Informatica 3

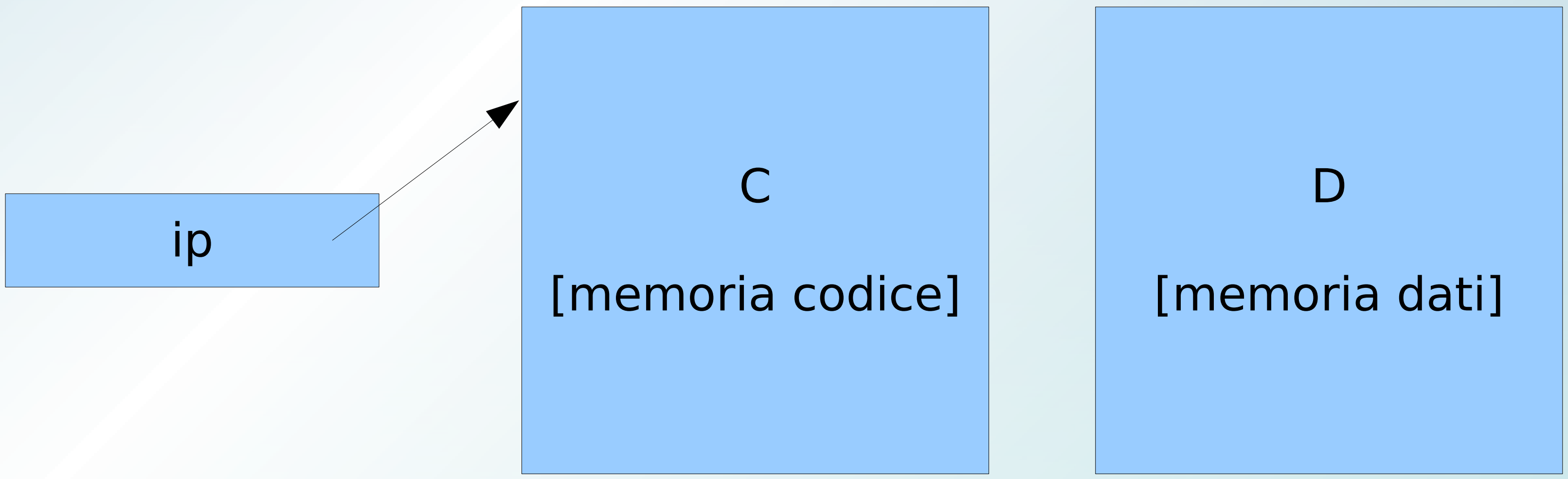
Marcello Restelli

Laurea in Ingegneria Informatica
Politecnico di Milano

9/15/07

Operational Semantics

- We describe the semantics of a programming language by means of an **abstract semantic processor**
  - SIMPLESEM
    - memory
    - instruction pointer
    - processor
- We show how language constructs can be executed by sequences of operations of the abstract processor



SIMPLESEM: notation

- D[X], C[X]
  - value stored in the X-th cell of D, C
- set target, source
  - for cell modification
  - set 10, D[20]
    - puts the value stored at location 20 into location 10
  - set 15, read
    - the value read from the input is stored at location 15
  - set write, D[50]
    - the value stored at location 50 is sent to output
  - set 99, D[15]+D[33]*D[41]
    - complex expressions acceptable

SIMPLESEM: control flow

- unconditional jump
  - jump 47
    - the next instruction becomes the one stored at address 47, i.e., ip becomes 47
- conditional jump
  - jumpt 47, D[3] > D[8]
    - jump occurs only conditionally
- indirect addressing
  - set D[10], D[20]
    - set the content of the cell, whose address is stored in D[10], to the content of D[20]
  - jump D[13]
    - jump to the address specified in the cell D[13]

Runtime Structure

- Languages can be classified according to their execution time structure
- Static languages
  - memory must be known and allocated before execution
  - no recursion
  - FORTRAN and COBOL
- Stack-based languages
  - memory is unknown at compile time, but usage is predictable and follows a last-in-first-out discipline
  - a predefined policy can be used for allocation/deallocation
- Dynamic languages
  - unpredictable memory usage
  - dynamic allocation
  - D handled as a HEAP

Runtime Structure of Programming Languages

- We use SIMPLESEM to study relevant concepts related to the execution time processing of programming languages
  - C1: a language with only simple statements
  - C2: adding simple routines
  - C3: supporting recursive functions
  - C4: supporting block structure
  - C5: toward more dynamic behavior

C1 Language

- Only simple types, int and float
- Fixed size arrays and structs
- Only simple statements
- No functions
- The program is a main routine enclosing
  - a set of data declaration
  - the statements that manipulate the data

```
main ( )
{
    int i, j;
    get (i, j);
    while (i != j)
        if (i > j)
            i -= j;
        else
            j -= i;
    print (i);
}
```

C1 Language: SIMPLESEM Representation

C

0	set 0, read
1	set 1, read
2	jumpt 8, D[0]=D[1]
3	jumpt 6, D[0]≤D[1]
4	set 0, D[0]-D[1]
5	jump 7
6	set 1, D[1] - D[0]
7	jump 2
8	set write, D[0]
9	halt

D

0	cell for i
1	cell for j

```
main ( )
{
        int i, j;
        get (i, j);
        while (i != j)
                if (i > j)
                        i -= j;
                else
                        j -= i;
        print (i);
}
```

ip | 0

C2 Language

- Extends C1 with routines
- C2 allows routines to declare **local data**
- C2 consists of
  - a set of data declaration
  - a set of routine definitions and/or declarations
  - a main routine with local data and statements
  - main cannot be called by other routines
  - Routines:
    - are not nested
    - cannot call themselves recursively
    - do not have parameters
    - do not return values

Computational Process of a Routine

- When a routine is called, a process instance is executed
- Routine instance
  - representation of the routine during the execution
- A routine is made up of
  - **code segment**
    - instructions of the unit
    - fixed content
  - **activation record**
    - information to execute the routine
      - local variables
      - return pointer
    - its content is not fixed
    - the relative position of the variables within the activation record is called **offset**

Static Allocation

- Size of activation records is determined at translation time
- Each unit's activation records can be allocated before execution, i.e., with static allocation
- Thus each variable can be bound to a D memory address before execution
- No memory allocation overhead at run-time
- Might waste memory space, memory is allocated for routines even if they are not used

A C2 Program

```
int i=1,j=2,k=3;
alpha()
{ int i=4,l=5;
  ...
  i+=k+l;
}

beta()
{ int k=6;
  ...
  i=j+k;
  alpha();
  ...
}

main ( )
{
  beta()
  ...
}
```

C	D	D
main code segment	global data	(i) (j) (k)
alpha code segment	alpha activation record	return pt (i) (j)
beta code segment	beta activation record	return pt (k)

A C2 Program

```
int i=1,j=2,k=3;
alpha()
{ int i=4,l=5;
  ...
  i+=k+l;
}

beta()
{ int k=6;
  ...
  i=j+k;
  alpha();
  ...
}

main ( )
{
  beta()
  ...
}
```

C

	...	main
14	set 6,16	
15	jump 100	
16	...	
	...	
50	...	alpha
	...	
58	set 4,D[4]+D[2]+D[5]	
59	...	
	jump D[3]	
100	...	beta
122	set 0,D[1]+D[7]	
123	set 3,125	
124	jump 50	
	...	
	jump D[6]	

D

0	(i)	8
1	(j)	2
2	(k)	3
3	(return pt)	125
4	(i)	12
5	(j)	5
6	(return pt)	16
7	(k)	6

ip 59

Separate Compilation for C2

file 1
```
extern beta();
int
i=1,j=2,k=3;
main()
{ ...
  beta();
  ...
}
```

file 2
```
extern int
k;
alpha()
{
  ...
}
```

file 3
```
extern int i,j;
extern
alpha();
beta()
{ ...
  alpha();
  ...
}
```

- Compile time
  - local variables can be bound to offset (not to an absolute address)
  - Imported global variables cannot be bound to offsets in the global AR
  - routine calls cannot be bound to code segments
- Link time
  - storage bound to code segments and activation records
  - all missing information can be filled

C3 Language

- C3 is derived from C2 adding
  - **recursion**
    - **direct:** the routine calls itself
    - **indirect:** routines call others, which in turn recall them
  - **functions**
    - routines can return values, as functions do

```
int n
int fact()
{
  int loc;
  if (n>1)
  {
     loc = n--;
     return
loc*fact();
  } else {
     return 1;
  }
}
```

```
main()
{
  get(n);
  if (n>=0)
     print(fact());
  else
     print(“input
error”);
}
```

What are the Issues?

