CMPE 152: Compiler Design

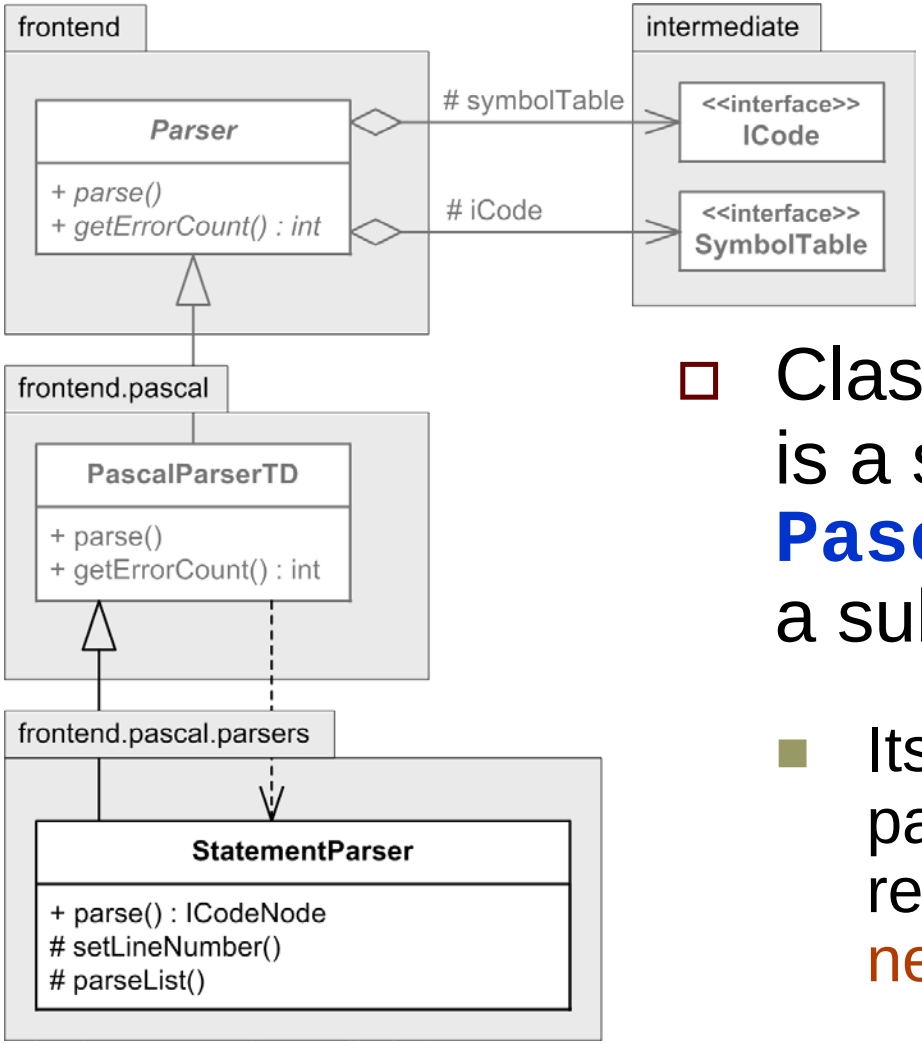
September 12 Class Meeting

Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak

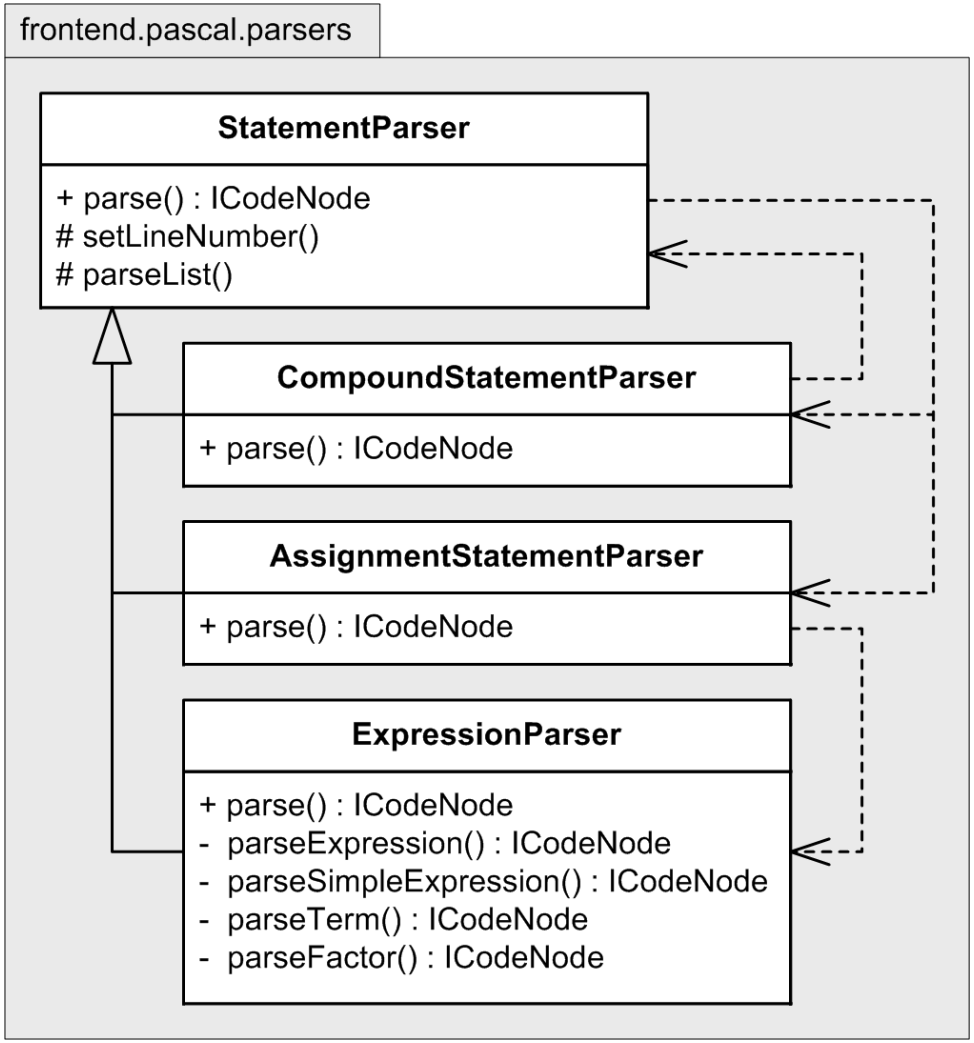
www.cs.sjsu.edu/~mak

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.

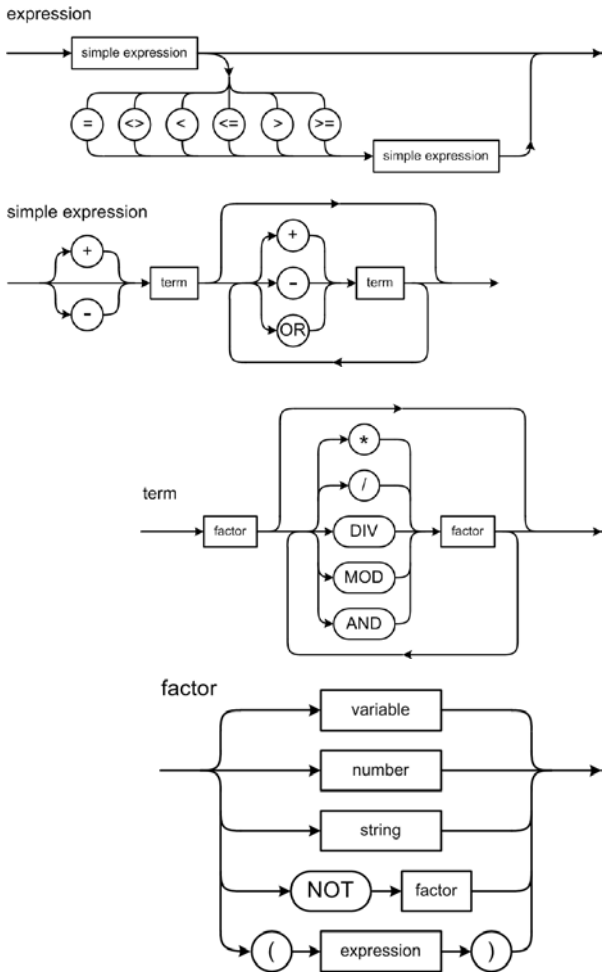
Statement Parser Subclasses



- **`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

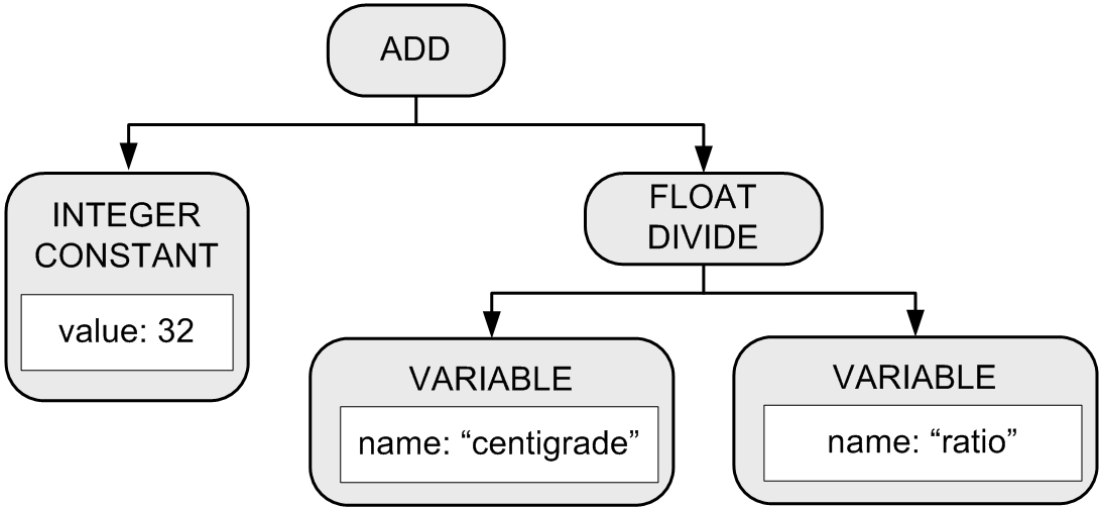
- Pascal statement parser subclass **`ExpressionParser`** has methods that correspond to the expression syntax diagrams:
  - **`parse_expression()`**
  - **`parse_simple_expression()`**
  - **`parse_term()`**
  - **`parse_factor()`**



