CS 153: Concepts of Compiler Design

September 28 Class Meeting

Department of Computer Science
San Jose State University

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

Now that we can parse declarations …

- We can parse variables that have subscripts and fields.

Chapter 10

  - Example:

```
var9.rec.flda[b][0,'m'].flda[d] := 'p’
```

- We can perform type checking.
  - A semantic action.

Type Checking

- Ensure that the types of the operands are type-compatible with their operator.
  - Example: You can only perform an integer division with the `DIV` operator and integer operands.
  - Example: The relational operators `AND` and `OR` can only be used with boolean operands.
- Ensure that a value being assigned to a variable is assignment-compatible with the variable.
  - Example: You cannot assign a string value to an integer variable.

Type Specifications and the Parse Tree

- Every Pascal expression has a data type.
- Add a type specification to every parse tree node.
  - "Decorate" the parse tree with type information.
  - In interface **`ICodeNode`**:

```
public void setTypeSpec(TypeSpec typeSpec);
public TypeSpec getTypeSpec();
```

  - In class **`ICodeNodeImpl`**:

```
private TypeSpec typeSpec;  // data type specification

public void setTypeSpec(TypeSpec typeSpec) { ... }
public TypeSpec getTypeSpec() { ... }
```

Class `TypeChecker`

- Static boolean methods for type checking:
  - `isInteger()`
  - `areBothInteger()`
  - `isReal()`
  - `isIntegerOrReal()`
  - `isAtLeastOneReal()`
  - `isBoolean()`
  - `areBothBoolean()`
  - `isChar()`
  - `areAssignmentCompatible()`
  - `areComparisonCompatible()`
  - `equalLengthStrings()`

In package `intermediate.typeimpl`.

Class **TypeChecker**, *cont'd*

```
public static boolean isInteger(TypeSpec type)
{
    return (type != null) && (type.baseType() == Predefined.integerType);
}

public static boolean areBothInteger(TypeSpec type1, TypeSpec type2)
{
    return isInteger(type1) && isInteger(type2);
}


...

public static boolean isAtLeastOneReal(TypeSpec type1, TypeSpec type2)
{
    return (isReal(type1) && isReal(type2)) ||
           (isReal(type1) && isInteger(type2)) ||
           (isInteger(type1) && isReal(type2));
}
```

Assignment and Comparison Compatible

- In classic Pascal, a value is assignment-compatible with a target variable if:
  - both have the same type
  - the target is real and the value is integer
  - they are equal-length strings

- Two values are comparison-compatible (they can be compared with relational operators) if:
  - both have the same type
  - one is integer and the other is real
  - they are equal-length strings

Assignment Compatible

```
public static boolean areAssignmentCompatible(TypeSpec targetType,
                                              TypeSpec valueType)
{
    if ((targetType == null) || (valueType == null)) return false;

    targetType = targetType.baseType();
    valueType  = valueType.baseType();

    boolean compatible = false;

    if (targetType == valueType) {
        compatible = true;
    }
    else if (isReal(targetType) && isInteger(valueType)) {
        compatible = true;
    }
    else {
        compatible = equalLengthStrings(targetType, valueType);
    }

    return compatible;
}
```

Same type

real := integer

Equal length strings

Type Checking Expressions

- The parser must perform type checking of every expression as part of its semantic actions.
- Add type checking to class **`ExpressionParser`** and to each statement parser.
- Flag type errors similarly to syntax errors.

Method **`ExpressionParser.parseTerm()`**

- Now besides doing <u>syntax checking</u>, our expression parser must also do <u>type checking</u> and determine the result type of each operation.

integer * integer ➔ integer result

one integer and one real, or both real ➔ real result

```
case STAR: {

    if (TypeChecker.areBothInteger(resultType, factorType)) {
        resultType = Predefined.integerType;
    }

    else if (TypeChecker.isAtLeastOneReal(resultType, factorType)) {
        resultType = Predefined.realType;
    }

    else {
        errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
    }

    break;
}
```

Type Checking Control Statements

- Method `IfStatementParser.parse()`

```
public ICodeNode parse(Token token)
    throws Exception
{
    token = nextToken();  // consume the IF
    ICodeNode ifNode = ICodeFactory.createICodeNode(ICodeNodeTypeImpl.IF);

    ExpressionParser expressionParser = new ExpressionParser(this);
    ICodeNode exprNode = expressionParser.parse(token);
    ifNode.addChild(exprNode);

    TypeSpec exprType = exprNode != null ? exprNode.getTypeSpec()
                                         : Predefined.undefinedType;
    if (!TypeChecker.isBoolean(exprType)) {
        errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
    }

    token = synchronize(THEN_SET);
    ...
}
```

ExpressionParser.parseFactor()

- Now an identifier can be more than just a variable name.

```
private ICodeNode parseFactor(Token token)
    throws Exception
{
    ...
    switch ((PascalTokenType) tokenType) {

        case IDENTIFIER: {
            return parseIdentifier(token);
        }
    ...
}
```

ExpressionParser.parseIdentifier()

- Constant identifier

```
CONST
    pi = 3.14159;
```

  - Previously defined in a CONST definition.
  - Create an INTEGER_CONSTANT, REAL_CONSTANT, or a STRING_CONSTANT node.
  - Set its **VALUE** attribute.

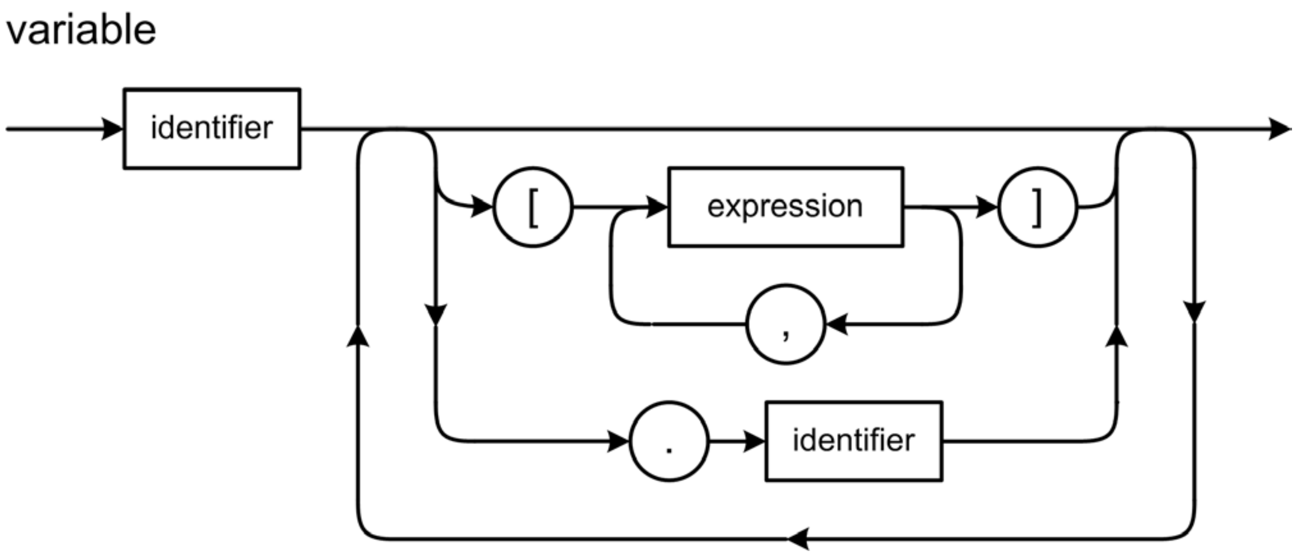
```
TYPE
    direction =
        (north, south,
         east, west);
```

- Enumeration identifier
  - Previously defined in a type specification.
  - Create an INTEGER_CONSTANT node.
  - Set its **VALUE** attribute.

ExpressionParser.parseIdentifier()

- Variable identifier
  - Call method `variableParser.parse()`.