- Recursions do not allow static allocation of activations records
  - The number of instances for each unit is unknown at compile time
  - How many times a routine will be called?
- Return value must be passed to the caller
  - Problem: the AR is deallocated when the routine is exited
  - We should store it in the AR of the calling unit

Consequences of Recursion

- The **size** of activation records is known in advance
- Different instances have the **same code segment** but **different activation records**
- The data memory D is managed as a stack
  - When a routine is entered, its activation record is allocated
  - When a routine is exited, the corresponding activation records must be discarded
  - Last activation record is discarded first, following a Last In First Out (LIFO) policy

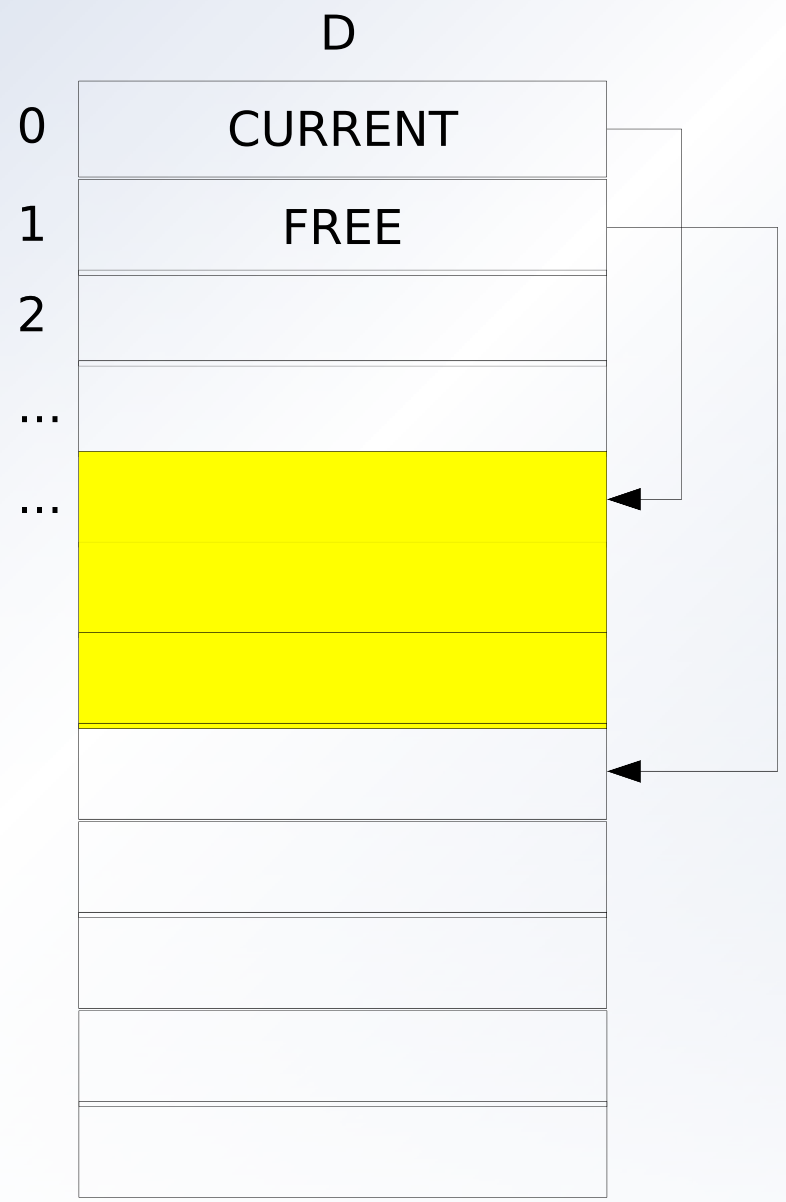
Solutions (1)

- At translation time, local variables can be bound relatively to an **offset** in the activation record
- Final step of **binding** (for computing the absolute address) has to be done at **execution time**
  - We need to know the base address of the current activation record
  - We use the first cell in D (D[0]) to store the base address of the activation record of the unit currently executing
  - We call the value in D[0] CURRENT
- We also need a pointer to the next available position where a new activation record might be stored
  - We use cell D[1] to store the address of the next free position on the stack
  - We call the value in D[1] FREE

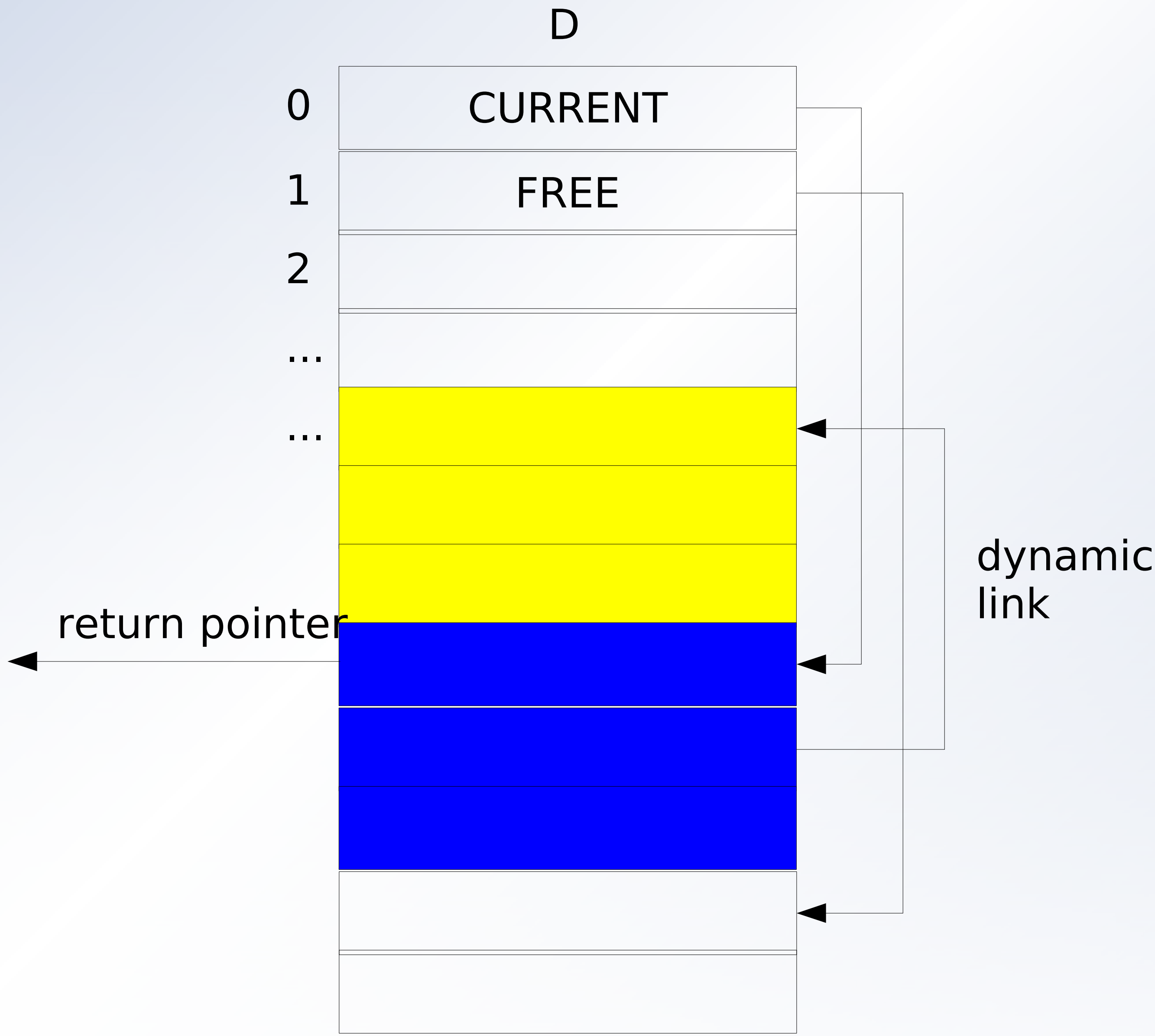
Solutions (2)