- 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`**

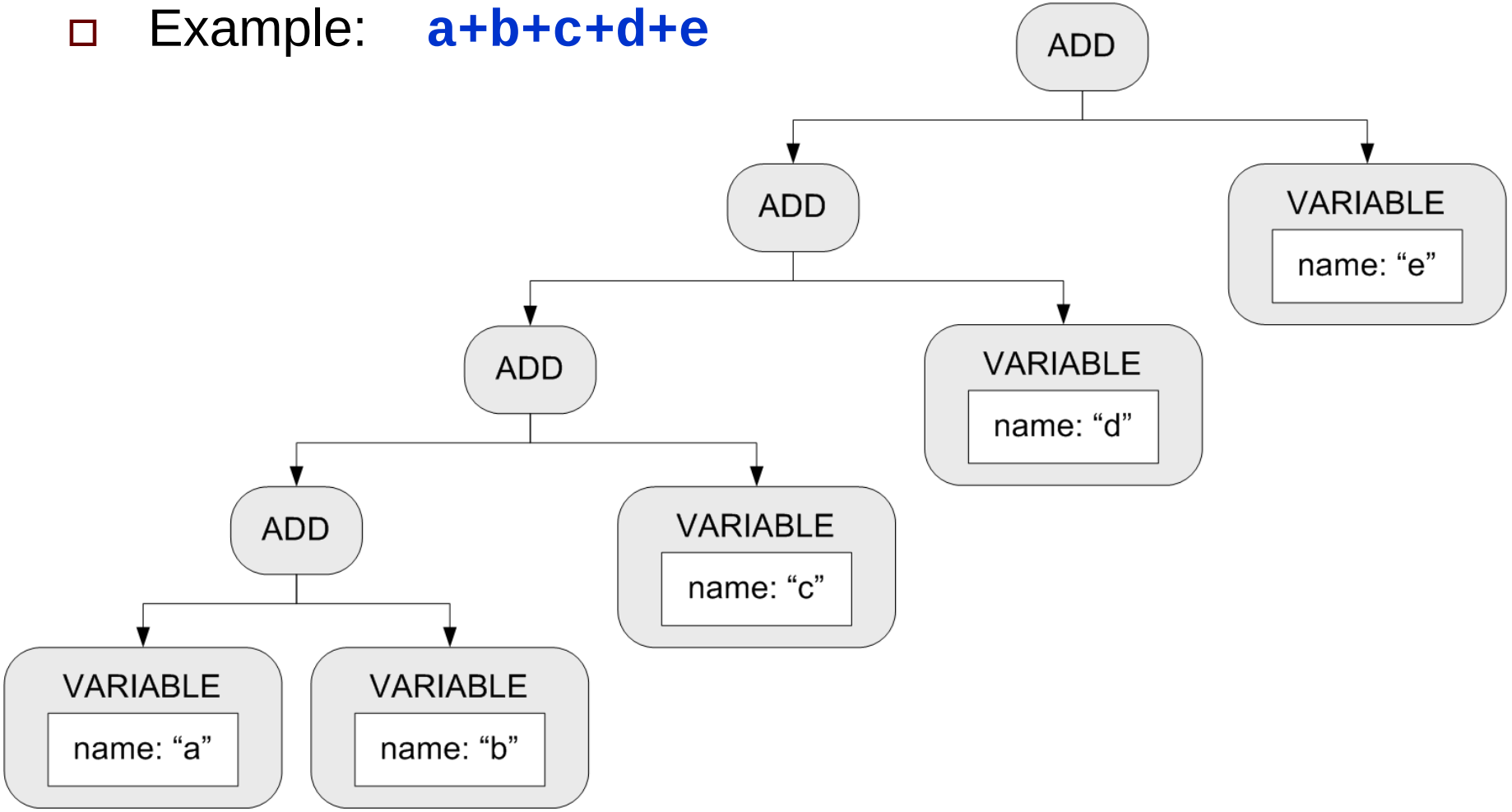


Do a **postorder traversal** of the parse tree.

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

Parsing Expressions, *cont'd*

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



Example: Function **parseExpression()**

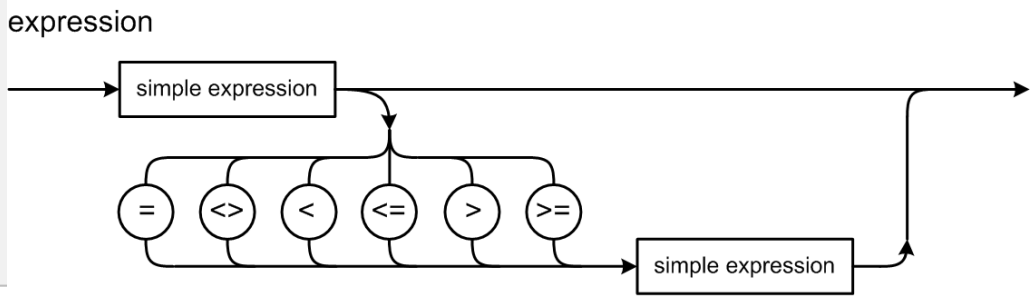
- First, we need to map Pascal token types to parse tree node types.
  - Node types need to be language-independent.

```
private:
    static map<PascalTokenType, ICodeNodeTypeImpl> REL_OPS_MAP;
```

```
REL_OPS_MAP[PT_EQUALS]         = NT_EQ;
REL_OPS_MAP[PT_NOT_EQUALS]     = NT_NE;
REL_OPS_MAP[PT_LESS_THAN]      = NT_LT;
REL_OPS_MAP[PT_LESS_EQUALS]    = NT_LE;
REL_OPS_MAP[PT_GREATER_THAN]   = NT_GT;
REL_OPS_MAP[PT_GREATER_EQUALS] = NT_GE;
```

