CMPE 152: Compiler Design

September 7 Class Meeting

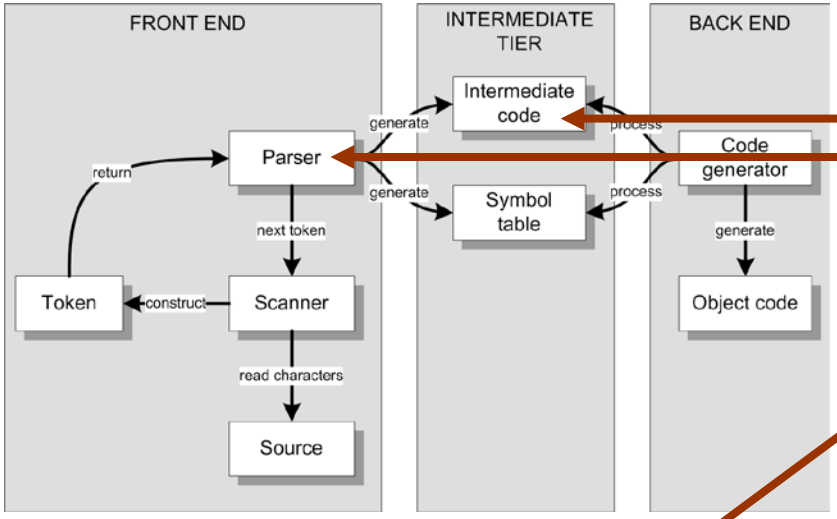
Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak

www.cs.sjsu.edu/~mak

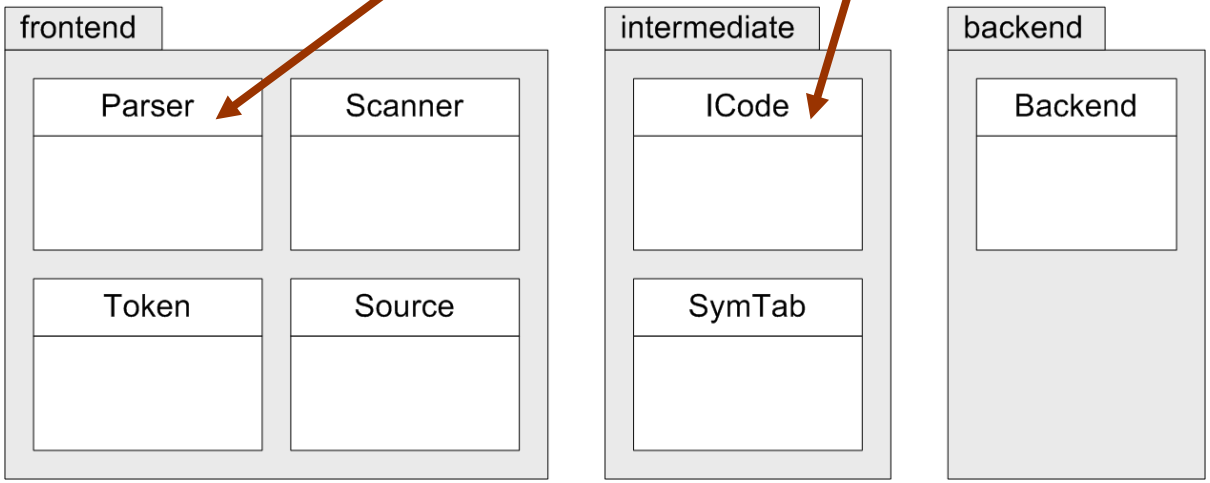
Quick Review of the Framework

FROM:



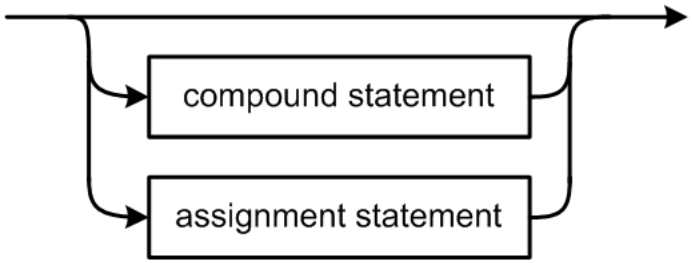
Next topic:
Parsing assignment statements and expressions, and generating parse trees.

TO:

