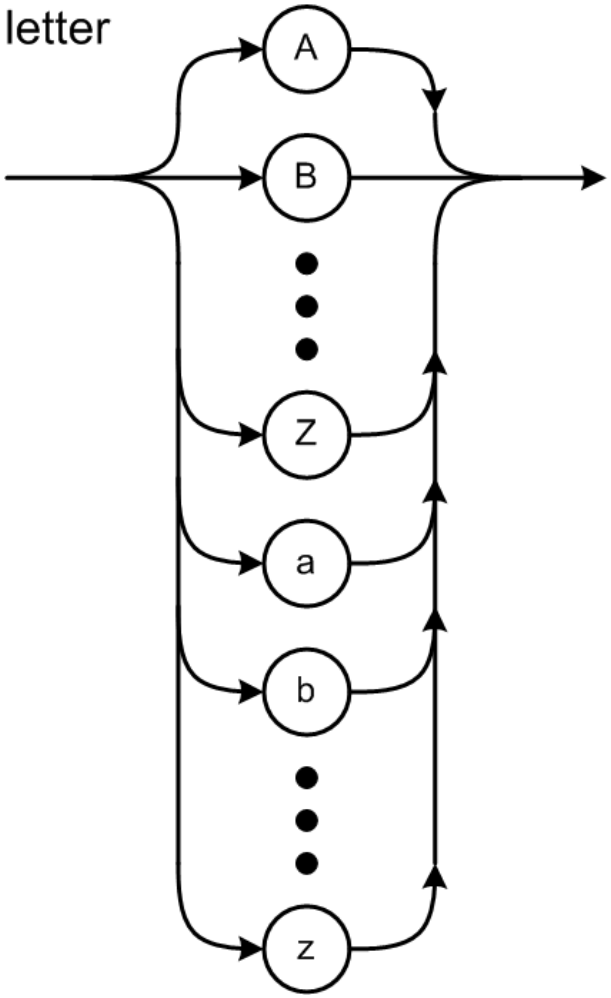
CS 153: Concepts of Compiler Design

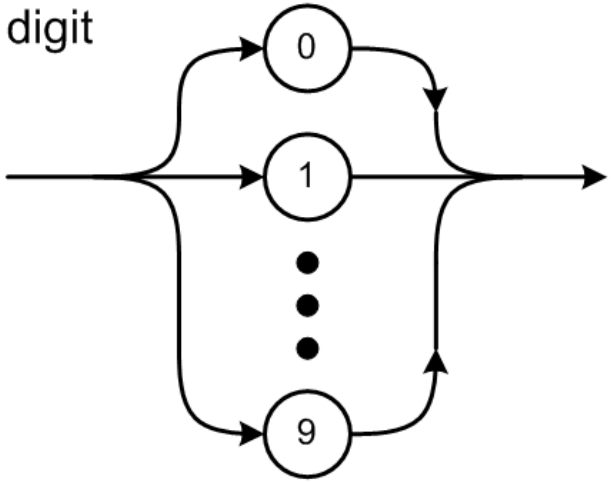
September 7 Class Meeting

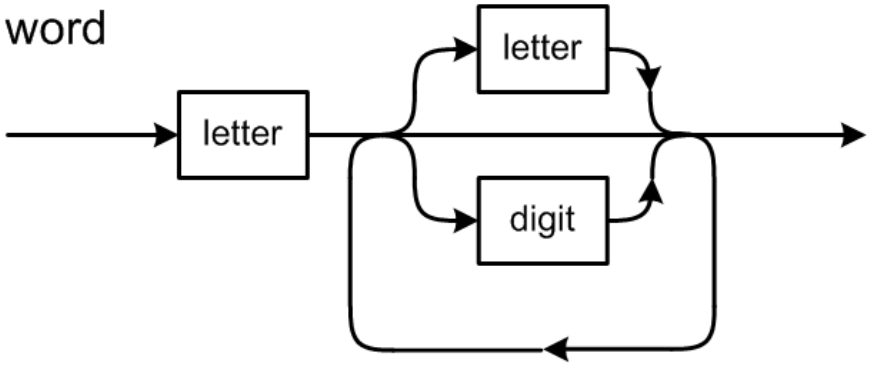
Department of Computer Science
San Jose State University

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

Syntax Diagrams



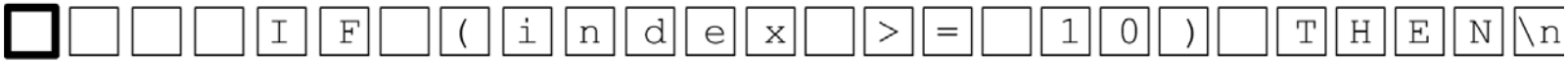




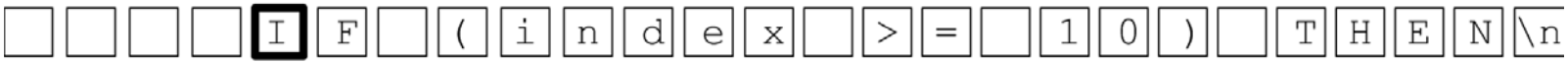
How to Scan for Tokens

- Suppose the source line contains

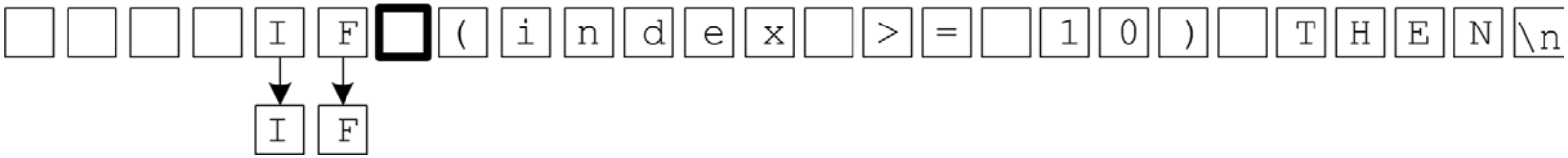
```
IF (index >= 10) THEN
```



- The scanner skips over the leading blanks. The current character is **I**, so the next token must be a word.

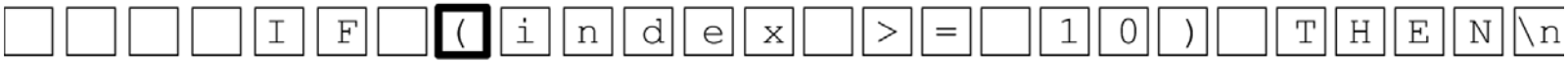


- The scanner extracts a word token by copying characters up to but not including the first character that is not valid for a word, which in this case is a blank. The blank becomes the current character.
  - The scanner determines that the word is a reserved word.

