Compiler Principle and Technology

Prof. Dongming LU
Apr. 22th, 2015

7. Runtime Environments

Contents

- The precious chapters studied the phases of a compiler that **perform static analysis** of the source language
  - Scanning, parsing, and static semantic analysis
  - Depends only on the **properties of the source language**

- Now turn to the task of studying **how a compiler generates executable code**
  - **Additional analysis**, such as that performed by an optimizer
  - Some of this can be machine independent, but much of the task of code generation is **dependent on the details of the target machine**

- **Runtime Environment**

  The structure of the target computer's **registers and memory** that serves to manage memory and maintain the information needed **to guide the execution process**

- **Three kinds of runtime environments**

  (1) **Fully static** environment; FORTRAN77

  (2) **Stack-Based** environment; C C++

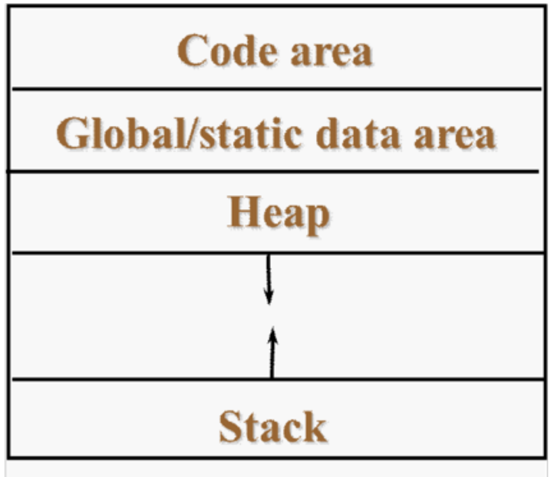
  (3) **Fully dynamic** environment; LISP

- Main issues discussed in the chapter:
  - For each environment, the language features and their properties
    - (1) **Scoping and allocation** issues;
    - (2) Nature of **procedure calls**;
    - (3) **Parameter passing** mechanisms
- Focus on the **general structure** of the environment
- **Note:**
  - The compiler can only maintain an environment **only indirectly**
  - It must **generate code** to perform the necessary maintenance operations during program execution.

7.1 Memory Organization During Program Execution

The memory of a typical computer:

- A register area;
- Addressable Random access memory (RAM):



The code area is fixed prior to execution, and can be visualized as follows:

Entry pointer to procedure1→	Code for procedure 1
Entry pointer to procedure2→	Code for procedure 2
	…
Entry pointer to procedure n→	Code for procedure n

In particular, the entry point for each procedure and function is known at compile time.

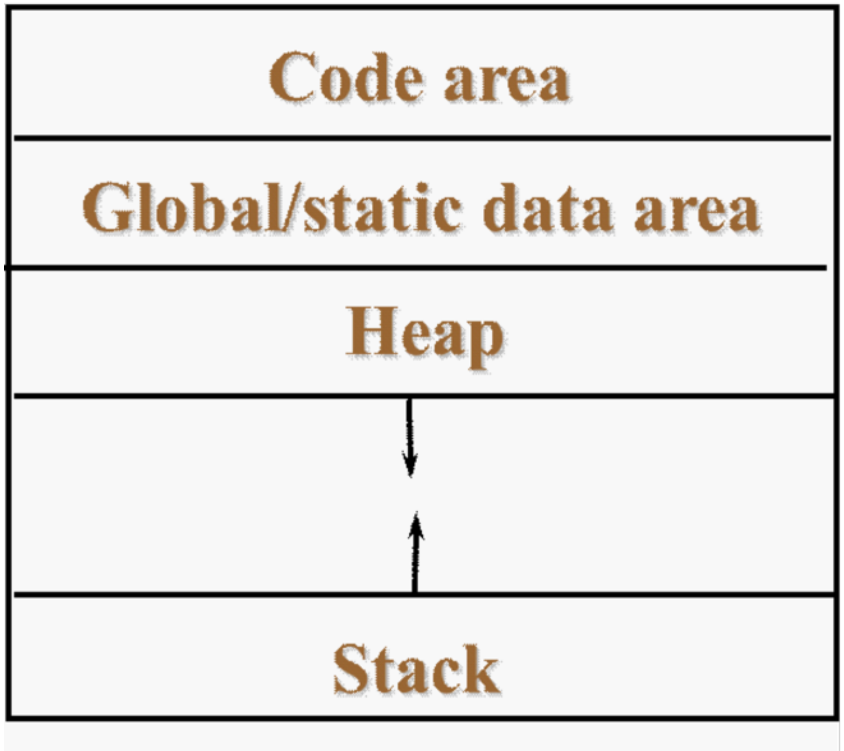
- The global and/or static data of a program can be **fixed in memory prior to execution**
  - Data are allocated separately in a fixed area in a similar fashion to the code
    - In Fortran77, all data are in this class;
    - In Pascal, global variables are in this class;
    - In C, the external and static variables are in this class

- The **constants** are usually allocated memory **in the global/static area**
  - Const declarations of C and Pascal;
  - Literal values used in the code,
    - Such as "Hello%D\n" and Integer value 12345:
    - Printf("Hello %d\n",12345);

- The memory area used for **dynamic data** can be organized in many different ways
  - **A stack area** used for data whose allocation occurs in FIFO fashion;
  - **A heap area** used for dynamic allocation occurs not in FIFO fashion.

- The architecture of the target machine includes a **processor stack** for procedure calls and returns.
  - A compiler will have to arrange for the **explicit allocation of the processor stack** in an appropriate place in memory.

The general organization of runtime storage:



Where, the arrows indicate the direction of growth of the stack and heap.

Procedure activation record

(An important unit of memory allocation)

Memory allocated for the local data of a procedure or function.

An activation record must contains the following sections:

Space for arguments (parameters)
Space for bookkeeping information, including return address
Space for local data
Space for local temporaries

Note: this picture only illustrates the general organization of an activation record.

- Some parts of an activation record have the **same size for all procedures**
  - Space for bookkeeping information
- Other parts of an activation record may **remain fixed for each individual procedure**
  - Space for arguments and local data
- Some parts of activation record may be **allocated automatically** on procedure calls:
  - Storing the return address
- Other parts of activation record may need to be **allocated explicitly** by instructions generated by the compiler:
  - Local temporary space

- **Depending on the language**, activation records may be allocated in different areas:
  - Fortran77 in the static area;
  - C and Pascal in the stack area; referred to as stack frames
  - LISP in the heap area.

- **Processor registers are also part of the structure of the runtime environment**
  - Registers may be used to store temporaries, local variables, or even global variables;
  - In newer RISC processor, keep entire static area and whole activation records;
  - Special-purpose registers to keep track of execution
    - PC program counter;
    - SP stack pointer;
    - FP frame pointer;
    - AP argument pointer

- **The sequence of operations when calling the functions**: ***calling sequence***
  - The allocation of memory for the activation record;
  - The computation and storing of the arguments;
  - The storing and setting of necessary registers to affect the call

- **The additional operations when a procedure or function returns: *return sequence (VS call)***
  - The placing of the return value where the caller can access it;
  - The readjustment of registers;
  - The possible releasing for activation record memory

- **The important aspects of the design of the calling sequence:**
  - (1) How to **divide the calling sequence** operations between the caller and callee
    - At a minimum, the caller is responsible for computing the arguments and placing them in locations where they may be found by the callee
  - (2) To what extent to **rely on processor support for calls** rather that generating explicit code for each step of the calling sequence

7.2 Fully Static Runtime Environments

The entire program memory can be visualized as follows:

<table>
<tr><td>Code for main procedure</td><td rowspan="4">Code area</td></tr>
<tr><td>Code for procedure 1</td></tr>
<tr><td>…</td></tr>
<tr><td>Code for procedure n</td></tr>
<tr><td></td><td rowspan="6">Data area</td></tr>
<tr><td>Global data area</td></tr>
<tr><td>Activation record of main procedure</td></tr>
<tr><td>Activation record of procedure 1</td></tr>
<tr><td>…</td></tr>
<tr><td>Activation record of procedure n</td></tr>
</table>

- All data are static, remaining fixed in memory for the duration of program execution

- For a language, such as FORTRAN77, **no pointer or dynamic allocation, no recursive procedure calling**
  - The global variables and all variables are allocated statically.
  - Each procedure has only a single activation record.
  - All variable, whether local or global, can be accessed directly via fixed address.

Advantages of Fully Static Runtime Environments

- Relative little overhead in terms **of bookkeeping information** to retain in each activation record;
- And **no extra information** about the environment needs to be kept in an activation record;
- The calling sequence is simple.
  - Each argument is computed and stored into its appropriate parameter location;
  - The return address is saved, and jump to the beginning of the code of the callee;
  - On return, a simple jump is made to the return address.

Example: A FORTRAN77 sample program

```
PROGRAM TEST
COMMON     MAXSIZE
INTEGER    MAXSIZE
REAL       TABLE(10),TEMP
MAXSIZE = 10

READ *, TABLE(1),TABLE(2),TABLE(3)
CALL QUADMEAN(TABLE,3,TEMP)
PRINT *, TEMP
END
```

```
      SUBROUTINE    QUADMEAN(A, SIZE,QMEAN)
      COMMON    MAXSIZE
      INTEGER   MAXSIZE,SIZE
      REAL A(SIZE),QMEAN,TEMP
      INTEGER K

      TEMP=0.0
      IF ((SIZE .GT. MAXSIZE) .OR. (SIZE .LT. 1) GOTO 99
      DO 10 K=1,SIZE
      TEMP=TEMP+A(K)*A(K)
10    CONTINUE
99    QMEAN = SQRT(TEMP/SIZE)
      RETURN
      END
```

A runtime environment for the program above.

Global area	MAXSIZE
Activation record of main procedure	TABLE (1) (2) … (10)
	TEMP
	3
Activation record of procedure QUADMEAN	A
	SIZE
	QMEAN
	Return address
	TEMP
	K
	Unnamed location

Note: The unnamed location is used to store temporary value during the computation of arithmetic expression.

limitations of Fully Static Runtime Environments

- Recursive calls are not allowed.
- The data object's size and their location in memory is decided when compiling.
- Dynamical memory allocation is not allowed.

7.3 Stack-Based Runtime Environments

- For a language, **in which**
  - Recursive calls are allowed;
  - Local variables are newly allocated at each call;
  - Activation records cannot be allocated statically

- Activation records must be **allocated in a stack-based fashion**
  - The stack of activation records grows and shrinks with the chain of calls in the executing program.
  - Each procedure may have several **different activation records on the call stack at one time**, each representing a distinct call.
  - **More complex strategy for bookkeeping and variable access,** which depends heavily on the properties of the language being compiled.

