CS 153: Concepts of Compiler Design

October 3 Class Meeting

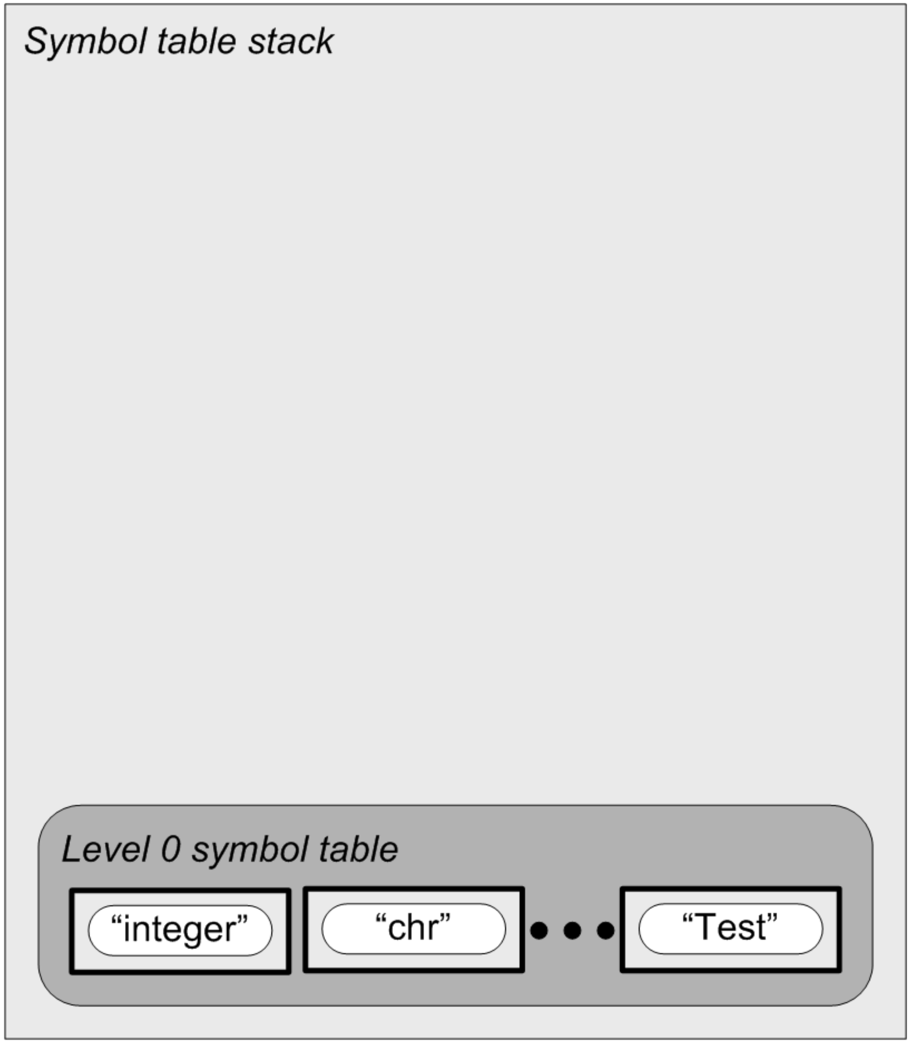
Department of Computer Science
San Jose State University

Fall 2017

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

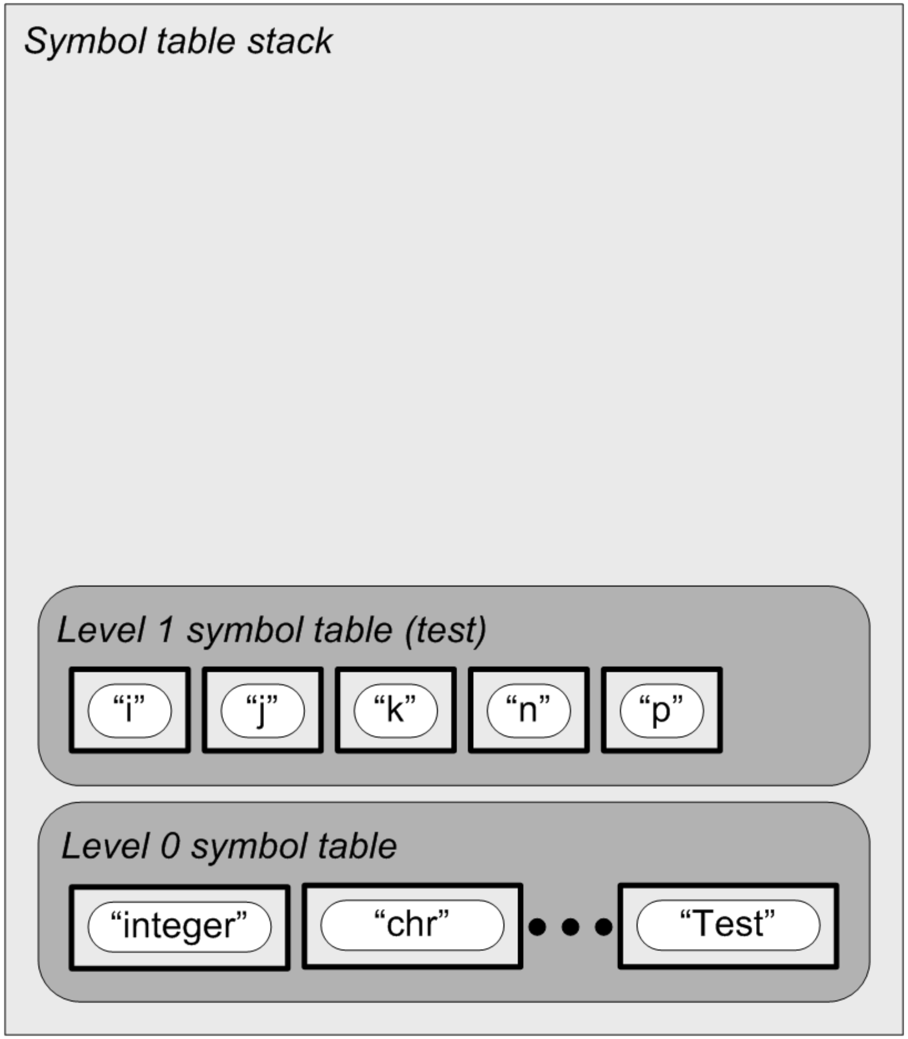
Nested Scopes and the Symbol Table Stack

```
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}.
```



