Runtime Environments

- General characteristics of code generation: uniform for a wide variety of architectures

- Runtime environment = structure of target machine for
  - management of memory
  - control of execution

- Environment
  - fully static (FORTRAN77)
  - stack-based (C, Pascal, Ada)
  - fully dynamic (Lisp)

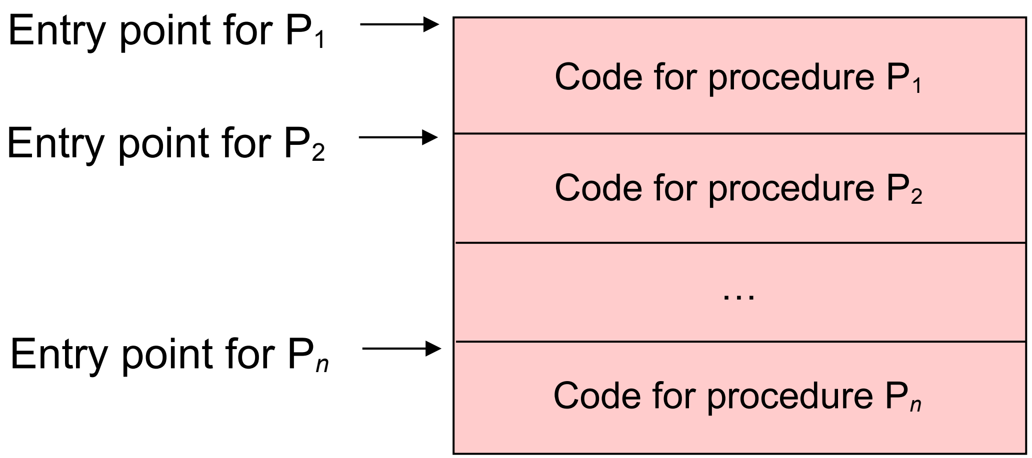
- Compiler: maintains the environment only <u>indirectly</u> → generation of specific code

- Interpreter: maintains the environment <u>directly</u> within its data structures

Memory Organization in Program Execution

- Memory
  - registers
  - RAM
    - code — fixed before execution, invariant ⇒ code addresses computable statically
    - data

- Code area: entry point of each subprogram known statically



Memory Organization in Program Execution (ii)

- Data allocation: only a small subset allocated statically (global/static variables)

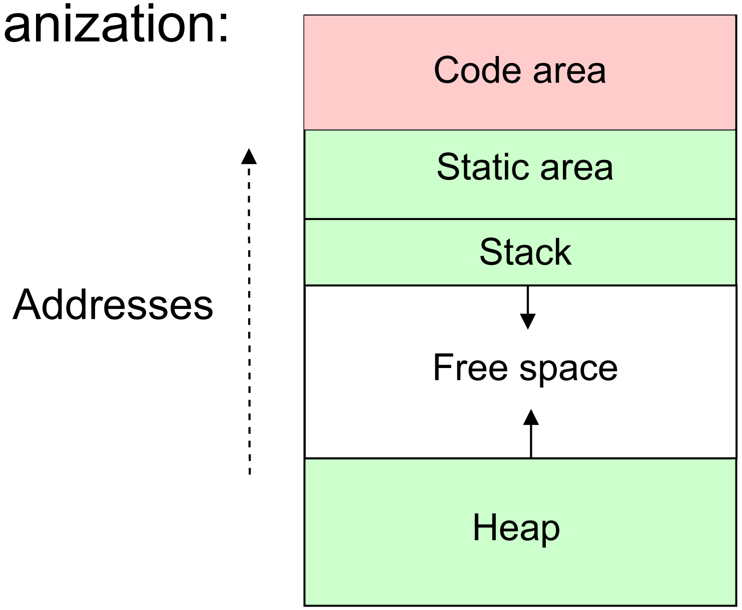
- Constant allocation
  - small → inserted directly into code by compiler
  - otherwise → allocated in static area

```
const int TOT = 1000;
```

```
x = y + 12345;
```

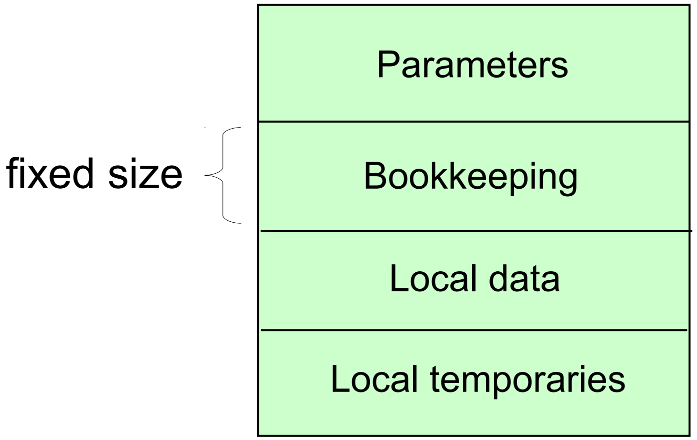
```
printf("Syntax error: %s\n", msg);
```

- Typical memory organization:



Activation Record

- Activation record = Important allocation-unit of memory $\rightarrow$ local data of called P



- $\forall$ P $\rightarrow$ space
  - fixed (bookkeeping)
  - varying
    - parameters
    - local data
      - fixed within single P
      - varying among $\neq$ P

- Allocation of AR: partially
  - automatic $\rightarrow$ processor (e.g: return address)
  - explicit $\rightarrow$ code generated by compiler (e.g: space for local temporaries)

Activation Record (ii)

- AR allocated in
  - static area (FORTRAN77)
  - stack (C, Pascal)
  - heap (Lisp)
- Registers = Part of runtime environment → storage of
  - temporaries
  - local vars
  - global vars
- Target machine with many registers →
  - static area
  - AR

  possibly maintained in registers
- Special-purpose registers → keep track of
  - execution
    - program counter (**pc**)
    - stack pointer (**sp**)
  - activation of P
    - frame pointer (**fp**)
    - argument pointer (**ap**)

Calling Sequence

- Specification of list of operations to execute when calling P = part of design of RTE

- Calling sequence: divided in
  - call sequence
    - Allocation of memory for AR
    - Computation and storage of actual parameters
    - Storage/assignment of registers necessary for the call
  - return sequence
    - Storage of return value
    - Readjustment of registers
    - Release of AR

- Design choices:
  1. Division of operations between caller / callee ⇨ how much code to place at point of call of P / beginning of execution of P
  2. To what extent (in call) rely on processor support / code explicitly generated

- Notes:
  - Minimal requirements for caller: computation and storage of actual parameters
  - State of registers + return address at call: saved by caller and/or callee
  - Assignment of bookkeeping data: cooperatively by caller/callee

Fully Static Runtime Environments

- Characteristics: static data (lifetime = life of program) $\rightarrow \nexists \left\{ \begin{array}{l} \text{pointers} \\ \text{dynamic allocation} \\ \text{recursion} \end{array} \right.$ (FORTRAN77)

  $\Downarrow$

  <u>all</u> variables allocated statically (not only globals)

  $\Downarrow$

  - Unique AR $\forall$ procedure (allocated statically <u>before</u> execution)
  - Every var (loc/glob) accessible by <u>fixed</u> address

- Structure of whole program memory:

main	Code area
$Proc_1$	
…	
$Proc_n$	
Global data	Data area
AR of main	
AR of $Proc_1$	
…	
AR of $Proc_n$	

Fully Static Runtime Environments (ii)