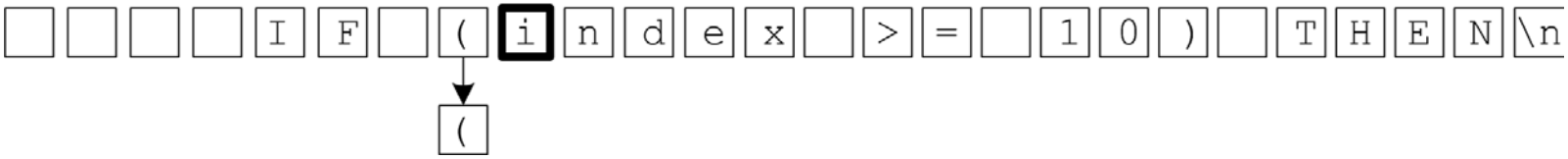


How to Scan for Tokens, *cont'd*

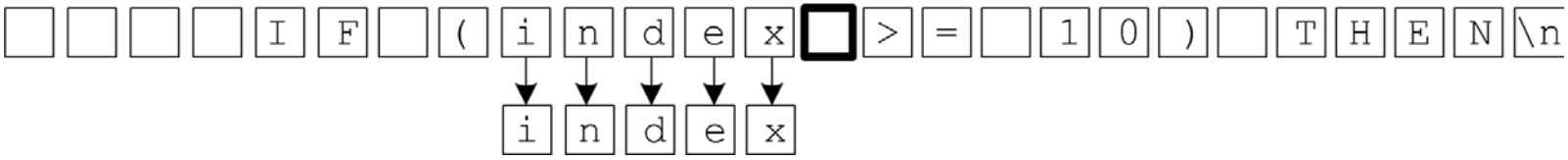
- The scanner skips over any blanks between tokens. The current character is **(**. The next token must be a special symbol.



- After extracting the special symbol token, the current character is **i**. The next token must be a word.

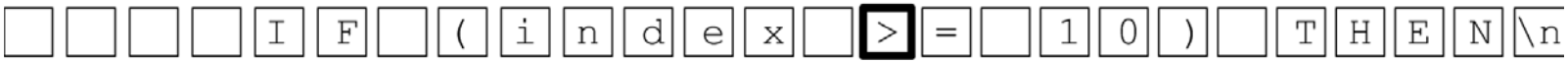


- After extracting the word token, the current character is a blank.

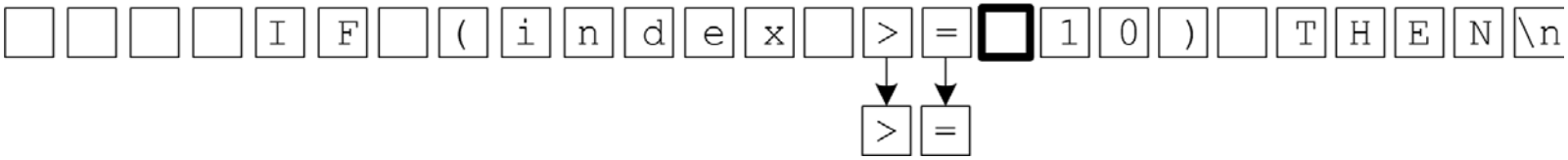


How to Scan for Tokens, *cont'd*

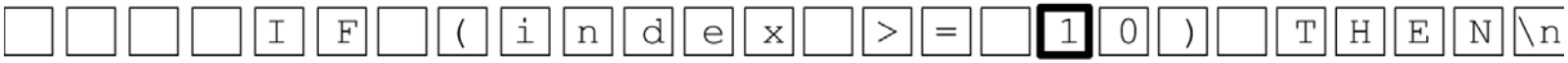
- Skip the blank. The current character is **>**.



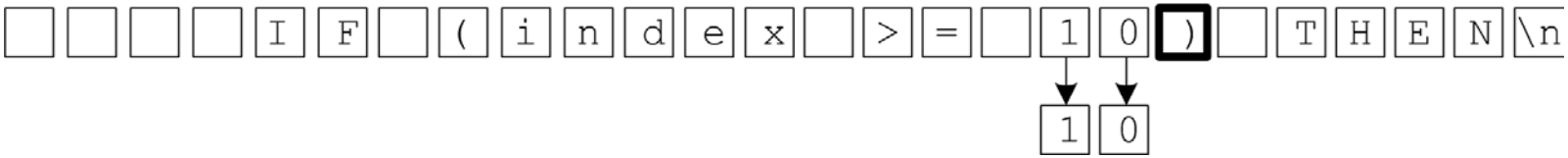
- Extract the special symbol token. The current character is a blank.



- Skip the blank. The current character is **1**, so the next token must be a number.

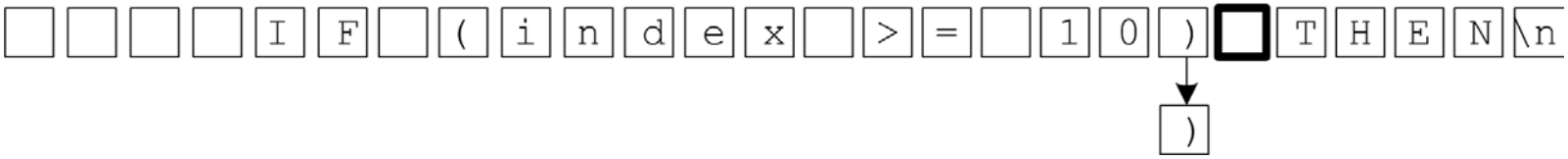


- After extracting the number token, the current character is **)**.

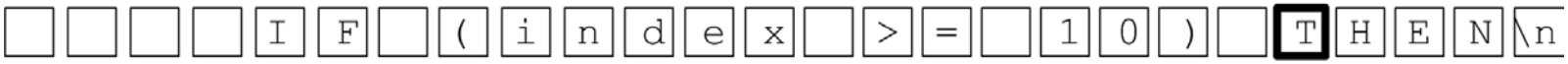


How to Scan for Tokens, *cont'd*

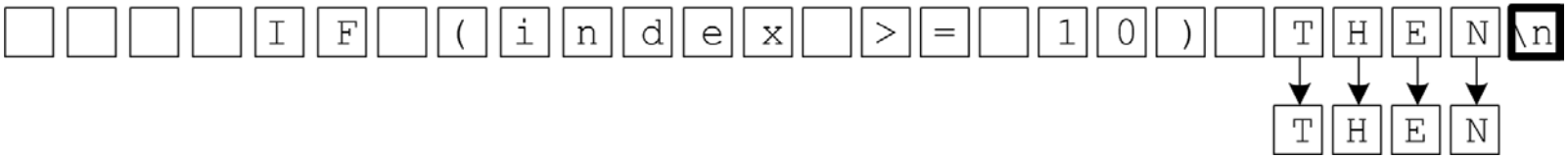
- Extract the special symbol token. The current character is a blank.



- Skip the blank. The current character is **T**, so the next token must be a word.



- Extract the word token.
  - Determine that it's a reserved word.



- The current character is **\n**, so the scanner is done with this line.

Basic Scanning Algorithm

- Skip any blanks until the current character is nonblank.
  - In Pascal, a comment and the end-of-line character each should be treated as a blank.
- The current (nonblank) character determines what the next token is and becomes that token's first character.
- Extract the rest of the next token by copying successive characters up to but not including the first character that does not belong to that token.
- Extracting a token consumes all the source characters that constitute the token.
  - After extracting a token, the current character is the first character after the last character of that token.

What to Store in Each Symbol Table Entry

- Each symbol table entry is designed to store information about an identifier.
- The attribute keys indicate what information we will store for each type of identifier.
  - Store common information in fixed fields (e.g., **`lineNumbers`**) and store identifier type-specific information as attributes.

```
public enum SymTabKeyImpl implements SymTabKey
{
    // Constant.
    CONSTANT_VALUE,

    // Procedure or function.
    ROUTINE_CODE, ROUTINE_SYMTAB, ROUTINE_ICODE,
    ROUTINE_PARMS, ROUTINE_ROUTINES,

    // Variable or record field value.
    DATA_VALUE
}
```

Modifications to Class **PascalParserTD**

```
while (!((token = nextToken()) instanceof EofToken)) {
    TokenType tokenType = token.getType();

    // Cross reference only the identifiers.
    if (tokenType == IDENTIFIER) {
        String name = token.getText().toLowerCase();

        // If it's not already in the symbol table,
        // create and enter a new entry for the identifier.
        SymTabEntry entry = symTabStack.lookup(name);
        if (entry == null) {
            entry = symTabStack.enterLocal(name);
        }

        // Append the current line number to the entry.
        entry.appendLineNumber(token.getLineNumber());
    }

    else if (tokenType == ERROR) {
        errorHandler.flag(token, (PascalErrorCode) token.getValue(), this);
    }
}
```

Cross-Reference Listing

- A cross-reference listing verifies the symbol table code:
  `java -classpath classes Pascal compile -x newton.pas`
- Modifications to the main **`Pascal`** class:

```
parser.parse();
iCode = parser.getICode();
symTabStack = parser.getSymTabStack();

if (xref) {
    CrossReferencer crossReferencer = new CrossReferencer();
    crossReferencer.print(symTabStack);
}

backend.process(iCode, symTabStack);
source.close();
```

- A new utility class **`CrossReferencer`** generates the cross-reference listing.

