CMPE 152: Compiler Design

September 14 Class Meeting

Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak

CS Graduates' Mid-Career Salaries

- See

  http://www.payscale.com/college-salary-report/best-schools-by-state/bachelors/california?page=7

  for some interesting salary rankings and San Jose State!

Top Down Recursive Descent Parsing

- The term is very descriptive of how the parser works.
- Start by parsing the topmost source language construct.
  - For now it's a statement.
  - Later, it will be the program.

Top Down Recursive Descent Parsing

- "Drill down" (descend) by parsing the sub-constructs.

  statement →assignment statement → expression
  →variable → *etc*.

- Use recursion on the way down.

  statement →**`WHILE`** statement → statement → *etc*.

Top Down Recursive Descent Parsing, *cont'd*

- This is the technique for hand-coded parsers.
  - Very easy to understand and write.
  - The source language grammar is encoded in the *structure* of the parser code.
  - Close correspondence between the parser code and the syntax diagrams.
- Disadvantages
  - Can be tedious coding.
  - Ad hoc error handling.
  - Big and slow!

Top Down Recursive Descent Parsing, *cont'd*

- Bottom-up parsers can be smaller and faster.
  - Error handling can still be tricky.
  - To be covered later this semester.

Syntax and Semantics

- Syntax refers to the "grammar rules" of a source language.
- The rules prescribe the "proper form" of its programs.
- Rules can be described by syntax diagrams.
- Syntax checking:
  Does this sequence of tokens follow the syntax rules?

Syntax and Semantics, *cont'd*

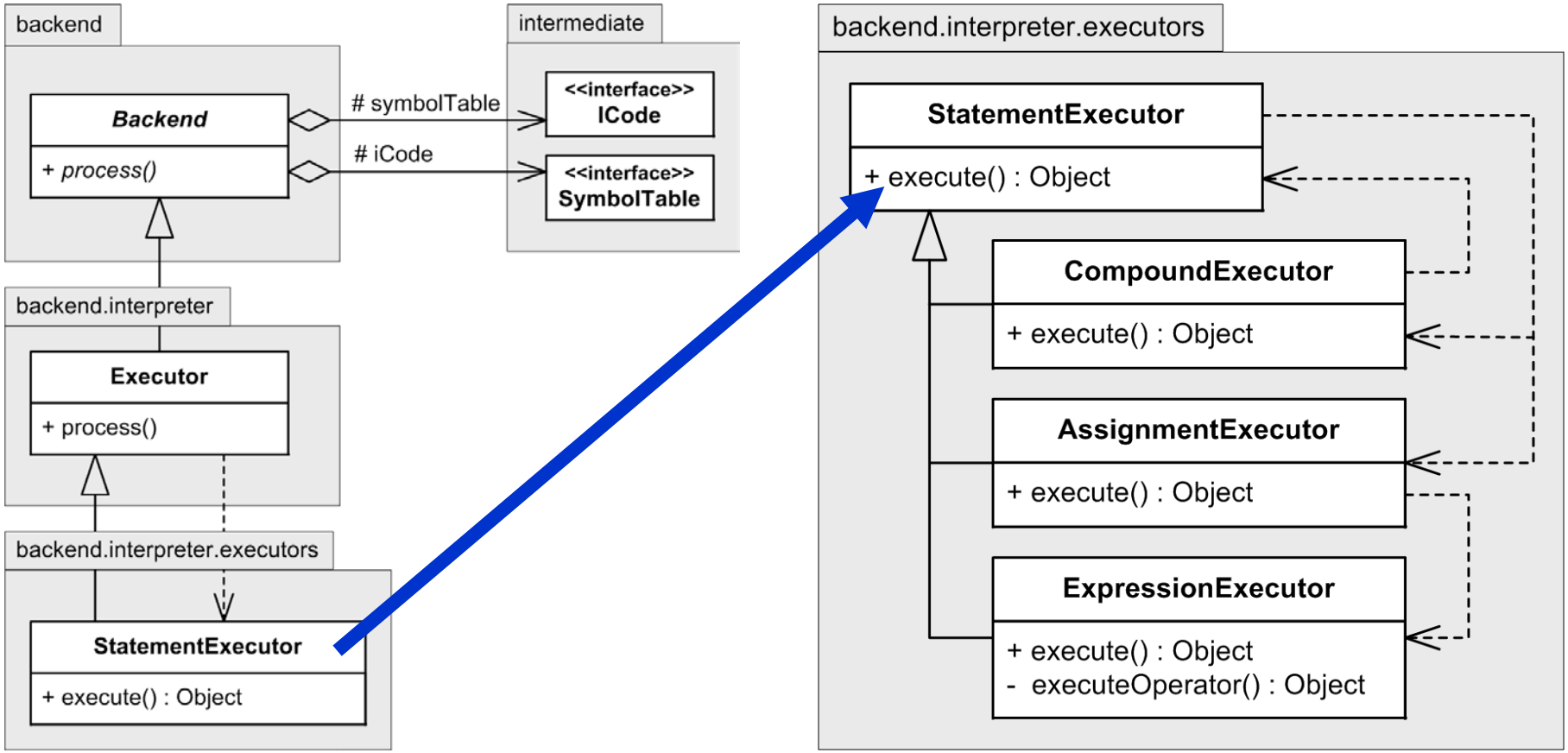
- Semantics refers to the meaning of the token sequences according to the source language.
- Example: Certain sequences of tokens constitute an `IF` statement according to the syntax rules.
- The semantics of the statement determine
  - How the statement will be executed by the interpreter, or
  - What code will be generated for it by the compiler.