Syntax Diagram for Variables



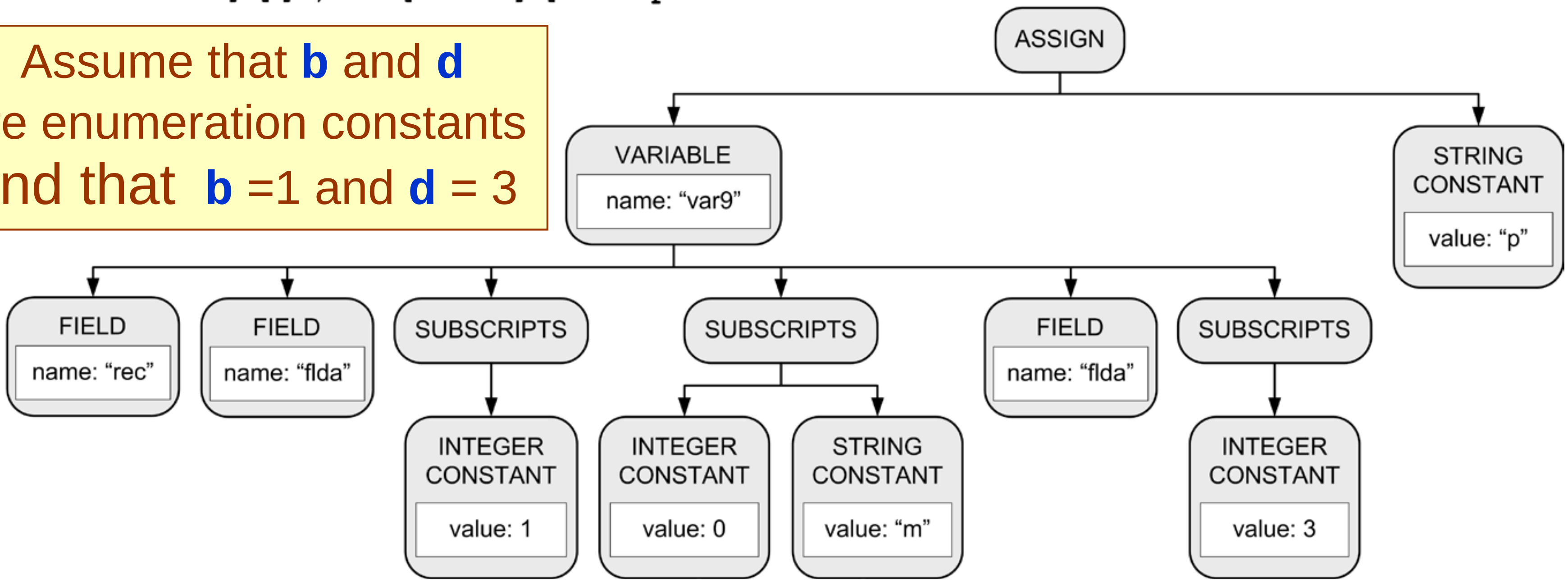
The outer loop back allows any number of subscripts and fields.

- A variable can have <u>any combination</u> of subscripts and fields.
  - Appear in an expression or as the target of an assignment statement.
  - Example: `var9.rec.flda[b][0,'m'].flda[d] := 'p'`
  - The parser must do <u>type checking for each subscript and field</u>.

Parse Tree for Variables

```
var9.rec.flda[b][0, 'm'].flda[d] := 'p'
```

Assume that **b** and **d** are enumeration constants and that **b** =1 and **d** = 3



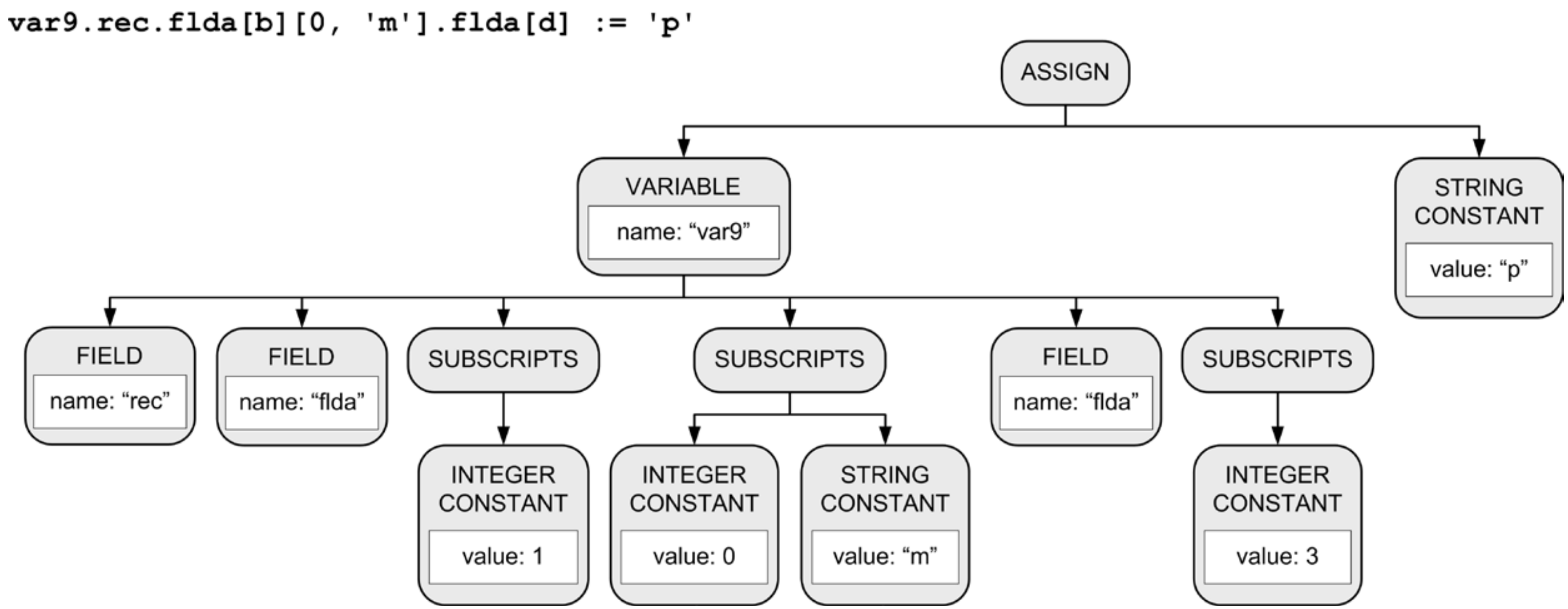
- VARIABLE nodes can now have child nodes:
  - SUBSCRIPTS
  - FIELD

Class `VariableParser`

- Parse variables that appear in statements.
  - Subclass of **`StatementParser`**.
  - Do not confuse with class **`VariableDeclarationsParser`**.
    - Subclass of **`DeclarationsParser`**.

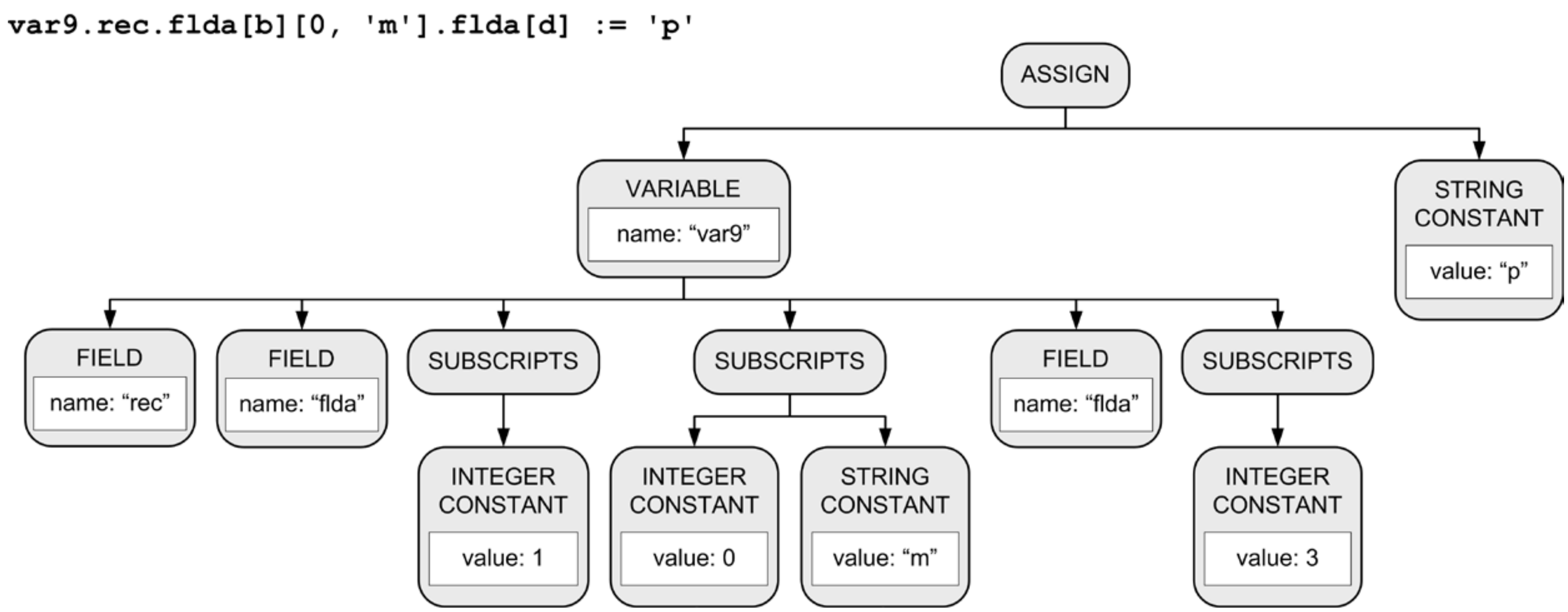
- Parsing methods
  - **`parse()`**
  - **`parseField()`**
  - **`parseSubscripts()`**

