Implementing Subprograms & Blocks

Semantics of Subprogram Calls and Returns

Subprogram Calls

- Pass parameters using parameter passing methods.
- Allocate storage space for local variables.
- Arrange to access nonlocal variables.
- Save the execution status of the caller.
- Save the return address.
- Transfer control to the callee.

Subprogram Returns

- Copy back using parameter passing methods if needed.
- Deallocate the storage used for locals.
- Restore the execution status of the caller.
- Return control to the caller.

Info Needed for Subprogram Calls and Returns

- Certain information must be available:
  - The **code** for the subprogram
  - The **state** while the body of the subprogram is executing.
    - Instruction part
      - A pointer to the instruction to be executed after the subprogram returns (Return address)
    - Environment part
      - The values of locals, nonlocals and parameters.

Info Needed for Subprogram Calls and Returns

- The code for the subprogram
  - **Fixed**
- The state while the body of the subprogram is executing.
  - **Changing**
  - **Different calls to the same subprogram will have different states!**

Activation Record (AR)

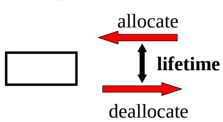
- The state info need for a subprogram call and return is stored in an **activation record (AR)**.
- In an activation record:
  - Instruction part
    - A pointer to the instruction to be executed after the subprogram return (Return address)
  - Environment part
    - The values of locals, nonlocals and parameters.

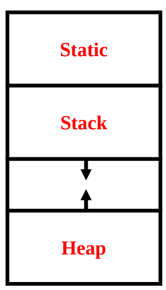
Subprogram, Call, Activation & Activation Record

- A subprogram
- A **call** to the subprogram
- An **activation** of the call
- An **activation record** for the activation

Storage for Activation Records

- Where do we allocate storage for the activation records?
  - It depends!





Two Types of Languages

- **FORTRAN-like languages**
  - No recursive subprograms
  - Static local variables
  - No nonlocal variables (Flat block structure)
- **Algol-like languages**
  - Recursive subprograms
  - Stack-dynamic local variables
  - Nonlocal variables (Nested block structure)

Storage for Activation Records

- FORTRAN-like languages
  - From **static** storage
- Algol-like languages
  - From **stack** storage

Implementing Subprogram Calls and Returns

- It depends on the type of the language!

Two Types of Languages

- FORTRAN-like languages
  - No recursive subprograms
  - Static local variables
  - No nonlocal variables (Flat block structure)
- Algol-like languages
  - Recursive subprograms
  - Stack-dynamic local variables
  - Nonlocal variables (Nested block structure)

Implementing Subprogram Calls and Returns

- FORTRAN-like languages
  - **Relatively simple!**
- Algol-like languages
  - **More difficult!**

1. Implementing FORTRAN77-like Subprograms

Call Semantics

1. Save the execution status of the caller.
2. Carry out the parameter-passing process.
3. Pass the return address.
4. Transfer control to the callee.

Return Semantics

1. If pass-by-value-result parameters are used, move the current values of those parameters to their corresponding actual parameters.
2. If it is a function, move the functional value to a place the caller can get it.
3. Restore the execution status of the caller.
4. Transfer control back to the caller.

Required Storage

- Status information of the caller
- Parameters
- Return address
- Functional value (if it is a function)
- Local variables
- The subprogram code

Activation Record

- The format or layout of the noncode part is called an **activation record**.

Functional Value
Local variables
Parameters
Return Address

Activation Record Instance

- An **activation record instance** is
  - A concrete example of an activation record.
  - The collection of data for a particular subprogram activation (call).

A subprogram call → **An activation record instance**

Static Allocation for Activation Record

- A FORTRAN 77 subprogram can have **only one activation record instance** at any given time!
- Why?
  - No recursive subprogram!

Static Allocation for Activation Record

- Statically allocate storage for Activation Record.
- Use it for each activation record instance.