Syntax and Semantics, *cont'd*

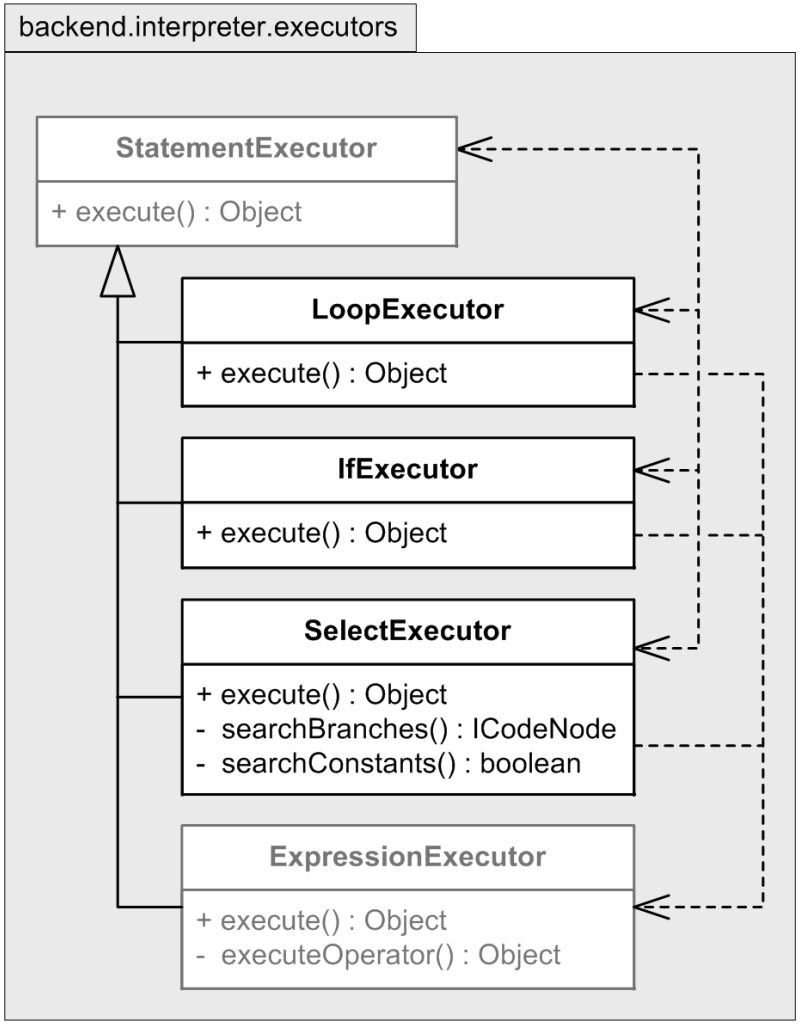
- Semantic actions by the front end parser:
  - Building symbol tables.
  - Type checking (which we'll do later).
  - Building proper parse trees.
  - The parse trees encode type checking and operator precedence in their structures.
- Semantic actions by the back end:
  - Interpreter: The executor runs the program.
  - Compiler: The code generator emits object code.

