CMPE 152: Compiler Design

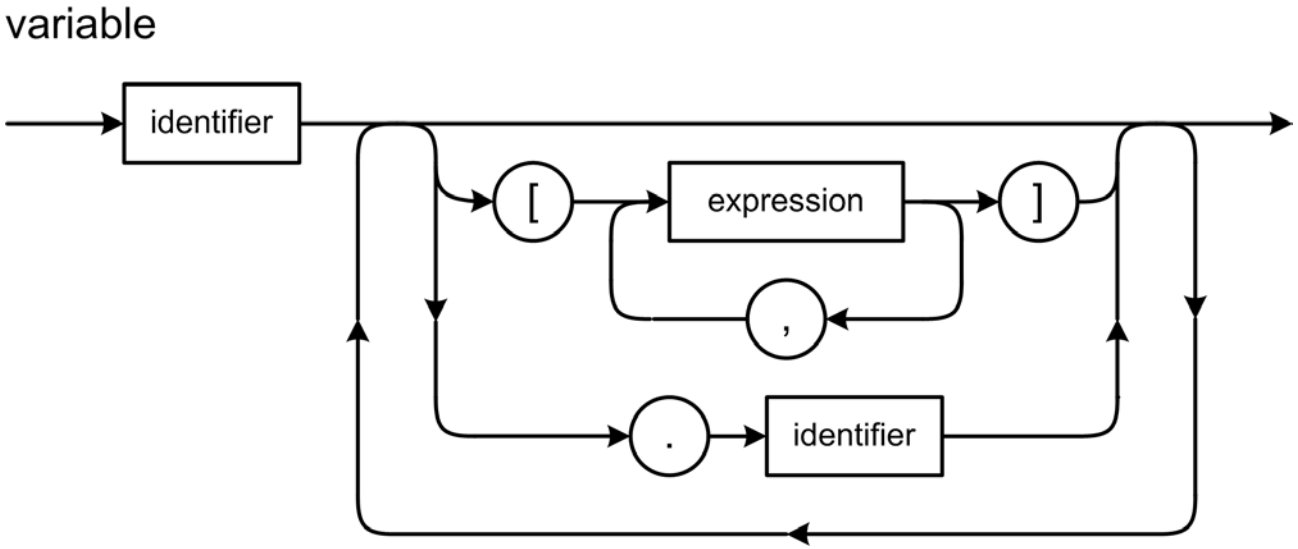
September 28 Class Meeting

Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak

www.cs.sjsu.edu/~mak

Syntax Diagram for Variables



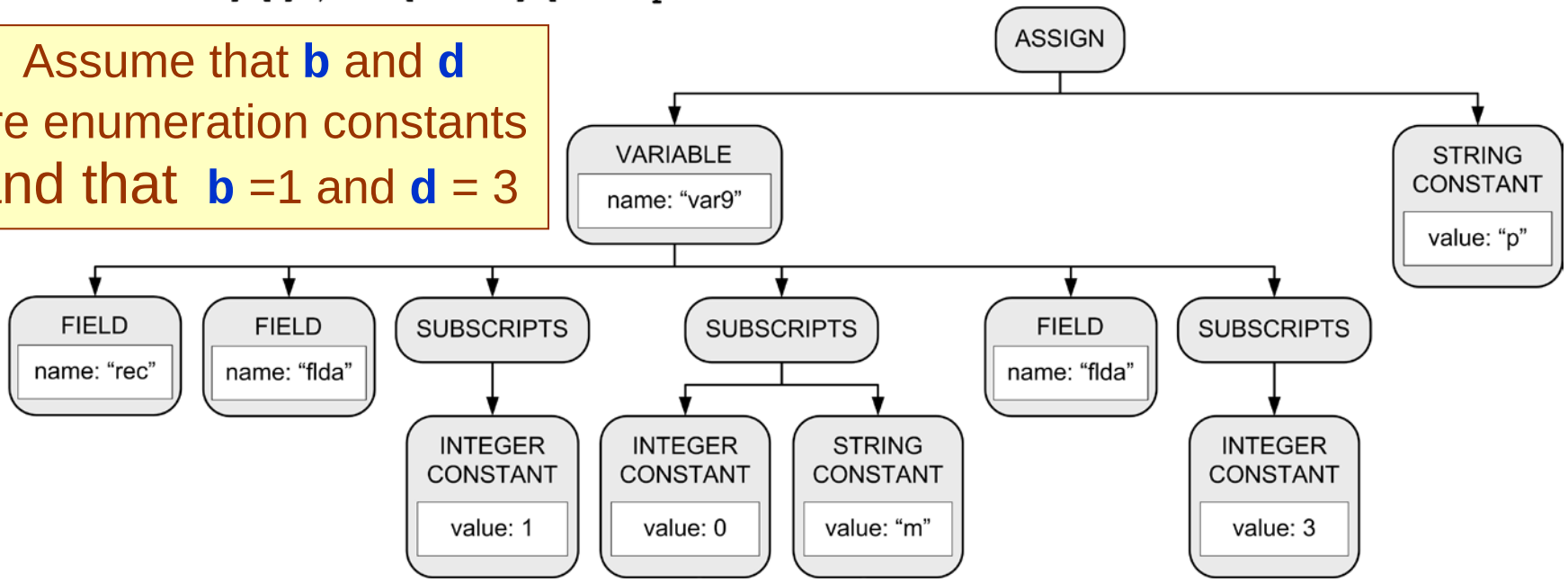
The outer loop back allows any number of subscripts and fields.

- A variable can have any combination 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 type checking for each subscript and field.

