Informatica 3

Part I: Programming Languages
Syntax and semantics (B)

Laurea in Ingegneria Informatica
Politecnico di Milano
Polo di Milano Leonardo

Syntax and semantics

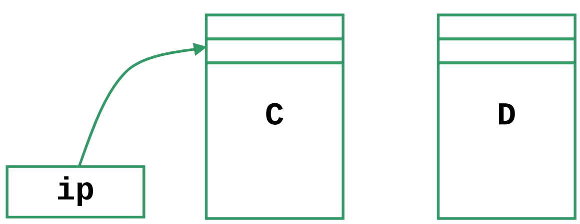
- An asbtract semantic processor
- Run-time structure

- Ghezzi & Jazayeri. Programming Language Concepts. Chapter 2.

Abstract semantic processor

- To define the language semantics
  we use an operational approach
- We introduce a simple abstract processor
- We show how language constructs can be executed by sequences of operations of the abstract processor

SIMPLESEM



- Code memory **`C`**
- Data memory **`D`**
- Instruction pointer **`ip`**

- Interpretation cycle
  1. Get the current instruction **`C[ip]`**
  2. Increment **`ip`**
  3. Execute the current instruction

SIMPLESEM: notation

- D[I], C[I]
  value stored in the I-th cell of D, C
- **set target, source** for cell modification
  - set 10, D[20]
    puts the value stored at loc. 20 into loc. 10
  - set 15, read
    the value read from the input is stored at loc. 15
  - set write, D[50]
    the value stored at loc. 50 is sent to output
  - set 99, D[15]+D[33]*D[41]
    complex expressions acceptable
  - set D[10], D[20]
    set the content of the cell, whose address is stored in D[10], to the content of D[20] – **Indirect addressing**

SIMPLESEM: control flow

- jump 47
  the next instruction becomes the one stored at address 47, i.e., **ip** becomes 47
- jumpt 47, D[3] > D[8]
  jump occurs only conditionally
- 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

- 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

The 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

A C1 program

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

SIMPLESEM representation

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

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

cell for i
cell for j

Program Activation Record

ip

0

The 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

Static allocation

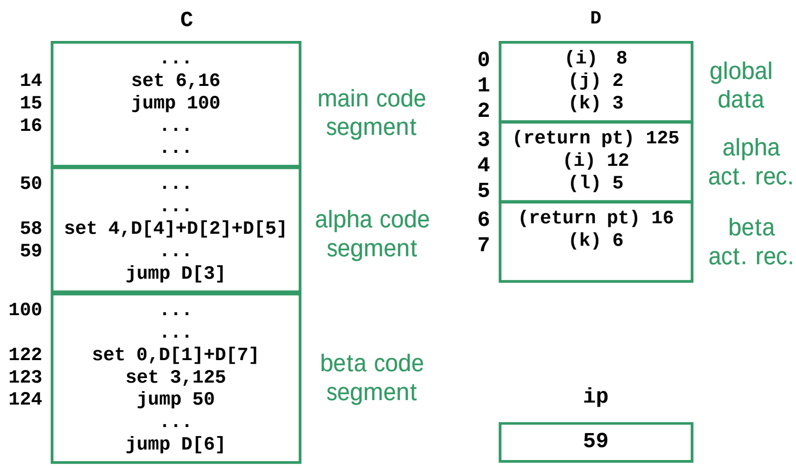
- Size of activation records
  is determined at translation time
- Each unit's activation records can be allocated before execution, i.e., with static allocation
- Thus each variable can be bound to a D memory address before execution
- No memory allocation overhead at run-time
- Might waste memory space,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

A C2 program

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

```
int i=1,j=2,k=3;
alpha()
{ int i=4,l=5;
  ...
  i+=k+l;
  ...
}
beta()
{ int k=6;
  ...
  i=j+k;
  alpha();
  ...
}
```