VariableParser.parse()



- Parse the variable identifier (example: `var9`)
- Create the VARIABLE node.

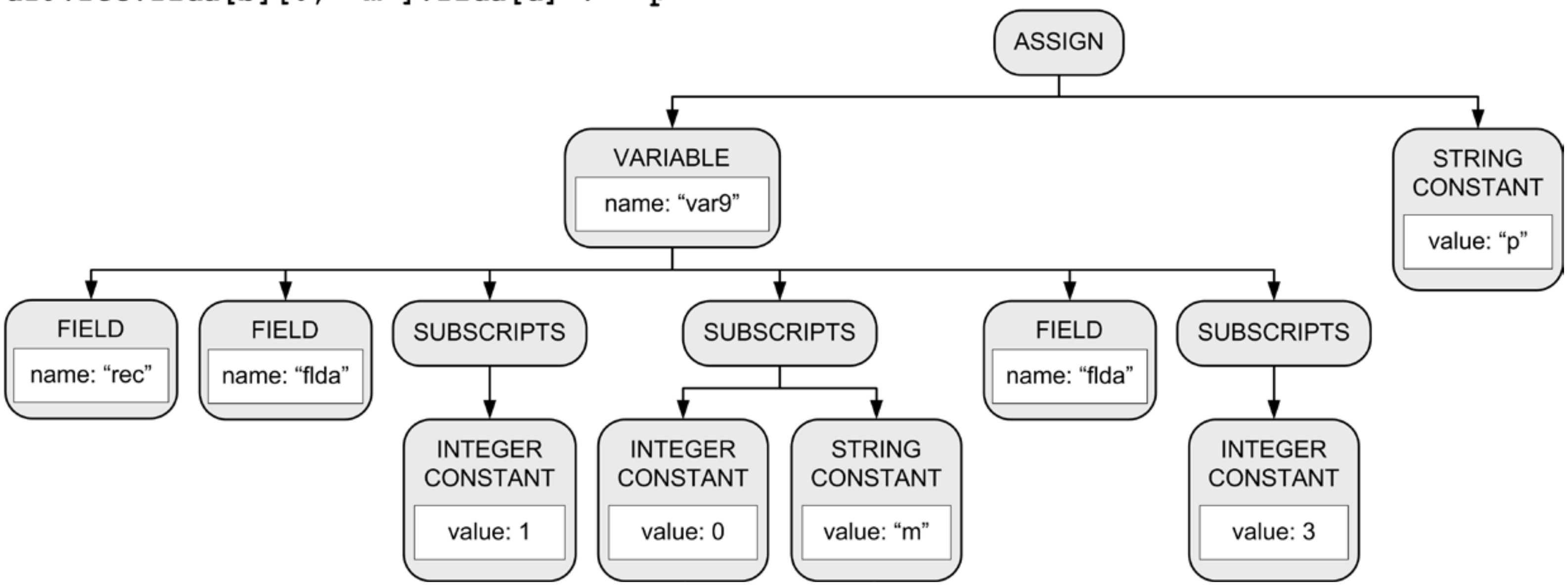
`VariableParser.parse()` *cont'd*



- Loop to parse any subscripts and fields.
  - Call methods **`parseField()`** or **`parseSubscripts()`**.
  - Variable **`variableType`** keeps track of the current type specification.
    - The <u>current type changes</u> as each field and subscript is parsed.

VariableParser.parseField()

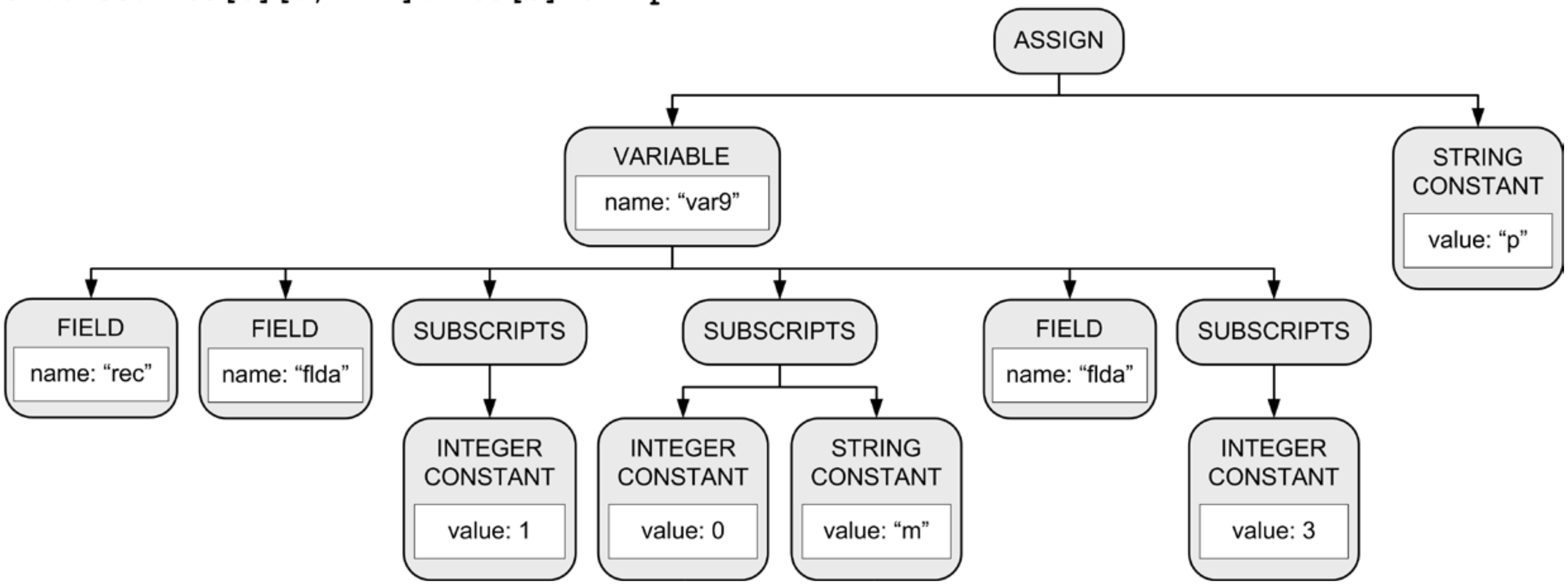
```
var9.rec.flda[b][0, 'm'].flda[d] := 'p'
```



- Get the record type’s symbol table.
  - Attribute **RECORD_SYMTAB** of the record variable’s type specification.