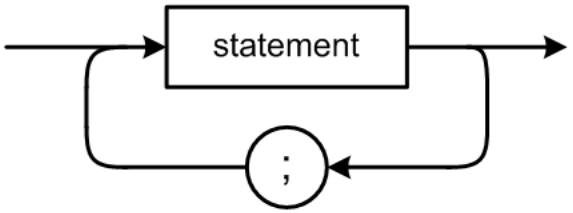


Pascal Statement Syntax Diagrams

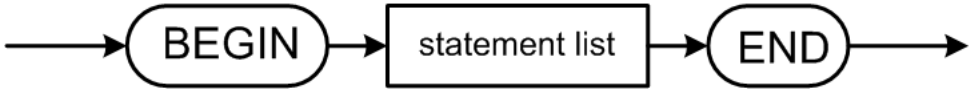
statement



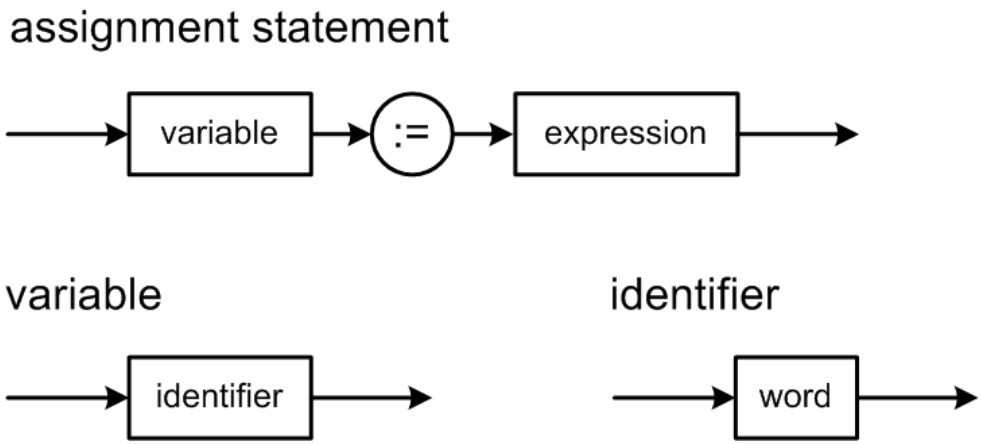
statement list



compound statement

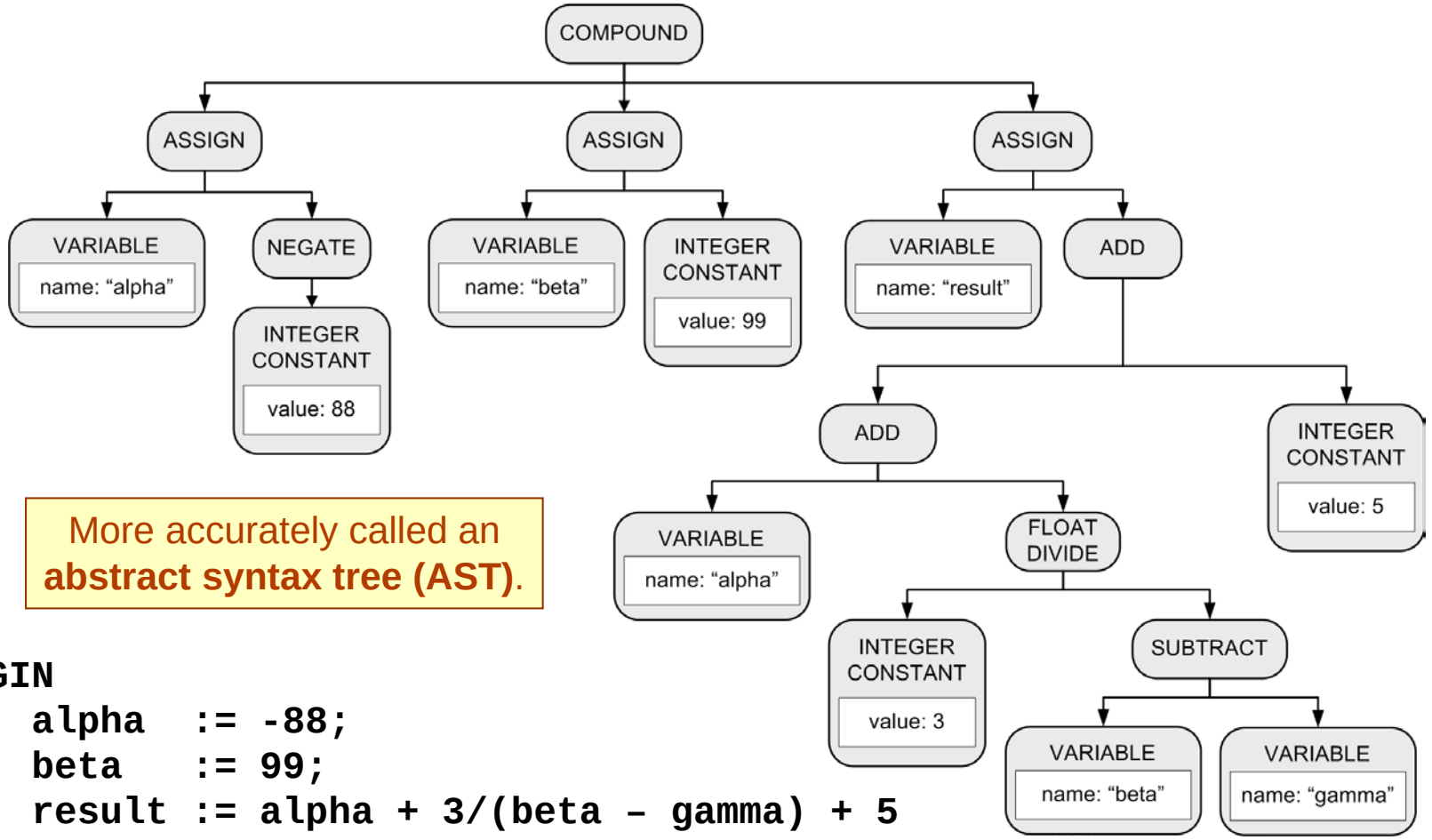


Pascal Statement Syntax Diagrams, *cont'd*



For now,
greatly simplified!

Parse Tree: Conceptual Design



More accurately called an **abstract syntax tree (AST)**.

```
BEGIN
    alpha  := -88;
    beta   := 99;
    result := alpha + 3/(beta – gamma) + 5
END
```