7.3.1 Stack-Based Environments Without Local Procedures

- All properties are global (such as C language), the stack-based environment requires two things:

  (1) **Frame pointer or fp**, a pointer to the current activation record to allow access to local variable; **Control link or dynamic link**, a point to a record of the immediately preceding activation

  (2) **Stack pointer or sp**, a point to the last location allocated on the call stack

◈ **Example:** The simple recursive implementation of Euclid's algorithm to compute the greatest common divisor of two non-negative integer

```
# 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;
    }
```

Suppose the user inputs the values 15 and 10 to this program.

The environment during the third call:

X:15 Y:10	Global/static area
	Activation record of main
U:15 V:10 Control link Return address	Activation record of First call to gcd
U:10 V:5 Control link Return address	Activation record of Second call to gcd
U:5 V:0 Control link Return address	Activation record of Third call to gcd
Free space	↓ Direction of Stack growth

Fp

sp

After the final call to gcd, each of the activations is removed in turn from the stack.

Example:

```
Int x=2;
void g(int);/*prototype*/
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;
}
```

(a) Runtime environment of the above program during the second call to g

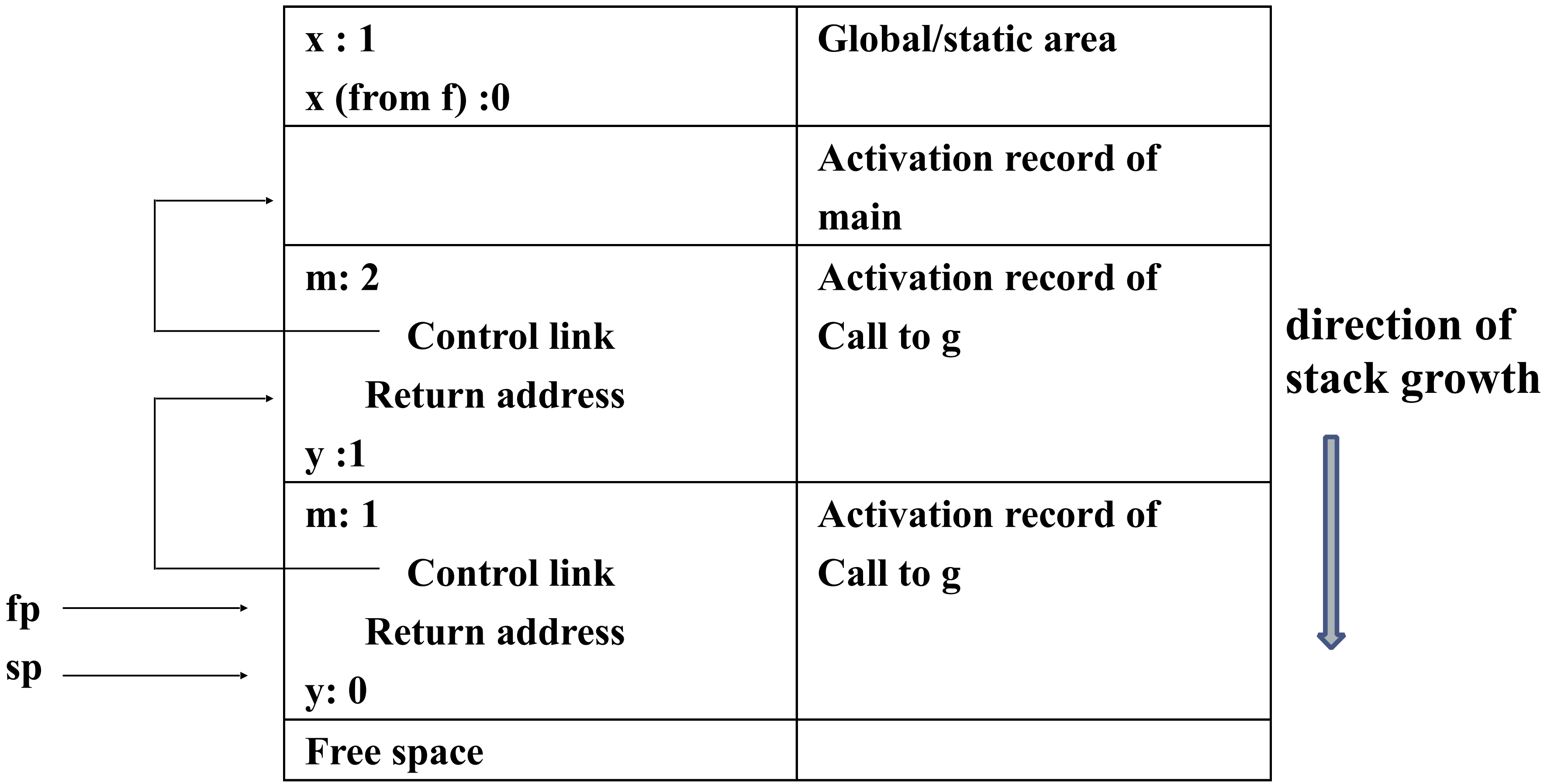
x : 2 x (from f) :1	Global/static area
	Activation record of main
M: 2 Control link Return address y :1	Activation record of Call to g
n: 1 Control link Return address	Activation record of Call to f
m: 1 Control link Return address y: 0	Activation record of Call to g
Free space	

fp

sp

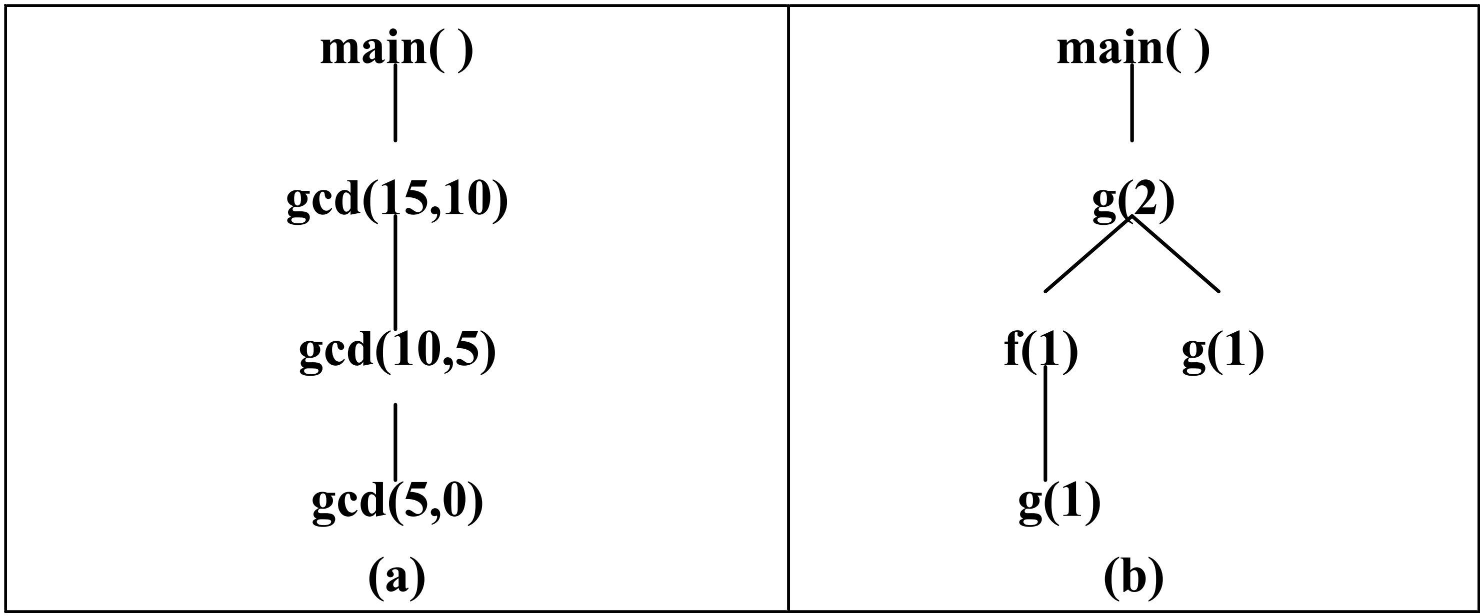
direction of stack growth

(b) Runtime environment of the above program during the third call to g



Activations tree: a useful tool for the analysis of complex calling structures

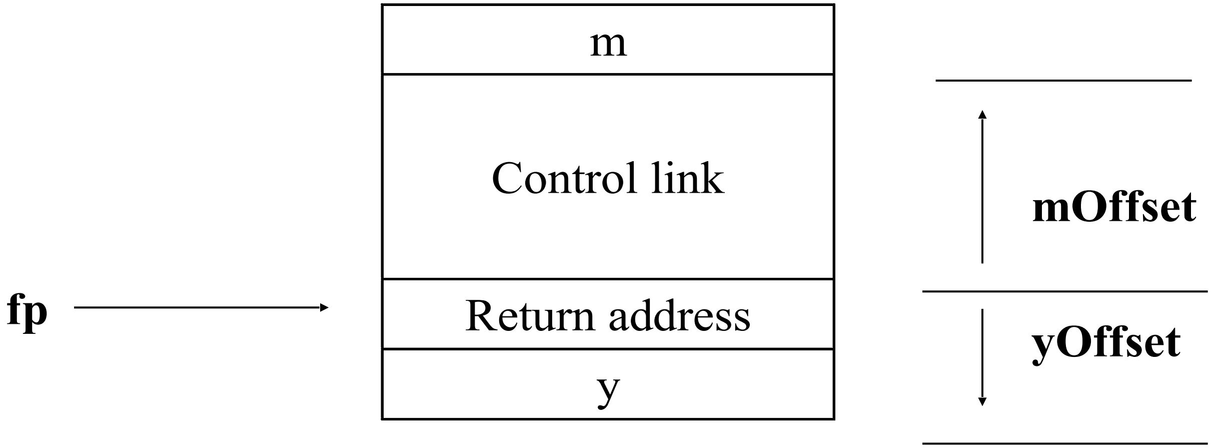
Example: activation trees for the program of figures 7.3 and 7.5



Note: In general, the stack of activation records at the beginning of a particular call has a structure equivalent to the path from the corresponding node in the activation tree to the root.

- Access to Names:
  - Parameters and local variable can **no longer be accessed by fixed addresses**
  - They must be found by offset from the current frame pointer.
  - In most language, the offset can be statically computable by the compiler.