Functional Value
Local variables
Parameters
Return Address

← **Static Storage**

Example: Implementing A FORTRAN 77 Subprogram

- A main program `MAIN`
- Three subprograms `A, B & C`
- The code and activation records:
  - See Figure 10.2 (p. 400)

2. Implementing ALGOL-like Subprograms

- This is more complicated than implementing FORTRAN 77-like subprograms.
- Why?
  - Local variables are often dynamically allocated.
  - Recursion must be supported.
  - Static scoping must be supported.
  - More parameter passing methods

Activation Record

- A typical activation record for an ALGOL-like language:

Local variables
Parameters
Dynamic Links
Static Links
Return Address

Activation Record

- The activation record format is static, but its size may be dynamic.
- An activation record instance **must be created dynamically** when a subprogram is called.

Dynamic and Static Links

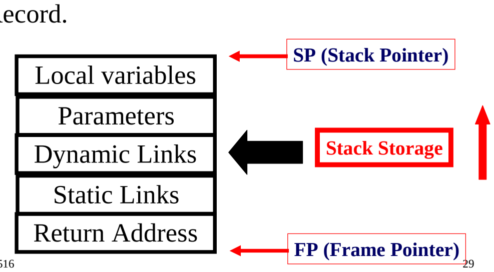
- The **dynamic link (DL)**
  - points to the top of an instance of the activation record of the caller.
- The **static link (SL)**
  - points to the bottom of the activation record instance of an activation of the static parent (to be used for access to nonlocal variables).

Activation Record

- An Algol-like subprogram can have **more than one activation record instance** at any given time!
- Why?
  - Recursive subprogram!

Dynamic Allocation for Activation Record

- Dynamically allocate storage for Activation Record.



Example: Activation Record

```
procedure sub(var total: real; part: integer);
var list: array[1..2] of integer;
    sum: real;
begin
…
end
```

Example: Activation Record

`sum`	Local
`list[3]`	Local
`list[2]`	Local
`list[1]`	Local
`part`	Parameter
`total`	Parameter
	DL
	SL
	RA

(1) Without Recursion and Nonlocal References

Example

```
void fun1(int x) {
   int y;
   … <----------------------------2
   fun3(y);
   …
}
void fun2(float r) {
   int s, t;
   ... <--------------------------1
   fun1(s);
   ...
}
void fun3(int q) {
   ... <--------------------------3
   ...
}
Void main() {
   float p;
   fun2(p);
}
```

Call sequence:
`main` calls `fun2`
`fun2` calls `fun1`
`fun1` calls `fun3`

Stack contents:

See FIGURE 10.5 (p. 405)

Dynamic Chain

- A **dynamic link** is a pointer to the AR of the caller.
  - Why?
- A **dynamic chain** is a sequence of dynamic links.
- The dynamic chain is a list of all AR's on the stack, i.e., all active subprograms.

Local Variables

- Local variables can be accessed by their offset from the beginning of the activation record.
  - This offset is called the **local_offset**.
- The local_offset of a local variable can be determined **at compile time** by the compiler.

(2) With Recursion

Example: Recursive Functions

```
int factorial(int n) {
     <----------------------------1
   if (n <= 1)
     return 1;
   else return (n * factorial(n - 1));
     <----------------------------2
 }
 void main() {
   int value;
   value = factorial(3);
     <----------------------------3
 }
```

Stack contents:

See FIGUREs 10.7 and 10.8 (p. 407 and p. 408)

(3) With Nonlocal References

Rules for Nonlocal References

1. Static scoping rule
2. Dynamic scoping rule

The Scope Rule

- The **scope rule** of a programming language determines
  - How a particular occurrence of a name (variable) is associated with a variable.
- Given an applied (use, reference) occurrence of a variable x, what is the binding (defining, declaration) occurrence of the variable x?

The Static Scoping Rule

- Based on program text.
- Just by examining the program text, we can determine which binding occurrence correspond to a given applied occurrence.
- The binding between applied occurrences and binding occurrences is FIXED, not changing throughout the program's execution.

The Static Scoping Rule

- Search declarations, first locally, then in increasingly larger **enclosing** scopes, until one is found for the given name.
- Find the **innermost enclosing** block containing the applied occurrence and a binding occurrence.

