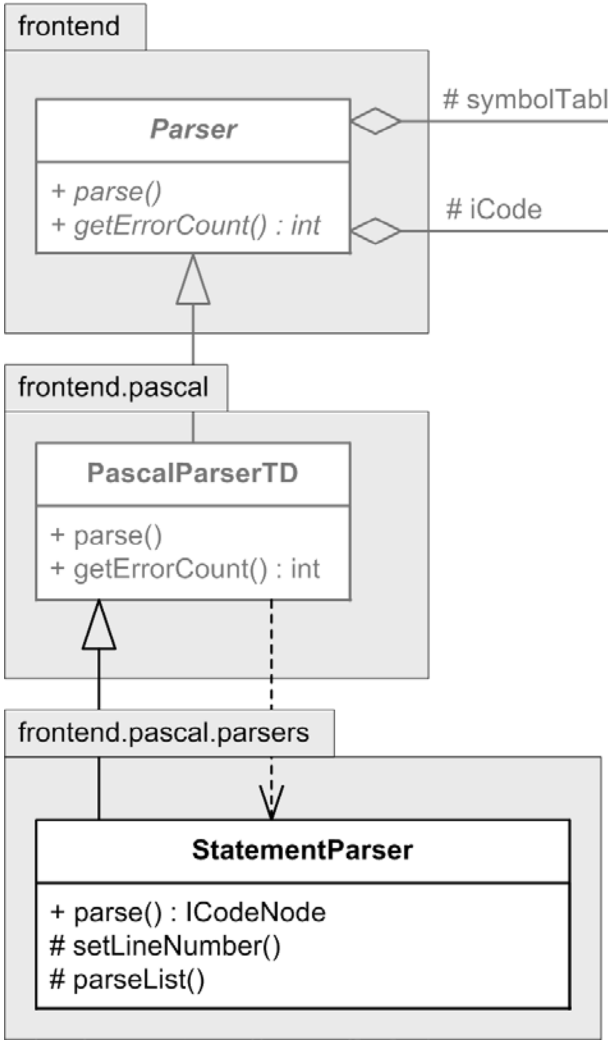
CS 153: Concepts of Compiler Design

September 12 Class Meeting

Department of Computer Science
San Jose State University

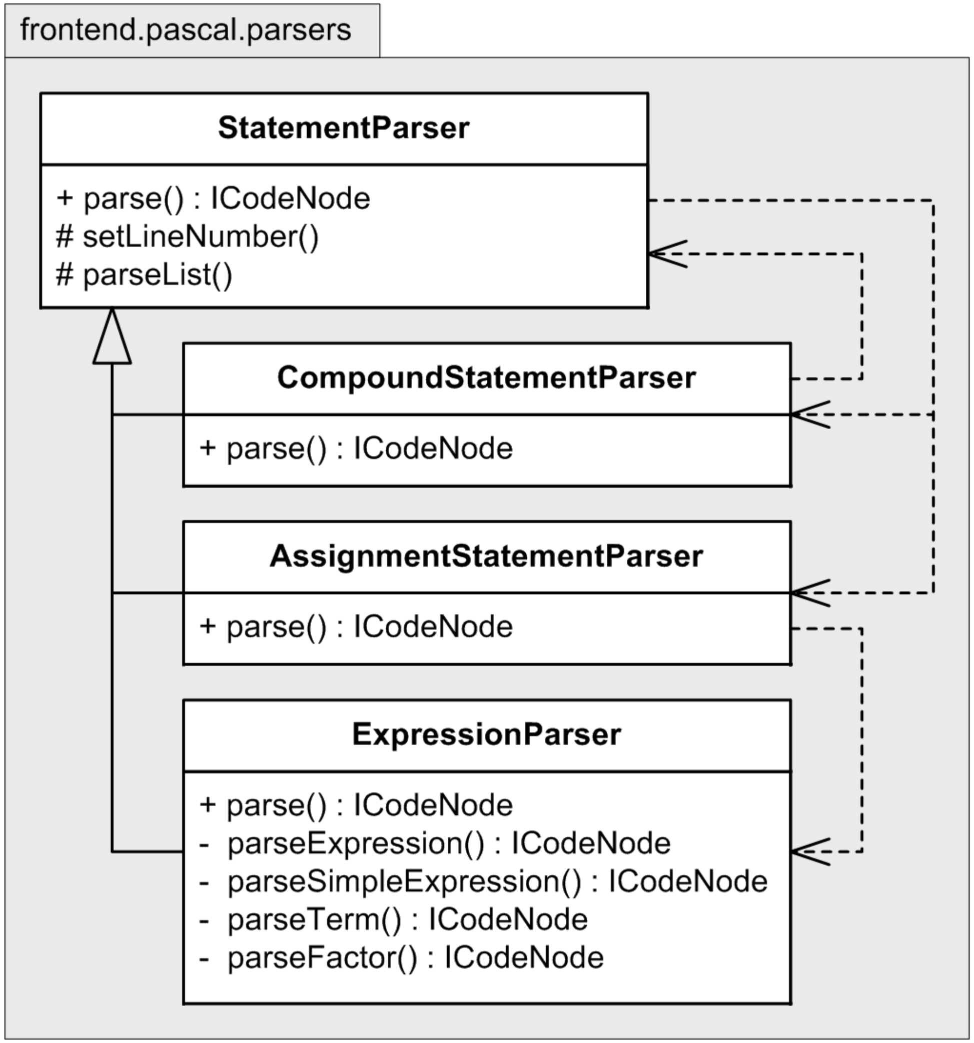
Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak

Statement Parser Class



- Class **StatementParser** is a subclass of **PascalParserTD** which is a subclass of **Parser**.
  - Its **parse()** method builds a part of the parse tree and returns the root node of the newly built subtree.

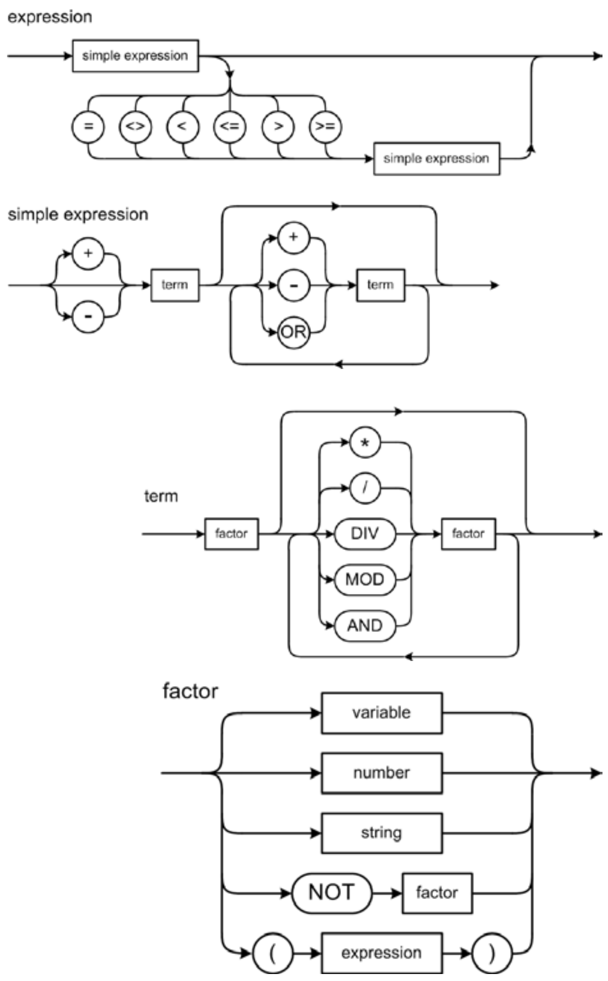
Statement Parser Subclasses



- `StatementParser` itself has subclasses:
  - `CompoundStatementParser`
  - `AssignmentStatementParser`
  - `ExpressionParser`
- The `parse()` method of each subclass returns the root node of the subtree that it builds.
- Note the dependency relationships among `StatementParser` and its subclasses.