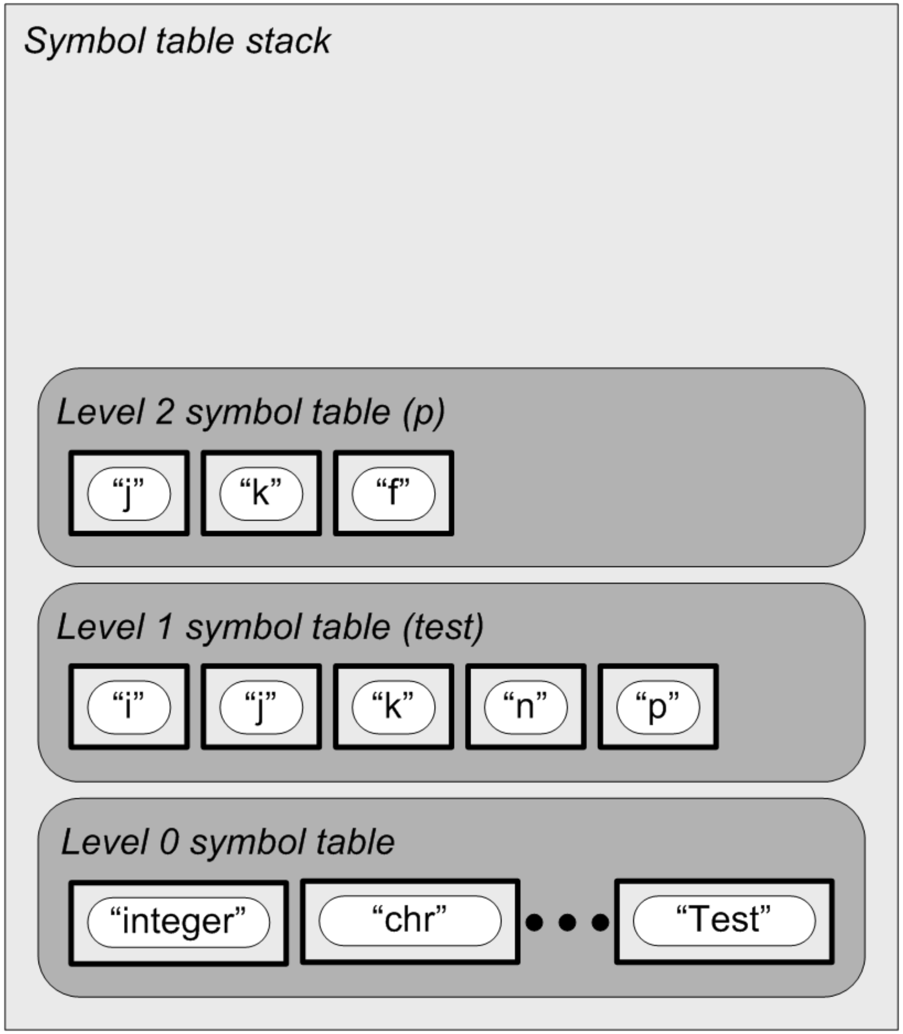
Nested Scopes and the Symbol Table Stack

```
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}.
```



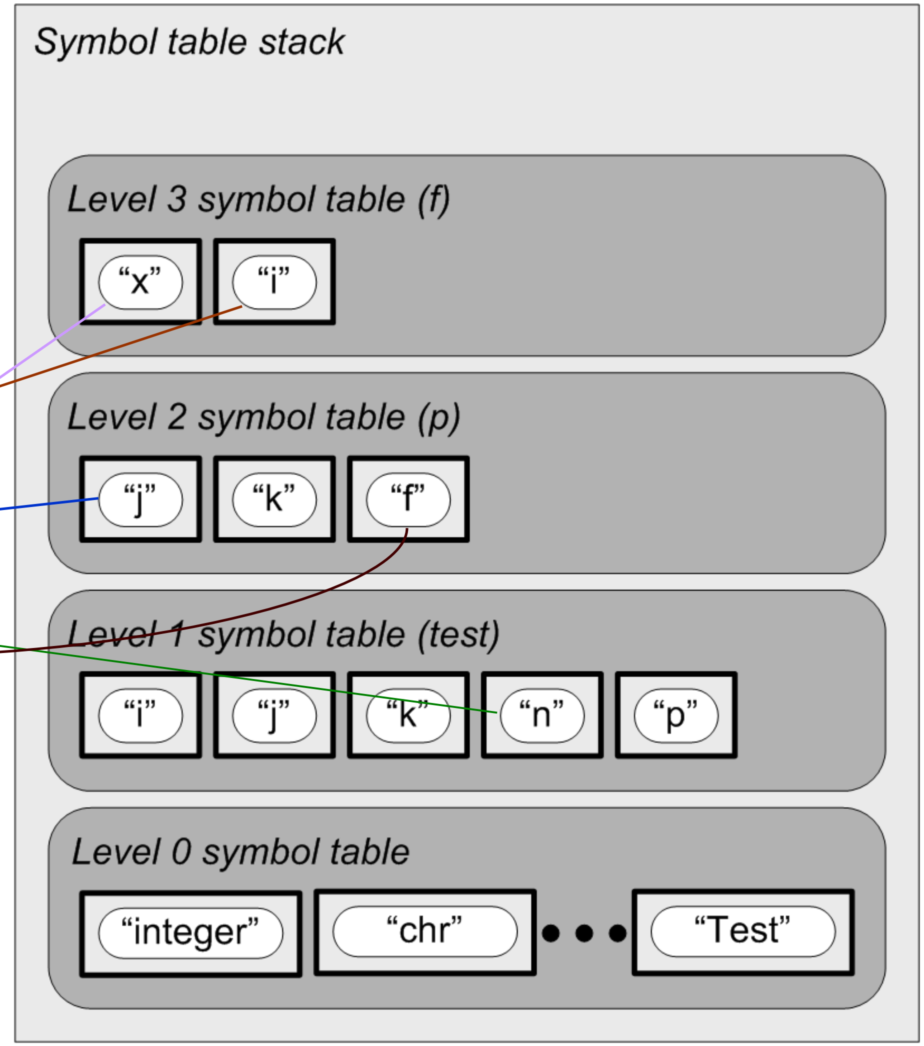
Nested Scopes and the Symbol Table Stack

```
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}.
```