Interpreter Design

- Recall the design of our interpreter in the back end:

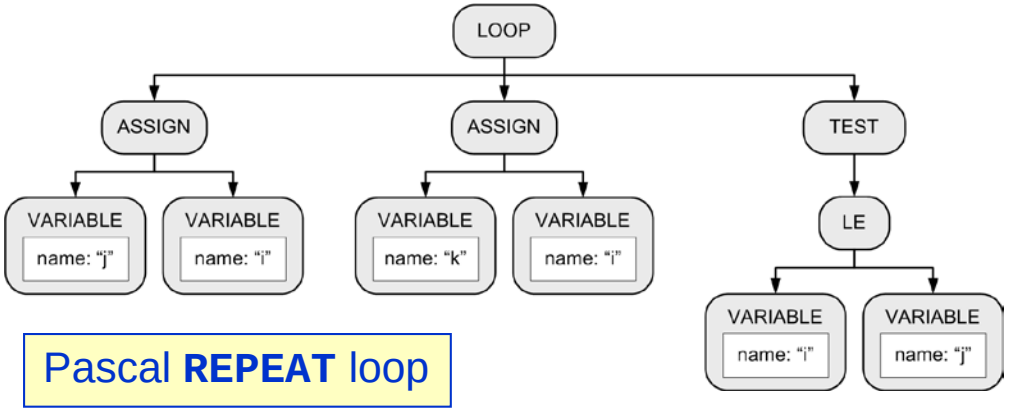


Control Statement Executor Classes

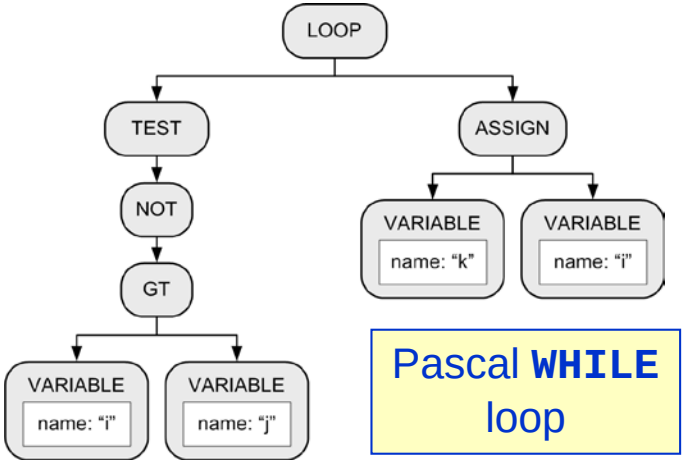


- New **`StatementExecutor`** subclasses:
  - **`LoopExecutor`**
  - **`IfExecutor`**
  - **`SelectExecutor`**
- The **`execute()`** method of each of these new subclasses executes the parse tree whose root node is passed to it.
  - Each returns null. Only the **`execute()`** method of **`ExpressionExecutor`** returns a value.