Parsing Expressions

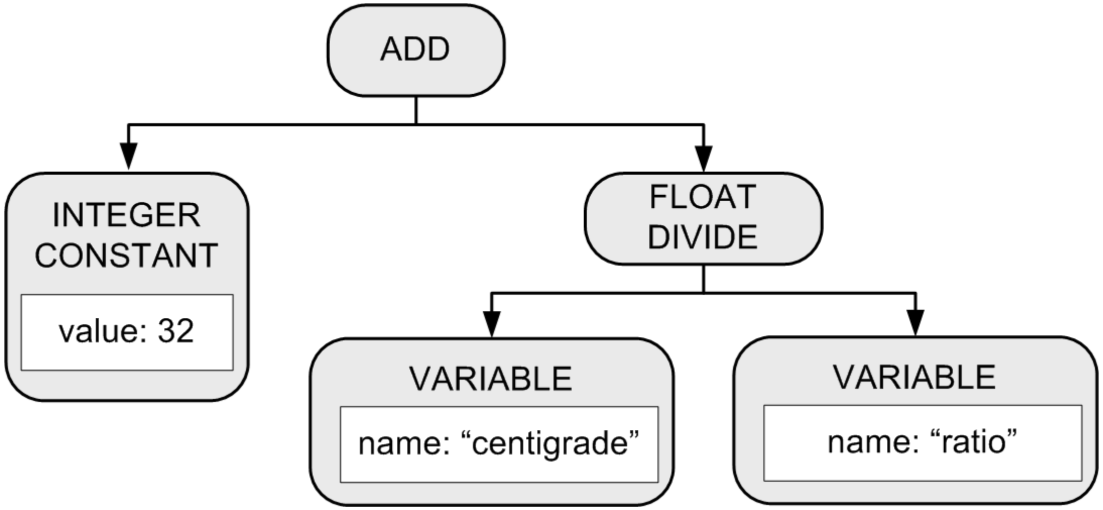
- Pascal statement parser subclass **ExpressionParser** has methods that correspond to the expression syntax diagrams:
  - parseExpression()
  - parseSimpleExpression()
  - parseTerm()
  - parseFactor()



- Each parse method returns the root of the subtree that it builds.
  - Therefore, **ExpressionParser**’s **parse()** method returns the root of the entire expression subtree.

Parsing Expressions, *cont'd*

- Pascal's operator precedence rules determine the order in which the parse methods are called.
  - The parse tree that `ExpressionParser` builds determines the order of evaluation.
  - Example: `32 + centigrade/ratio`

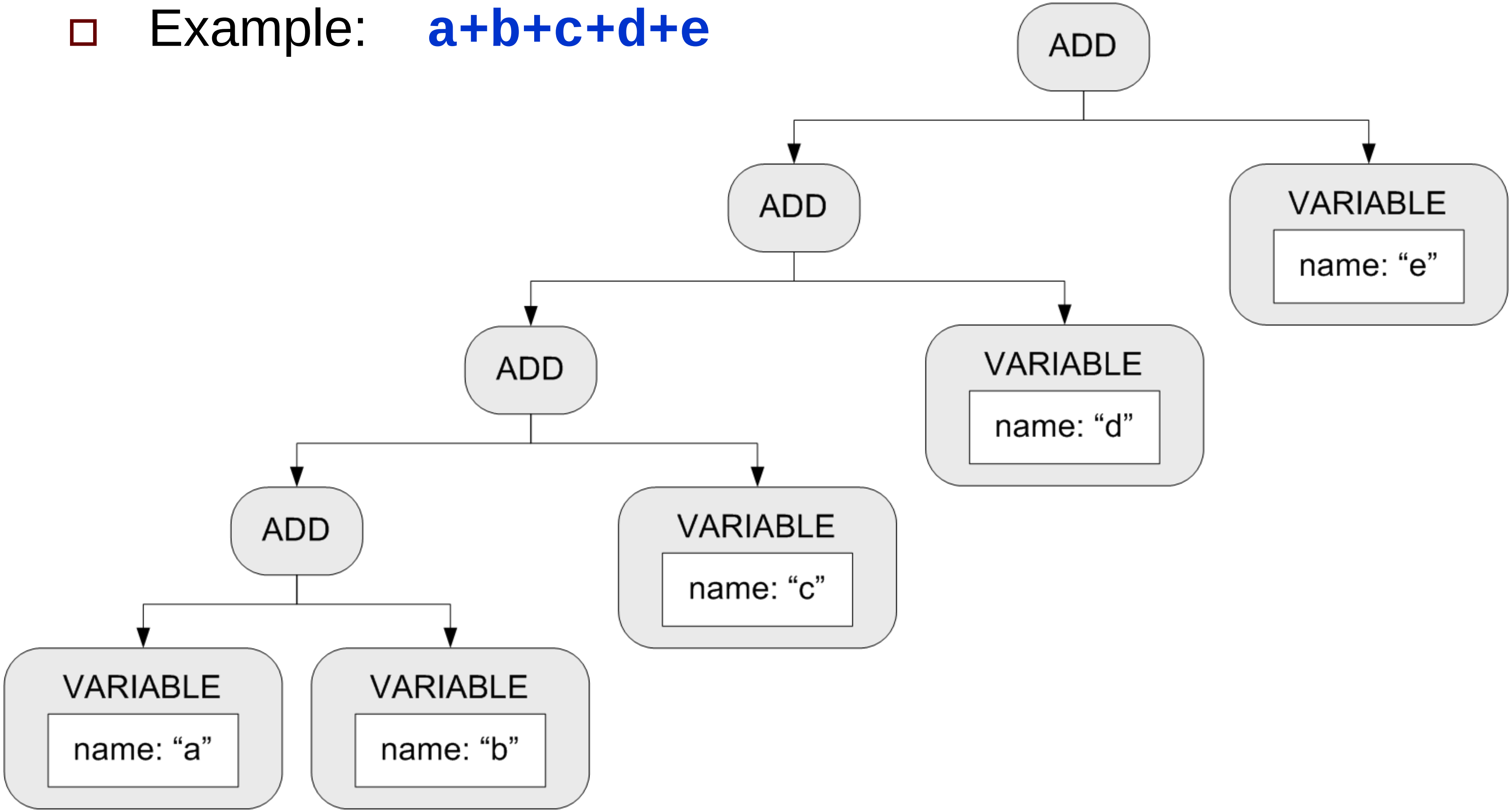


Do a **postorder traversal** of the parse tree.

Visit the **left subtree**, visit the **right subtree**, then visit the **root**.

Parsing Expressions, *cont'd*

- Example: **a+b+c+d+e**



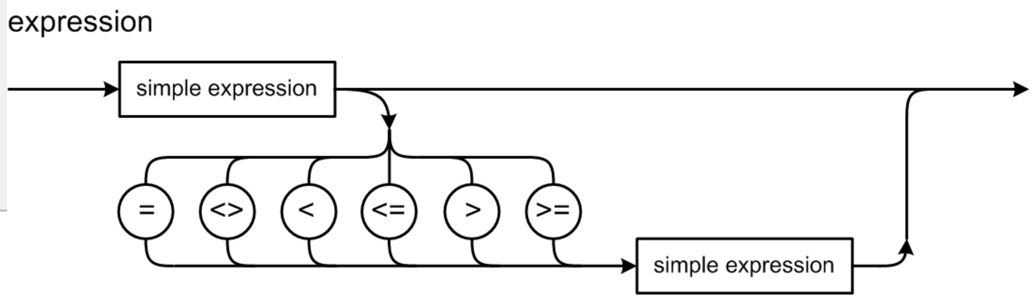
Example: Method **`parseExpression()`**

- First, we need to map Pascal token types to parse tree node types.
  - Node types need to be language-independent.
  - We'll use a hash table.

```
// Map relational operator tokens to node types.
private static final HashMap<PascalTokenType, ICodeNodeType>
    REL_OPS_MAP = new HashMap<PascalTokenType, ICodeNodeType>();
static {
    REL_OPS_MAP.put(EQUALS, EQ);
    REL_OPS_MAP.put(NOT_EQUALS, NE);
    REL_OPS_MAP.put(LESS_THAN, LT);
    REL_OPS_MAP.put(LESS_EQUALS, LE);
    REL_OPS_MAP.put(GREATER_THAN, GT);
    REL_OPS_MAP.put(GREATER_EQUALS, GE);
};
```

