Top-Down Parsing

- Now we know:
  - how to build a parsing table for an LL(1) grammar (use FIRST/FOLLOW sets)
  - how to construct a recursive-descent parser from the parsing table

- Can we use recursive descent to build an abstract syntax tree too?

Creating the AST

abstract class Expr {
    class Hierarchy
    class Expr {
        Expr left, right;
        Expr(Expr L, Expr R) {
            left = L; right = R;
        }
    }
    class Num extends Expr {
        int value;
        Num(int v) { value = v; }
    }
}

AST Representation

(1 + 2 + (3 + 4)) + 5

How can we generate this structure during recursive-descent parsing?

Creating the AST

- Just add code to each parsing routine to create the appropriate nodes!
- Works because parse tree and call tree have same shape
- parse_S, parse_S', parse_E all return Expr:
  
  ```
  void parse_E() {
      switch(token) {
      case num: // E → number
          Expr result = Num(token.value);
          token = input.read(); return result;
      case '+': // E → ( S )
          token = input.read();
          Expr result = parse_S();
          if (token != ')') throw new ParseError();
          token = input.read(); return result;
      default: throw new ParseError();
      }
  }
  ```

AST Creation: parse_E

```
Grammars

- Have been using grammar for language of "sums with parentheses" e.g., \((1 + (3 + 4)) + 5\)
- Started with simple, right-associative grammar:
  \[ S \rightarrow E + S \mid E \rightarrow \text{num} \mid ( S ) \]
- Transformed it to an LL(1) grammar by left-factoring:
  \[ S \rightarrow SE' \]
  \[ E' \rightarrow E \mid S \mid \text{num} \mid ( S ) \]
- What if we start with a left-associative grammar?
  \[ S \rightarrow S + E \]
  \[ E \rightarrow \text{num} \mid ( S ) \]

Left vs. Right Associativity

Right recursion: right-associative

\[ S \rightarrow E + S \]
\[ S \rightarrow E \rightarrow \text{num} \]

Left recursion: left-associative

\[ S \rightarrow S + E \]
\[ S \rightarrow E \rightarrow \text{num} \]

Left-Recursve Grammars

- Left-recursive grammars are not LL(1)! 
  \[ S \rightarrow S \alpha \]
  \[ S \rightarrow \beta \]
- \( \text{FIRST}(\beta) \subseteq \text{FIRST}(S\alpha) \)
- Both productions will appear in the predictive table, at row \( S \) in all the columns corresponding to symbols in \( \text{FIRST}(\beta) \)
**Eliminate Left Recursion**

- Method for left-recursion elimination:
  - Replace
    \[ X \rightarrow X \alpha_1 | ... | X \alpha_m \]
    \[ X \rightarrow \beta_1 | ... | \beta_n \]
  - with
    \[ X \rightarrow \beta_1 X' | ... | \beta_n X' \]
    \[ X' \rightarrow \alpha_1 X' | ... | \alpha_m X' | \varepsilon \]
- (See the complete algorithm in the Dragon Book)

**Creating an LL(1) Grammar**

- Start with a left-recursive grammar:
  \[ S \rightarrow S + E \]
  \[ S \rightarrow E \]
  and apply left-recursion elimination algorithm:
  \[ S \rightarrow E S' \]
  \[ S' \rightarrow + E S' | \varepsilon \]
- Start with a right-recursive grammar:
  \[ S \rightarrow E + S \]
  \[ S \rightarrow E \]
  and apply left-factoring to eliminate common prefixes:
  \[ S \rightarrow E S' \]
  \[ S' \rightarrow + S | \varepsilon \]

**EBNF**

- Extended Backus-Naur Form is a form of specifying grammars which allows some regular expression syntax on RHS
  - \( *, +, (, ) \) ? operators (also [X] means \( X^* \))
  - \[ S \rightarrow E S' \]
  - \[ S' \rightarrow + S \]
- EBNF version: no position on + associativity

**Top-down Parsing EBNF**

- Recursive-descent code can directly implement the EBNF grammar:
  \[ S \rightarrow E \ ( + E ) \]
  ```cpp
  void parse_S () { // parses sequence of E + E + E ... 
      parse_E ();
      while (true) {
          switch (token) {
              case '+': token = input.read (); parse_E (); break;
              case ')': case EOF: return result;
              default: throw new ParseError ();
          }
      }
  }
  ```

**Reassociating the AST**

```cpp
Expr parse_S () { 
    Expr result = parse_E ();
    while (true) {
        switch (token) {
            case '+': token = input.read ();
            result = new Add (result, parse_E () );
            break;
            case ')': case EOF: return result;
            default: throw new ParseError ();
        }
    }
}
```

**Top-Down Parsing Summary**

```
Language grammar
    ↓
Left-recursive elimination
    ↓
LL(1) grammar
    ↓
predictive parsing table
    ↓
recursive-descent parser
    ↓
parser with AST generation
```
Next: Bottom-up Parsing

- A more powerful parsing technology
- LR grammars -- more expressive than LL
  - construct right-most derivation of program
  - left-recursive grammars, virtually all programming languages
  - Easier to express programming language syntax
- Shift-reduce parsers
  - Parsers for LR grammars
  - automatic parser generators (e.g. yacc,CUP)

Bottom-up Parsing

- Right-most derivation -- backward
  - Start with the tokens
  - End with the start symbol

```
S → S + E | E
E → num | (S)
```

```
(1+2+(3+4))+5 ← (E+2+(3+4))+5
← (S+E)+(E)+5 → (S+3)+(E)+5
← (S+E)+(S+3)+(E)+5
← (S+E)+(S+E)+(S+3)+(E)+5
← (S+E)+(S+E)+(S+E)+5
← (S+E)+(S+E)+(S+E)+(S+E)+5
← S+E ← (S+E)+5
← S ← (1+2)+(3+4)+5
```

Progress of Bottom-up Parsing

- Advantage of bottom-up parsing: can postpone the selection of productions until more of the input is scanned

Top-down Parsing

```
S → S + E | E
E → num | (S)
```

```
(1+2+(3+4))+5
S → S+ → E+E → (S)+E → (S+E)+E
→ (S+3+E)+E → (E+E)+E → (1+2+E)+E...
```

- In left-most derivation, entire tree above a token (2) has been expanded when encountered

Top-down vs. Bottom-up

- Bottom-up: Don’t need to figure out as much of the parse tree for a given amount of input
**Shift-reduce Parsing**

- **Parsing actions**: is a sequence of shift and reduce operations.
- **Parser state**: a stack of terminals and non-terminals (grows to the right)
- **Current derivation step**: always stack+input

**Derivation step**

<table>
<thead>
<tr>
<th>stack</th>
<th>unconsumed input</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1+2+5)→S</td>
<td>(1+2+5)→S</td>
</tr>
<tr>
<td>(E+2+5)→S</td>
<td>(E+2+5)→S</td>
</tr>
<tr>
<td>(E+2+5)→S</td>
<td>(E+2+5)→S</td>
</tr>
<tr>
<td>(S+2+5)→S</td>
<td>(S+2+5)→S</td>
</tr>
<tr>
<td>(S+E+2+5)→S</td>
<td>(S+E+2+5)→S</td>
</tr>
<tr>
<td>(S+E+2+5)→S</td>
<td>(S+E+2+5)→S</td>
</tr>
</tbody>
</table>

**Problem**

- How do we know which action to take: whether to shift or reduce, and which production?
- **Issues**:
  - Sometimes can reduce but shouldn’t
  - Sometimes can reduce in different ways

**Action Selection Problem**

- Given stack α and look-ahead symbol b, should parser:
  - shift b onto the stack (making it αb)
  - reduce X → γ assuming that stack has the form αγ (making it αX)
- If stack has form αγ, should apply reduction X → γ (or shift) depending on stack prefix α
  - α is different for different possible reductions, since γ’s have different length.