I. Top Down vs Bottom Up
   A. Basic idea of top-down parsing
      1. Build the tree from the root down, left to right
      2. Use the next few tokens to predict which production to use next
      3. This was the earliest form of parsing
   B. Pros
      1. It is fast (linear time w/o backup)
      2. It is easy to understand and implement
      3. It is easy to automate
   C. Cons
      1. More difficult to build the grammar
      2. Must be able to predict using first few tokens
         • Generally this works because programs are meant to be read
         • Doesn’t work well with expressions
      3. Can’t have left recursion
      4. Not that good for expressions
   D. Basic idea of bottom-up parsing
      1. Build up the parse tree from the leaves
      2. Decide higher level productions only after you have a sequence of lower level items
      3. This is less intuitive (until you understand it)
      4. This is somewhat more complex
      5. But is it just as fast and handles a wider range of grammars

II. Operator Precedence Parsing
   A. As a simple example, let’s look at parsing expressions
   B. Classic approach uses 2 stacks
      1. One for values (or expressions)
      2. One for operators
   C. Basic algorithm
      1. Read the next token
2. If non-operator, push onto value stack, else
3. Pop operator & values off stacks as appropriate
   • Op stack done by comparing token and item on the operator stack
   • Value stack done based on operators popped off stack
   • New value is then pushed onto the stack if operator done
4. Push operator onto stack if appropriate

D. Example: 1+2*3+(4-5)

E. Deciding when to pop or push
   1. Based on precedence
   2. Can be extended to handle unary ops, (), calls, etc.

F. Fast and easy manual compiler
   1. Use operator precedence for expressions
   2. Use top-down (recursive descent) parsing for the rest

G. Can be done with a single stack
   1. A little more complex – need to track last operator

III. Shift-Reduce Parsing
A. Basic idea
   1. Approach
      • Similar to operator precedence parsing
      • But extended to handle the whole grammar
      • Typically table-drive (not manually coded)
      • Uses a single stack rather than 2 stacks
      • Maintains a state value that reflects current operator
         i Used for precedence determination
         ii Used to determine how much to pop off the stack
      • States are also kept on the stack
   2. Look at tokens from left-to-right (one at a time)
      • Fits with our lexing scheme
   3. For each token
      • Decide whether to pop items off stack and replace with a non-terminal
         i The contents of the stack should reflect the RHS of a rule
         ii The determination is based on the state
         iii The replacement is the LHS of that rule
         iv This is called REDUCE
      • This could happen zero or more times
         i As with operator precedence
The new state (to compare with for this token) is popped with the reduce

• Then push the new token onto the stack
  
i  And set the new state
  
ii  This is called SHIFT

B. Example
  1. Grammar
     \[
     \begin{align*}
     E & ::= E + T \quad 1 \\
     E & ::= T \quad 2 \\
     T & ::= T \ast F \quad 3 \\
     T & ::= F \quad 4 \\
     F & ::= ( E ) \quad 5 \\
     F & ::= id \quad 6
     \end{align*}
     \]

  2. Parse: \( a+b\ast(c+d) \)

C. Issue: how to decide when to SHIFT and when to REDUCE
  1. Basic idea:
     • Keep track of what productions you might be parsing and where you might be in those productions
     • This involves precompiling what productions might be applicable
     • This would be done in a top-down fashion
  2. Dotted grammar rule
     • \( E ::= E \ast + T \)
  3. This information is represented as a state
     • A state is essentially a set of dotted rules from the grammar
     • We will show how to construct this later on
  4. Based on the current state and the next token, decide whether to REDUCE (pop) or SHIFT
     • REDUCE if at the end of a rule and token follows the rule
     • SHIFT if the token is legal at this point
     • ERROR if token can’t go here
     • Also determine the next state

D. This information is generally encoded in a table
  1. Sample Table

<table>
<thead>
<tr>
<th>STATE</th>
<th>Id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
<th>E</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s5</td>
<td></td>
<td>s4</td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>s6</td>
<td></td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>s7</td>
<td></td>
<td>r2</td>
<td></td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• Shift entries: s#
  i  Push current token and current state onto stack
  ii  Go to state #
• Reduce entries: r#
  i  Reduce using rule #
  ii  Pop items from RHS of rule off the stack
  iii  Decide next state from state on top of stack and new LHS
    ➢ Using right 3 columns
  iv  Push item from LHS of rule and new state onto stack
• Accept entries: A
  i  All done
• Empty entries
  i  Indicate an error
  ii  Could specialize with particular error id if desired (e#)
  iii  Otherwise syntax error
    ➢ Can tell what token should be next by examining row as well