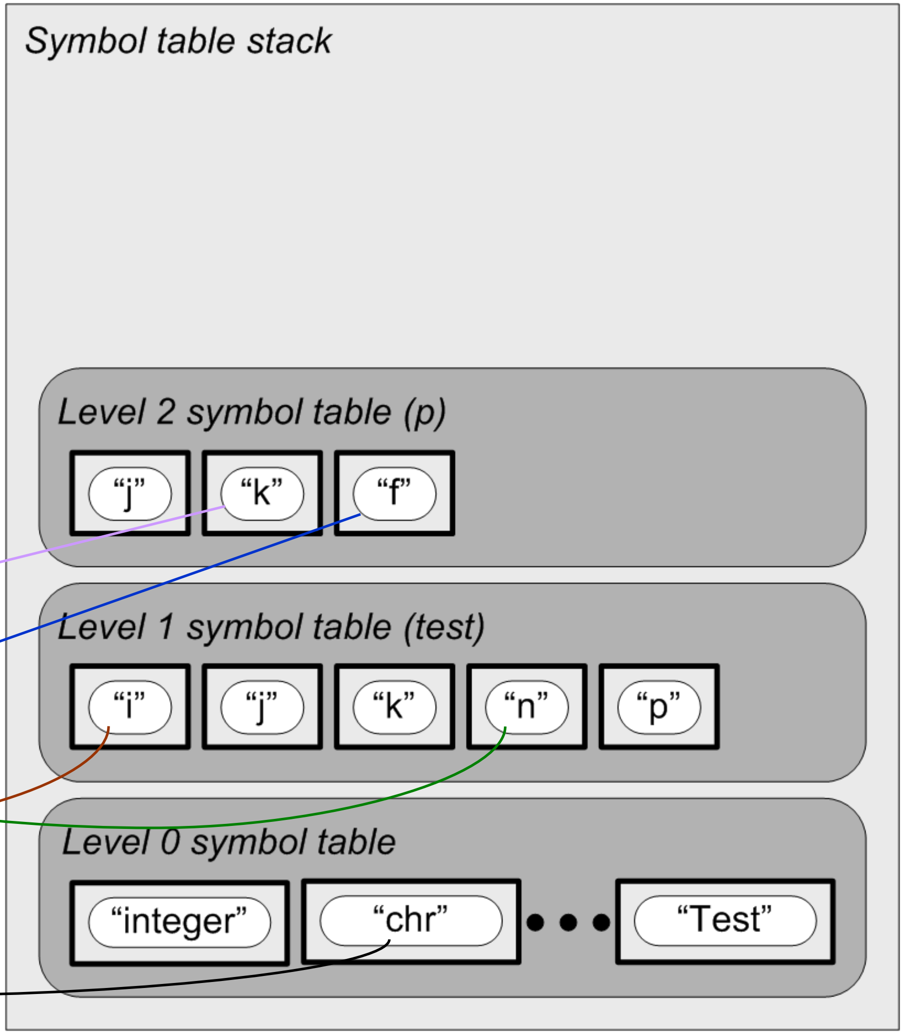
Nested Scopes and the Symbol Table Stack

```
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}.
```



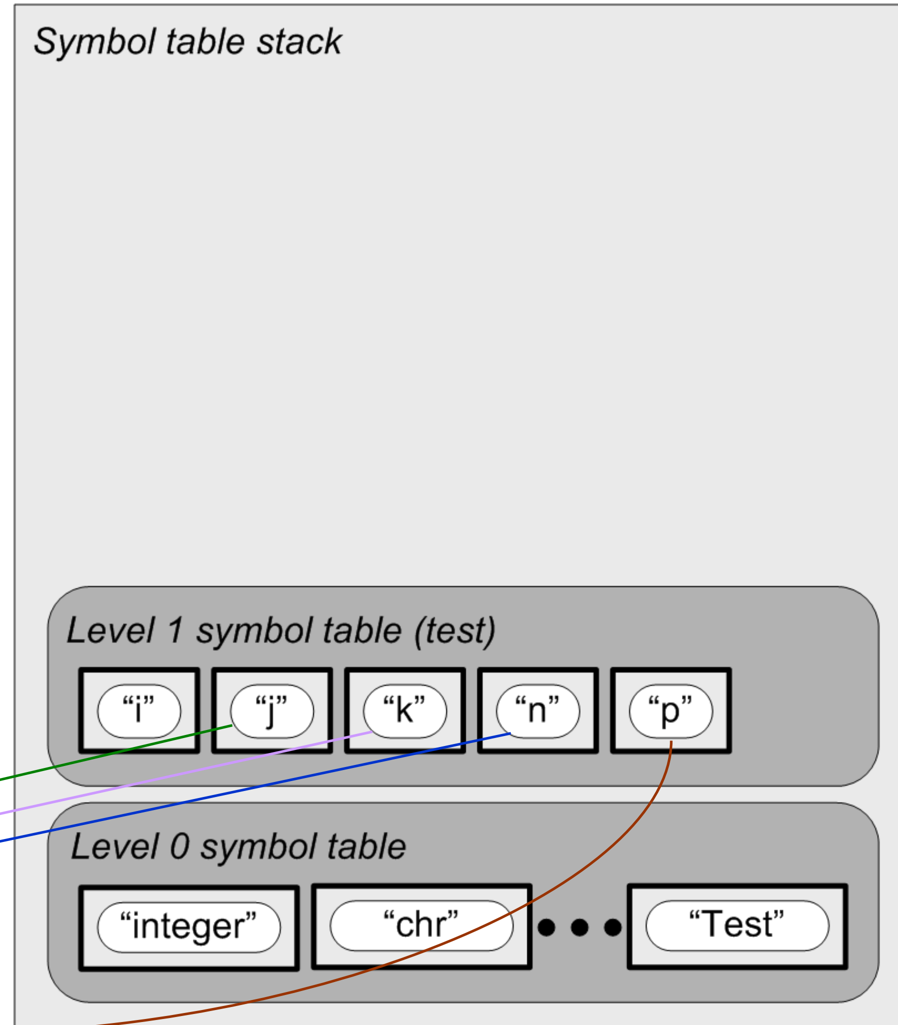
Nested Scopes and the Symbol Table Stack

```
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}.
```



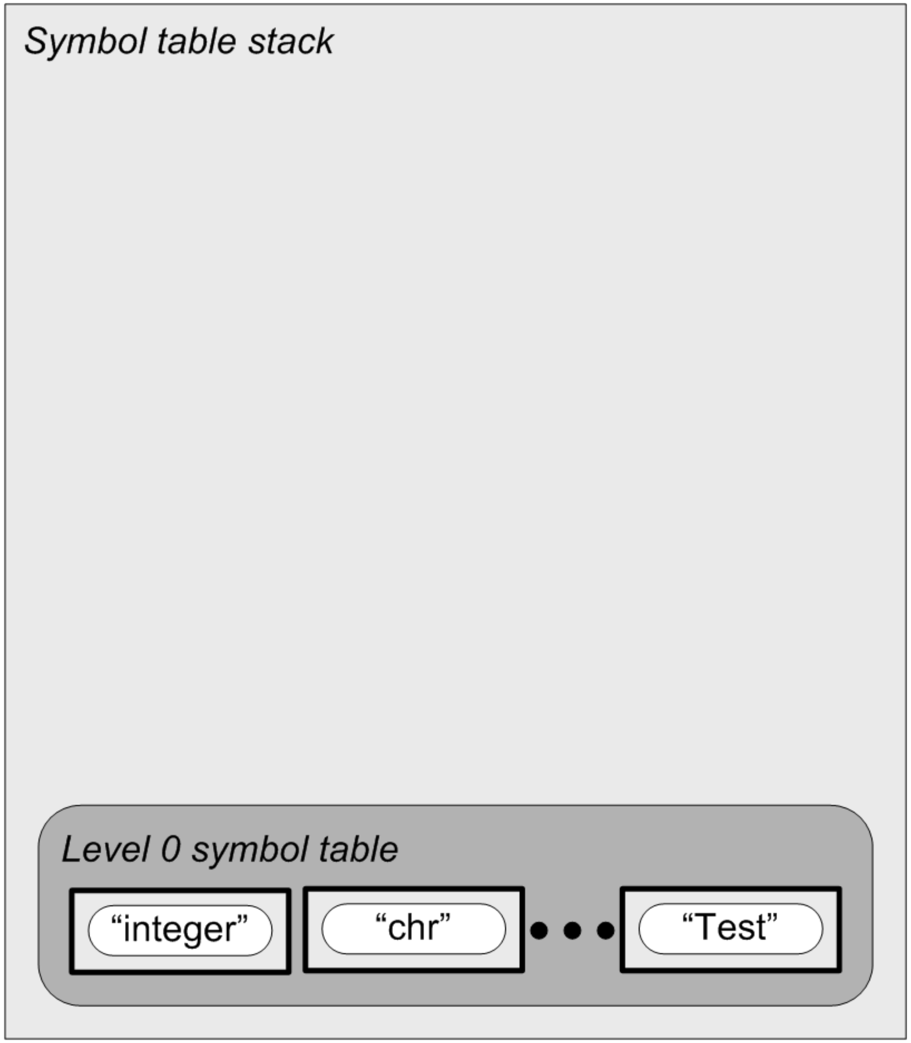
Nested Scopes and the Symbol Table Stack

```
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}.
```