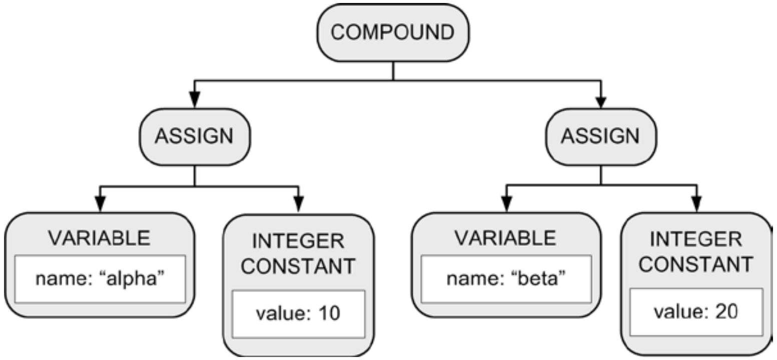
Method `parseExpression()`, *cont'd*

```
private ICodeNode parseExpression(Token token)
    throws Exception
{
    ICodeNode rootNode = parseSimpleExpression(token);
    token = currentToken();
    TokenType tokenType = token.getType();

    if (REL_OPS.contains(tokenType)) {
        ICodeNodeType nodeType = REL_OPS_MAP.get(tokenType);
        ICodeNode opNode = ICodeFactory.createICodeNode(nodeType);
        opNode.addChild(rootNode);

        token = nextToken();  // consume the operator
        opNode.addChild(parseSimpleExpression(token));
        rootNode = opNode;
    }

    return rootNode;
}
```



Printing Parse Trees

- Utility class **`ParseTreePrinter`** prints parse trees.
  - Prints in an XML format.



```
<COMPOUND line="11">
    <ASSIGN line="12">
        <VARIABLE id="alpha" level="0" />
        <INTEGER_CONSTANT value="10" />
    </ASSIGN>
    <ASSIGN line="13">
        <VARIABLE id="beta" level="0" />
        <INTEGER_CONSTANT value="20" />
    </ASSIGN>
</COMPOUND>
```

Pascal Syntax Checker I

- □ The **`-i`** compiler option prints the intermediate code:

```
java -classpath classes Pascal execute -i assignments.txt
```

- □ Add to the constructor of the main **`Pascal`** class:

```
if (intermediate) {
    ParseTreePrinter treePrinter =
                         new ParseTreePrinter(System.out);
    treePrinter.print(iCode);
}
```

Pascal Syntax Checker I, *cont'd*

- Demo (Chapter 5)

- For now, all we can parse are compound statements, assignment statements, and expressions.
- More syntax error handling.

What Have We Accomplished So Far?

- A working scanner for Pascal.
- A set of Pascal token classes.
- Symbol table and intermediate code classes.
- A parser for Pascal compound and assignment statements and expressions.
  - Generate parse trees.
  - Syntax error handling.
- A messaging system with message producers and message listeners.
- Placeholder classes for the back end code generator and executor.
- So … we are ready to put all this stuff into action!

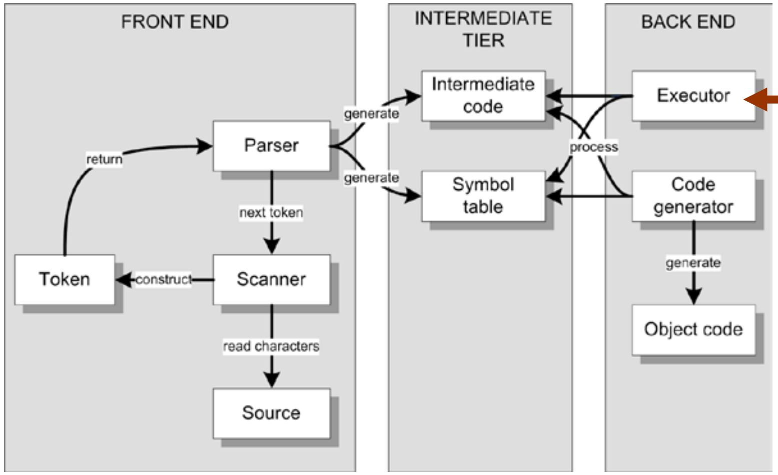
Temporary Hacks for Now

- Only one symbol table in the stack.
- Variables are scalars (not records or arrays) but otherwise have no declared type.
    - We haven’t parsed any Pascal declarations yet!
- We consider a variable to be “declared” (and we enter it into the symbol table) the first time it appears on the left-hand-side of an assignment statement (it’s the target of the assignment).

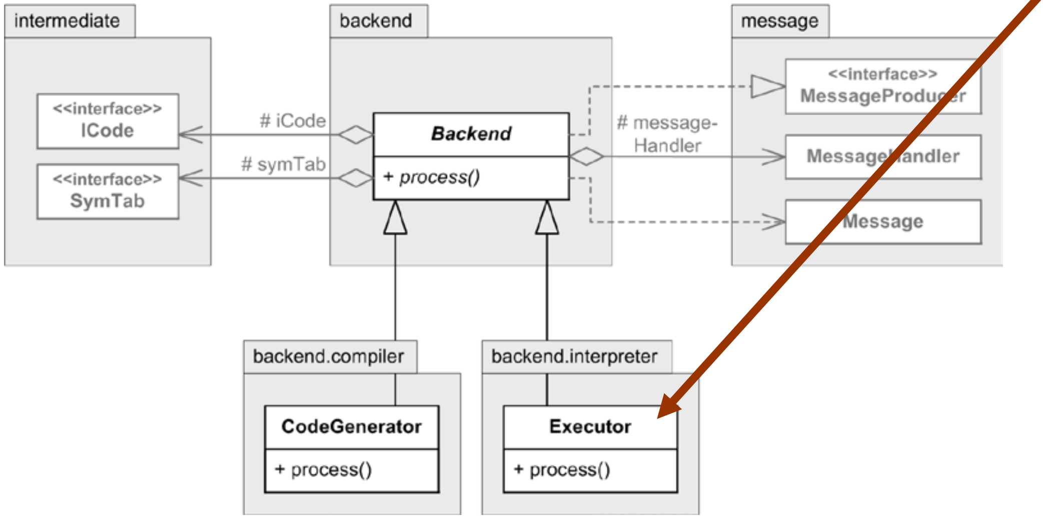
A New Temporary Hack

- Today, we’re going to store runtime computed values into the symbol table.
  - As attribute **`DATA_VALUE`**