2. Example
• Start with initial state on stack and parse a+b*(c+d) again
• SHOW THE USE OF THE STACK HERE – i.e. show the algorithms
  i  Include the notion of a value in place of the token/nonterminal
    ➢ Note that the nonterminal/terminal is on the stack, but is
      never looked at
  ii  Make sure what happened after a reduce is clear

E. What do we need to construct such a table
1. The set of states
• This is essentially the set of logical dotted rules
2. For each state what do to in that state
Suppose we have a single dotted rule, when do we shift

i. When the next token can be part of that rule
ii. Either the dot is in front of that token
iii. Or the dot is in front of a non-terminal which can start with that token
iv. Take the or if there are multiple dotted rules

Suppose we have a single dotted rule, when do we reduce

i. When the dot is at the end of the rule
ii. And the next token can follow an instance of that rule in a derivation

3. Given this information, we can construct the table

F. FIRST and FOLLOW (Covered by Qi in his slides)

1. To begin, we need to know what tokens are applicable as above
2. FIRST and FOLLOW are sets of symbols (tokens or non-tokens)
3. FIRST(x)
   • Let α be a string of grammar symbols
   • FIRST(α) is the set of all terminals that can begin strings derived from α
4. Computing FIRST
   • FIRST(a) = a for all terminals a
   • FIRST(A) for all non-terminals needs to be computed
   • FIRST(Xα) :
     i. FIRST(X) - ϵ is in FIRST(Xα)
     ii. IF ϵ is in FIRST(X) then FIRST(α) is in FIRST(Xα)
   • FIRST(A)
     i. For all non-terminals X set FIRST(X) = Φ
     ii. Repeat until nothing changes
        ➢ For each production X ::= α
           ▪ Add FIRST(α) to FIRST(X)
5. FOLLOW(A)
   • Let A be a non-terminal
   • FOLLOW(A) is the set of all terminals that can appear immediately to the right of A in some derivation, i.e. S ➔ αAβ for some α and β. If A can be the rightmost symbol in a derivation, $ is in FOLLOW(A).
   • Computed iteratively as with FIRST
   • Algorithm
     Compute FIRST
For all nonterminals \( X \), set \( \text{FOLLOW}(X) = \emptyset \)
Add $ to \text{FOLLOW}(S)
For each production \( A ::= \alpha \) DO
   For each occurrence of a nonterminal on RHS \( \alpha = \Upsilon B \beta \) DO
      Add \( \text{FIRST}(\beta) - \epsilon \) to \( \text{FOLLOW}(B) \)
REPEAT Until nothing changes
   For each production \( A ::= \alpha \) DO
     For each occurrence of a nonterminal on RHS, \( \alpha = \Upsilon B \beta \), DO
     IF \( \beta = \epsilon \) OR \( \epsilon \) is in \( \text{FIRST}(\beta) \) THEN
        Add \( \text{FOLLOW}(A) \) to \( \text{FOLLOW}(B) \)
6. \text{FIRST} and \text{FOLLOW} are inherent to parser generators
   - Both top-down and bottom up
   - For top-down: when scanning \( X \), what do I do if the next token is \( K \)
   - Can compute \( \text{FIRST}_k \) and \( \text{FOLLOW}_k \)
G. Set of States Construction
   1. Start with \( S \rightarrow \bullet \alpha \) as the first state
   2. Expand a state by adding other reachable states to it (CLOSURE)
      - IF \( \alpha \) starts with a non-terminal \( N \), add \( N \rightarrow \bullet \beta \) for each \( N \) rule
   3. Given a state, compute the next possible state for each nonterminal or terminal
      - Move the dot for all rules that match that symbol
      - Then compute the closure of the state
      - Merge equal states
H. Table Construction
   1. One row for each state
   2. Connections between states indicate shifts
   3. Reduce occurs when a terminal is in the follow set for a rule
IV. Shift-Reduce Grammars
   A. What can go wrong?
      1. The problems occur when there is ambiguity in building the table
         - That is why we need to understand how the table is built
      2. Two possible cases of ambiguity
         - Shift-Reduce conflict
            i. For a particular terminal and state, there is a shift rule and a reduce rule
Shift: some grammar rule in the state has a dot before the terminal
Reduce: dot at end of a grammar rule where follow of that rule includes terminal

ii In many cases, choosing shift is correct here
   Why?
   FOLLOW is an approximation, includes FOLLOW in every context, not this particular context

• Reduce-Reduce conflict
  i For a particular terminal and state, there are 2 (or more) reduce rules
    Multiple rules with dot at end where terminal is in FOLLOW set
  ii These are more difficult to correct
  iii Default is to use the rule order to select one, but this might not be correct

• Why no Shift-Shift conflicts?