SIMPLESEM representation



Separate compilation for C2

```
file 1                file 2          file 3
extern beta();        extern int k;   extern int i,j;
int i=1,j=2,k=3;      alpha()         extern alpha();
main()                {               beta()
{ ...                   ...           { ...
  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: Supporting recursive functions

- C3 is derived from C2 adding two new features
- Direct and indirect recursion
  - routines can call themselves or can call another one, which in turn recalls them, in a recursive fashion
- Functions:
  routines can return values, as functions do

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
  - This might be difficult due to the deallocation of the AR when the routine is exited

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

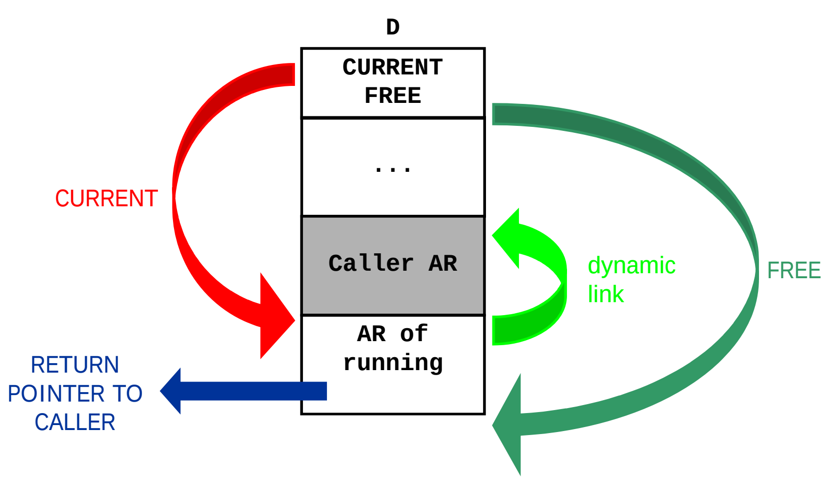
Solutions

- 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 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

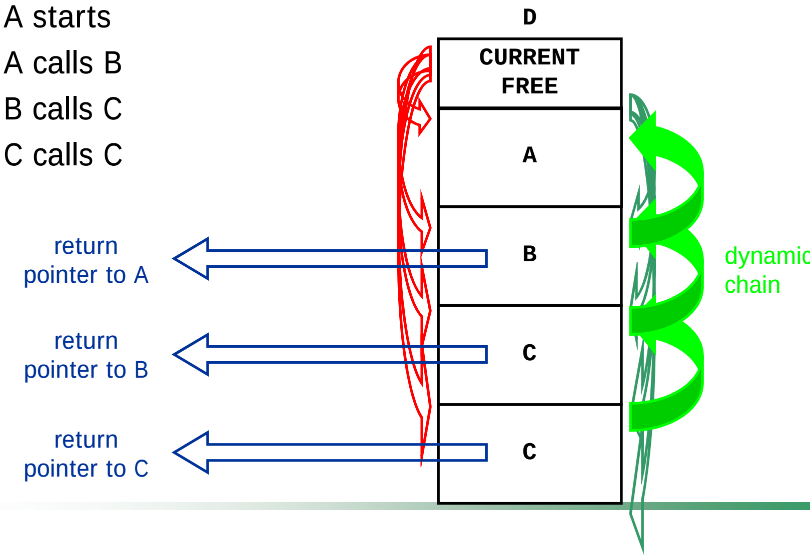
- The information on the caller changes
  - The caller can be one of the possible routine instances
- To make the return from an activation possible, information on the caller must be stored in the activation record
  - instruction to execute next (return pointer) [offset 0]
  - reference to the caller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

Solutions



Example

- A starts
- A calls B
- B calls C
- C calls C



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
```

- 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

```
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”);
}
```

SIMPLESEM representation: D memory

	D	
0	(CURRENT)	global data
1	(FREE)	
2	(n)	
3	(ret. Val.)	
4	(ret. pt)	fact activation record
5	(dyn link)	
6	(loc)	

SIMPLESEM representation of main()

```
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
```

SIMPLESEM representation of fact()

```
                             ;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]]
```