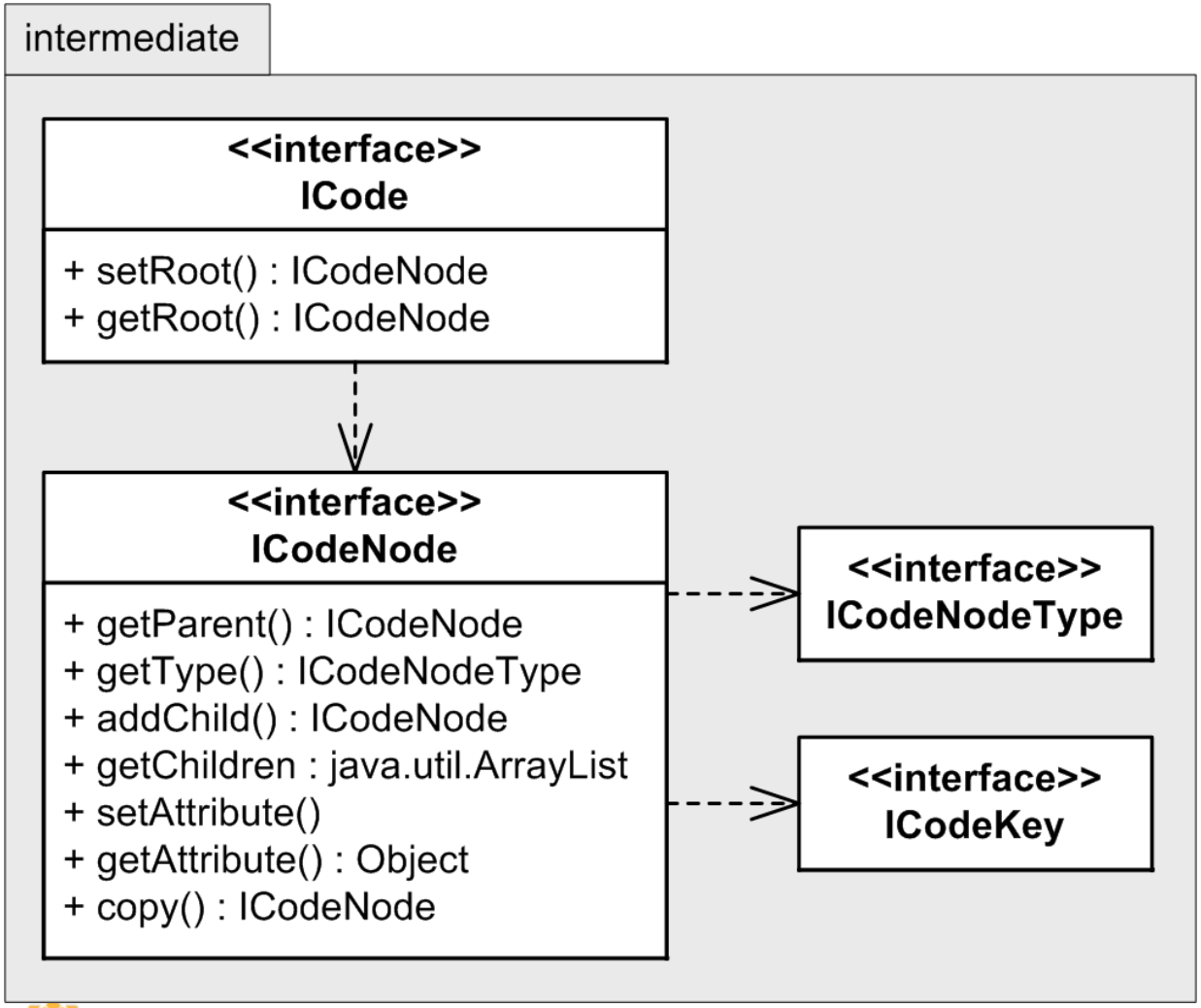
Parse Tree: Conceptual Design

- At the conceptual design level,
  we don't care *how* we implement the tree.
  - This should remind you of how
    we first designed the symbol table.

Parse Tree: Basic Tree Operations

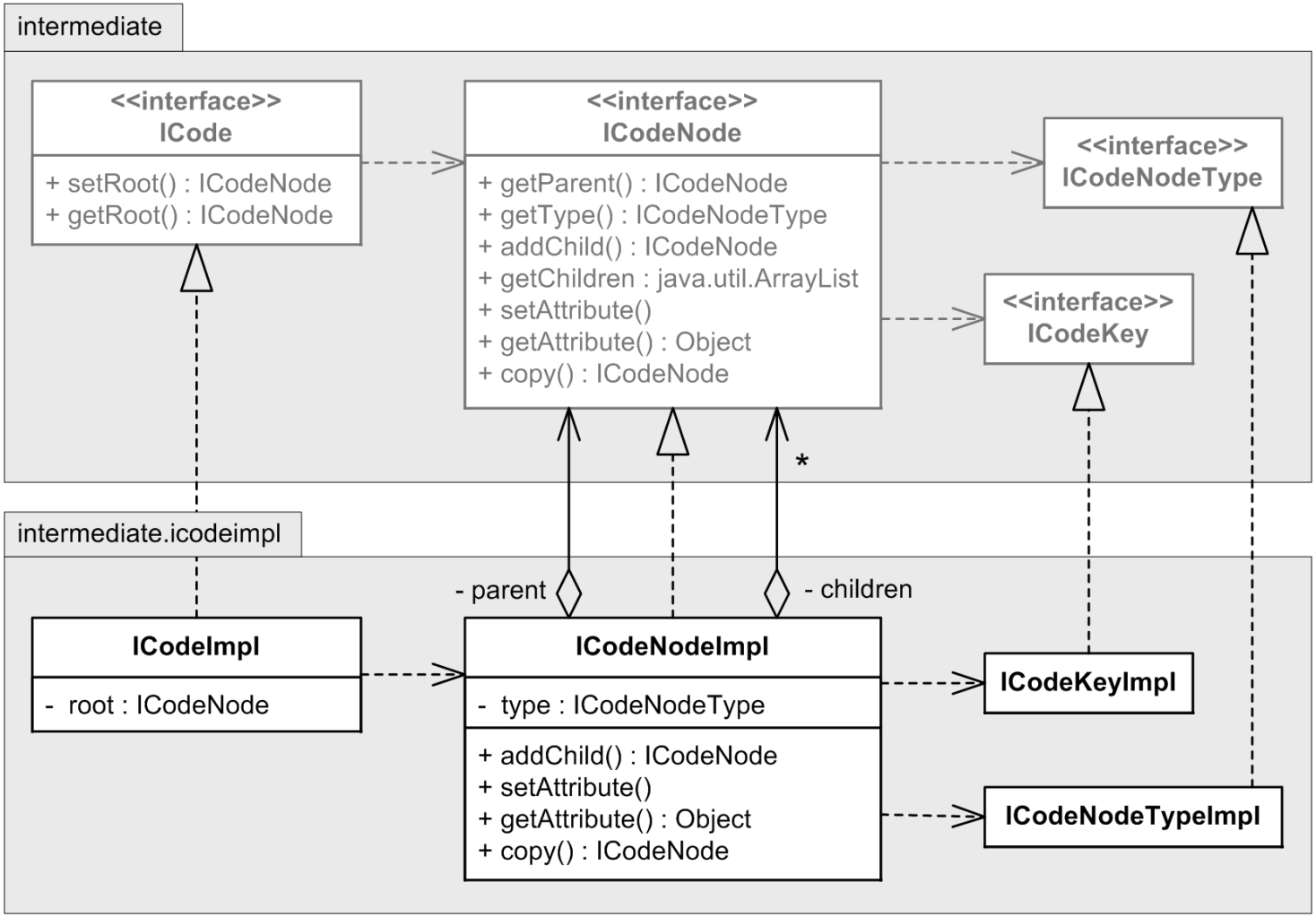
- Create a new node.
- Create a copy of a node.
- Set and get the root node of a parse tree.
- Set and get an attribute value in a node.
- Add a child node to a node.
- Get the list of a node's child nodes.
- Get a node's parent node.

