift conflict?

SLR Automata

- You may have noted (e.g., from our example in the previous class) that LR(0) automata can be overly aggressive in choosing to reduce.
  - Such automata typically reduce whenever we’ve reached the end of a right-hand side.
- SLR automata only reduce when the next token is in Follow of the left-hand side of the reduction.
- Such choices may help us resolve reduce-reduce conflicts.
- However, we still have similar conflicts.
  - Shift-reduce conflicts include the case in which Follow of the final lhs of the final item overlaps with first of the remainder
  - Reduce-reduce conflicts include the case Follow of both left-hand-sides overlap.
- Nonetheless, SLR automata are commonly used in parser generators.

LR(1) Automata and Beyond

- LR(1) automata require a more complicated construction process, one that involves lookahead.
- In effect, instead of using the Follow table (as SLR automata do), LR(1) automata build more specific follow tables that correspond to the possible follow symbols according to a particular context.
- Each LR(1) item contains not just an augmented production, but also a token that can follow the nonterminal when we’ve reached the current state.
- The tokens are inserted by the closure routine.
- Given a state with $n : Stuff . m MoreStuff$,
  - When we insert the $m$ rules into the state, we indicate that each of them must be followed by a token in first ($MoreStuff$).
  - If $MoreStuff$ is nullable, then the $m$ items can also be followed by whatever can follow $n$ (in the given LR(1) item).
- That’s all we’re covering in this class.
  - There are also LR(k) parsers (k tokens of lookahead)
- Most parser generators use a variant of LR(1) called LALR(1).
  - Instead of including the follow tokens in the rules, we post-compute them based on the LR(0) automaton.
Such automata are much smaller than LR(1) automata.
However, they are even more of a pain to describe carefully.

**Parser Generators**

- As you might guess, parser generators provide a popular way to handle the translation of a grammar to a parser.
- In Yacc and Bison, we also get to associate a value with each symbol (and that value gets stored on the stack with the symbol and state).
- As in the case of f?lex, we can also put code in the rules.
  - Code is treated like an "always accept" token: When you get to it, you execute it.
- While you can put code anywhere in the rule, it is safest to put it at the end of the rule.