Demo

Cross-Reference Listing Output

```
001 PROGRAM newton (input, output);
002
003 CONST
004     EPSILON = 1e-6;
005
006 VAR
007     number       : integer;
008     root, sqRoot : real;
009
010 BEGIN
011     REPEAT
012         writeln;
013         write('Enter new number (0 to quit): ');
014         read(number);

            ...

035     UNTIL number = 0
036 END.

                  36 source statements.
                   0 syntax errors.
                0.06 seconds total parsing time.
```

Cross-Reference Listing Output, *cont'd*

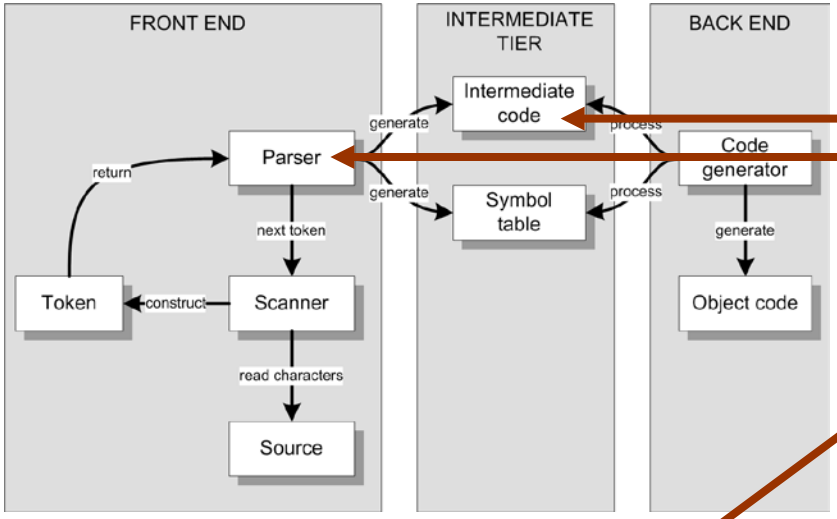
```
===== CROSS-REFERENCE TABLE =====

Identifier       Line numbers
----------       ------------
abs              031 033
epsilon          004 033
input            001
integer          007
newton           001
number           007 014 016 017 019 023 024 029 033 035
output           001
read             014
real             008
root             008 027 029 029 029 030 031 033
sqr              033
sqroot           008 023 024 031 031
sqrt             023
write            013
writeln          012 017 020 024 025 030

                   0 instructions generated.
                0.00 seconds total code generation time.
```

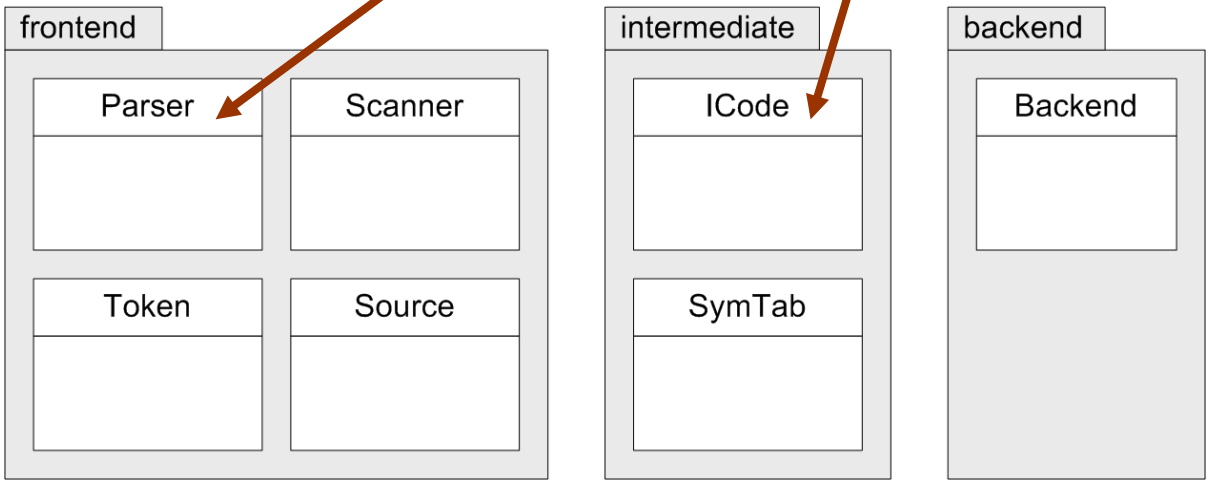
Quick Review of the Framework

FROM:



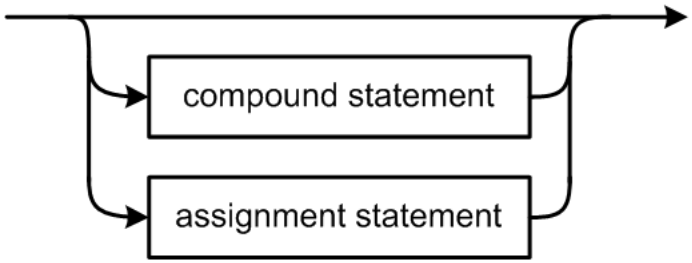
Next topic:
Parsing assignment statements and expressions, and generating parse trees.

TO:

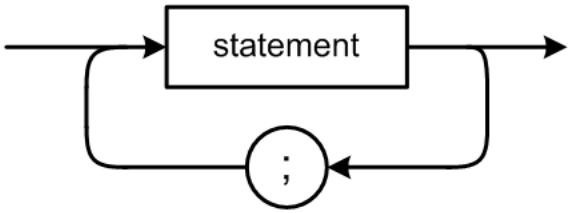


Pascal Statement Syntax Diagrams

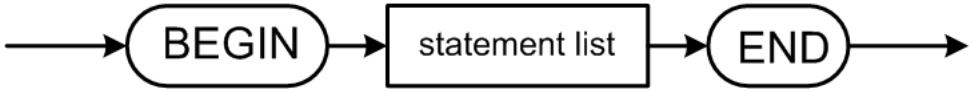
statement



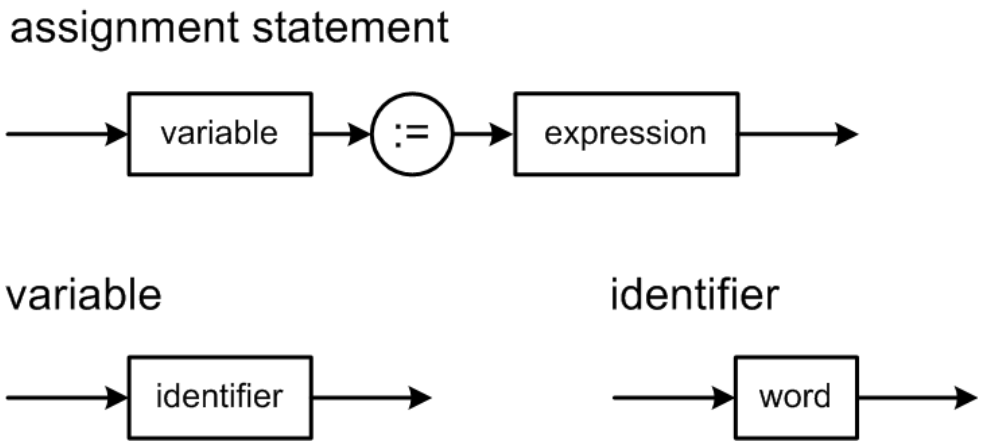
statement list



compound statement

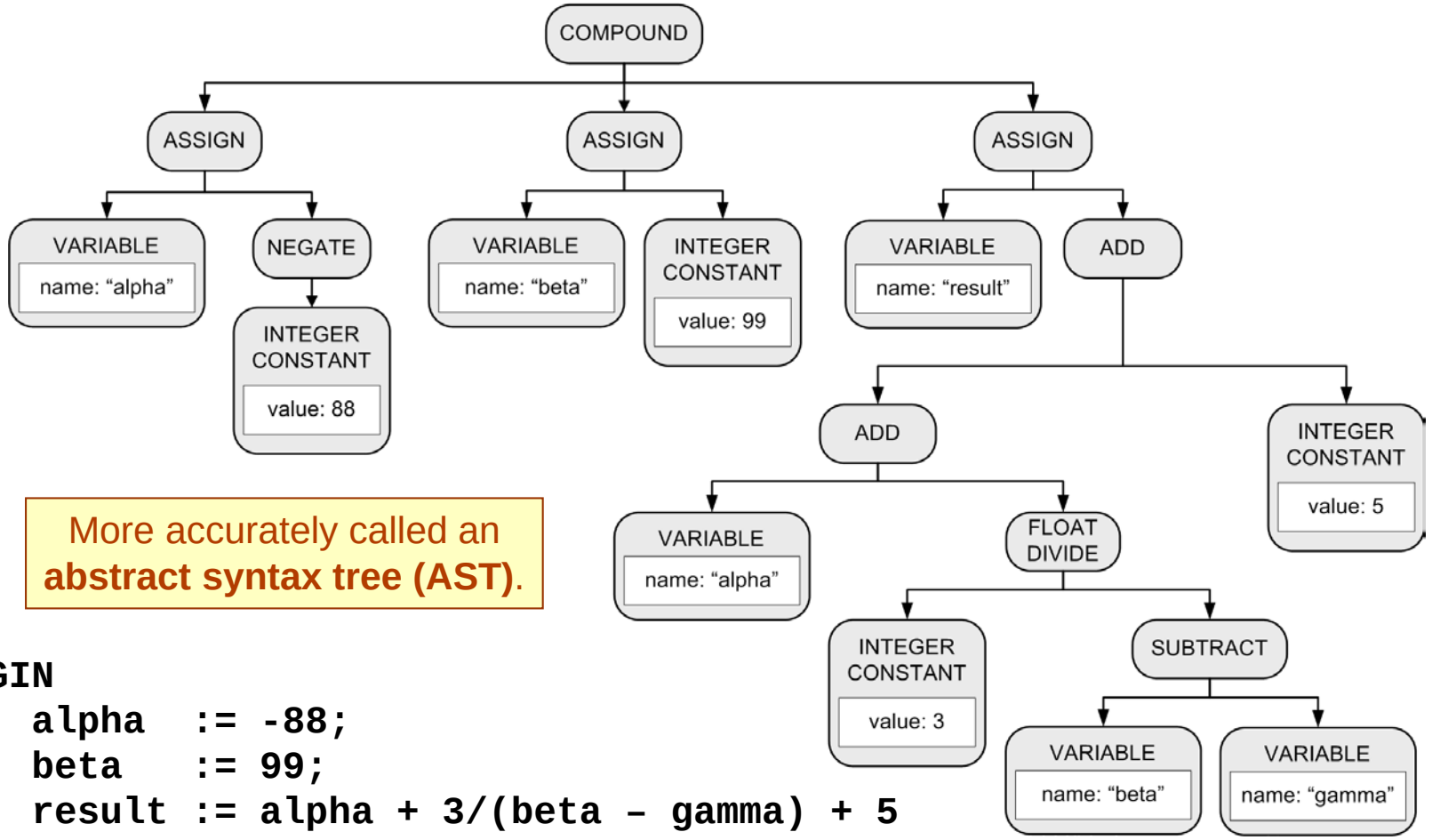


Pascal Statement Syntax Diagrams, *cont'd*



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
```

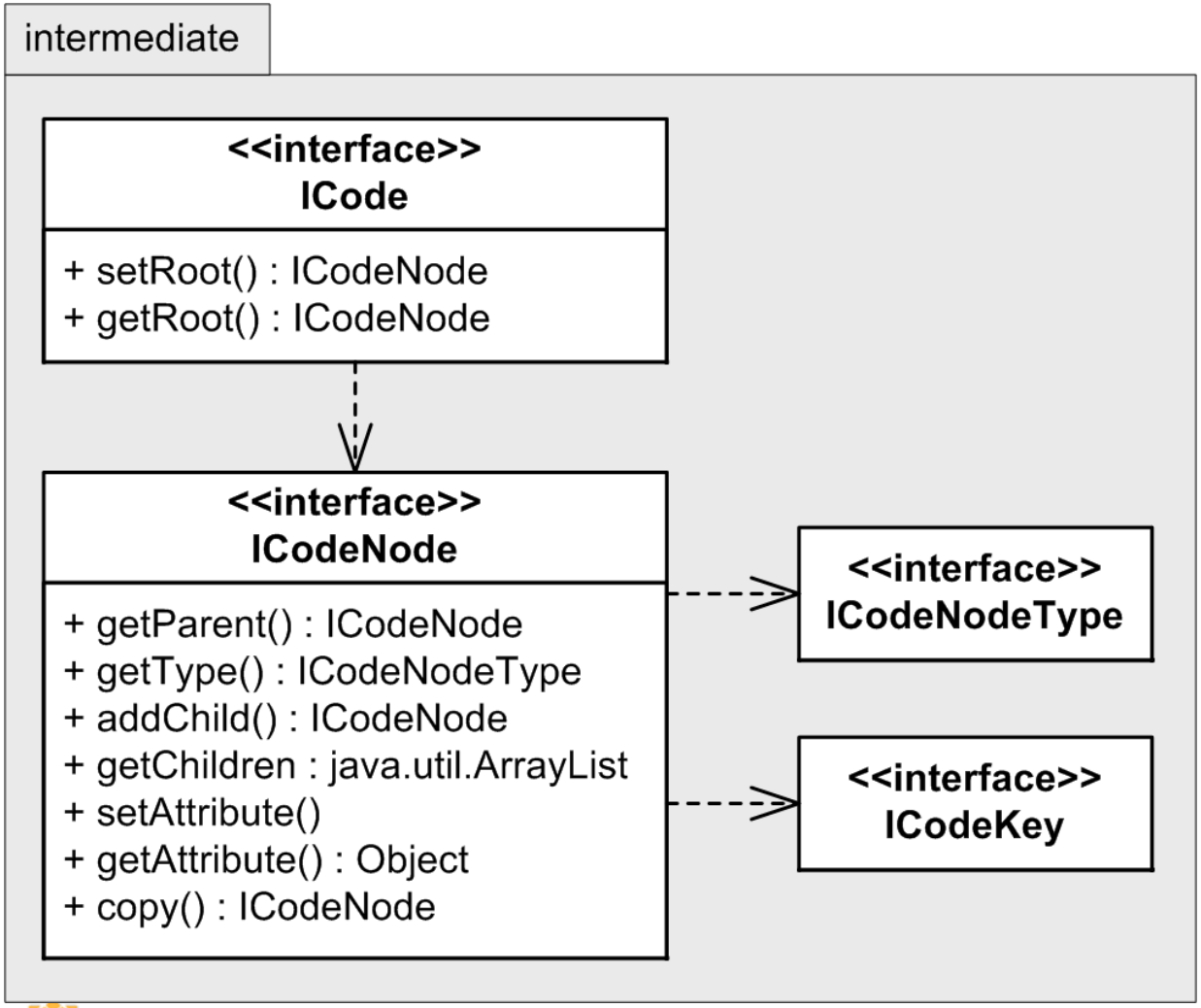
Parse Tree: Conceptual Design

- 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

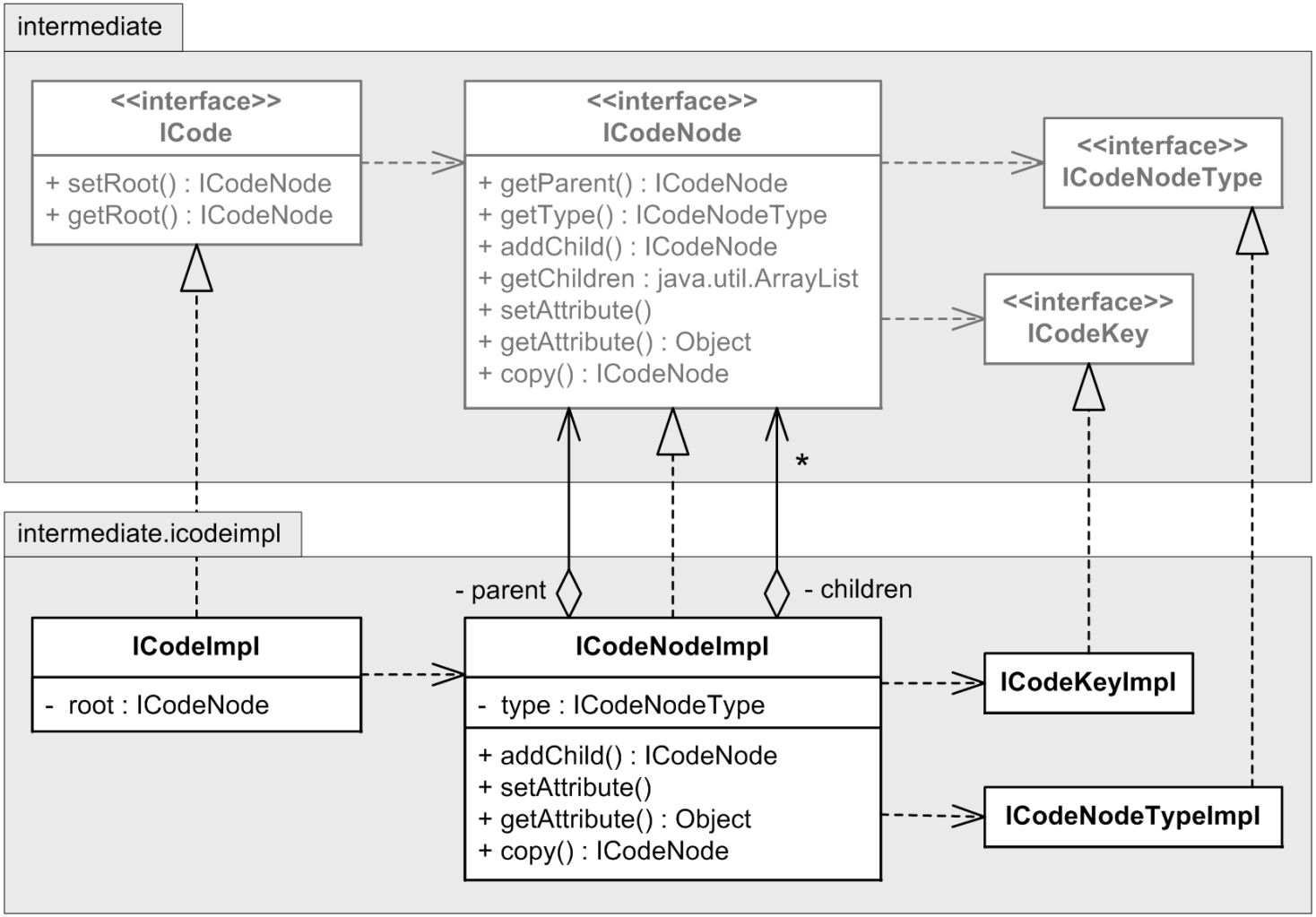
- Create a new node.
- Create a copy of a node.
- Set and get the root node of a parse tree.
- Set and get an attribute value in a node.
- Add a child node to a node.
- Get the list of a node's child nodes.
- Get a node's parent node.

Intermediate Code Interfaces



Goal: Keep it source language-independent.

Intermediate Code Implementations



An Intermediate Code Factory Class

```
public class ICodeFactory
{
    public static ICode createICode()
    {
        return new ICodeImpl();
    }

    public static ICodeNode createICodeNode(ICodeNodeType type)
    {
        return new ICodeNodeImpl(type);
    }
}
```

Coding to the Interfaces (Again)

```
// Create the ASSIGN node.
ICodeNode assignNode =
    ICodeFactory.createICodeNode(ASSIGN);

// Create the VARIABLE node (left-hand side).
ICodeNode variableNode =
    ICodeFactory.createICodeNode(VARIABLE);

// Adopt the VARIABLE node as the first child.
assignNode.addChild(variableNode);
```

Intermediate Code (ICode) Node Types

```
public enum ICodeNodeTypeImpl implements ICodeNodeType
{
    // 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,
}
```

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

```
public class ICodeNodeImpl
    extends HashMap<ICodeKey, Object>
    implements ICodeNode
{
    private ICodeNodeType type;              // node type
    private ICodeNode parent;                // parent node
    private ArrayList<ICodeNode> children;   // children array list

    public ICodeNodeImpl(ICodeNodeType type)
    {
        this.type = type;
        this.parent = null;
        this.children = new ArrayList<ICodeNode>();
    }
    ...
}
```

- Each node is a **`HashMap<ICodeKey, Object>`**.
- Each node has an **`ArrayList<ICodeNode>`** of child nodes.

A Parent Node Adopts a Child Node

```
public ICodeNode addChild(ICodeNode node)
{
    if (node != null) {
        children.add(node);
        ((ICodeNodeImpl) 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!

