CS143 Compilers

Lecture 4

July 11, 2001

Bottom-Up Parsing

• Bottom-up parsing is a method of parsing input in which the parse tree is built up from the leaves.

• Parsing in this manner yields a rightmost derivation of the input string from the start symbol.

• We say that the input string is reduced to the start symbol. Each step in the rightmost derivation, starting from the original input string and proceeding in reverse, is called a *reduction*.

• A reduction by the production $A \rightarrow \beta$ is a derivation satisfying $S \Rightarrow^*_{rm} \alpha A\gamma \Rightarrow_{rm} \alpha\beta\gamma$, where the *rm* subscript denotes a rightmost derivation and α and γ are strings of grammar symbols.

Handles

• A handle of a string w of grammar symbols is a substring α that matches the right side of a production $A \rightarrow \beta$ whose replacement by A represents a step in the reverse of a rightmost derivation of w from the start symbol S. Specifically, $S \Rightarrow^*_{rm} \alpha A\gamma \Rightarrow^*_{rm} \alpha\beta\gamma \Rightarrow^*_{rm} w$, where α and γ are strings of grammar symbols and w is the input string.

• If the grammar is not ambiguous, then each step of a rightmost derivation corresponds to exactly one handle.

Handle Pruning

• The process of obtaining a rightmost derivation in reverse from an input string is called “handle pruning”.

• Handle pruning proceeds by locating the handle β in the input string and replacing β by some nonterminal A such that $A \rightarrow \beta$. This replacement yields a new string of grammar symbols.

• This process is repeated on the new string until it is reduced to the start symbol S, at which time we have obtained a rightmost derivation.

Stack-Implementation of Shift-Reduce Parsing

• We now describe a method of bottom-up parsing called *shift-reduce parsing*.

• Shift-reduce parsing uses a stack to hold grammar symbols. As the input is scanned from left to right, input symbols are *shifted* onto the stack.

• When a handle appears on top of the stack, it is *reduced* to the appropriate nonterminal, which replaces the handle on the stack.

• If the start symbol appears on top of the stack, the input is *accepted*, and parsing is complete.

Why use a stack?

• A key aspect of shift-reduce parsing is that a handle will always be found on top of the stack that shift-reduce parsing maintains.

• Once a reduction is made and a handle is replaced by a nonterminal, zero or more symbols must be shifted onto the stack in order to construct the next handle. This makes a stack a very convenient data structure to use.

• Valid strings of grammar symbols that can appear on top of the stack will play a central role in building shift-reduce parsers.

Viable Prefixes

- Given an input string *w*, a *viable prefix* is a string of grammar symbols α satisfying the following properties:
  - $S \Rightarrow^*_{rm} \alpha\beta \Rightarrow^*_{rm} w$
  - α does not extend past the rightmost handle in the string $\alpha\beta$
- It follows from this definition that a viable prefix can appear on the top of the stack during parsing.
- There is no error as long as the input that has been scanned can be reduced to a viable prefix.

Conflicts During Shift-Reduce Parsing

Two types of conflicts can occur during shift-reduce parsing:

- A *shift/reduce conflict* occurs when the parser cannot determine whether it should shift the next input symbol onto the stack, or reduce the handle on top of the stack.
- A *reduce/reduce* conflict occurs when a handle is on top of the stack, but the parser cannot determine which production to use for the reduction.

LR Parsers

We now tackle the fundamental problem in shift-reduce parsing--recognizing when to shift or reduce:

- Knowing when you have a viable prefix, so that more terminals may be shifted onto the stack
- Knowing when you have a handle on top of the stack, so that you may perform a reduction

To this end, we use a technique called LR(*k*) parsing, where L denotes left-to-right scanning, R denotes rightmost derivation, and *k* is the number of input symbols of lookahead.

Advantages of LR Parsing

LR parsing is the most commonly used method, for good reason:

- It can be used to recognize nearly all programming-language constructs that can be described by a context-free grammar.
- Although it is more powerful than other shift-reduce parsing methods, it is just as efficient.
- It can be used with all LL(1) grammars, and many grammars that are not LL(1).
- It can detect a syntax error as soon as it is possible to do so.

Building LR Parsers

The primary drawback of LR parsing is that it is extremely difficult to build an LR parser for a typical programming language by hand. However, many tools are available that can implement LR parsers automatically, such as yacc or bison.

We will discuss three methods for building LR parsers:

- SLR (simple LR), which is the easiest and least powerful
- canonical LR, the most powerful and most expensive. This is used by yacc.
- LALR (lookahead LR), intermediate in power and cost, and most commonly used.

Anatomy of an LR Parser

- An LR parser consists of these components:
  - an input buffer that holds the input string and the end-of-input marker $
  - a stack that holds grammar symbols and parsing states
  - a parsing table that contains two parts, *action* and *goto*. This table is used to guide parsing decisions
  - a driver program that manages these components
- Only the parsing table varies from one LR parsing technique to another.

The LR Parsing Table

- The parsing table consists of two parts, called action and goto.
- *action* is indexed by a set of states, and the terminal symbols of the grammar. The driver program uses the state currently on top of the stack and the current input symbol as indices into this table to determine its course of action.
- *goto* is indexed by the set of states and the nonterminal symbols. After a reduction, the driver program uses the state on top of the stack and the left side of the production used for the reduction as indices into this table to determine the new state that should be on top of the stack.