`VariableParser.parseField()` *cont'd*

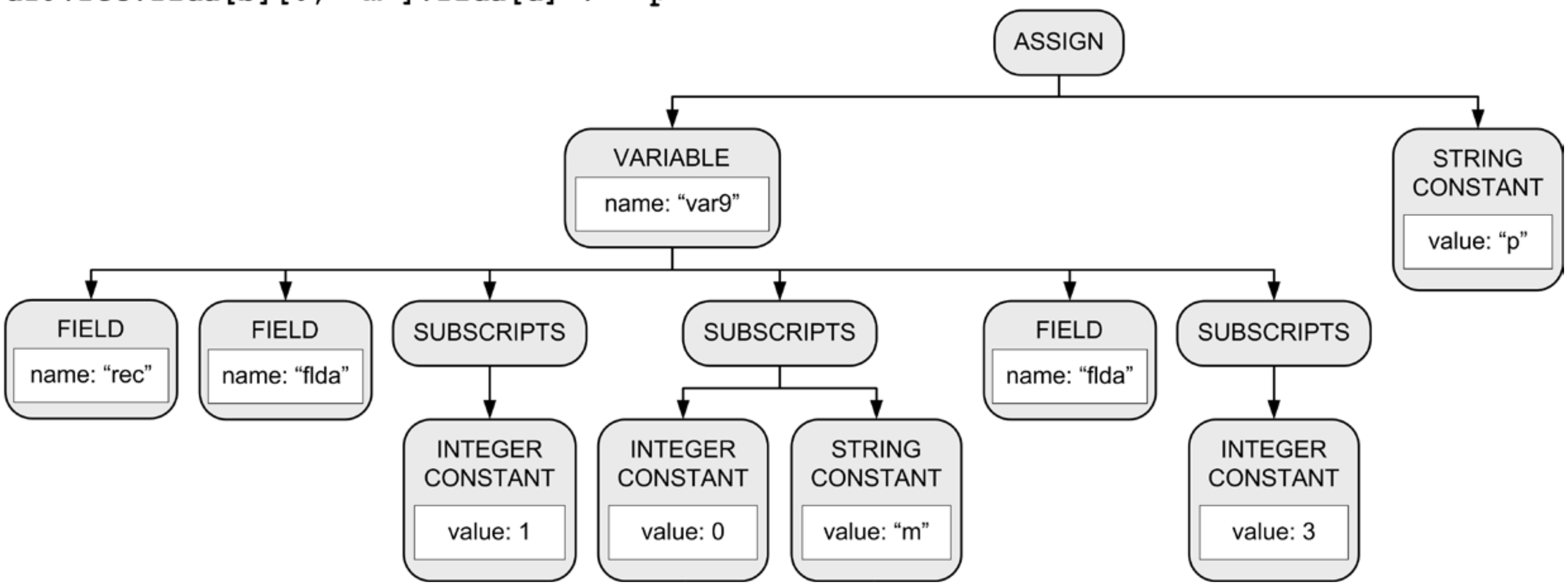
```
var9.rec.flda[b][0, 'm'].flda[d] := 'p'
```



- Verify that the field identifier is in the record type's symbol table.
- Create a FIELD node that is adopted by the VARIABLE node.

VariableParser.parseSubscripts()

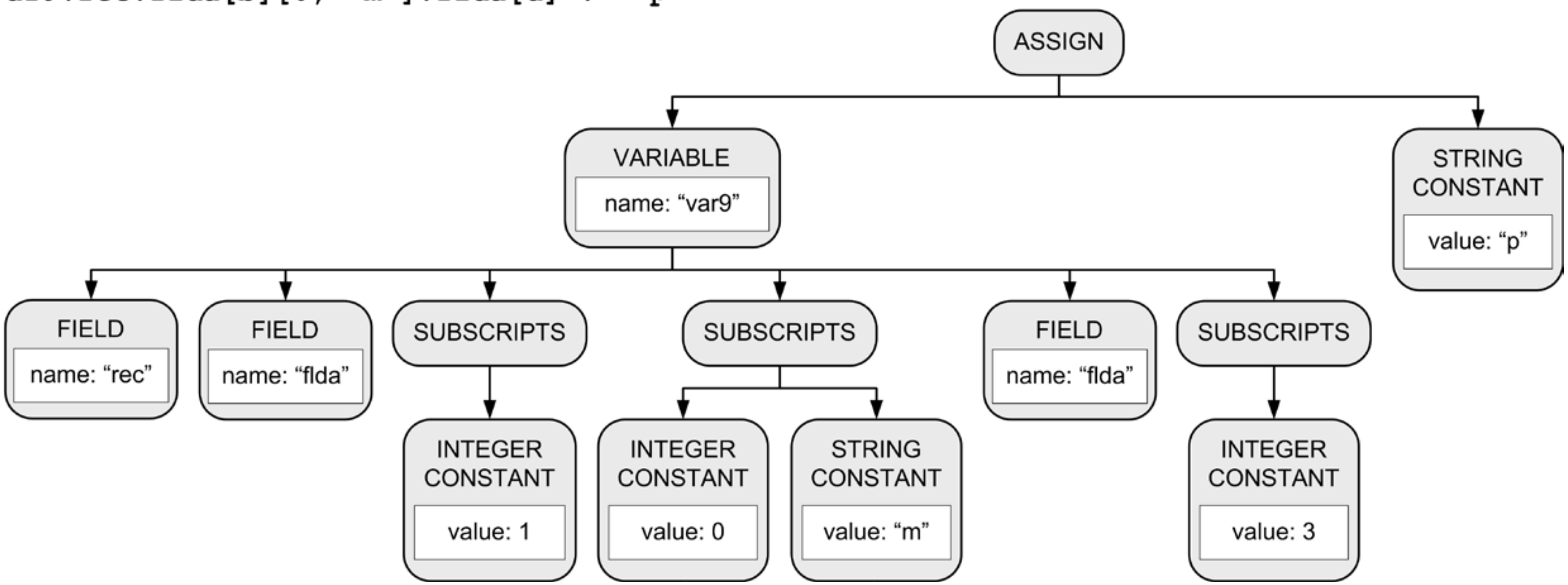
```
var9.rec.flda[b][0, 'm'].flda[d] := 'p'
```



- Create a SUBSCRIPTS node.
- Loop to parse a comma-separated list of subscript expressions.
  - The SUBSCRIPTS node adopts each expression parse tree.

VariableParser.parseSubscripts()

```
var9.rec.flda[b][0, 'm'].flda[d] := 'p'
```

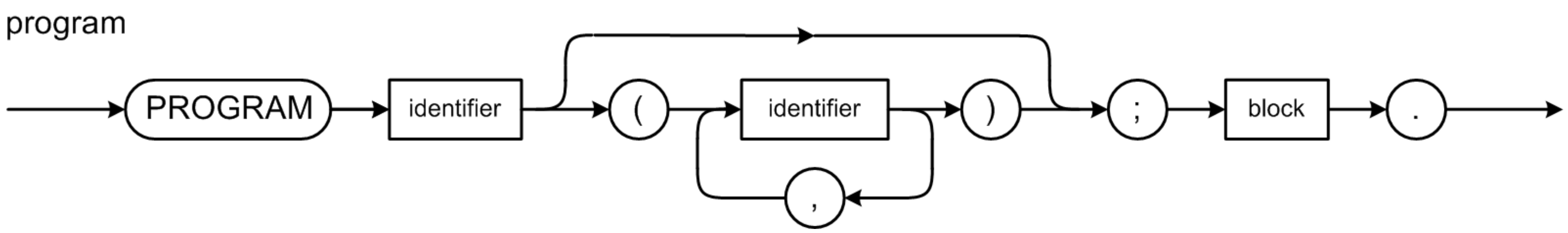


- Verify that each subscript expression is <u>assignment-compatible</u> with the corresponding index type.