- The information on the caller also changes
  - The caller can be one of the possible routine instances
- To make the **return** from an activation possible, information on the caller must be stored in the activation record
  - instruction to execute next (**return pointer**) [offset 0]
  - reference to the caller Activation Record (**dynamic link**) [offset 1]
  - dynamic links define the **dynamic chain**

Example



Example



Initialization

- IP is set to the address of the first location of C that contains executable code
- The statement at location 0 initializes FREE
  - D[1] is set to the address of the first free location after the main’s activation record

Semantics of Call and Return

- Routine call

```
set 1, D[1] + 1       allocate space for return value
set D[1], ip + 4      set return point
set D[1] + 1, D[0]    set dynamic link
set 0, D[1]           set CURRENT
set 1, D[1] + AR      set FREE
jump start_addr       address of the first instruction
```

- Return from routine

```
set 1, D[0]           set FREE
set 0, D[D[0] +1]     set CURRENT
jump D[D[1]]          jump to the stored return point
```

A C3 Example: Data Representation

```
int n
int fact()
{
  int loc;
  if (n>1)
  {
     loc = n--;
     return
loc*fact();
  } else {
     return 1;
  }
}
main()
{
  get(n);
  if (n>=0)
     print(fact());
  else
     print(“input
error”);
}
```

D

0	(CURRENT)	global data
1	(FREE)	
2	(n)	
3	(return value)	
4	(return pt)	fact activation record
5	(dynamic link)	
6	(loc)	

A C3 Example: Representation of main()

```
int n
int fact()
{
  int loc;
  if (n>1)
  {
     loc = n--;
     return
loc*fact();
  } else {
     return 1;
  }
}
main()
{
  get(n);
  if (n>=0)
     print(fact());
  else
     print(“input
error”);
}
```

```
0    set 2, read              ;reads the value of n
1    jumpt 10, D[2] < 0       ;tests the value of n
2    set 1, D[1] + 1          ;call to fact start -
                              ;space for result saved
3    set D[1], ip + 4         ;set return pointer
4    set D[1] + 1, D[0]       ;set dynamic link
5    set 0, D[1]              ;set CURRENT
6    set 1, D[1] + 3          ;set FREE
                              ;3 is the size of fact’s
AR
7    jump 12                  ;12, start address
of fact
8    set write, D[D[1]-1]     ;prints result of call
9    jump 11                  ;end of call
10   set write, "input error"
11   halt                     ;end of main
```

A C3 Example: Representation of fact()

```
int n
int fact()
{
  int loc;
  if (n>1)
  {
     loc = n--;
     return
loc*fact();
  } else {
     return 1;
  }
}
main()
{
  get(n);
  if (n>=0)
     print(fact());
  else
     print(“input
error”);
}
```

```
                                ;starts of fact()
12  jumpt 23, D[2] <= 1     ;tests the value of n
13  set D[0] + 2, D[2]   ;assigns n to loc
14  set 2, D[2] – 1        ;decrements n
15  set 1, D[1] + 1        ;call to fact starts
                                ;space for result
16  set D[1], ip + 4       ;set return pointer
17  set D[1] + 1, D[0]   ;set dynamic link
18  set 0, D[1]             ; set CURRENT
19  set 1, D[1] + 3        ;FREE:3 is the size of fact’s
AR
20  jump 12                       ;12 is the starting addr.
of                                 fact()
21  set D[0] - 1, D[D[0] + 2] * D[D[1] - 1]
                                ;return value stored
22  jump 24
23  set D[0] - 1, 1        ;return value (1) stored
24  set 1, D[0]             ;return from the routine
starts
25  set 0, D[D[0] + 1]
26  jump D [D[1]]
```

A C3 Example: Representation of fact()

```
int n
int fact()
{
  int loc;
  if (n>1)
  {
    loc = n--;
    return
loc*fact();
  } else {
    return 1;
  }
}
main()
{
  get(n);
  if (n>=0)
    print(fact());
  else
    print(“input
error”);
}
```

D

0	(CURRENT) 4	global data
1	(FREE) 7	
2	(n) 3	
3	(return value)	
4	(return pt) 8	fact activation record
5	(dynamic link) 2	
6	(loc)	

D

0	(CURRENT) 12
1	(FREE) 15
2	(n) 1
3	(return value)
4	(return pt) 8
5	(dynamic link) 2
6	(loc) 3
7	(return value)
8	(return pt) 21
9	(dynamic link) 4
10	(loc) 2
11	(return value) 1
12	(return pt) 21
13	(dynamic link) 8
14	(loc)

C4 Language

- C4 adds the concept of **block**
- C4’ allows local declarations to appear within any compound statement
- C4” supports the ability to nest a routine definition within another
- The features of C4’ and C4” are collectively called **block structure**
  - to control variables' scope
  - to define variables' lifetime
  - to decompose the program into smaller units
  - Memory space is bound to a variable when the block in which it is declared is entered during execution
  - The binding is removed when the block is exited

C4': Nesting Compound Statements

- In C4’, blocks have the following form of compound statement:
  - {<declaration list>; <statement list>}
- Blocks can appear whenever a statement can appear
- A compound statement defines the scope of its locally declared variables
- Such variables are visible within the compound statement, including any nested compound statement

C4': An Example

- f() has local declaration of x,y,w
- x is redeclared in //2
- the outer declaration of x is invisible until the loop termination
- y is redeclared in //3
- the outer declaration of y is invisible until the while ends
- w is redeclared in //4
- the outer declaration of w is invisible until the end of the block
- X declaration in //4 masks x in //2

