## Parsing Strategies

**Eric Roberts**
CS 106B
March 4, 2015

### The Problem of Parsing
- The rules for forming an expression can be expressed in the form of a grammar, as follows:
  
  \[
  \begin{align*}
  E &\rightarrow \text{constant} \\
  E &\rightarrow \text{identifier} \\
  E &\rightarrow E \ op \ E \\
  E &\rightarrow (E) \\
  \end{align*}
  \]

- The process of translating an expression from a string to its internal form is called **parsing**.

### A Two-Level Grammar
- The problem of parsing an expression can be simplified by changing the grammar to one that has two levels:
  - An expression is either a term or two expressions joined by an operator.
  - A term is either a constant, an identifier, or an expression enclosed in parentheses.
- This design is reflected in the following revised grammar.

  \[
  \begin{align*}
  E &\rightarrow T \\
  E &\rightarrow E \ op \ E \\
  T &\rightarrow \text{constant} \\
  T &\rightarrow \text{identifier} \\
  T &\rightarrow (E) \\
  \end{align*}
  \]

### Ambiguity in Parse Structures
- Although the two-level grammar from the preceding slide can recognize any expression, it is **ambiguous** because the same input string can generate more than one parse tree.

### Exercise: Parsing an Expression
- Diagram the expression tree that results from the input string

  \[
  \text{odd} = 2 \times n + 1
  \]

### The `parser.cpp` Implementation

```cpp
/* Implementation notes: readE * Dequeue op from stack, prec; */
* This function scans the next expression from the scanner by * matching the input to the following ambiguous grammar: * \[
*   E \rightarrow T \\
*   E \rightarrow E \ op \ E \\
* \]
* This version of the method uses precedence to resolve ambiguity. */
Expression *readE(TokenScanner & scanner, int prec) {
    Expression *exp = readT(scanner);
    string token;
    while (true) {
        token = scanner.nextToken();
        int tprec = precedence(token); // Type of token the scanner processed
        if (tprec <= prec) break;
        Expression *rhs = readE(scanner, tprec);
        exp = new CompoundExp(token, exp, rhs);
    }
    scanner.saveToken(token);
    return exp;
}
```
Exercise: Coding a BASIC Program

- On the second practice midterm, one of the problems concerned the **hailstone sequence**. For any positive integer \( n \), you compute the terms in the hailstone sequence by repeatedly executing the following steps:
  - If \( n \) is equal to 1, you’ve reached the end of the sequence and can stop.
  - If \( n \) is even, divide it by two.
  - If \( n \) is odd, multiply it by three and add one.
- Write a BASIC program that reads in an integer and prints out its hailstone sequence.

Modules in the Starter Folder

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic.cpp</td>
<td>You write this one, but it’s short.</td>
</tr>
<tr>
<td>exp.h</td>
<td>You need to remove the <code>=</code> operator and add a few things to <code>EvaluationContext</code>.</td>
</tr>
<tr>
<td>exp.cpp</td>
<td></td>
</tr>
<tr>
<td>parser.h</td>
<td>You need to remove the <code>=</code> operator.</td>
</tr>
<tr>
<td>parser.cpp</td>
<td></td>
</tr>
<tr>
<td>program.h</td>
<td>You’re given the interface, but need to write the private section and the implementation.</td>
</tr>
<tr>
<td>program.cpp</td>
<td></td>
</tr>
<tr>
<td>statement.h</td>
<td>You’re given the interface and need to supply the implementation.</td>
</tr>
<tr>
<td>statement.cpp</td>
<td></td>
</tr>
</tbody>
</table>

Your Primary Tasks

1. Figure out how the pieces of the program go together and what you need to do.
2. Code the `Program` class, keeping in mind what methods need to run in constant time.
3. Implement the `Statement` class hierarchy: