CS 153: Concepts of Compiler Design

September 14 Class Meeting

Department of Computer Science
San Jose State University

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

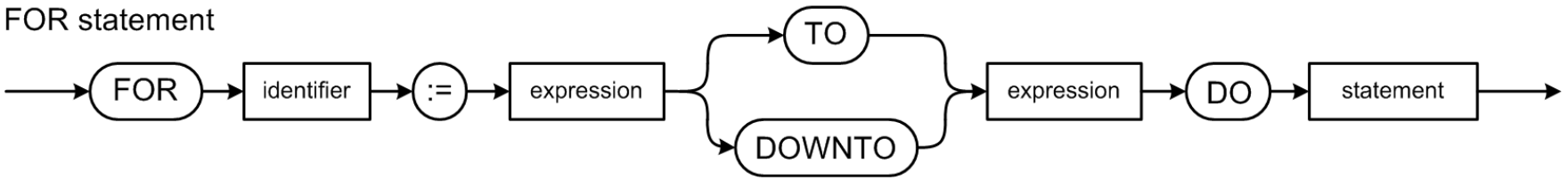
CS Graduates' Mid-Career Salaries

- 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!

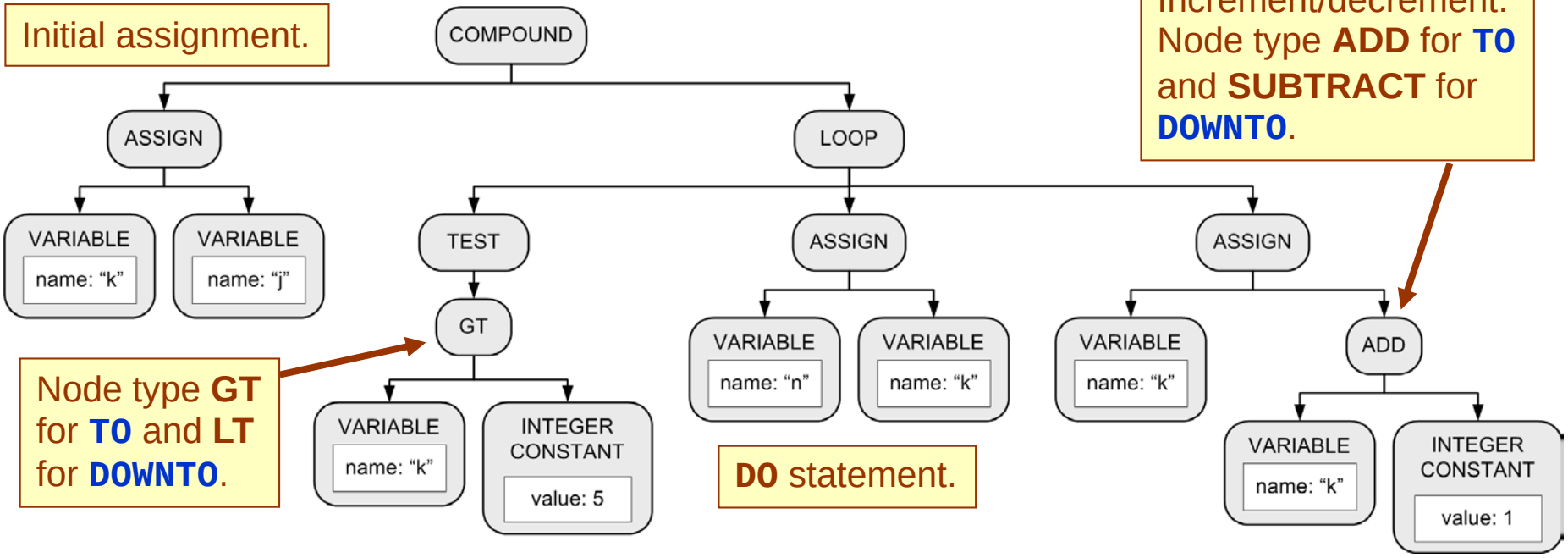
FOR Statement



- Example: `FOR k := j TO 5 DO n := k`

Initial assignment.

Increment/decrement: Node type **ADD** for **TO** and **SUBTRACT** for **DOWNTO**.



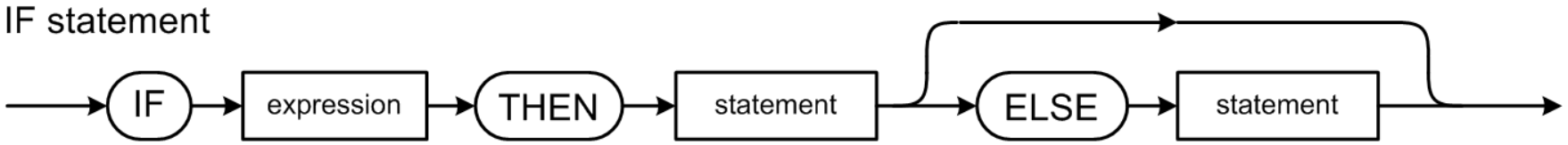
Node type **GT** for **TO** and **LT** for **DOWNTO**.

DO statement.

Pascal Syntax Checker II: **FOR**

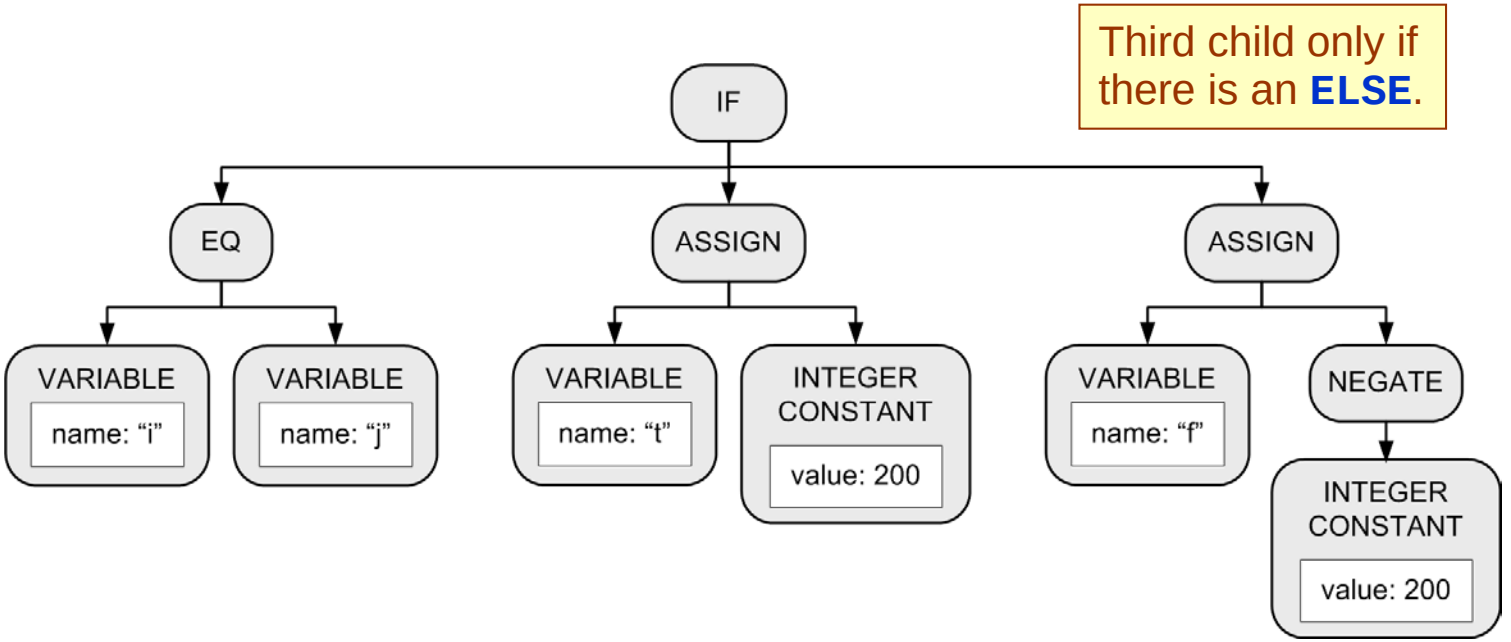
- Demo.
  - `java -classpath classes Pascal compile -i for.txt`
  - `java -classpath classes Pascal compile -i forerrors.txt`

IF Statement



- Example:

```
IF (i = j) THEN t := 200
           ELSE f := -200;
```



The "Dangling" **ELSE**

- Consider:

```
IF i = 3 THEN IF j = 2 THEN t := 500 ELSE f := -500
```

- Which **THEN** does the **ELSE** pair with?
  - Is it:

```
IF i = 3 THEN IF j = 2 THEN t := 500 ELSE f := -500
```

  - Or is it:

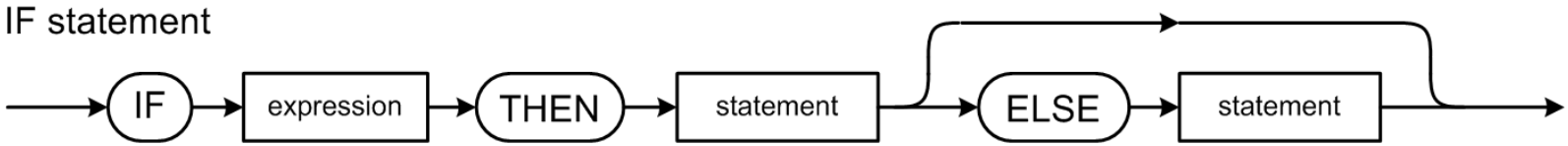
```
IF i = 3 THEN IF j = 2 THEN t := 500 ELSE f := -500
```

The "Dangling" **ELSE**, *cont'd*

- According to Pascal syntax, the nested **IF** statement is the **THEN** statement of the outer **IF** statement

```
IF i = 3 THEN IF j = 2 THEN t := 500 ELSE f := -500
```

- Therefore, the **ELSE** pairs with the closest (i.e., the second) **THEN**.



Scanner and Parser Rules of Thumb

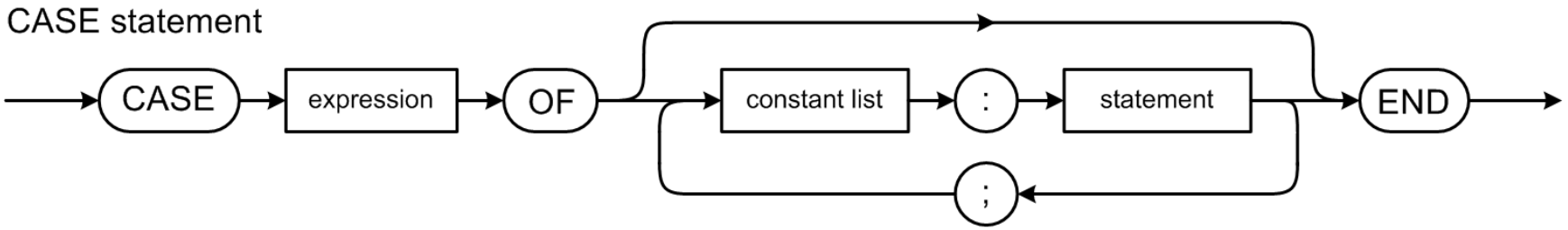
- Scanner
  - At any point in the source file, extract the longest possible token.
  - Example:
    - `<<=` is one shift-left-assign token
    - Not a shift-left token followed by an assign token
- Parser
  - At any point in the source file, parse the longest possible statement.
  - Example:

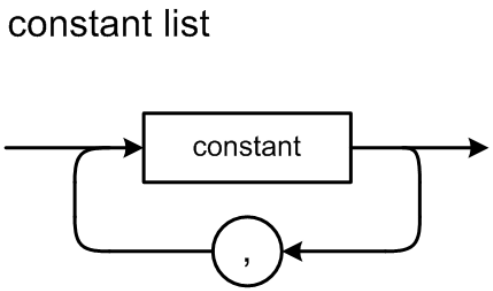
```
IF i = 3 THEN IF j = 2 THEN t := 500 ELSE f := -500
```

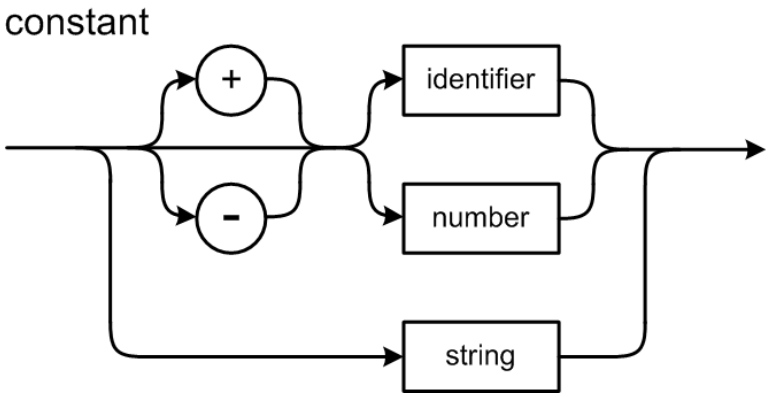
Pascal Syntax Checker II: `IF`

- Demo.
  - `java -classpath classes Pascal compile -i if.txt`
  - `java -classpath classes Pascal compile -i iftest.txt`

CASE Statement







- Example:

```
CASE i+1 OF
     1:         j := i;
     4:         j := 4*i;
     5, 2, 3:   j := 523*i;
END
```

Note that Pascal's **CASE** statement does not use **BREAK** statements.