SIMPLESEM representation

	D	
0	(CURRENT) 4	global data
1	(FREE) 7	
2	(n) 3	
3	(ret. Val.)	
4	(ret. pt) 8	fact act. rec.
5	(dyn link) 2	
6	(loc)	

After the first call to fact()

	D	
0	(CURRENT) 12	global data
1	(FREE) 15	
2	(n) 3	
3	(ret. Val.)	
4	(ret. pt) 8	fact act. rec.
5	(dyn link) 2	
6	(loc) 3	
7	(Ret. Val.)	
8	(ret. pt) 21	fact act. rec.
9	(dyn link) 4	
10	(loc) 2	
11	(ret. Val.) 1	
12	(ret. pt) 21	fact act. rec.
13	(dyn link) 8	
14	(loc)	
15		

After the 3rd call to fact()

C4: Supporting block structure

- 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

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

Example

```
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
```

- 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

Compound statement

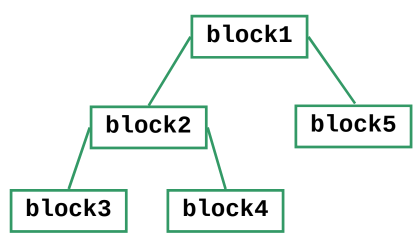
- A compound statement defines a lifetime of locally declared data
- Memory space is bound to a variable when the block in which it is declared is entered during execution
- The binding is removed when the block is exited

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`

C4”: Nesting routines

```
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
```

- Routine may be declared within another routine
- f3 can be called only 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

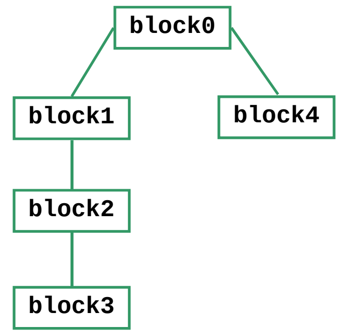
- C and C++ support only the nesting of compound statements
- Pascal and Modula-2 allow the nesting of routines
- Ada allows both

C4”: Nesting routines

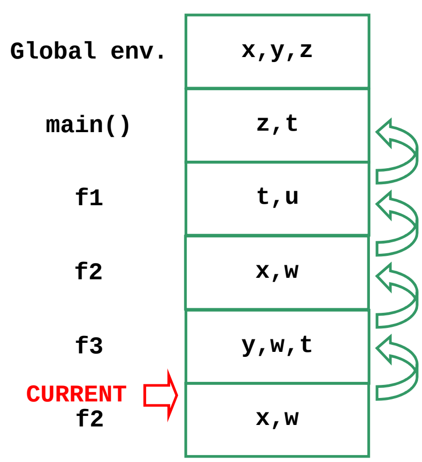
```
int x, y, z;
f1 ()
{//block 1
  int t,u;          //1
  f2()
  {//block 2
    int x, w;       //2
    f3 ()
    {//block 3
      int y, w, t;  //3
    }//end block 3
    x = y+t+w+z;
   }//end block 2
}//end block 1

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

- The Static Nesting Tree



A sketch of the runtime stack

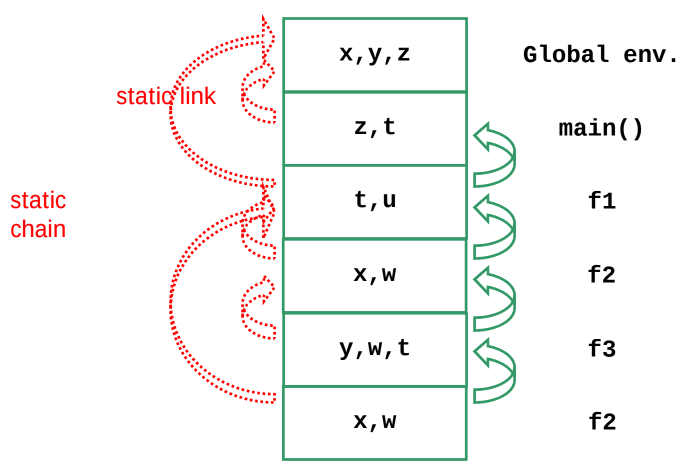


- Suppose we execute x=y+t+w+z in f2
- The binding of x and w is obvious
- What about t, y, and z?
- Dynamic
  - y and t to f3
  - z to main
- But, the binding of t, y, and z should follow the static rules

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

A sketch of the runtime stack



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
- 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 d^{th} 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]
  - Es: fp(0) = D[0], fp(1)=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 #
                     ;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: Toward 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