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

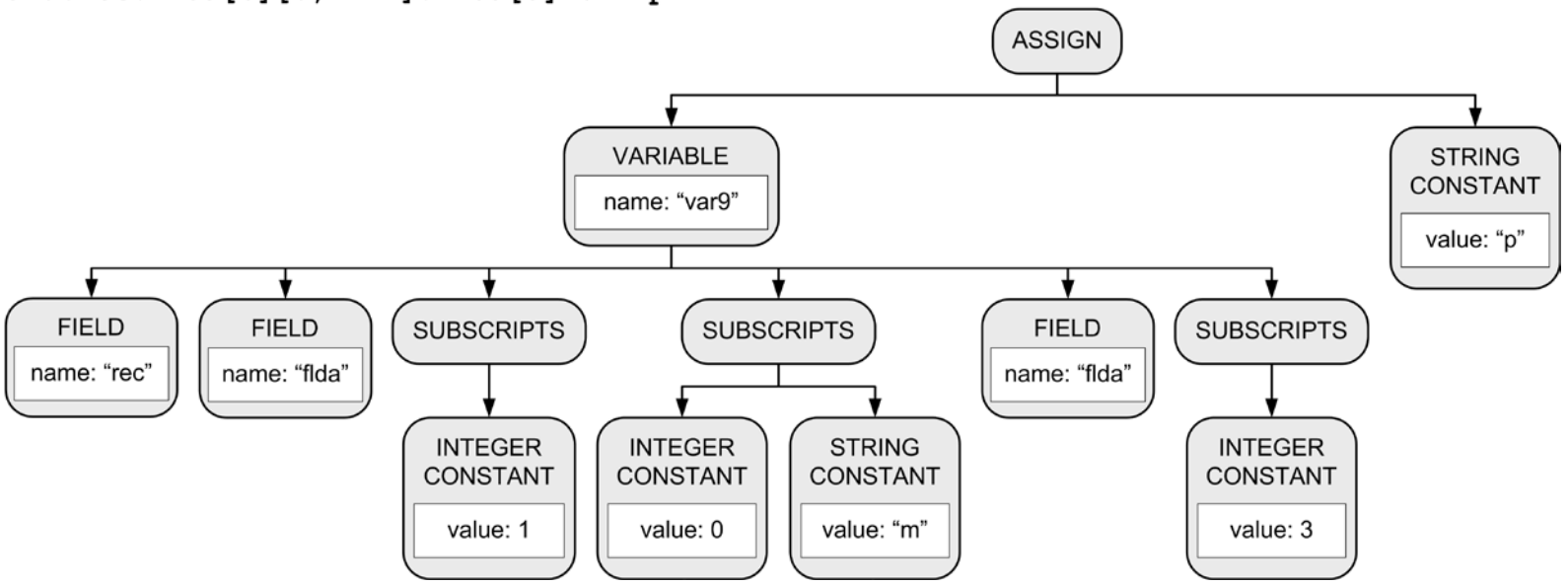
Class `VariableParser`

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

- Parsing methods
  - **`parse()`**
  - **`parse_field()`**
  - **`parse_subscripts()`**

VariableParser::parse()

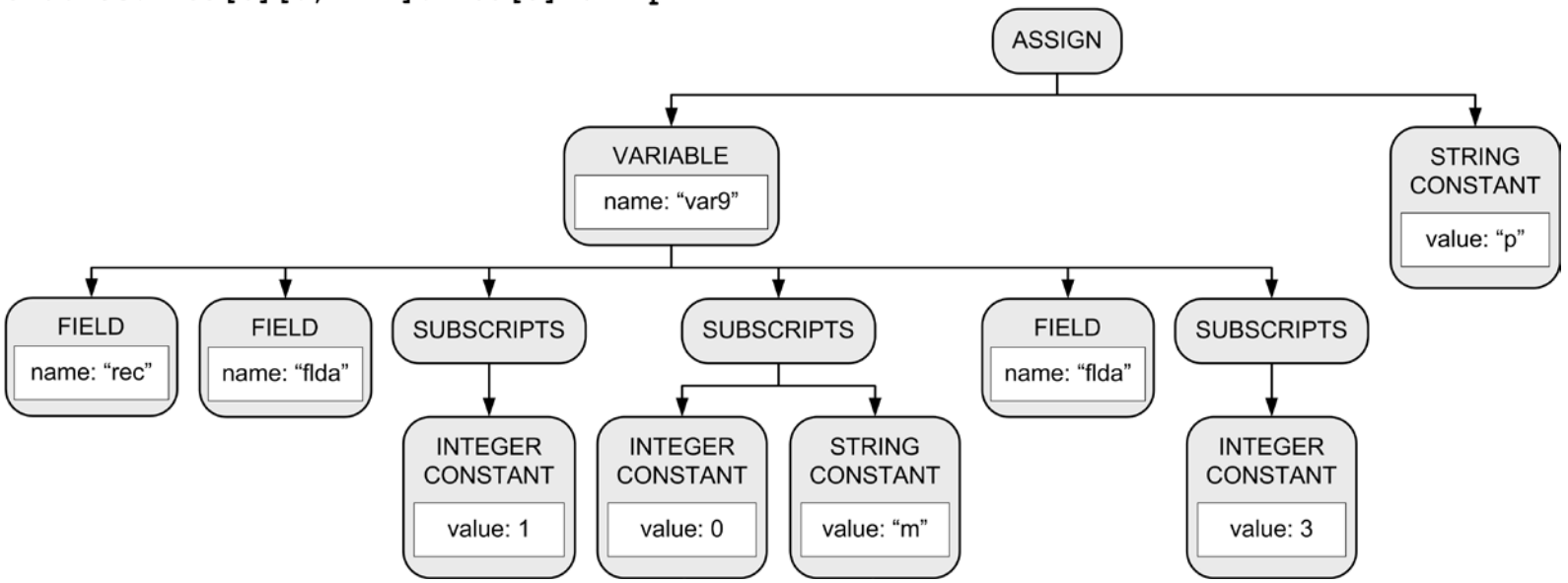
```
var9.rec.flda[b][0, 'm'].flda[d] := 'p'
```



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

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

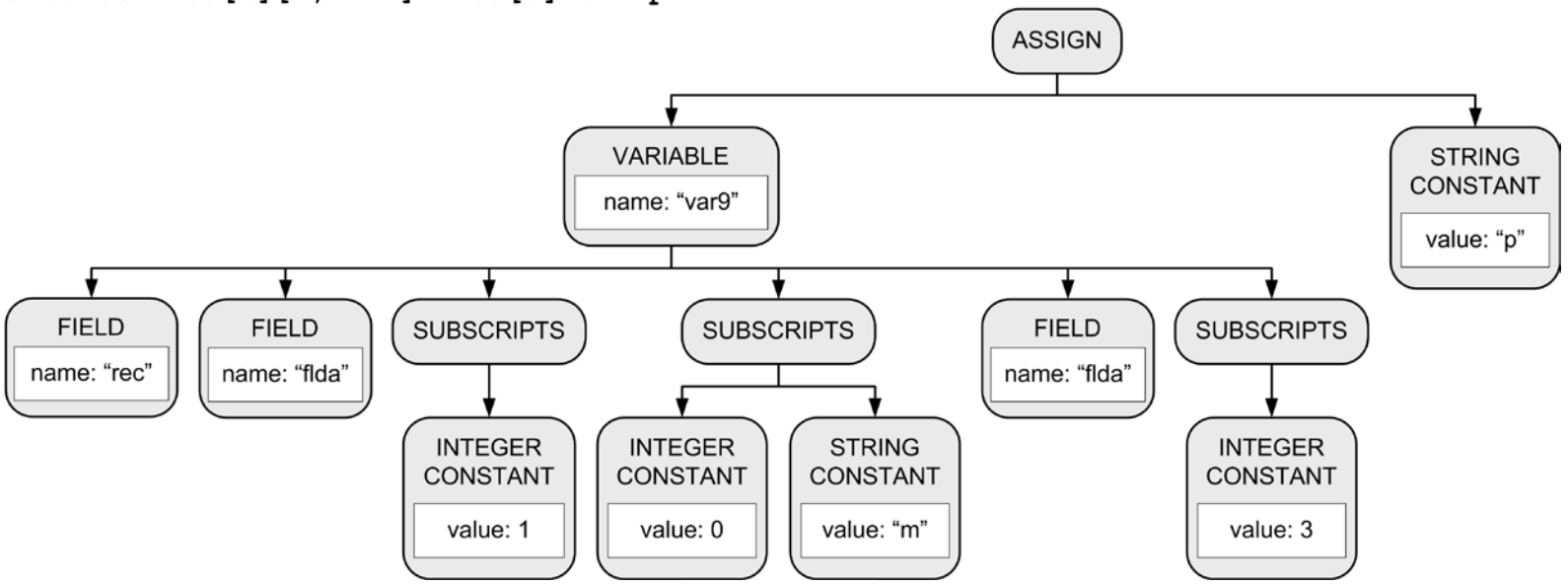
```
var9.rec.flda[b][0, 'm'].flda[d] := 'p'
```