- Reduced bookkeeping data in each AR → return address only

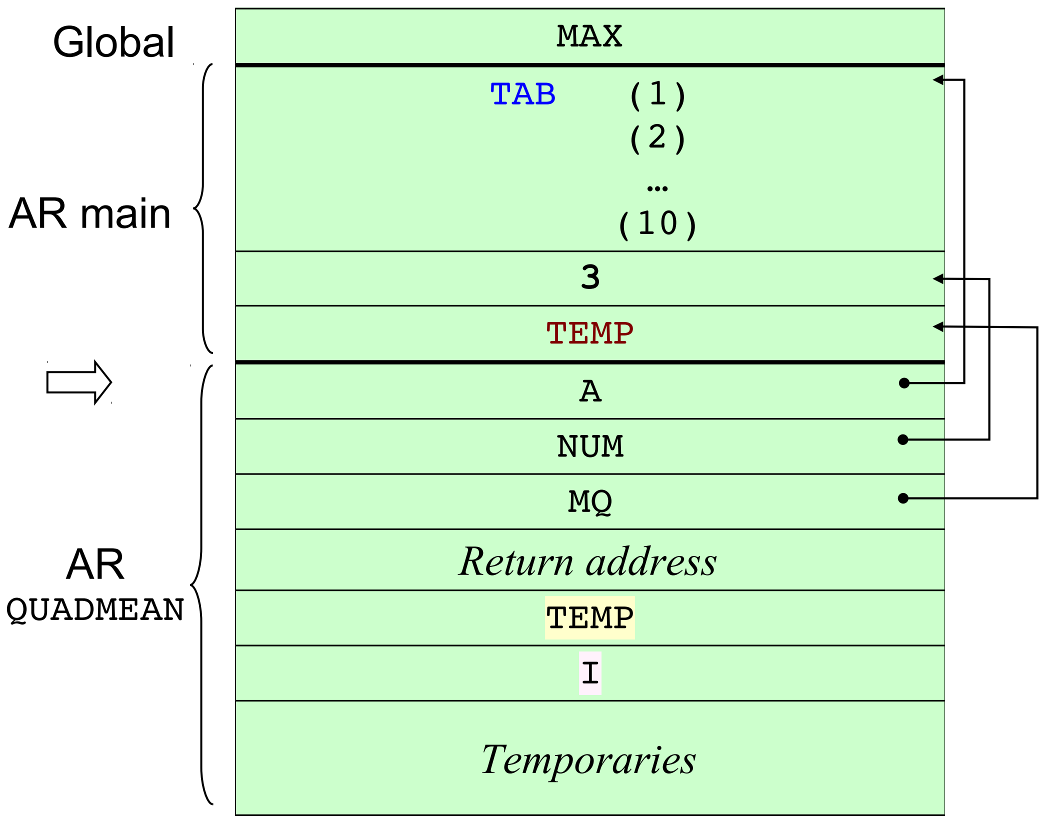
- Simple calling sequence:
  1. Computation/storage of actual parameters in AR of callee
  2. Saving of return address of caller
  3. Jump to address of callee
  4. At <u>return</u> → jump to return address

- Most hw architectures:
  - Call → Jump to subroutine: automatic saving of return address
  - Return → Automatic restoration of return address

Fully Static Runtime Environments (iii)

- Example (FORTRAN77: computation of quadratic mean of three numbers)

```
    PROGRAM P
    COMMON MAX
    INTEGER MAX
    REAL TAB(10), TEMP
    MAX = 10
    READ *, TAB(1), TAB(2), TAB(3)
    CALL QUADMEAN(TAB, 3, TEMP)
    PRINT *, TEMP
    END

    SUBROUTINE QUADMEAN(A, NUM, MQ)
    COMMON MAX
    INTEGER MAX, NUM
    REAL A(NUM), MQ, TEMP
    INTEGER I
    TEMP = 0.0
    IF ((NUM.GT.MAX).OR.(NUM.LT.1)) GOTO 99
    DO 10 I=1, NUM
       TEMP = TEMP + A(I)*A(I)
10  CONTINUE
99  MQ = SQRT(TEMP/NUM)
    RETURN
    END
```



- ○ Parameters passed by reference → need for dereferencing to access values →
  - Array parameters: <u>not</u> copied
  - Constant parameters: stored in memory and accessed like variables
- ○ Need for space for temporaries: `TEMP + A(I)*A(I)` `TEMP/NUM`

Stack-Based Runtime Environments

- L with ⟨ recursion / local vars allocated ∀ call ⟩ $\Rightarrow$ Impossible static allocation of AR

- AR allocated on runtime stack ⟨ push ← call of P / pop ← end of P ⟩ $\Rightarrow$ stack of ARs

- Possible several ARs of the same P at the same time (one ∀ call)

- Increase in complexity for ⟨ bookkeeping / access to vars ⟩ $\Rightarrow$ f(L) $\Rightarrow$ classification

Stack-Based Environments without Local Procedures

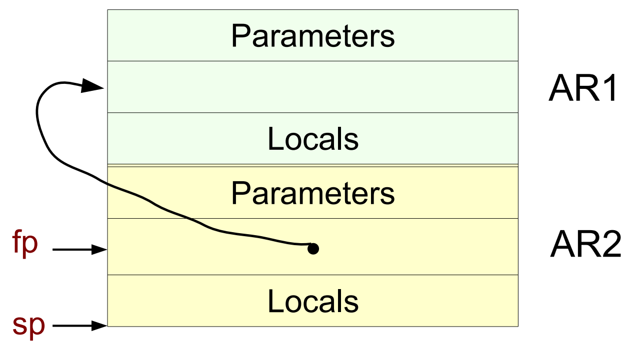
Requirements:

1. Access to local vars → pointer to current AR ≡ frame pointer (fp): normally in register

2. Restoration at end of callee → pointer to AR of caller ≡ control link

   Typically: allocated between areas ⟨ parameters / local vars ⟩ ⟹ pointing to location of control link of previous AR

3. Pointer to last location on stack ≡ stack pointer (sp)

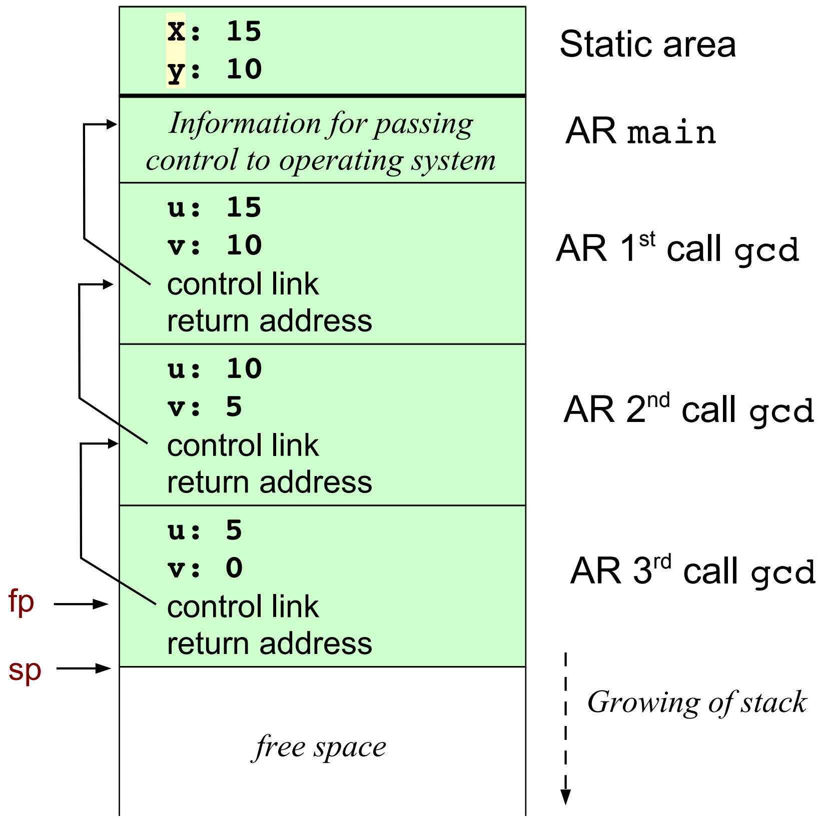


Stack-Based Environments without Local Procedures (ii)

