CMPE 152: Compiler Design

October 3 Lab

Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak

www.cs.sjsu.edu/~mak

Runtime Memory Management

- The interpreter must manage the memory 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 fail miserably 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 currently executing routine
  - a procedure
  - a function
  - the main program itself

Runtime Activation Records

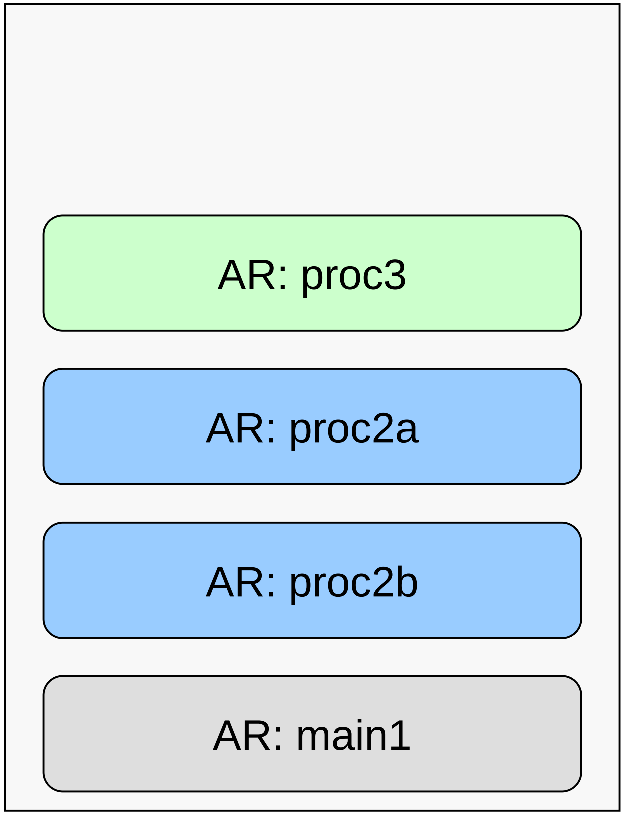
- In particular, an activation record contains the routine's local memory.
  - values of local variables
  - values of formal parameters
- This local memory is a memory map.
  - **Key**: The name of the local variable or formal parameter.
  - **Value**: The current value of the variable or parameter.

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.
```



RUNTIME STACK

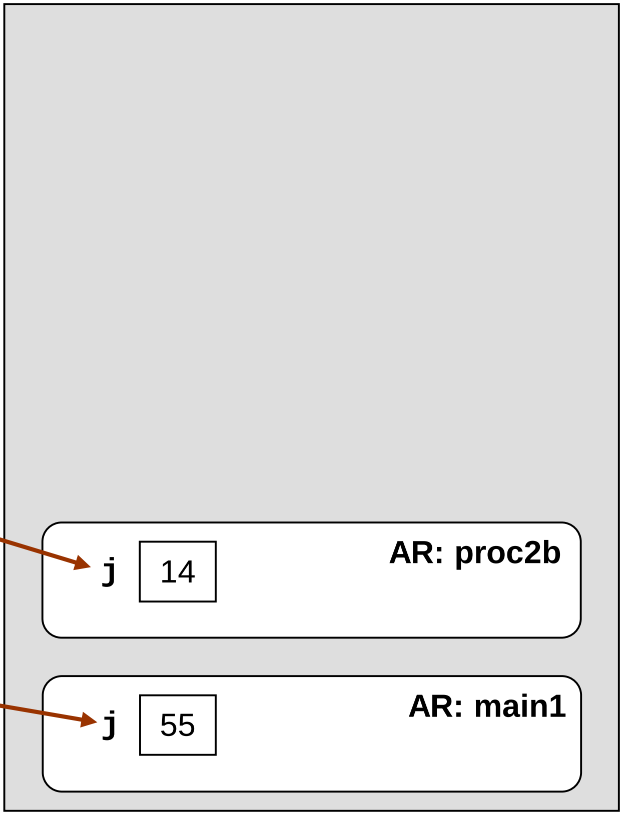
Call a routine: Push its activation record onto the runtime stack.

Return from a routine: Pop off its activation record.

main1 → proc2b → proc2a → proc3

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.
```