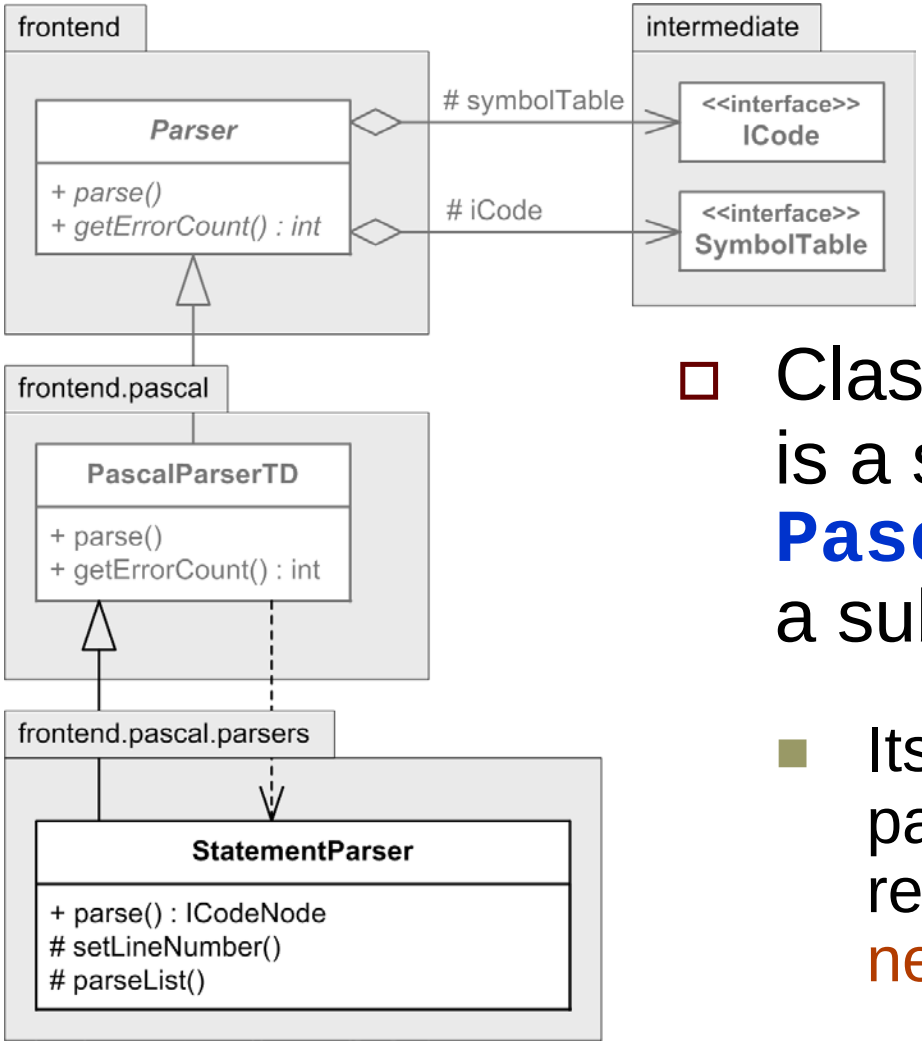
What Attributes to Store in a Node?

```
public enum ICodeKeyImpl implements ICodeKey
{
    LINE, ID, VALUE;
}
```

- Not much! Not every node will have these attributes.
  - **LINE**: statement line number
  - **ID**: symbol table entry of an identifier
  - **VALUE**: data value

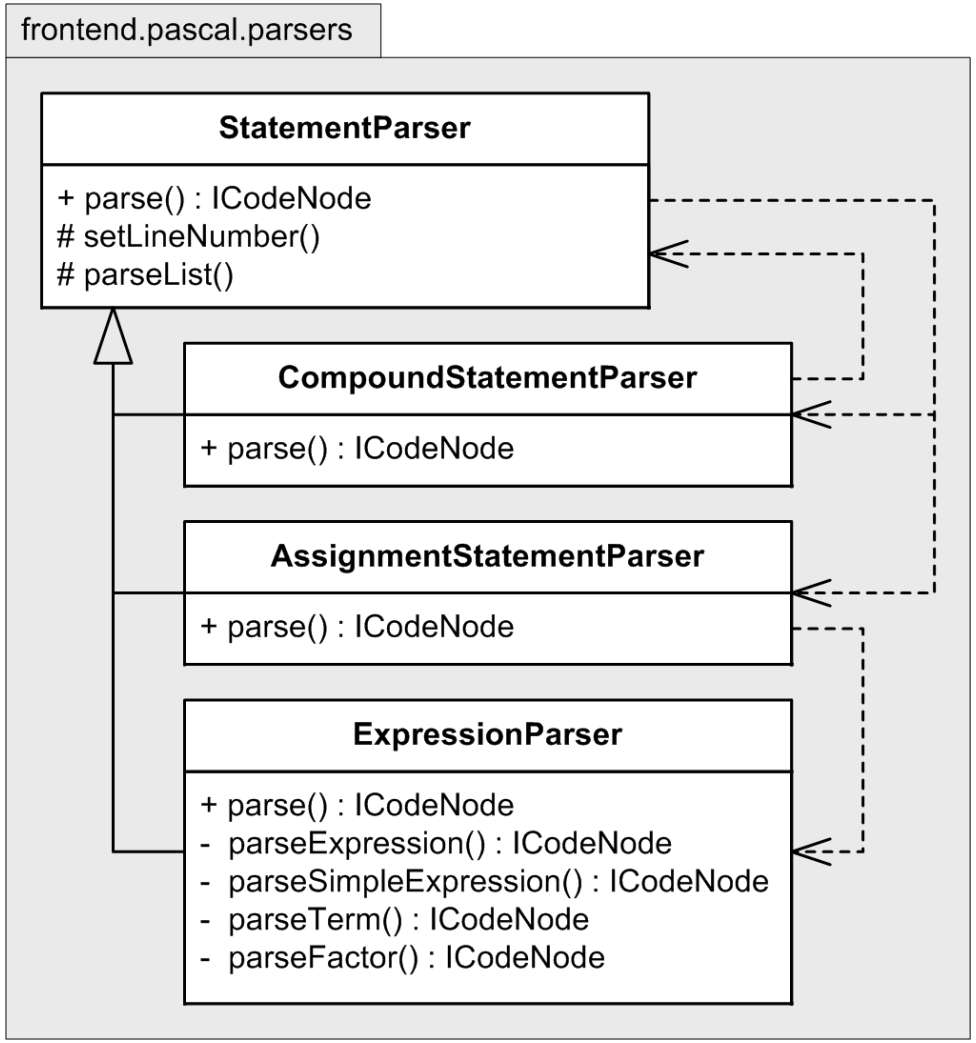
- Most of the information about what got parsed is encoded in the node type and in the tree structure.

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

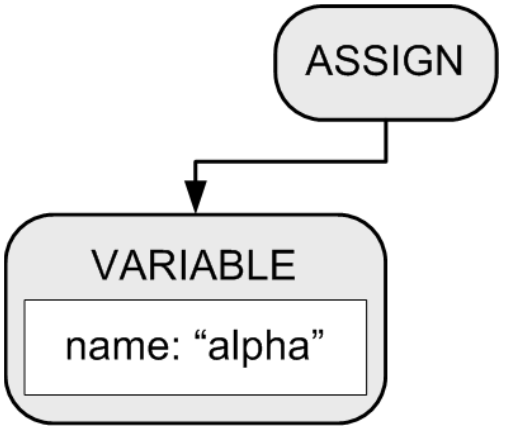
- Each **`parse()`** method builds a subtree and returns the root node of the new subtree.
- The caller of the **`parse()`** method adopts the subtree's root node as a child of the subtree that the caller is building.
  - The caller then returns the root node of its subtree to its caller.
  - This process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

Building a Parse Tree

- Example:

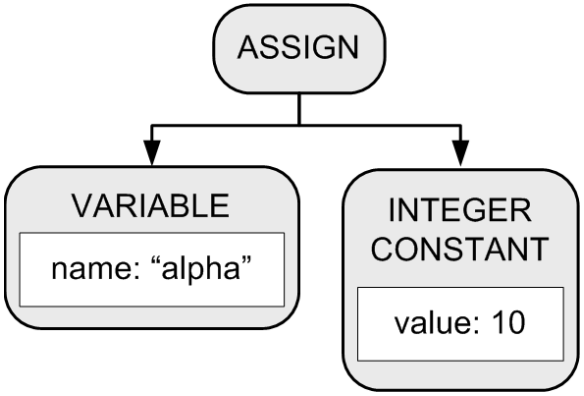
```
BEGIN
    alpha := 10;
    beta  := 20
END
```