Demo

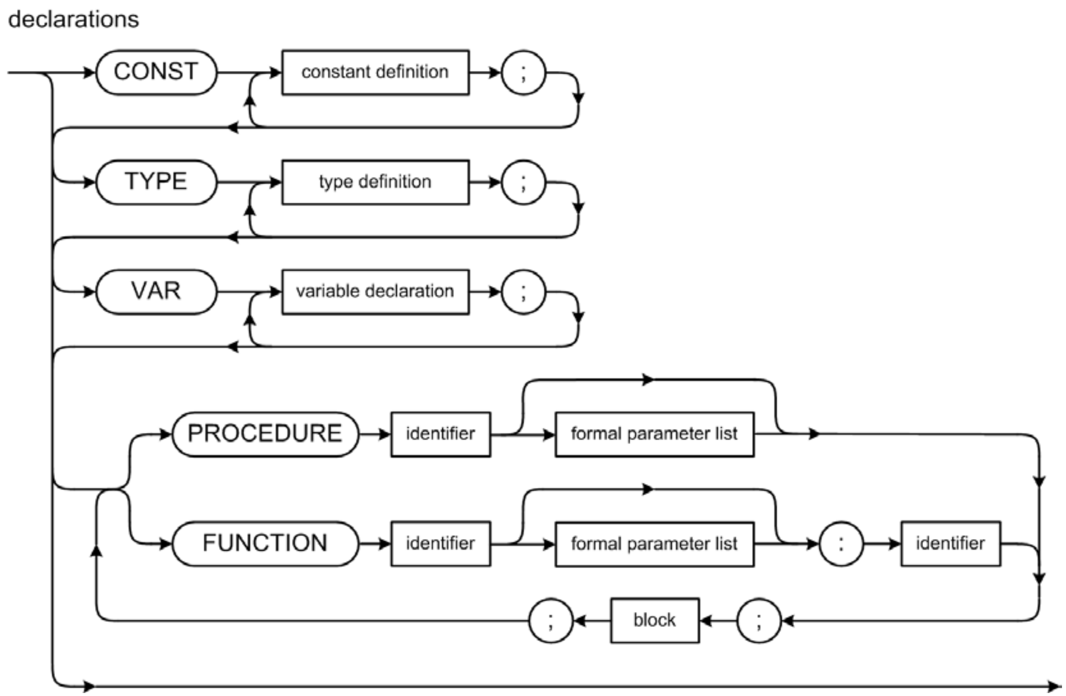
- Pascal Syntax Checker III
  - Parse a Pascal block
    - declarations
    - statements with variables
  - Type checking

Pascal Program Header



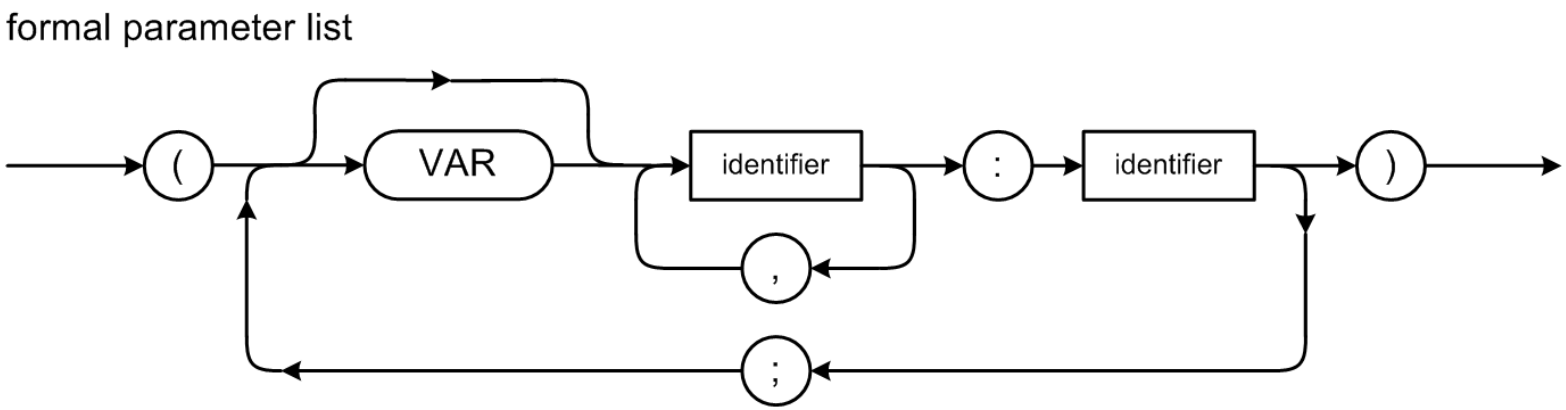
- The <u>program parameters</u> are optional.
  - Identifiers of input and output file variables.
  - Default files are standard input and standard output.
- Examples:
  - `PROGRAM newton;`
  - `PROGRAM hilbert(input, output, error);`

Pascal Programs, Procedures, and Functions



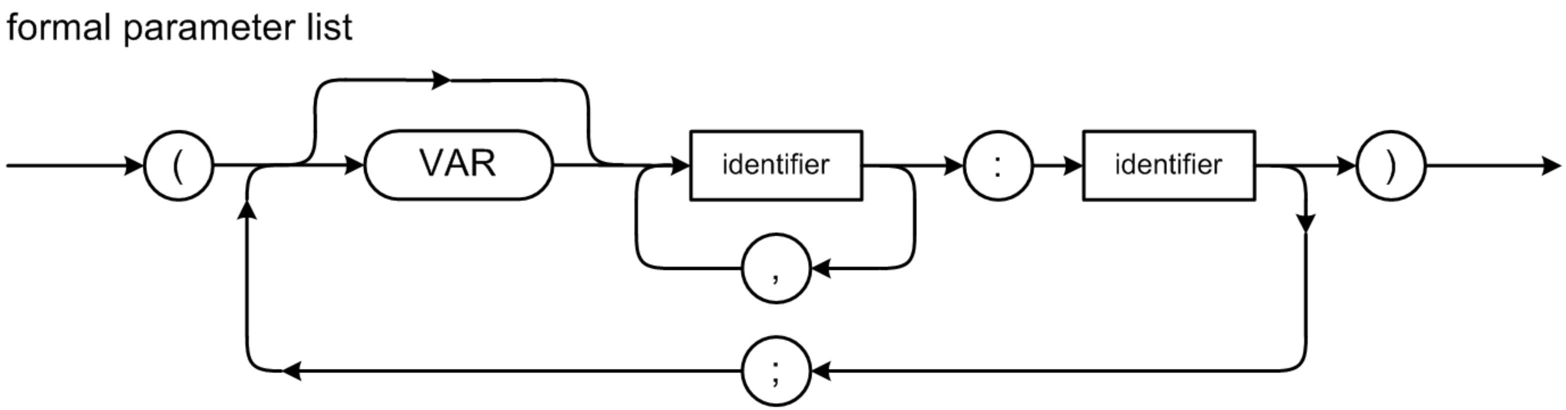
- <u>Procedure and function declarations</u> come last.
  - Any number of procedures and functions, and in any order.
  - A <u>formal parameter list</u> is optional.

Formal Parameter List



- By default, parameters are passed by value.
- The actual parameter value in the call is copied and the formal parameter is assigned the copied value.
  - The routine cannot change the actual parameter value.

Formal Parameter List, *cont'd*



- **VAR** parameters are passed by reference.
- The formal parameter is assigned a reference to the actual parameter value.
  - The routine can change the actual parameter value.

Example Procedure and Function Declarations

```
PROCEDURE proc (j, k : integer; VAR x, y, z : real; VAR v : arr;
                VAR p : boolean; ch : char);
    BEGIN
        ...
    END;

PROCEDURE SortWords;
    BEGIN
        ...
    END;

FUNCTION func (VAR x : real; i, n : integer) : real;
    BEGIN
        ...
        func := ...;
        ...
    END;
```

Value and **VAR** parameters.

No parameters.

Function return type.

Assign the function return value.

Forward Declarations

- In Pascal, you cannot have a statement that calls a procedure or a function before it has been declared.

- To get around this restriction, use forward declarations.
  - Example:

```
FUNCTION foo(m : integer; VAR t : real) : real;
    forward;
```

- Instead of a block, you have `forward`.
  - `forward` is not a reserved word.

Forward Declarations, *cont'd*

- When you finally have the full declaration of a forwarded procedure or function, you do not repeat the formal parameters or the function return type.