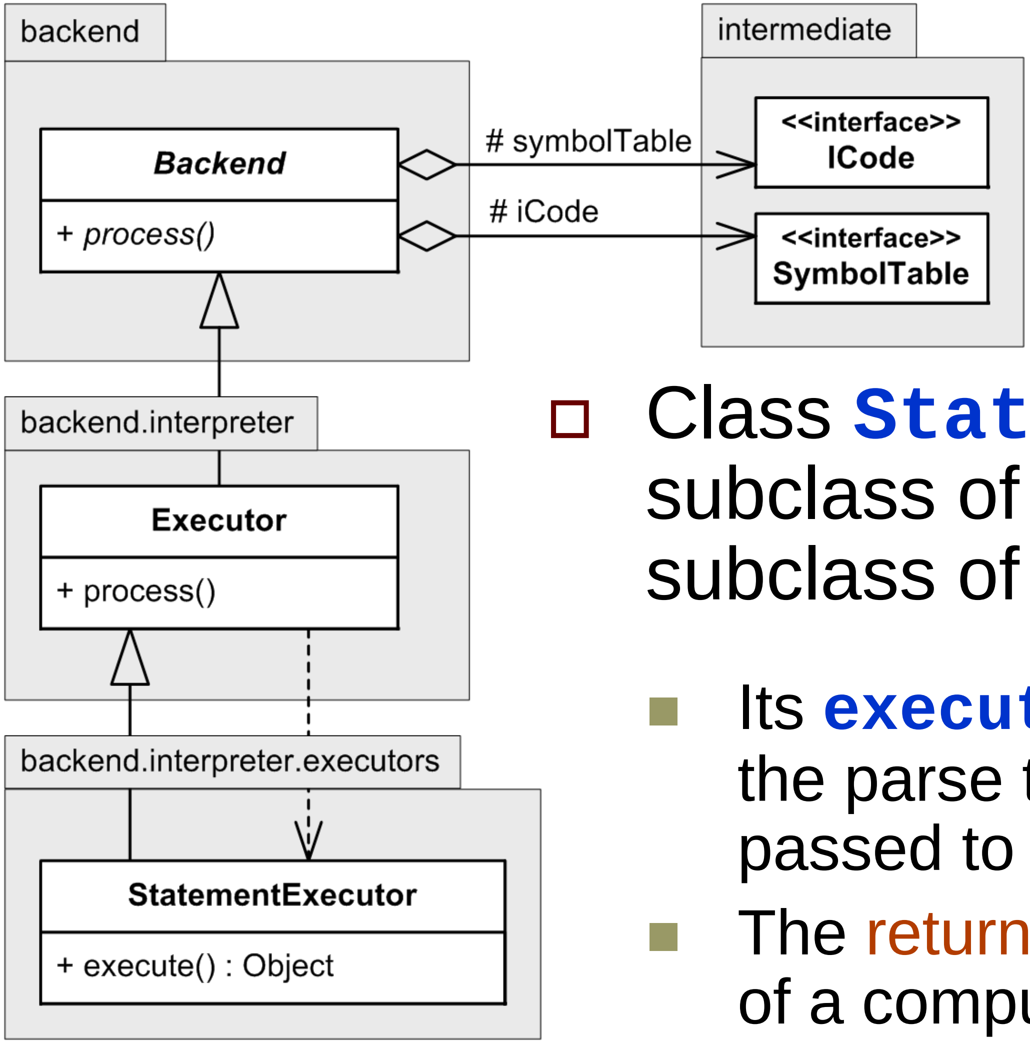
Quick Review of the Framework



Today’s topic:
Executing compound statements, assignment statements, and expressions.

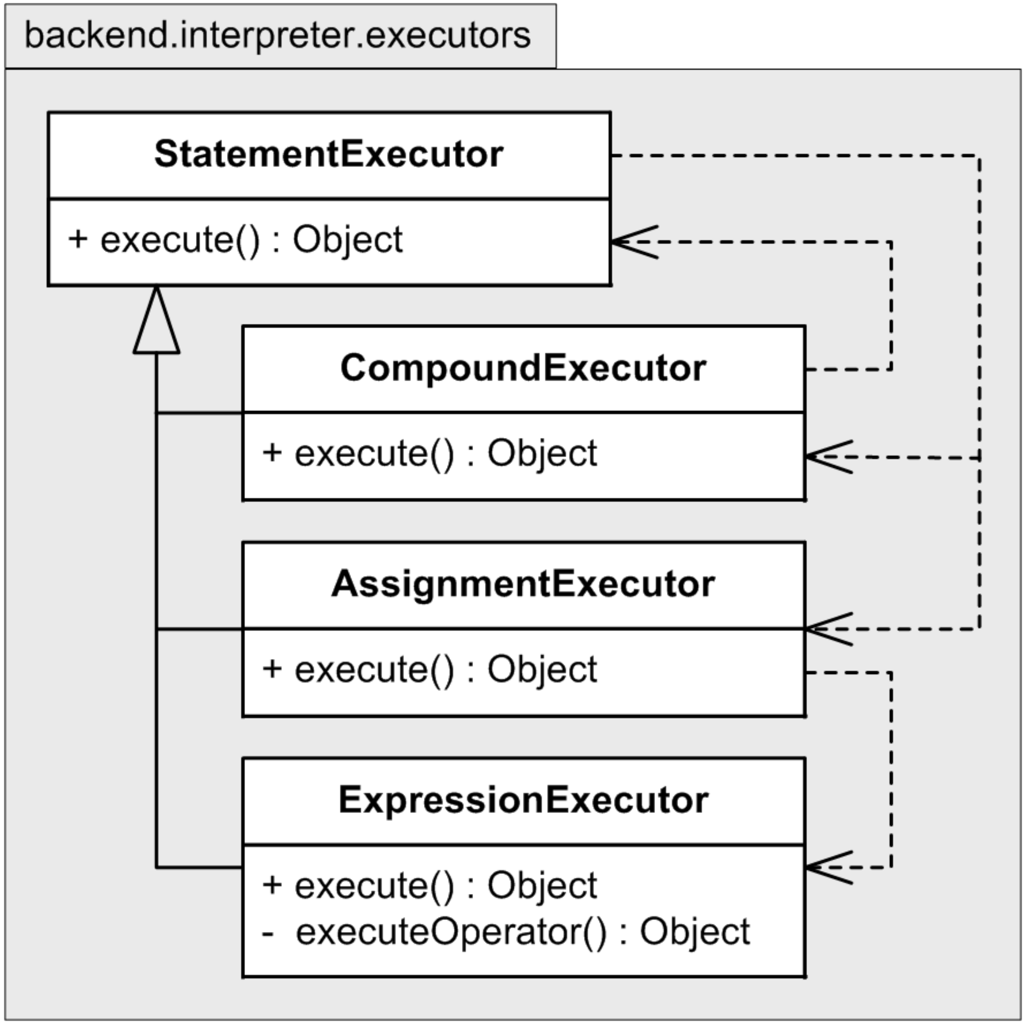


The Statement Executor Class



- Class **`StatementExecutor`** is a subclass of **`Executor`** which is a subclass of **`Backend`**.
  - Its **`execute()`** method interprets the parse tree whose root node is passed to it.
  - The return value is either the value of a computed expression, or null.

The Statement Executor Subclasses

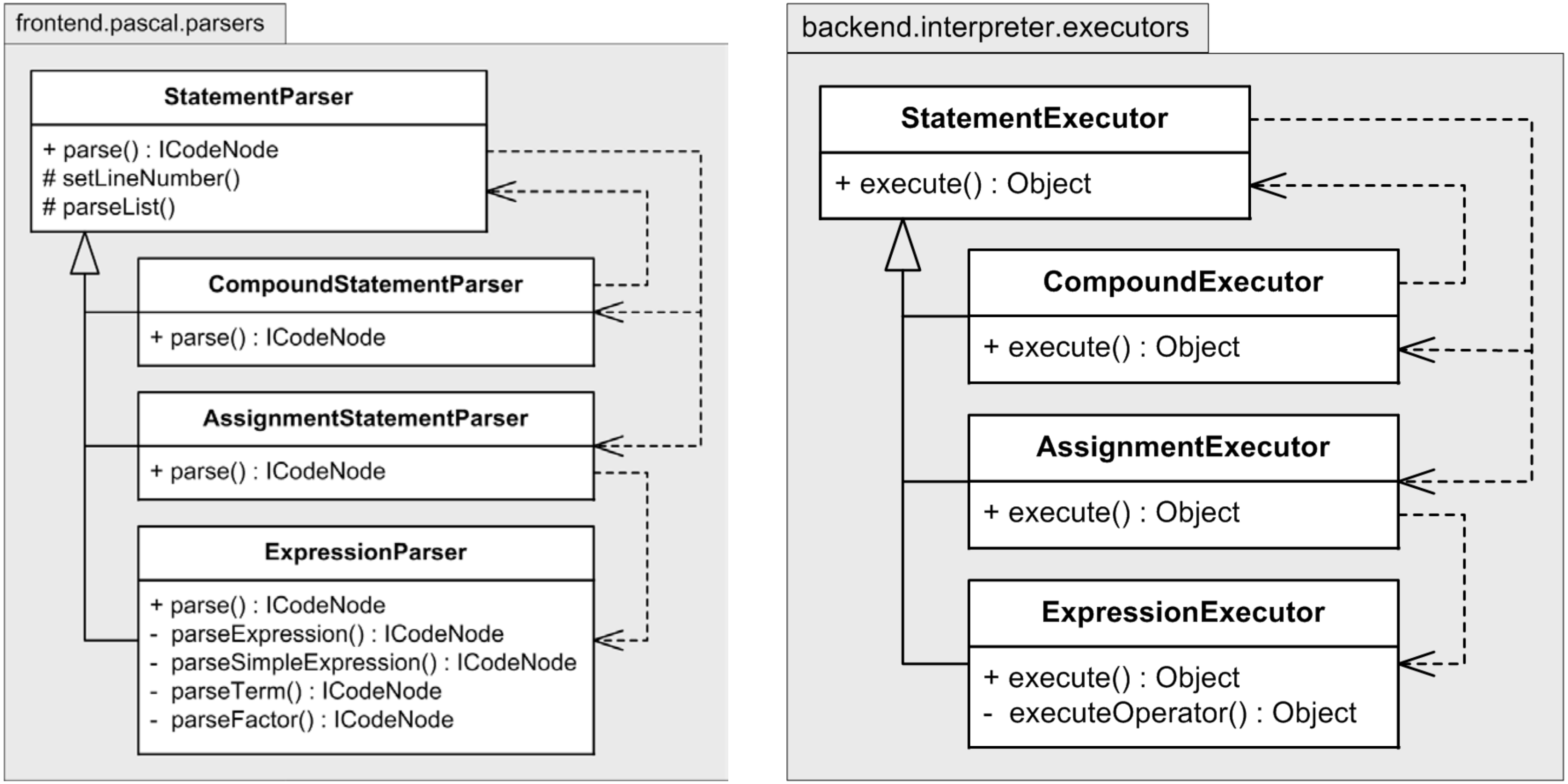


- **StatementExecutor** itself has subclasses:
  - **CompoundExecutor**
  - **AssignmentExecutor**
  - **ExpressionExecutor**

- The **execute()** method of each subclass also interprets the parse tree whose root node is passed to it.