Consider the procedure g in the C program of Figure 7.5.



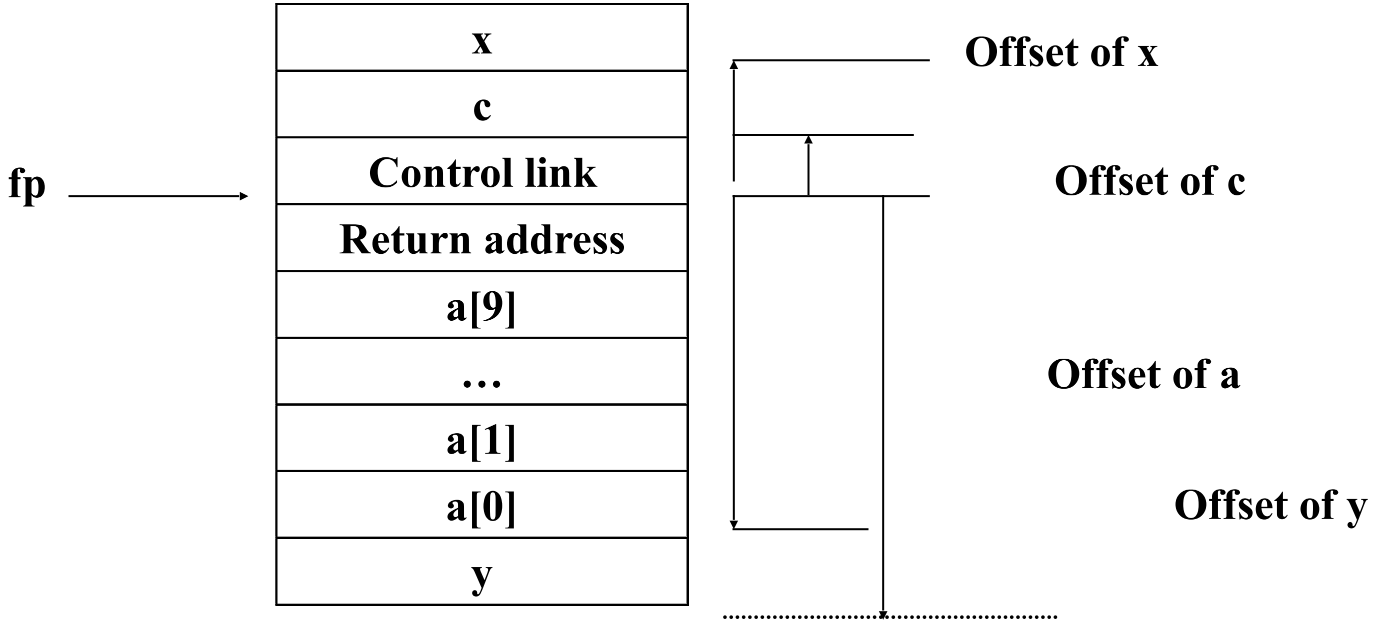
Note:

- Each activation record of g has exactly the same form, and the parameter m and the local variable y are always in exactly the same relative location in the activation record.
- Both m and y can be accessed by their fixed offset from the fp.
- We have mOffset=+4 and yOffset=-6.
- The references to m and y can be written in machine code as 4(fp) and –6(fp).

Example 7.4 Consider the C procedure

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

The activation record for a call to f would appear as



Assuming two bytes for integers, four bytes for addresses, one byte for character and eight bytes for double-precision floating point, we would have the following offset values:

Name	Offset
x	+5
c	+4
a	-24
y	-32

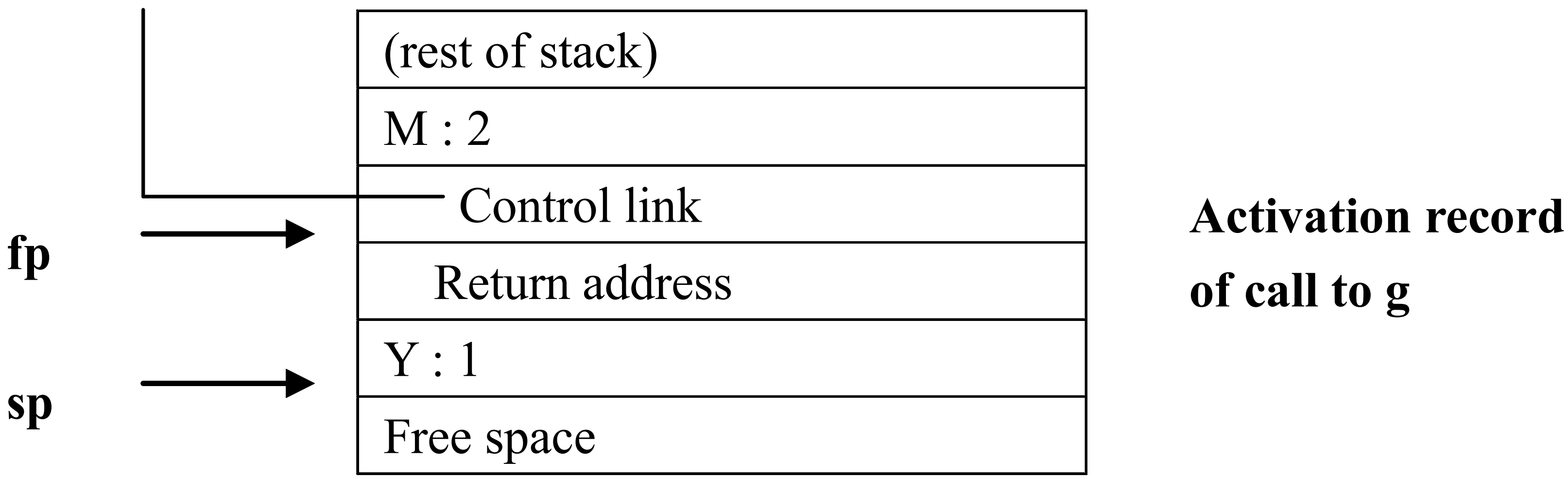
Now, an access of a[i], would require the computation of the address:

$$(-24+2*i)(fp)$$

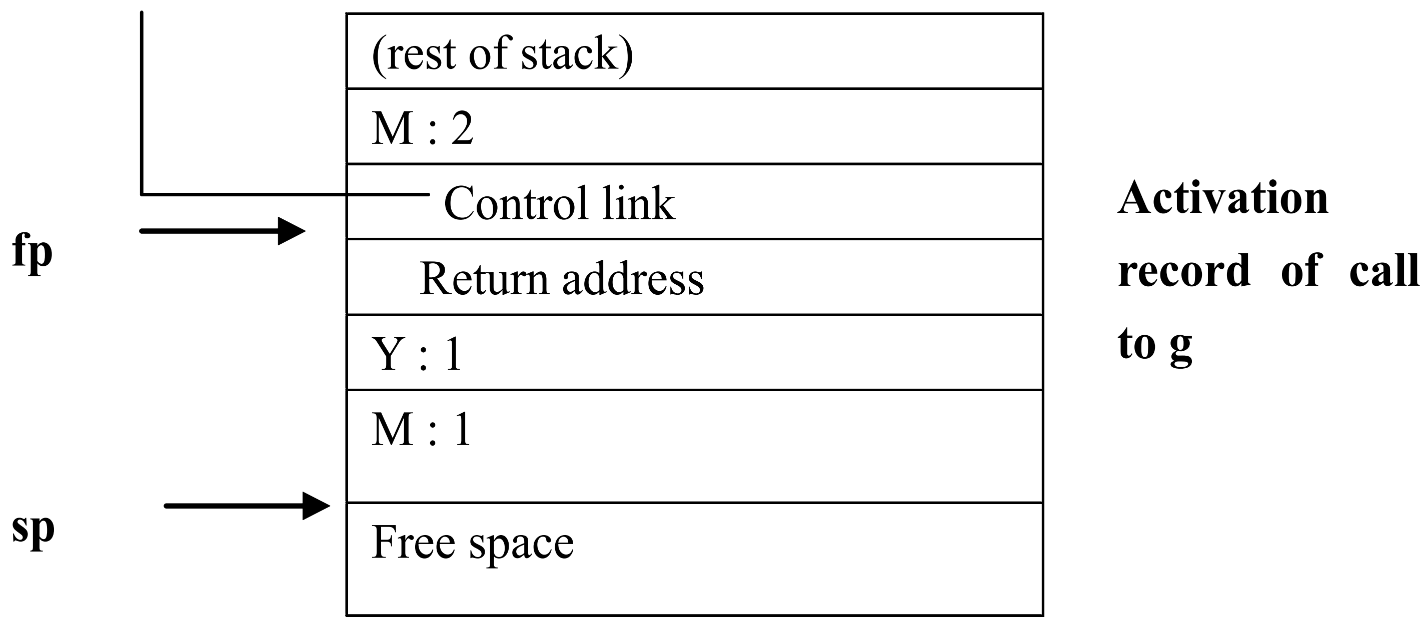
The calling sequence:

- When a procedure is called
  - **Compute the arguments** and store them in their correct positions in the new activation record of the procedure;
  - **Store the fp** as the control link in the new activation record;
  - **Change the fp** so that it points to the beginning of the new activation record;
  - **Store the return address** in the new activation record;
  - **Perform a jump** to the code of the procedure to be called.
- When a procedure exits
  - Copy the fp to the sp.
  - Load the control link into the fp.
  - Perform a jump to the return address;
  - Change the sp to pop the arguments.