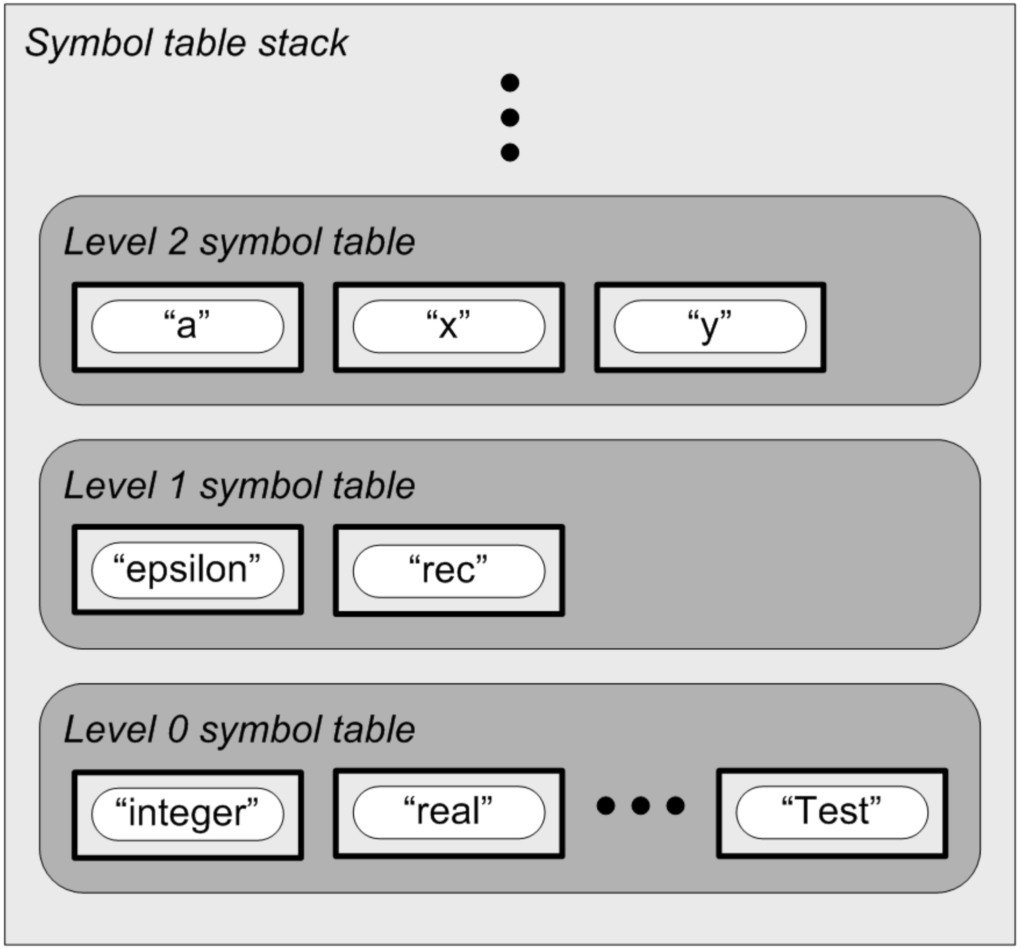
```
FUNCTION foo(m : integer; VAR t : real) : real;
    forward;

PROCEDURE proc;
    VAR x, y : real;
    BEGIN
        x := foo(12, y);
    END;

FUNCTION foo;
    BEGIN
        ...
        foo := ...;
        ...
    END;
```

Use the function before its full declaration.

Now the full function declaration.

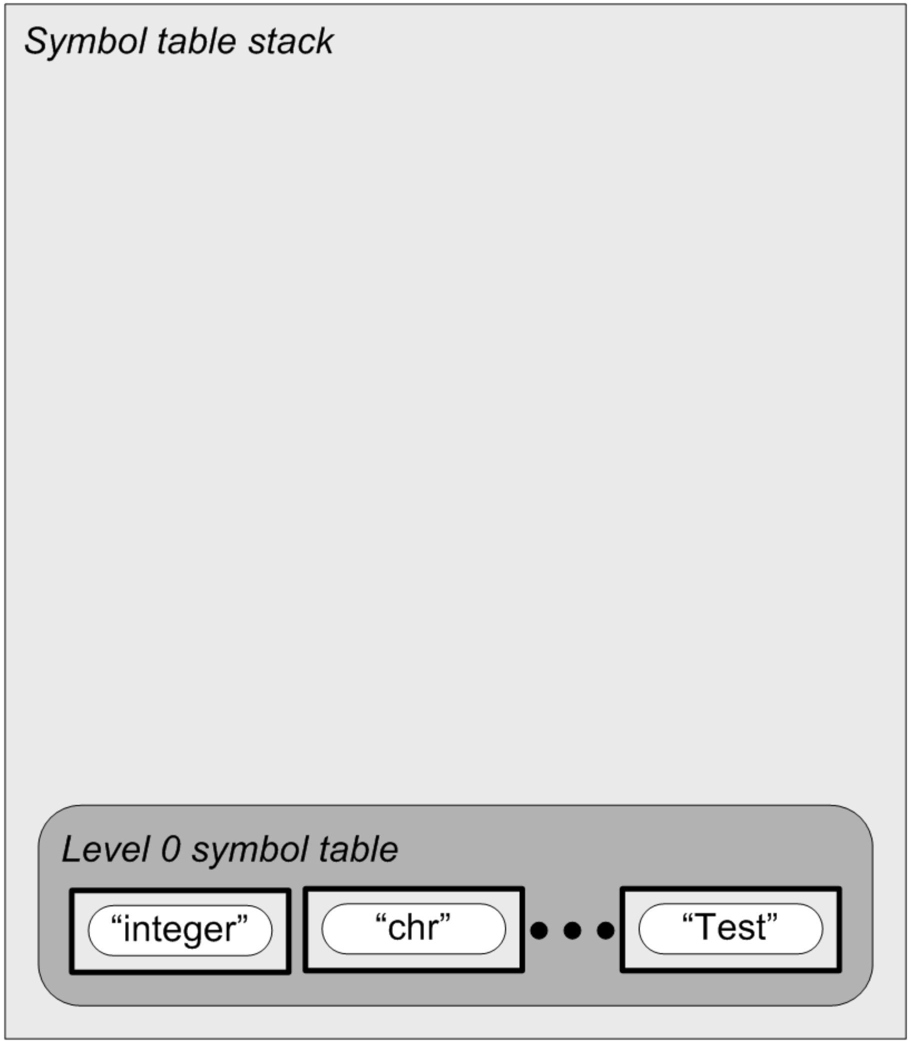
Records and the Symbol Table Stack

```
PROGRAM Test;
CONST
    epsilon = 1.0e-6;
TYPE
    rec = RECORD
              a : real;
              x, y : integer;
          END;
•••
```



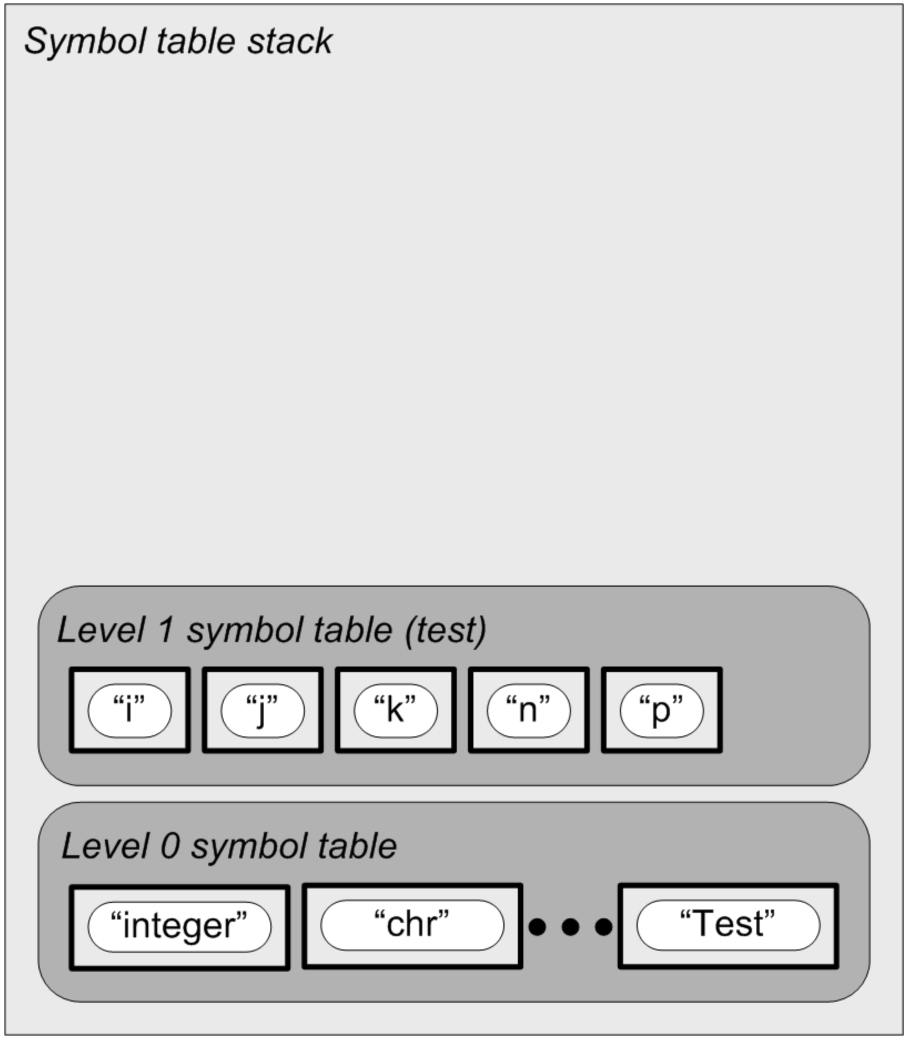
Nested Scopes and the Symbol Table Stack

```
PROGRAM Test;

VAR i, j, k, n : integer;

PROCEDURE p(j : real);
  VAR k : char;

  FUNCTION f(x : real) : real;
    VAR i:real;

    BEGIN {f}
      f := i + j + n + x;
    END {f};

  BEGIN {p}
    k := chr(i + trunc(f(n)));
  END {p};

BEGIN {test}
  p(j + k + n)
END {test}.
```