- Note the dependency relationships among **StatementExecutor** and its subclasses.

More Architectural Symmetry



- The statement executor classes in the back end are symmetrical with the statement parser classes in the front end.

Runtime Error Handling

- Just as the front end has an error handler for syntax errors, the interpreter back end has an error handler for runtime errors.
  - Similar **`flag()`** method.
  - Here, *run time* means *the time when the interpreter is executing the source program*.

- Runtime error message format
  - Error message
  - Source line number where the error occurred

Runtime Error Messages

- Here are the errors and their messages that our interpreter will be able to detect and flag at run time.

```
public enum RuntimeErrorCode
{
    UNINITIALIZED_VALUE("Uninitialized value"),
    VALUE_RANGE("Value out of range"),
    INVALID_CASE_EXPRESSION_VALUE("Invalid CASE expression value"),
    DIVISION_BY_ZERO("Division by zero"),
    INVALID_STANDARD_FUNCTION_ARGUMENT("Invalid standard function argument"),
    INVALID_INPUT("Invalid input"),
    STACK_OVERFLOW("Runtime stack overflow"),
    UNIMPLEMENTED_FEATURE("Unimplemented runtime feature");

    ...
}
```

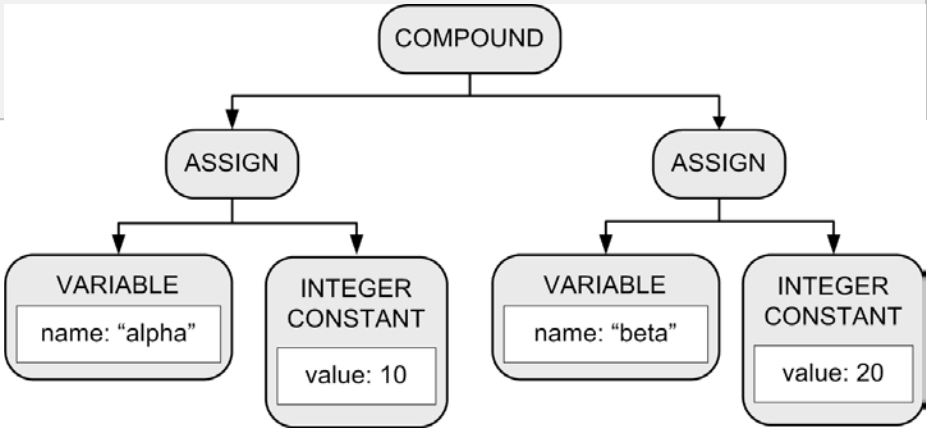
Class StatementExecutor

```
public Object execute(ICodeNode node)
{
    ICodeNodeTypeImpl nodeType = (ICodeNodeTypeImpl) node.getType();

    switch (nodeType) {

        case COMPOUND: {
            CompoundExecutor compoundExecutor = new CompoundExecutor(this);
            return compoundExecutor.execute(node);
        }

        case ASSIGN: {
            AssignmentExecutor assignmentExecutor = new AssignmentExecutor(this);
            return assignmentExecutor.execute(node);
        }

        ...
    }
}
```

- The node type tells which executor subclass to use.

Class **CompoundExecutor**

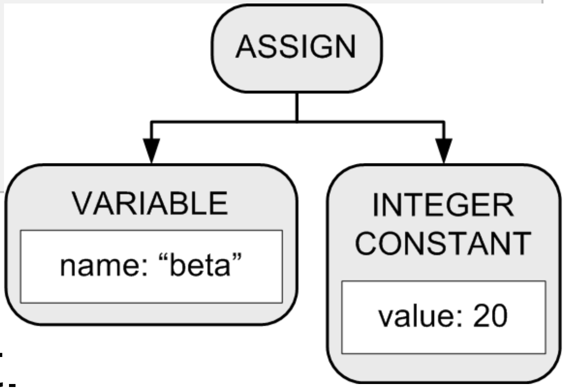
```
public Object execute(ICodeNode node)
{
    StatementExecutor statementExecutor = new StatementExecutor(this);
    ArrayList<ICodeNode> children = node.getChildren();

    for (ICodeNode child : children) {
        statementExecutor.execute(child);
    }

    return null;
}
```



- □ Get the list of all the child nodes of the COMPOUND node.
- □ Then call **`statementExecutor.execute()`** on each child.

Class AssignmentExecutor

```
public Object execute(ICodeNode node)
{
    ArrayList<ICodeNode> children = node.getChildren();
    ICodeNode variableNode = children.get(0);
    ICodeNode expressionNode = children.get(1);

    ExpressionExecutor expressionExecutor = new ExpressionExecutor(this);
    Object value = expressionExecutor.execute(expressionNode);

    SymTabEntry variableId = (SymTabEntry) variableNode.getAttribute(ID);
    variableId.setAttribute(DATA_VALUE, value);

    sendMessage(node, variableId.getName(), value);

    ++executionCount;
    return null;
}
```



- Temporary hack: Set the computed value into the symbol table.
- Send a message about the assignment.

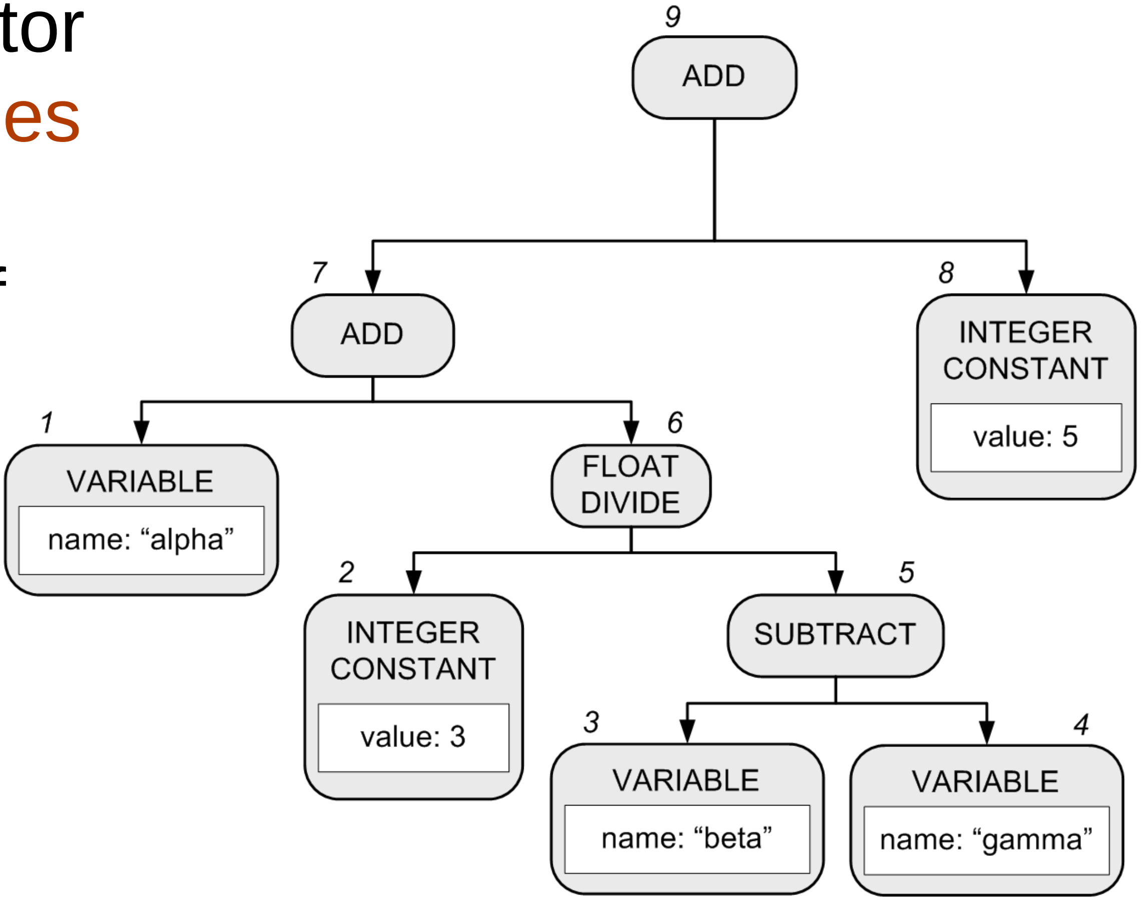
The Assignment Message

- Very useful for debugging.
- Necessary for now since we don’t have any other way to generate runtime output.
- Message format
  - Source line number
  - Name of the variable
  - Value of the expression

Executing Expressions

- Recall that Pascal’s operator precedence rules are encoded in the structure of the parse tree.
  - At run time, we do a postorder tree traversal.