- Loop to parse any subscripts and fields.
  - Call methods **`parse_field()`** or **`parse_subscripts()`**.
  - Variable **`variable_type`** keeps track of the current type specification.
  - The current type changes as each field and subscript is parsed.

VariableParser.parseField()

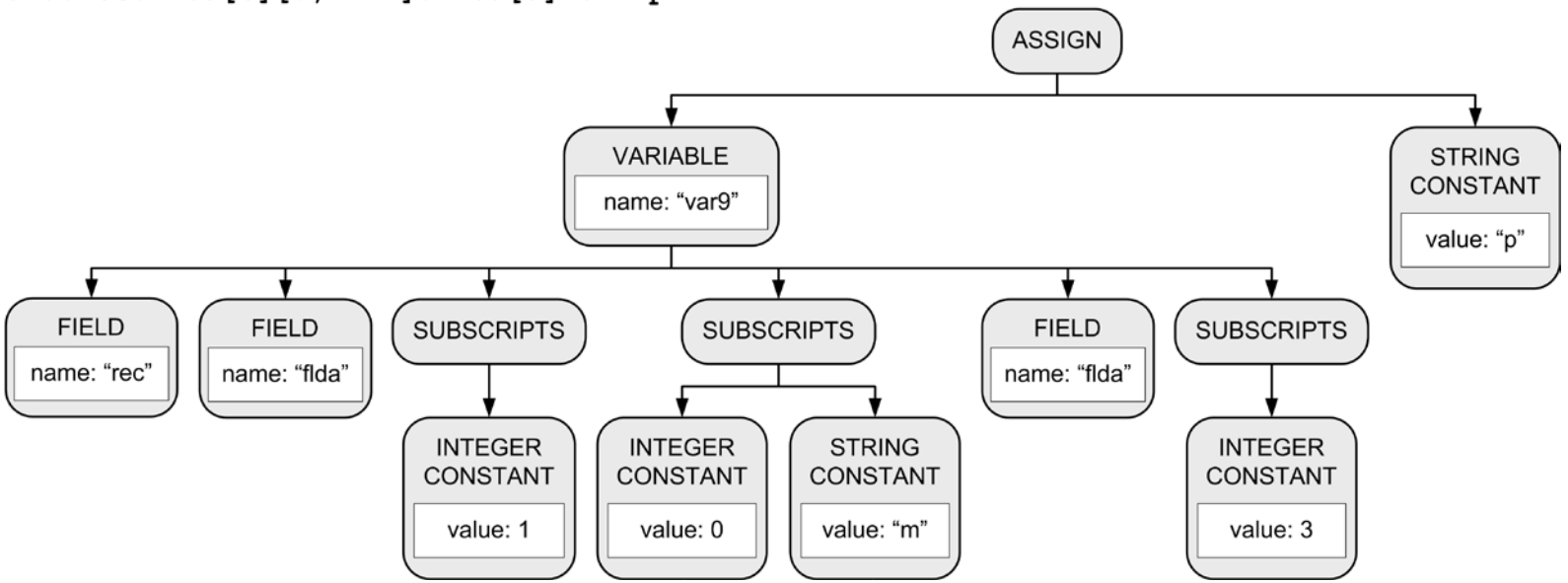
```
var9.rec.flda[b][0, 'm'].flda[d] := 'p'
```



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

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

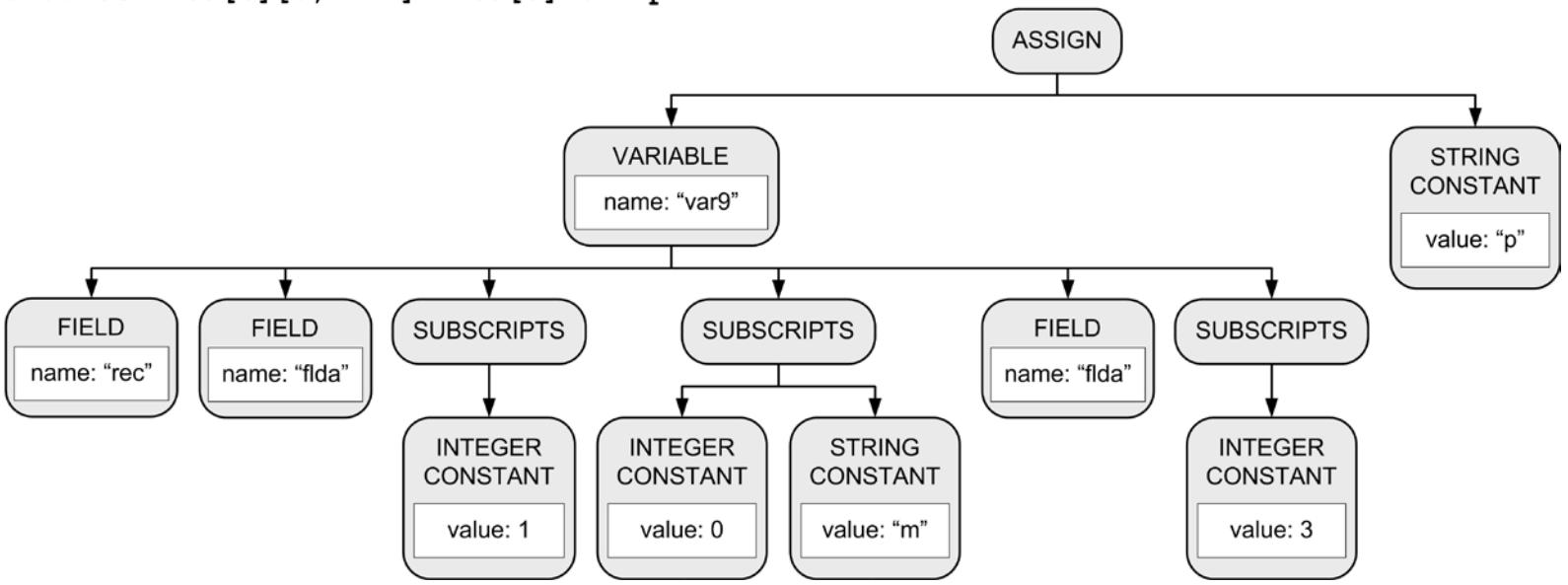
```
var9.rec.flda[b][0, 'm'].flda[d] := 'p'
```