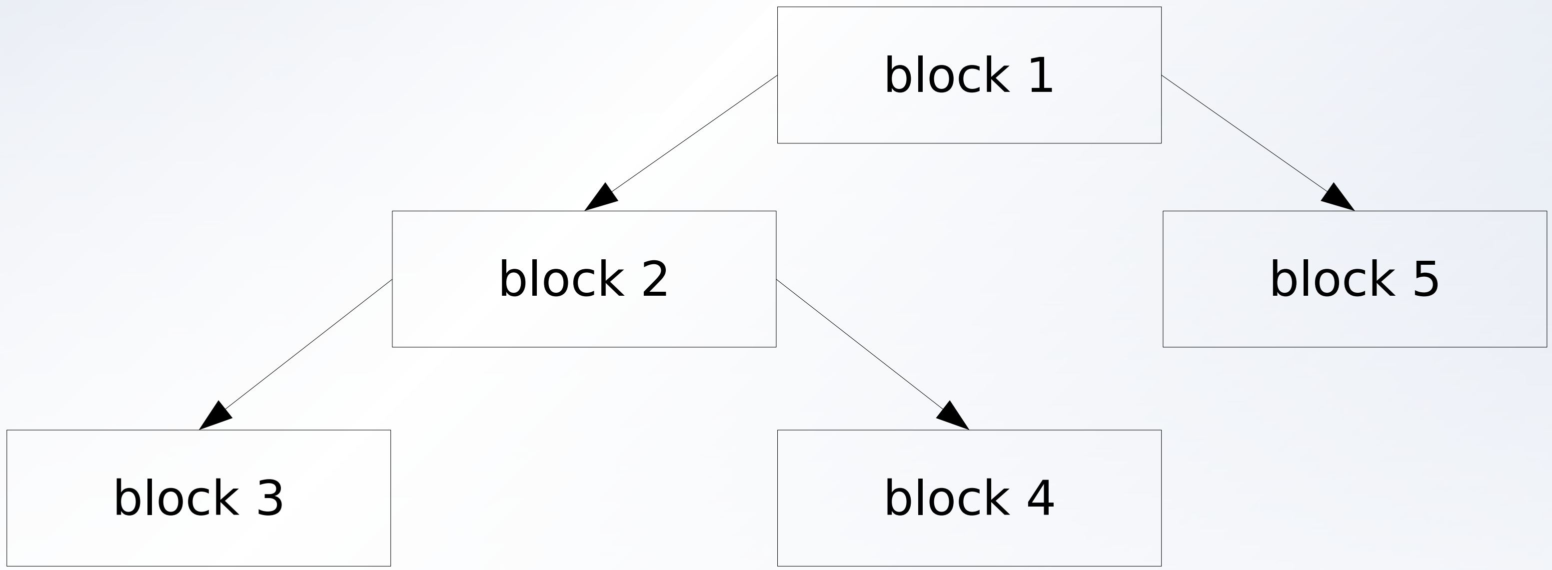
```
int f()
{                        //block 1
  int x,y,w;             //1
  while(...)
  {                      //block 2
    int x,z;             //2
    ...
    while (...)
    {                    //block 3
       int y;            //3
       ...
    }                    //end block 3
    if (...)
    {                    //block 4
       int x,w;          //4
       ...
    }                    //end block 4
  }                      //end block 2
  if (...)
  {                      //block 5
    int a,b,c,d;         //5
    ...
  }                      //end block 5
}                        //end block 1
```

Compound Statements in Routine

- Two implementation options in SIMPLESEM
  - **Statically** including the memory needed by the compound statement in the activation record of the enclosing routine
  - **Dynamically** allocating new memory space corresponding to local data as each compound statement is entered during execution
- The **static** scheme is **simpler** and **more time efficient** (no overhead at runtime)
- The **dynamic** scheme is **more space-efficient**

Static Scheme

- Describe the block structure by a static nesting tree (SNT)
- An SNT shows how block are nested into another
- Store in the same cells the variables of disjoint blocks
- Activation records are **overlayed**



An Overlayed Activation Record

return pointer
dynamic link
x in //1
y in //1
w in //1
x in //2 – a in //5
z in //2 – b in //5
y in //3 – x in //4 – c in //5
w in //4 – d in //5

- Overlays can be defines at translation time
- The runtime behavior of C4' is the same as for C3

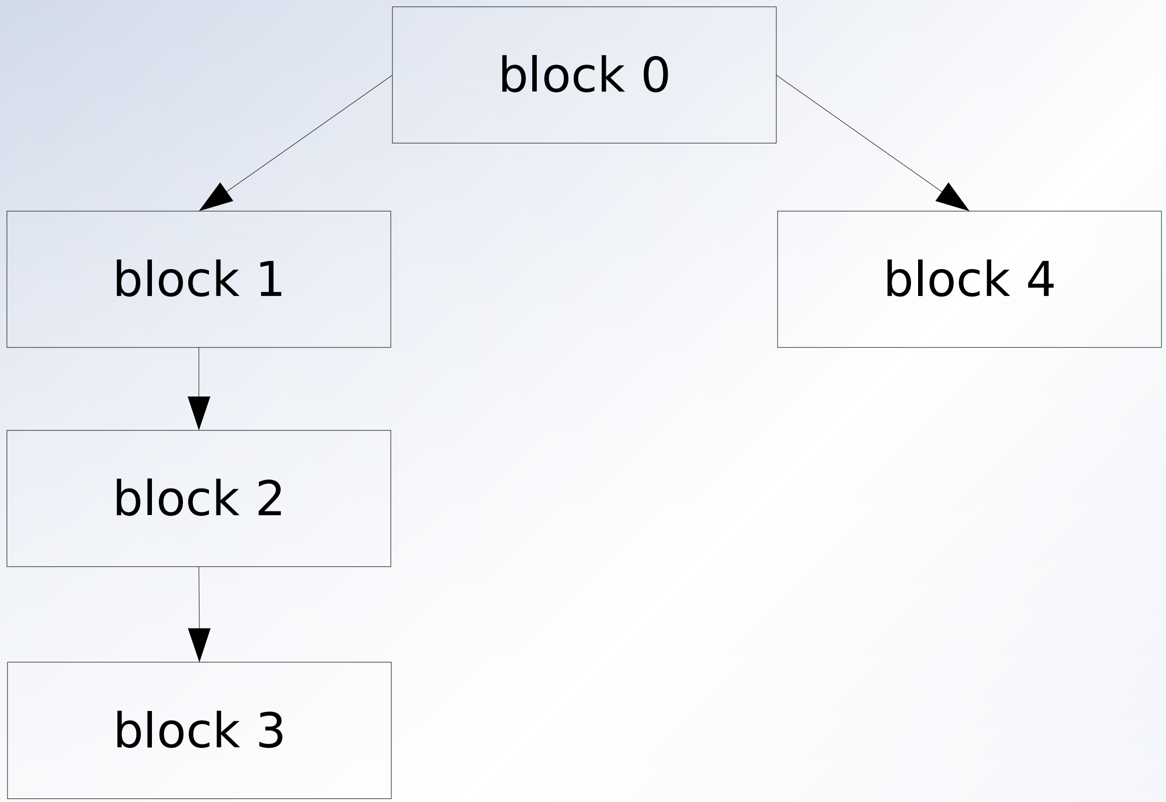
C4": Nesting Routines

- Routine may be declared within another
- f3 can be called within f2
- f3 can be called also by f3
- f2 can be called:
  - - within f1 (local call)
  - - within f2 (direct recursion)
  - - within f3 (non local call)
- As before, local declarations mask outer declarations
- C and C++ support only the nesting of compound statements
- Pascal and Modula-2 allow the nesting of routines
- Ada allows both

```
int x, y, z;
f1 ()
{//block 1
  int t,u;              //1
  f2()
  {//block 2
    int x, w;           //2
    f3 ()
    {//block 3
       int y, w, t; //3
    }//end block 3
    x = y+t+w+z;
   }//end block 2
}//end block 1

main ( );
{//block 4
  int z, t;             //4
}//end block 4
```

C4": Static Nesting Tree




```
int x, y, z;
f1 ()
{//block 1
  int t,u;              //1
  f2()
  {//block 2
    int x, w;           //2
    f3 ()
    {//block 3
       int y, w, t; //3
    }//end block 3
    x = y+t+w+z;
   }//end block 2
}//end block 1

main ( );
{//block 4
  int z, t;             //4
}//end block 4
```

A Sketch of the Runtime Stack

- Suppose we execute x=y+t+w+z in f2
- The binding of x and w is obvious
- What about t, y, and z?
- Dynamic
  - y and t to f3
  - z to main
- But, the binding of t, y, and z should follow the static rules

A Sketch of the Runtime Stack

- Binding
  - The sequence of activation records stored in the stack represents the sequence of unit instances as they are generated at runtime
  - But the non local environment is determined by the scope rules of the language that are based on the static nesting of routines

offset	RdA
0	return pointer
1	dynamic link
2	static link
3	local variables
...	...