CASE Statement, *cont'd*

- Example:
  - `CASE i+1 OF`

    `END`

Pascal Syntax Checker II: **CASE**

- Demo.
  - `java -classpath classes Pascal compile -i case.txt`
  - `java -classpath classes Pascal compile -i caseerrors.txt`

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*

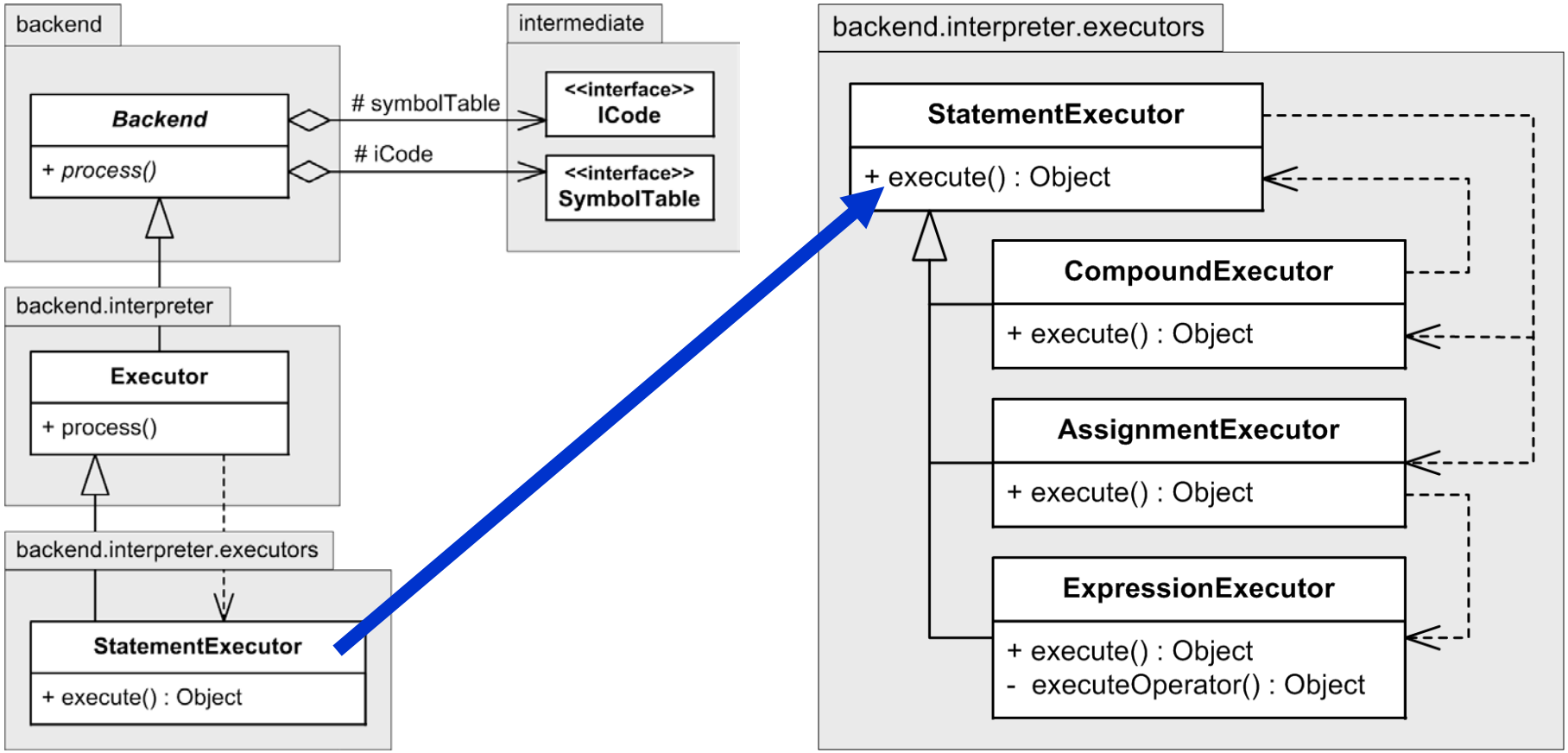
- Semantics refers to the meaning of the token sequences according to the source language.
- Example: Certain sequences of tokens constitute an `IF` statement according to the syntax rules.
- The semantics of the statement determine
  - How the statement will be executed by the interpreter, or
  - What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