```
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 for the array dimensions
- The array object is allocated on top of the newly allocated activation record
- Access to array is performed indirectly through the pointer to the array base location

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

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

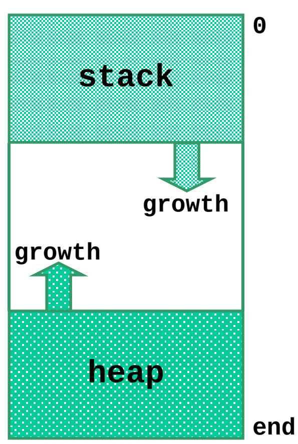
C5”: fully dynamic allocation

- Data can be allocated explicitly through an executable allocation instruction
- For instance, in C++, we can declare and allocate the following structure for a binary tree element:

```
struct node {
   int info;
   node* left;
   node* right:
};
node* n = new node;
```

- We cannot allocate such data on the stack, as done for automatically allocated data, we use a “heap”

C5”: fully dynamic allocation



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
- With dynamic typing, a variable in the activation record is represented by a pointer to the data object in the heap (size can change dynamically)
  - **Also the variable descriptor is kept in the heap**
- Dynamic typing requires dynamic type checking and policy for size changes
- With dynamic scoping, the dynamic chain supports access to nonlocal objects

Dynamic scoping

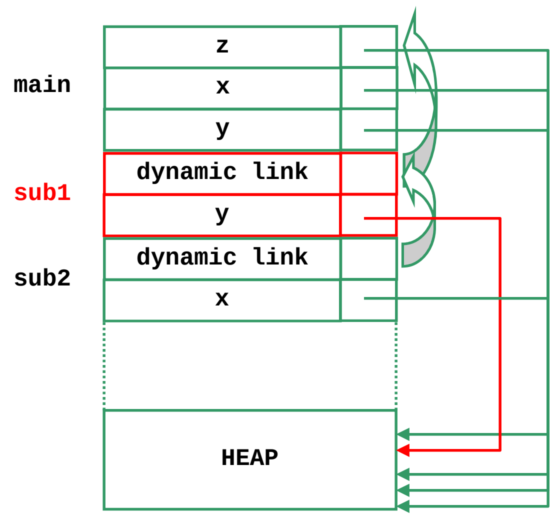
```
sub2()
{
  declare x;
  ...
  ... x ...;
  ... y ...;
}

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

Runtime view

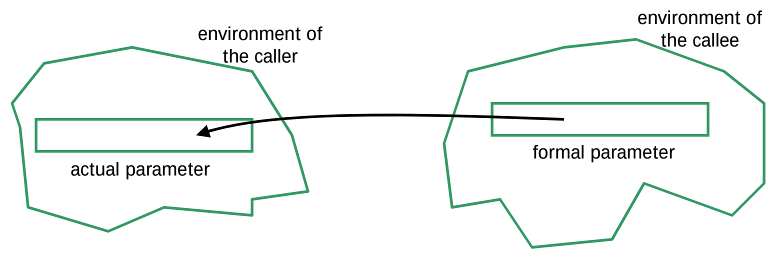


Parameter passing

- Data parameters
  - by reference or by sharing
  - by copy
  - by name
- Routine parameters

Call by reference

- ...or call by sharing
- Caller passes the address of the actual parameter
- Formal parameter are reference to the location of the actual parameter



Call by reference in C4

- The activation record contains one cell for each parameter
- Suppose a parameter is described as <d,o>
- The caller initializes the content of the cell with the address of the actual parameter

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

- If 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 copy

- 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:
  - Two formal parameters become aliases
  - A formal parameter and a nonlocal variable are aliases

By value-result vs. By reference

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

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

The two parameters are 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], then 11 is copied in the same cell. Therefore a[i]= a[j]=11

By value-result vs. By reference