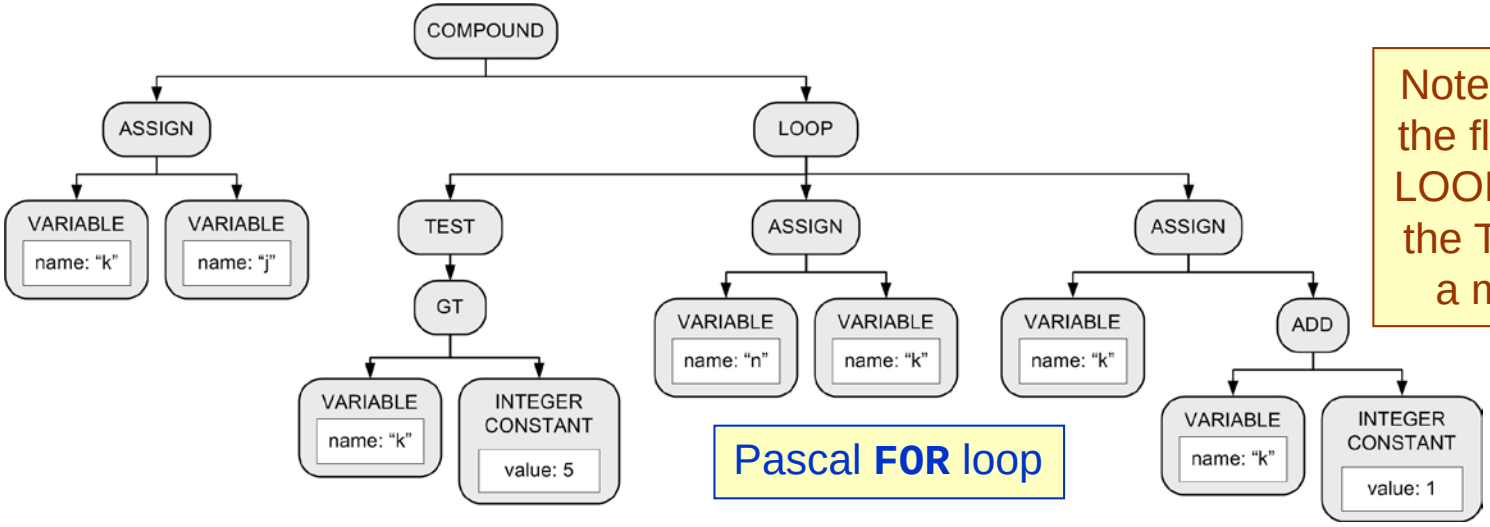
Executing a LOOP Parse Tree



Pascal **REPEAT** loop



Pascal **WHILE** loop

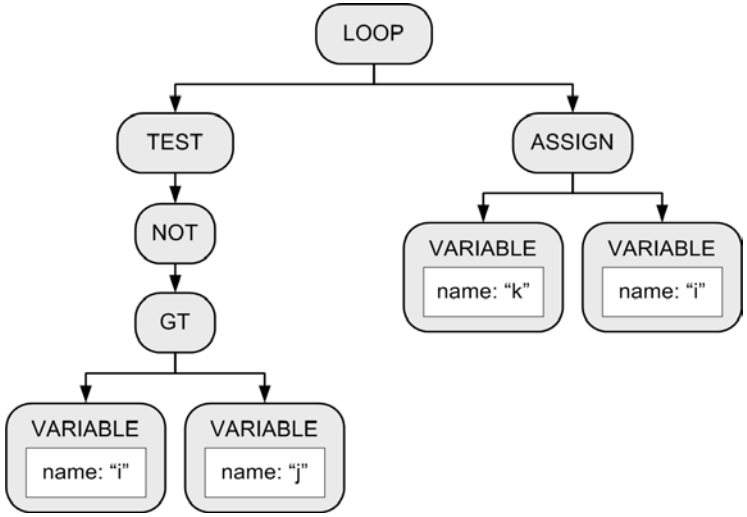


Pascal **FOR** loop

Note that we have the flexibility in the LOOP tree to have the TEST child be a middle child.

Executing a LOOP Parse Tree, *cont'd*

- Get all the children of the LOOP node.
- Repeatedly execute all the child subtrees in order.



- If a child is a TEST node, evaluate the node's relational expression subtree.
  - If the expression value is true, break out of the loop.
  - If the expression value is false, continue executing the child statement subtrees.

Executing a LOOP Parse Tree, *cont'd*

```
vector<ICodeNode *> loop_children = node->get_children();
ExpressionExecutor expression_executor(this);
StatementExecutor statement_executor(this);

while (!exit_loop)
{
    ++execution_count;  // count the loop statement itself

    for (ICodeNode *child : loop_children) {
        ICodeNodeTypeImpl child_type =
                                (ICodeNodeTypeImpl) child->get_type();

        if (child_type == NT_TEST)
        {
            if (expr_node == nullptr)
            {
                expr_node = child->get_children()[0];
            }

            DataValue *data_value =
                                expression_executor.execute(expr_node);
            exit_loop = data_value->b;
        }
        else
        {
            statement_executor.execute(child);
        }

        if (exit_loop) break;
    }
}
```

Keep looping until **exitLoop** becomes true.

Execute all the subtrees.

TEST node: Evaluate the boolean expression and set **exitLoop** to its value.