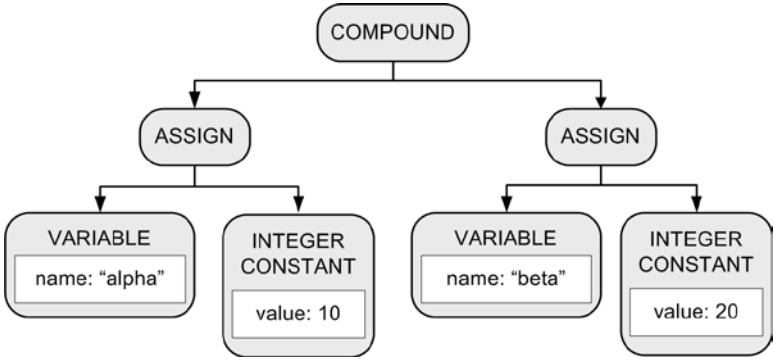
Method **parseExpression()**, *cont'd*

```
ICodeNode *ExpressionParser::parse_expression(Token *token) throw (string)
{
    ICodeNode *root_node = parse_simple_expression(token);

    token = current_token();
    TokenType token_type = token->get_type();

    map<PascalTokenType, ICodeNodeTypeImpl>::iterator it =
                          REL_OPS_MAP.find((PascalTokenType) token_type);
    if (it != REL_OPS_MAP.end())
    {
        ICodeNodeType node_type = (ICodeNodeType) it->second;
        ICodeNode *op_node = ICodeFactory::create_icode_node(node_type);
        op_node->add_child(root_node);

        token = next_token(token);  // consume the operator

        op_node->add_child(parse_simple_expression(token));

        root_node = op_node;
    }

    return root_node;
}
```



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:

```
./Chapter5cpp execute -i assignments.txt
```