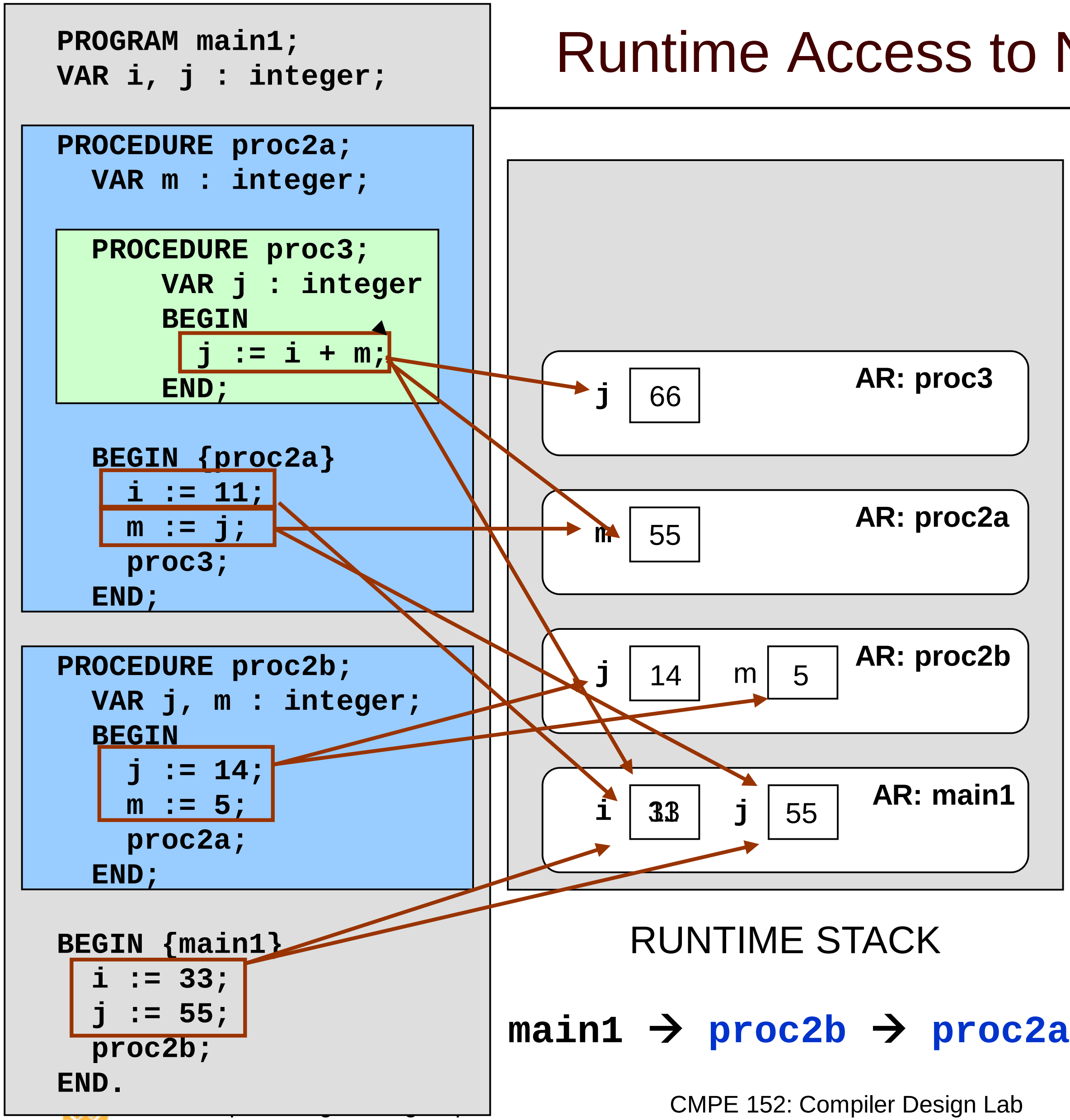


main1 → proc2b

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

The Runtime Display

- A vector called the runtime display makes it easier to access nonlocal values.
- This has nothing to do with video displays!

The Runtime Display, *cont'd*

- Element *n* of the display always points to the topmost activation record at scope nesting level *n* on the runtime stack.

- The display must be updated as activation records are pushed onto and popped off the runtime stack.