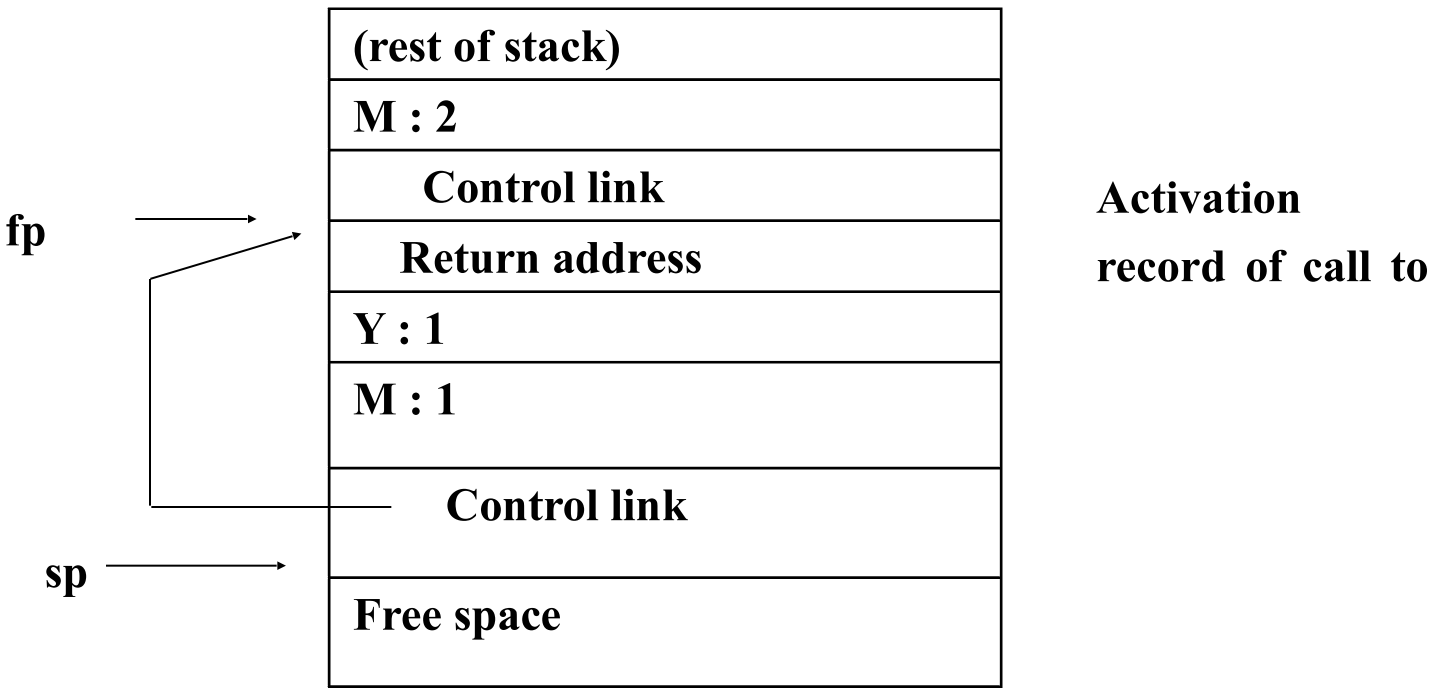
Example 7.5 Consider the situation just before the last call to g



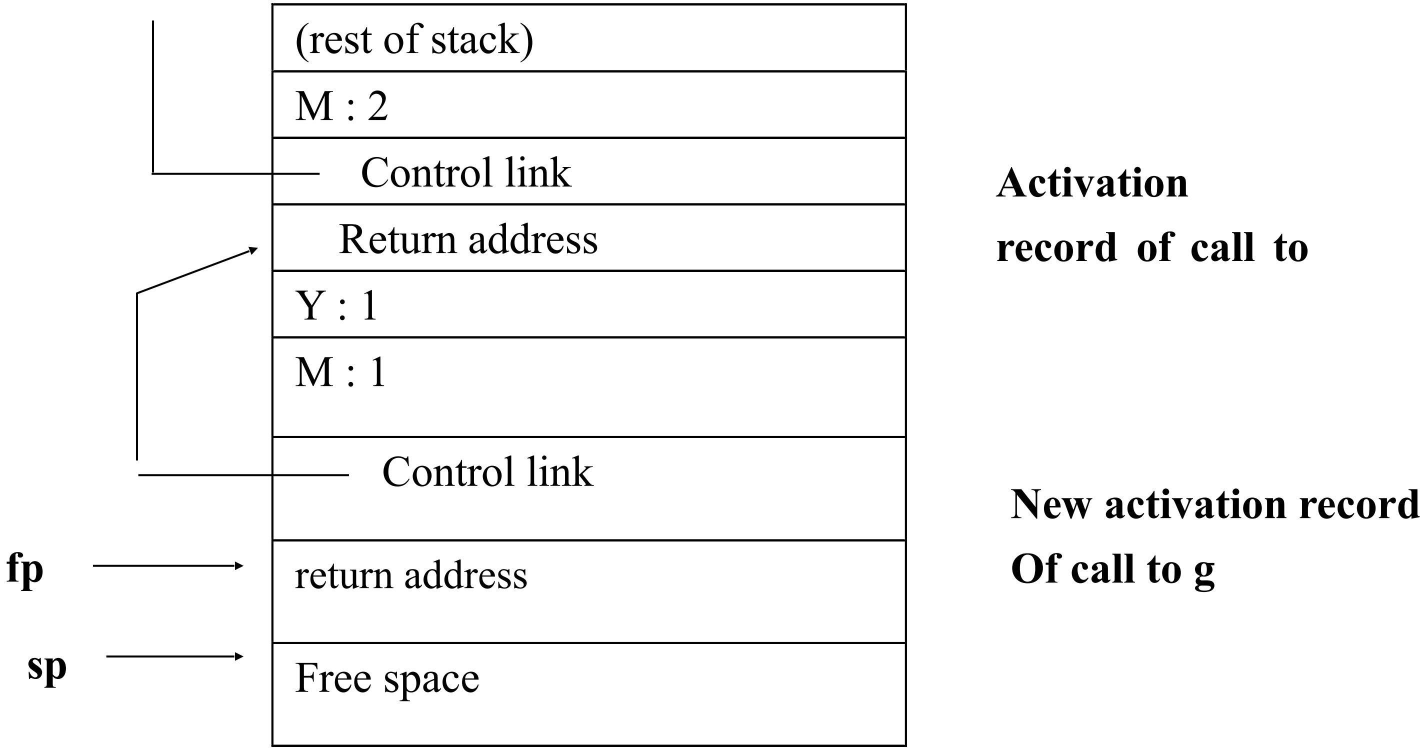
As the new call to g is made, first the value of parameter m is pushed onto the runtime stack:



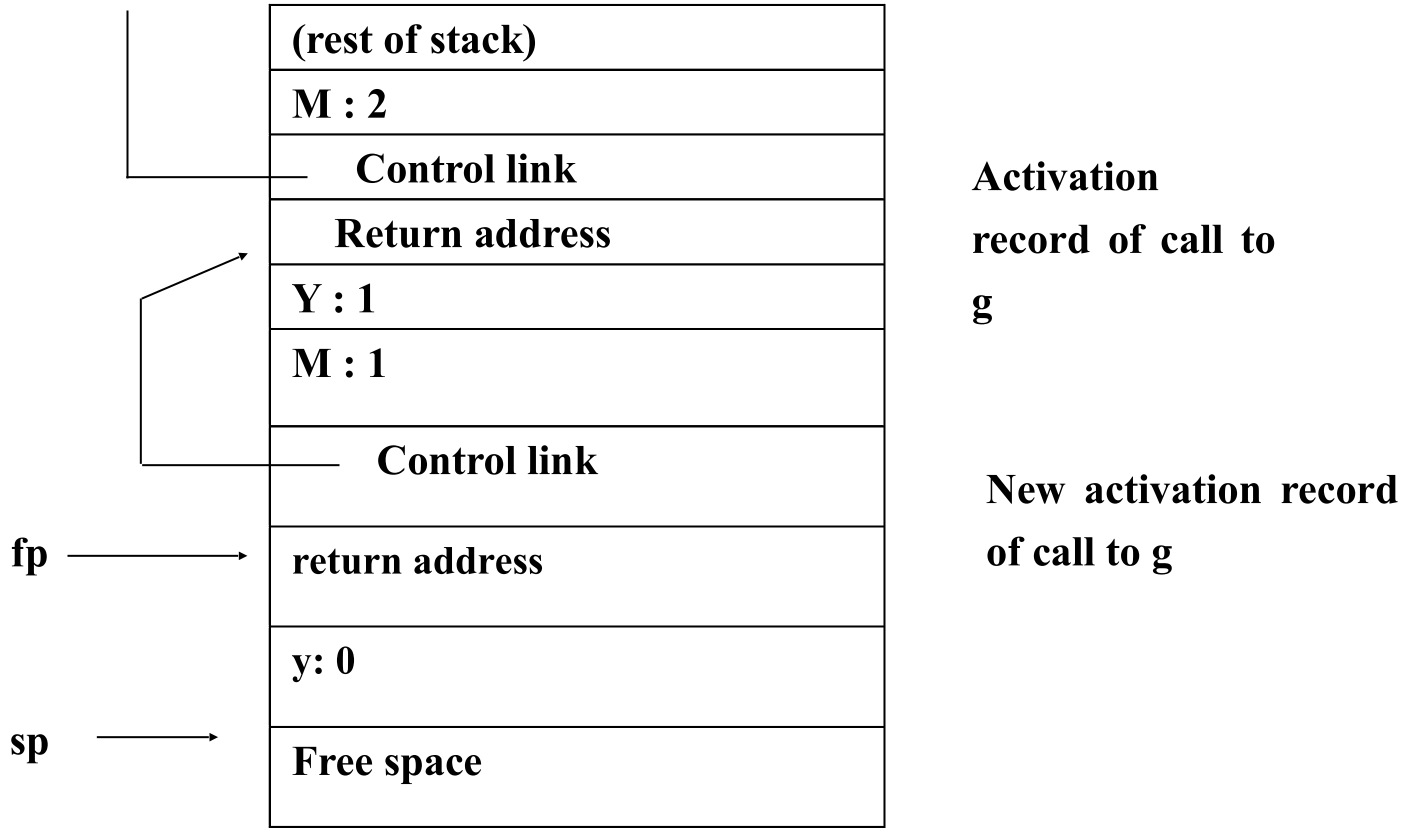
Then the fp is pushed onto the stack:



Now, the sp is copied into the fp, the return address is pushed onto the stack, and the jump to the new call of g is made:



Finally, g allocates and initializes the new y on the stack to complete the construction of the new activation record:



- **Dealing with variable-length data**
  - The number of arguments in a call may vary from call to call, and
  - The size of an array parameter or a local array variable may vary from call to call

  - An example of situation 1 is the printf function in C:
    - Printf("%d%s%c", n, prompt, ch)
      - Has four arguments, while
    - Printf("Hello, world\n")
      - Has only one argument

- C compiler typically deal with this by **pushing the arguments to a call in reverse order** onto the runtime stack.
- The **first parameter is always located at a fixed offset from the fp** in the implementation described above