Static-Scoped Languages

- A subprogram is callable only when all of its static ancestor program units are active!
- In a given subprogram, only variables declared in the static ancestor scopes are visible and can be accessed.
- Activation record instances of all of the static ancestors are guaranteed to exist on the stack.

CS516 43

Nonlocal References with Static Scoping Rule

- Observation:
  - All variables that can be nonlocally accessed reside in some activation record instance in the stack.
- The process of locating a nonlocal reference:
  1. **Find the correct activation record instance in which the variable is allocated.**
  2. **Use the local offset within that activation record instance to access it.**

CS516 44

How to Find the Correct Activation Record Instance?

- Find the innermost enclosing block containing the applied occurrence and a binding occurrence.

CS516 45

Implementing Nonlocal References with Static Scoping Rule

- Using **static chains**
- Using **display**

CS516 46

1. Static Chain

- The **static link** in an activation record instance for a subprogram S points to an activation record instances of S's static parent (enclosing subprogram).
  - The **most recent ARI** of the static parent!

CS516 47

Static Chain

- A **static chain** is a chain of static links.
- The static chain from an activation record instance for a subprogram S links all the static ancestors of S.

CS516 48

How to Find the Correct Activation Record Instance Using Static Chain?

- To find the declaration for a reference to a nonlocal variable?
  - Search the static chain until the activation record instance that contains the variable (as a local variable) is found!
- How many static links to be followed?
  - Can be determined at compile time!

Static Depth of A Subprogram

- Given a subprogram S,
- The **static_depth** of S is an integer associated with the subprogram:
  - How deeply it is nested in the outmost program!
  - 0 (the outmost), 1, 2, …
  - Also called **SNL** **(Static Nesting Level)**

Example: Static Depth

```
program A;
var x: int;
   procedure B;
      procedure C;
       …
       x:=x+1;
       …
      end;{C}
      …
      x:=x+1;
      …
   end;{B}
   …
   x:=x+1;
   …
end;{A}
```

A ----- static_depth = 0

B ----- static_depth = 1

C ----- static_depth = 2

Nesting Depth of A Nonlocal Reference

- Given a nonlocal reference to a variable X,
- The **nesting_depth** or **chain_offset** of the nonlocal reference is

  (The static_depth of the the subprogram containing the reference to X)

  MINUS

  (The static_depth of the the subprogram containing the declaration for X)
  - Also called **SD** **(Static Distance)**

Example: Nesting Depth

```
program A;
var x: int;
   procedure B;
      procedure C;
       …
       x:=x+1;
       …
      end;{C}
      …
      x:=x+1;
      …
   end;{B}
   …
   x:=x+1;
   …
end;{A}
```

A ----- static_depth = 0
B ----- static_depth = 1
C ----- static_depth = 2

SD Nesting depth of X in C: 2

SD Nesting depth of X in B: 1

SD Nesting depth of X in A: 0

How to Access Nonlocal Variables Using Static Chain?

- A reference to a nonlocal variable X can be represented by the pair **(chain_offset, local_offset)** where
  - **chain_offset** = The number of static links to the correct ARI.
  - **local_offset** = The offset from the beginning of the AR of the subprogram containing the declaration for X.

Example: Nonlocal Variable Access Using Static Chain

```
program A;
var x: int;
    procedure B;
        procedure C;
         …
         x:=x+1;
         …
        end;{C}
        …
        x:=x+1;
        …
    end;{B}
    …
    x:=x+1;
    …
end;{A}
```

A ----- static_depth = 0
B ----- static_depth = 1
C ----- static_depth = 2

Nesting depth of X in C: 2
Nesting depth of X in B: 1
Nesting depth of X in A: 0

Reference to X in C: (2, local-offset)

Reference to X in B: (1, local-offset)

Reference to X in A: (0, local-offset)