Access to Nonlocal Variables

- Nonlocal variables may be accessed through the sequential search along the static chain
- But this solution is **inefficient** since it requires runtime overhead and never necessary
- Reference to nonlocal variables can be bound statically since the distance along the static chain is fixed
- Variable references can be bound statically to a pair **<distance, offset>**
- **distance** indicates the number of steps along the static chain
  - local variables: distance = 0
  - variable defined in the external unit: distance = 1
- **offset** indicates the variable's relative address within the activation record

Nonlocal Variables in SIMPLESEM

- Let d be the distance on the static chain
- Let fp(d) addresses the dth activation record along the static chain (fp stands for frame-pointer)
- Assume that the link to the static chain is in position 2 in the activation record
- Given a variable described as <d,o>
- fp(d) = if d=0 then D[0] else D [fp(d-1)+2]
  - Examples
    - fp(0) = D[0]
    - fp(1)=D[fp(0)+2]=D[D[0]+2]
- The variable value is at D[fp(d)+o]

C4": Routine Call

```
set 1, D[1] + 1            ;allocate space on the stack for
                           ;the return value
set D[1], ip + 5           ;set the value of the return pointer in
                           ;the callee activation record. 5 is the
                           ;number of ins. needed to imp. the call
set D[1] + 1, D[0]         ;set the dynamic link of callee to the
                           ;caller's activation record
set D[1] + 2, fp(d)        ;set the static link
set 0, D[1]                    ;set CURRENT
set 1, D[1] + AR           ;set FREE, AR is the size of the callee's
                           ;activation record
jump start_addr            ;start_addr of memory C where
                           ;the callee's code starts
```

C5: Towards more dynamic behaviors

- So far
  - data storage requirements of each unit are known at compile time
  - the mapping between variables and activation records can be performed at compile time, i.e., each variable is bound to its offset statically
- What if language does not conform to this assumptions?

C5': Variable Size Known at Runtime

- Dynamic arrays in Ada

```
type VECTOR is array (INTEGER range <>);
  --defines arrays with unconstrained index
A: VECTOR (1..N);
B: VECTOR (1..M);
   --N and M must be bound to some int value when
   --declarations elaborated at runtime
```

- At translation, the descriptor for the dynamic array is allocated
- The descriptor includes
  - a pointer to the dynamic array base location
  - cells for upper and lower bounds
- The array object is
  - allocated on top of the newly allocated activation record
  - deallocated at the end of its declaration unit
- Access to array is performed indirectly through pointer

Allocation of the Activation Record

1. storage for data whose size is statically known and descriptors for dynamic arrays
2. when the declaration of a dynamic array is encountered
   1. the actual size is evaluated
   2. the activation record is extended to make room for the array elements (FREE is incremented)
3. the pointer in the descriptor is set to the newly allocated area

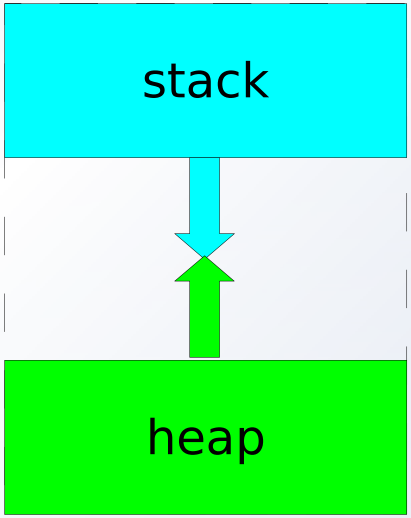
Example: if array descriptor **A** is at position **m**, and **I** is at position **s** in the activation record, then

$$\texttt{A[I] = 0} \rightarrow \texttt{set}\underbrace{\texttt{[D[D[0] + m]}}_{\text{base address of A}} \texttt{ + D[}\underbrace{\texttt{D[0] + s]}}_{\text{value of I}}\texttt{], 0}$$

C5'': Fully Dynamic Allocation

```
struct nodo
{
   int num;
   nodo* succ;
};

nodo* n = new nodo;
```



- The lifetime of dynamic variables does not depend on the lifetime of the units where they are defined
- These data are not allocated on the **stack**, but on the **heap**
  - we will store dynamic data from the last cell of D
  - for sake of ease, we will assume that D is large enough

The Structure of Dynamic Languages

- Dynamic languages adopt dynamic rather than static rules
  - E.g., APL, SNOBOL4, and LISP use dynamic typing and dynamic scoping rules
- **Dynamic typing**
  - A variable in the activation record is represented by a pointer to the data object in the heap (size can change dynamically)
  - It requires dynamic type checking and policy for size changes
- **Dynamic scoping**
  - the dynamic chain supports access to non-local objects

Dynamic Scoping

```
sub2()
{
  declare x;
  ... x ...;
  ... y ...;
}
sub1()
{
  declare y;
  ... x ...;
  ... z ...;
    sub2();
}
main()
{
  declare x,y,z;
  z = 0;
  x = 5;
  y = 7;
  sub1();
  sub2();
}
```

- declaration introduces the name not the type
- scope depends on the runtime call chain

Dynamic Scoping

```
sub2()
{
  declare x;
  ... x ...;
  ... y ...;
}
sub1()
{
  declare y;
  ... x ...;
  ... z ...;
    sub2();
}
main()
{
  declare x,y,z;
  z = 0;
  x = 5;
  y = 7;
  sub2();
}
```

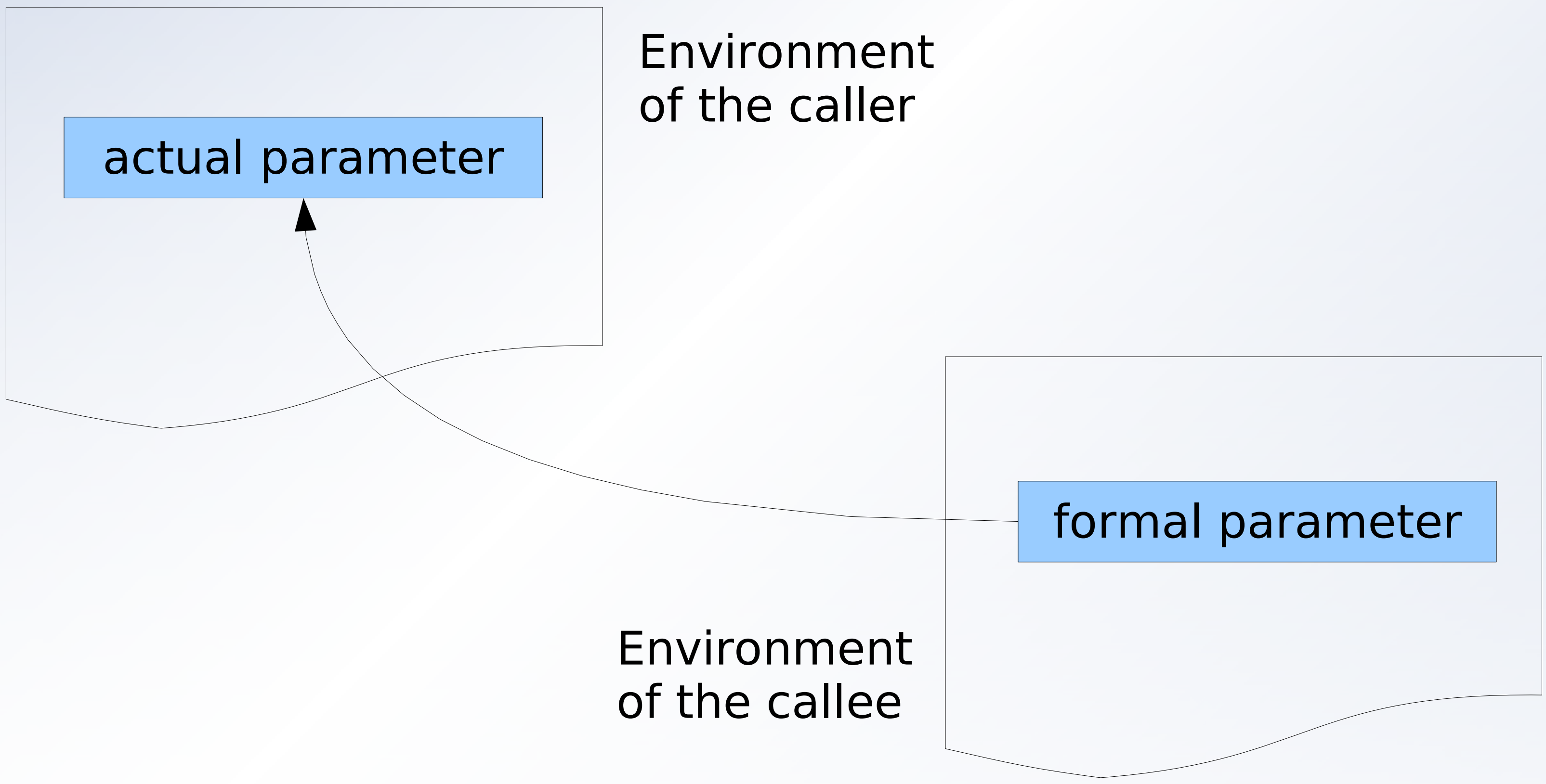
- declaration introduces the name not the type
- scope depends on the runtime call chain