Example 1: Greatest common divisor of two nonnegative numbers by Euclid's algorithm

```
#include <stdio.h>

int x, y;

int gcd(int u, int v)
{
  if (v == 0) return(u);
  else return(gcd(v, u%v));
}

main()
{
  scanf("%d%d", &x, &y);
  printf("%d\n", gcd(x, y));
  return(0);
}
```

`mcd(15,10)` $\rightarrow$ `mcd(10,5)` $\rightarrow$ `mcd(5,0)` $\Rightarrow$ 5

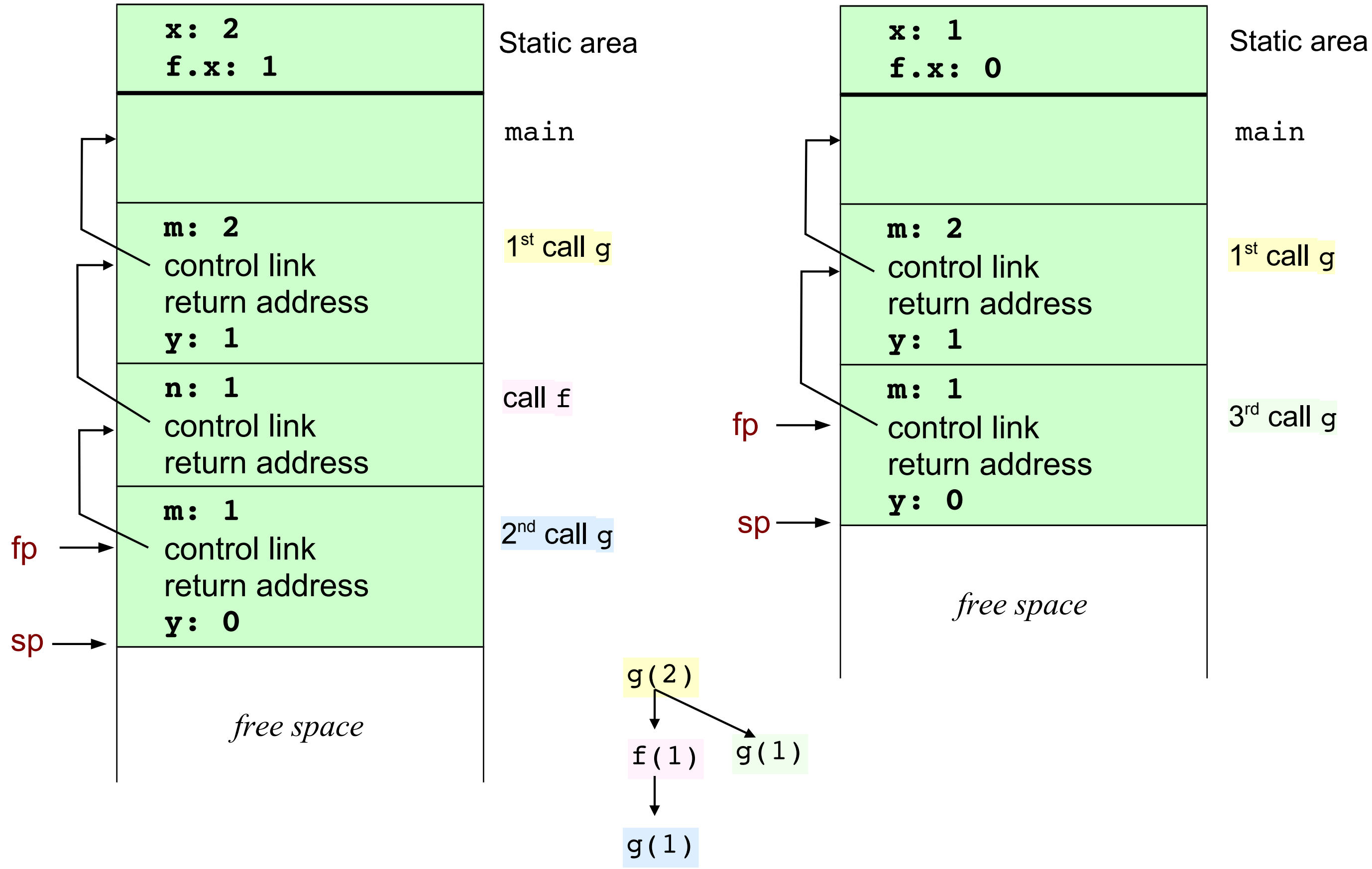


- New call $\rightarrow$ fp points to control link of new AR
- Execution of `printf` $\rightarrow \exists$ only AR of `main`
- Parameters passed by value $\rightarrow \nexists$ allocation of actual parameters in caller! *(unlike FORTRAN77)*

Stack-Based Environments without Local Procedures (iii)

Example 2:

```
int x=2;
void g(int);

void f(int n)
{
  static int x=1;

  g(n);
  x--;
}

void g(int m)
{
  int y=m-1;

  if (y>0)
  {
    f(y);
    x--;
    g(y);
  }
}

main()
{
  g(x);
  return(0);
}
```

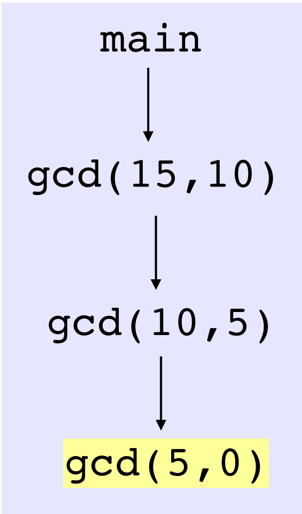


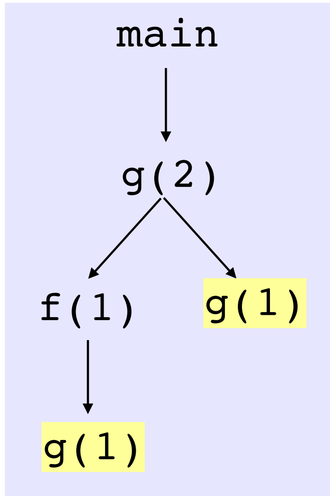
Note:

- AR 3rd call `g` → overwrites space before occupied by AR(`f`)
- Static var `f.x`: not allocated in AR(`f`) → area of global variables

Stack-Based Environments without Local Procedures (iv)

- Activation tree: tool for analysis of complex structures of calls
  (relevant to execution of one program)





- In general: State of runtime stack → corresponding to sequence of dynamic ancestors

Stack-Based Environments without Local Procedures (v)

- Access to names { parameters, local vars } ⇒ no longer possible by fixed address (like FORTRAN77)