Example

```
program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
      A := B + C;  <-----------------------1
      end;  { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
        SUB1;
        E := B + A:   <--------------------2
        end; { SUB3 }
      begin { SUB2 }
      SUB3;
      A := D + E;  <-----------------------3
      end; { SUB2 }
    begin { BIGSUB }
    SUB2(7);
    end; { BIGSUB }
  begin
  BIGSUB;
  end. { MAIN_2 }
```

Call sequence:

MAIN_2 calls BIGSUB
BIGSUB calls SUB2
SUB2 calls SUB3
SUB3 calls SUB1

Example

```
program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
      A := B + C;  <-----------------------1
      end;  { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
        SUB1;
        E := B + A:   <--------------------2
        end; { SUB3 }
      begin { SUB2 }
      SUB3;
      A := D + E;  <-----------------------3
      end; { SUB2 }
    begin { BIGSUB }
    SUB2(7);
    end; { BIGSUB }
  begin
  BIGSUB;
  end. { MAIN_2 }
```

Stack contents
at position 1:

See FIGURE 10.9 (p. 414)

Example

```
program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
      A := B + C;  <-----------------------1
      end;  { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
        SUB1;
        E := B + A:   <--------------------2
        end; { SUB3 }
      begin { SUB2 }
      SUB3;
      A := D + E;  <-----------------------3
      end; { SUB2 }
    begin { BIGSUB }
    SUB2(7);
    end; { BIGSUB }
  begin
  BIGSUB;
  end. { MAIN_2 }
```

Nonlocal references:

At position 1 in SUB1:
- A - (0, 3)
- B - (1, 4)
- C - (1, 5)

At position 2 in SUB3:
- E - (0, 4)
- B - (1, 4)
- A - (2, 3)

At position 3 in SUB2:
- A - (1, 3)
- D - an error
- E - (0, 5)

QUIZ: Static Chain

```
program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
      A := B + C;  <-----------------------1
      end;  { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
        SUB1;
        E := B + A:   <--------------------2
        end; { SUB3 }
      begin { SUB2 }
      SUB3;
      A := D + E;  <-----------------------3
      end; { SUB2 }
    begin { BIGSUB }
    SUB2(7);
    end; { BIGSUB }
  begin
  BIGSUB;
  end. { MAIN_2 }
```

Stack contents?
(1) At position 2
(2) At position 3

How to Maintain the Static Chain?

- During program execution!
- At a subprogram call:
  - **The static link (SL) must point to the most recent ARI of the static parent.**

Static Chain - Maintenance

- **Method 1:**
  - Search the dynamic chain until the first ARI for the static parent is found.
  - Easy, but slow.

Static Chain - Maintenance

- **Method 2:**
  - Treat subprogram declarations and calls like variable declarations and references.

Static Chain - Maintenance

- Given a subprogram call to S:
  - Have the compiler compute the nesting depth between the caller and the subprogram that declared S.
  - Store this nesting depth and send it with the call.
  - The SL of the S' ARI is determined by moving down the static chain of the caller the number of static links equal to the nesting depth.

Example: Static Chain - Maintenance

```
program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
      A := B + C;  <----------------------1
      end;  { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
        SUB1;
        E := B + A:   <-------------------2
        end; { SUB3 }
      begin { SUB2 }
      SUB3;
      A := D + E;  <----------------------3
      end; { SUB2 }
    begin { BIGSUB }
    SUB2(7);
    end; { BIGSUB }
  begin
  BIGSUB;
  end. { MAIN_2 }
```

- **At the call to SUB1 in SUB3, this nesting-depth is 2, which is sent to SUB1 with the call.**
- **The static link in the new ARI for SUB1 is set to point to the ARI that is pointed to by the second static link in the static chain from the ARI for SUB3.**

Static Chain - Evaluation

- A nonlocal reference is slow.
  - **(Nesting-Depth or SD + 1)** memory references!
- It is difficult to estimate the costs of nonlocal references for time-critical (real-time) programs.

2. Display

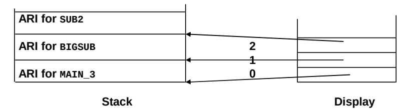
- The idea:
  - Put the static links in an array called a **display**.
  - Rather than being stored in the activation records.