Intermediate Code Interfaces



Goal: Keep it source language-independent.

Intermediate Code Implementations



An Intermediate Code Factory Class

```
ICode *ICodeFactory::create_icode()
{
    return new ICodeImpl();
}

ICodeNode *ICodeFactory::create_icode_node(const ICodeNodeType type)
{
    return new ICodeNodeImpl(type);
}
```

Coding to the Interfaces (Again)

```
// Create the ASSIGN node.
ICodeNode *assign_node =
        ICodeFactory::create_icode_node((ICodeNodeType) NT_ASSIGN);

...

// Create the variable node and set its name attribute.
ICodeNode *variable_node =
        ICodeFactory::create_icode_node(
                        (ICodeNodeType) NT_VARIABLE);
NodeValue *node_value = new NodeValue();
node_value->id = target_id;
variable_node->set_attribute((ICodeKey) ID, node_value);

// The ASSIGN node adopts the variable node as its first child.
assign_node->add_child(variable_node);
```

Intermediate Code (ICode) Node Types

```
enum class ICodeNodeTypeImpl
{
    // Program structure
    PROGRAM, PROCEDURE, FUNCTION,

    // Statements
    COMPOUND, ASSIGN, LOOP, TEST, CALL, PARAMETERS,
    IF, SELECT, SELECT_BRANCH, SELECT_CONSTANTS, NO_OP,

    // Relational operators
    EQ, NE, LT, LE, GT, GE, NOT,

    // Additive operators
    ADD, SUBTRACT, OR, NEGATE,

    // Multiplicative operators
    MULTIPLY, INTEGER_DIVIDE, FLOAT_DIVIDE, MOD, AND,

    // Operands
    VARIABLE, SUBSCRIPTS, FIELD,
    INTEGER_CONSTANT, REAL_CONSTANT,
    STRING_CONSTANT, BOOLEAN_CONSTANT,

    // WRITE parameter
    WRITE_PARM,
};
```

Do not confuse **node types** (**`ASSIGN`**, **`ADD`**, etc.) with **data types** (integer, real, etc.).

We use the enumerated type **`ICodeNodeTypeImpl`** for node types which is different from the enumerated type **`PascalTokenType`** to help maintain source language independence.

Intermediate Code Node Implementation

```
class ICodeNodeImpl : public ICodeNode
{
public:
    ICodeNodeImpl(const ICodeNodeType type);
    ...

protected:
    map<ICodeKey, NodeValue *> contents;
    ICodeNodeType type;             // node type
    ICodeNode *parent;              // node's parent
    vector<ICodeNode *> children;   // node's children
}
```

```
ICodeNodeImpl::ICodeNodeImpl(const ICodeNodeType type)
    : type(type), parent(nullptr)
{
}
```

- Each node has a m**ap<ICodeKey, NodeValue *>**.
- Each node has an **vector<ICodeNode *>** of child nodes.

A Parent Node Adopts a Child Node

```
ICodeNode *ICodeNodeImpl::add_child(ICodeNode *node)
{
    if (node != nullptr)
    {
        children.push_back(node);
        node->parent = this;
    }
    return node;
}
```