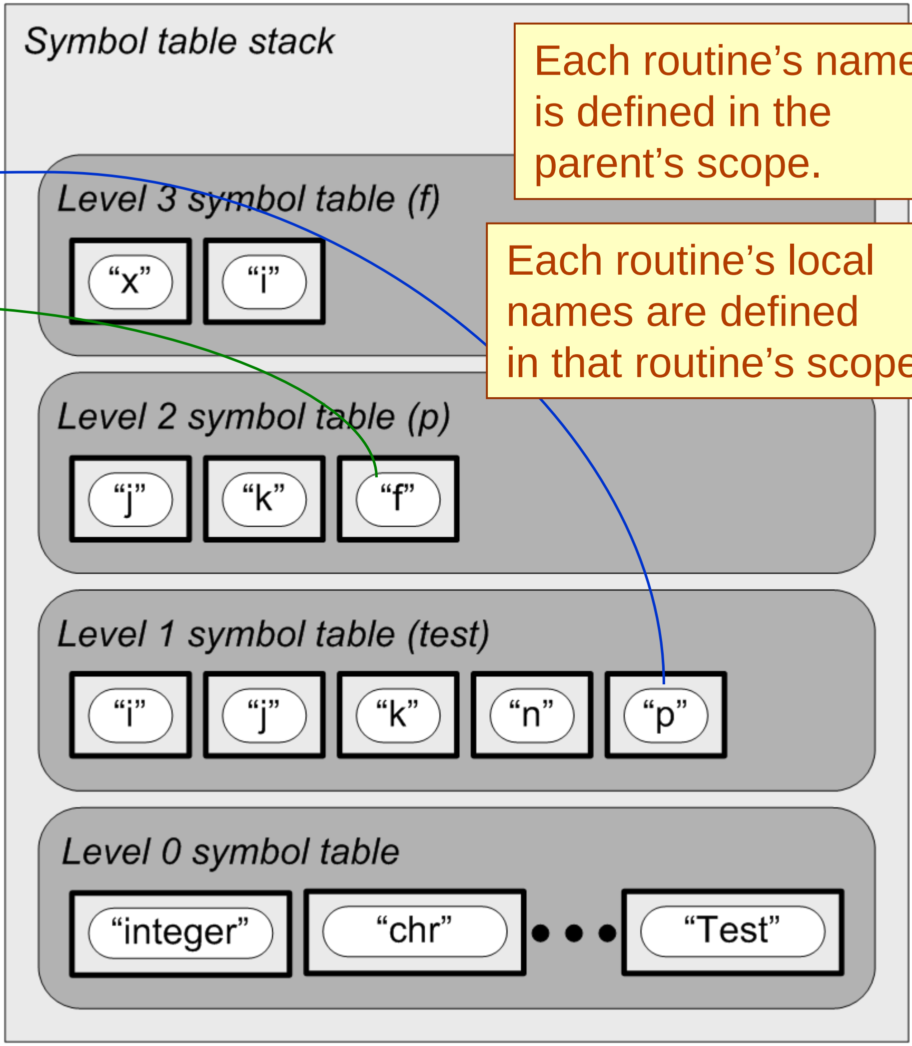
Nested Scopes and the Symbol Table Stack

```
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}.
```



Nested Scopes and the Symbol Table Stack

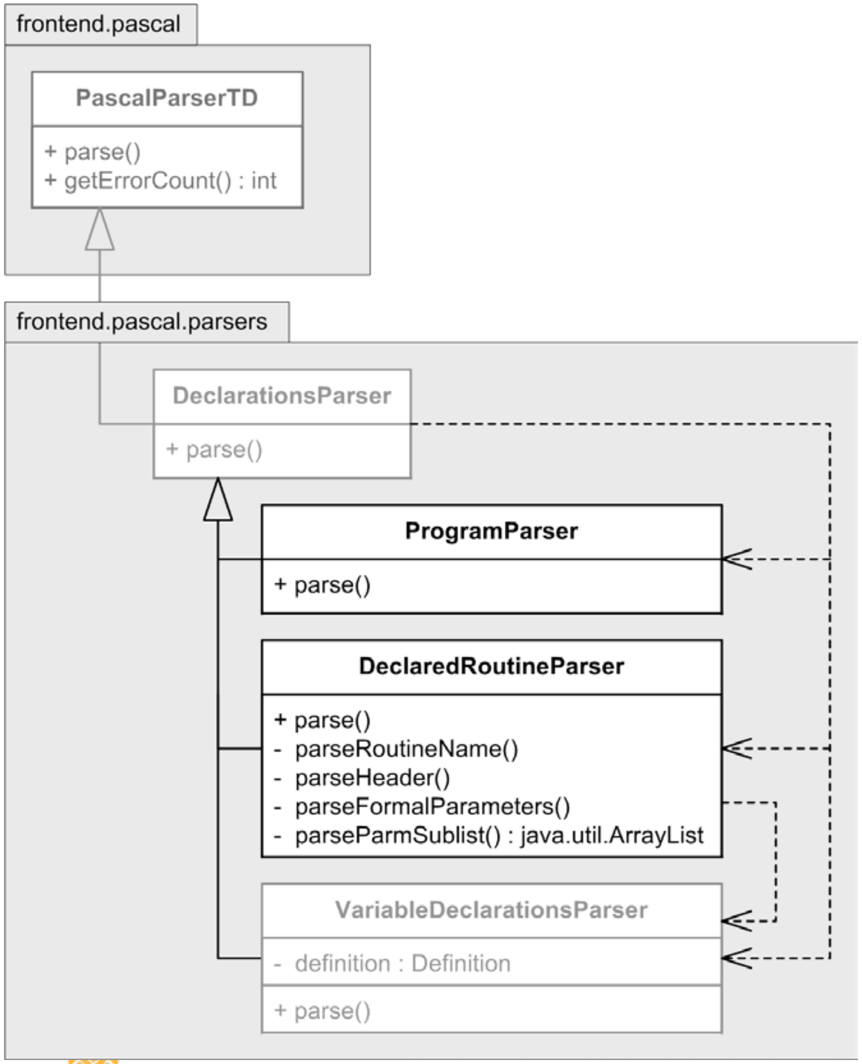
```
PROGRAM Test;

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}.
```



Each routine's name is defined in the parent's scope.