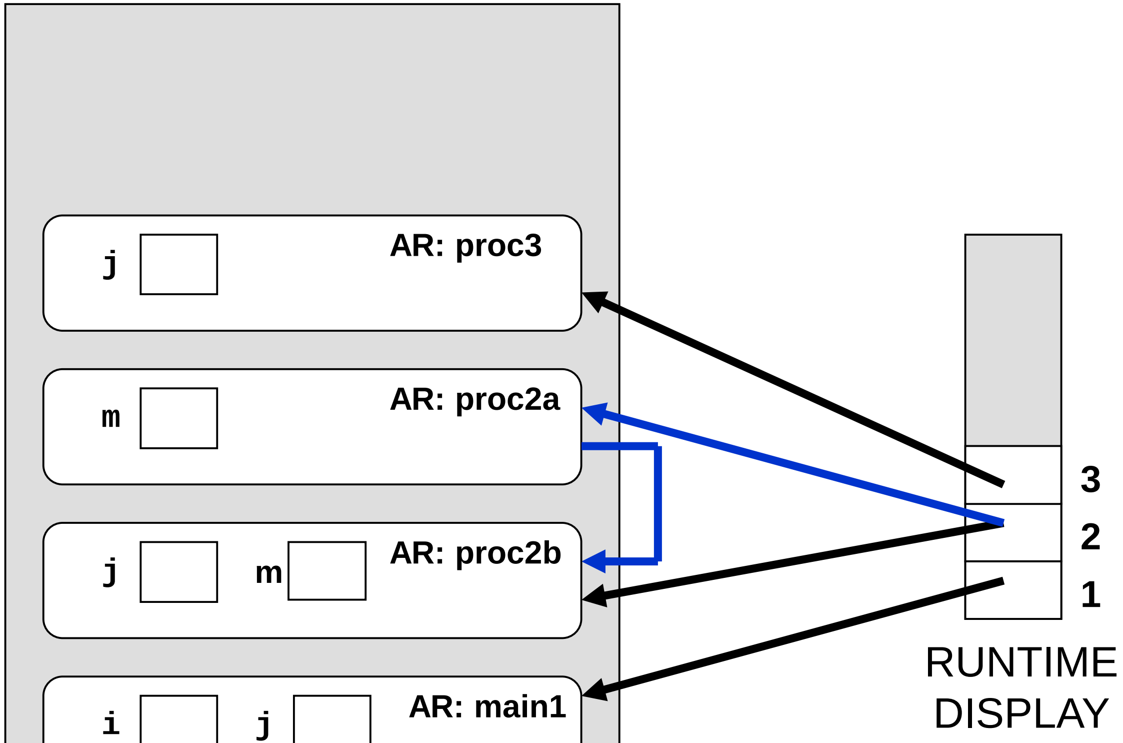
The Runtime Display, *cont'd*

- Whenever a new activation record at level *n* is pushed onto the stack, it links to the previous topmost activation record at level *n*.

- This link helps to restore the runtime stack as activation records are popped off when returning from procedures and functions.