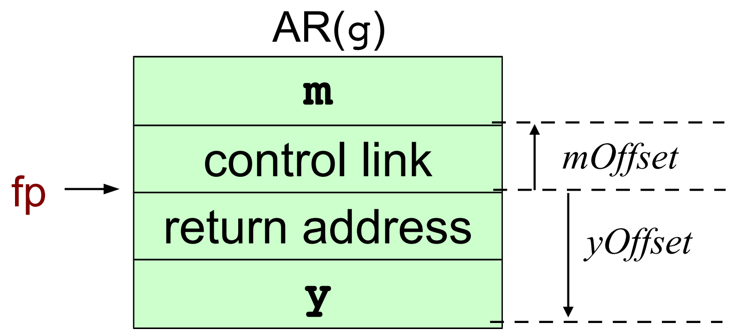
⇓

Access by offset from fp of current AR

↓

Can be computed statically in most PLs

```
void g(int m)
{
  int y=m-1;
  ...
}
```



- ∀ AR(g) { same size; same structure; `m`, `y`: in same relative position { *mOffset*, *yOffset* } }

Assuming { growing of stack from higher to lower addresses; size(*integer*) = 2; size(*address*) = 4 } ⇒ { $mOffset = 4$, $yOffset = -6$ }

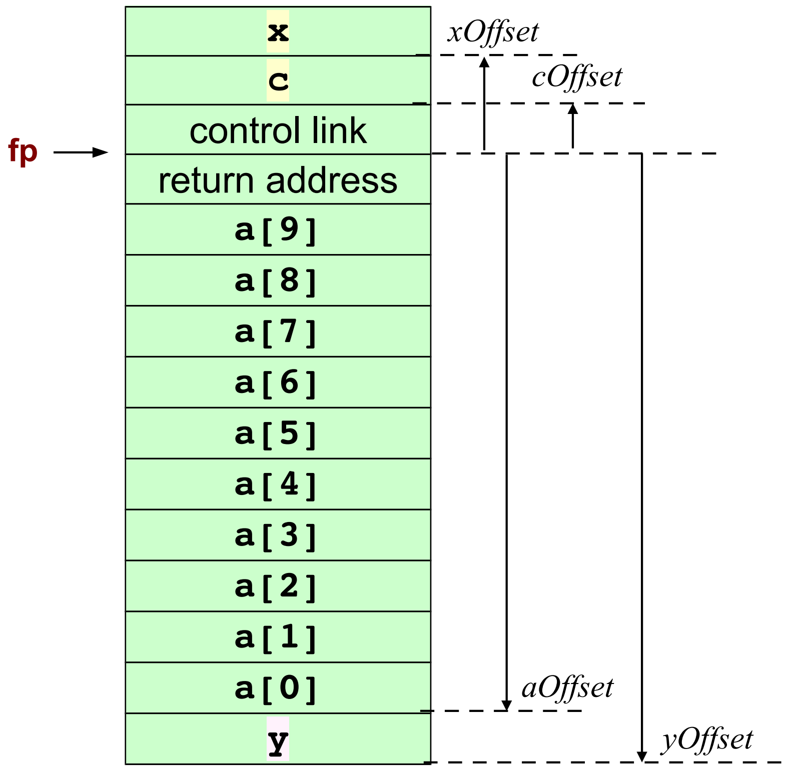
- In standard Assembly notation: referencing { `m` → `4(fp)`, `y` → `-6(fp)` }

Stack-Based Environments without Local Procedures (vi)

- Allocation of complex structures (array, record, …):

```
void f(int x, char c)
{
  int a[10];
  double y;
  ...
}
```

⇒ call to `f`:



- Hp:

Type	*Size*
int	2
&	4
char	1
double	8

⇒

Var	*Offset*
x	+5
c	+4
a	-24
y	-32

⇓

- Access to `a[i]` → requires computation of address (-24+2∗i)(fp) ← scale factor

- Access to var ⟨ global / static ⟩ ⇒ directly

Stack-Based Environments <u>without</u> Local Procedures (vii)

- Calling sequence (ignoring saving of registers) → `call P`

call sequence

<u>At call:</u>

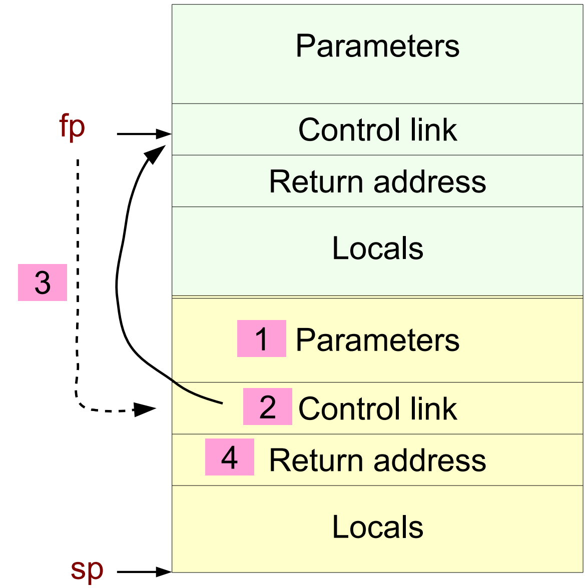
1. Compute actual parameters and insert them into AR(`P`)
2. Store value of **fp** into *control link* of AR(`P`)
3. Assign **fp** by address of *control link* of AR(`P`)
4. Store return address into *return address*
5. Jump to code of `P`

Allocation (and initialization) of local vars of `P`

<u>At return:</u>

6. Assign **fp** by value of *control link*
7. Jump to *return address*
8. Update **sp**

return sequence



Stack-Based Environments without Local Procedures (viii)

- Management of size-varying data
  - variable number of actual parameters
  - variable size of array
    - parameter
    - local var

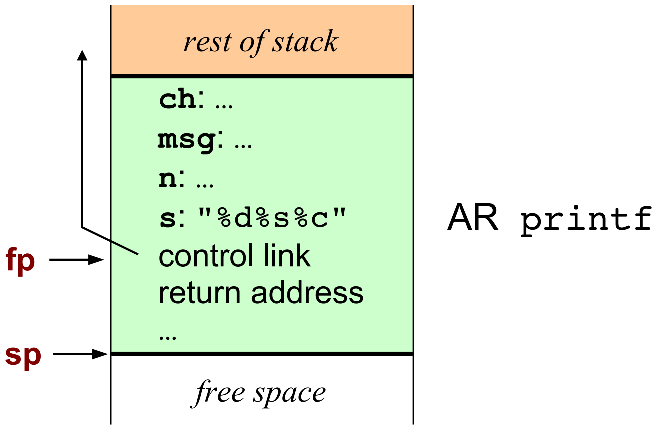
- Variable number of parameters:

```
printf("%d%s%c", n, msg, ch);
```
⇒ 4 arguments

```
printf("Syntax error\n");
```
⇒ 1 argument

Typical solution: Allocation of parameters in reverse order on stack
First parameter: allocated at a fixed distance from fp (+4)

```
printf(char *s){…}
```



