

CENG444: Run-time Organization Parameter Passing

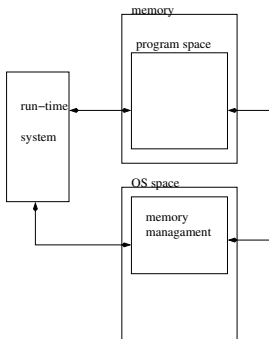
Cem Bozşahin

Cognitive Science Department, Informatics Institute,
Middle East Technical Univ. (ODTÜ), Ankara

[Feel free to share for nonprofit use only]

- Dynamic aspects of language processing need to be dealt with during execution: local names, implicit flow of control, storage organization, etc.
 1. storage organization (run-time stack management for flow of control)
 2. storage allocation (static or dynamic layout of program/data space)
 3. Run-time counterparts of SL constructs (nesting, block structure)

4. Access to names (local/nonlocal, lexical/dynamic scoping)
5. Parameter passing
6. symbol-table access and management



functional division of labor
for run-time, executable code and OS

- Translating IC declarations to storage layout within a block:

Relative addressing within a block

- base address of the block
- offset from base
- amount of space (can be deduced from type of the symbol)

```
ex: struct symtab {  
    char *name;  
    struct typenode *type;  
    int blockno;  
    int addr;};  
  
func p ();  
var a:  int; b:real; c:int
```

assuming one word for integers and two words for reals:

	p
0	a
1	b
3	c

syntax-directed way of calculating storage:

```
P -> proc ID { blockno++; offset=0} (Params) Decl Body  
{ ...}
```

```
Decl -> Decl ; D | D { ... }
```

```
D -> ID : T { enter(ID.val,T.type,blockno,offset)  
              offset=offset+T.width}
```

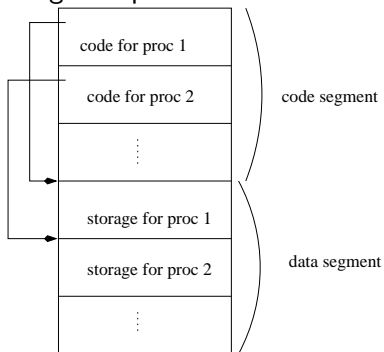
```
T -> int { T.type=INT; T.width=2}
```

```
T -> array [num] of T { T.type=array(num.val,T1.type)  
                        T.width=num.val * T1.width}
```

Records can be treated like procedures as far as storage layout is concerned.

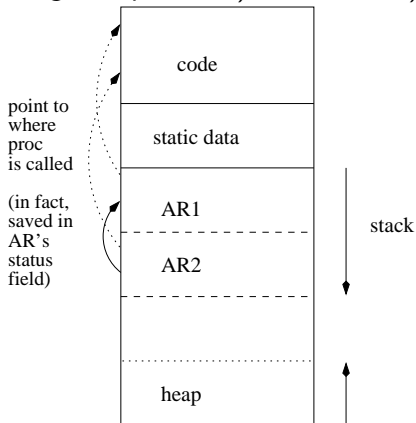
- Run-time management of storage
 - lifetime of variables
 - parameter passing
 - function call/return
 - dynamic storage

Program space when there is no recursion



FORTRAN's storage allocation
(predefined storage for every proc)

Recursion: keep a stack of Activation Records (AR)
Program space is 1) static, and 2) dynamic in two directions.

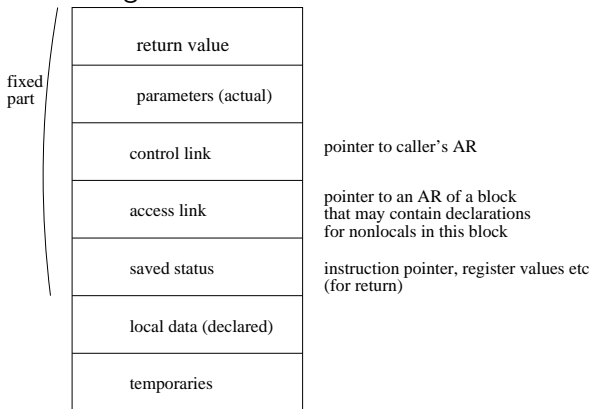


- The structure of AR

fixed part: things that can be located by the caller without having to know the internal structure of the callee.

block-dependent part: procedural-specific stuff (local vars etc.)