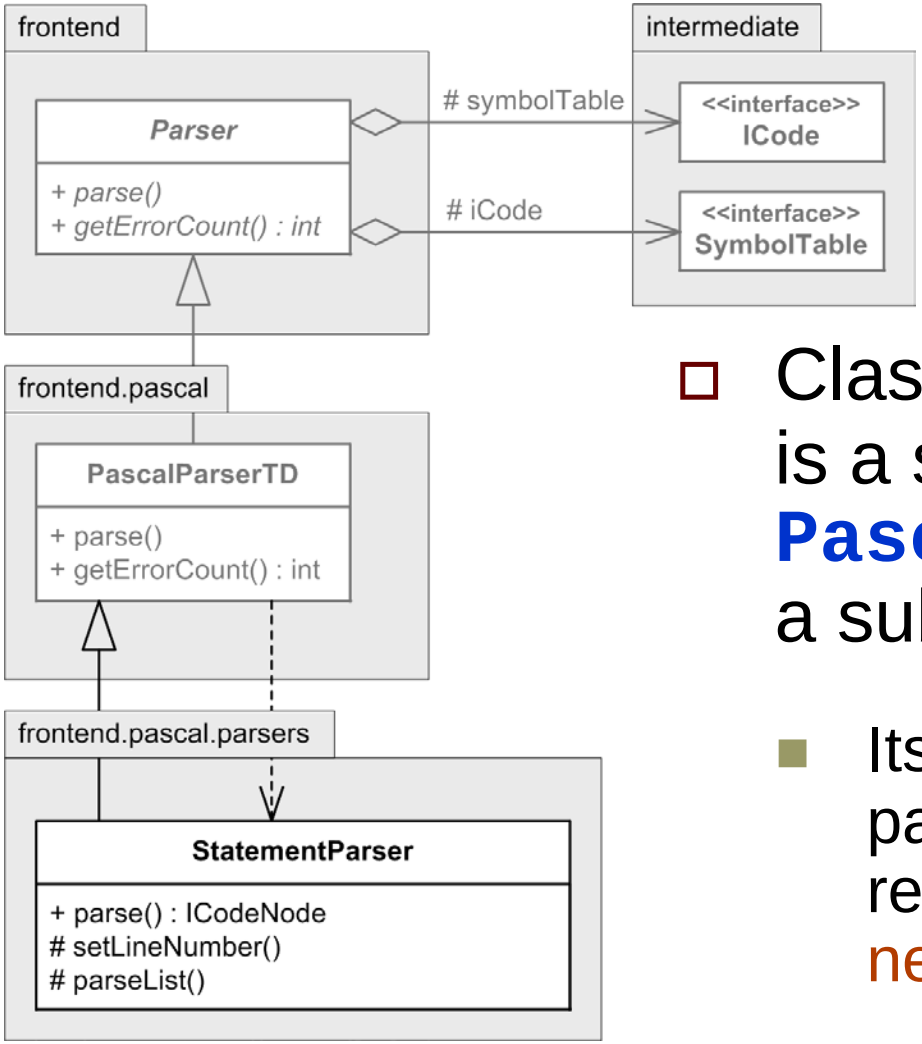
- When a parent node adds a child node, we can say that the parent node "adopts" the child node.
- Keep the parse tree implementation simple!

What Attributes to Store in a Node?

```
enum class ICodeKeyImpl
{
    LINE, ID, LEVEL, VALUE,
};
```

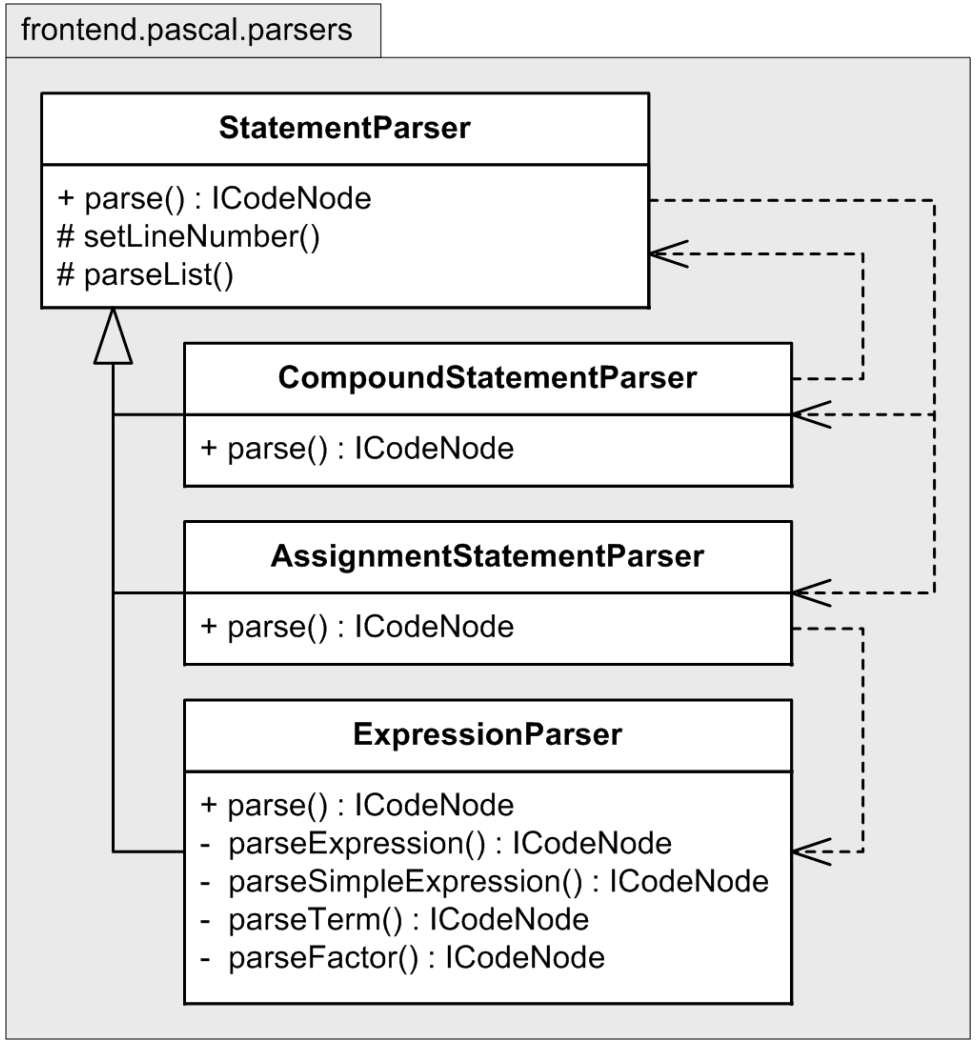
- Not much! Not every node will have these attributes.
  - **LINE**: statement line number
  - **ID**: symbol table entry of an identifier
  - **VALUE**: data value
- Most of the information about what got parsed is encoded in the node type and in the tree structure.

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:
  - **`CompoundStatement-Parser`**
  - **`AssignmentStatement-Parser`**
  - **`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.