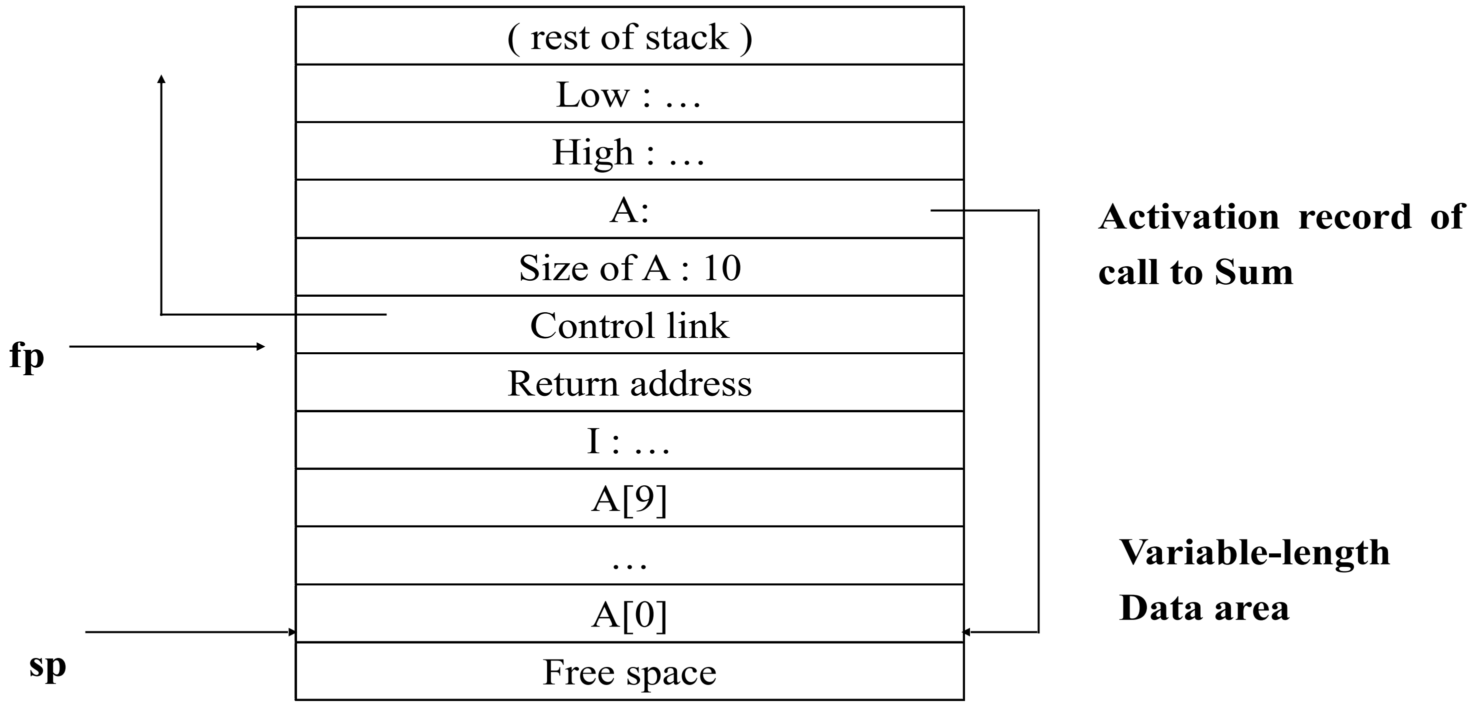
- Another option is to use a processor mechanism such as **ap (argument pointer)** in VAX architecture.

- An example of situation 2 is the unconstrained array of Ada:

```
Type int_vector is
        Array(INTEGER range <>) of INTEGER;
Procedure sum (low, high: INTEGER;
        A: Int_vector) return INTEGER
Is
        Temp: Int_Array (low..high);
Begin
…
end sum;
```

- A typical method is to **use an extra level of indirection for the variable-length data, storing a pointer to the actual data** in a location that can be predicated at compile time.

We could implement an activation record for SUM as follows:



Now, for instance, access to A[i] can be achieved by computing

@6(fp)+2*i

- Note:
  - In the implementation described in the previous example, the caller must know the size of any activation record of Sum
  - The size of the parameter part and the bookkeeping part is known to the compiler at the point of call
  - The size of the local variable part is not, in general, known at the point of call. Variable-length local variables can be dealt with in a similar way

- **Local Temporaries and Nested Declarations:**
  - Two more complications to the basic stack-based runtime environment

  (1) Local temporaries are partial results of computations that must be saved across procedure calls, for example:

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

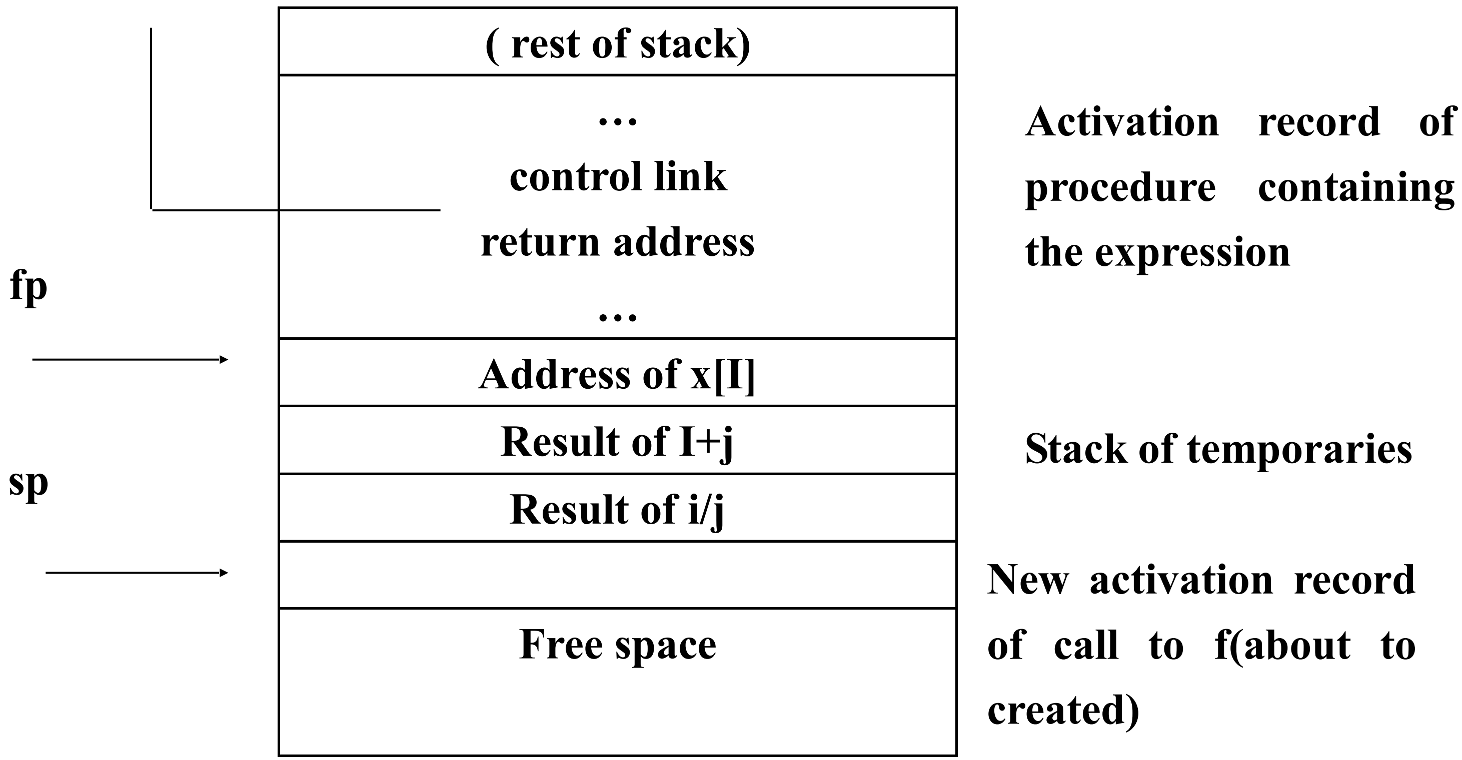
- The three partial results need to be saved across the call to f:

  The address x[i];

  The sum i+j;

  The quotient i/k;

The runtime stack might appear as follows at the point just before the call to f:



- Nested declarations present a similar problem. Consider the C code

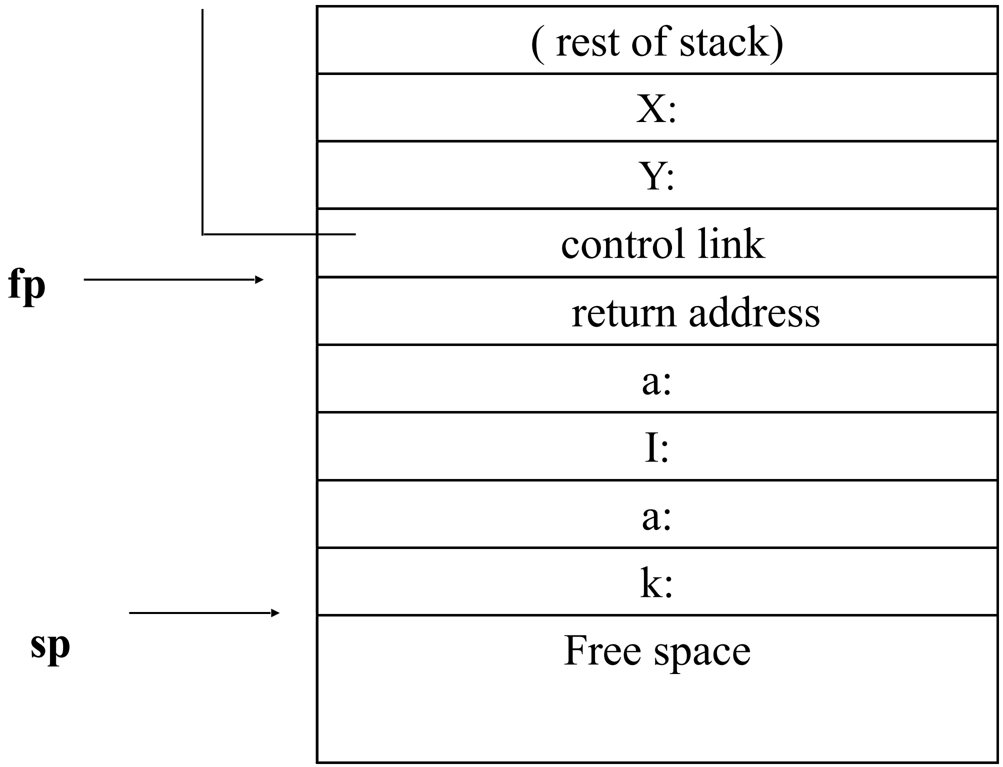
```
Void p( int x, double y)
{
   char a;
   int I;
...
  A:{      double x;
           Int j;
           ...
     }
...
  B:{
           char *a;
           Int k;
           ...
      }
           ...
}
```

- There are two block (also called compound statement), labeled A and B nested inside the body of procedure p.
- The nested local declaration does not need to be allocated until entered;
- Both A and B do not need to be allocated simultaneously.

