9

Bottom Up Parsing

Motivation

- In the last lecture we looked at a table driven, top-down parser
  - A parser for LL(1) grammars
- In this lecture, we’ll look a a table driven, bottom up parser
  - A parser for LR(1) grammars
- In practice, bottom-up parsing algorithms are used more widely for a number of reasons

Right Sentential Forms

Consider this example

- We start with id+id*id
- What rules can apply to some portion of this sequence?
  - Only rule 6: F → id
- Are there more than one way to apply the rule?
  - Yes, three
- Apply it so the result is part of a “right most derivation”
  - If there is a derivation, there is a right most one
  - If we always choose that, we can’t get into trouble

Bottom up parsing

- A bottom up parser looks at a sentential form and selects a contiguous sequence of symbols that matches the RHS of a grammar rule, and replaces it with the LHS
- There might be several choices, as in the sentential form E+T*F
- Which one should we choose?

Bottom up parsing

- If the wrong one is chosen, it leads to failure
- E.g.: replacing E*T with E in E+T*F yields E+F, which can’t be further reduced using the given grammar
- The handle of a sentential form is the RHS that should be rewritten to yield the next sentential form in the right most derivation
Sentential forms

- Think of a sentential form as one of the entries in a derivation that begins with the start symbol and ends with a legal sentence.
- It's like a sentence but it may have unexpanded non-terminals.
- We can also think of it as a parse tree where some leaves are as yet unexpanded non-terminals.

Handles

- A handle of a sentential form is a substring α such that:
  - α matches the RHS of some production A → α; and
  - replacing α by the LHS A represents a step in the reverse of a rightmost derivation of s.
- For this grammar, the rightmost derivation for the input abbcde is:
  1: S => aABe => aAde => aAbcde => abbcde
- The string aAbcde can be reduced in two ways:
  1: aAbcde => aAde (using rule 2)
  2: aAbcde => aAbcBe (using rule 4)
- But (2) isn't a rightmost derivation, so Abc is the only handle.

Phrases

- A phrase is a subsequence of a sentential form that is eventually "reduced" to a single non-terminal.
- A simple phrase is a phrase that is reduced in a single step.
- The handle is the leftmost simple phrase.

On to shift-reduce parsing

- How to do it w/o having a parse tree in front of us?
- Look at a shift-reduce parser - the kind that yacc uses.
- A shift-reduce parser has a queue of input tokens & an initially empty stack. It takes one of 4 possible actions:
  - Accept: if the input queue is empty and the start symbol is the only thing on the stack
  - Reduce: if there is a handle on the top of the stack, pop it off and replace it with the rule’s LHS
  - Shift: push the next input token onto the stack
  - Fail: if the input is empty and we can’t accept
- In general, we might have a choice of (1) shift, (2) reduce, or (3) maybe reducing using one of several rules
- The algorithm we next describe is deterministic
**Shift-Reduce Algorithms**

A shift-reduce parser scans input, at each step decides to:

* **Shift** if there is only $S$ on the stack and no input
* **Reduce** if a handle is on top of stack, shift otherwise

- **Succeed** if there is only $S$ on the stack and no input
- **Reduce** conflict: can't decide whether to shift or to reduce
  - **Example**: "dangling else"
  
  ```
  Stmt -> if Expr then Stmt
  | if Expr then Stmt else Stmt
  | ...
  ```

- **Reduce-reduce conflict**: can't decide which of several possible reductions to make
  - **Example**: 
    ```
    Stmt -> id ( params )
    | Expr := Expr
    | ...
    Expr := id ( params )
    ```

- **Given the input $a(i, j)$ the parser does not know whether it is a procedure call or an array reference.**
### Parser actions

Initial configuration: (S0, a1...an$)

- **1 If action[S0, a1] = Shift S, the next configuration is**:
  $$S_0X_1S_1X_2...X_nS_nS_{n+1}A_{n+2}...S$$

- **2 If action[S0, a1] = Reduce A$\rightarrow\beta$ and S = GOTO[S0, a1]**, where $r$ = the length of $\beta$, the next configuration is:
  $$S_0X_{r}S_{r+1}X_{r+2}S_{r+3}...S_{n+1}AS, a_{r+2}...a_{n+1}S$$

- **3 If action[S0, a1] = Accept, the parse is complete and no errors were found**

- **4 If action[S0, a1] = Error, the parser calls an error-handling routine**

### Example

**Stack**

<table>
<thead>
<tr>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Id + $</td>
<td>Shift 5</td>
</tr>
<tr>
<td>0 Id</td>
<td>Reduce 6</td>
</tr>
<tr>
<td>0 F 3</td>
<td>Reduce 4</td>
</tr>
<tr>
<td>0 E 1</td>
<td>Reduce 1</td>
</tr>
</tbody>
</table>

**Accept**

### Yacc as a LR parser

- The Unix `yacc` utility is just such a parser.
- It does the heavy lifting of computing the table.
- To see the table information, use the `--v` flag when calling `yacc`, as in `yacc --v test.y`.