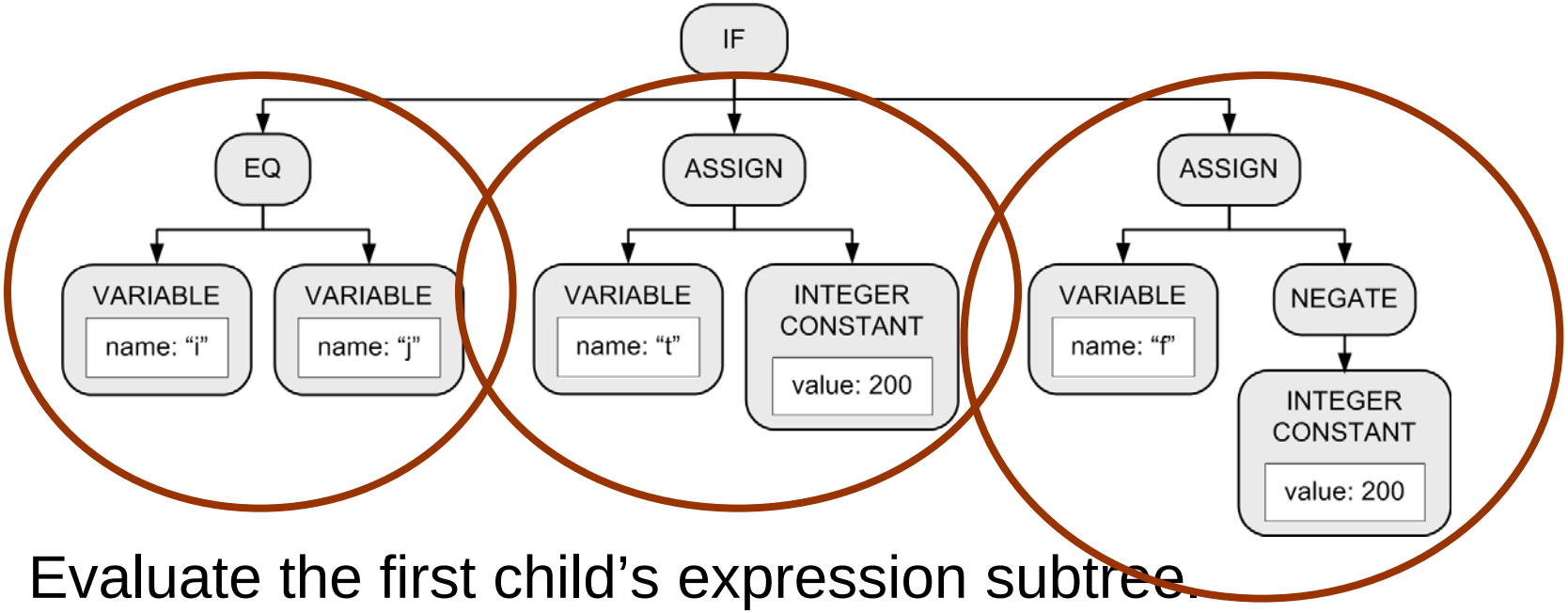
Statement subtree: Execute it.

Break out of the **for** loop if **exitLoop** is true.

Simple Interpreter II: Loops

- Demos
  - `./Chapter8cpp execute repeat.txt`
  - `./Chapter8cpp execute while.txt`
  - `./Chapter8cpp execute for.txt`

Executing an IF Parse Tree



- Evaluate the first child's expression subtree.
- If the expression value is true ...
  - Execute the second child's statement subtree.
- If the expression value is false …
  - If there is a third child statement subtree, then execute it.
  - If there isn't a third child subtree, then we're done with this tree.

Executing an IF Parse Tree, cont'd

```
DataValue *IfExecutor::execute(ICodeNode *node)
{
    vector<ICodeNode *> children = node->get_children();
    ICodeNode *expr_node = children[0];
    ICodeNode *then_stmt_node = children[1];
    ICodeNode *else_stmt_node = children.size() > 2 ? children[2] : nullptr;

    ExpressionExecutor expression_executor(this);
    StatementExecutor statement_executor(this);

    DataValue *data_value = expression_executor.execute(expr_node);
    if (data_value->b)
    {
        statement_executor.execute(then_stmt_node);
    }
    else if (else_stmt_node != nullptr)
    {
        statement_executor.execute(else_stmt_node);
    }

    ++execution_count;  // count the IF statement itself
    return nullptr;
}
```

Get the IF node's two or three children.