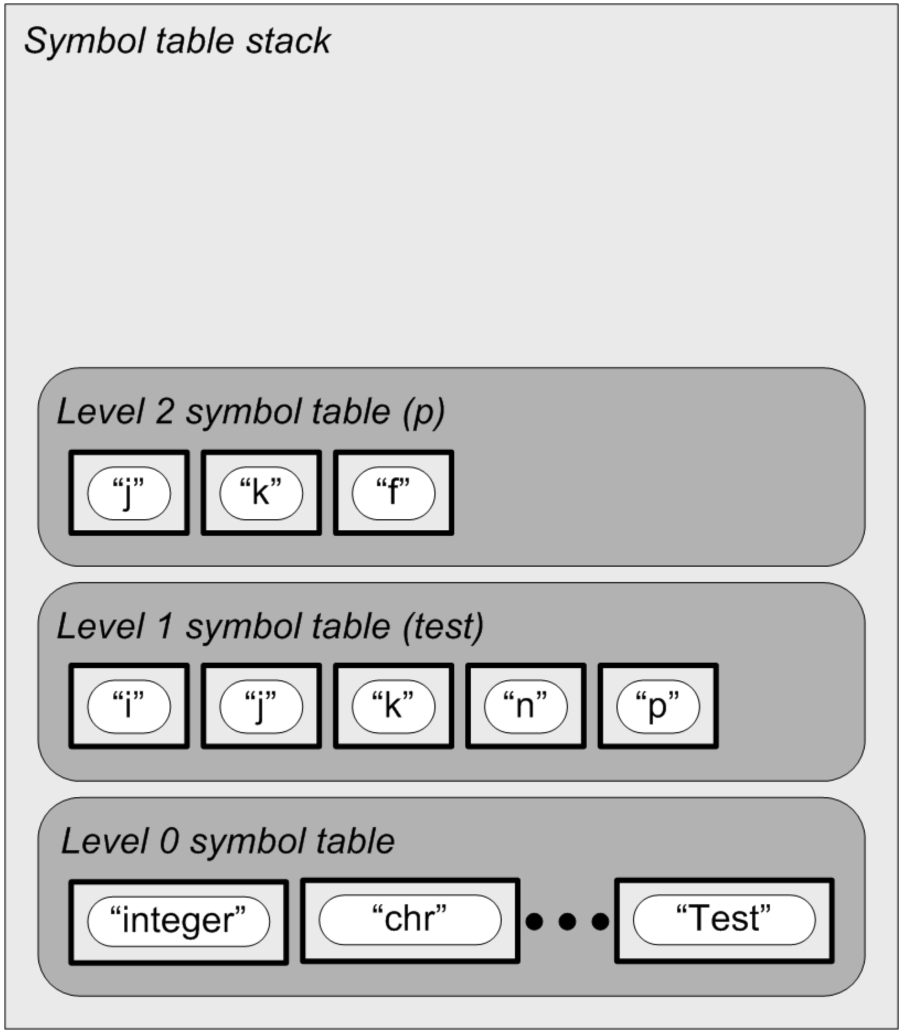
Nested Scopes and the Symbol Table Stack

```
PROGRAM Test;

VAR i, j, k, n : integer;

PROCEDURE p(j : real);
  VAR k : char;

  FUNCTION f(x : real) : real;
    VAR i:real;

    BEGIN {f}
      f := i + j + n + x;
    END {f};

  BEGIN {p}
    k := chr(i + trunc(f(n)));
  END {p};

BEGIN {test}
  p(j + k + n)
END {test}.
```



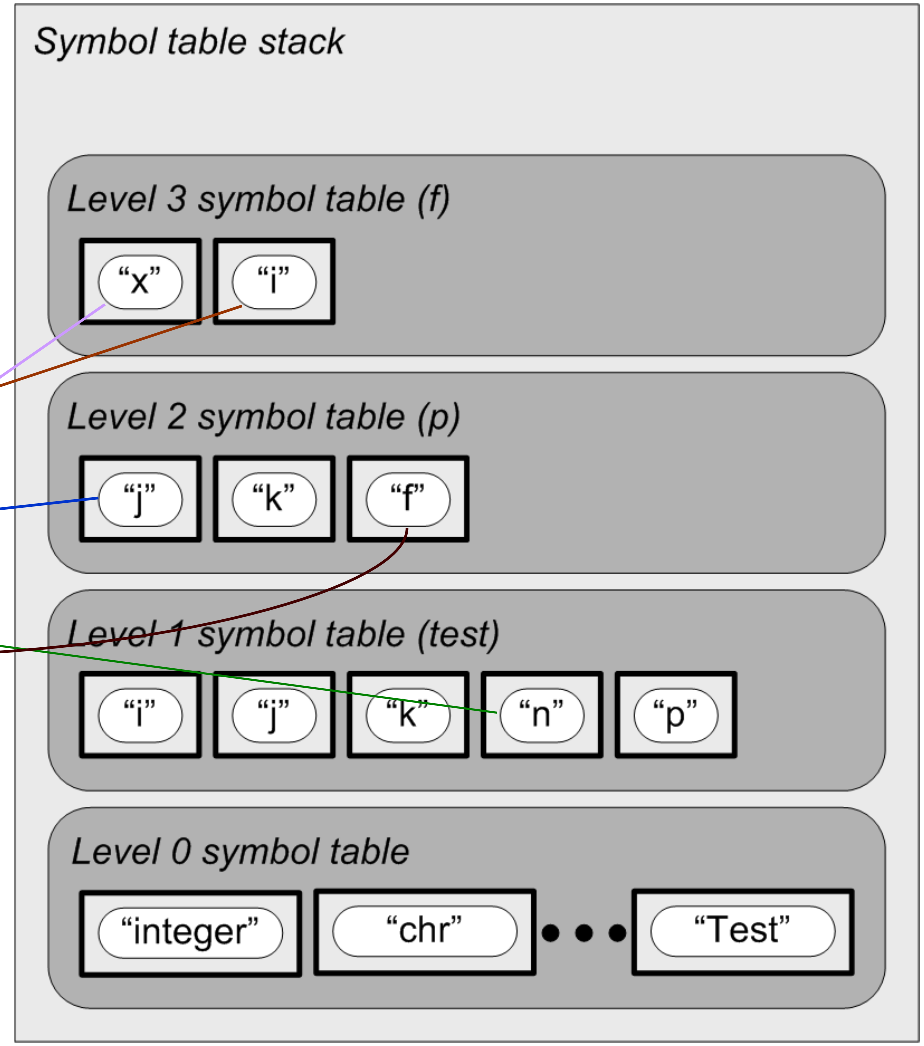
Nested Scopes and the Symbol Table Stack

```
PROGRAM Test;

VAR i, j, k, n : integer;

PROCEDURE p(j : real);
  VAR k : char;

  FUNCTION f(x : real) : real;
    VAR i:real;

    BEGIN {f}
      f := i + j + n + x;
    END {f};

  BEGIN {p}
    k := chr(i + trunc(f(n)));
  END {p};

BEGIN {test}
  p(j + k + n)
END {test}.
```