- Why does the caller need the fixed part? Some things must be put in there *before* control is transferred to the callee, and some things must be accessible *after* the callee returns.



the structure of a single activation record

- Parameter passing mechanisms:
- Formal arguments and actual arguments must be associated during run-time to do parameter passing.
- Declaration of formal arguments allow space allocation of definite size in the callee's AR

1. call-by-value: Caller evaluates and puts values of arguments in callee's AR
2. call-by-reference: caller puts addresses of actual arguments in callee's AR (r-value of formal parameter is l-value of actual parameter)
what if actual argument is an expression with no l-value?
3. copy-restore (value result, mixture of 1 & 2): caller evaluates actual parameters and passes r-values to callee. Caller saves their l-values. Upon return, it restores the r-value of formal parameters to l-values of actuals.
4. call-by-name: literally substitute actuals for the formals. Local names in callee that clash with names in actuals are renamed.

l-value of actuals are passed in 2 and 3; r-value in 1, and the text itself in 4.

ex: a hypothetical language whose parameter passing is left unspecified

```
proc swap(x,y)
    temp:=x;
    x:= y;
    y:= temp;
endproc;
```

assume we make the call `swap(i,a[i])`

in call-by-value, no change in actuals

in call-by-reference, l-values of `i`, `a[i]` are passed on. `x,y` indirectly refers to the same l-value.

in copy-restore, r-values of `i`, `a[i]` are passed on, but on return, `x,y`'s r-values are put in l-values of `i`, `a[i]`

in call-by-name, i is substituted for x , and $a[i]$ for y :

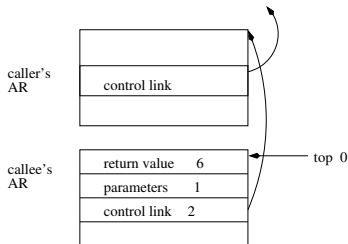
```
temp := i;
```

```
i := a[i];
```

```
a[i] := temp
```

if l_0 is initial value of i , $a[a[l_0]]$ becomes l_0 , not $a[l_0]$.

- Argument passing shows that there are some things that the caller must pass on to the callee before the call,
- and return values of the callee must be accessible to the caller after the call is over.
- How is this done?



CALLER

0. push control stack (enw entry for callee's AR)
1. evaluate actual parameters and put them in callee's AR (they are a fixed distance away from caller's AR)
2. set the control link of the CALLEE to point to the CALLER (it is also fixed distance away from caller's AR)
3. save next instruction in CALLEE's machine status (also fixed distance away)
9. caller retrieves return value (although stack is popped, it is still fixed distance away from the caller's AR)

CALLEE

4. save machine status
5. initialize and execute
- 6.. put return value in place
7. restore machine status and reset instruction pointer to caller's next stmt (was saved as part of machine status in 3)
8. pop control stack

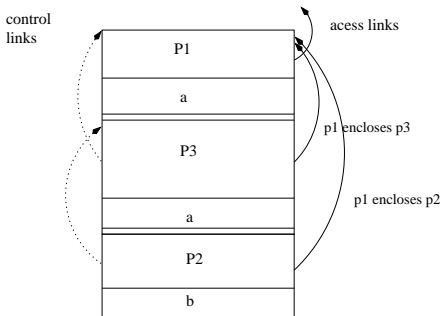
- Why do we need Access links in addition to control links?

In *dynamic scoping*, the calling sequence determines the environment. Since calling sequence is shown by control links in ARs, there is no need for extra links.

In *lexical scoping*, textual enclosures in the program determine the environment. But this enclosure is not necessarily the same as calling sequence (hence we need extra information)

```
proc p1 ()  
    var a;  
    ...  
    proc p2 ()  
        var b;  
        begin  
            x := a +b ;  
        end  
    proc p3()  
        var a;  
        begin  
            p2();  
        end
```

- Access link: if q is nested immediately in p (textual enclosure), q 's access link points to the *most recent activation* of p (and there must be one, why?)



Looking for globals in lexical scope: if not in current AR, follow the access link

- Accessing the access links: if a proc at nesting depth n_p refers to a nonlocal a at n_a :
- $n_a \leq n_p$ due to lexical scoping. Follow $n_p - n_a$ number of access links to reach the AR of a 's definition block. This value can be computed at compile time (for each nonlocal in every block)
The tuple $(n_p - n_a, a\text{'s offset})$ defines the address of any nonlocal

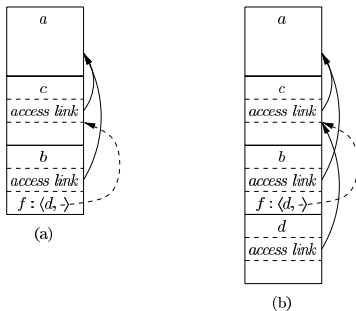


Figure 7.13: Actual parameters carry their access link with them

a calls c; c is declared within a

c calls b

b calls d; d is declared within c

This idea works even if arguments are functions, such as f in the figure:

```
fun a(x) =  
  let  
    fun b(f) =  
      ... f ... ;  
    fun c(y) =  
      let  
        fun d(z) = ...  
        in  
          ... b(d) ...  
        end  
      in  
        ... c(1) ...  
      end;  
  end;
```