Syntax and Semantics, *cont'd*

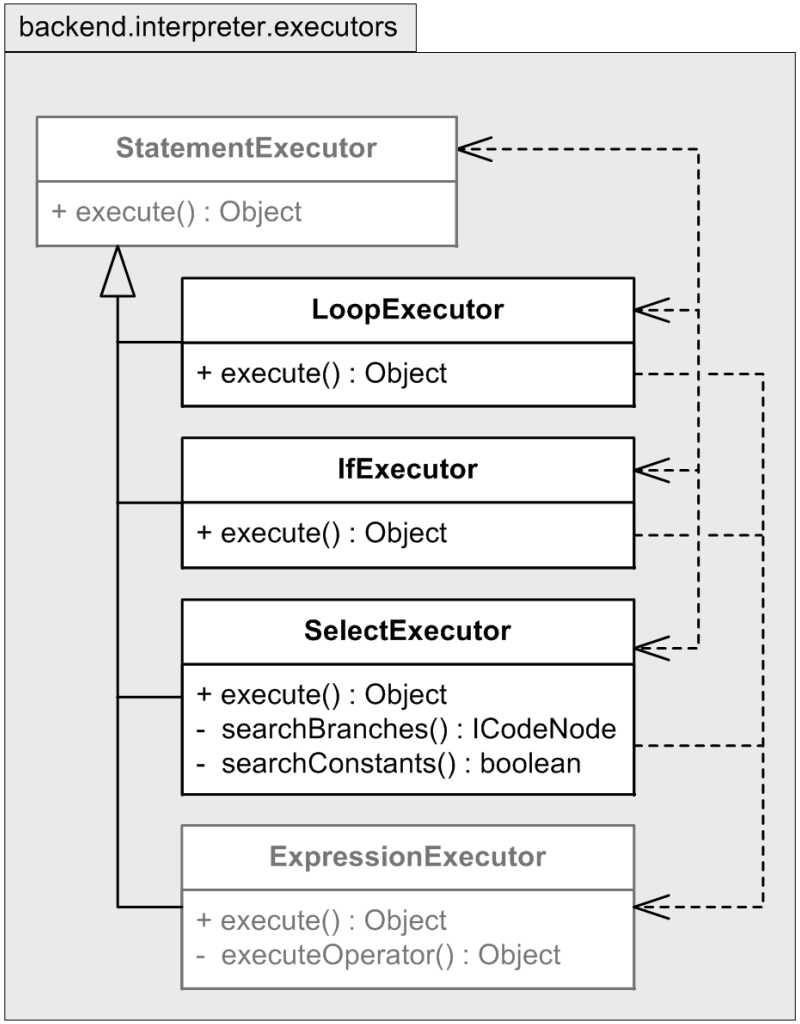
- Semantic actions by the front end parser:
  - Building symbol tables.
  - Type checking (which we'll do later).
  - Building proper parse trees.
  - The parse trees encode type checking and operator precedence in their structures.
- Semantic actions by the back end:
  - Interpreter: The executor runs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