Building a Parse Tree

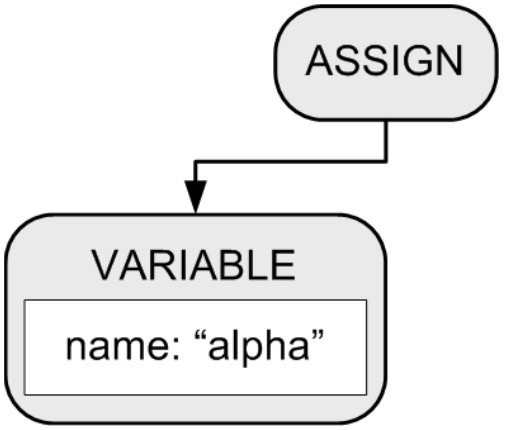
- Each **`parse()`** method builds a subtree and returns the root node of the new subtree.
- The caller of the **`parse()`** method adopts the subtree's root node as a child of the subtree that the caller is building.
  - The caller then returns the root node of its subtree to its caller.
  - This process continues until the entire source has been parsed and we have the entire parse tree.

Building a Parse Tree

- Example:

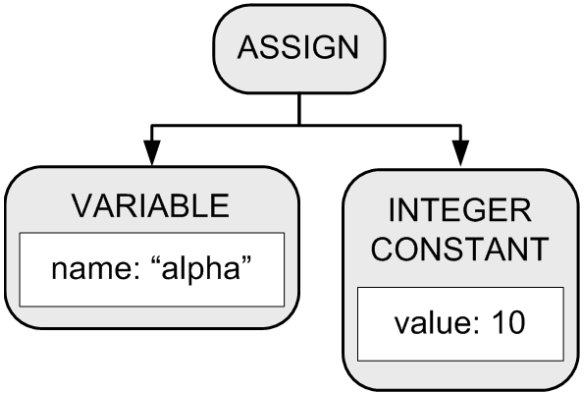
```
BEGIN
    alpha := 10;
    beta  := 20
END
```

1. **`CompoundStatementParser`**'s **`parse()`** method creates a COMPOUND node.



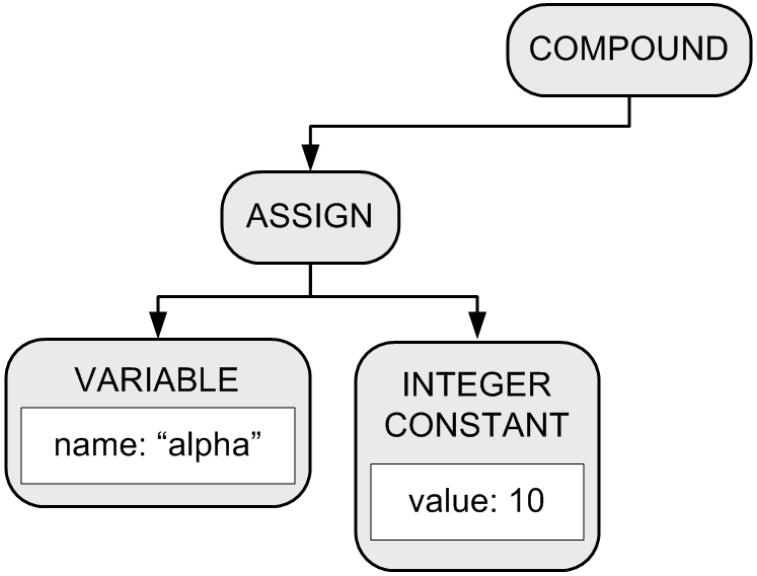
2. **`AssignmentStatementParser`**'s **`parse()`** method creates an ASSIGN node and a VARIABLE node, which the ASSIGN node adopts as its first child.

Building a Parse Tree



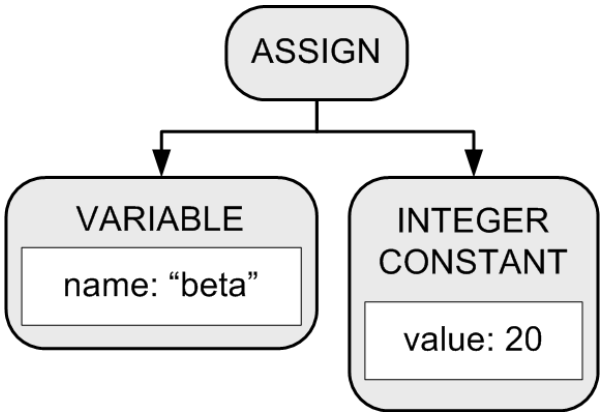
3. **`ExpressionParser`**’s **`parse()`** method creates an INTEGER CONSTANT node which the ASSIGN node adopts as its second child.

```
BEGIN
    alpha := 10;
    beta  := 20
END
```