Parameter Passing

- Data parameters
  - by reference or by sharing
    - caller passes the address of the actual parameter
  - by copy
    - parameters are used as local variables
  - by name
    - name of actual parameters are replaced with the name of formal ones
- Routine parameters

Call by Reference

- Caller passes the address of the actual parameter
- Reference to formal parameter treated as indirect reference



Semantics of Call by Reference

- We need to extend C4
- The activation record contains one cell for each parameter
- Suppose an actual parameter is described as <d,o>
- The caller initializes the content of the cell with the address of the actual parameter (off is the offset of the formal parameter)

  set D[0] + off, fp(d) + o

- If the actual parameter is a by-reference parameter:

  set D[0] + off, D[fp(d) + o]

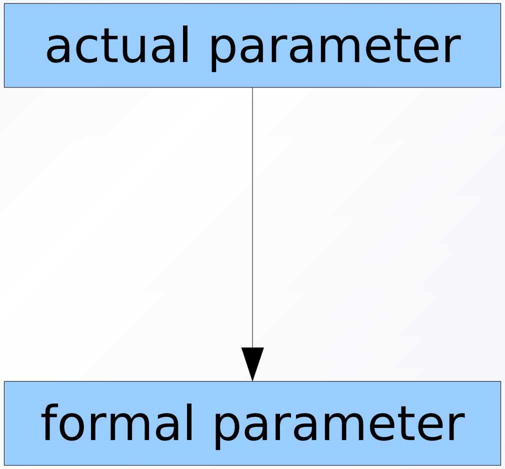
- parameters accessed via indirect addressing
  - Es.: x is a formal parameter, off is its offset
  - “x=0” is translated as “set D[D[0] + off], 0”
- What if an actual parameter is an expression or a constant?

Call by copy

- Formal parameters do not share storage with actual parameters
- Formal parameters act as local variables
- There are three modes corresponding to different policies to initialize the local variables corresponding to the formal parameters
  - call by value
  - call by result
  - call by value-result
- The SIMPLESEM implementation is straightforward
  - the parameters are considered as local variables
  - at the beginning and at the end of the routine call the values are copied accordingly to the type of passage

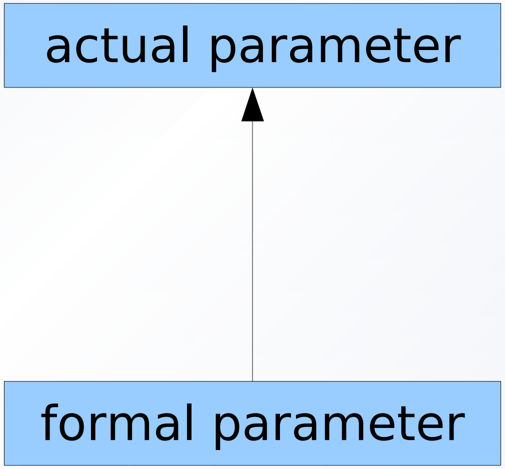
Call by Value

- caller evaluates actual parameters
- corresponding formals initialized with such values
- no flow of information back to the caller



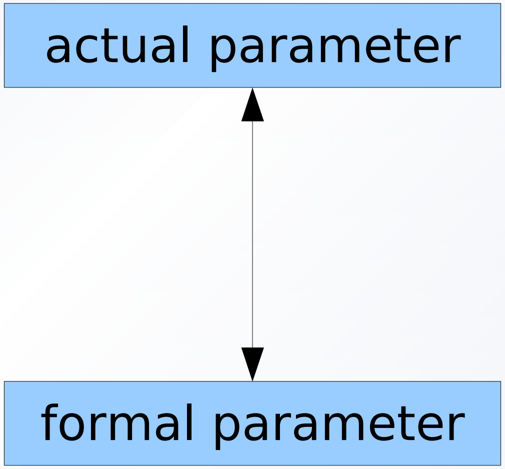
Call by Result

- local variables corresponding to formal parameters are not set at subprogram call
- at return, values of formals copied back into actual parameters
- no flow of information from caller to callee



Call by Value-Result

- both copied at call and at return
- information flow from caller to callee



By value-result vs. By reference

- Different effect in the following cases:
  - **1) Two formal parameters become aliases**
  - By reference
    - a[i] is set to 0
    - then a[j] (i.e., a[i]) is incremented
    - when returning a[i]=a[j]=1
  - By value-result
    - x and y are set to 10
    - x is set to 0
    - y is incremented to 11
    - when returning:
    - 0 is copied in a[i], and 11 is copied in the same cell. Therefore a[i]=a[j]=11

```
foo(x,y)
{
  x=0;
  y++;
}

i=j;
a[i]=10;
foo(a[i],a[j]);
```

By value-result vs. By reference

- Different effect in the following cases:
  - **2) A formal parameter and a nonlocal variable are aliases**
  - By reference
    - when returning a=2
  - By value-result
    - when returning a=11:

```
goo(x)
{ ...
  a=1;
  x=x+a;
}

a = 10;
...
goo(a);
```

Call by Name

- Defined by textual substitution of variable names between formal and actual parameters
- As in “call by reference”, formal parameters denote locations in the environment of caller
- Unlike with “call by reference”, a formal parameters is not bound to a location at the point of call, but it can be bound to a different l-value each time it is used
- Each assignment can refer to a different location
- Appears to be simple, but the call-by-name substitution can be deceiving, leading to unexpected results and leads to programs that are hard to read
- It is also hard to implement
  - Each formal parameter is replaced by a routine, thunk, which evaluates the reference to the actual parameter and the value of the formal parameter

Call by Name: Example

```
swap (int a,b);
{
    int temp;
    temp = a;
    a = b;
    b = temp;
};

...

i = 3;
a[3] = 4;
swap(i,a[i]);
```

```
swap (int i,a[i]);
{
    int temp;
    temp = i;       // temp=3
    i = a[i];       // i=4
    a[i] = temp;    // a[i] is a[4]
                    // a[4]=3
                    // a[3] is unaffected!
};
```

Languages and Parameter Passing

	Per riferimento	Per valore	Per nome
Fortran	X		
Algol 60		X	X
Simula 67	X	X	X
Pascal, C++	X	X	
C	con puntatori	X	