- 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()

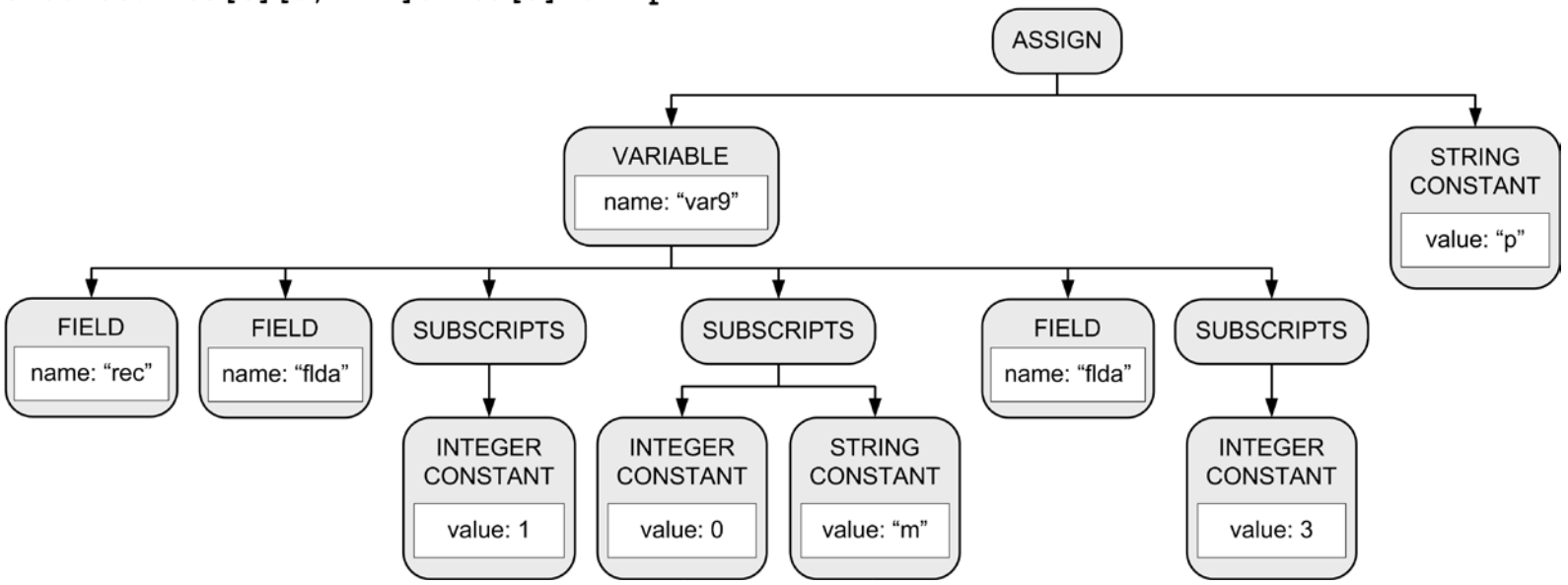
```
var9.rec.flda[b][0, 'm'].flda[d] := 'p'
```



