CS143
Compilers

Lecture 5
July 16, 2001

Canonical LR Parsing

- In SLR parsing tables, a reduction $A \rightarrow \alpha$ will occur in any state whose set of LR(0) items contains $[A \rightarrow \alpha\bullet]$, when the next input symbol a is in FOLLOW(A).
- However, this will be an incorrect action if the viable prefix $\beta\alpha$ that is currently on top of the stack, if replaced by βA, cannot be followed by a without causing a syntax error.
- Canonical LR parsing avoids this problem by taking into consideration which terminals can follow given grammar symbols in a *right-sentential form*, i.e. a string of grammar symbols that can be derived from the start symbol S.

LR(1) Items

- An LR(1) item is an LR(0) item $[A \rightarrow \alpha\bullet\beta]$, along with a terminal symbol a, which is called the *lookahead*.
- Such an item is written as $[A \rightarrow \alpha\bullet\beta, a]$.
- An LR(1) items can be viewed as a state in an NFA that implements a canonical LR parser. Reaching such a state implies that we have recognized α in the input, we may next recognize β in the input, and the viable prefix $\alpha\beta$, if replaced by the viable prefix αA, may be followed by a.
- We will see that constructing sets of LR(1) items is very similar to building sets of LR(0) items.

LR(1) Closure

function closure(I)
repeat
  for each item $[A \rightarrow \alpha\bullet B\beta, a]$ in I
    for each production $B \rightarrow \gamma$ in G
      for each terminal b in FIRST(βa) such that $[B \rightarrow \bullet\gamma, b]$ is not in I
        add $[B \rightarrow \bullet\gamma, b]$ to I
until no more items can be added to I
return I

LR(1) Goto

function *goto*(I, X)
let J = { }
for each item $[A \rightarrow \alpha\bullet X\beta, a]$ in I
  add the item $[A \rightarrow \alpha X\bullet\beta, a]$ to J
return *closure*(J)

Constructing sets of LR(1) Items

procedure *items*(G')
  C = { *closure*({ $[S' \rightarrow \bullet S, \$]$ }) }
  repeat
    for each set of items I in C and each grammar symbol X such that *goto*(I, X) is not empty and not in C
      add *goto*(I, X) to C
  until no more sets of items can be added to C

LR Parsing tables

- We are now ready to construct a canonical LR parsing table for a grammar G. The construction is very similar to that of SLR tables.
- A grammar for which the canonical LR table does not have conflicting states is said to be LR(1).
- In practice, LR tables have far more states than SLR tables. However, the set of LR(1) grammars is much larger than the set of SLR(1) grammars, and can be used to express commonly used constructs that cannot be conveniently expressed by an SLR(1) grammar.

- Construct the collection of LR(1) items for G'; $C = \{ I_0, I_1, \ldots, I_n \}$
- For each I_i
  - If $[A \rightarrow \alpha \bullet a \beta, b]$ is in I_i and $goto(I_i, a) = I_j$ then set $action[i,a]$ to "shift j"
  - If A is not S' and $[A \rightarrow \alpha \bullet, a]$ is in I_i then set $action[i,a]$ to "reduce $A \rightarrow \alpha$" .
  - If $[S' \rightarrow S \bullet]$ is in I_i then set $action[i,\$]$ to "accept"
  - For each nonterminal A, if $goto(I_i, A) = I_j$ then $goto[i, A]=j$
  - All other entries are "error"
- The initial state is the state constructed from the set of items containing $[S' \rightarrow \bullet S]$

Constructing LALR Parsing Tables

- While canonical LR parsers can handle many more grammars than SLR parsers, they also have parsing tables with far more states.
- LALR parsing is an attempt at achieving the best of both worlds. An LALR parsing table always has as many states as an SLR table, and an LALR parser can handle more grammars than an SLR parser.
- While the class of LR(1) grammars is still larger than the class of LALR(1) grammars, most common constructs can be handled by an LALR parser, which is the type of parser often generated by parser generators.

Merging sets of LR(1) Items

- The core of a set of LR(1) items I is the set of LR(0) items obtained by removing the lookaheads from each item in I.
- Construction of an LALR parsing table can proceed by obtaining the canonical collection of sets of LR(1) items, and then merging sets that have the same core.
- This reduces the number of states to the number of states than an SLR parser would have. Such merging can introduce a reduce/reduce conflict, but can never introduce a shift/reduce conflict.

A simple LALR parsing table construction

- Construct the collection of sets of LR(1) items, C = { I0, I1, …, In }
- For each distinct core within C, find all sets that have that core and merge them into a single set, obtaining a new collection C' = { J0, J1, …, Jm }
- Construct a canonical LR parsing table from C'. If a conflict occurs in the action table, then the grammar is not LALR(1).

Efficient Construction of LALR Parsing Tables

- While the previous method of constructing LALR tables is conceptually simple, it is also very inefficient since several sets of items are constructed and later merged. For a typical programming language, the wasted space representing these sets is enormous.
- Another way to proceed is to compute the *kernels* of the sets of LR(0) items and then determine the proper lookahead symbols to include in each set. The *kernel* of a set of LR(0) items is the subset of I containing all items that are either the initial item [$S \rightarrow S'$], or do not have the dot at the beginning of the right side.

Using Ambiguous Grammars

- An LR parser cannot be used with a grammar that is ambiguous. However, converting an ambiguous grammar to an equivalent unambiguous grammar can greatly increase the number of states in an LR parsing table.
- Therefore, it is desirable to construct LR parsing tables directly from the original ambiguous grammar, and then use rules of precedence and associativity in order to eliminate conflicts.

Using Precedence and Associativity

- Precedence and associativity are often used to resolve shift-reduce conflicts in a parsing table.
- If the next input symbol is some operator *a*, and a shift-reduce conflict occurs where the handle is consists of another operator *b* and its operands, these rules of thumb can be used:
  - If *a* has higher precedence than *b*, then shift.
  - If *a* has lower precedence than *b*, then reduce.
  - If *a* and *b* have equal precedence, then shift if right assocative, and reduce if left associative.

Error Recovery in LR Parsing

- As with top-down parsers, LR parsers can recover from syntax errors using panic-mode recover or phrase-level recovery.
- To implement panic mode, a set of synchronizing nonterminals can be chosen for each error entry in the table. We can scan the stack until we find a state *s* that has a goto on a nonterminal *A* from this set. Then, we discard input symbols until a symbol that can follow *A* is found, at which time we push the state goto[*s*,*A*] onto the stack and resume parsing.