A simpler method is to treat them in a similar way to temporary expression. For instance, just after entering block A in the sample C just given, the runtime stack would appear as follows:

	(rest of stack)	
	X:	
	Y:	
fp	**control link**	**Activation record of call to p**
	return address	
	a:	
	I:	**Allocated area for block A**
	x:	
sp	**J:**	
	Free space	

And just after entry to block B it would look as follows:



Activation record of call to p

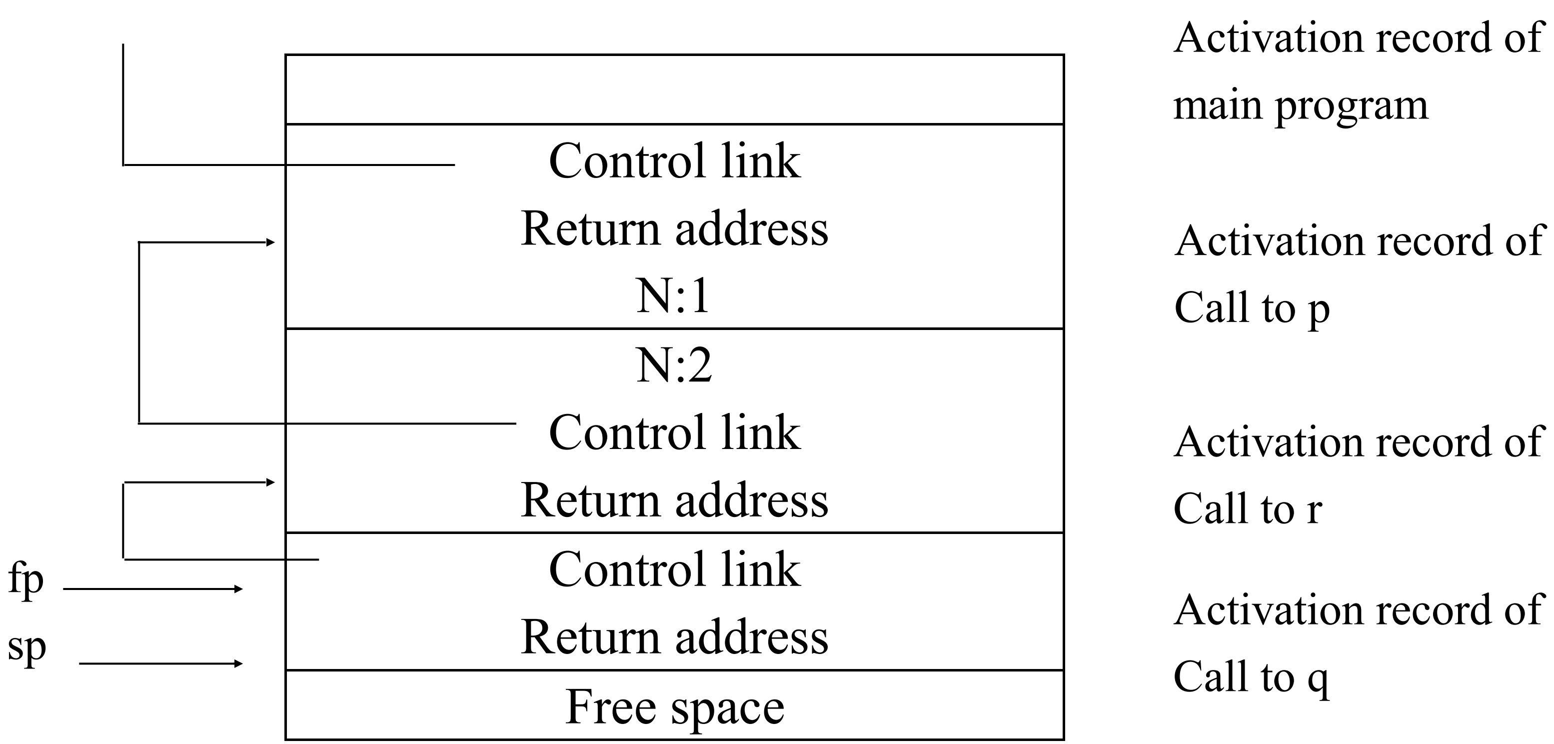
Allocated area for block B

7.3.2 Stack-Based Environment with local Procedures

- Consider the non-local and non-global references
- Example: Pascal program showing nonlocal, nonglobal reference

```
Program nonlocalRef;
Procedure P;
Var N: integer;
        Procedure Q;
        Begin
        (* a reference to N is now non-local andnon-global *)
        end; (* q*)
        Procedure R(N: integer);
        Begin
          Q;
        End;(*r *)
Begin(*p*)
         N:=1;
         R(2);
End ;(*p*)
Begin (* main*)
        P;
End.
```

The runtime stack for the program above:

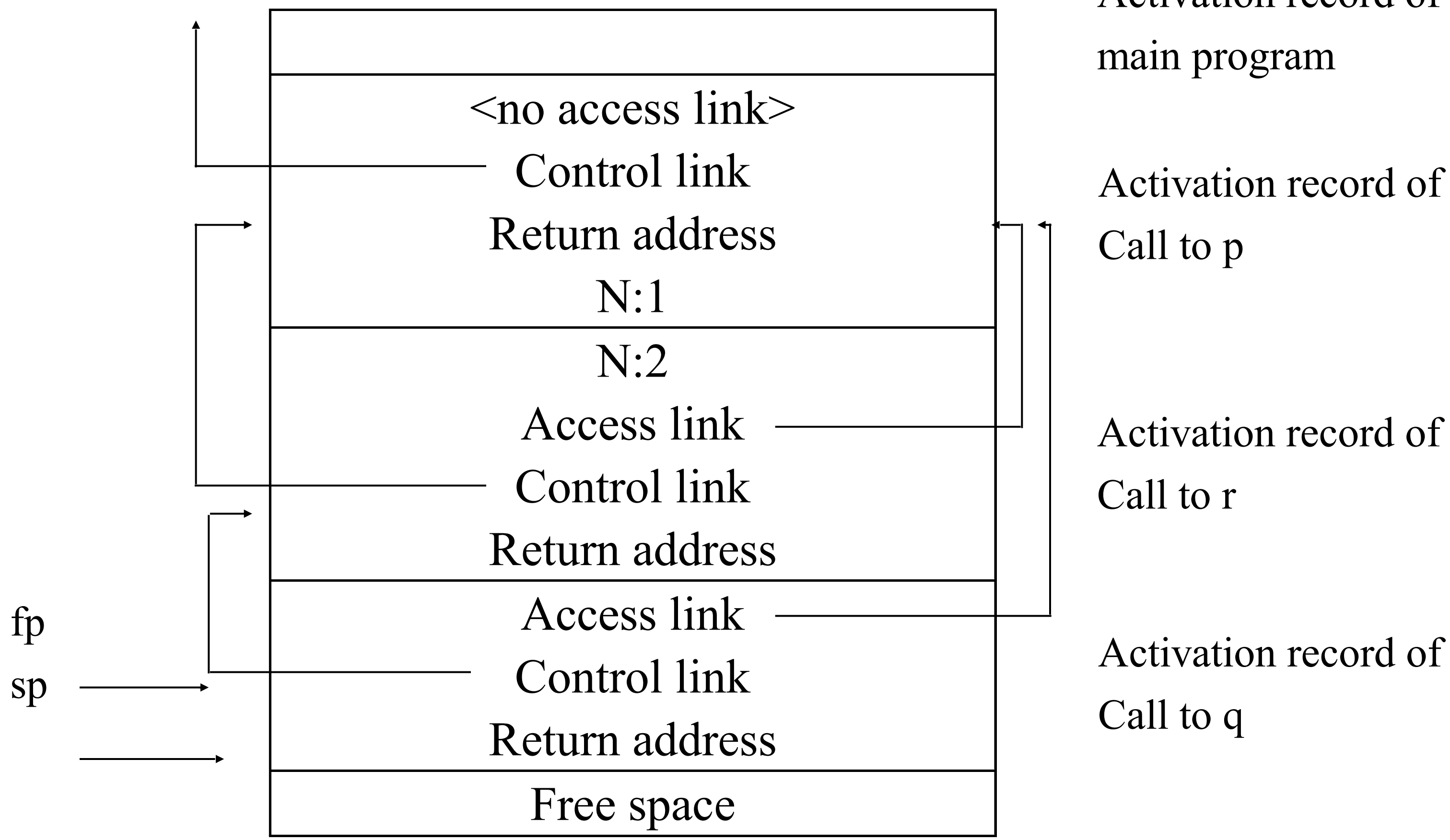


- **N cannot be found using any of the bookkeeping** information that is kept in the runtime environment up to now

- To solve the above problem about variable access, we add an extra piece of bookkeeping information called the ***access link*** to each activation record

- **Access link** represents the defining environment of the procedure;
- **Control link** represents the calling environment of the procedure.

The runtime stack for the program above with access links added.