Display

- The display contains a list of pointers to ARIs in the stack.
  - **One for each active static depth (static nesting level)!**
  - **Display[i]** = The **most recent** ARI of a subprogram with static depth (SNL) **i**
    - There are k+1 entries in the display where k is the static depth of the currently executing subprogram units.
    - k=0 is for the main program unit.

Example: Display

```
program MAIN_3;
  procedure BIGSUB;
    procedure SUB1;
      …
    end; {SUB1}
    procedure SUB2;
      procedure SUB3;
        …
      end; {SUB3}
      …
    end; {SUB2}
   SUB2;
  end; {BIGSUB}
BIGSUB;
end. {MAIN_3}
```



How to Access Nonlocal Variables Using Display

- A reference to a nonlocal variable X can be represented by the pair **(display_offset, local_offset)** where
  - **display_offset** = The same as **chain_offset**.
  - **local_offset** = The offset from the beginning of the AR of the subprogram containing the declaration for X.

How to Access Nonlocal Variables Using Display

- Use the **display_offset** to get the pointer to the correct ARI with the variable.
  - Display[display-offset]
- Use the **local_offset** to get to the variable within the ARI.
  - **Two memory references** for any nonlocal reference!

How to Maintain the Display?

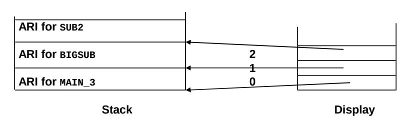
- During program execution!
- At a subprogram call:
  - Maintain the display condition:
  - **Display[i]** = The **most recent** ARI of a subprogram with static depth (SNL) **i**
- At a subprogram return:
  - Maintain the display condition:
  - **Display[i]** = The **most recent** ARI of a subprogram with static depth (SNL) **i**

Example: Display

```
program MAIN_3;
  procedure BIGSUB;
    procedure SUB1;
      …
    end; {SUB1}
    procedure SUB2;
      procedure SUB3;
        …
      end; {SUB3}
      SUB1;
    end; {SUB2}
  SUB2;
  end; {BIGSUB}
BIGSUB;
end. {MAIN_3}
```

MAIN3 calls BIGSUB calls SUB2

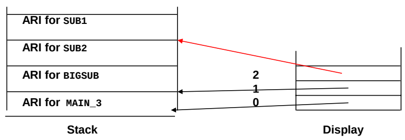
Calls SUB1?



Example: Display

```
program MAIN_3;
  procedure BIGSUB;
    procedure SUB1;
      …
    end; {SUB1}
    procedure SUB2;
        procedure SUB3;
          …
        end; {SUB3}
     SUB1;
    end; {SUB2}
   SUB2;
  end; {BIGSUB}
BIGSUB;
end. {MAIN_3}
```

MAIN3 calls BIGSUB calls SUB2 calls SUB1

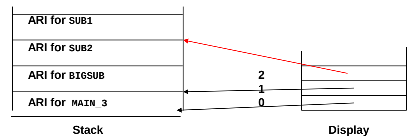


Example: Display

```
program MAIN_3;
  procedure BIGSUB;
    procedure SUB1;
      …
    end; {SUB1}
    procedure SUB2;
        procedure SUB3;
          …
        end; {SUB3}
     SUB1;
    end; {SUB2}
   SUB2;
  end; {BIGSUB}
BIGSUB;
end. {MAIN_3}
```

MAIN3 calls BIGSUB calls SUB2 calls SUB1

Returns from SUB1?



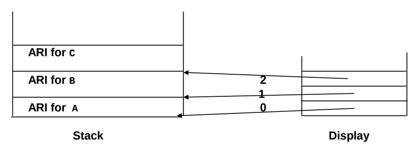
Display - Maintenance

- At a call to subprogram P with static_depth k:
  - Save in the new ARI for P a copy of the pointer stored at position k in the display.
  - Put the link to the ARI for P at position k in the display.
- At an exit:
  - Move the saved display pointer from the ARI for P back into the display at position k.