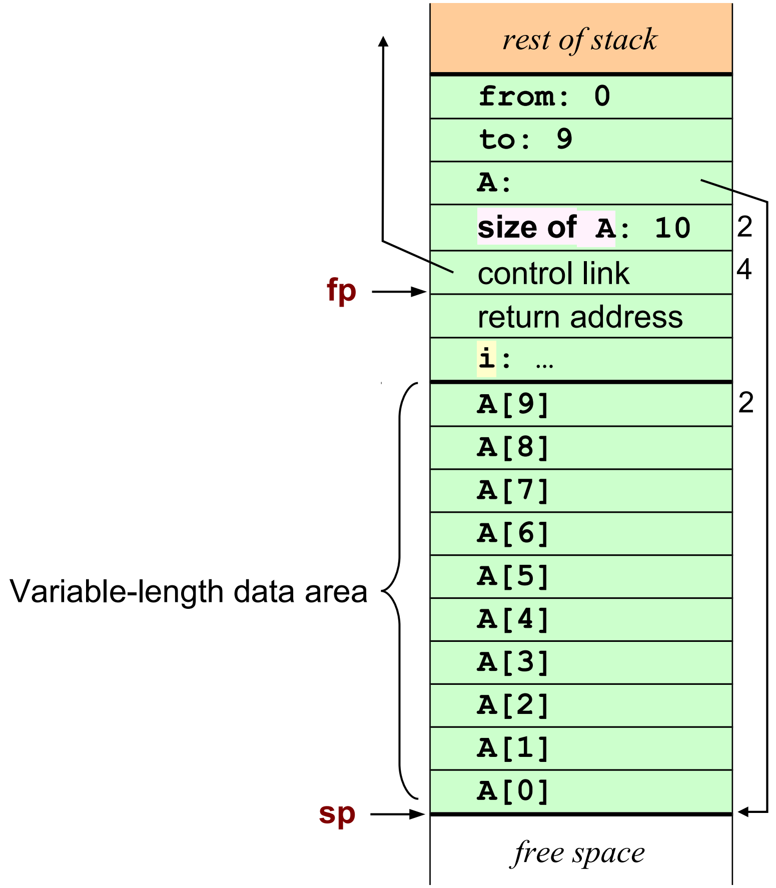
Stack-Based Environments <u>without</u> Local Procedures (ix)

- Variable size of local objects:

```
type IntVet is array (INTEGER range <>) of INTEGER;
procedure sum(from, to: INTEGER; A: IntVet) return INTEGER
is i: INTEGER;
begin ... end sum;
```

⇨

```
sum(0, 9, v)
```



<u>Notes</u>:

- Storage of size of `A`
- Access to `A[i]` → `@6(fp)+2*i` (@ = indirect address)
- Compiler: knows size of { parameter, bookkeeping } at point of call
- Local vars with variable size: treated similarly
- C: array passed by reference → ∄ pb of dynamic management (∄ storage of size)

Stack-Based Environments without Local Procedures (x)

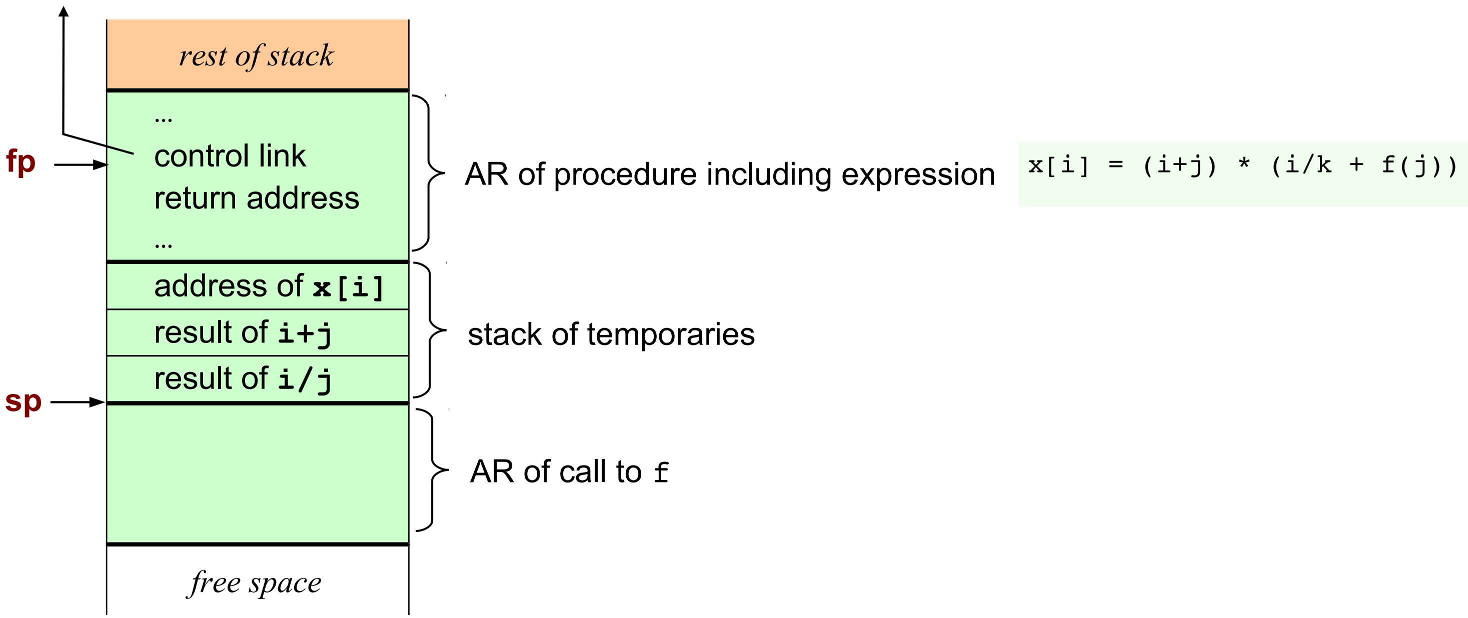
- Local temporaries = partial results to be saved between procedure calls

```
x[i] = (i+j) * (i/k + f(j))
```

⇩

left to right evaluation →
necessary maintaining 3 partial results (temporaries) in call to `f`:
- address of `x[i]`
- `(i+j)`
- `(i/k)`

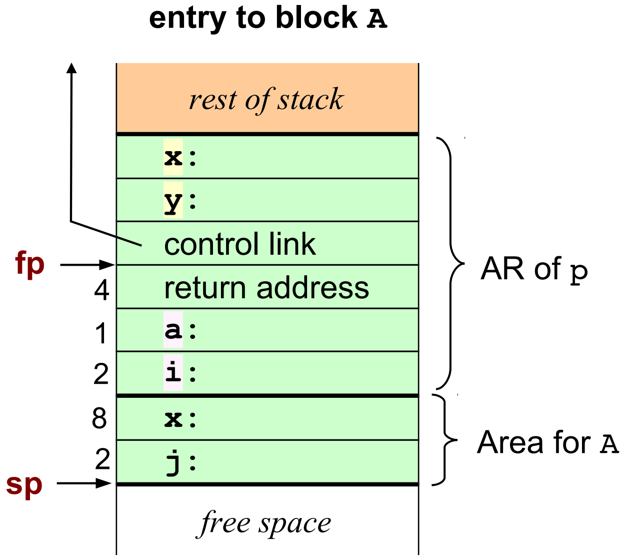
- Possible saving of temporaries in
  - registers
  - stack (before call to `f`)