- 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'
```

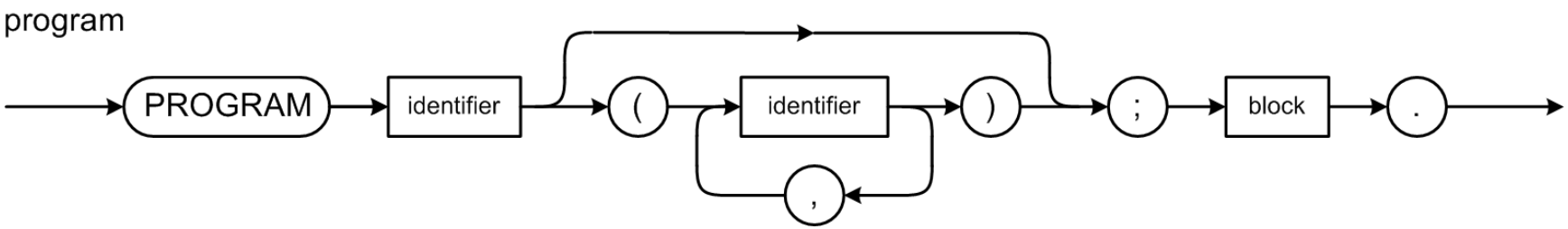


- Verify that each subscript expression is assignment-compatible with the corresponding index type.

Demo

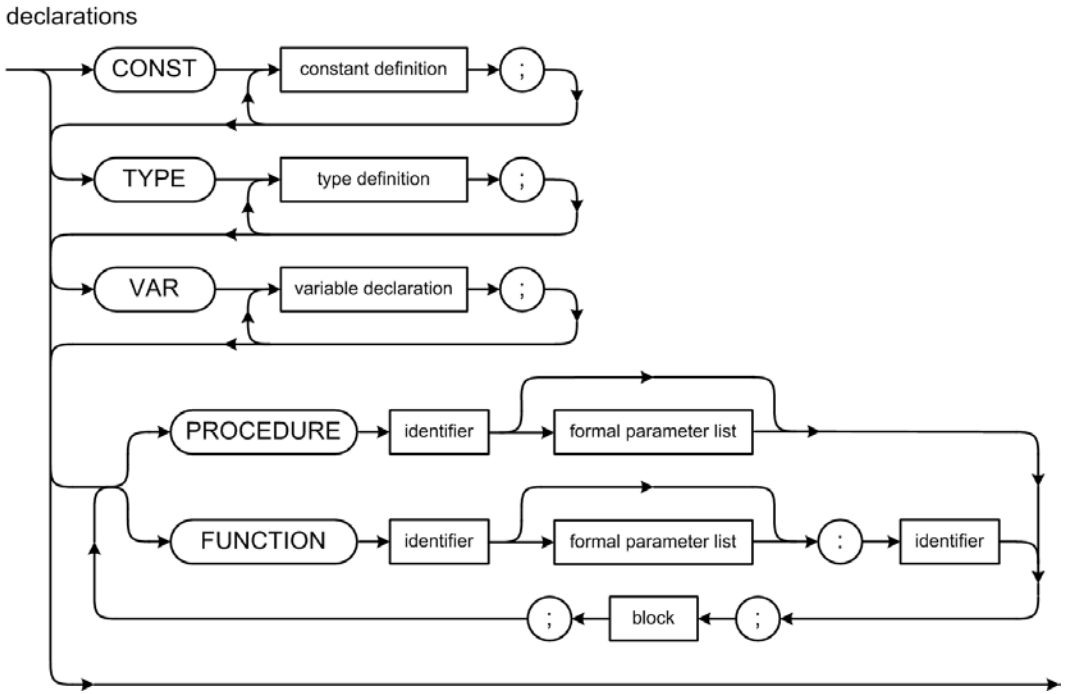
- Pascal Syntax Checker III
  - Parse a Pascal block
    - declarations
    - statements with variables
  - Type checking

Pascal Program Header



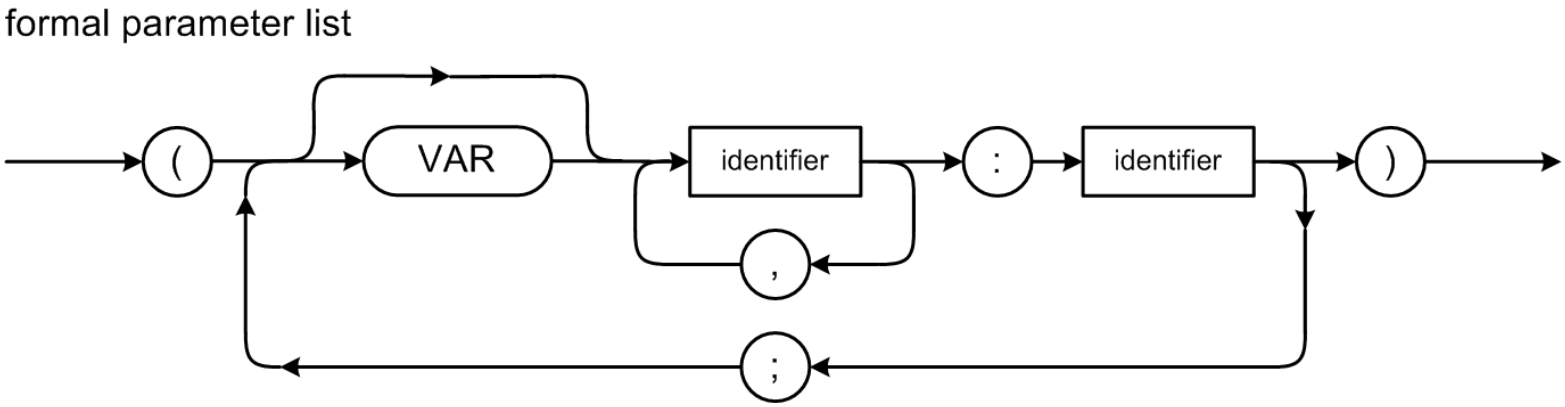
- The program parameters are optional.
  - Identifiers of input and output file variables.
  - Default files are standard input and standard output.
- Examples:
  - `PROGRAM newton;`
  - `PROGRAM hilbert(input, output, error);`

Pascal Programs, Procedures, and Functions



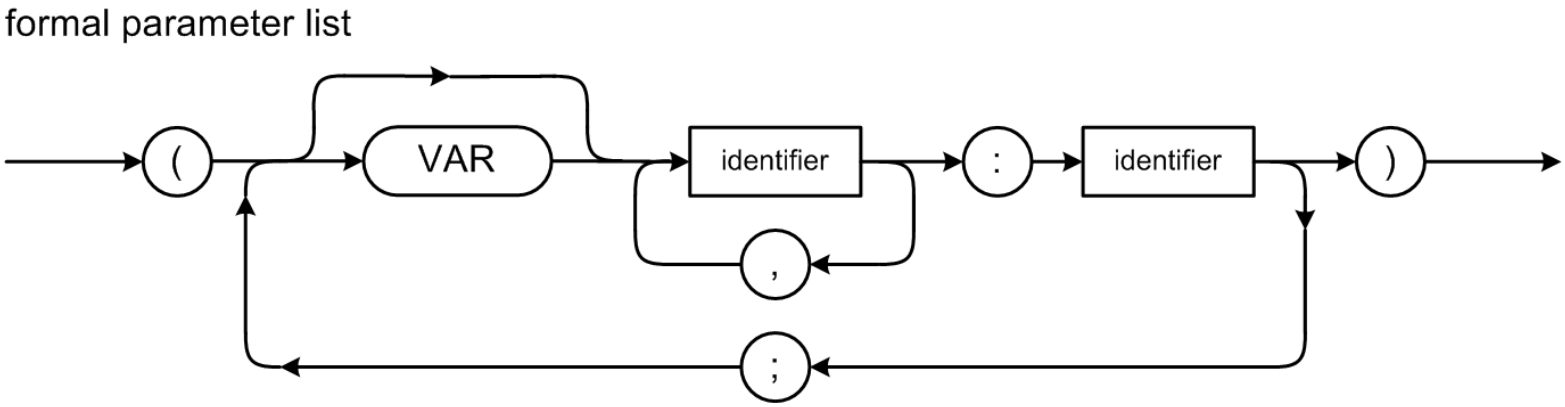
- Procedure and function declarations come last.
  - Any number of procedures and functions, and in any order.
  - A formal parameter list is optional.

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.

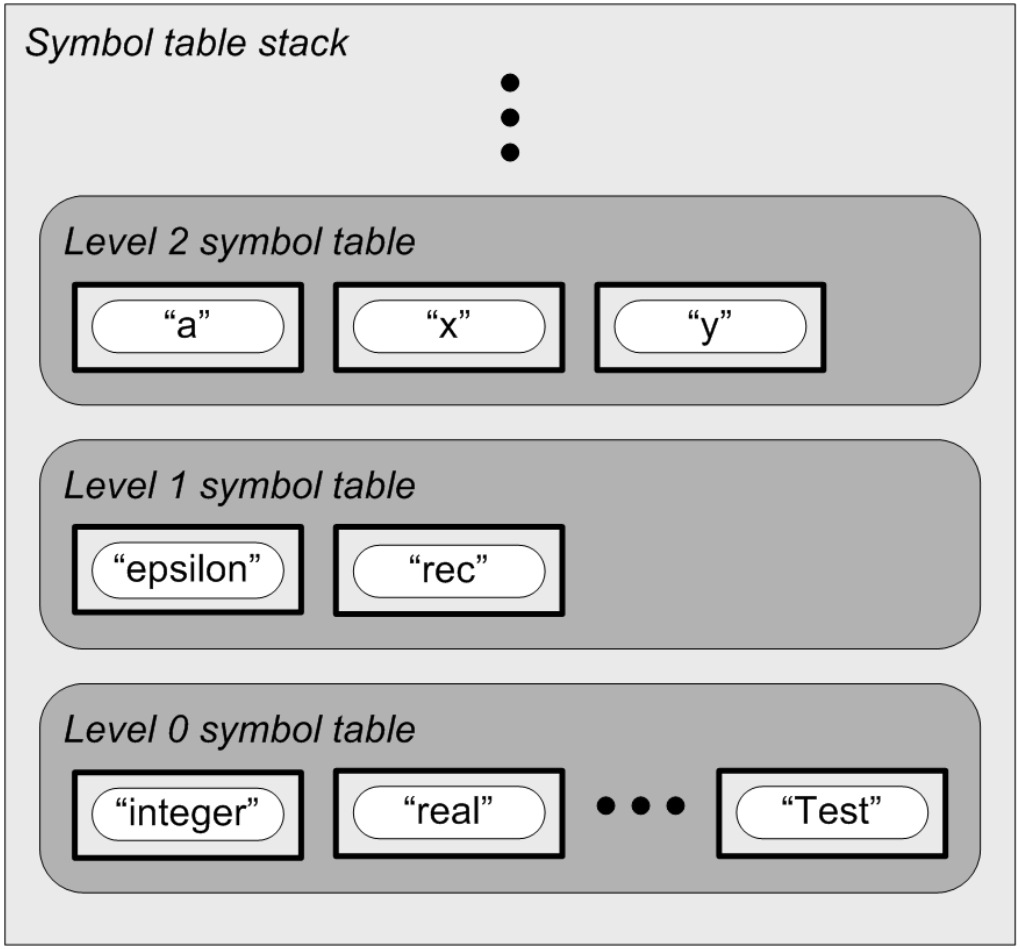
```
FUNCTION foo(m : integer; VAR t : real) : real;
    forward;