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



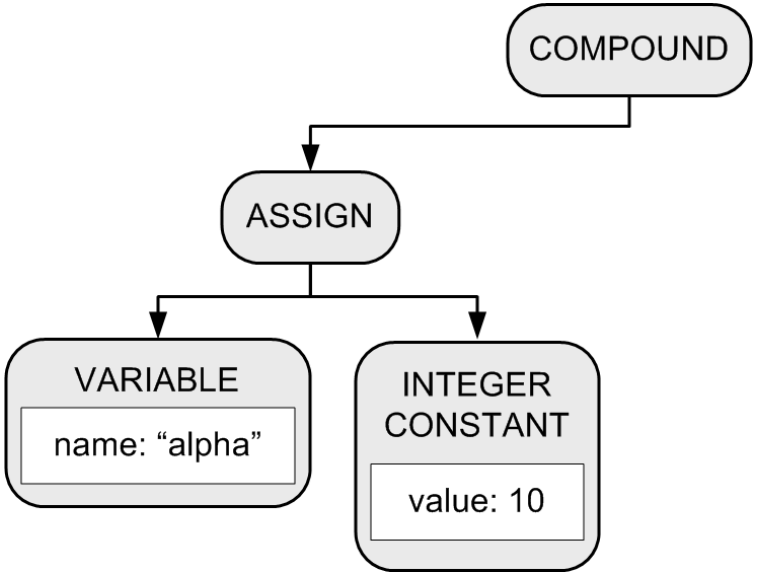
2. **`AssignmentStatementParser`**'s **`parse()`** method creates an ASSIGN node and a VARIABLE node, which the ASSIGN node adopts as its first child.

Building a Parse Tree



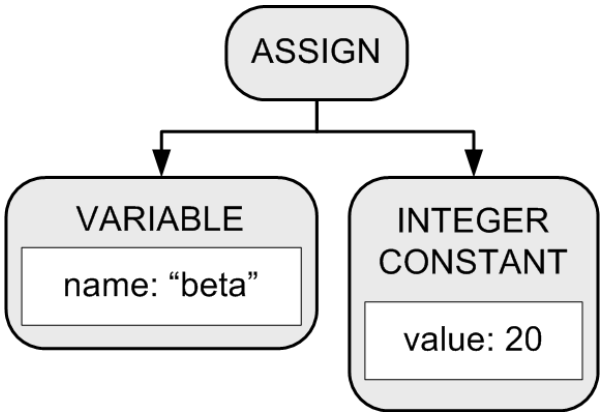
3. **`ExpressionParser`**’s **`parse()`** method creates an INTEGER CONSTANT node which the ASSIGN node adopts as its second child.

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



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

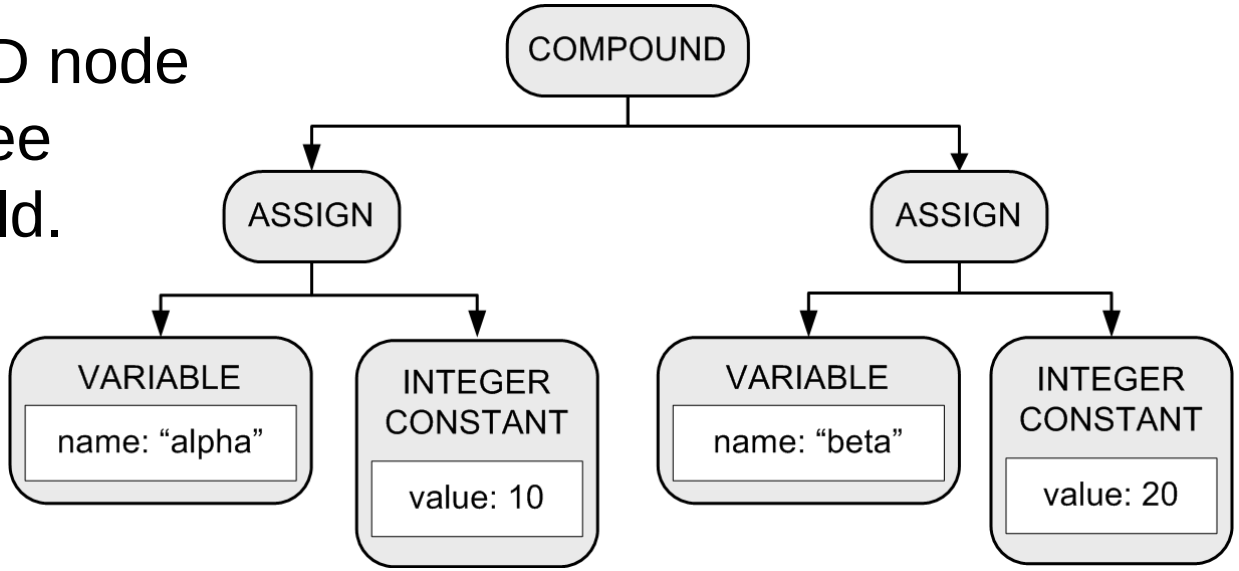
Building a Parse Tree



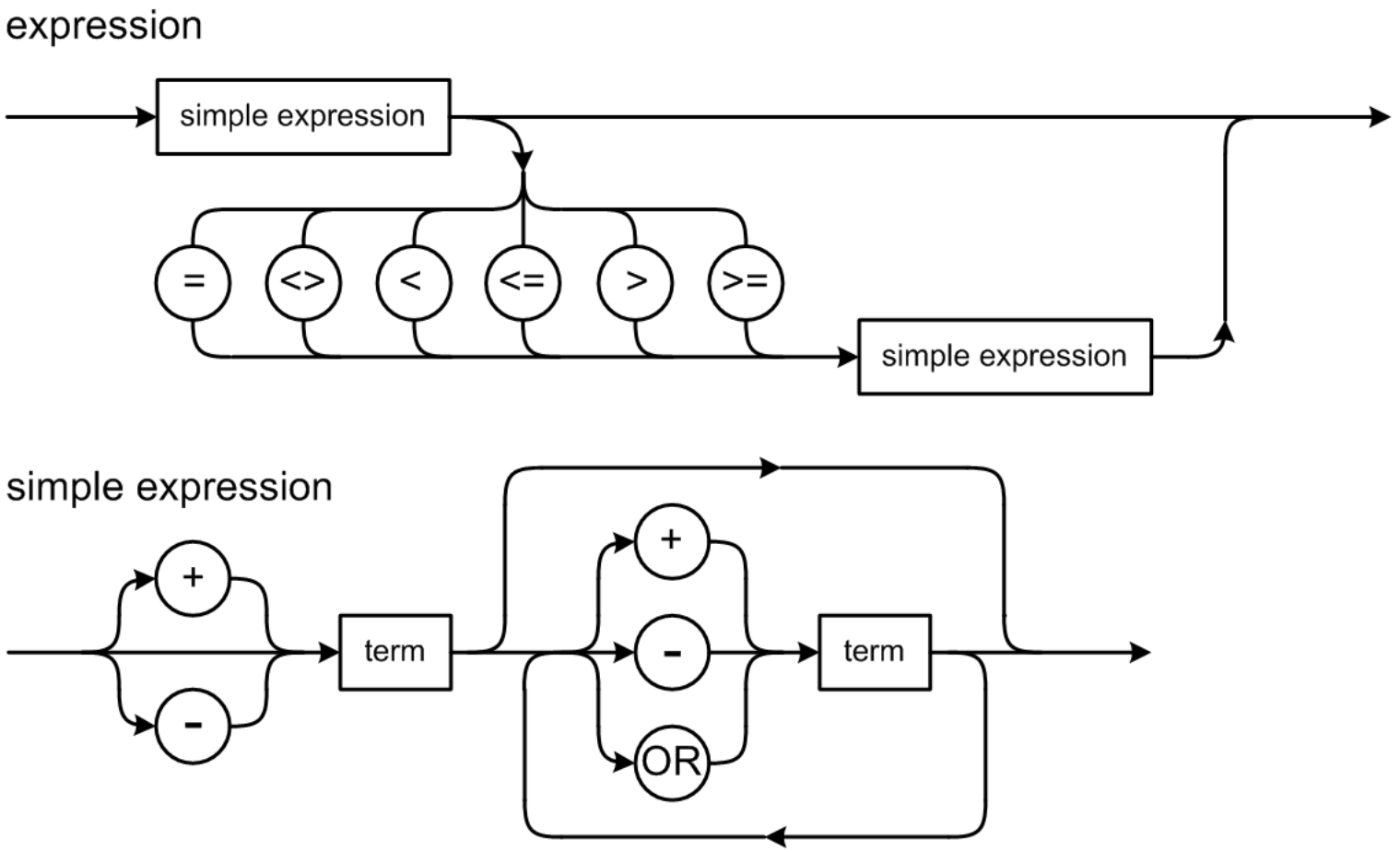
5. Another set of calls to the **`parse()`** methods of **`AssignmentStatementParser`** and **`ExpressionParser`** builds another assignment statement subtree.