QUIZ: Display?

```
program A;
  procedure B;
     procedure C;
       B;
     end; {C}
   C;
  end; {B}
 B;
end. {A}
```

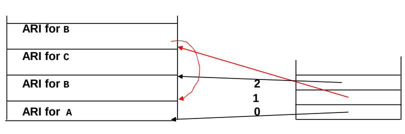
A calls B calls C



QUIZ: Display?

```
program A;
  procedure B;
     procedure C;
       B;
     end; {C}
   C;
  end; {B}
 B;
end. {A}
```

A calls B calls C calls B

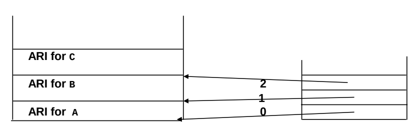


QUIZ: Display?

```
program A;
  procedure B;
     procedure C;
       B;
     end; {C}
   C;
  end; {B}
 B;
end. {A}
```

A calls B calls C calls B

Returns from B?



Display

- The display can also be kept in registers if there are enough.
  - It speeds up access and maintenance.

Static Chain vs Display Methods

- *References to locals*
  - Not much difference
- *References to nonlocals*
  - If it is one level away, they are equal.
  - If it is farther away, the display is faster.
  - Display is better for time-critical code, because all nonlocal references cost the same.

Static Chain vs Display Methods

- *Procedure calls*
  - For one or two levels of depth, static chain is faster.
  - Otherwise, the display is faster.
- *Procedure returns*
  - Both have fixed time, but the static chain is slightly faster.

Static Chain vs Display Methods

- Static chain is better:
  - If …
- Display is better:
  - If …

QUIZ: Static Chain vs Display

```
program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
      A := B + C;  <-----------------------1
      end;  { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
        SUB1;
        E := B + A:
        end; { SUB3 }
      begin { SUB2 }
      SUB3;
      A := D + E;
      end; { SUB2 }
    begin { BIGSUB }
    SUB2(7);
    end; { BIGSUB }
  begin
  BIGSUB;
  end. { MAIN_2 }
```

Call sequence:

MAIN_2 calls BIGSUB
BIGSUB calls SUB2
SUB2 calls SUB3
SUB3 calls SUB1

(1) Static chain?
(2) Display?

Implementing Dynamic Scoping

The Dynamic Scoping Rule

- Based on calling sequences of program units. (The program's dynamic flow of control)
- Not based on their textual layout (temporal versus spatial).

CS516 85

The Dynamic Scoping Rule

- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.
- Find the most recently active block containing the applied occurrence and a binding occurrence.

CS516 86

Implementing Dynamic Scoping

1. Using **Deep Access**
2. Using **Shallow Access**

CS516 87

Deep Access

- **Follow the dynamic chain!**
  – No need to maintain static links.
- Nonlocal references are found by searching all the activation record instances on the stack using the dynamic chain.

CS516 88

Deep Access vs Static Chain

- The deep access method:
  – The length of chain **cannot** be statically determined.
  – **Every activation record instance must store the names of variables.**

CS516 89

2. Shallow Access

- Put locals in a central place
- Method:
  – One stack for each variable name.
  – See Figure 10.12 (p. 422).

CS516 90

Deep Access vs Shallow Access

- Deep access:
  - Slow access
  - Fast calls and returns
- Shallow access:
  - Fast access
  - Slow calls and returns

Implementing Blocks

Blocks

- Blocks are entered and exited in strictly textual order.
  - No calls to blocks!
- Blocks can be treated as **parameterless subprograms** that are always called from the same place in the program.

Implementing Blocks

1. Treat blocks as parameterless subprograms
   - Use activation records and static chains or display.
2. Allocate locals **on top of the ARI** of the subprogram that contains the block.
   - Must use a different method to access locals.
   - A little more work for the compiler writer.

Example: Blocks

```
main() {
  int x, y, z;
  while (…) {
    int a, b, c;
    …
    while (…) {
      int d, e;
      …
    }
  }
  while (…) {
  int f, g;
  …
  }
…
}
```