LR Parsing Actions

Entries in the action table can be one of the following:

– shift the current input symbol onto the stack, followed by the state indicated by the entry

– reduce using the production indicated by the entry. The grammar symbols on the right side are on the stack, with accompanying states. These symbols and states are popped, and the nonterminal on the left side is pushed, along with a state indicated by the goto table.

– accept the input and terminate parsing

– report a syntax error

The LR Parsing Algorithm

- The LR parsing algorithm takes as input a string *w* and an LR parsing table with functions *action* and *goto* for a grammar *G*.
- The algorithm outputs a sequence of reductions that can be used to build the parse tree for *w*, corresponding to a rightmost derivation.
- Initially, the parser has the initial state s_0 on its stack, and *w*$ in the input buffer.
- The driver program then proceeds as described on the next slide:

```
repeat
    let s be the state on top of the stack
    let a be the current input symbol
    if action[s,a] = shift s'
        push a and s' onto the stack
        advance to the next input symbol
    else if action[s,a] = reduce A → β
        pop 2|β| symbols off of the stack
        let s' be the state now on top of the stack
        push A and goto[s',A] onto the stack
        output A → β
    else if action[s,a] = accept, return
    else error
```

LR Grammars

- A grammar for which one can construct an LR parsing table without multiply-defined entries is said to be an LR grammar.
- LR grammars can describe more languages than LL grammars because LR parsers have more information available for making parsing decisions than LL parsers.
- An LR parser must be able to recognize the right side of a production having seen the entire string that is derived from that right side, while an LL parser must recognize the use of a production having seen only a portion of what is derived from its right side.

Constructing SLR Parsing Tables

- The last ingredient in our LR parser is the table. We now address the issue of how this table is to be constructed.
- We begin with an SLR table, the simplest to construct.
- The construction entails building an NFA that can recognize viable prefixes, and converting this NFA into a DFA.
- Specifically, we create objects called LR(0) items, which correspond to NFA states, and construct sets of items, which correspond to DFA states.

LR(0) Items

- Given a grammar *G*, an LR(0) item is a production of *G* along with a position of the right side, indicated by a dot.
- An item indicates how much of a production we have seen at a given point in the parsing process.
- When the dot is at the far left of the right side, it means that we haven't seen recognized any of the right side yet. When it is on the far right, it means we have scanned a string derived from the right side and can perform a reduction. In general, we have seen a string derived from the portion of the right side that is to the left of the dot.

The Closure Operation

- The *closure* operation is similar to the ε-*closure* operation used to convert NFA's into DFA's. It accepts a set of LR(0) items *I* as input, which corresponds to a set of NFA states, and computes a new set of items.
- The items in this new set denote all strings of grammar symbols that we can expect to see in the input, based on the state of the parser defined by the items in *I*.

Computing Closure

```
function closure(I)
    J = I
    repeat
        for each item $A \to \alpha \bullet B \beta$, and each production
      $B \to \gamma$ such that $B \to \bullet \gamma$ is not in J,
            add $B \to \bullet \gamma$ to J
    until no more items can be added to J
    return J
```

The Goto Operation

- The goto operation corresponds to the construction of a DFA state from a set of states and the move function of an NFA. It accepts a set of items *I* and a grammar symbol *X* as input, and returns a new set of items *goto*(*I*,*X*).
- *goto*(*I*,*X*) is the closure of the set of items obtained from I by moving the dot over *X*, where applicable. This corresponds to making a transition from *I* on *X* in the DFA that recognizes the language of the gramar.

The Sets-of-Items Construction

- Now we are ready to construct a collection of sets of LR(0) items that represent the states in our DFA. This process is analogous to subset construction.
- Before beginning, we augment the grammar by adding a new start symbol *S'* and a production *S'* → *S*. This effectively defines the initial and accepting states of our DFA.
- The following slide details the construction of the *canonical collection C* of sets of items.

Algorithm

```
procedure items(G')
    C = { closure({ [$S' \to \bullet S$] }) }
    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
```

Valid Items

- An LR(0) item is said to be *valid* for a viable prefix if:
  - the viable prefix ends with the portion of the right side of the item's production to the left of the dot.
  - there is a rightmost derivation of a string beginning with this viable prefix from S' that uses the item's production.
- Valid items can serve as a guide in parsing decisions, thus helping to fill in the entries of a parsing table.
- Given a DFA built from sets of items, the set of items reached from the initial state along a path labeled with a viable prefix γ is exactly the set of valid items for γ.

SLR Parsing Tables

- We will now use the sets of items constructed earlier to fill in the entries in an SLR parsing table.
- Each set of items is valid for a set of viable prefixes.
- Terminal symbols can always be added to a viable prefix to obtain a valid string that can be derived from the start symbol. Therefore, items with a dot before a terminal produce shift actions in the parsing table.
- A viable prefix that ends with a handle corresponds to an item with a dot at the end of the right side. Such items produce reduce actions in the parsing table.
- The next slide details the construction of the parsing table.

- Construct the collection of LR(0) items for G'; $C = \{ I_0, I_1, \ldots, I_n \}$
- For each I_i
  - If $[A \rightarrow \alpha \bullet a \beta]$ 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 $[A \rightarrow \alpha \bullet]$ is in I_i then set *action*[i,a] to "reduce $A \rightarrow \alpha$" for all a in FOLLOW(A).
  - 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 $[S' \rightarrow \bullet S]$