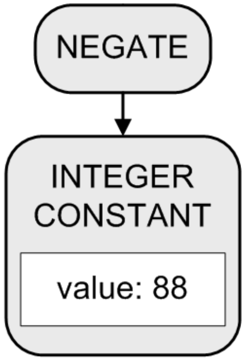
```
alpha + 3/(beta - gamma) + 5
```



Class `ExpressionExecutor`

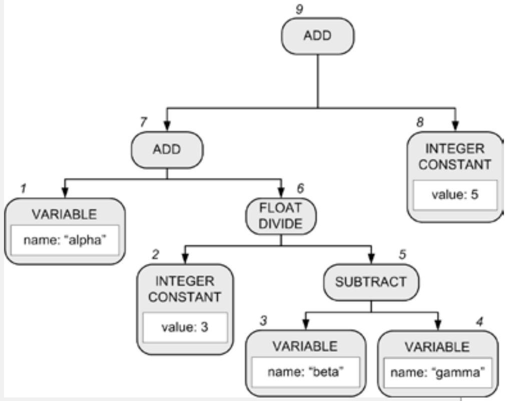
```
public Object execute(ICodeNode node)
{
    ICodeNodeTypeImpl nodeType = (ICodeNodeTypeImpl) node.getType();

    switch (nodeType) {
        ...
        case NEGATE: {

            // Get the NEGATE node's expression node child.
            ArrayList<ICodeNode> children = node.getChildren();
            ICodeNode expressionNode = children.get(0);

            // Execute the expression and return the negative of its value.
            Object value = execute(expressionNode);
            if (value instanceof Integer) {
                return -((Integer) value);
            }
            else {
                return -((Float) value);
            }
        }
        ...

        // Must be a binary operator.
        default: return executeBinaryOperator(node, nodeType);
    }
}
```

All node types: **VARIABLE**, **INTEGER_CONSTANT**, **REAL_CONSTANT**, **STRING_CONSTANT**, **NEGATE**, **NOT**, and the default.



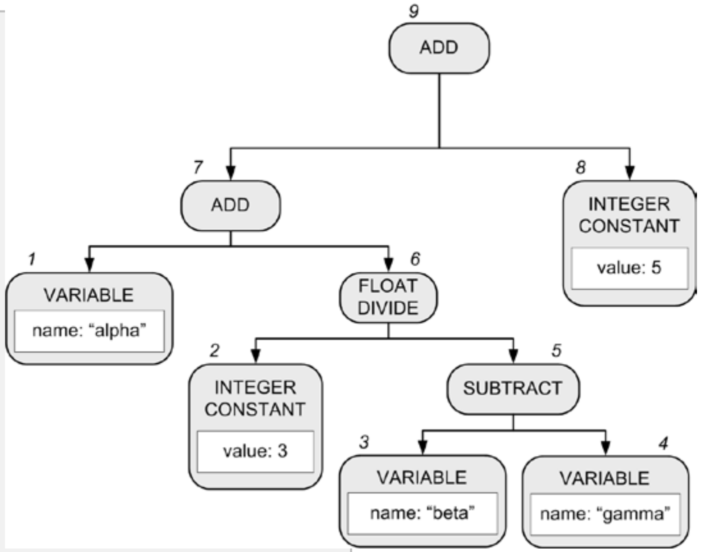
Method **executeBinaryOperator**

```
// Set of arithmetic operator node types.
private static final EnumSet<ICodeNodeTypeImpl> ARITH_OPS =
    EnumSet.of(ADD, SUBTRACT, MULTIPLY, FLOAT_DIVIDE, INTEGER_DIVIDE, MOD);

private Object executeBinaryOperator(ICodeNode node,
                                     ICodeNodeTypeImpl nodeType)
{
    // Get the two operand children of the operator node.
    ArrayList<ICodeNode> children = node.getChildren();
    ICodeNode operandNode1 = children.get(0);
    ICodeNode operandNode2 = children.get(1);

    // Operand values.
    Object operand1 = execute(operandNode1);
    Object operand2 = execute(operandNode2);

    boolean integerMode = (operand1 instanceof Integer) &&
                          (operand2 instanceof Integer);
    ...
}
```



Method **executeBinaryOperator**, *cont’d*

```
if (ARITH_OPS.contains(nodeType)) {

    if (integerMode) {
        int value1 = (Integer) operand1;
        int value2 = (Integer) operand2;

        switch (nodeType) {
            case ADD:      return value1 + value2;
            case SUBTRACT: return value1 - value2;
            case MULTIPLY: return value1 * value2;

            case FLOAT_DIVIDE: {
                if (value2 != 0) {
                    return ((float) value1)/((float) value2);
                }
                else {
                    errorHandler.flag(node, DIVISION_BY_ZERO, this);
                    return 0;
                }
            }

            case INTEGER_DIVIDE: ...
            case MOD: ...
        }
    } ...
```



Class **`ExpressionExecutor`**, *cont'd*

- Does not do type checking.
  - It's the job of the language-specific front end to flag any type incompatibilities.
- Does not know the operator precedence rules.
  - The front end must build the parse tree correctly.
  - The executor simply does a post-order tree traversal.

Class **`ExpressionExecutor`**, *cont'd*

- The bridge between the front end and the back end is the symbol table and the intermediate code (parse tree) in the intermediate tier.
  - Loose coupling (again!)

Simple Interpreter I

```
BEGIN
    BEGIN {Temperature conversions.}
        five  := -1 + 2 - 3 + 4 + 3;
        ratio := five/9.0;

        fahrenheit := 72;
        centigrade := (fahrenheit - 32)*ratio;

        centigrade := 25;
        fahrenheit := centigrade/ratio + 32;

        centigrade := 25;
        fahrenheit := 32 + centigrade/ratio
    END;

    {Runtime division by zero error.}
    dze := fahrenheit/(ratio - ratio);
```

continued …

Simple Interpreter I, *cont'd*

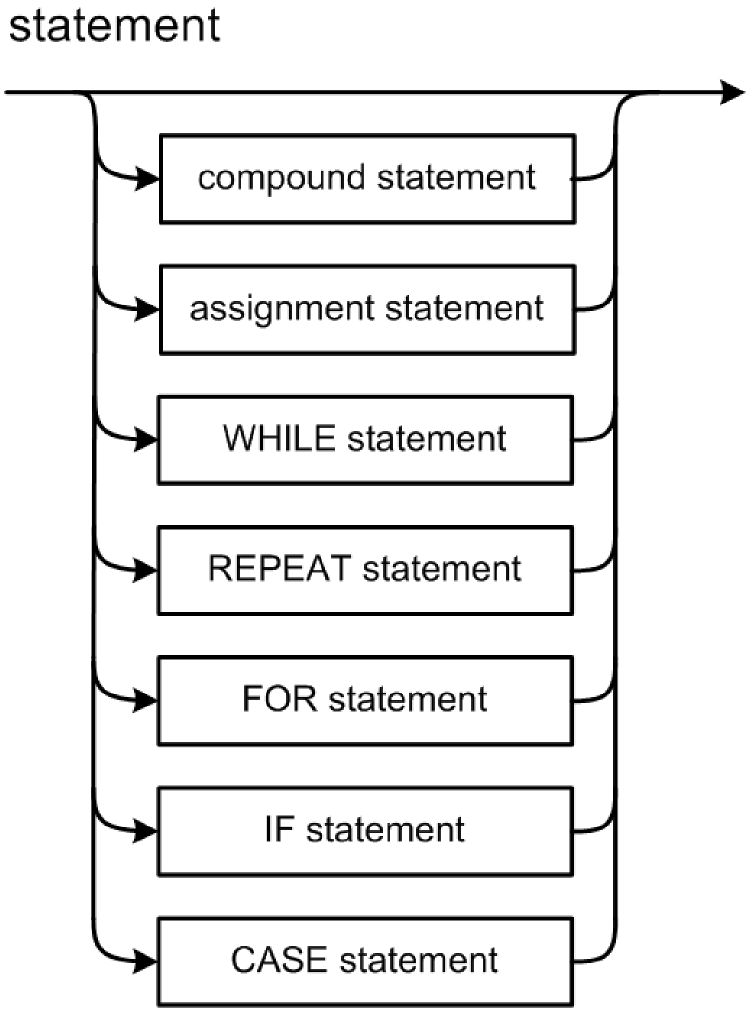
```
    BEGIN {Calculate a square root using Newton's method.}
        number := 4;
        root := number;
        root := (number/root + root)/2;
        root := (number/root + root)/2;
        root := (number/root + root)/2;
        root := (number/root + root)/2;
        root := (number/root + root)/2;
    END;

    ch  := 'x';
    str := 'hello, world'
END.
```