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

```
bool intermediate = flags.find('i') != string::npos;
...
if (intermediate)
{
    ParseTreePrinter *tree_printer = new ParseTreePrinter();
    tree_printer->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!

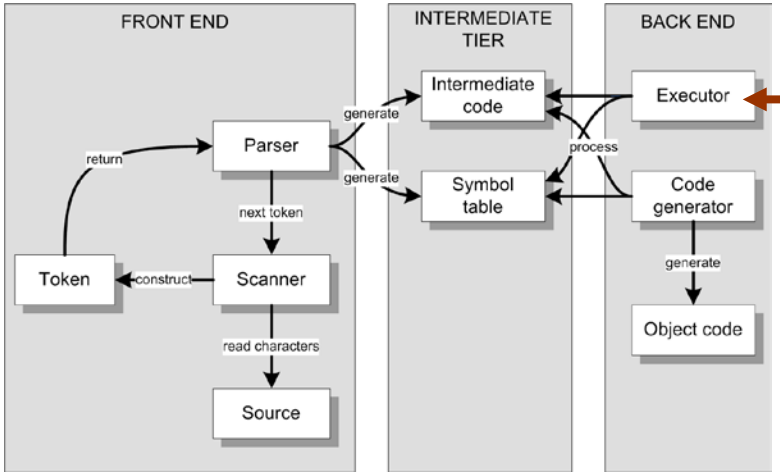
Temporary Hacks for Now

- 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).

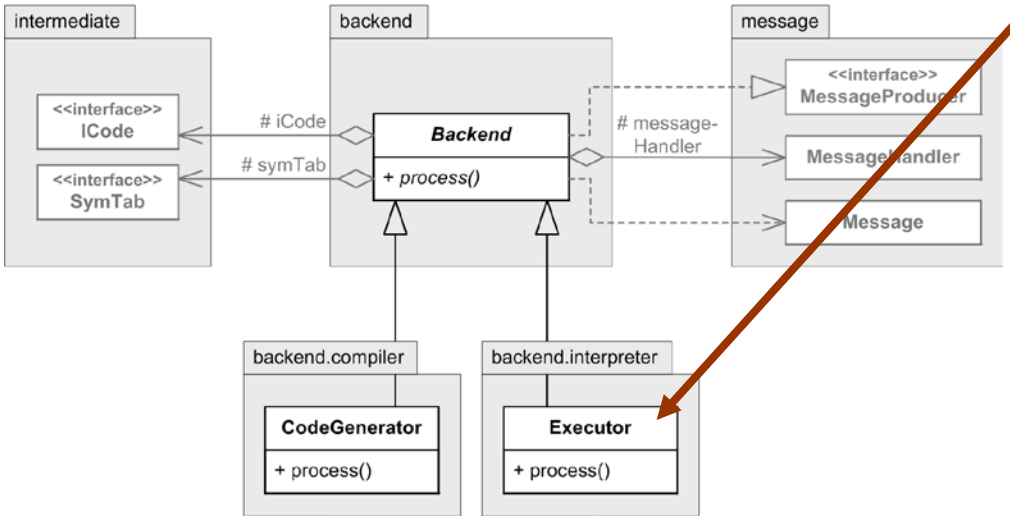
A New Temporary Hack

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

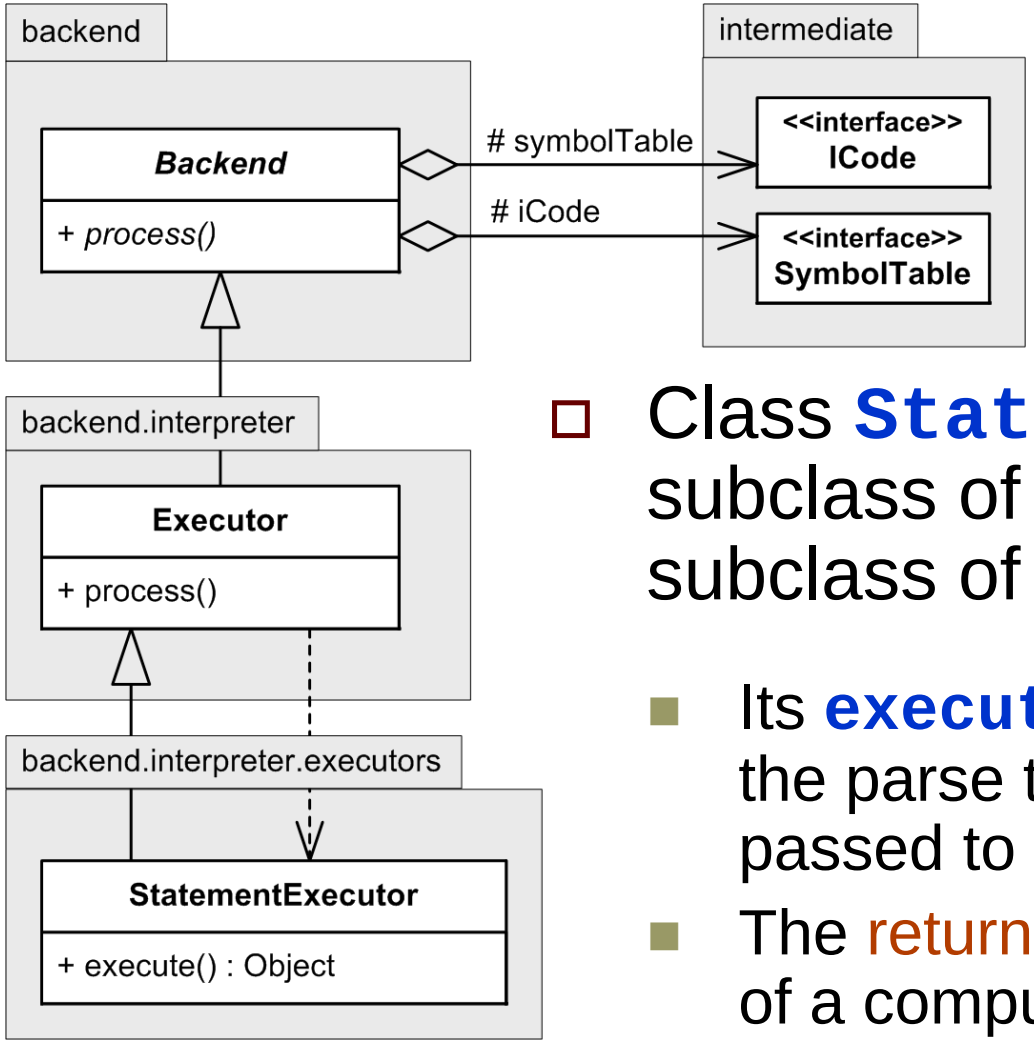