Figure 7.12: Sketch of ML program that uses function-parameters

- Setting up the access links: assume p calls x :

if $n_p < n_x$, x is nested in p . Access link of x points to the access link in AR of the caller (second from top in stack)

if $n_p \geq n_x$, x is not declared within p (due to property of lexical scope). the procedures at nesting depths $1, 2, n_x - 1$ must be the same (again due to lexical scope). $n_p - n_x$ number of links reach the level of x again, hence $n_p - n_x + 1$ links reach the block that most closely encloses p and x . This can be computed at compile time as well.

- If nesting is deep and calling sequences are not hierarchical, this method causes a lot of hopping along access links.
- One alternative is to use the *display* mechanism.
- Intel chips have hardware implementation of *displays*.

- displays make use of the fact that, in lexical scope, two procedures at same nesting depth cannot refer to each other's bindings as nonlocal reference. So, it is possible to use only one storage for any nesting depth n .

$d[n]$ is an array of AR pointers, it's value is set during calls and returns. $d[i]$ points to the AR of the most current activation of active procedure at level i .

When a new AR for a proc at level i is set up

1. save the value of $d[i]$ in the new AR (later to be restored to this value upon return)
2. set $d[i]$ to point to the new AR

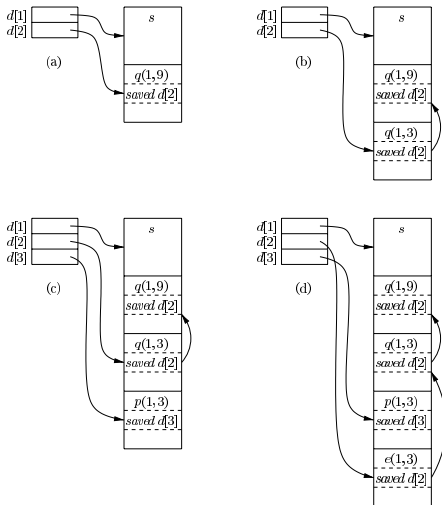


Figure 7.14: Maintaining the display

The figure refers to these functions:

```
1) fun sort(inputFile, outputFile) =  
    let  
2)      val a = array(11,0);  
3)      fun readArray(inputFile) = ...  
4)          ... a ... ;  
5)      fun exchange(i,j) =  
6)          ... a ... ;  
7)      fun quicksort(m,n) =  
          let  
8)          val v = ... ;  
9)          fun partition(y,z) =  
10)              ... a ... v ... exchange ...  
          in  
11)              ... a ... v ... partition ... quicksort  
          end  
    in  
12)      ... a ... readArray ... quicksort ...  
    end;
```

Figure 7.10: A version of quicksort, in ML style, using nested functions

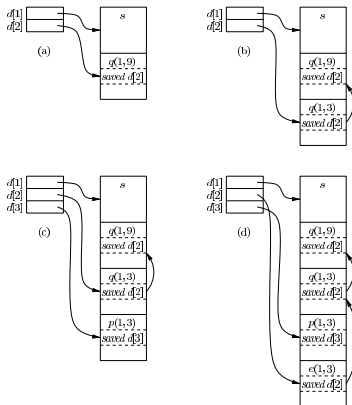


Figure 7.14: Maintaining the display

```

1) fun sort(inputFile, outputFile) =
    let
2)       val a = array(11,0);
3)       fun readArray(inputFile) = ...
4)         ... a ... ;
5)       fun exchange(i,j) =
6)         ... a ... ;
7)       fun quicksort(m,n) =
            let
8)           val v = ... ;
9)           fun partition(y,z) =
10)            ... a ... v ... exchange ...
11)            in
                ... a ... v ... partition ... quicksort
            end
12)        in
            ... a ... readArray ... quicksort ...
        end;

```

Figure 7.10: A version of quicksort, in ML style, using nested functions

- Why does this scheme work? let's say p calls q:
 - if $n_p < n_q$ then $n_q = n_p + 1$ because otherwise q will not be visible to p. This means that there is no need to change the first n_p entries in the display.
 - if $n_p \geq n_q$ then $d[i]$ entries for $i = 1, 2, \dots, n_q - 1$ are the same. Save $d[n_q]$ and reset.

- Lexical scoping makes most of the procedure interactions predictable at compile time.
- Access to nonlocals can be fixed at compile time too.
- Dynamic scoping puts the burden on run-time (though nonlocal management is trivial thru ARs).
- Dynamic scoping seems better suited to domains of computations in which a procedure is expected to behave differently depending on who calls it (as in some AI applications).