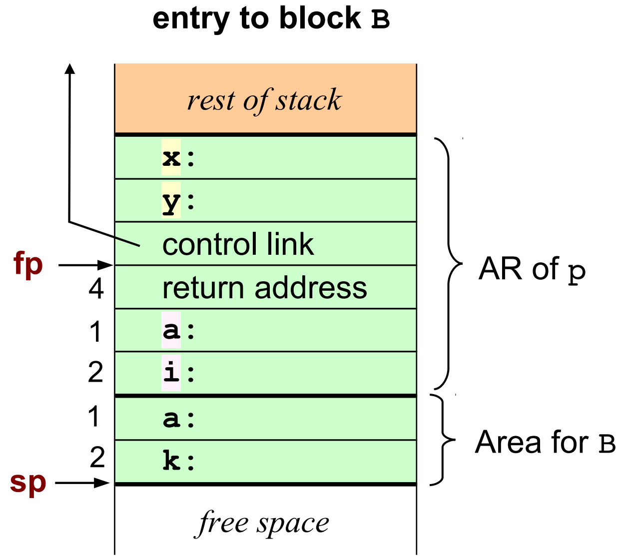


Stack-Based Environments without Local Procedures (xi)

- Nested declarations → allocation only when block is executed

```
void p(int x, double y)
{   char a;
    int i;
    ...
  A:{  double x;
       int j;
       ...
    }
    ...
  B:{  char  a;
       int k;
       ...
    }
    ...
}
```





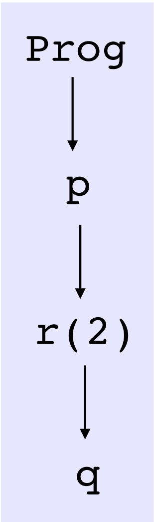
- Possible solution: treating blocks as procedures → AR of block ...
- But: inefficient because block is simpler since
  - has no parameters
  - has no return address
  - executed immediately (instead of being called)
- More efficient solution: handling of declarations in blocks like for temporary expressions
- Requirement: allocation of nested declarations such that offsets of vars are computable

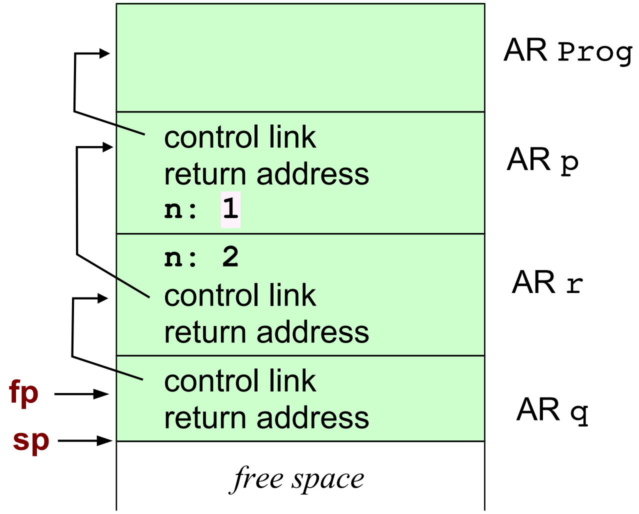
Example $\begin{cases} \texttt{j(A)} \rightarrow -17 \\ \texttt{k(B)} \rightarrow -10 \end{cases}$ ⟹ Allocation of data in block before any variable-length data

Stack-Based Environments <u>with</u> Local Procedures

- Pb: reference to vars not local / global ⇒ necessary extending structure of runtime stack

```
program Prog;
  procedure p;
  var n: integer;
    procedure q;
    begin
      … n …
    end;
    procedure r(n: integer);
    begin
      q
    end;
  begin
    n := 1;
    r(2)
  end;
begin
  p
end.
```

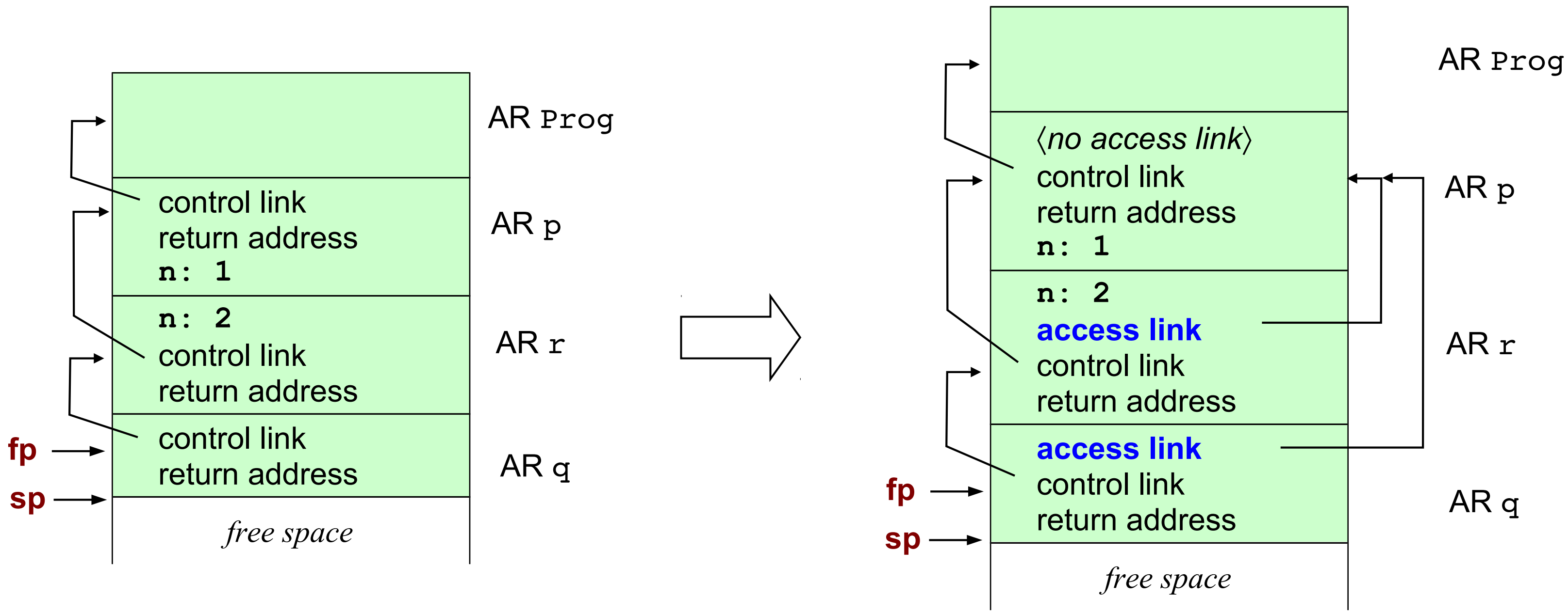




- Reference to n in q: that defined in p → control link: unsuitable to support access with static scope (only for dynamic scope)

Stack-Based Environments with Local Procedures (ii)

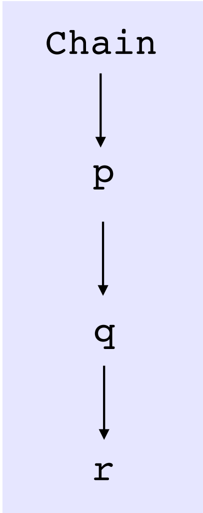
- Solution for static scope: **access link**, like *control link*, but pointing to AR of defining environment of procedure rather than AR of calling environment