Quick Review of the Framework



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

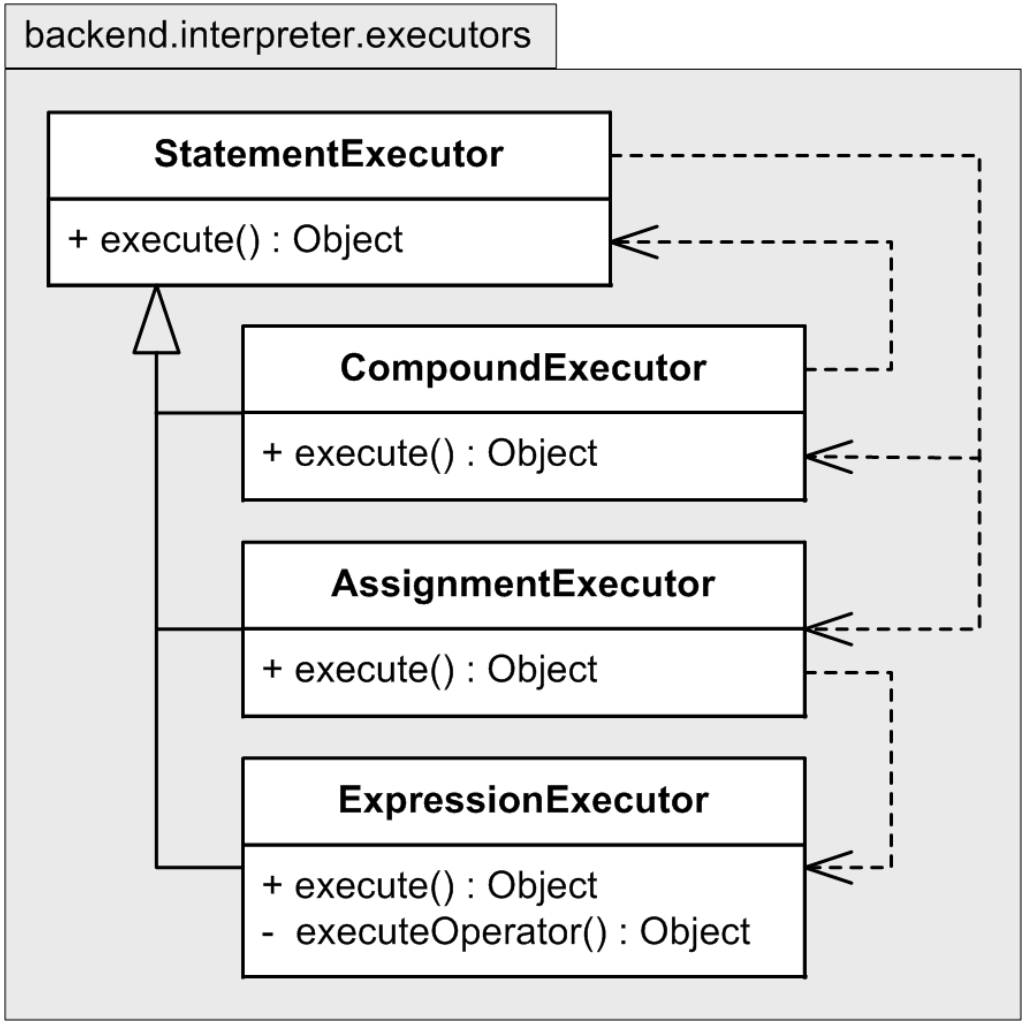


The Statement Executor Class



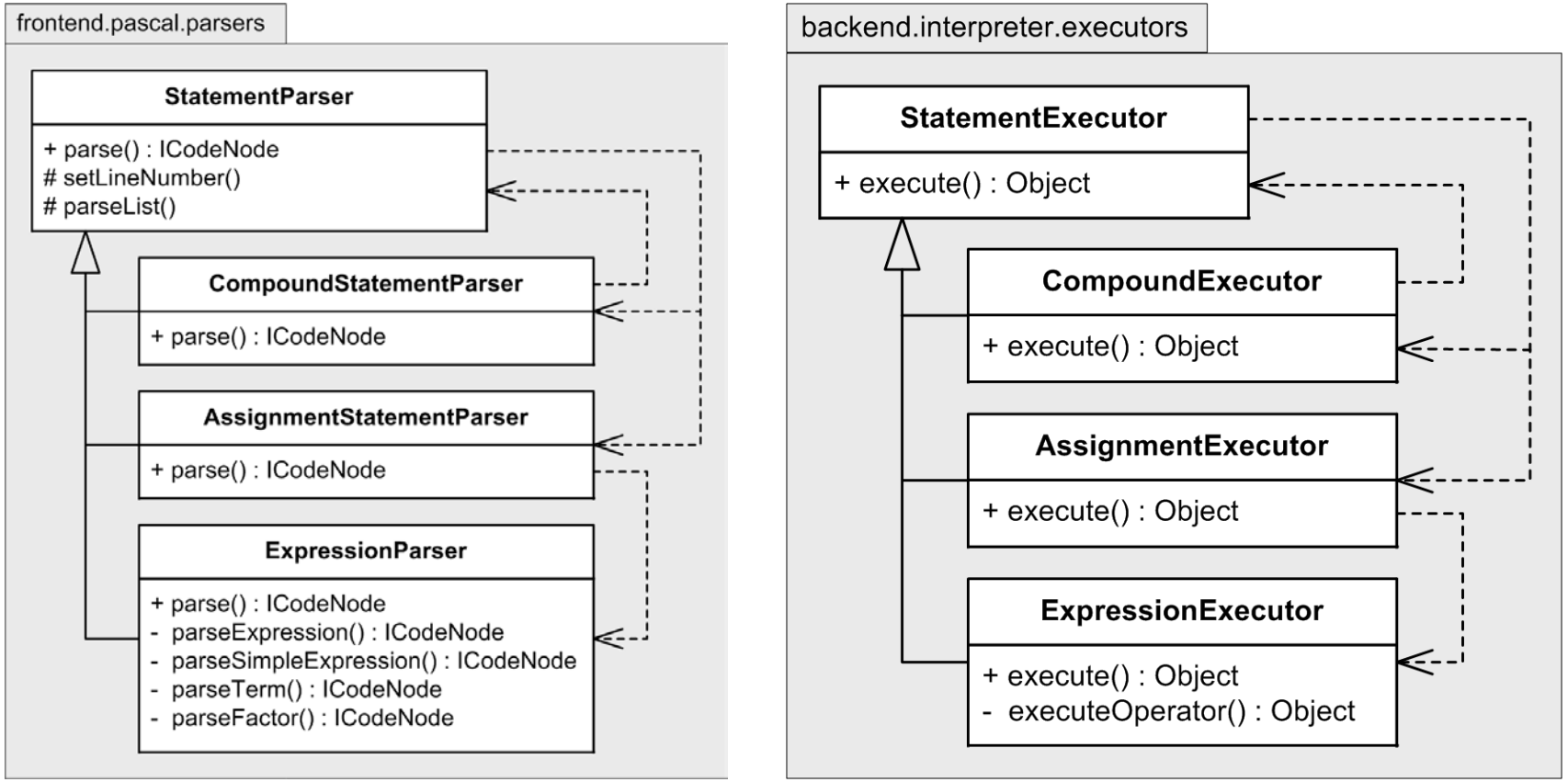
- Class **`StatementExecutor`** is a subclass of **`Executor`** which is a subclass of **`Backend`**.
  - Its **`execute()`** method interprets the parse tree whose root node is passed to it.
  - The return value is either the value of a computed expression, or null.

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.