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.

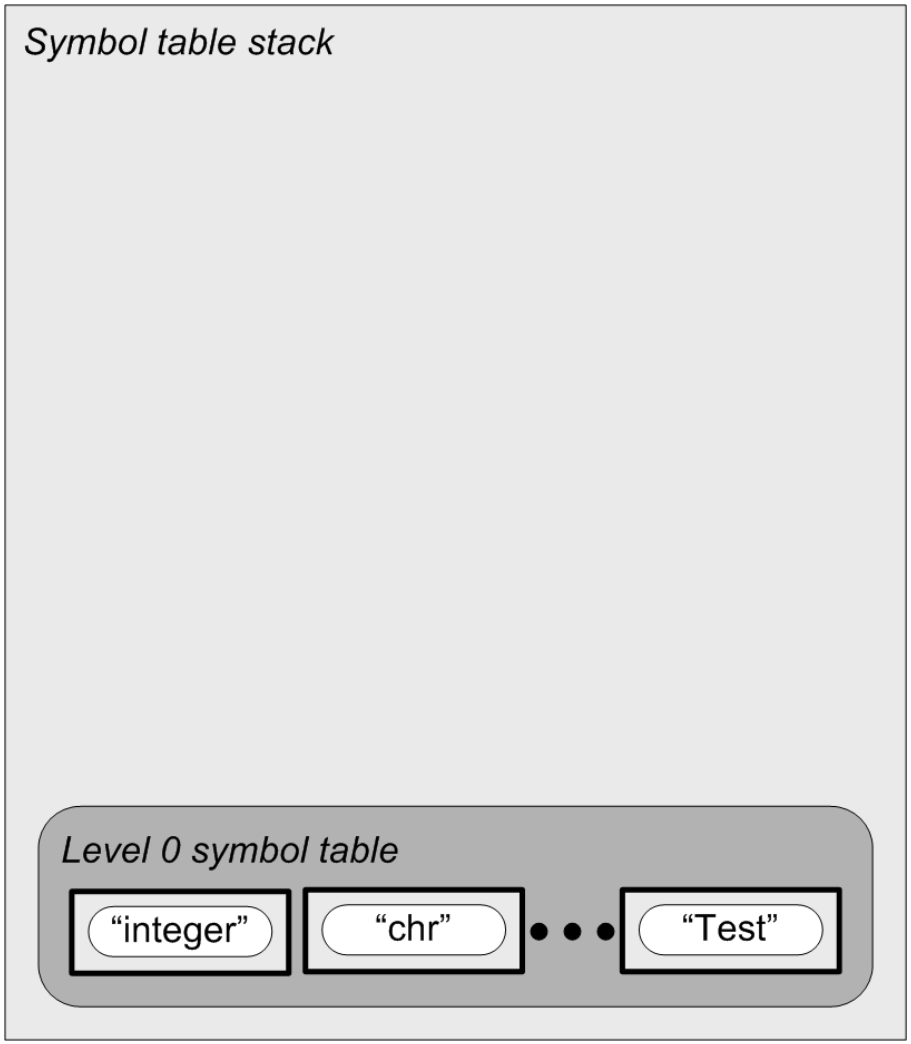
Records and the Symbol Table Stack

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



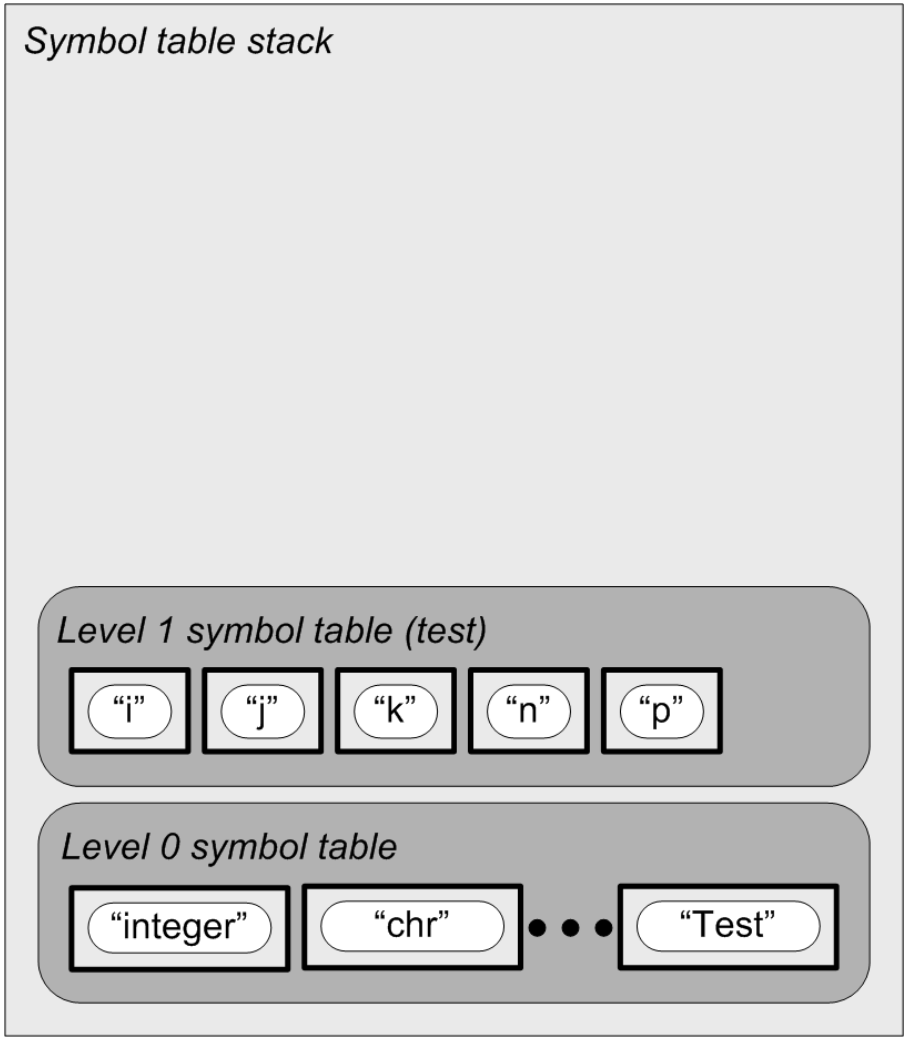
Nested Scopes and the Symbol Table Stack

```
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}.
```