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

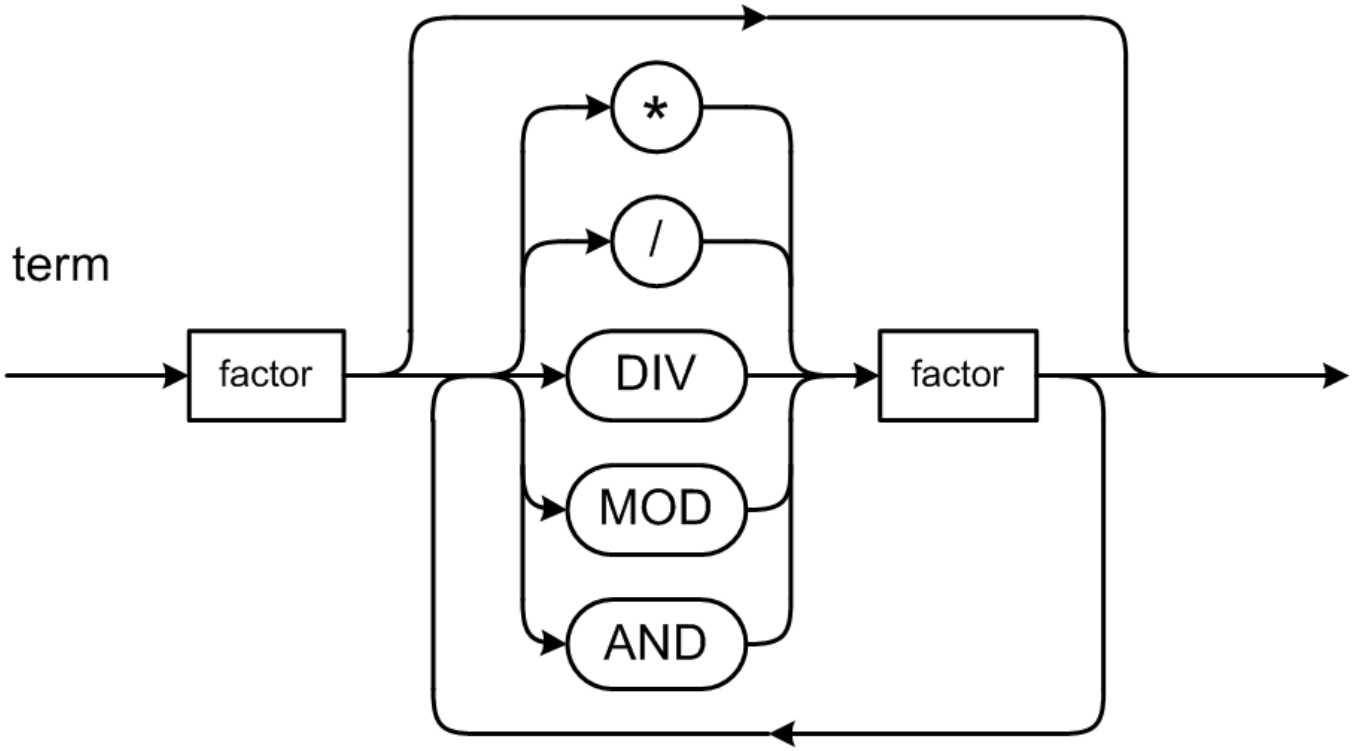
```
BEGIN
    alpha := 10;
    beta  := 20
END
```



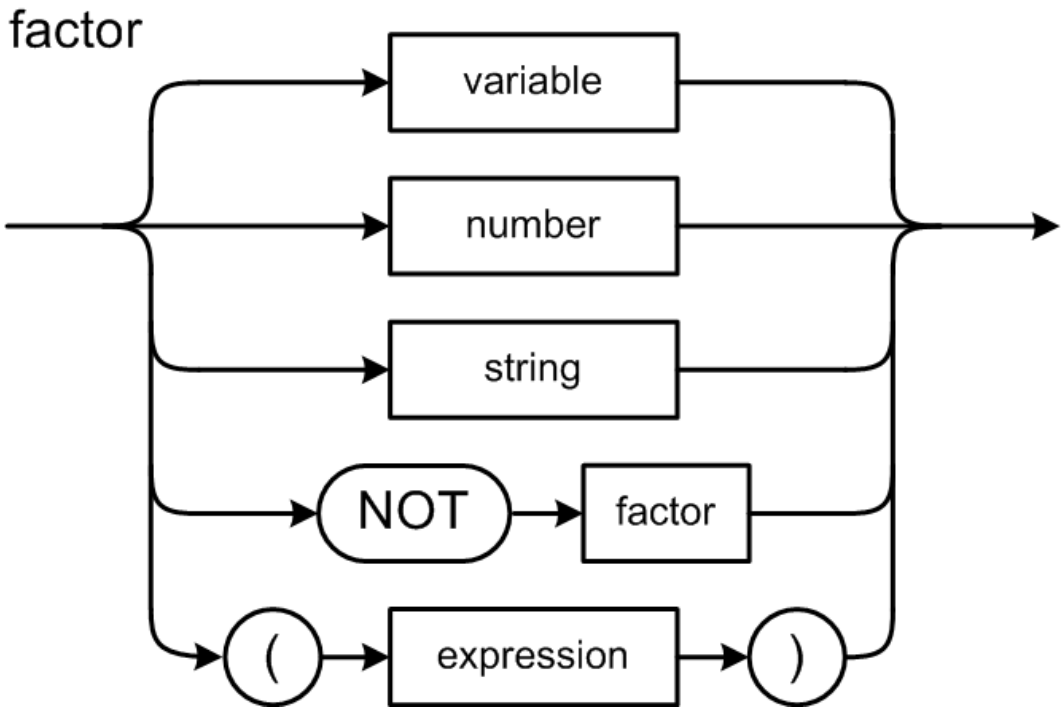
Pascal Expression Syntax Diagrams



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`