Interpreter Design

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

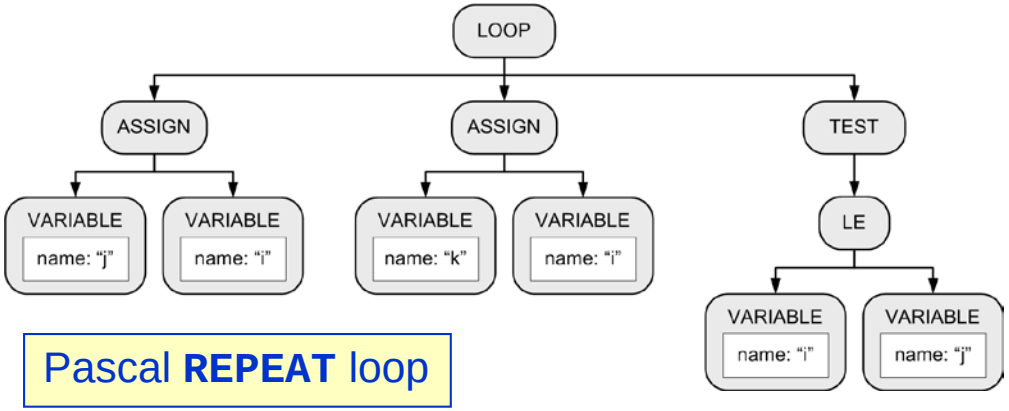


Control Statement Executor Classes

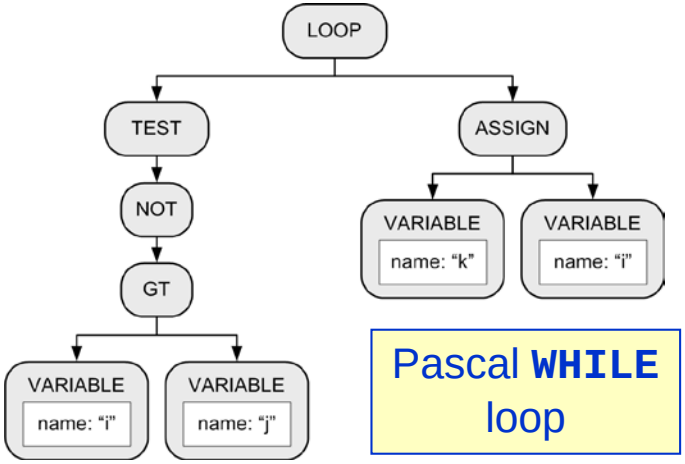


- New **`StatementExecutor`** subclasses:
  - **`LoopExecutor`**
  - **`IfExecutor`**
  - **`SelectExecutor`**
- The **`execute()`** method of each of these new subclasses executes the parse tree whose root node is passed to it.
  - Each returns null. Only the **`execute()`** method of **`ExpressionExecutor`**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