4. The COMPOUND node adopts the ASSIGN node as its first child.

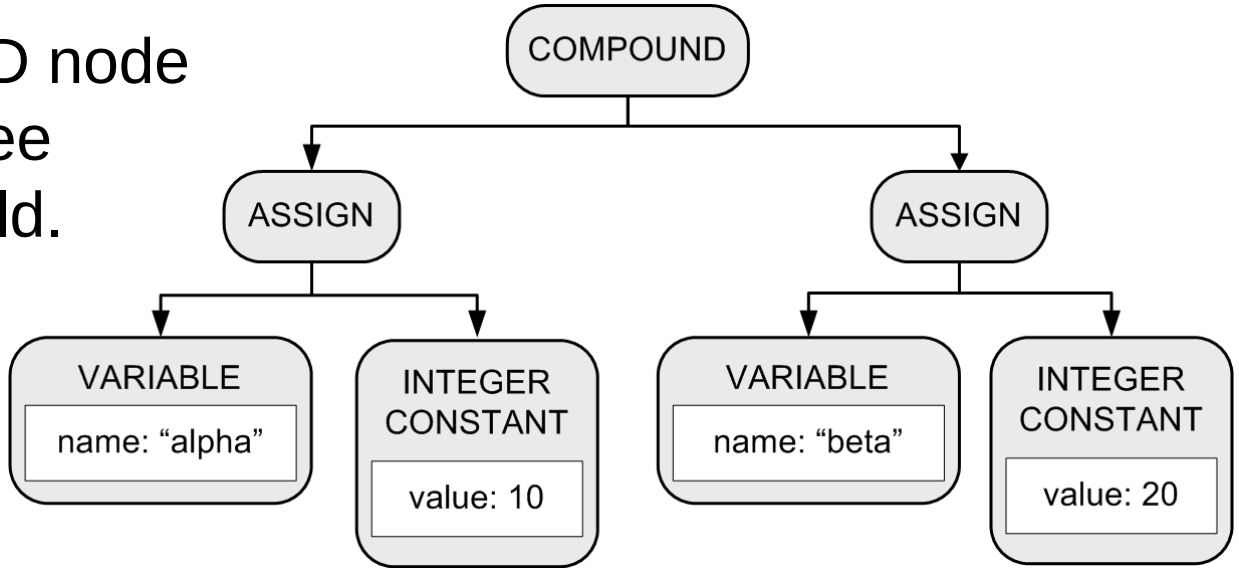
Building a Parse Tree



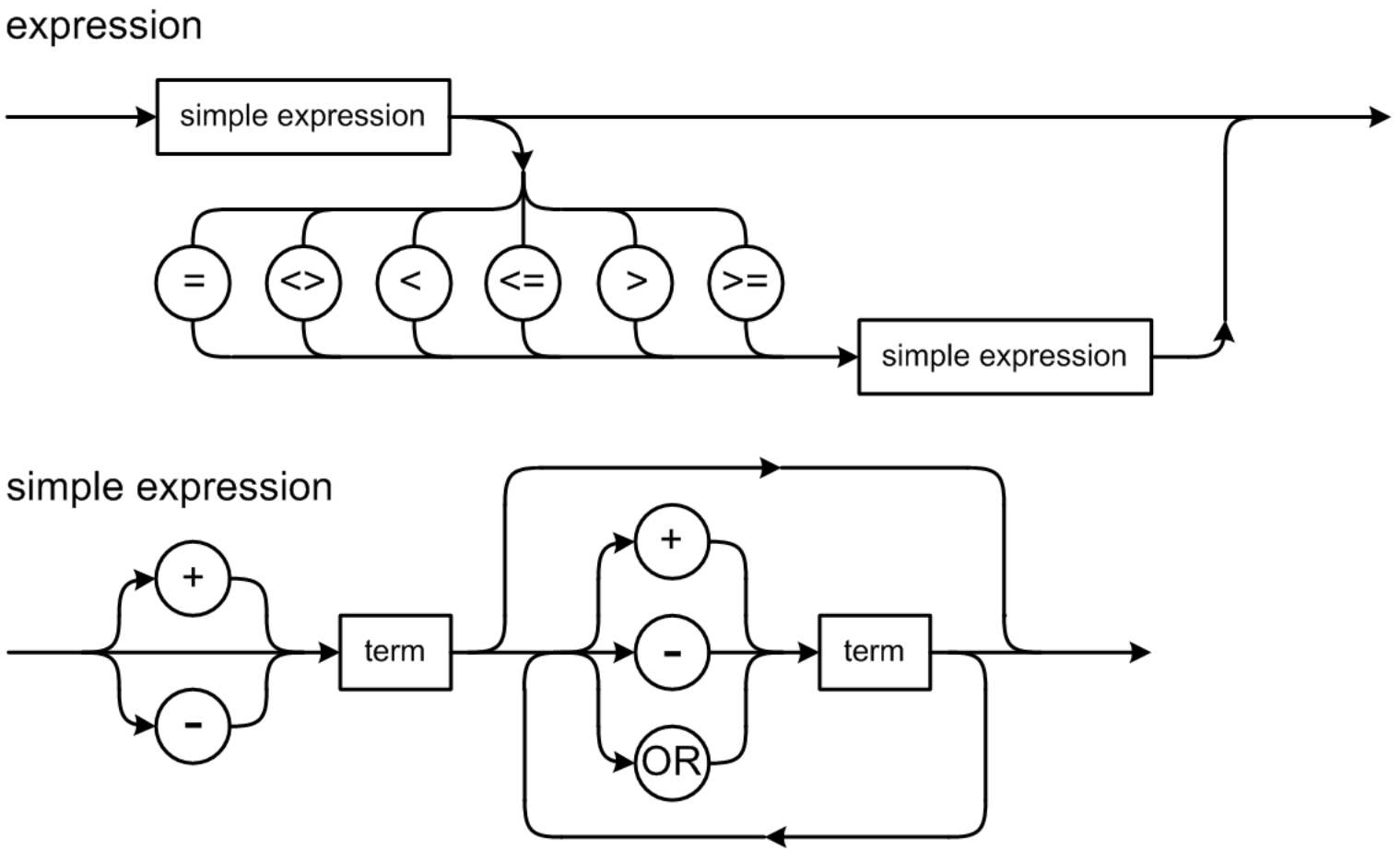
5. Another set of calls to the **`parse()`** methods of **`AssignmentStatementParser`** and **`ExpressionParser`** builds another assignment statement subtree.