```
goo(x)
{ ...
  a=1;
  x=x+a;
}
a = 10;
...
goo(a);
```

The formal parameter and the non-local variable are aliases

- By reference
  - when returning a=2
- By value-result
  - when returning a=11

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

Call by name

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

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

// after i=4 and a[4]=3
// a[3] is unaffected
```

Call by name

- The actual parameter belongs to the referencing environment of the caller (and not of the callee)

```
int c;  // global
swap(int a, int b)
{
  int temp;
  temp = a; a = b;
  b = temp; c++;
}
y()
{
  int c,d;
  swap(c,d);
}
```

- when y() is called the replacement rules specifies different meanings of c
- call-by-name easily 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

Parameter passing

- FORTRAN, call by reference
- ALGOL 60, call by name is the stand, call by value also possible
- SIMULA 67, call by value, call by reference, and call by name
- C++, Pascal and Modula-2, call by value and call by reference
- C, call by value, call by reference is explicit via pointers
- Ada parameter passing based on the intended use, in (for input parameters), out (for output parameters), inout (for input/output parameters), in is the default

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

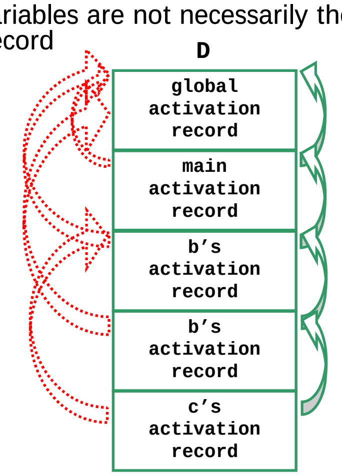
- Information to pass to the callee:
  - reference to routine's code
  - routine's nonlocal environment (static link, SL)
- Two cases:
  a) actual routine parameter is within the caller's scope
  b) actual routine parameter is a formal that was passed to the caller

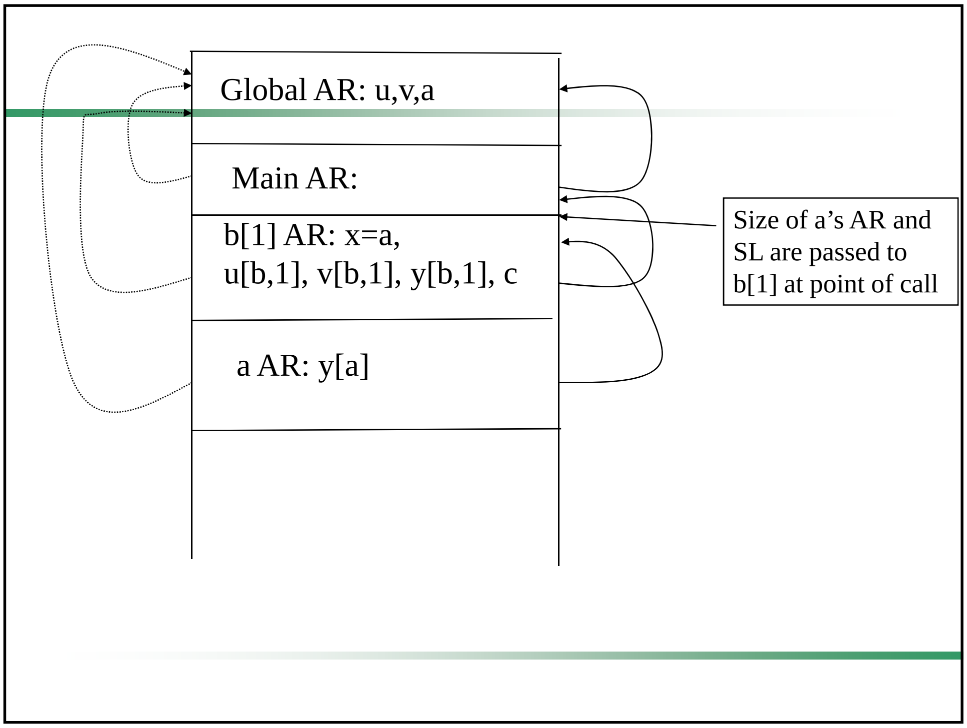
a) The static link is a pointer to the activation record that is d steps away along the static chain originating in the caller (d, distance between the call and the routine's declaration)