```
enum class RuntimeErrorCode
{
    UNINITIALIZED_VALUE,
    VALUE_RANGE,
    INVALID_CASE_EXPRESSION_VALUE,
    DIVISION_BY_ZERO,
    INVALID_STANDARD_FUNCTION_ARGUMENT,
    INVALID_INPUT,
    STACK_OVERFLOW,
    UNIMPLEMENTED_FEATURE,
};
```

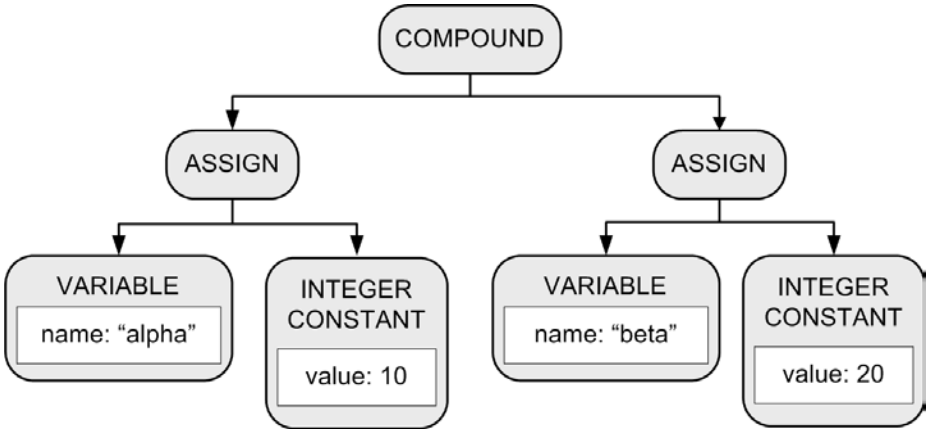
Class **StatementExecutor**