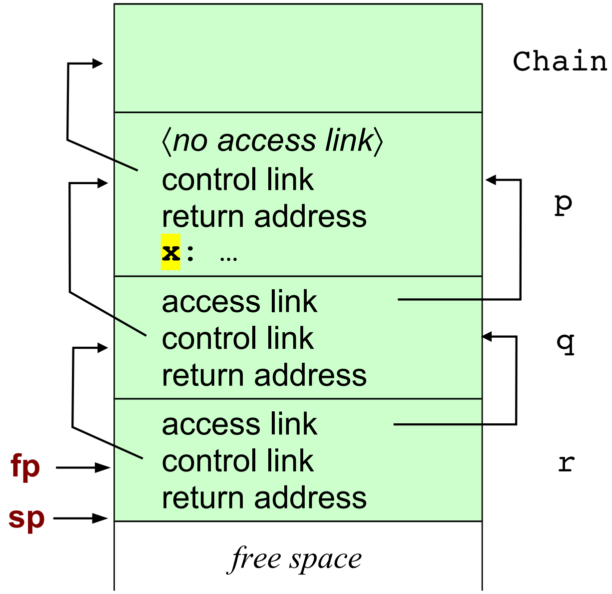


- AR of `p`: does not contain access link because `p` is global → any non-local reference is necessarily global! (alternative solution: for uniformity, access link with null value)

Stack-Based Environments with Local Procedures (iii)

```
program Chain;
  procedure p;
  var x: integer;
    procedure q;
      procedure r;
      begin (r)
        x := 2;
        …
        if … then
          p
      end;
    begin (q)
      r
    end;
  begin (p)
    q
  end;
begin (Chain)
  p
end.
```





- Access to x in r → traversing of two access links ≡ **access chaining**

⇓

```
1) load 4(fp) into register r
2) load 4(r) into register r
3) access x as -6(r)
```

Stack-Based Environments <u>with</u> Local Procedures (iv)

- Compiler: needs to know (statically!) how many nesting levels to traverse
- ∀ declaration → necessary computing attribute **nesting level**
- Typically: global declarations → 0, then: increment by 1 ∀ nesting

```
program Chain;
  procedure p;
  var x: integer;
    procedure q;
      procedure r;
      begin
        x := 2;
        …
        if … then
          p
      end;
    begin
      r
    end;
  begin
    q
  end;
begin
  p
end.
```

⇒

Element	*Nesting level*
`p`	0
`x`	1
`q`	1
`r`	2
inside `r`	3

- Formula: Number of chaining steps = ***nl***(reference point) – ***nl***(declaration point) ≡ *m*

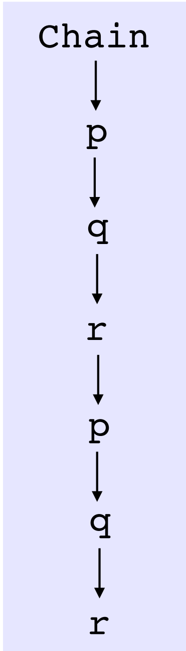
```
x := 2    →    3-1 = 2
```

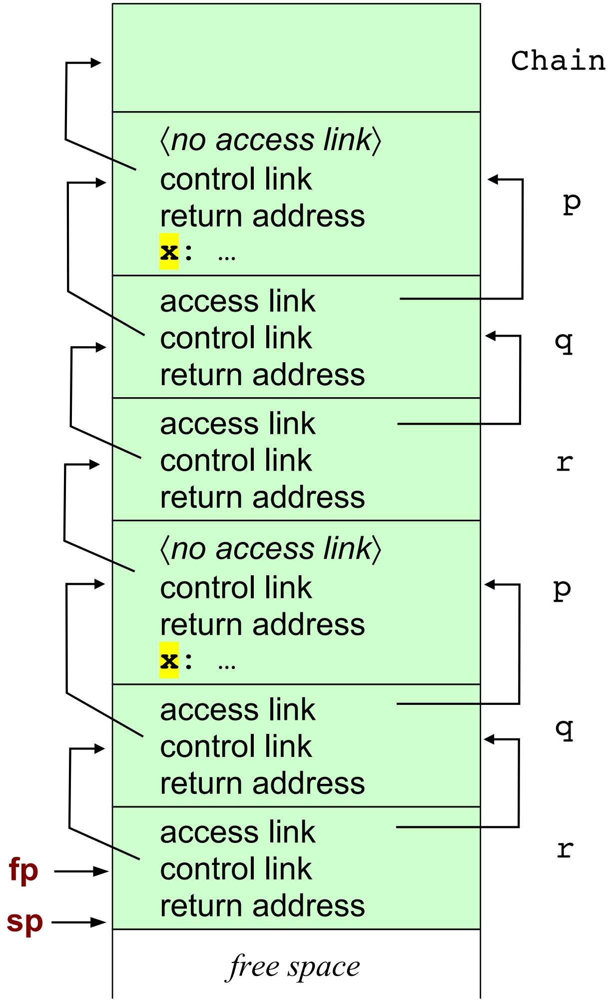
- Mapping to code: *m* `load` of a register `rg` from $\begin{cases} \texttt{fp} : \text{first} \\ \texttt{rg} : \text{next} \end{cases}$

Stack-Based Environments with Local Procedures (v)

- Chaining: works even with several activations of defining environment (chosen the nearest)

```
program Chain;
  procedure p;
  var x: integer;
    procedure q;
      procedure r;
      begin
        x := 2;
        …
        if … then
          p
      end;
    begin
      r
    end;
  begin
    q
  end;
begin
  p
end.
```

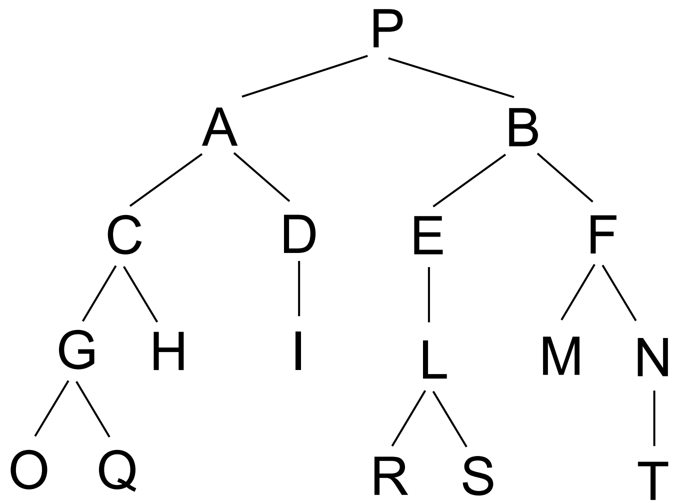


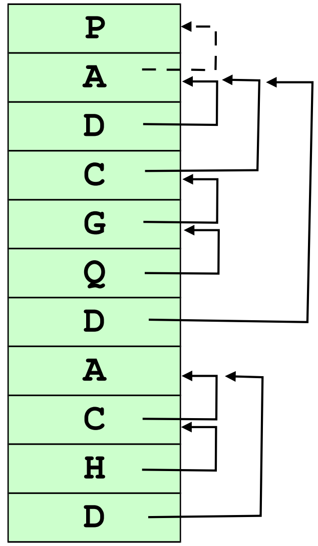


- Note: Multiplicity of AR instances of same procedure!

Stack-Based Environments <u>with</u> Local Procedures (v)

- <u>Pb</u> in calling sequence: determination of (dynamic!) value of access link





- <u>Rules</u>: (P_1 calls P_2)

  1. If called global procedure → access link (***al***) null
  2. If P_1 parent of P_2 → ***al***(P_2) = AR(P_1)
  3. If P_1 sibling of P_2 → ***al***(P_2) = ***al***(P_1)
  4. Otherwise (in general) → ***al***(P_2) = ***al***(…(***al***(***al***(P_1))…), where ***al*** applied (***nl***(P_1) – ***nl***(P_2) + 1) times

Stack-Based Environments with Procedure Parameters

- Impossible for compiler to generate code for computing access link of a procedure parameter which is called $\rightarrow \nexists$ static correlation!