- Demo (Chapter 6)
  - `java -classpath classes Pascal execute assignments.txt`

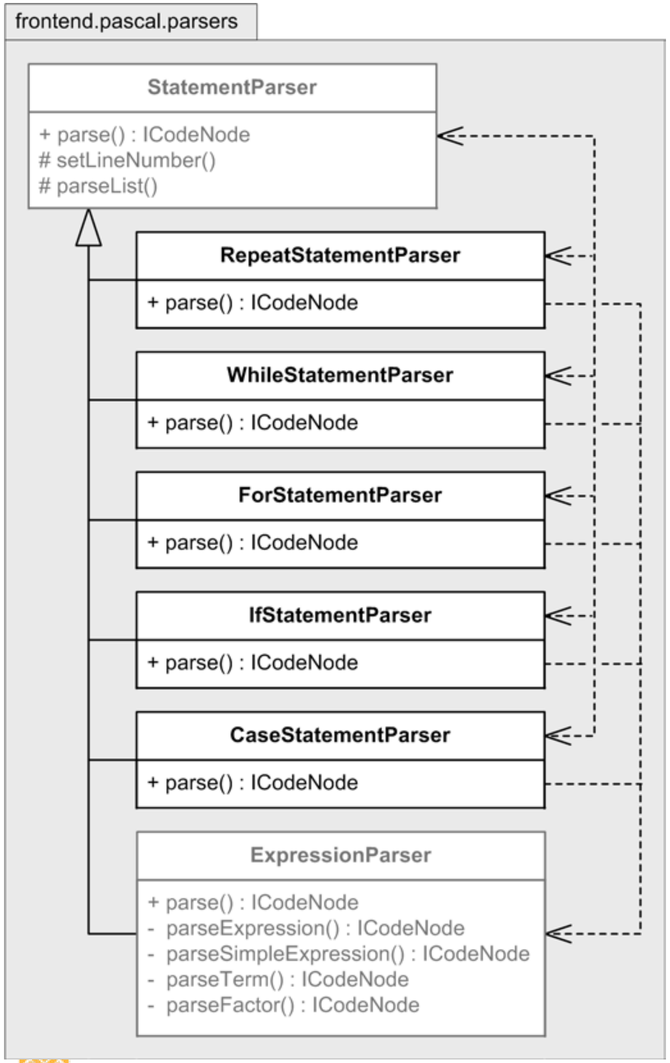
Pascal Control Statements

- Looping statements
  - `REPEAT UNTIL`
  - `WHILE DO`
  - `FOR TO`
  - `FOR DOWNTO`
- Conditional statements
  - `IF THEN`
  - `IF THEN ELSE`
  - `CASE`

Statement Syntax Diagram

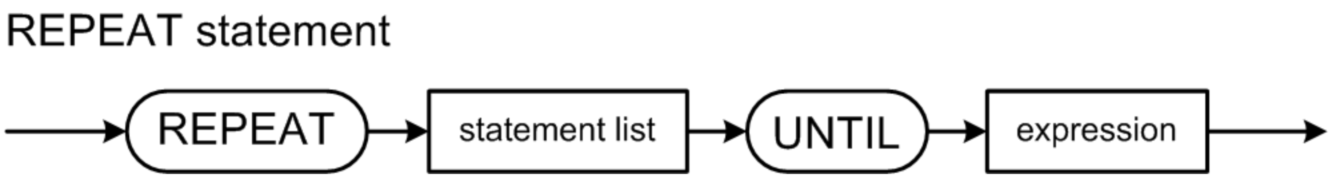


Pascal Statement Parsers



- New statement parser subclasses.
  - `RepeatStatementParser`
  - `WhileStatementParser`
  - `ForStatementParser`
  - `IfStatementParser`
  - `CaseStatementParser`
- Each `parse()` method builds a parse subtree and returns the root node.

REPEAT Statement



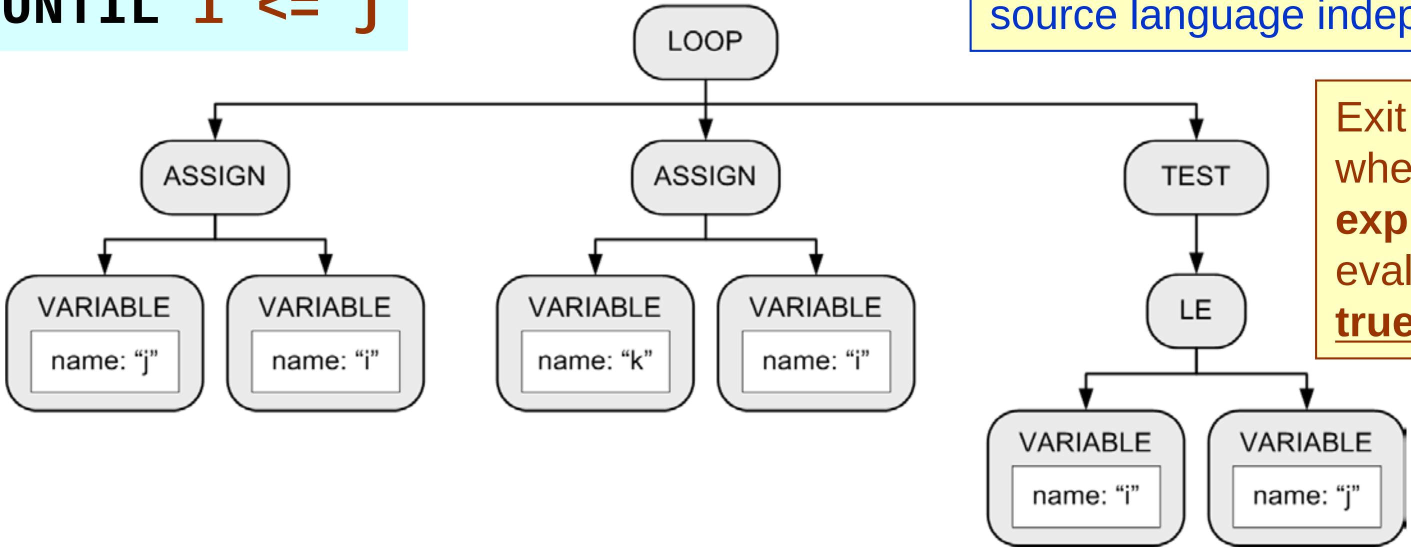
- Example:

```
REPEAT
      j := i;
      k := i
UNTIL i <= j
```

- Keep looping until the boolean expression becomes true.
  - Execute the loop at least once.

Use **LOOP** and **TEST** nodes for source language independence.

Exit the loop when the **test expression** evaluates to **true**.



Syntax Error Handling

- Recall that syntax error handling in the front end is a three-step process.

  1. Detect the error.
  2. Flag the error.
  3. Recover from the error.

- Good syntax error handling is important!

Options for Error Recovery

- Stop after the first error.
  - No error recovery at all.
  - Easiest for the compiler writer, annoying for the programmer.
  - Worse case: The compiler crashes or hangs.