6. The COMPOUND node adopts the subtree as its second child.

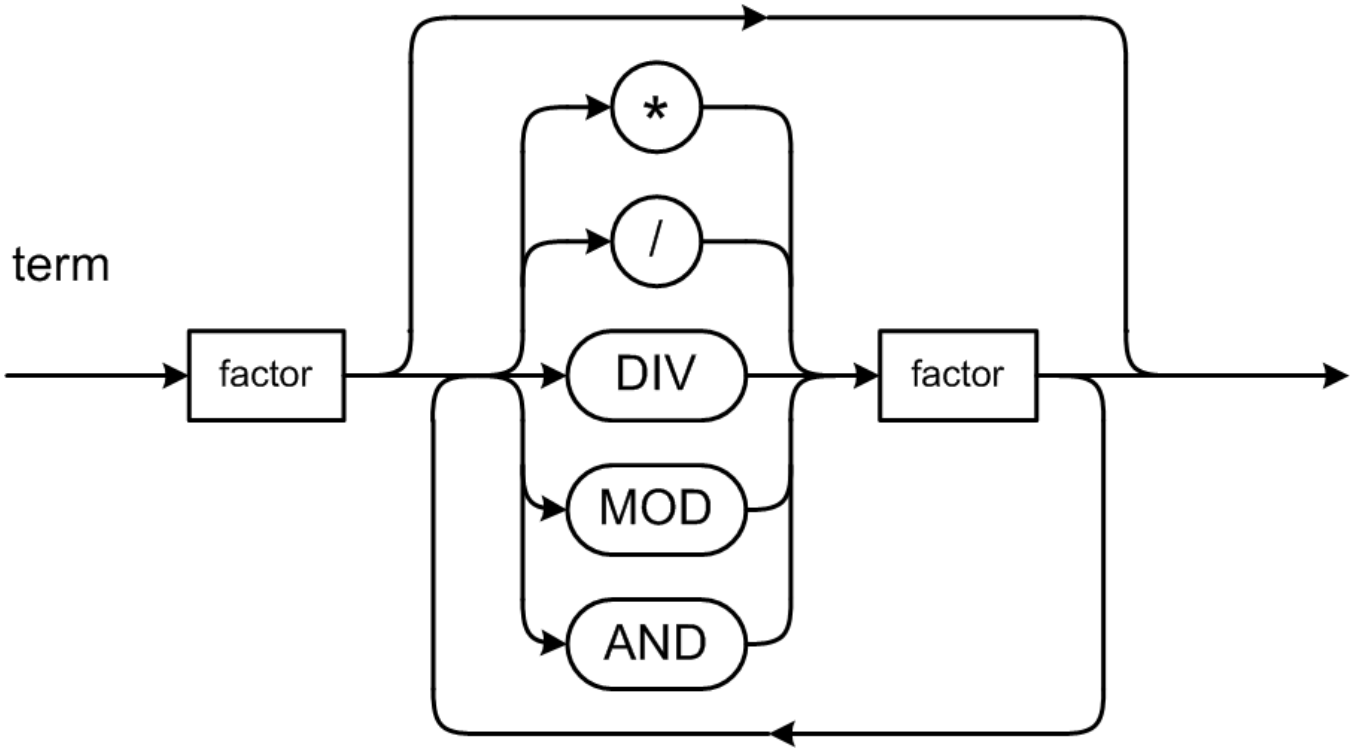
```
BEGIN
    alpha := 10;
    beta  := 20
END
```



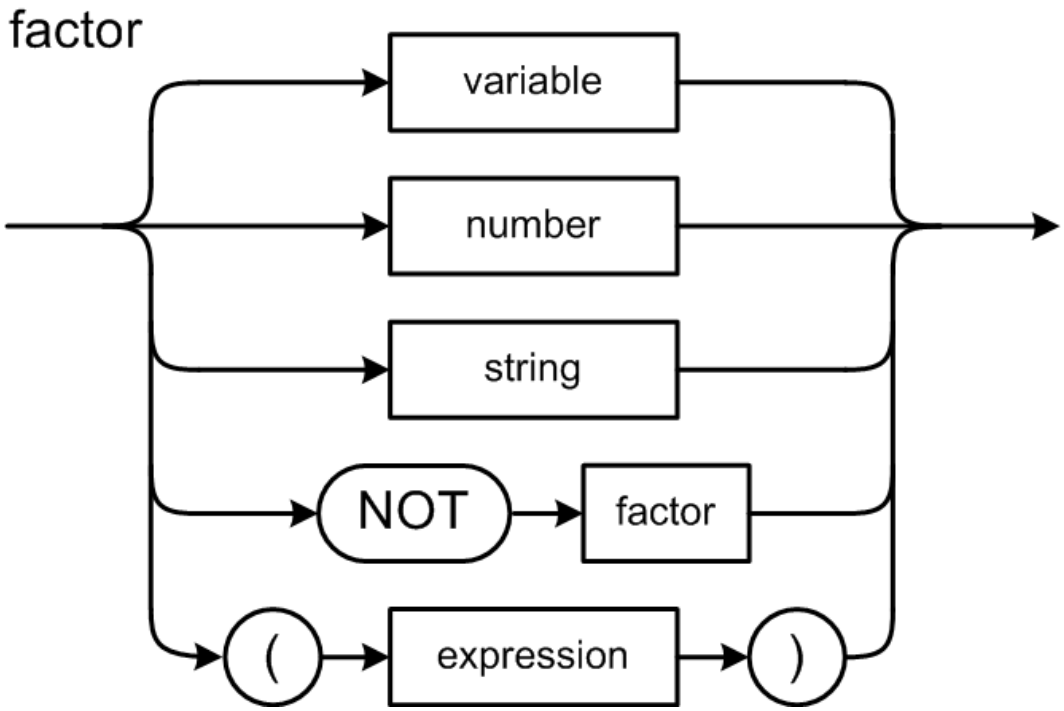
Pascal Expression Syntax Diagrams



Expression Syntax Diagrams, *cont'd*



Expression Syntax Diagrams, *cont'd*



Pascal's Operator Precedence Rules

Level	Operators
1 (factor: *highest*)	`NOT`
2 (term)	multiplicative: `* / DIV MOD AND`
3 (simple expression)	additive: `+ - OR`
4 (expression: *lowest*)	relational: `= <> < <= > >=`

- If there are no parentheses:
  - Higher level operators execute before the lower level ones.
  - Operators at the same level execute from left to right.
- Because the factor syntax diagram defines parenthesized expressions, parenthesized expressions always execute first, from the most deeply nested outwards.

Example Decomposition

- `alpha + 3/(beta – gamma) + 5`