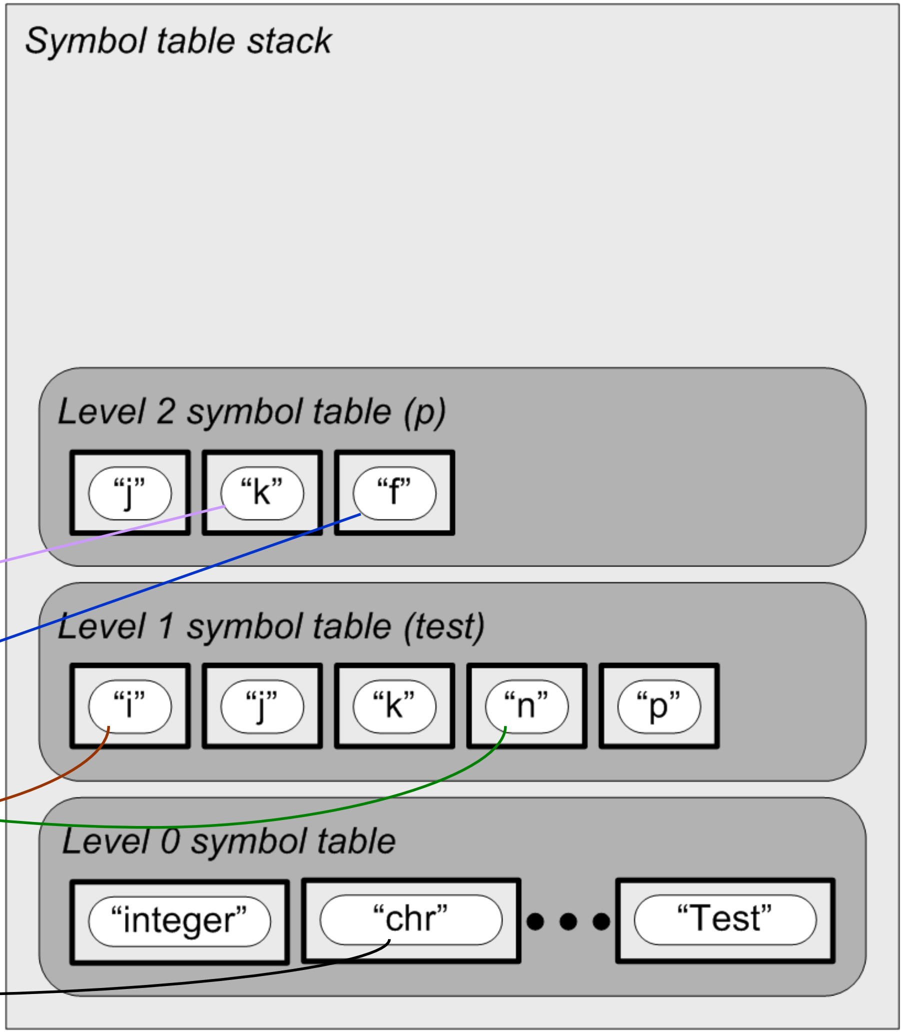
Nested Scopes and the Symbol Table Stack

```
PROGRAM Test;

VAR i, j, k, n : integer;

PROCEDURE p(j : real);
  VAR k : char;

  FUNCTION f(x : real) : real;
    VAR i:real;

    BEGIN {f}
      f := i + j + n + x;
    END {f};

  BEGIN {p}
    k := chr(i + trunc(f(n)));
  END {p};

BEGIN {test}
  p(j + k + n)
END {test}.
```



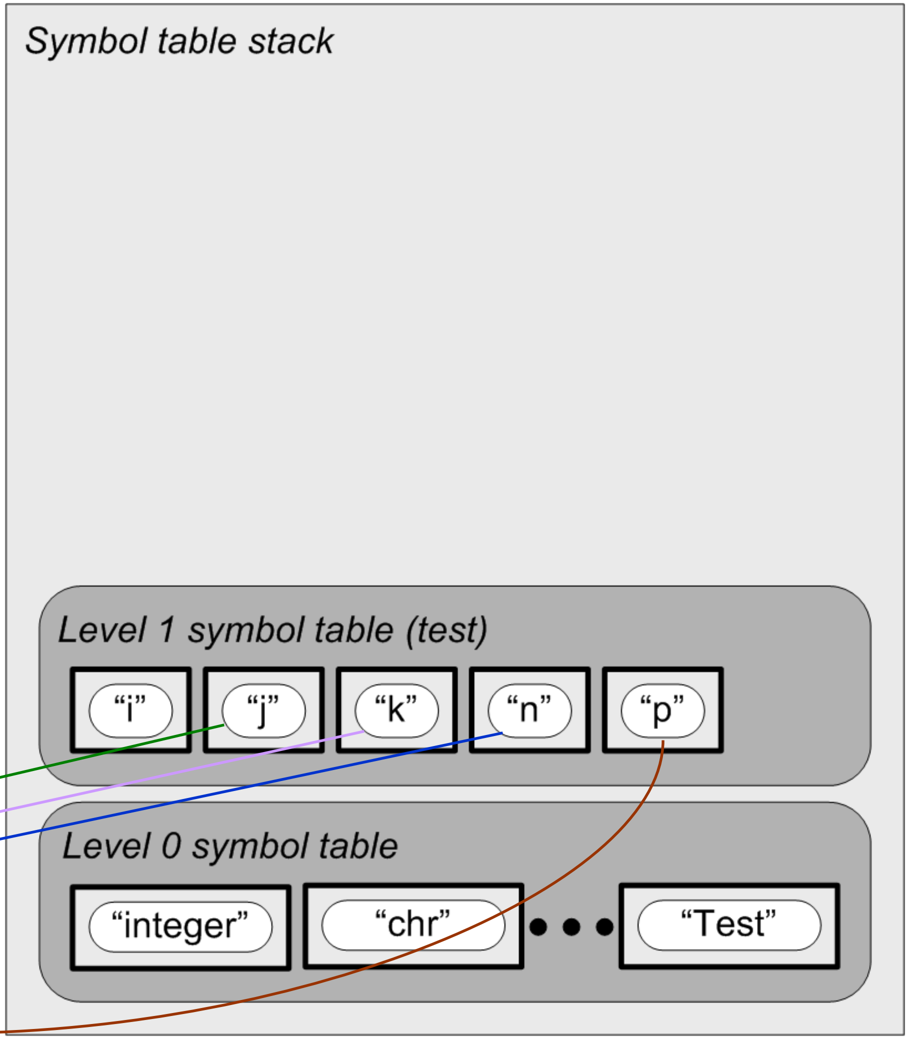
Nested Scopes and the Symbol Table Stack

```
PROGRAM Test;

VAR i, j, k, n : integer;

PROCEDURE p(j : real);
  VAR k : char;

  FUNCTION f(x : real) : real;
    VAR i:real;

    BEGIN {f}
      f := i + j + n + x;
    END {f};

  BEGIN {p}
    k := chr(i + trunc(f(n)));
  END {p};

BEGIN {test}
  p(j + k + n)
END {test}.
```



Nested Scopes and the Symbol Table Stack

```
PROGRAM Test;

VAR i, j, k, n : integer;

PROCEDURE p(j : real);
  VAR k : char;

  FUNCTION f(x : real) : real;
    VAR i:real;

    BEGIN {f}
      f := i + j + n + x;
    END {f};

  BEGIN {p}
    k := chr(i + trunc(f(n)));
  END {p};

BEGIN {test}
  p(j + k + n)
END {test}.
```



Nested Scopes and the Symbol Table Stack

```
PROGRAM Test;

VAR i, j, k, n : integer;

PROCEDURE p(j : real);
  VAR k : char;

  FUNCTION f(x : real) : real;
    VAR i:real;

    BEGIN {f}
      f := i + j + n + x;
    END {f};

  BEGIN {p}
    k := chr(i + trunc(f(n)));
  END {p};

BEGIN {test}
  p(j + k + n)
END {test}.
```