```
DataValue *StatementExecutor::execute(ICodeNode *node)
{
    ICodeNodeTypeImpl node_type = (ICodeNodeTypeImpl) node->get_type();

    ...

    switch (node_type)
    {
        case NT_COMPOUND:
        {
            CompoundExecutor compound_executor(this);
            return compound_executor.execute(node);
        }

        case NT_ASSIGN:
        {
            AssignmentExecutor assignment_executor(this);
            return assignment_executor.execute(node);
        }

        ...
    }
}
```

The **node_type** tells which executor subclass to use.

Class **CompoundExecutor**

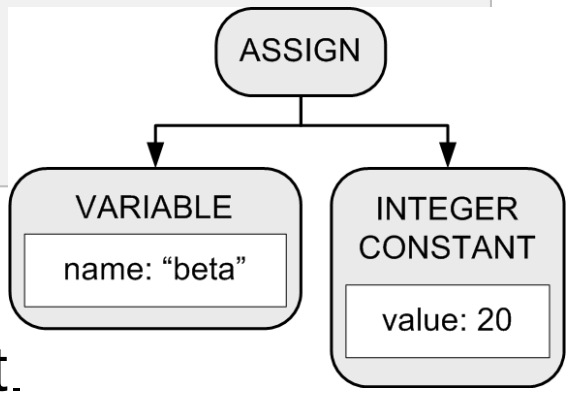
```
DataValue *CompoundExecutor::execute(ICodeNode *node)
{
    StatementExecutor statement_executor(this);
    vector<ICodeNode *> children = node->get_children();
    for (ICodeNode *child : children) statement_executor.execute(child);

    return nullptr;
}
```



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

Class **AssignmentExecutor**

```
DataValue *AssignmentExecutor::execute(ICodeNode *node)
{
    vector<ICodeNode *> children = node->get_children();
    ICodeNode *variable_node = children[0];
    ICodeNode *expression_node = children[1];

    ExpressionExecutor expression_executor(this);
    DataValue *result_value = expression_executor.execute(expression_node);

    NodeValue *node_value = variable_node->get_attribute((ICodeKey) ID);
    SymTabEntry *id = node_value->id;
    id->set_attribute((SymTabKey) DATA_VALUE, new EntryValue(result_value));

    send_assignment_message(node, id->get_name(), result_value);

    ++execution_count;
    return nullptr;
}
```



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

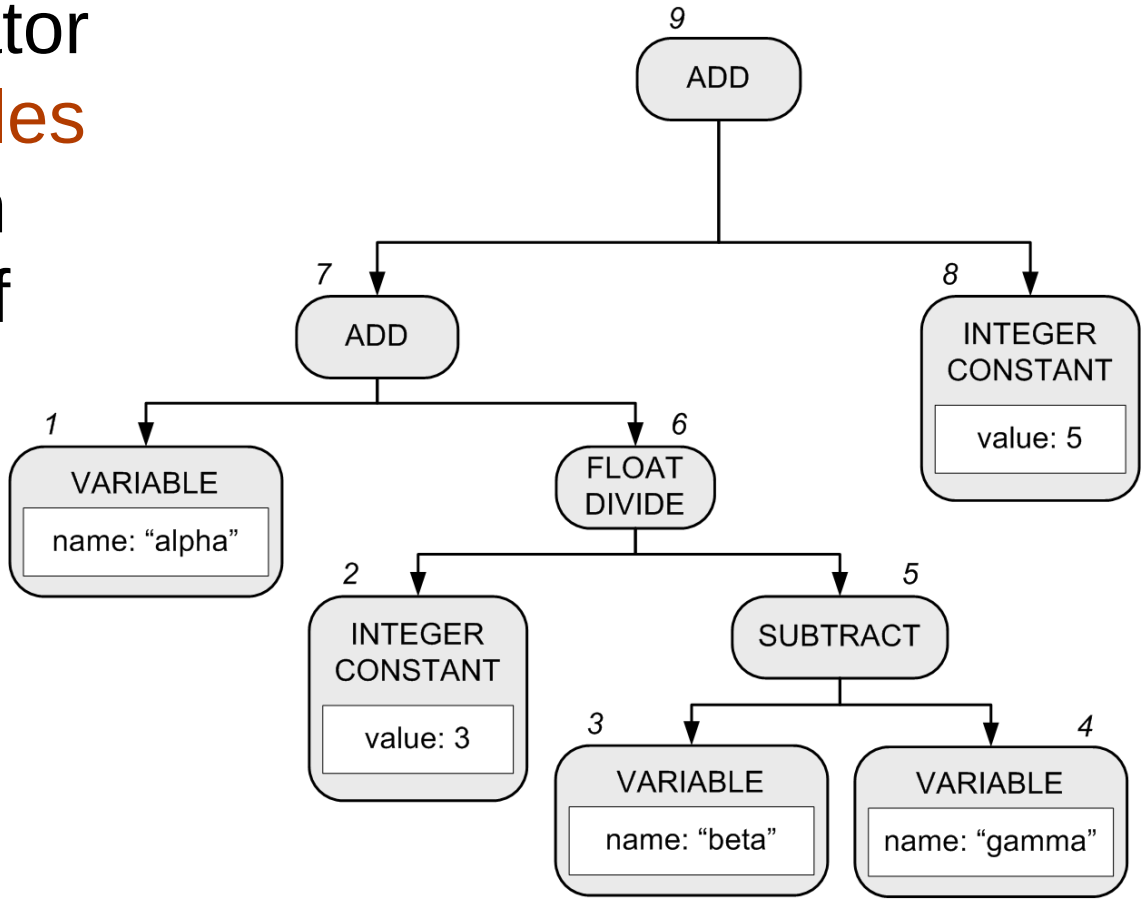
The Assignment Message

- 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.

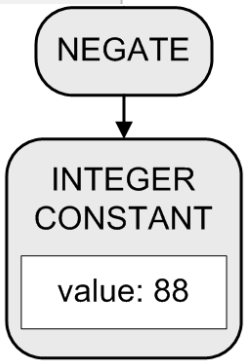
```
alpha + 3/(beta - gamma) + 5
```



Class **ExpressionExecutor**

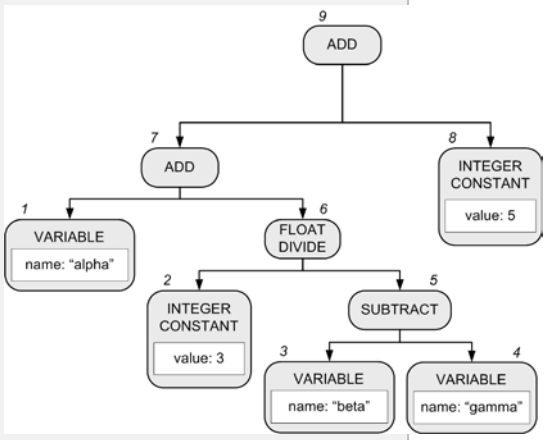
```
public DataValue *ExpressionExecutor::execute(ICodeNode *node)
{
    ICodeNodeTypeImpl node_type = (ICodeNodeTypeImpl) node->get_type();