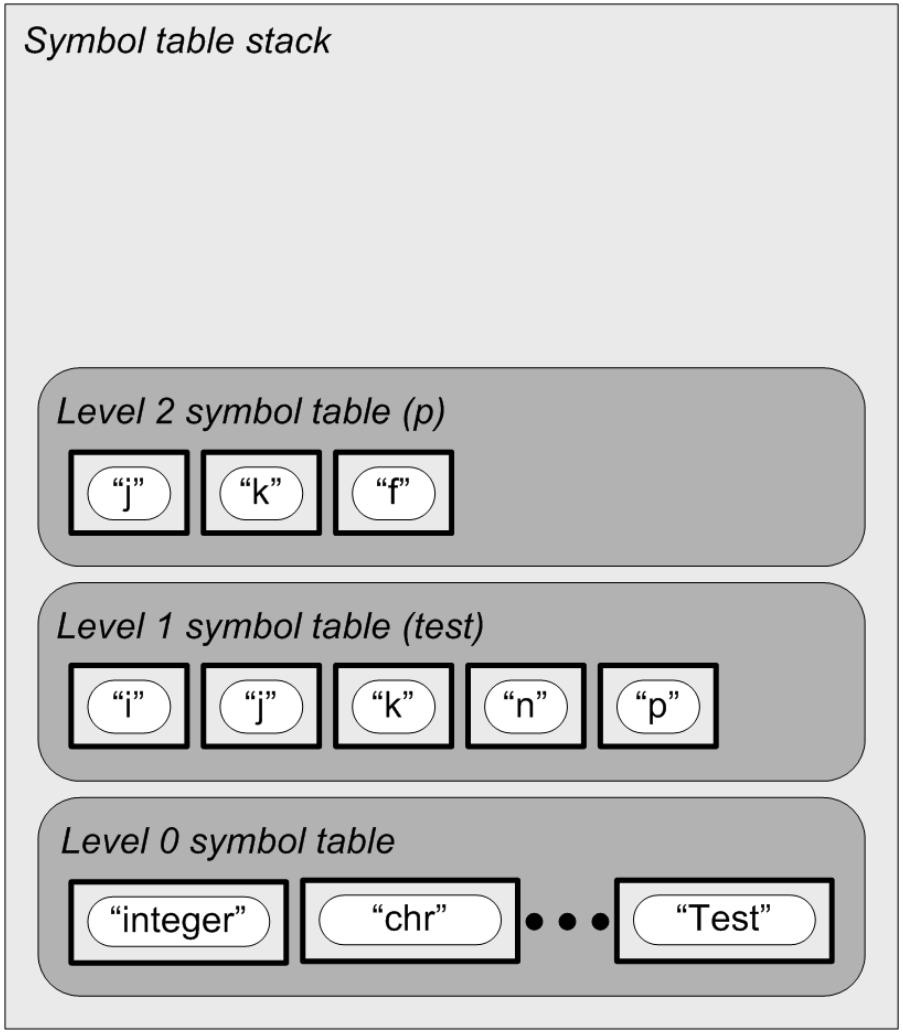
Nested Scopes and the Symbol Table Stack

```
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}.
```



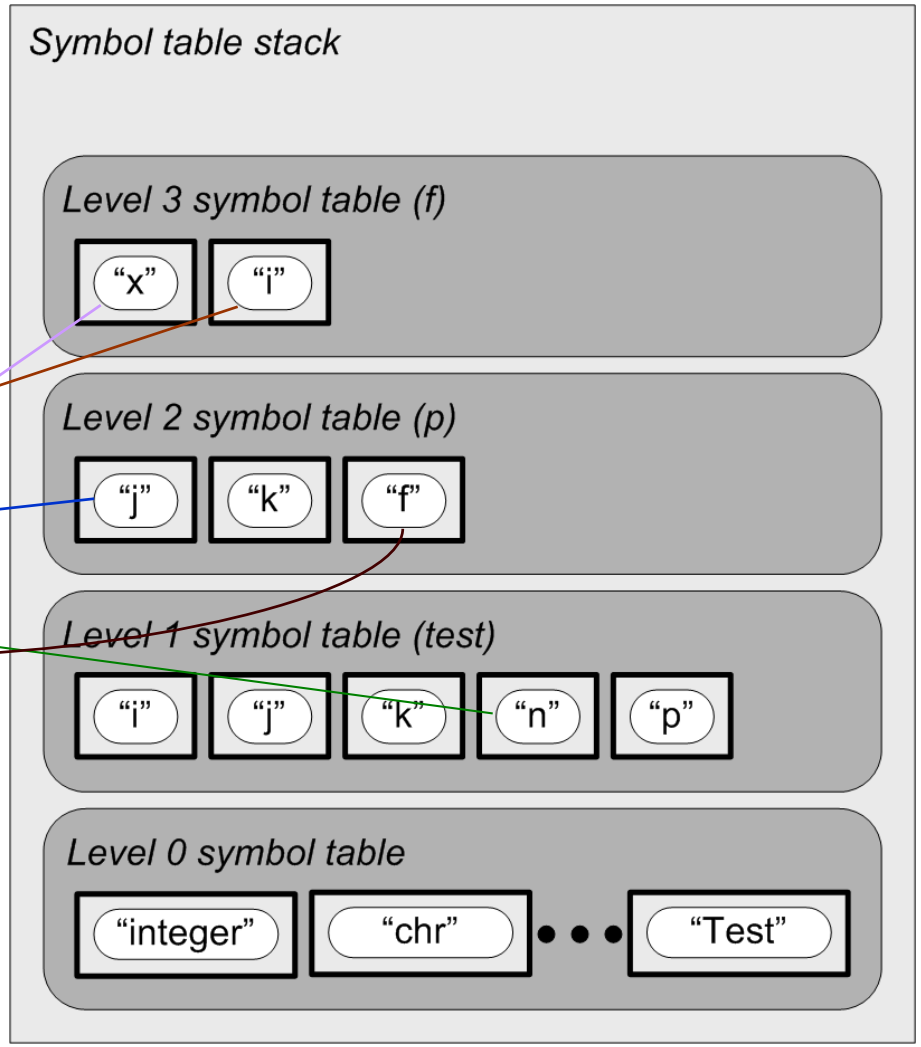
Nested Scopes and the Symbol Table Stack

```
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}.
```