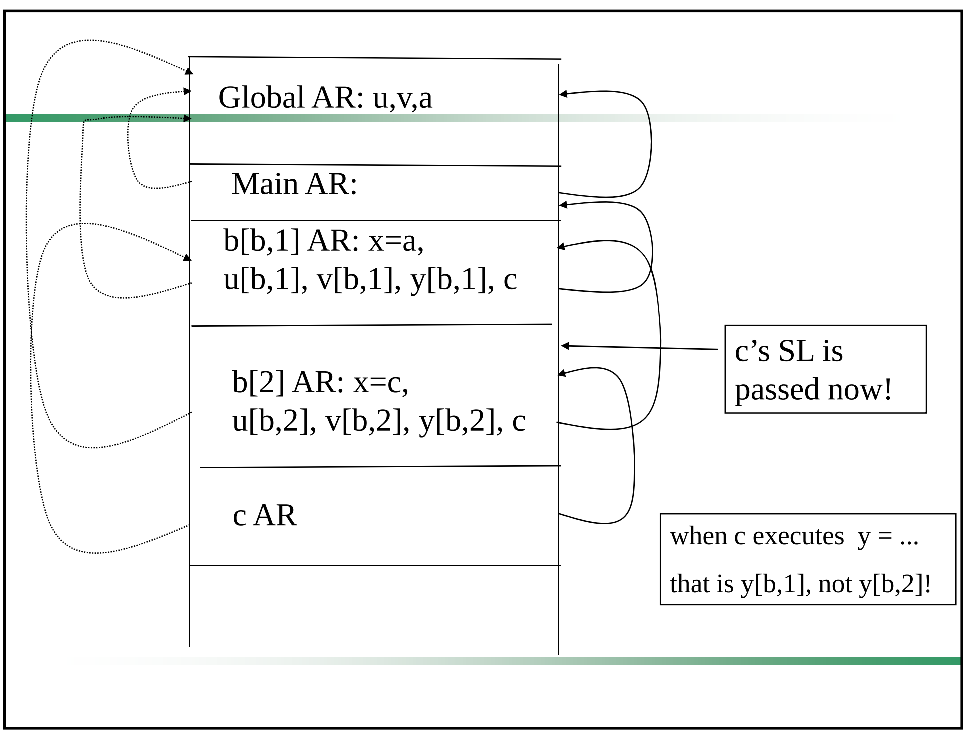
b) The static link is the one that was passed to the caller

Routine parameters

- Non local variables are not necessarily those of the latest activation record




Global AR: u,v,a
Main AR:
b[1] AR: x=a,
u[b,1], v[b,1], y[b,1], c
a AR: y[a]
Size of a's AR and
SL are passed to
b[1] at point of call


Global AR: u,v,a
Main AR:
b[b,1] AR: x=a,
u[b,1], v[b,1], y[b,1], c
b[2] AR: x=c,
u[b,2], v[b,2], y[b,2], c
c AR
c's SL is
passed now!
when c executes y = ...
that is y[b,1], not y[b,2]!

Summary lesson from the history:

- Most “practical” languages have powerful and general semantic features
  - parameter passing
  - dynamic typing/scoping/allocation
  - higher level constructs (routines as parameters), etc.
- Such power and generality often leads to semantic intricacies
- We must know and master such intricacies
- We must be able to detect intricacies and to analyze tricky programs
- We must avoid abusing such intricacies in program synthesis
- E.g., deep levels of nesting, tricky aliases, complex correspondences of procedure parameters, etc.