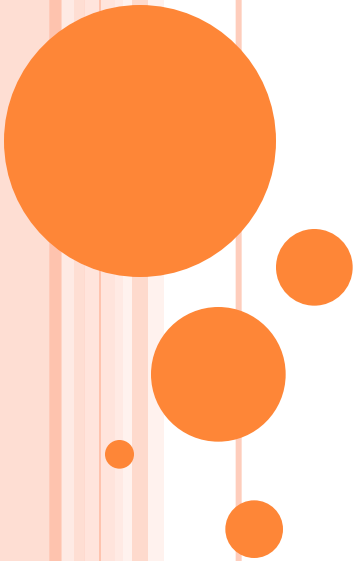
Block Structured Languages



Run-Time Major Structures

- **Soft structure**

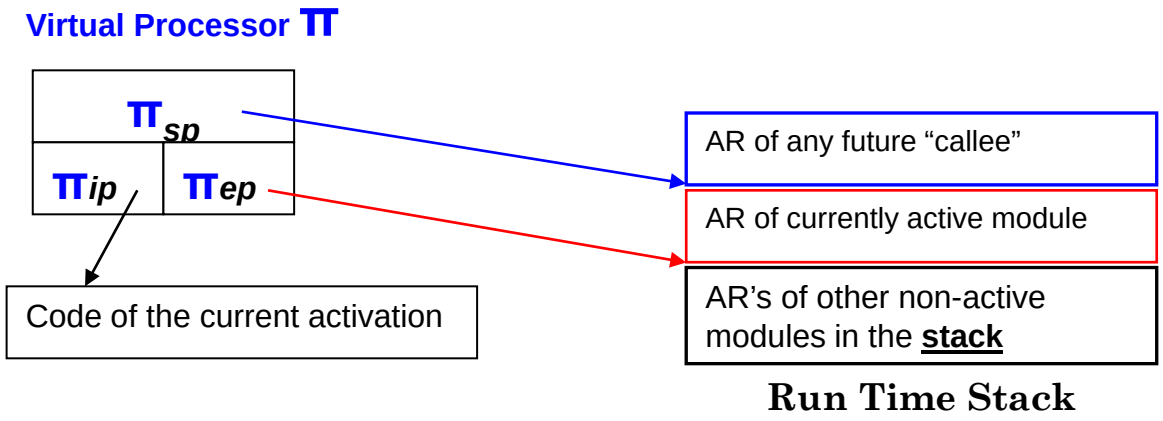
- **Hardware structure**

Soft structure

- **Activation state with the following two parts**
- **Fixed Part:** the machine code of the program
- **Variable** Part: The *activation record* (AR) that holds the state of the "activation" (procedure/function in execution).
  - environment part (ep): defines the context of an activation; it consists of:
    - locals and formal parameters.
    - **Static Link (SL):** pointer to all visible non-local accessible scopes (starting at the AR of the definer of the activation) of the current activation.
  - Instruction Pointer (IP): pointer into the current activation code.
  - Dynamic Link (DL): a pointer to the caller's AR, where the return address is stored.

HARDWARE STRUCTURE

- Environment Pointer (πep): a pointer to the current activation environment (its AR).
- Instruction Pointer (πip): a pointer to the next instruction to be executed.
- Stack Pointer (π_{sp}): a pointer to the next AR record to be placed at the top of the stack, in case of the current "activation" calls a procedure or function.



Name Access (local/non-local) 1

- **Static Chain:** a chain starting from the current AR' SL until the main program AR, following the SL of all intermediate ARs.

- **Dynamic Chain:** a chain starting from the current AR' DL until the main program AR, following the DL of all intermediate ARs.

Static Nesting Level (SNL)

- **Static Nesting Level (*snl*)**

  - the ***snl*** of a name use or declaration is the number of surrounding contour diagram boxes around its use or declaration, respectively.

NAME ACCESS (LOCAL/NON-LOCAL) 2

- The **static distance** (***sd***) of a name is:

$$sd = snl_{\text{use}} - snl_{\text{decl}}$$

- Remember: for **locally** declared names, ***sd* = 0**

- The compiler will maintain a symbol table that keeps a record of properties (e.g., type, snl_{use}, snl_{def}, ***sd***, and "offset" within the local declarations of its definer, etc.) for each name; this will help at compile time for type checking and run-time name access (see page 216).