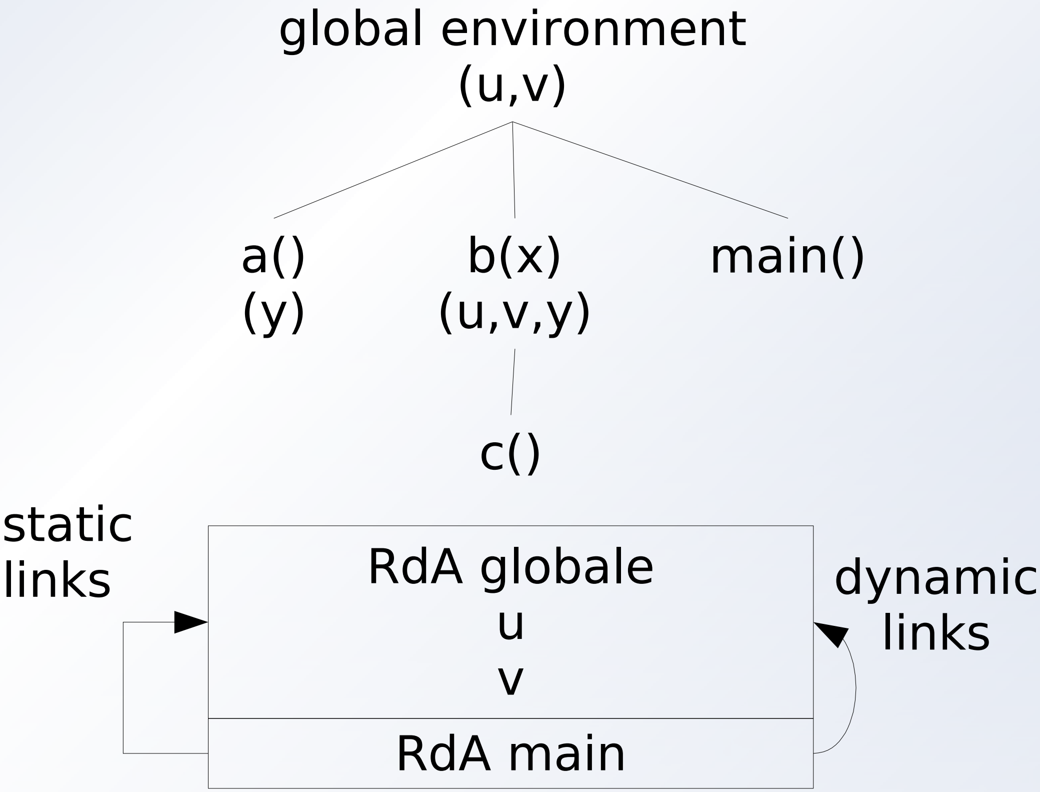
Routine Parameters

```
int u, v;
a()
{
  int y;
  ...
};
b(routine x)
{
   int u, v, y;
   c()
   { ...
      y = ...;
      ...
   };
     x();
     b(c);
     ...
}
main ( )
{
   b(a);
};
```

- Routine parameters behave differently when the language is dynamically or statically scoped
- We will considered only statically scoped languages
- Information to pass to the callee at runtime:
  - Size of the AR
  - routine’s nonlocal environment (static link, SL)

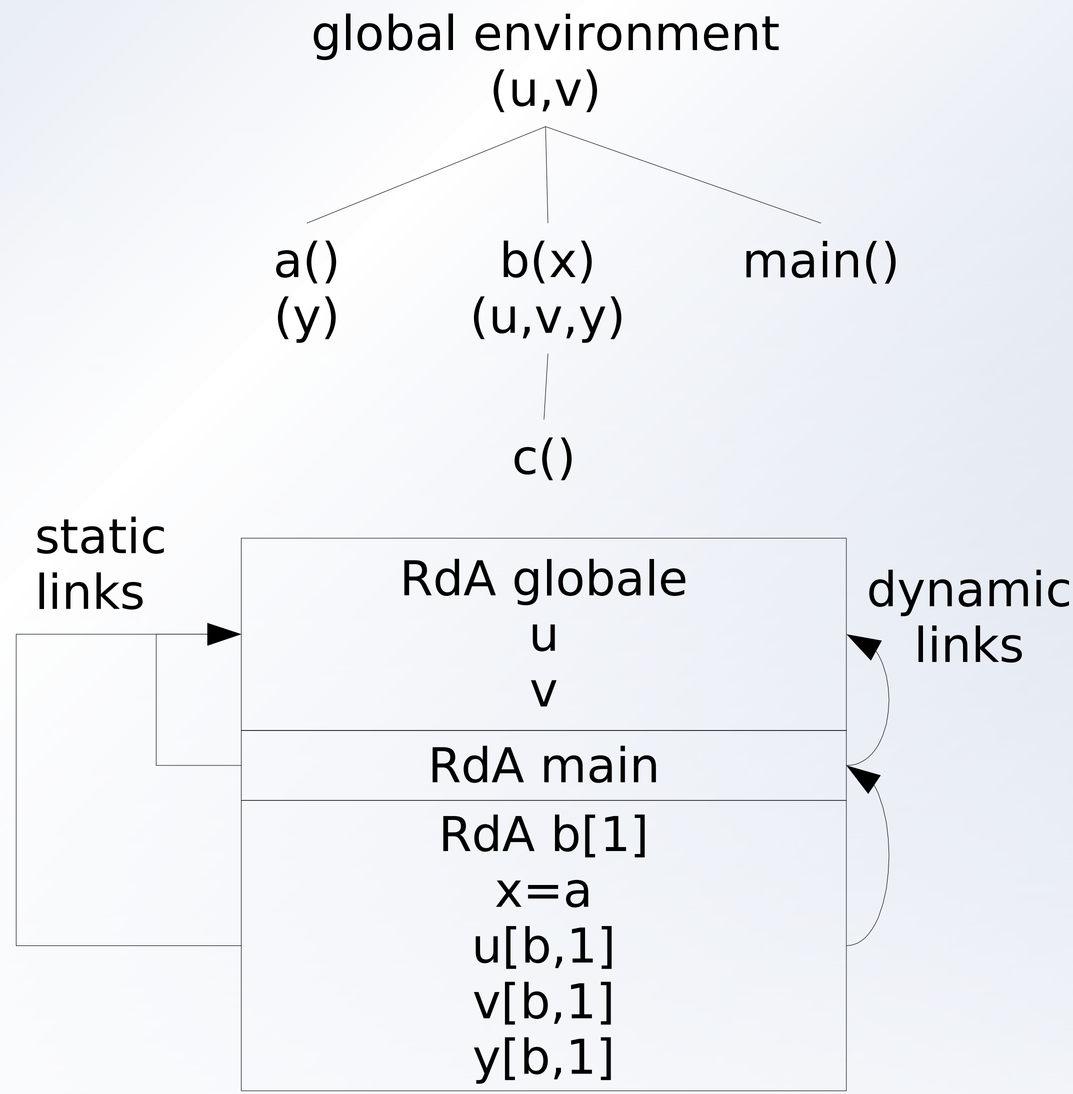
Routine Parameters

```
int u, v;
a()
{
  int y;
  ...
};
b(routine x)
{
   int u, v, y;
   c()
   { ...
      y = ...;
      ...
   };
     x();
     b(c);
     ...
}
main ( )
{
   b(a);
};
```