Each routine's local names are defined in that routine's scope.

Parsing Programs, Procedures, and Functions



- Classes **`ProgramParser`** and **`DeclaredRoutineParser`** are subclasses of **`DeclarationsParser`**.
- **`DeclaredRoutineParser`** depends on **`VariableDeclarationsParser`** to parse the formal parameter list.

`DeclarationsParser.parse()`

- Modify the method to look for the **`PROCEDURE`** and **`FUNCTION`** reserved words.
- Call **`DeclaredRoutineParser.parse()`** to parse the procedure or function header.
- Note: **`ProgramParser.parse()`** also calls **`DeclaredRoutineParser.parse()`** to parse the program header.

Class `DeclaredRoutineParser`

- Parse any declared (programmer-written) routine:
  - a procedure
  - a function
  - the program itself

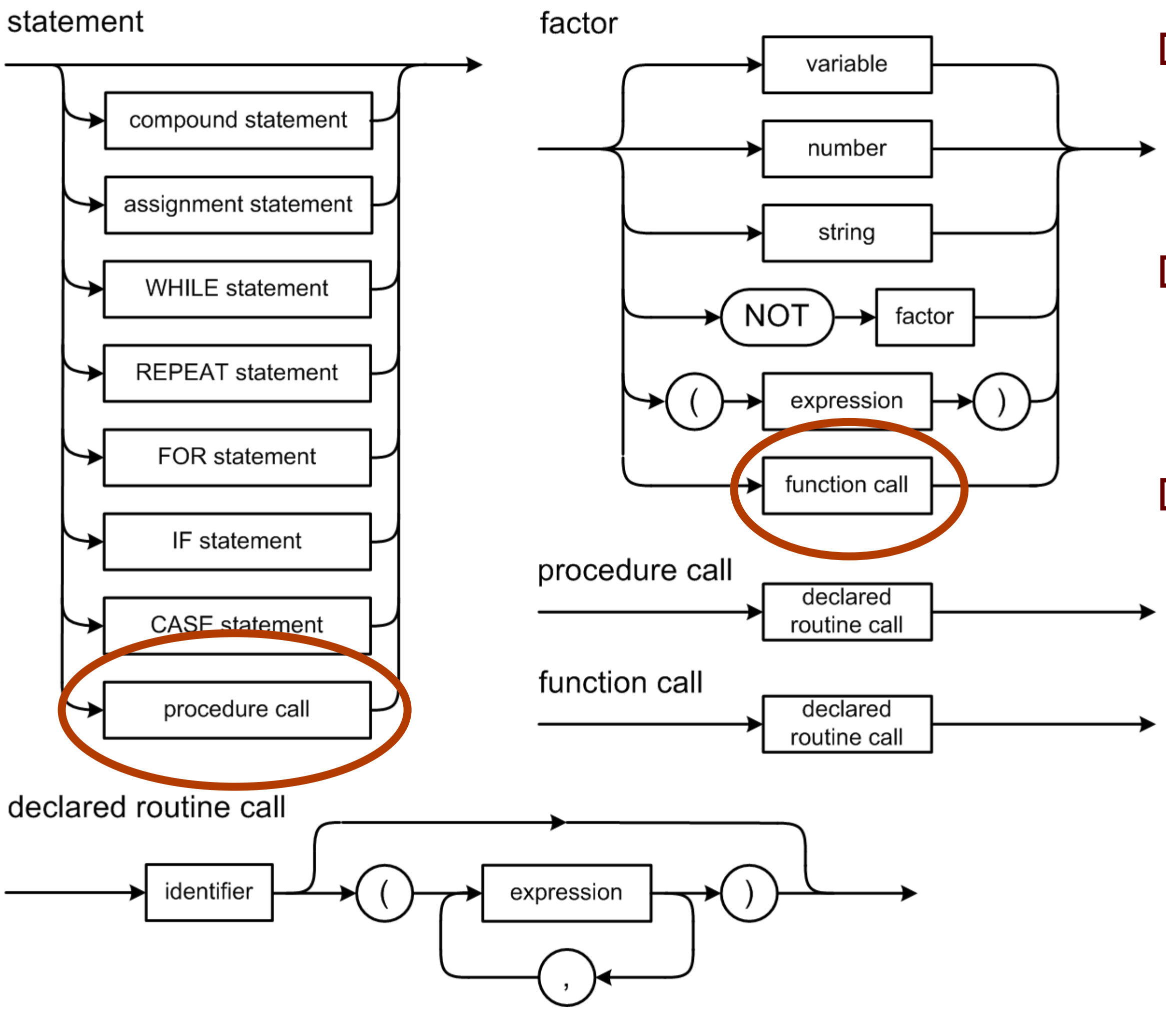
`DeclaredRoutineParser` Methods

- **`parse()`**
  - First call **`parseRoutineName()`**
  - Enter the routine name into the enclosing scope's symbol table.
  - Push a new symbol table for the routine onto the symbol table stack.
  - Pop off the symbol table for the routine when done parsing the routine.

`DeclaredRoutineParser` Methods, *cont'd*

- **`parseRoutineName()`**
- **`parseHeader()`**
- **`parseFormalParameters()`**
- **`parseParmSublist()`**
  - Call **`VariableDeclarationsParser .parseIdentifierSublist()`**
  - Enter each formal parameter into the routine's symbol table defined either as a VALUE_PARM or a VAR_PARM.

Procedure and Function Calls



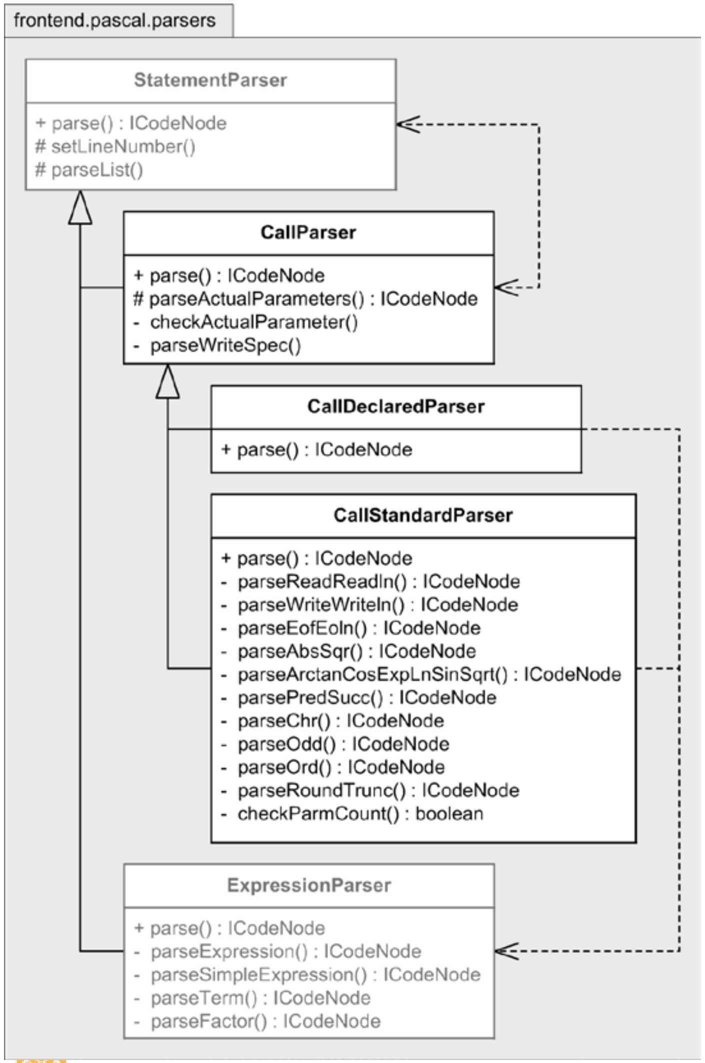
- A procedure call is a statement.
- A function call is a factor.
- If there are no actual parameters, there are also no parentheses.