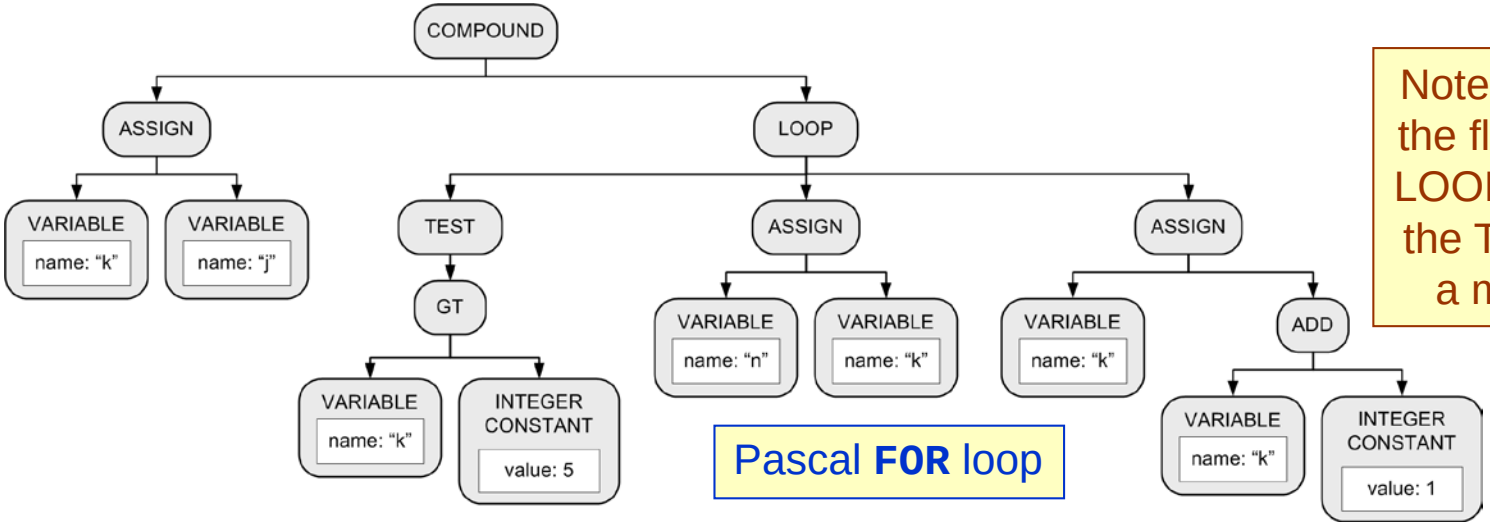
Executing a LOOP Parse Tree



Pascal **REPEAT** loop



Pascal **WHILE** loop

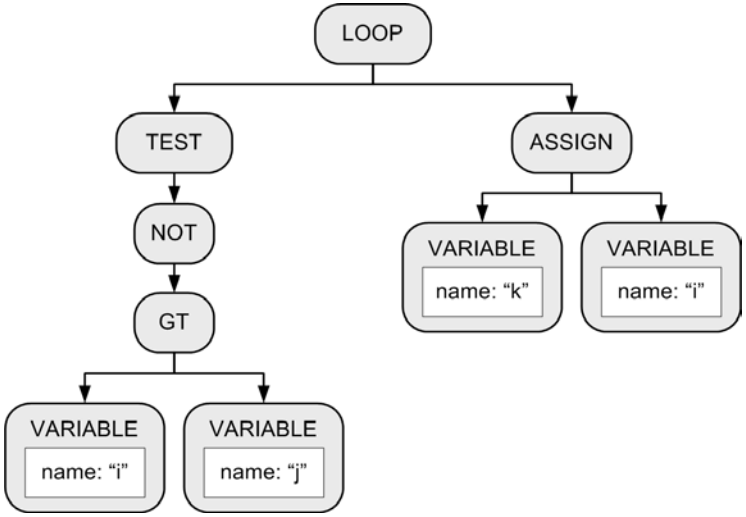


Pascal **FOR** loop

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*

```
ArrayList<ICodeNode> loopChildren = node.getChildren();
ExpressionExecutor expressionExecutor = new ExpressionExecutor(this);
StatementExecutor statementExecutor = new StatementExecutor(this);

while (!exitLoop) {
    ++executionCount;  // count the loop statement itself

    for (ICodeNode child : loopChildren) {
        ICodeNodeTypeImpl childType = (ICodeNodeTypeImpl) child.getType();

        if (childType == TEST) {
            if (exprNode == null) {
                exprNode = child.getChildren().get(0);
            }
            exitLoop = (Boolean) expressionExecutor.execute(exprNode);
        }
        else {
            statementExecutor.execute(child);
        }

        if (exitLoop) break;
    }
}
```

Keep looping until **exitLoop** becomes true.

Execute all the subtrees.

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

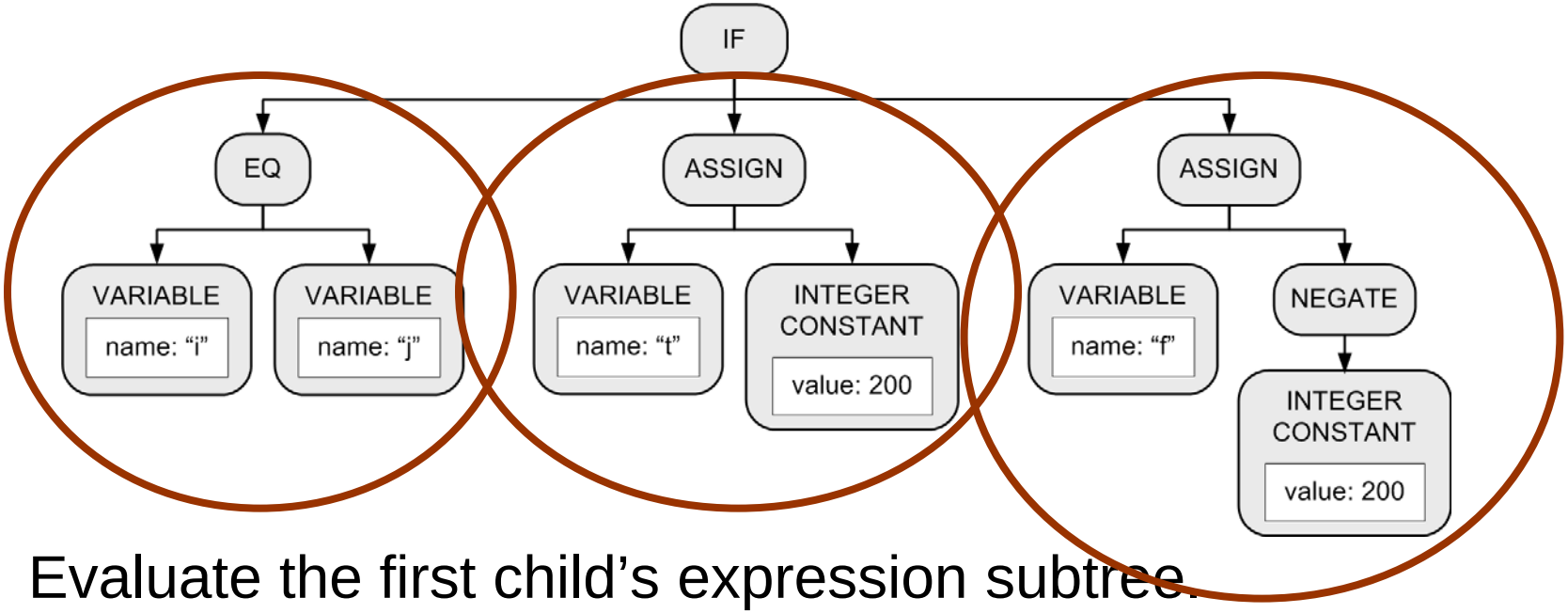
Statement subtree: Execute it.