B. To correct these problems, one rewrites the grammar
1. Notation
   • A grammar is LR1 if there are no conflicts
   • A grammar is LALR1 if the only conflicts are SR where Shift is the correct action

2. Rewrite can be aimed at one of two things
   • Removing the conflicting rule from the set
     i Merging multiple grammar rules into one
     ii Merging common prefix/suffix for grammar rules
     iii Splitting a nonterminal where there are different FOLLOW sets
     iv There are a number of tricks you learn and follow
   • Provide explicit disambiguation rules
     i For example rule priorities (operator precedence)

3. General approach to fixing conflicts
   • Find the conflict and understand the reason for it
   • Create common left prefixes
   • Replace nonterminal with appropriate RHS
   • Generalize the grammar
   • Add special tokens and do the work in lexical analysis
   • User precedence when shift-reduce conflicts are okay

4. Procedure
• Write a grammar that you think will work
• Run it through bison/flex to see what the conflicts are
• Work to understand and eliminate the conflicts
  i  What set of states caused it
  ii What is the FOLLOW set
  iii bison provides –verbose or –report options
• Then either
  i  Add explicit precedence rules
  ii  Or rewrite the grammar and try again
• Can be time consuming
• Best to eliminate all conflicts
  i  Should eliminate all reduce-reduce conflict
  ii  Document any remaining shift-reduce conflicts

C. Examples
1. stmt -> IF expr THEN stmt
   stmt -> IF expr THEN stmt ELSE stmt
2. expr -> expr [ value ]
   lhsexpr -> lhsexpr [ expr ]
   expr -> lhsexpr = expr
3. E -> E + E
   E -> E * E
   E -> ( E )
   E -> a
4. E -> T ( E )
   E -> F ( E_list )
   E -> F
   E_list -> E
   E_list -> E_list ', ' E
   T -> a
   F -> a
• This yields a reduce-reduce conflict on (   
• Can eliminate it by placing rule for T in rule for E
  i  E -> a ( E )
  ii  But this yields a shift-reduce conflict on (   
• Generalize by noting that there are the same construct
  i  Check that cast only has one argument during semantic analysis
5. E -> V
E -> F
V -> a
F -> a

6. A -> C B y
   A -> D B z
   C -> a
   C -> c
   D -> a
   B -> b
   • Yields a reduce/reduce conflict on B (why)
   • Replacing C/D with RHS in rules for A fixes this

V. Error handling in LR parsing
   A. Finding errors
   B. Recovering from errors
      1. Add an special ‘error’ symbol in the grammar
      2. When the parser finds an error
         • Pop tokens off the stack until error can be pushed on
         • Want a rule that fits error
      3. Then skip input tokens until you find a valid continuation
         • Already know from the rule what can follow
         • Extend to ensure that at least k tokens can follow
   C. Message
      1. Can encode message in the rule or in the table
   D. Error management class
      1. Handling out-of-order errors
      2. Tracking maximum error level
      3. Formatting errors

VI. YACC/BISON/…
   A. Let you specify rules and an action to take when rule is finished
      1. Intermediate actions need to be created by splitting the rule
         • Actually creates a new nonterminal with a ε-rule
         • Can create new conflicts when added
            i New rule needs to be reduced, if there are other rules to reduce, etc.
      2. Rules can have explicit priority (precedence)
   B. Actions can refer to values from right hand side of rule $i$
   C. Actions can set a value associated with the LHS nonterminal $$
D. Values associated with terminals are from the lexical analyzer

E. What should we use the actions for?
   1. Building the AST (next week)
   2. Providing feedback to the lexical analyzer
      • Defining symbols as types for example
      • Switch lexical start states can be done here as well

VII. Homework
   A. Read Chapter 4
   B. Exercises 4.1, 4.3