Class `Predefined`

- Load the global symbol table with the predefined identifiers.
  - **`integer`**, **`real`**, **`char`**, **`boolean`**, **`true`**, **`false`**.

- Now it must also load the global symbol table with the identifiers of the predefined procedures and functions.
  - **`write`**, **`writeln`**, **`read`**, **`readln`**
  - **`abs`**, **`arctan`**, **`chr`**, **`cos`**, **`eof`**, **`eoln`**, **`exp`**, **`ln`**, **`odd`**, **`ord`**, **`pred`**, **`round`**, **`sin`**, **`sqr`**, **`sqrt`**, **`succ`**, **`trunc`**

Classes for Parsing Calls



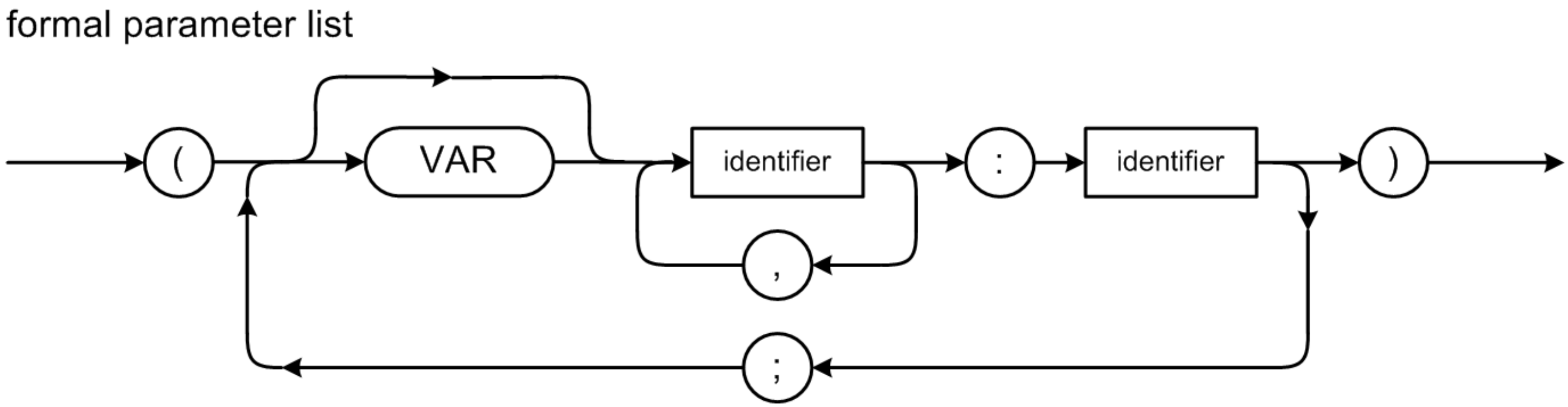
- A new statement parser subclass: **`CallParser`**.
- **`CallParser`** has two subclasses, **`CallDeclaredParser`** and **`CallStandardParser`**.
  - **`CallStandardParser`** parses calls to the standard (predefined) Pascal routines.
- Each **`parse()`** method returns the root of a parse subtree.

Class `CallParser`

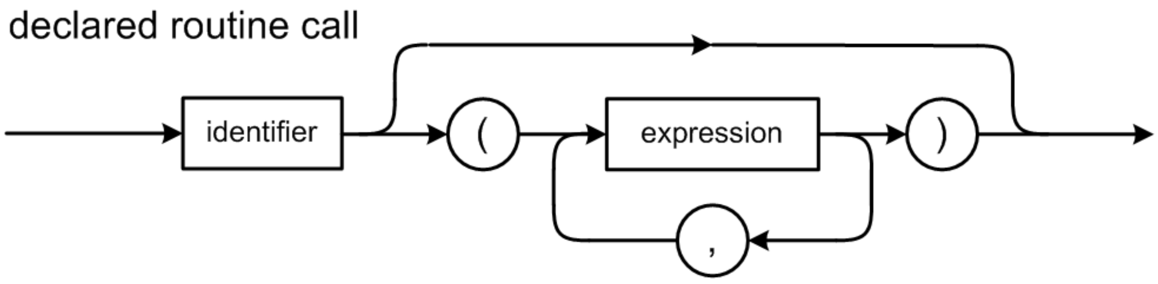
- `CallParser.parse()` calls either:
  - `CallDeclaredParser.parse()`
  - `CallStandardParser.parse()`
- Protected method `parseActualParameters()` is used by both subclasses.

Formal Parameters and Actual Parameters

- Formal parameters:
  In a procedure or function declaration:



- Actual parameters:
  In a procedure or function call:



Formal and Actual, *cont'd*

- In class **`CallParser`**,
  method **`parseActualParameters()`**
  calls **`checkActualParameter()`**
  to check each actual parameter against
  the corresponding formal parameter.

- There must be a <u>one-to-one correspondence</u>.

Formal and Actual, *cont'd*

- Value parameter (pass a copy)
  - The actual parameter can be any expression.
  - The value of the actual parameter must be assignment-compatible with the corresponding formal parameter.
- VAR parameter (pass by reference)
  - The actual parameter must be a variable
    - Can have subscripts and fields
  - The actual parameter must have the same type as the type of the corresponding formal parameter.