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

Simple Interpreter II: Loops

- Demos
  - `java -classpath classes Pascal execute repeat.txt`
  - `java -classpath classes Pascal execute while.txt`
  - `java -classpath classes Pascal 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

```
public Object execute(ICodeNode node)
{
    ArrayList<ICodeNode> children = node.getChildren();
    ICodeNode exprNode = children.get(0);
    ICodeNode thenStmtNode = children.get(1);
    ICodeNode elseStmtNode = children.size() > 2 ? children.get(2) : null;

    ExpressionExecutor expressionExecutor = new ExpressionExecutor(this);
    StatementExecutor statementExecutor = new StatementExecutor(this);

    boolean b = (Boolean) expressionExecutor.execute(exprNode);
    if (b) {
        statementExecutor.execute(thenStmtNode);
    }
    else if (elseStmtNode != null) {
        statementExecutor.execute(elseStmtNode);
    }

    ++executionCount;  // count the IF statement itself
    return null;
}
```

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`

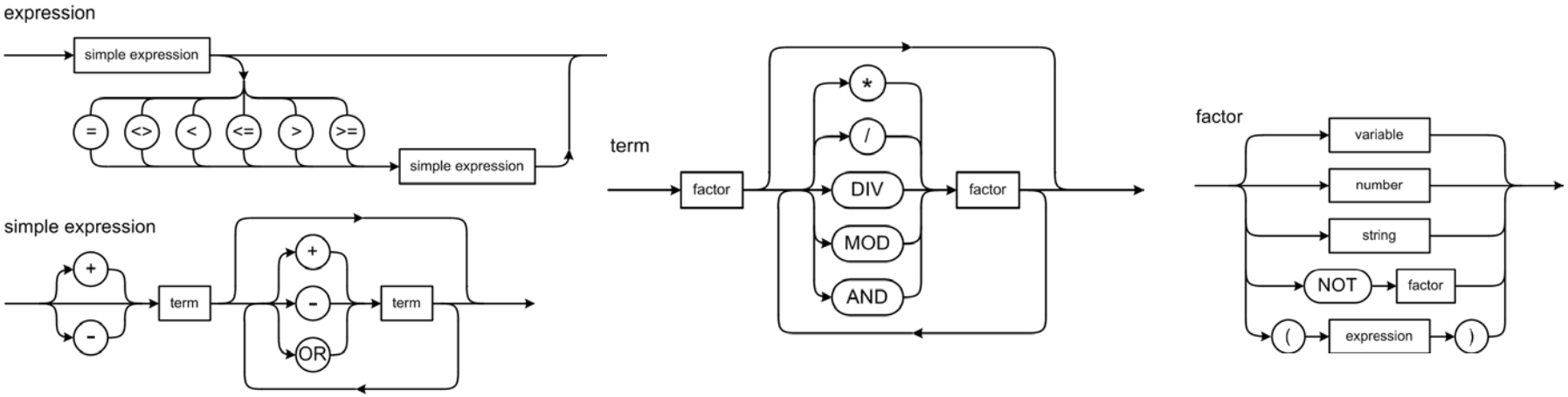
- Demo
  - `java -classpath classes Pascal execute if.txt`

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.