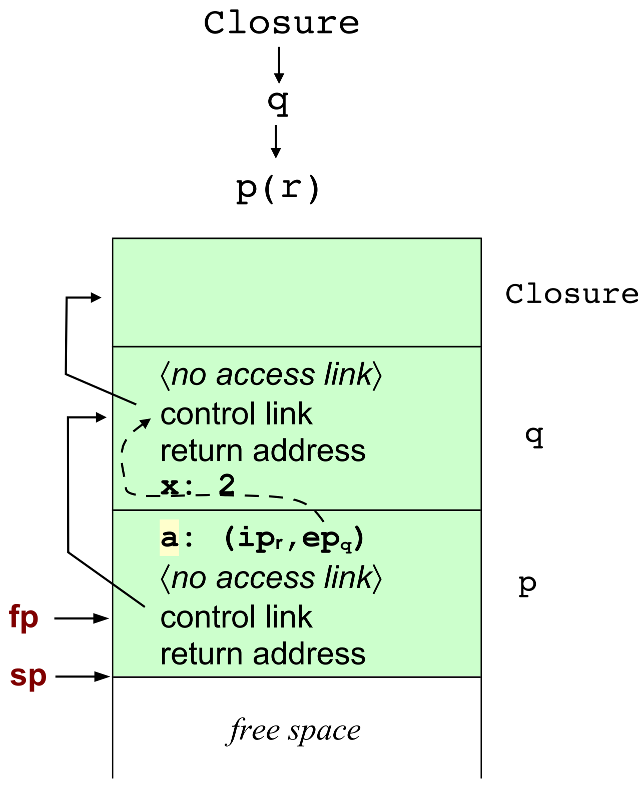
- Solution: access link passed along with pointer to code

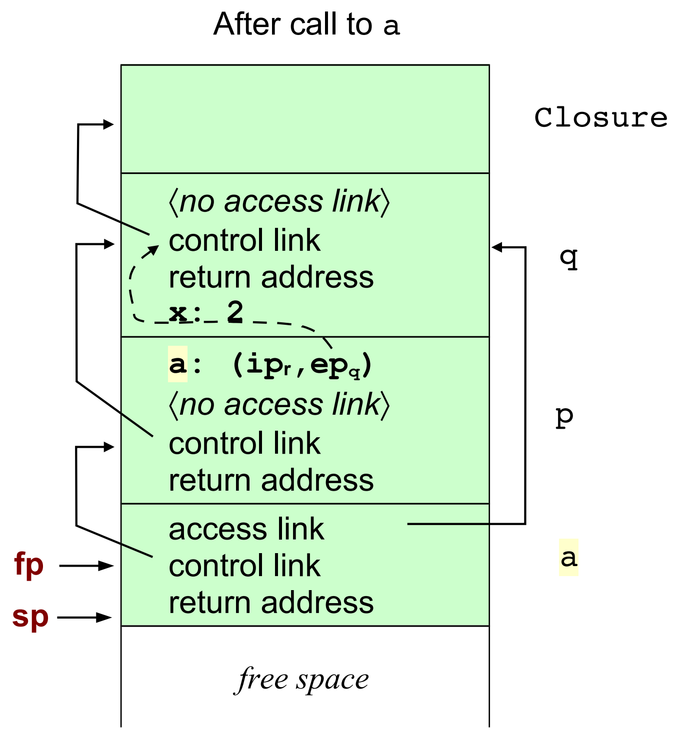
- Procedure parameter = (**ip**, **ep**) $\equiv$ closure (closes holes left by nonlocal references)

$$\begin{cases} \textbf{ip} \equiv \text{instruction pointer} \\ \textbf{ep} \equiv \text{environment pointer} \end{cases}$$

Stack-Based Environments with Procedure Parameters (ii)

```
program Closure(output);
  procedure p(procedure a);
  begin (p)
    a
  end;
  procedure q;
  var x: integer;
    procedure r;
    begin (r)
      writeln(x)
    end;
  begin (q)
    x := 2;
    p(r)
  end;
begin (Closure)
  q
end.
```

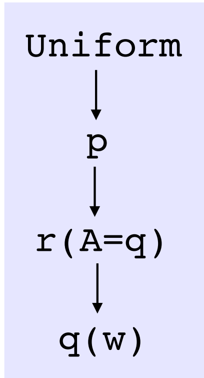


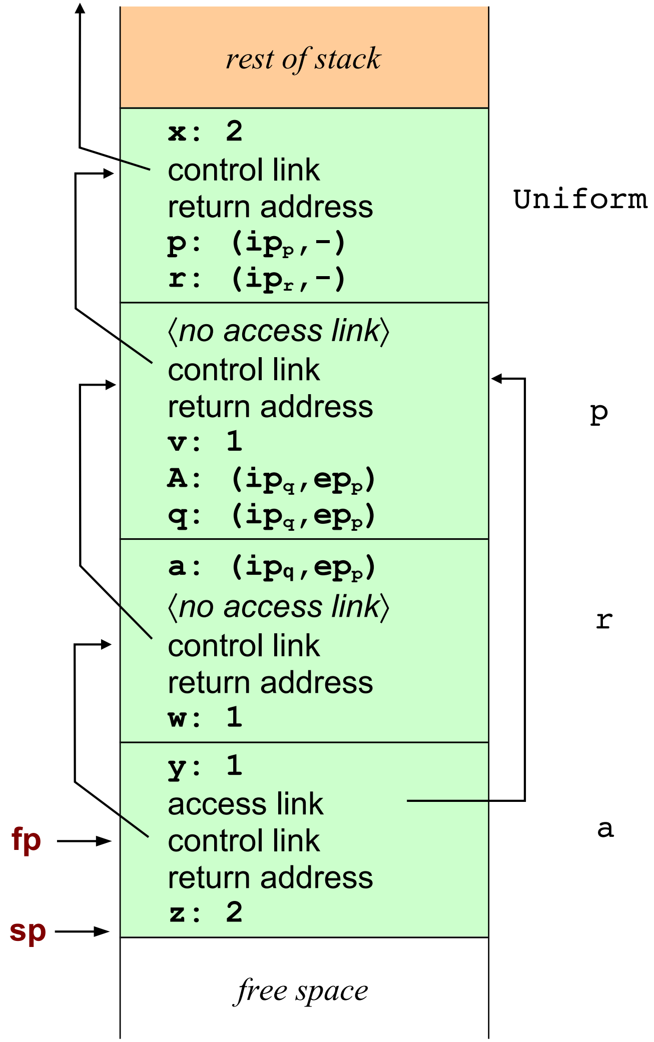


- Calling sequence: shall distinguish procedures
  - ordinary → address(code) known by compiler
  - parameter → indirect jump by $\mathtt{ip}_\mathtt{r}$
- Uniformity choice: representation of all procedures as (ip , ep)

Stack-Based Environments with Procedure Parameters (iii)

```
program Uniform;
  var x: integer
  procedure p;
    var v: integer,
        A: procedure(i: integer);
    procedure q(y: integer);
      var z: integer;
    begin (q)
      z := y+1
    end;
  begin (p)
    v := 1;
    A := q;
    r(A)
  end;
  procedure r(a: procedure(j: integer));
    var w: integer;
  begin (r)
    w := 1;
    a(w)
  end;
begin (Uniform)
  x := 2;
  p
end.
```





- Languages
  - C: ∄ local procedures (functions)
  - Modula-2: procedure parameters/variables only global ⇨ no problem
  - Ada: ∄ procedure parameters/variables