- Become hopelessly lost.
  - Attempt to continue parsing the rest of the source program.
  - Spew out lots of irrelevant and meaningless error messages.
  - No error recovery here, either …
    - … but the compiler writer doesn’t admit it!

Options for Error Recovery, *cont'd*

- Skip tokens after the erroneous token until …
  - The parser finds a token it recognizes, and
  - It can safely resume syntax checking the rest of the source program.

Parser Synchronization

- Skipping tokens to reach a safe, recognizable place to resume parsing is known as synchronizing.
  - "Resynchronize the parser" after an error.

- Good error recovery with top-down parsers is more art than science.
  - How many tokens should the parser skip?
    - Skipping too many (the rest of the program?) can be considered "panic mode" recovery.
  - For this class, we'll take a rather simplistic approach to synchronization.

Method **synchronize()**

- The **synchronize()** method of class **PascalParserTD**.
  - Pass it an enumeration set of "good" token types.
  - The method skips tokens until it finds one that **is in** the set.

```
public Token synchronize(EnumSet syncSet)
    throws Exception
{
    Token token = currentToken();

    if (!syncSet.contains(token.getType())) {
        errorHandler.flag(token, UNEXPECTED_TOKEN, this);

        do {
            token = nextToken();
        } while (!(token instanceof EofToken) &&
                 !syncSet.contains(token.getType()));
    }

    return token;
}
```

Flag the **first** bad token.

Recover by skipping tokens **not in** the synchronization set.

Resume parsing at this token! (It's the first token after the error that **is in** the synchronization set.

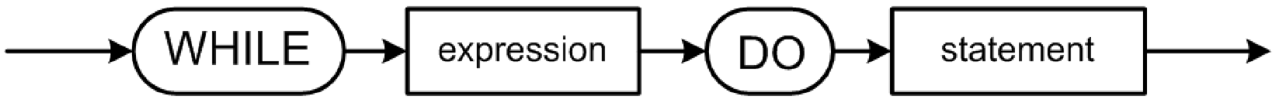
Pascal Syntax Checker II: **REPEAT**

- Demo (Chapter 7)

```
java -classpath classes Pascal compile -i repeat.txt
java -classpath classes Pascal compile –i repeaterrors.txt
```

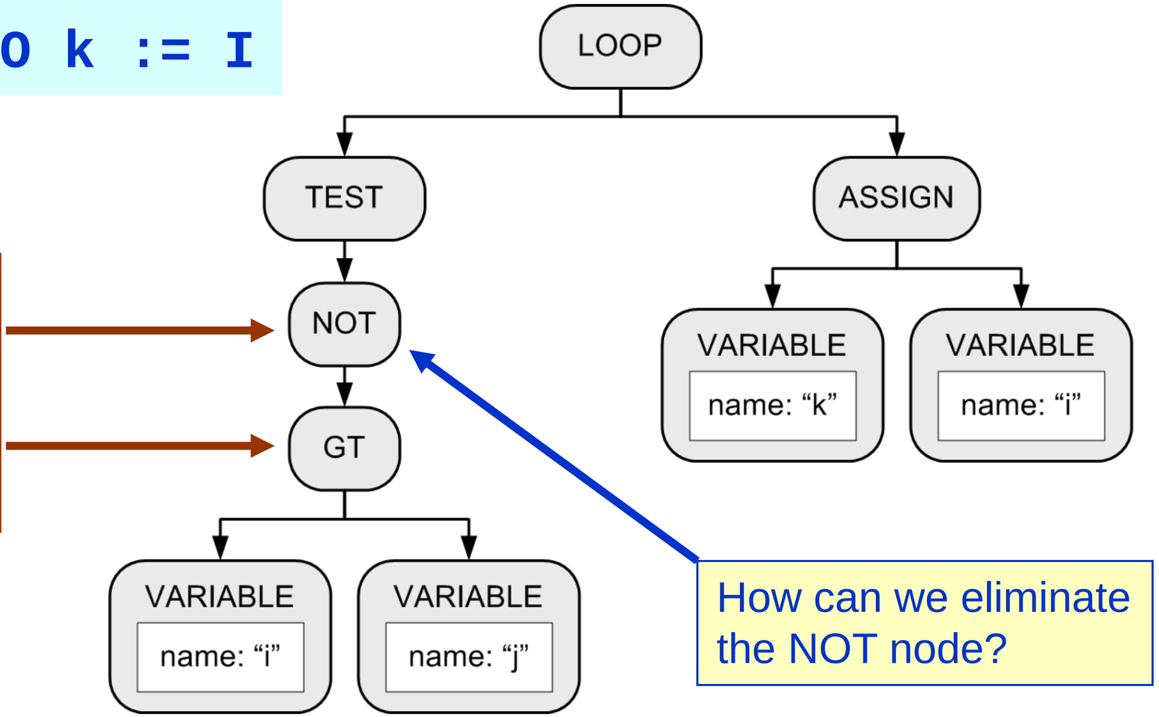
WHILE Statement

WHILE statement



- Example

```
WHILE i > j DO k := I
```



Exit the loop when the **test expression** evaluates to **false**.

How can we eliminate the NOT node?

Class WhileStatementParser

- From parent class **StatementParser**:

```
// Synchronization set for starting a statement.
protected static final EnumSet<PascalTokenType> STMT_START_SET =
    EnumSet.of(BEGIN, CASE, FOR, PascalTokenType.IF, REPEAT, WHILE,
               IDENTIFIER, SEMICOLON);

// Synchronization set for following a statement.
protected static final EnumSet<PascalTokenType> STMT_FOLLOW_SET =
    EnumSet.of(SEMICOLON, END, ELSE, UNTIL, DOT);
```

- In class **WhileStatementParser**:

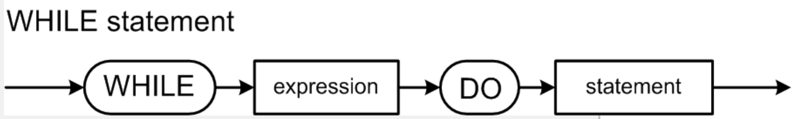
```
// Synchronization set for DO.
private static final EnumSet<PascalTokenType> DO_SET =
    StatementParser.STMT_START_SET.clone();
static {
    DO_SET.add(DO);
    DO_SET.addAll(StatementParser.STMT_FOLLOW_SET);
}
```

DO_SET contains all the tokens that can **start** a statement or **follow** a statement, plus the **DO** token.

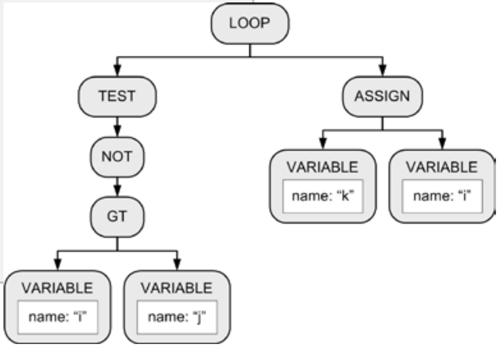
Class **WhileStatementParser**, *cont'd*

```
public ICodeNode parse(Token token)
    throws Exception
{
    token = nextToken();  // consume the WHILE

    ICodeNode loopNode = ICodeFactory.createICodeNode(LOOP);
    ICodeNode testNode = ICodeFactory.createICodeNode(TEST);
    ICodeNode notNode = ICodeFactory.createICodeNode(ICodeNodeTypeImpl.NOT);

    loopNode.addChild(testNode);
    testNode.addChild(notNode);

    ExpressionParser expressionParser = new ExpressionParser(this);
    notNode.addChild(expressionParser.parse(token));

    token = synchronize(DO_SET);
    if (token.getType() == DO) {
        token = nextToken();  // consume the DO
    }
    else {
        errorHandler.flag(token, MISSING_DO, this);
    }

    StatementParser statementParser = new StatementParser(this);
    loopNode.addChild(statementParser.parse(token));

    return loopNode;
}
```

We're in this method because the parser has already seen **WHILE**.



Synchronize the parser here! If the current token is not **DO**, then skip tokens until we find a token that is in **DO_SET**.



Pascal Syntax Checker II: **WHILE**

- We can recover (better) from syntax errors.
- Demo.
  - `java -classpath classes Pascal compile -i while.txt`
  - `java -classpath classes Pascal compile -i whileerrors.txt`