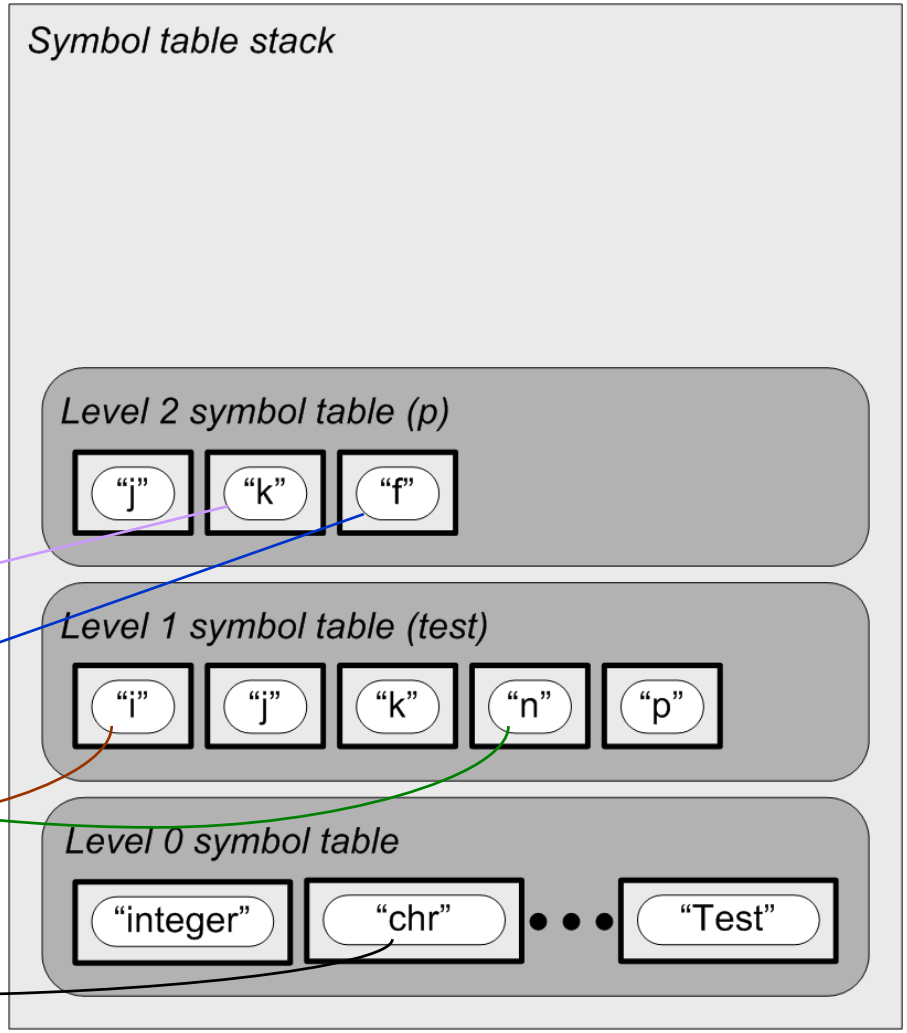
Nested Scopes and the Symbol Table Stack

```
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}.
```



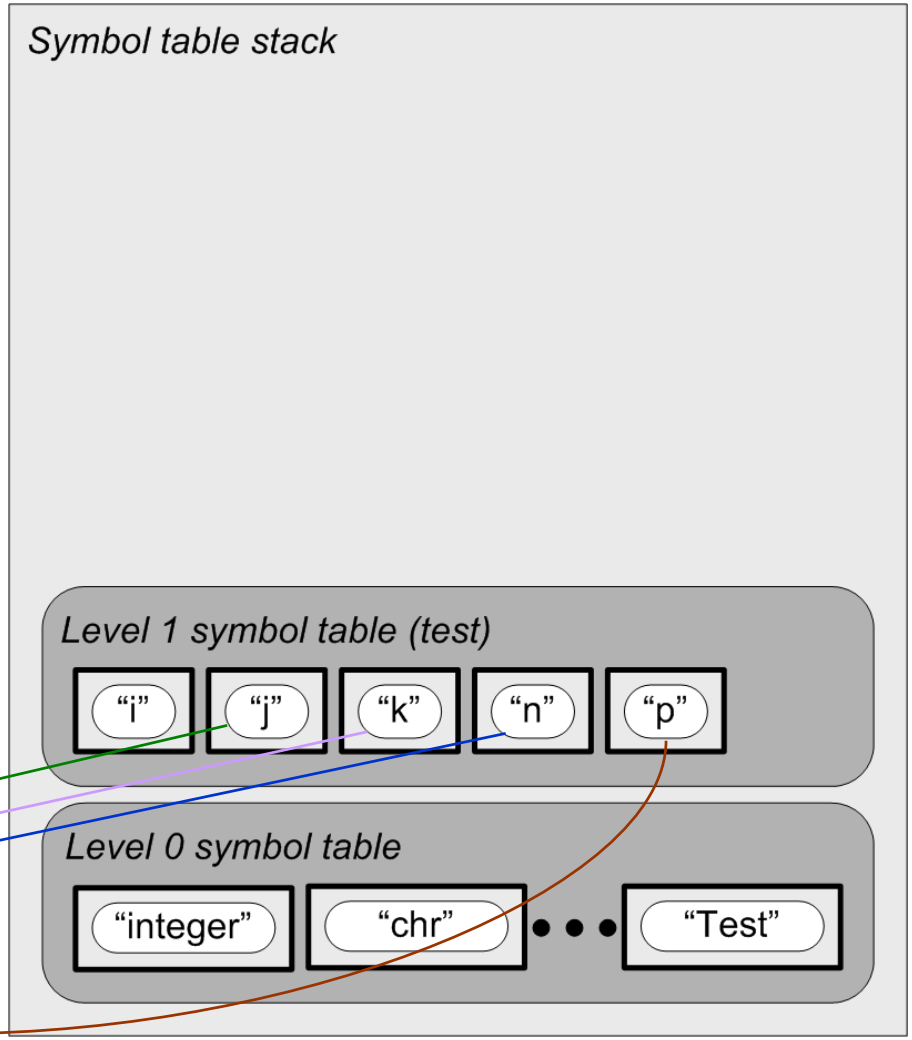
Nested Scopes and the Symbol Table Stack

```
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}.
```