Formal Parameters and Actual Parameters

- Example declarations:

```
TYPE
    arr = ARRAY [1..5] OF real;
VAR
    i, m : integer;
    a : arr;
    v, y : real;
    t : boolean;

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

- Example call:

```
        proc( i, -7+m, a[m], v, y, a, t, 'r' )
```

CallParser.parseActualParameters()

- Example call to procedure **proc**:

```
proc(i, -7 + m, a[m], v, y, a, t, 'r')
```

generates the parse tree:

```
<CALL id="proc" level="1" line="81">
    <PARAMETERS>
        <VARIABLE id="i" level="1" type_id="integer" />
        <ADD type_id="integer">
            <NEGATE>
                <INTEGER_CONSTANT value="7" type_id="integer" />
            </NEGATE>
            <VARIABLE id="m" level="1" type_id="integer" />
        </ADD>
        <VARIABLE id="a" level="1" type_id="real">
            <SUBSCRIPTS type_id="real">
                <VARIABLE id="m" level="1" type_id="integer" />
            </SUBSCRIPTS>
        </VARIABLE>
        <VARIABLE id="v" level="1" type_id="real" />
        <VARIABLE id="y" level="1" type_id="real" />
        <VARIABLE id="a" level="1" type_id="arr" />
        <VARIABLE id="t" level="1" type_id="boolean" />
        <STRING_CONSTANT value="r" type_id="char" />
    </PARAMETERS>
</CALL>
```

The **CALL** node can have a **PARAMETERS** node child.

The **PARAMETERS** node has a child node for each actual parameter.

Parsing Calls to the Standard Routines

- Class **`CallStandardParser`** parses calls to the standard procedures and functions.
- These calls are handled as <u>special cases</u>.
- Method **`parse()`** calls private <u>ad hoc</u> parsing methods.

Parsing Calls to the Standard Routines

- Example: method **`parseAbsSqr()`**
  - One integer or one real actual parameter.
  - The return type is the same as the parameter type.

- Each actual argument to standard procedure **`write`** or **`writeln`** can be followed by specifiers for the field width and the number of decimal places after the point.
  - Example: **`writeln(str:10, k:12, x:20:5)`**

Pascal Syntax Checker IV

- Now we can parse <u>entire</u> Pascal programs!
  - Demo
  - Onward to interpreting Pascal programs.

Review: The Front End Parser

- Now it can parse an entire Pascal program.
- For the main program and each procedure and function:
  - Create a symbol table for the local identifiers.
  - Create a parse tree for the compound statement.
  - The symbol tables and parse trees live in the intermediate tier.
  - The symbol table entry for a program, procedure, or function name points to the corresponding symbol table and parse tree.

Review: The Front End Parser, *cont'd*

- The front end creates only <u>correct</u> symbol tables and parse trees.

- The structure of the parse tree encodes the <u>operator precedence rules</u>.

Review: The Back End Executor

- Up until now, it can only execute a main program.
  - Using only the main program's symbol table and parse tree.
- It assumes the symbol table and parse tree were correctly built by the front end.
  - No syntax or type errors.
  - The parse tree structure accurately represents the operator precedence rules.

Review: The Back End Executor, *cont'd*

- The back end executor executes the program by "walking" the parse tree and accessing the symbol table.
- The executor does not need to know what the source program looks like or even what language it was written in.

Review: Temporary Hacks

- Store values of variables calculated at run time into the variable's symbol table entry.
  - Why won't this always work?

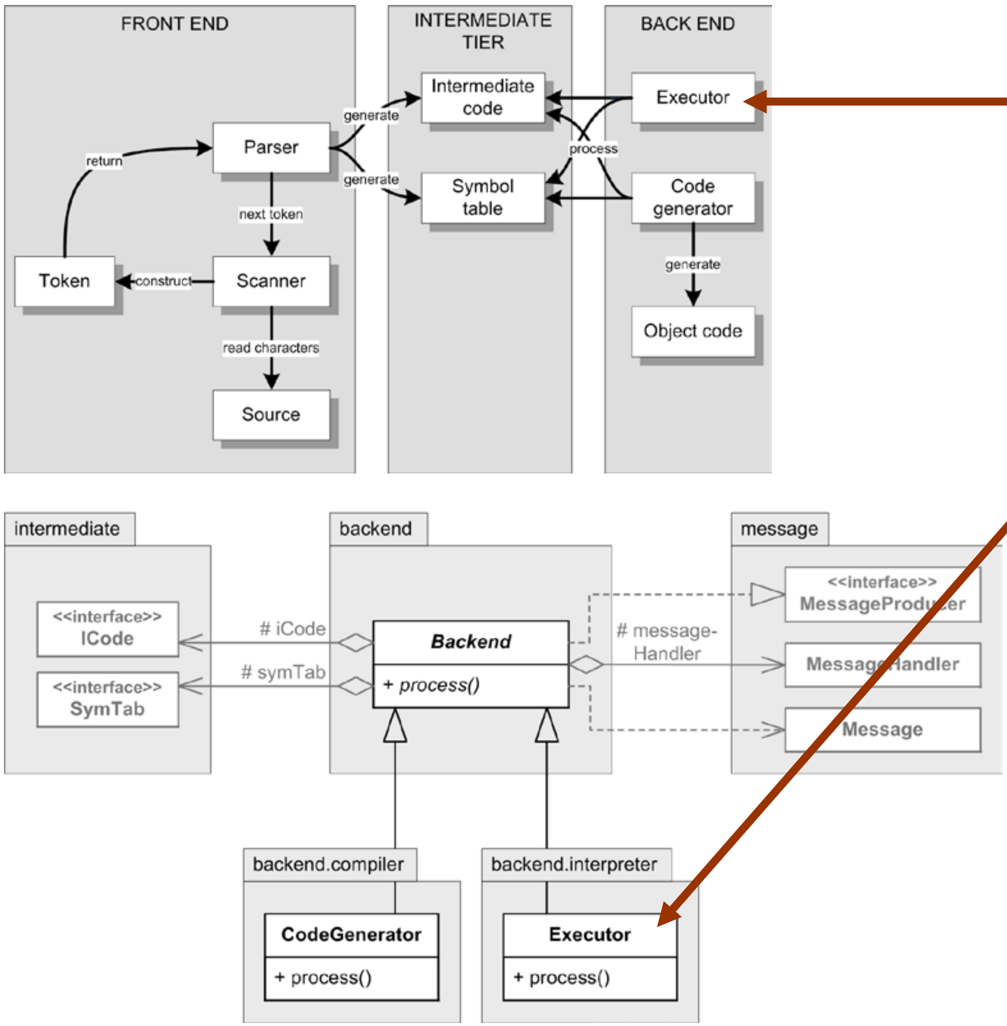
- Print out debugging information each time we assign a value to a variable.

  - We don't know how to execute a **`write`** or **`writeln`** call yet.

Temporary Hacks Removed!

- A variable is “declared” the first time it appears as the target of an assignment statement.
- We now can parse <u>variable declarations</u>.
- A variable can hold a value of any type.
- Now the parser can check for <u>assignment compatibility</u>.

Back to the Interpreter Back End



Execute entire Pascal programs!

The Interpreter

- The back end executes the source program.
- Uses the symbol tables and the parse trees built by the front end parser.
- One symbol table per main program, procedure, and function for that routine’s local identifiers.
- The symbol table is pointed to by the routine name’s symbol table entry.

The Interpreter, *cont'd*

- One parse tree per main program, procedure, and function.
  - For that routine's compound statement.
  - Pointed to by the symbol table entry for the routine's name.
- <u>The interpreter never sees the original source program.</u>
  - Only the symbol tables and parse trees.

Compile Time vs. Run Time

- Compile time:
  - The time during which the front end is <u>parsing</u> the source program and building the symbol tables and the parse trees.
  - It's called "compile time" whether the back end is a compiler or an interpreter.
  - AKA translation time

Compile Time vs. Run Time

- Run time:
  - The time during which the back end interpreter is <u>executing</u> the source program.
  - <u>English grammar alert!</u>
  - **Noun:** run time
    Example: The program executes during run time.
  - **Adjective:** runtime (or run-time)
    Example: Division by zero is a runtime error.

Runtime Memory Management

- The interpreter must <u>manage the memory</u> that the source program uses during run time.
- Up until now, we've used the hack of storing values computed during run time into the symbol table.
- Why is this a bad idea?
  - This will <u>fail miserably</u> if the source program has recursive procedure and function calls.

Symbol Table Stack vs. Runtime Stack

- The front end parser builds symbol tables and manages the symbol table stack as it parses the source program.
  - The parser pushes and pops symbol tables as it enters and exits nested scopes.
- The back end executor manages the runtime stack as it executes the source program.
  - The executor pushes and pops activation records as it calls and returns from procedures and functions.

These are similar concepts, so don’t confuse them!

Runtime Activation Records

- An activation record (AKA stack frame) maintains information about the <u>currently executing routine</u>
  - a procedure
  - a function
  - the main program itself

Runtime Activation Records

- In particular, an activation record contains the routine's local memory.
  - values of local variables
  - values of formal parameters
- This local memory is a memory map.
  - **Key**: The name of the local variable or formal parameter.
  - **Value**: The current value of the variable or parameter.

Local memory is a hash table!

Runtime Activation Records, *cont'd*

In this example, the names of the routines indicate their nesting levels.