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.
```

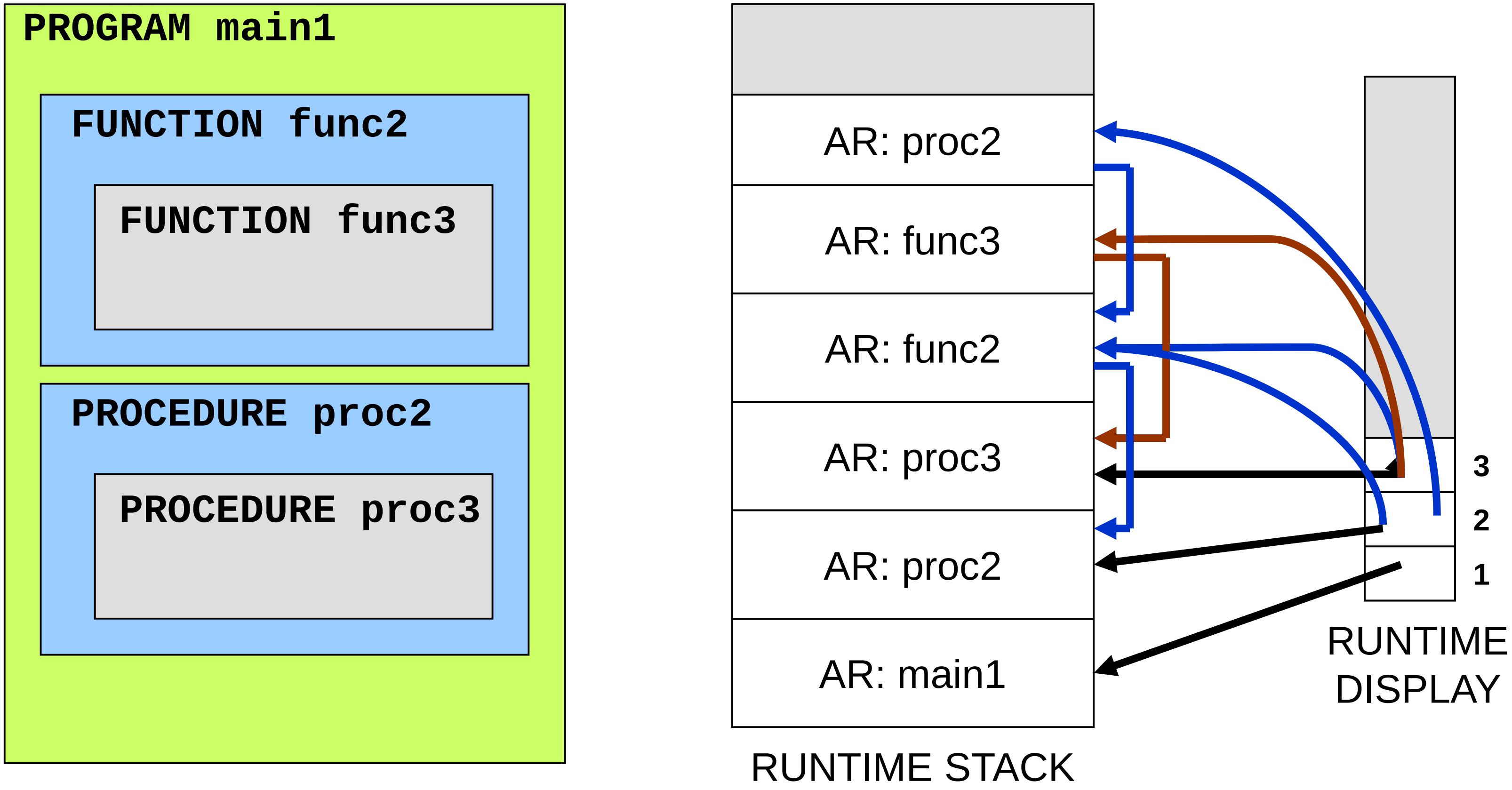


RUNTIME STACK

The runtime display allows faster access to **nonlocal** values.

main1 → proc2b → proc2a → proc3

Recursive Calls



main1 → proc2 → proc3 → func2 → func3 → proc2

Assignment #4: Complex Type

- Add a built-in complex data type to Pascal.
  - Add the type to the global symbol table.
  - Implement as a record type with real fields **re** and **im**.

- Declare complex numbers:

```
VAR
    x, y, z : complex;
```

- Assign values to them:

```
BEGIN
    z.re := 3.14;
    z.im := -8.2;
    ...
```

Assignment #4, *cont'd*

- Do complex arithmetic:

```
z := x + y;
```

- The backend executor does all the work of evaluating complex expressions. Use the following rules:
  - $(a + bi) + (c + di) = (a + c) + (b + d)i$
  - $(a + bi) - (c + di) = (a - c) + (b - d)i$
  - $(a + bi)(c + di) = (ac - bd) + (ad + bc)i$
  - $\frac{a+bi}{c+di} = \frac{(ac+bd)+(bc-ad)i}{c^2+d^2}$

Assignment #4, *cont'd*

- More details to come about this assignment.
- For now, start with Chapter10cpp.zip.
- Examine **`wci::intermediate::symtabimpl::Predefined`** to see how the built-in types like **`integer`** and **`real`** are defined.
- Examine **`wci::frontend::pascal::parsers::RecordTypeParser`** to see what information is entered into the symbol table for a **`record`** type.