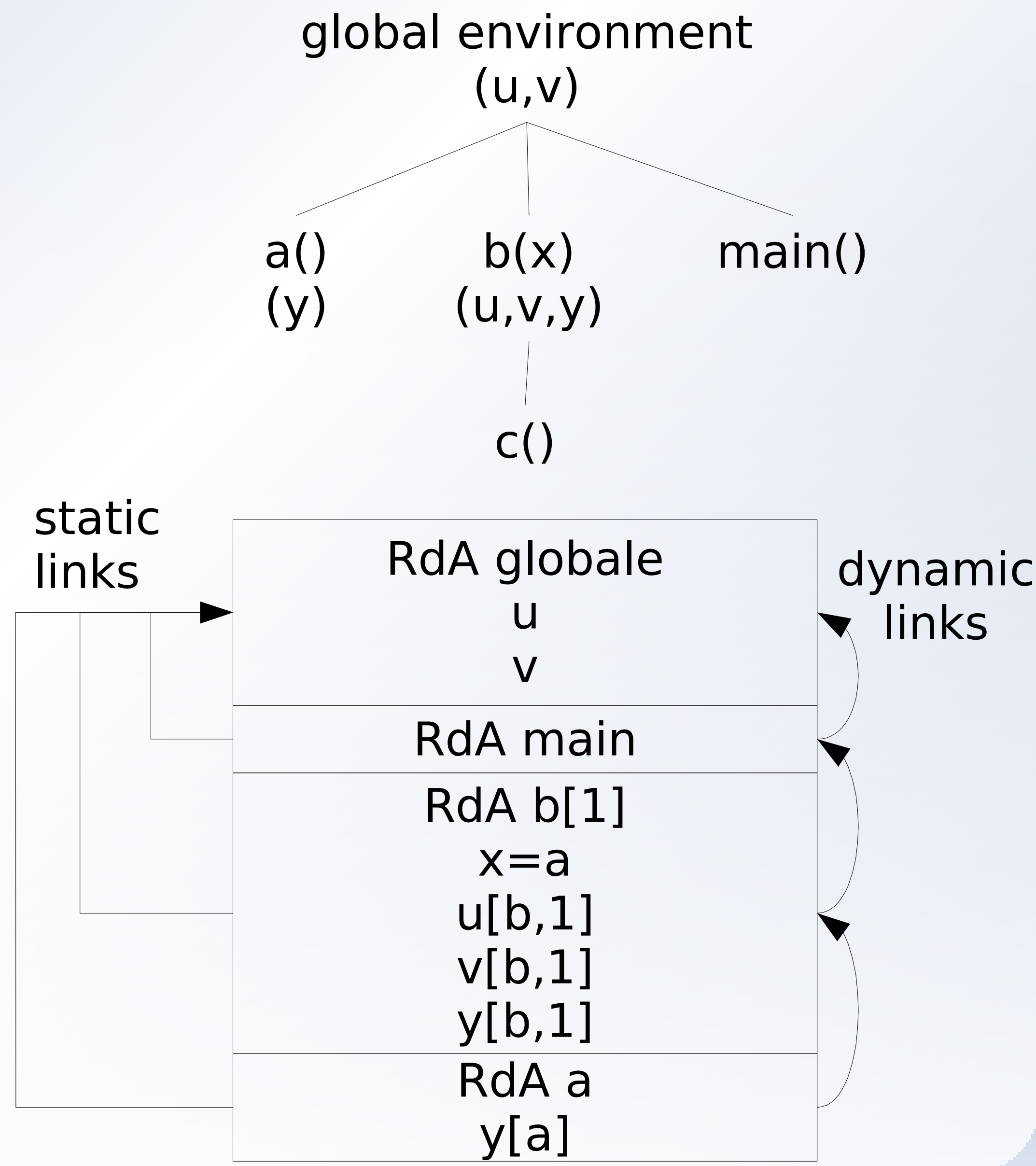
Routine Parameters

```
int u, v;
a()
{
  int y;
  ...
};
b(routine x)
{
   int u, v, y;
   c()
   { ...
      y = ...;
      ...
   };
     x();
     b(c);
     ...
}
main ( )
{
   b(a);
};
```



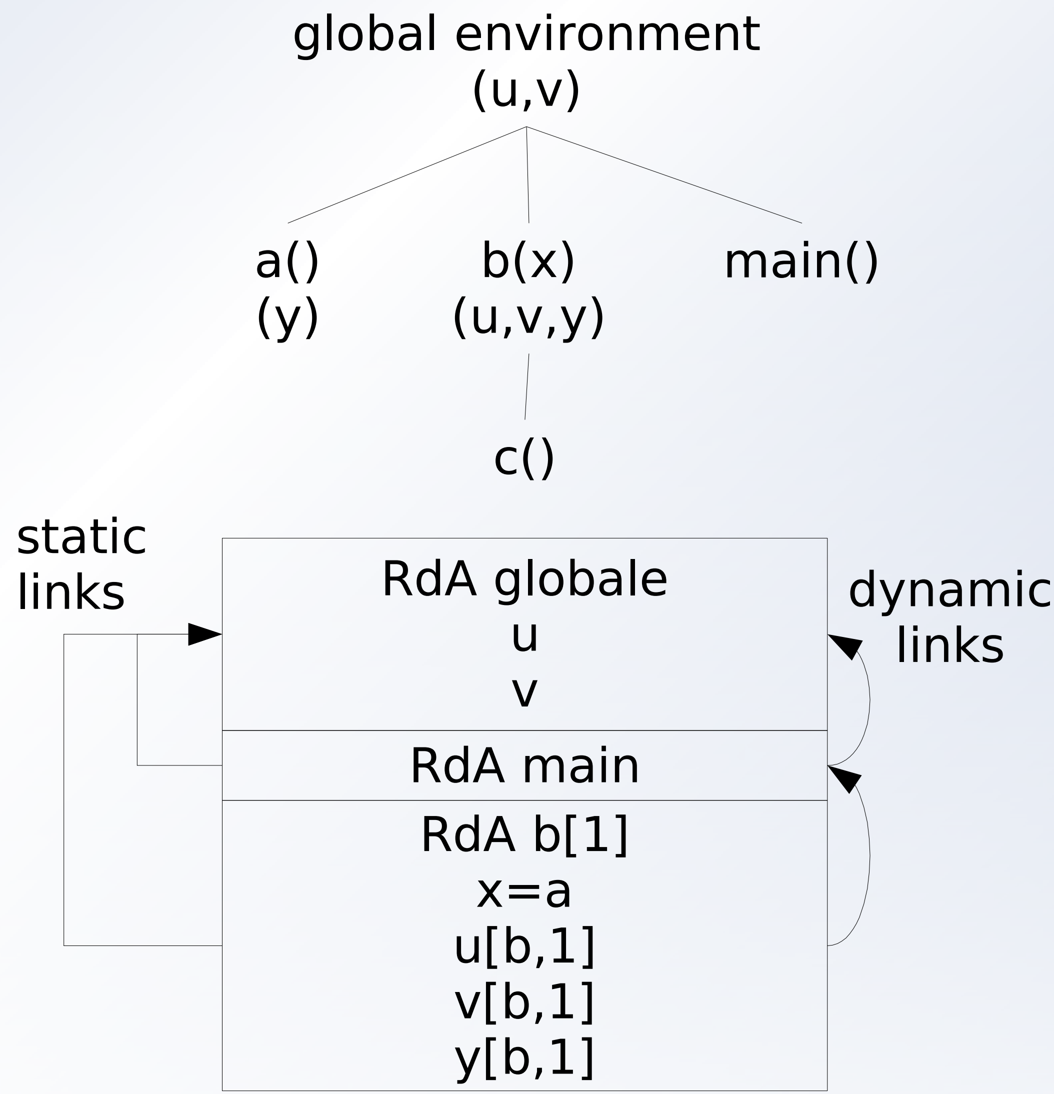
Routine Parameters

```
int u, v;
a()
{
  int y;
  ...
};
b(routine x)
{
   int u, v, y;
   c()
   { ...
      y = ...;
      ...
   };
     x();
     b(c);
     ...
}
main ( )
{
   b(a);
};
```



Routine Parameters

```
int u, v;
a()
{
  int y;
  ...
};
b(routine x)
{
   int u, v, y;
   c()
   { ...
      y = ...;
      ...
   };
     x();
     b(c);
     ...
}
main ( )
{
   b(a);
};
```



Routine Parameters

```
int u, v;
a()
{
  int y;
  ...
};
b(routine x)
{
   int u, v, y;
   c()
   { ...
      y = ...;
      ...
   };
     x();
     b(c);
     ...
}
main ( )
{
   b(a);
};
```