```
PROGRAM main1;

PROCEDURE proc2a;

  PROCEDURE proc3;
    BEGIN
      ...
    END;

  BEGIN {proc2a}
    proc3;
  END;

PROCEDURE proc2b;
  BEGIN
    proc2a;
  END;

BEGIN {main1}
  proc2b;
END.
```

AR: proc3
AR: proc2a
AR: proc2b
AR: main1

RUNTIME STACK

main1 → proc2b → proc2a → proc3

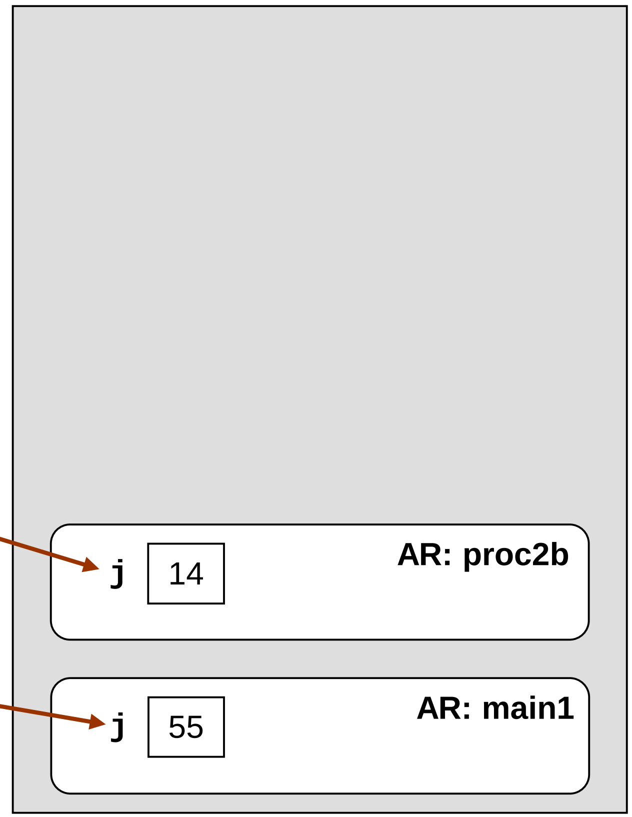
Call a routine: Push its activation record onto the runtime stack.

Return from a routine: Pop off its activation record.

Runtime Access to Local Variables

```
PROGRAM main1;
VAR j : integer;

PROCEDURE proc2b;
  VAR j : integer;

  BEGIN
    j := 14;
  END;

BEGIN {main1}
  j := 55;
  proc2b;
END.
```

main1 → proc2b

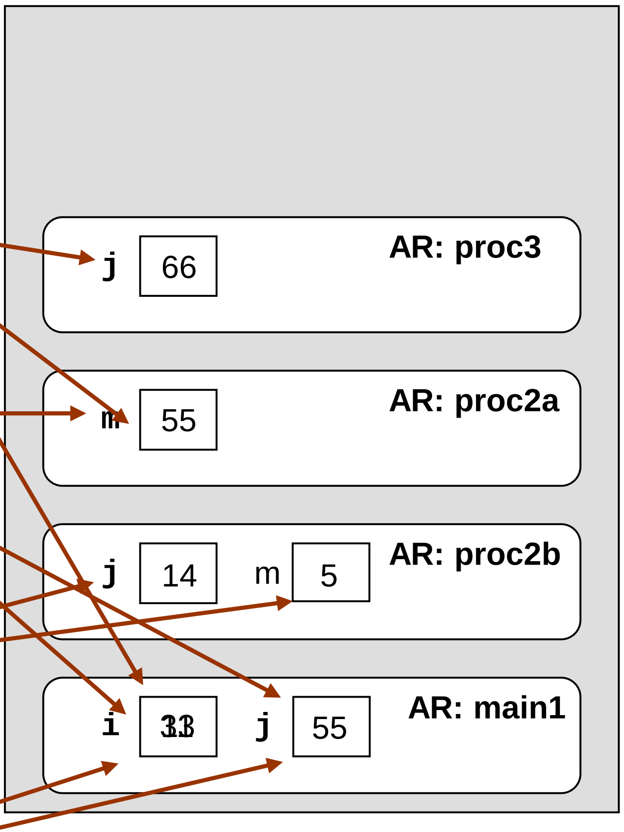


RUNTIME STACK

Accessing **local values** is simple, because the currently executing routine's activation record is on top of the runtime stack.

Runtime Access to Nonlocal Variables

```
PROGRAM main1;
VAR i, j : integer;

  PROCEDURE proc2a;
    VAR m : integer;

    PROCEDURE proc3;
        VAR j : integer
        BEGIN
          j := i + m;
        END;

    BEGIN {proc2a}
      i := 11;
      m := j;
      proc3;
    END;

  PROCEDURE proc2b;
    VAR j, m : integer;
    BEGIN
      j := 14;
      m := 5;
      proc2a;
    END;

  BEGIN {main1}
    i := 33;
    j := 55;
    proc2b;
  END.
```



RUNTIME STACK

main1 → proc2b → proc2a → proc3

- Each parse tree node for a <u>variable</u> contains the variable's <u>symbol table entry</u> as its VALUE attribute.
  - Each symbol table entry has the variable's <u>nesting level</u> ***n***.
- To access the value of a variable at nesting level ***n***, the value must come from the <u>topmost activation record at level *n*</u>.
  - Search the runtime stack from <u>top to bottom</u> for the topmost activation record at level ***n***.