Execute the boolean expression to determine which statement subtree child to execute next.

Simple Interpreter II: `IF`

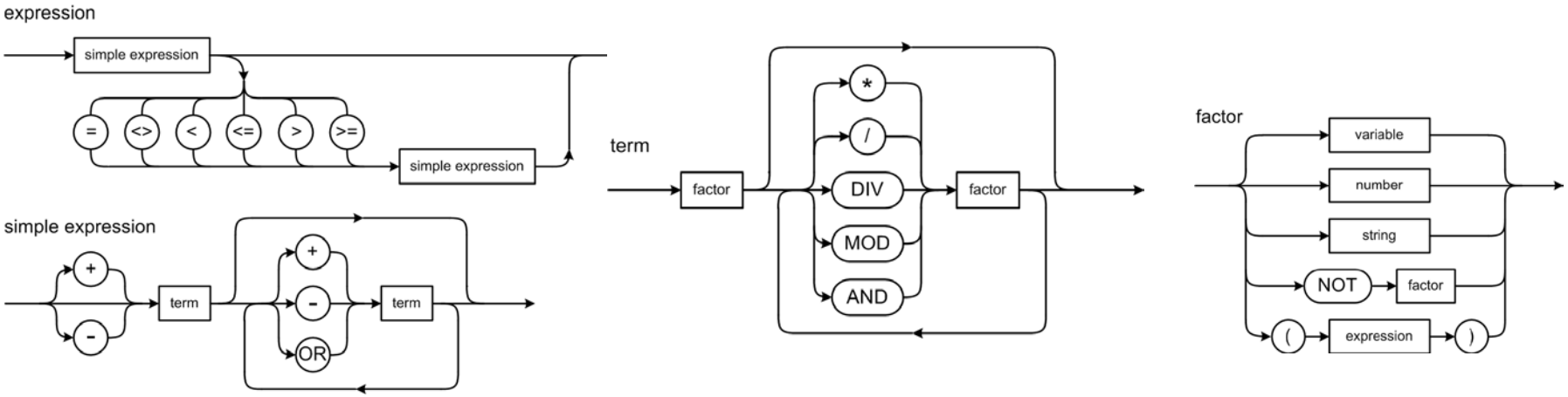
- Demo
  - `./Chapter8cpp execute if.txt`

Assignment #3

- Modify the parser code from Chapter 6:
  - Parse Pascal set expressions.
- Modify the interpreter code from Chapter 8:
  - Execute set expressions.

Assignment #3, *cont'd*

- What does the syntax diagram for set values look like?
- Where does the set value diagram fit in with the other expression syntax diagrams?



Assignment #3, *cont'd*

- What kinds of parse trees should you design?
- What trees should the parser build when it parses:
  - `[3, 1, 4, 2]`
  - `[high, mid..47, 2*low]`
  - `s2 := evens - teens + [high, mid..47, 2*low]`

Assignment #3, *cont'd*

- How does the executor in the back end evaluate set expressions at run time?
- What does the executor do when it's passed the root of a set value parse tree?
- What Java data structure does the executor use to represent a set value?
- What does it enter into a set variable's symbol table entry as the variable's value?

Assignment #3, *cont'd*

- How does the executor evaluate set expressions?
  - union, intersection, difference
  - equality, inequality
  - contains, is contained by
  - is a member of
  - **Tip:** At run time, use the Java set operations: http://www.java2s.com/Code/Java/Collections-Data-Structure/Setoperationsunionintersectiondifferencessymmetricdifferenceissubsetissuperset.htm or the C++ set operations: http://en.cppreference.com/w/cpp/algorithm

Assignment #3, *cont'd*

- The **`AssignmentExecutor`** sends a message each time its **`execute()`** method executes an assignment statement.
  - source line number
  - target variable name
  - value
- The message listener is the main **`Pascal`** class.
  - Do you need to modify the listener to print set values?

Assignment #3, *cont'd*

- Tutorial on Pascal sets: http://www.tutorialspoint.com/pascal/pascal_sets.htm
- Due Friday, September 29 at 11:59 PM.