    switch (node_type)
    {
        ...
        case NT_NEGATE:
        {
            // Get the NEGATE node's expression node child.
            vector<ICodeNode *> children = node->get_children();
            ICodeNode *expression_node = children[0];

            // Execute the expression and return the negative of its value.
            DataValue *result_value = execute(expression_node);
            return (result_value->type == INTEGER)
                                ? new DataValue(-result_value->i)
                                : new DataValue(-result_value->f);
        }
        ...

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

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



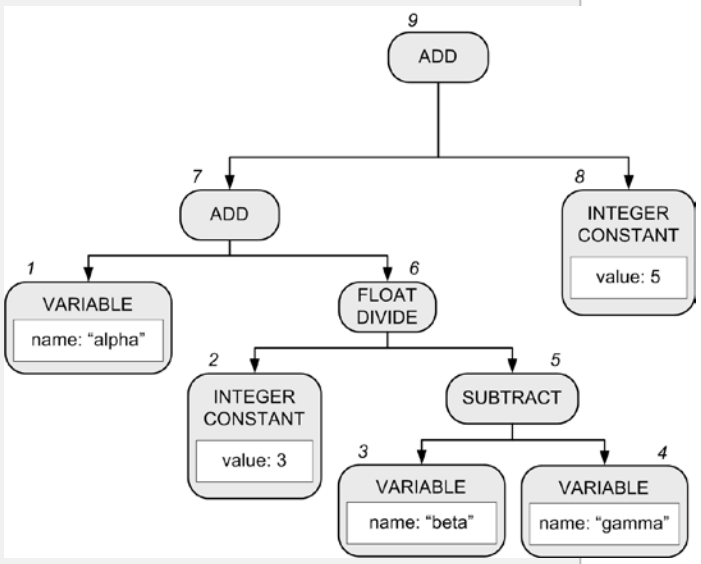
Method **executeBinaryOperator**

```
// Set of arithmetic operator node types.
set<ICodeNodeTypeImpl> ExpressionExecutor::ARITH_OPS =
{
    NT_ADD, NT_SUBTRACT, NT_MULTIPLY,
    NT_FLOAT_DIVIDE, NT_INTEGER_DIVIDE, NT_MOD,
};

DataValue *ExpressionExecutor::execute_binary_operator(
                        ICodeNode *node, const ICodeNodeTypeImpl node_type)
{
    // Get the two operand children of the operator node.
    vector<ICodeNode *> children = node->get_children();
    ICodeNode *operand_node1 = children[0];
    ICodeNode *operand_node2 = children[1];

    // Operands.
    DataValue *operand1 = execute(operand_node1);
    DataValue *operand2 = execute(operand_node2);

    bool integer_mode = (operand1->type == INTEGER) &&
                        (operand2->type == INTEGER);
     ...
}
```



Method **executeBinaryOperator**, *cont'd*

```
if (ARITH_OPS.find(node_type) != ARITH_OPS.end())
{
    if (integer_mode)
    {
        int value1 = operand1->i;
        int value2 = operand2->i;

        switch (node_type)
        {
            case NT_ADD:
                return new DataValue(value1 + value2);

            case NT_SUBTRACT:
                return new DataValue(value1 - value2);

            case NT_MULTIPLY:
                return new DataValue(value1 * value2);

            case NT_FLOAT_DIVIDE:
            {
                // Check for division by zero.
                if (value2 != 0)
                {
                    return new DataValue(((float) value1) /
                                         ((float) value2));
                }
                else
                {
                    error_handler.flag(node, DIVISION_BY_ZERO, this);
                    return new DataValue(0);
                }
            }
            ...
        }
    }
}
```



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`