- **Note:**
  - The activation record of procedure p itself contains no access link, as any nonlocal reference with p must be a global reference and is accessed via the global reference mechanism
  - This is the simplest situation, where the nonlocal reference is to a declaration **in the next outermost scope.**

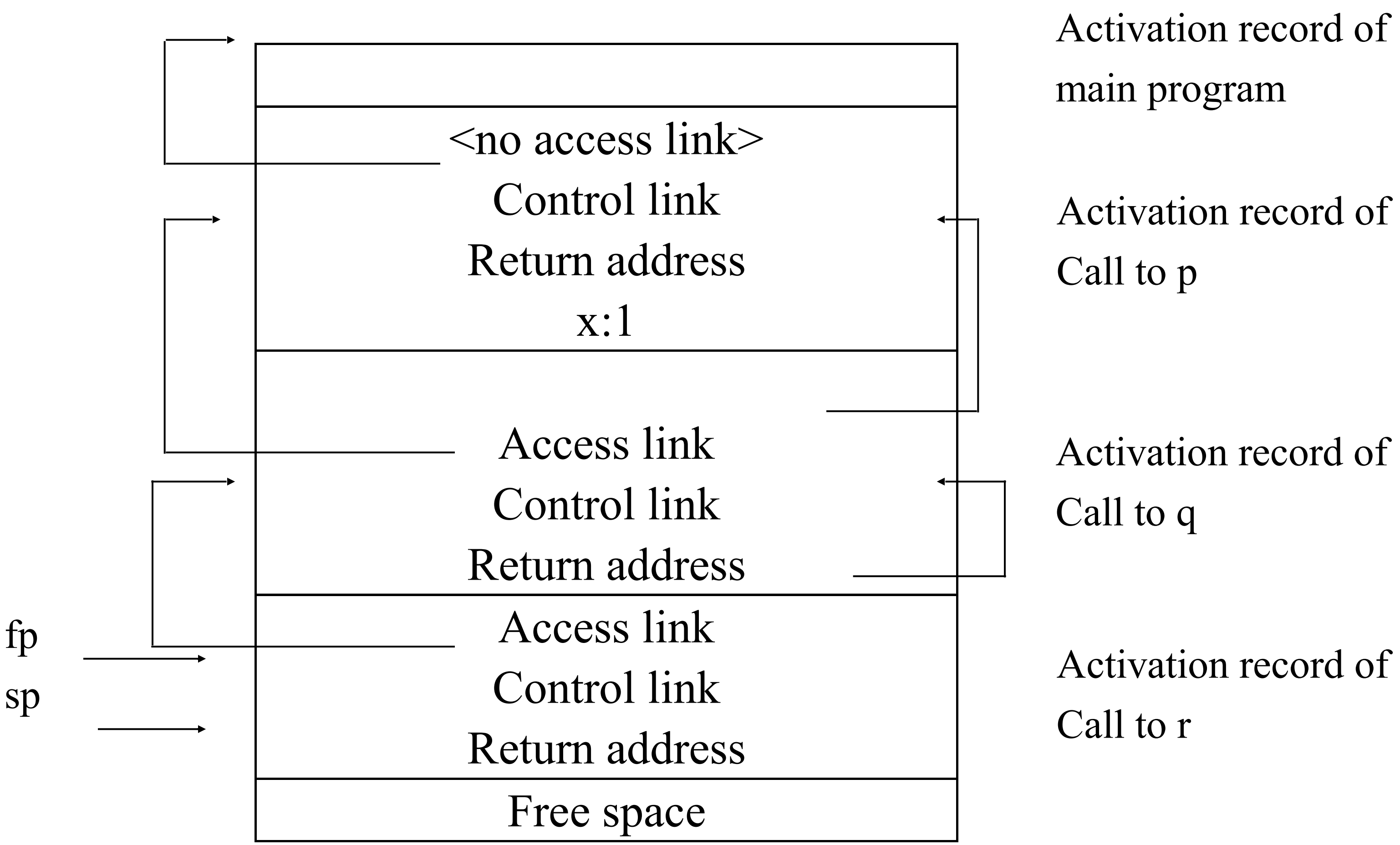
Example: Pascal code demonstrating access chaining

```
Program chain
Procedure p;
Var x: integer;
        Procedure q;
                Procedure r;
                Begin
                        X:=2;
                        …
                        if …then p;
                end;(*r*)
        begin
                r;
        end;(*q*)
begin
        q;
end;(*p*)
```

```
begin(* main *)
        p;
end.
```

- In this code, the assignment to x inside r, which refers to the x of p, must **traverse two scope levels** to find x
- In this environment, x must be reached by following tow access links, a process that is called **access chaining**

The runtime stack after the first call to r:



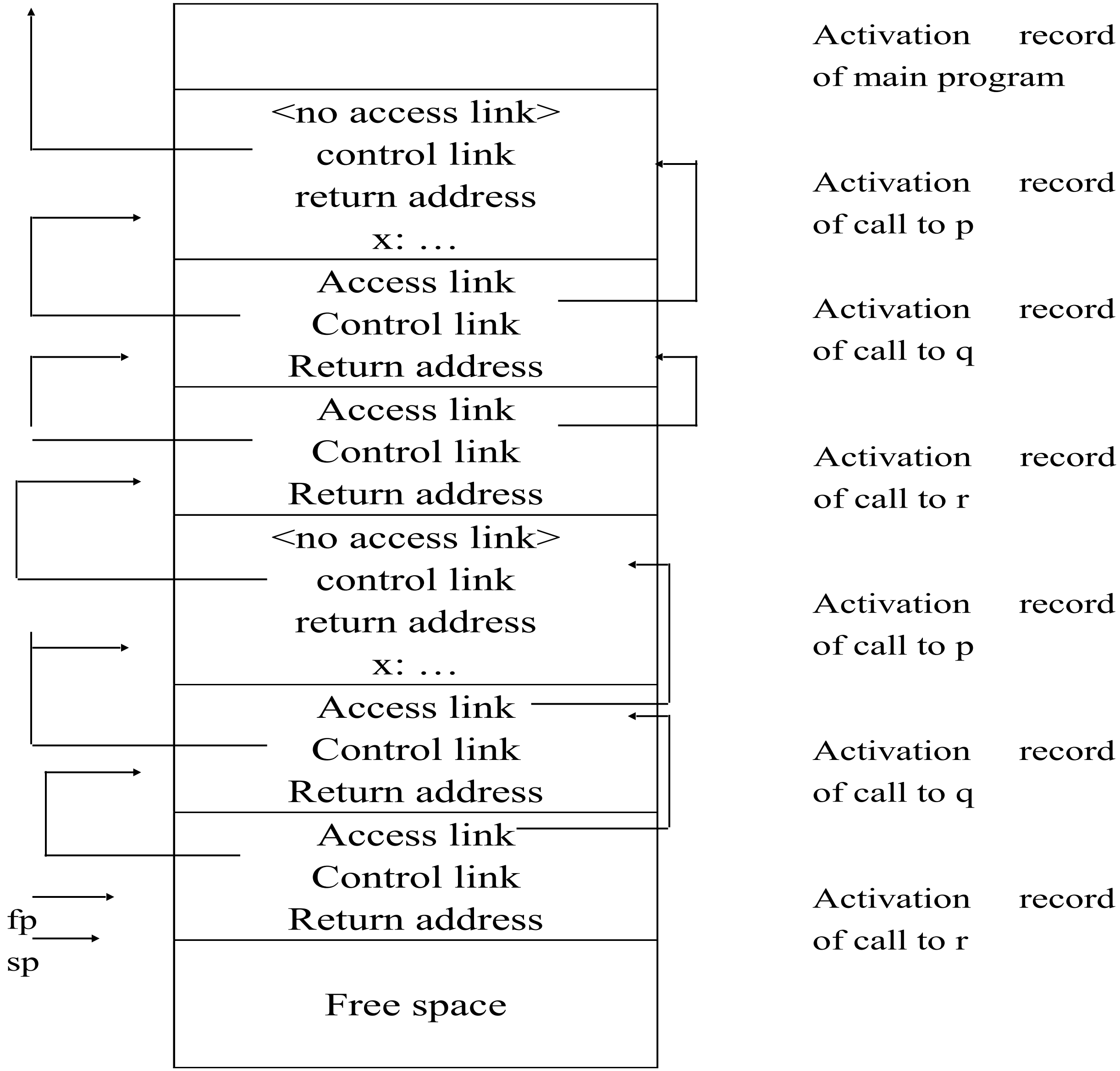
- Note:
  - The **amount of chaining**, **determined by** comparing the nesting level at the point of access with one of the declaration of the name
  - In the above situation, the assignment to x is at nesting level 3, and x is declared at the nesting level 1, so two access links must be followed
  - However, the access chaining is an inefficient method for variable access, for each nonlocal reference with a large nesting difference, a lengthy sequence of instruction must be executed.
  - There is a method of implementing **access links in a lookup table indexed by nesting level**, called **display**, to avoid the execution overhead of chaining.

- **The calling sequence**
- The changes needed to implement access links:

  (1) The access link must be pushed onto the runtime stack just before the fp during a call

  (2) The sp must be adjusted by an extra amount to remove the access link after an exit
- How to find the access link of a procedure during a call?
  - Using the (compile-time) nesting level information attached to the declaration of the procedure
  - Generate an access chain as if to access a variable at the same nesting level
  - **The access link and the control link are the same, if the procedure is local**

Given the code of **PROGRAM CHAIN** the runtime stack after the second call to r (*assuming a recursive call to p*) would look as follows:

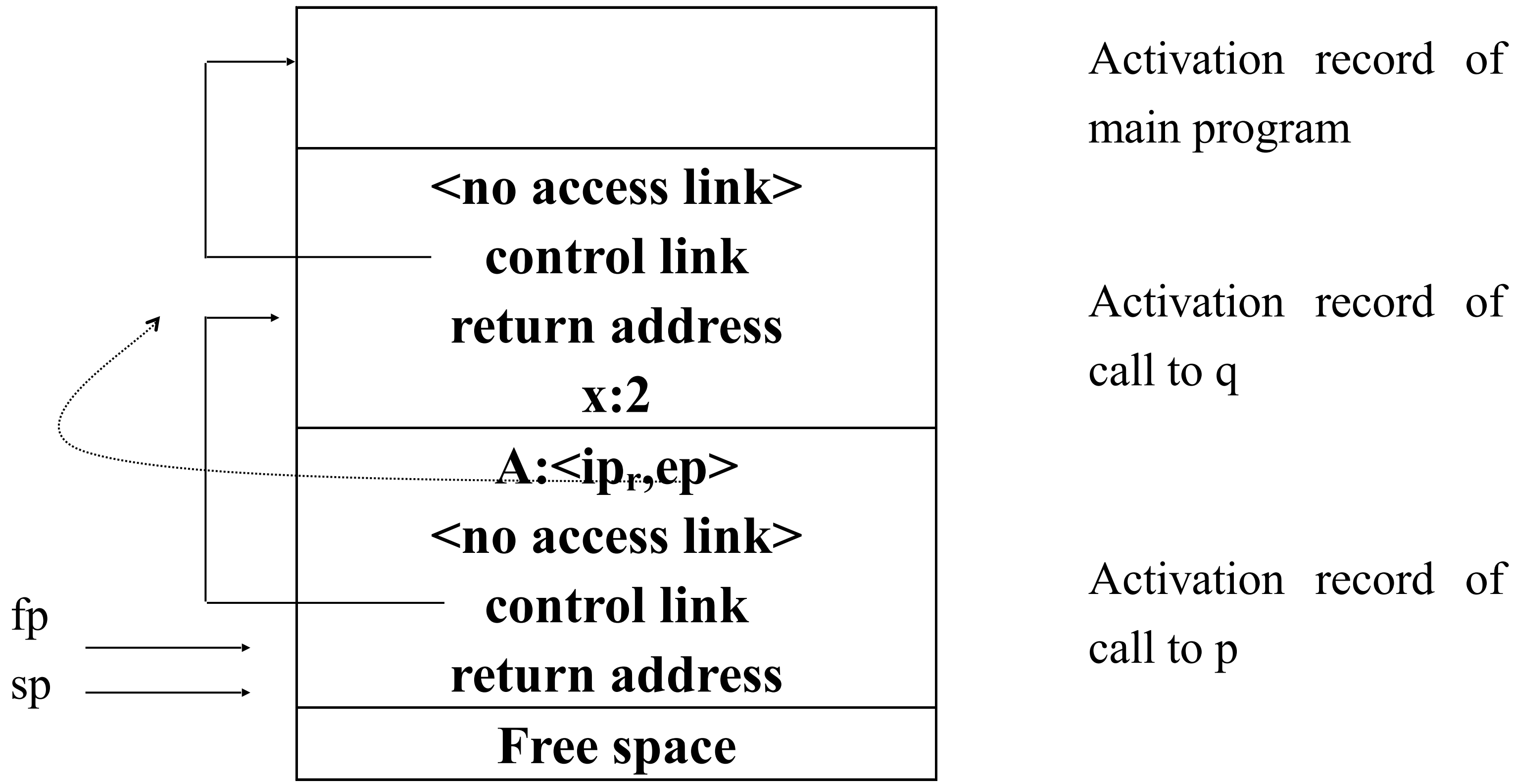


7.3.3 Stack-Based Environment with Procedure Parameters

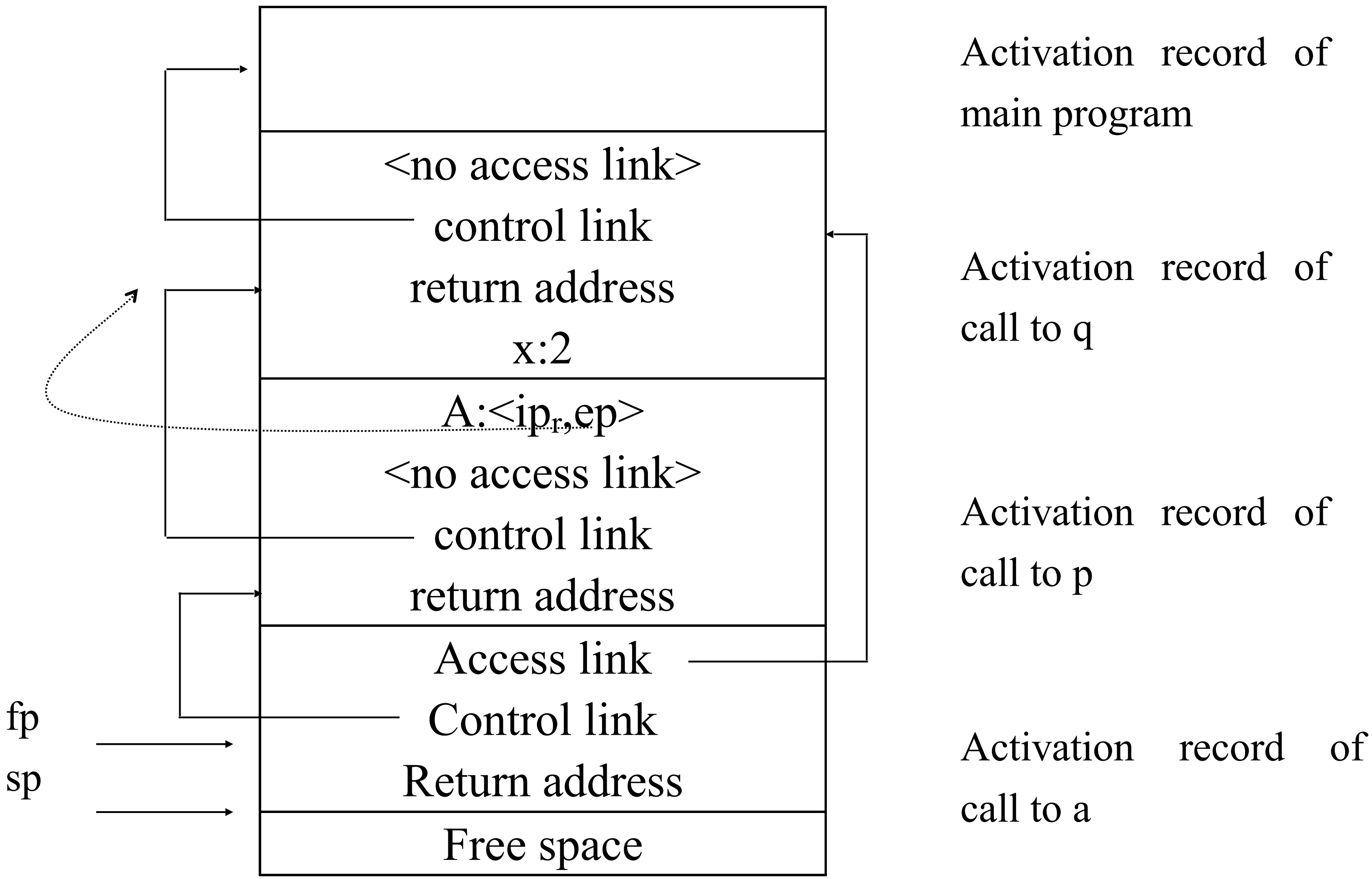
Example 7.7 Consider the standard pascal program of Figure 7.14, which has a procedure p, with a parameter *a* that is also a procedure.

```
Program closureEx(output)
Procedure p (procedure a);
Begin
        a;
end;
procedure q;
var x:integer;
        procedure r;
        begin
                writeln(x);
        end;
begin
        x:=2;
        p(r);
end; (* q*)
begin (* main *)
        q;
end.
```

The runtime stack just after the call to p in the code of Figure 7.14

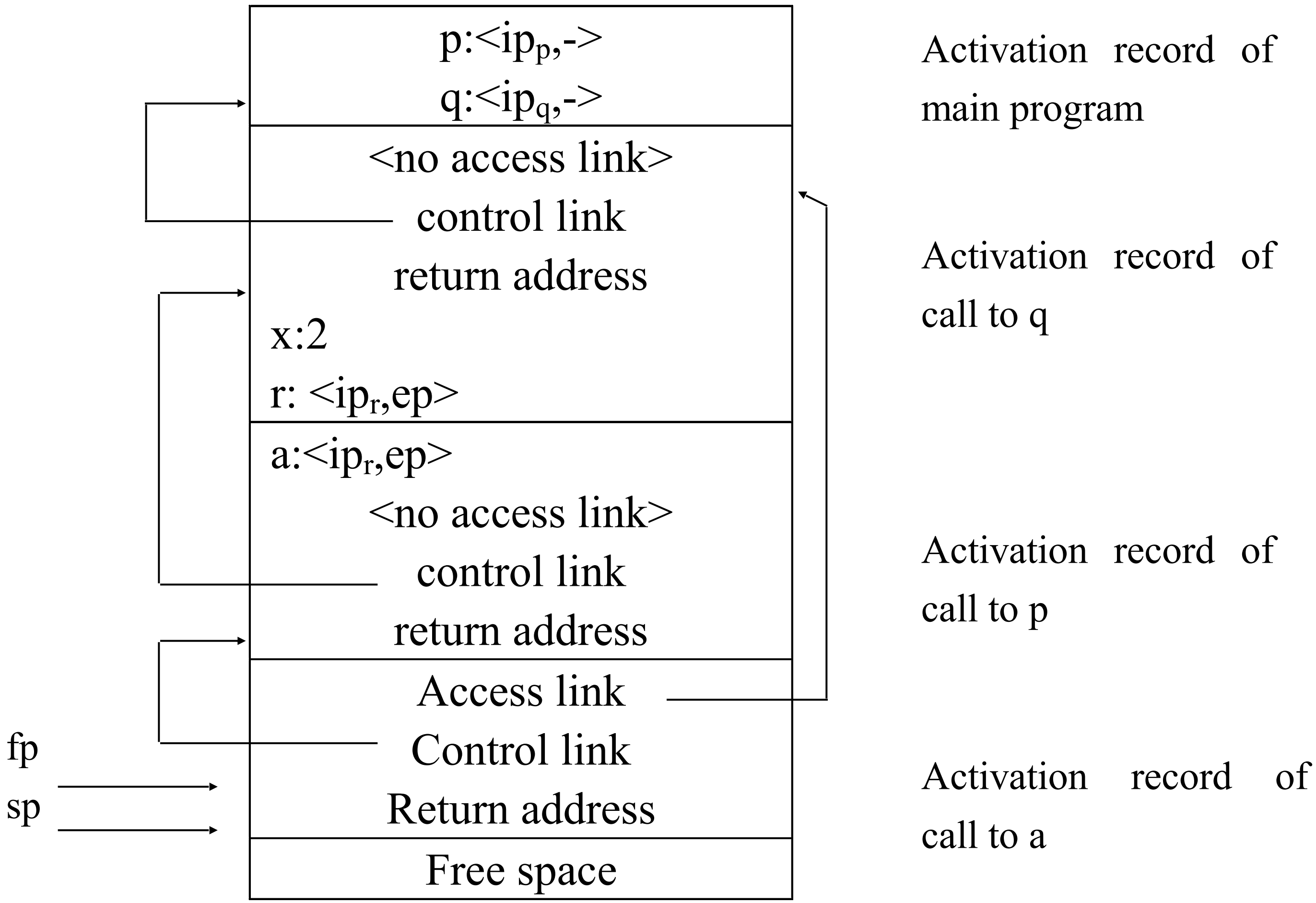


The runtime stack just after the call to a in the code of Figure 7.14



- Note:
    - The calling sequence must distinguish clearly between ordinary procedures and procedure parameters
    - When calling ordinary procedure, fetch the access link using the nesting level and jump directly to the code of the procedure
    - A procedure parameter has its access link stored in the local activation record, and an indirect call must be performed to the ip stored in the current activation record
    - If all procedure values are stored in the environment as closures, the following page shows the environment

Runtime stack just after the call to a in the code of Figure 7.14 with all procedures kept as closures in the environment



End of Part One

THANKS