Question

Knowing that for any used name in the program, its $\boldsymbol{snl_{use}}$ must be greater than or equal to its $\boldsymbol{snl_{def}}$ (i.e., its ***sd*** is positive)**,** is it a sufficient condition to have positive ***sd*** for a name to be used?

Question

Knowing that for any used name in the program, its snl_{use} must be greater than or equal to its snl_{def} (i.e., its ***sd*** is positive), is it a sufficient condition to have positive ***sd*** for a name to be used?

Answer:

No, the name has to also be **visible**

Accessing Non-Local Names: (static scoping) - 1

- ***offset*** available in the symbol table,
- Compiler computes the ***sd*** of every non-local name,
- then it generates machine code to carry out the following at run time:

  a) To locate the AR of the environment of definition of the non-local name, traverse the static chain ***sd*** times in order to get to the defining module's AR of the target non-local name.

  b) Add the name's offset (stored in the symbol table) to the obtained address of the AR of the name defining activation (obtained in "a" above).

Accessing Non-Local Names: (static scoping) - 2

AP := M[*πep*].SL;

traverse first link of the static chain to the AR of the definer of the current activation.

(*sd* – 1) { AP := M[AP].SL};

traverse remaining links of the static chain, up to the AR of the definer of the non-local.

***fetch* M[AP + offset];**

offset with the AR of the definer to get to the non-local name.

Accessing Non-Local Names: (static scoping) - 3

- Total number of memory references is "***sd***+1",
  - expensive execution time in case of deeply nested modules.

Question:

Discuss the above concern as a language designer, and as a programmer.

Accessing Non-Local Names: (static scoping) - 3

- Total number of memory references is "***sd***+1",
  - expensive execution time in case of deeply nested modules.

- Question: Discuss the above concern as a language designer, and as a programmer.
- The language designer will investigate the language environment, if users tend to have **deeply nested modules**, and then think of other mechanisms than the "static chain".
- If the programmers are using a language with "static chain", then they should not deeply nest proc/functions, for their program to execute faster!

Accessing Non-Local Names: (static scoping) - 4

Question

What is the scenario that makes the static and dynamic chains the same?

Accessing Non-Local Names: (static scoping) - 5

Question

What is the scenario that makes the static and dynamic chains the same?

When the callers and the definers are the same.

Procedure "Call" Sequence with Static Chain: (Recursive Procedures) 2

$M[\pi_{sp}]$.PAR[1] := evaluate (actual-parameter[1]);
$M[\pi_{sp}]$.PAR[2] := evaluate (actual-parameter[2]);
...
$M[\pi_{sp}]$.PAR[n] := evaluate (actual-parameter[n]);

$M[\pi_{ep}]$.IP := π_{ip}; save the **resume** address into the caller's AR

$M[\pi_{sp}]$.DL := π_{ep} ; set the dynamic link in the callee's AR

sd * (π_{ep} **:=** $M[\pi_{ep}]$**.SL)** ; get to the environment of the callee's definer

$M[\pi_{sp}]$.SL := π_{ep} ; set the static link of the callee

Procedure "Call" Sequence with Static Chain: (Recursive Procedures) 2

π_{ep} **:=** π_{sp} ; set the virtual processor EP to the callee AR

π_{sp} := π_{sp} **+ size** (callee's AR) ; set the hard virtual processor stack pointer ready for the next AR allocation for any future proc/function call!

π_{ip} **:=** entry address of the callee's code

; transfer the control to execute at the callee's code by the proper setting of the virtual processor's IP.

resume (return address): (*house keeping code*);

Procedure "Return" Sequence with Static Chain

- π_{sp} := π_{sp} – **size-of** (callee's AR)
  ;"pop" the callee's AR from the stack

- π_{ep} := M[π_{ep}].DL

; reactivate the virtual processor back into the caller

- π_{ip} := M[π_{ep}].IP
  ; goto the resume address and start executing at the caller's code

Procedural Parameters Are Represented by "Closures"

- We know the values of an integer or real type, but for the type "procedure/function" the question is: what is the state (value) of such type, and how to represent it?

