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 at 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

- Recall the definition of a derivation and a rightmost derivation
- Each of the lines is a right sentential form
- A form of the parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation

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?

- 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
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.

A handle of a sentential form is a substring \( \alpha \) such that:
- \( \alpha \) matches the RHS of some production \( A \rightarrow \alpha \); and
- replacing \( \alpha \) 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 \( \text{abcede} \) is:
\[
S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde
\]

The string \( \text{abcede} \) can be reduced in two ways:
1. \( \text{aAbe} \Rightarrow \text{aAde} \Rightarrow \text{aAbcde} \Rightarrow \text{abcde} \) (using rule 2)
2. \( \text{abbcde} \Rightarrow \text{abcdbe} \) (using rule 4)

But (2) is not a rightmost derivation, so Abc is the only handle.

Note: the string to the right of a handle will only contain terminals (why?)

For sentential form \( E+T*id \) what are the phrasing:
- \( E+T*id \)
- \( T*id \)
- \( id \)
- \( id*id \)

• 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 RHS
  - Shift: push the new 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 next token to top of parse stack (along with state info) or
• Reduce the stack by POPing several symbols off the stack (& their state info) and PUSHing the corresponding non-terminal (& state info)

The stack is always of the form
\[ S_0 \ X_1 \ S_1 \ X_2 \ S_2 \ldots X_n \ S_n \]

- A reduction step is triggered when we see the symbols corresponding to a rule’s RHS on the top of the stack

LR parser table
LR shift-reduce parsers can be efficiently implemented by precomputing a table to guide the processing

When to shift, when to reduce
• Key problem in building a shift-reduce parser is deciding whether to shift or to reduce
  – repeat: reduce if a handle is on top of stack, shift otherwise
  – Succeed if there is only S on the stack and no input
• A grammar may not be appropriate for a LR parser because there are conflicts, which can not be resolved
• Conflict occurs when the parser can’t decide whether to:
  – shift or reduce the top of stack (a shift/reduce conflict), or
  – reduce the top of stack using one of two possible productions (a reduce/reduce conflict)
• There are several varieties of LR parsers (LR(0), LR(1), SLR and LALR), with differences depending on amount of lookahead and on construction of the parse table

Conflicts
Shift-reduce conflict: can’t decide whether to shift or to reduce
• Example : "dangling else"
  Stmts -> if Expr then Stmts | if Expr then Stmts else Stmts | ...
  What to do when else is at the front of the input?
Reduce-reduce conflict: can’t decide which of several possible reductions to make
• Example :
  Stmts -> 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.

LR Table
• An LR configuration stores the state of an LR parser
  \( S_0 X_1 S_1 X_2 S_2 \ldots X_n S_n \ a_{a_1 \ldots a_n} S \)
• LR parsers are table driven, where the table has two components, an ACTION table and a GOTO table
• The ACTION table specifies the action of the parser (shift or reduce) given the parser state and next token
  – Rows are state names; columns are terminals
• The GOTO table specifies which state to put on top of the parse stack after a reduce
  – Rows are state names; columns are non-terminals
### Example

#### Stack | Input | Action
---|---|---
0 | id id id $ | Shift 5
1 | E | Reduce 6 goto(5,E)
2 | E | Reduce 6 goto(5,E)
3 | T | Reduce 2 goto(5,T)
4 | T | Reduce 2 goto(5,T)
5 | F | Reduce 3 goto(5,F)
6 | F | Reduce 3 goto(5,F)
7 | F | Reduce 3 goto(5,F)

#### 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

### Parser actions

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

**Parser actions:**
1. If ACTION[S_m, a_i] = Shift S, the next configuration is: (S_m X_1 S_{m+1} X_{m+2} ... X_{m+r} S_{m+r} a_i ... a_n $)
2. If ACTION[S_m, a_i] = Reduce A and S = GOTO[S_m-r, A], where r is the length of S, the next configuration is: (S_m X_1 S_{m+1} X_{m+2} ... X_{m-r} S_{m-r} A S a_i ... a_n $)
3. If ACTION[S_m, a_i] = Accept, the parse is complete and no errors were found
4. If ACTION[S_m, a_i] = Error, the parser calls an error-handling routine

#### Table

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S5</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>S6</td>
<td>accept</td>
</tr>
<tr>
<td>2</td>
<td>R2</td>
<td>S7 R2</td>
</tr>
<tr>
<td>3</td>
<td>R4</td>
<td>R4 R4</td>
</tr>
<tr>
<td>4</td>
<td>S5</td>
<td>R6 R6</td>
</tr>
<tr>
<td>5</td>
<td>S5</td>
<td>S4 S4</td>
</tr>
<tr>
<td>6</td>
<td>S5</td>
<td>S4 S4</td>
</tr>
<tr>
<td>7</td>
<td>S5</td>
<td>S4 S4</td>
</tr>
<tr>
<td>8</td>
<td>S6</td>
<td>S11</td>
</tr>
<tr>
<td>9</td>
<td>R1</td>
<td>S7 R1 R1</td>
</tr>
<tr>
<td>10</td>
<td>R3</td>
<td>R3 R3</td>
</tr>
<tr>
<td>11</td>
<td>R5</td>
<td>R5 R5</td>
</tr>
</tbody>
</table>

#### Table

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S5</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>S6</td>
<td>accept</td>
</tr>
<tr>
<td>2</td>
<td>R2</td>
<td>S7 R2</td>
</tr>
<tr>
<td>3</td>
<td>R4</td>
<td>R4 R4</td>
</tr>
<tr>
<td>4</td>
<td>S5</td>
<td>R6 R6</td>
</tr>
<tr>
<td>5</td>
<td>S5</td>
<td>S4 S4</td>
</tr>
<tr>
<td>6</td>
<td>S5</td>
<td>S4 S4</td>
</tr>
<tr>
<td>7</td>
<td>S5</td>
<td>S4 S4</td>
</tr>
<tr>
<td>8</td>
<td>S6</td>
<td>S11</td>
</tr>
<tr>
<td>9</td>
<td>R1</td>
<td>S7 R1 R1</td>
</tr>
<tr>
<td>10</td>
<td>R3</td>
<td>R3 R3</td>
</tr>
<tr>
<td>11</td>
<td>R5</td>
<td>R5 R5</td>
</tr>
</tbody>
</table>