The answer is the **closure**

- A closure is a pair <ip,ep> Where:
- ***ip*** is the code instruction pointer
- ***ep*** is a pointer to the environment of the definition of the procedure/function that is represented with such closure.

Procedural Parameters Are Represented by "Closures"

- The compiler will evaluate a ***closure*** for every actual procedure parameter, (or locally declared name); and passes it to its corresponding formal, in the caller activation record (or keep it in the activation record of the host of declared proc/func).

- Then, inside the callee, if such formal parameter procedure (function) is called, the compiler will generate the following machine code calling sequence:

Calling Formal Parameter Procedure (F_P) 1

$M[\pi_{sp}].PAR[1] := evaluate\ (actual\text{-}parameter[1]);$

$M[\pi_{sp}].PAR[2] := evaluate\ (actual\text{-}parameter[2]);$

...

$M[\pi_{sp}].PAR[n] := evaluate\ (actual\text{-}parameter[n]);$

$M[\pi_{ep}].IP := \pi_{ip}$; save resume address into the caller's AR

$M[\pi_{sp}].DL := \pi_{ep}$; set the dynamic link in the callee's AR

Calling Formal Parameter Procedure (F_P) 2

$M[\pi_{sp}].SL$:= fp.ep ; fill in the SL of fp's AR with the environment pointer that is extracted form the fp's closure

π_{ep} := π_{sp} ; set the virtual processor π_{ep} to the callee AR

π_{sp} := π_{sp} + size-of (callee's AR) ; set the hard virtual processor stack pointer ready for the next AR allocation for any future proc/function call, while in fp!

π_{ip} := fp.ip ; go to the code of fp

Notice that we assumed that the fp's closure exists and we used it, but at some point the compiler has to build it as follows:

How to Build the Closure (for procedure p)?

$M[\pi_{sp}]$.PAR[i].ip := entry (*p*); build the *ip* part of the closure of procedure *p*

π_{AP} := $M[\pi_{ep}]$.SL ; traverse the first static link

(*sd-1)* * (π_{AP} := $M[\pi_{AP}]$.SL) ; get to the environment of p's definition

$M[\pi_{sp}]$.PAR[i].ep := π_{AP} ; build the ep part of the closure of procedure *p*

See page 226, figure 6.5 for examples of formal procedure parameters.

Functions/procedures As First Class Citizens(FCC)

Until now we passed unctions/procedures as parameters;

What about returning them as values from other functions (i.e., treating them as FCC)?

Functions/procedures As First Class Citizens(FCC)

```
1.  Program Test;
2.    type
3.    fun : function (integer): integer;
4.    var
5.     m: integer;
6.     h, incr, sqr : fun;
7.     function Compose  (f, g : fun): fun;
8.     function  apply_copm (x : integer): int;
9.        begin
10.         return (f (g (x) ) );
11.       end;
12.     begin (* Compose *)
13.       return (apply_comp);
14.    end (* Compose *);
15.   begin (* Test*)
16.     h := Compose (incr, sqr);
17.     m := h (3)                    (*  m = 10 *)
18.   end. (* Test*)
```

Functions/procedures As First Class Citizens(FCC)

- If we follow the run time stack dynamic formation, we will find out that upon exiting from Compose line 16, there will be a closure <ip,ep>$_{apply\_comp}$

- With its ep pointing to its definer "Compose", but we already exited from Compose! Hence, there is a problem of an orphan "ep"!!!!!!!!!!!!!!!!!

Functions/procedures As First Class Citizens(FCC)

- To solve the above problem, we must give up the "stack" as a model of computation, and use a "**heap**", where we can keep an activation record even after we exit its module!

- Thus, for languages that return functions/procedures from other functions, we can not use the "stack" as a model of computation, at run time; instead we use a "heap".

Retention

- It is the ability to retain the AR of a procedure/function, even after exiting its code. Such retention cannot be achieved in languages that utilize the "stack" as their model of computation. It is used in languages that use "heap" as their model of computation.

Question

Would the feature of “retention” violate the static scoping rules manifested by the contour diagram semantics ?

Question

Would the feature of "retention" violate the static scoping rules manifested by the contour diagram semantics?

Answer

Yes! A deeply nested function can be returned to a higher ***snl***, exposing it to be called by a lesser ***snl*** procedure!