Advanced function calling techniques

1. revision of activation records
2. nested functions
3. function parameters
4. functions as results
5. curried functions

revision of activation records

convention used in diagrams in this section

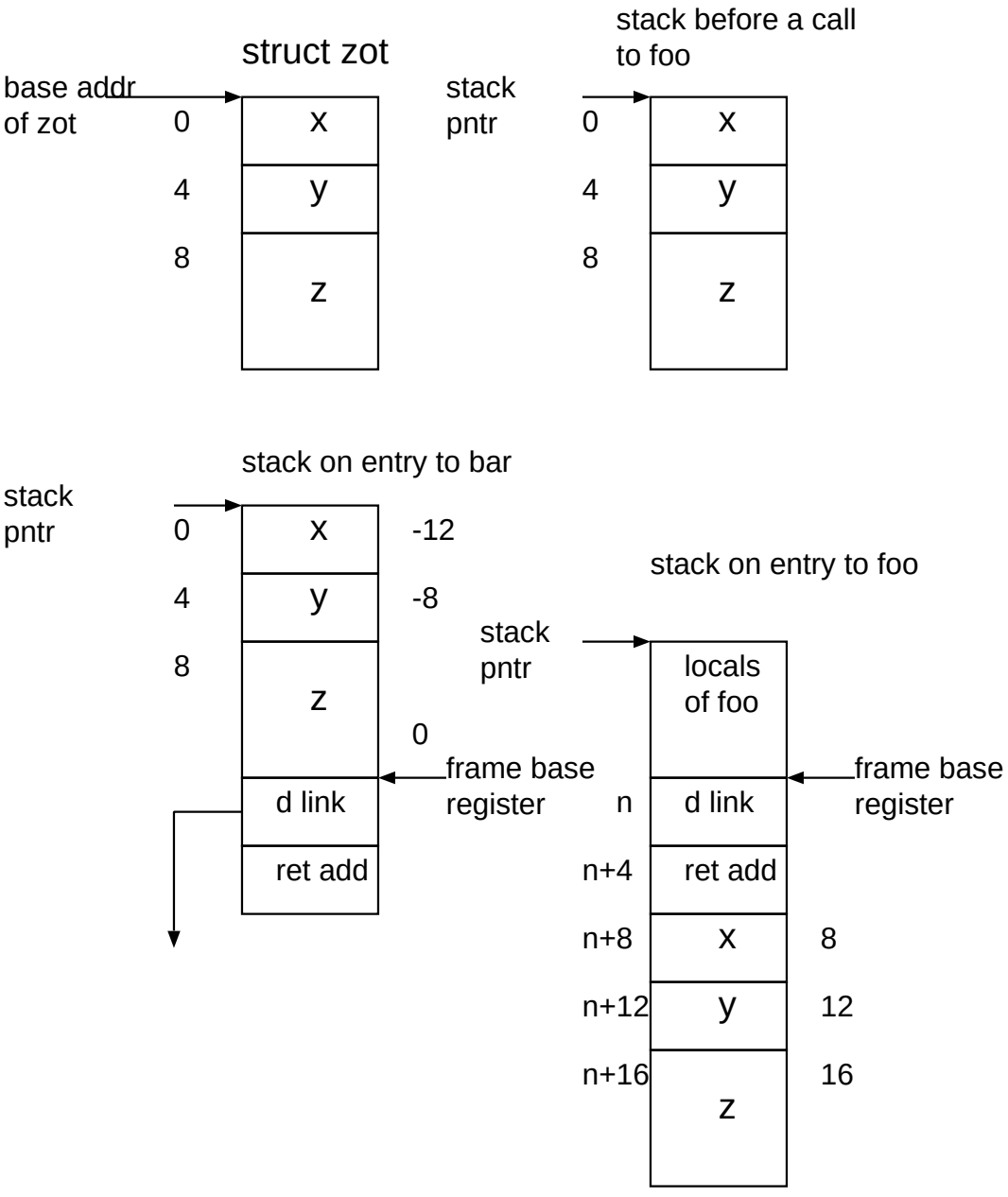
Low addresses are show at the top of the page

high addresses at the bottom

consider:

```
struct{int x,y;double z}zot;
int foo(int x, y;double z};
void bar()
{ int x, y;
  double z:
}
```

Stack frames and structures



Note that the addresses of parameters and variables can be specified relative either to the frame pointer or to the stack pointer. If your code does not dynamically push things onto the stack or if your compiler keeps track of the stack position, then the SP register may be prefered.

Key points:

If you address via the frame pointer (EBP) then the parameters have +ve addresses and the locals have -ve addresses.

If you address using the stack pointer they all have +ve addresses.

If you use the SP you have to take into account temporaries that you push on the stack.

Var Params

We have been assuming value parameters.

If we have var parameters (parameters which can be assigned to changing the value of the actual parameter) then the address of the parameter rather than the value of the parameter has to be passed on the stack. The compiler then places and extra level of indirection onto the addressing of the parameter.

Nested Functions

Consider the following Extended-Hi example where we allow function nesting.

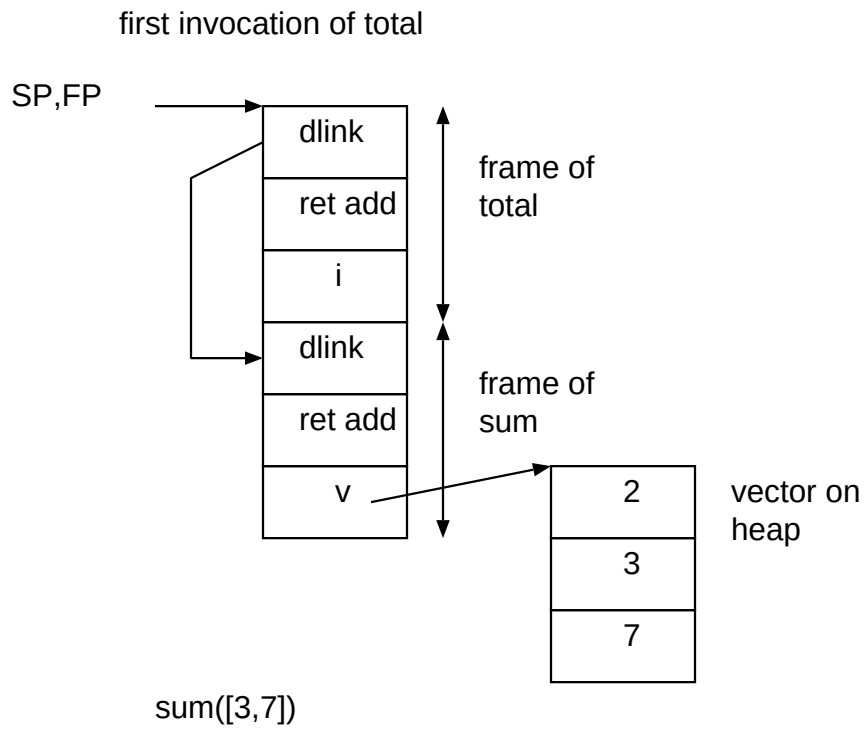
```
sum(vec1 v->scalar)
{
  total(scalar i->scalar)
  if i<1 then 0 else v[i]+total(i-1);
  total(length(v))
}
```

Total recurses on i, but each invocation accesses the same copy of v.

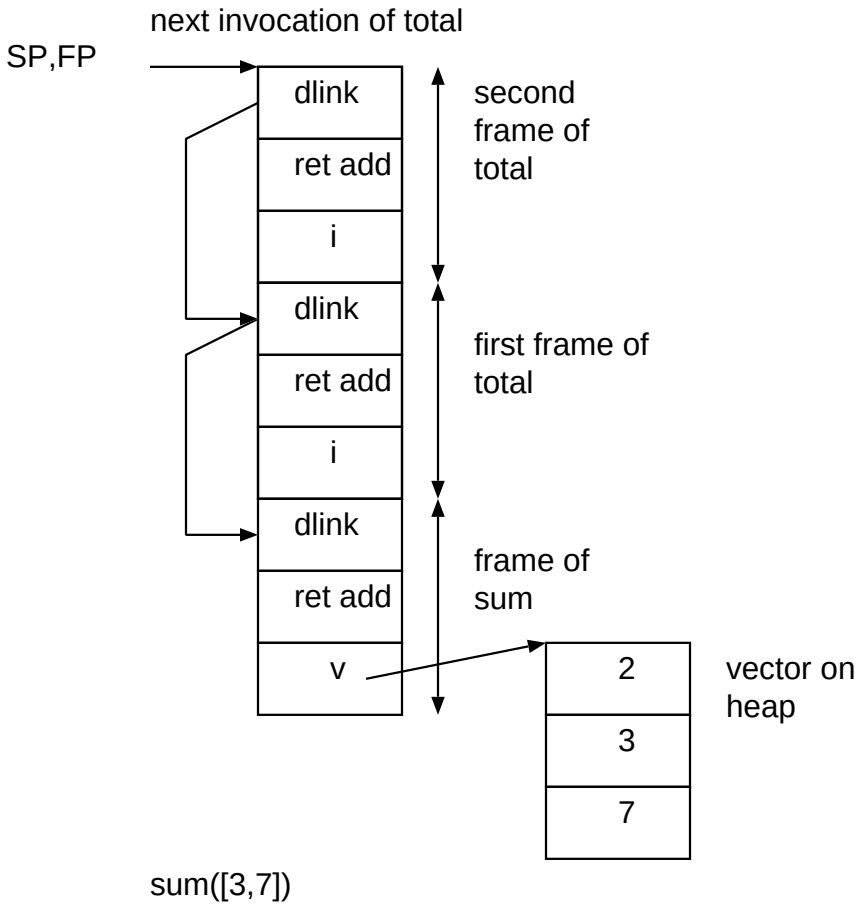
Can we use the d-link to access v?

NO

Consider the following:



At this point we can access v at mem[dlink+8], but what happens on the next recursion?



if we use mem[dlink+8] we get the previous version of i, v is now at mem[mem[dlink]+8]

We need an alternative approach lets look at 3 practical alternatives:

- Displays
- Static Links
- Lambda Lifting

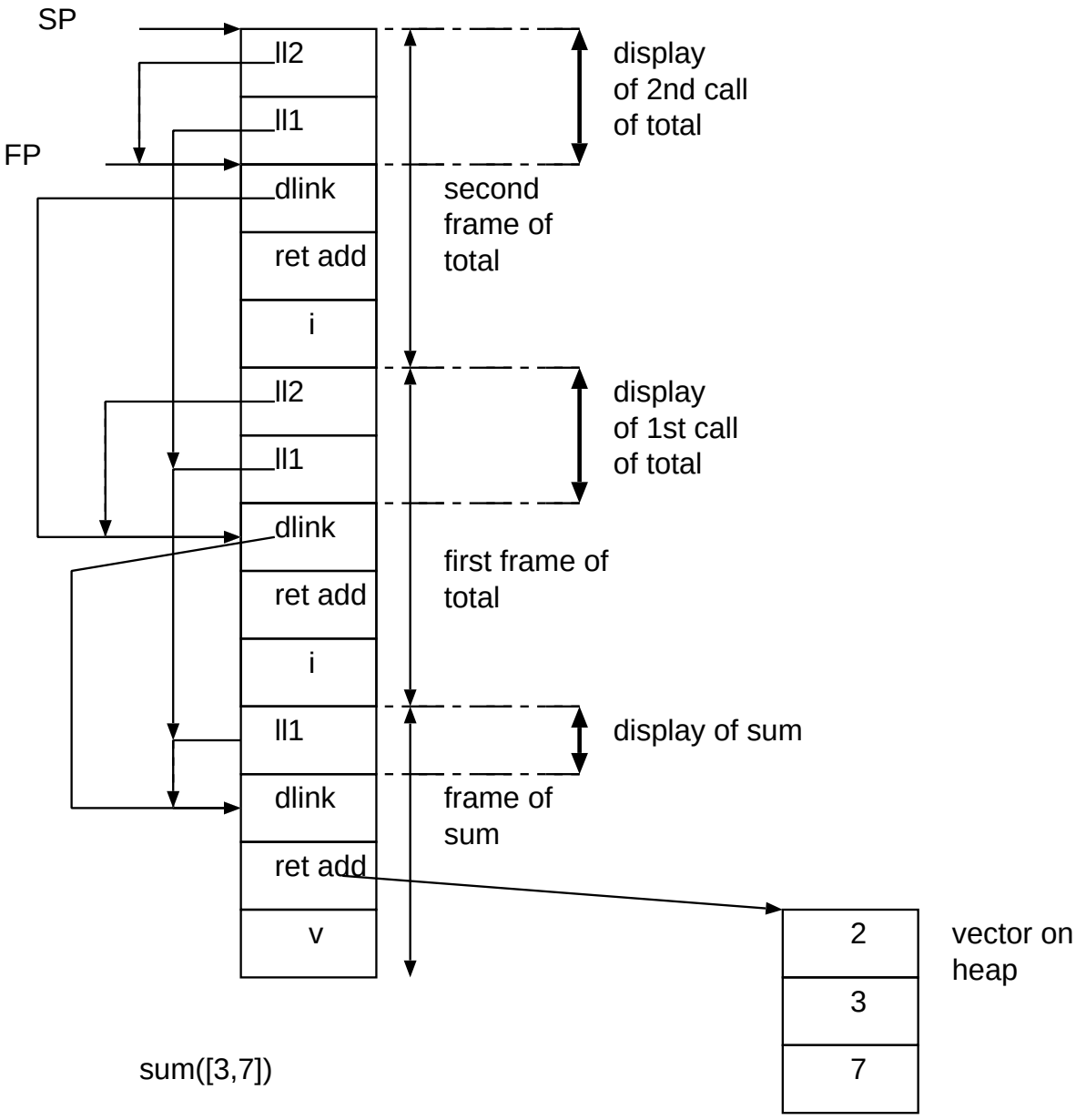
Displays can use the Enter instruction defined as:

```
enter storage,level
 push ebp
 temp:=esp
 if level>0
 then
      repeat (level-1) times
        ebp:=ebp-4
        push dword[ebp]
      end repeat
      push temp
 fi
 ebp:=temp
 esp:=esp - storage
```

Up to now we have assumed procedures use

```
enter xxx,0
```

Consider the effect of using enter 0,1 for sum and enter 0,2 for total :



All variables are now addressed as a pair (lexlevel,offset), where an outer level function is lexical level 1, the first nested function is lexical level 2 etc.

A parameter can now be addressed as

```
mem[ display[lexlevel]+offset]
```

The display is an array in memory at the start of the current frame. Using this notation, parameter i is always addressed as

```
mem[display[2]+8]= mem[ mem[fp-8]+8]
```

and v is always at

```
mem[display[1]+8]
```

Optimisations

FP always points to the current lexical level so at lexical level 2 we have

```
   mem[display[2]+8]
= mem[ mem[fp-8]+8]
= mem[fp+8]
```

Likewise we can chose to cache other display values in registers so avoiding repeated dereferencing of the display on stack.

Static Links

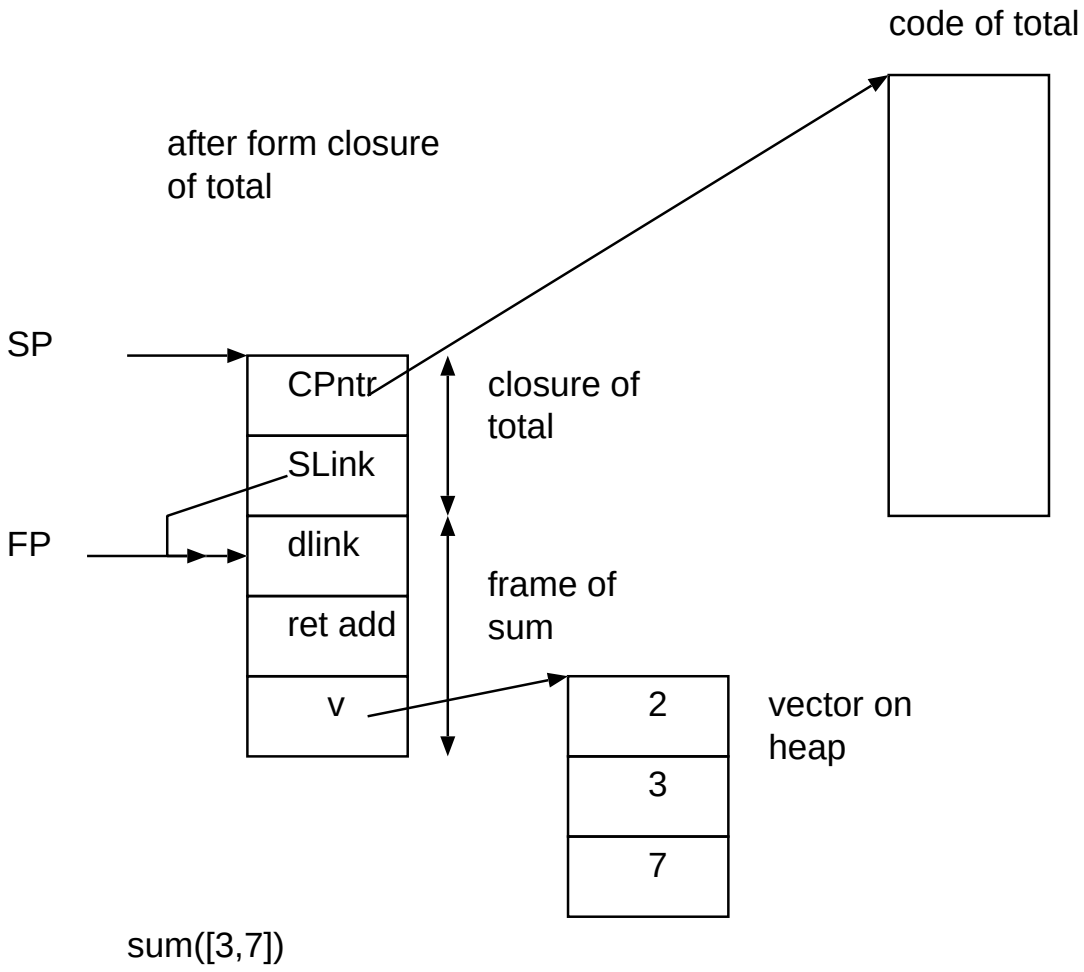
An alternative approach is to use a static chain as well as a dynamic chain. This substitutes a linked list for the array used in the display.

When an nested procedure declaration like *total* is encountered, the compiler plants code to

```
push ebp   ; frame pointer
push total ; start address of total
```

This pair of words is called a *closure*

It binds the code of *total* with the non-local variables accessible to that code, namely the frame of *sum*.


code of total
after form closure
of total
SP
CPntr
SLink
FP
dlink
ret add
v
closure of
total
frame of
sum
2
3
7
vector on
heap
sum([3,7])

Calling with closures

to call a closure we

1. push the parameters on the stack

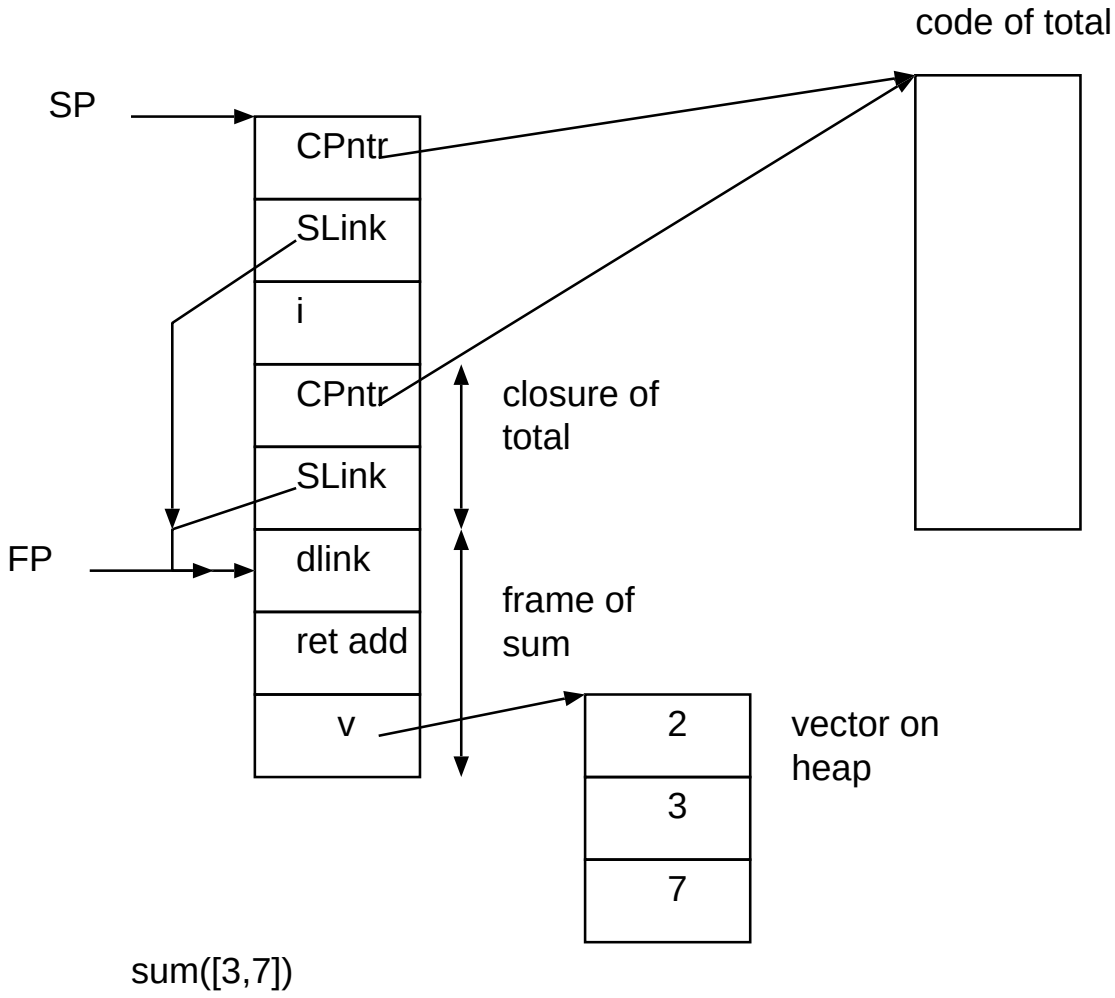
2. push both words of the closure onto the stack

3. perform a RET instruction which transfers control to the address last pushed on the stack.

Let us look at this with a diagram:

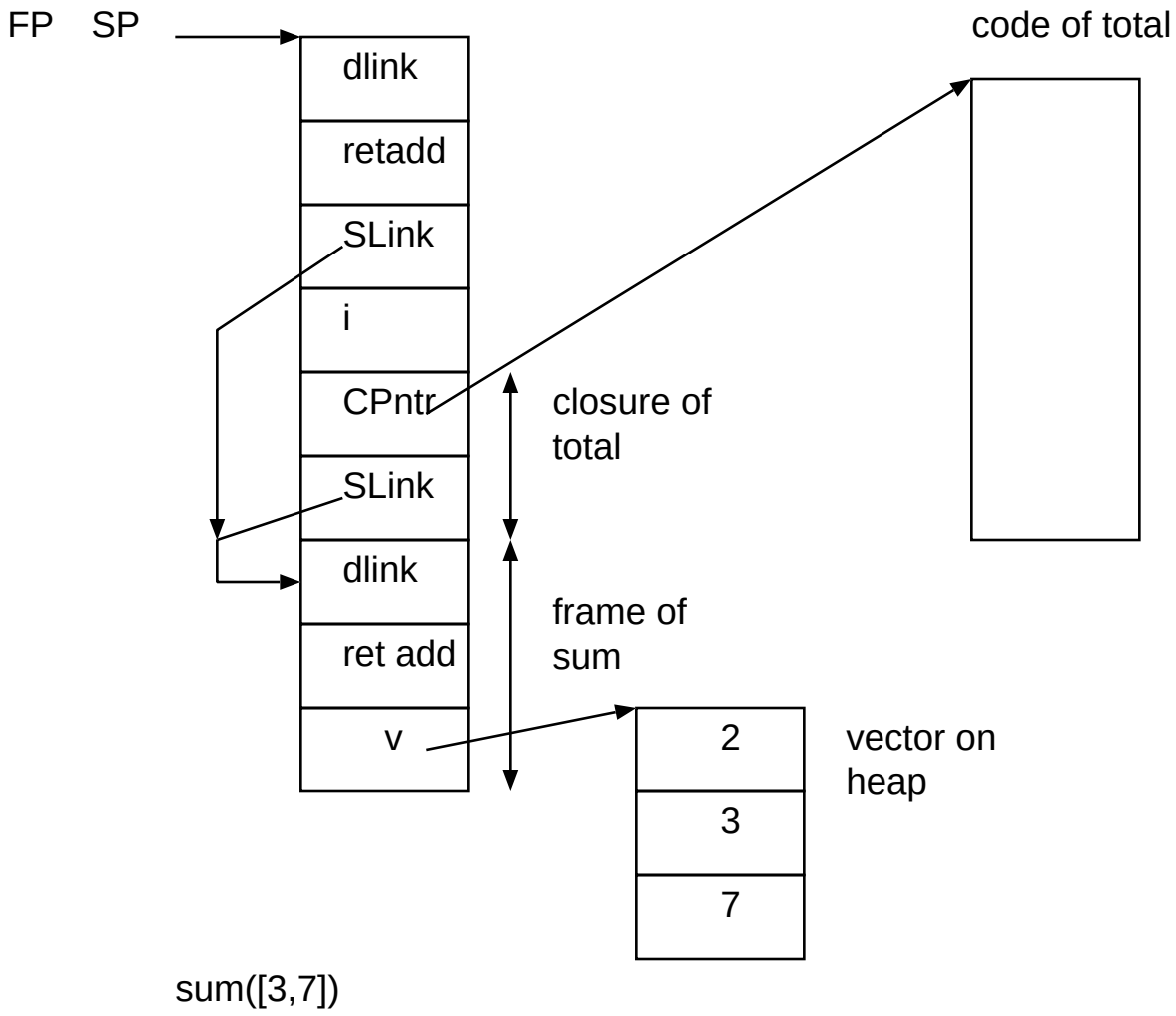
after setting up the call to total

after pushing params
but before calling total



we then execute RET and enter total for the first time:

on entering the first
call of total



Within total the parameter v is accessed as

```
mem[slink+8]=mem[mem[fp+8]+8]
```

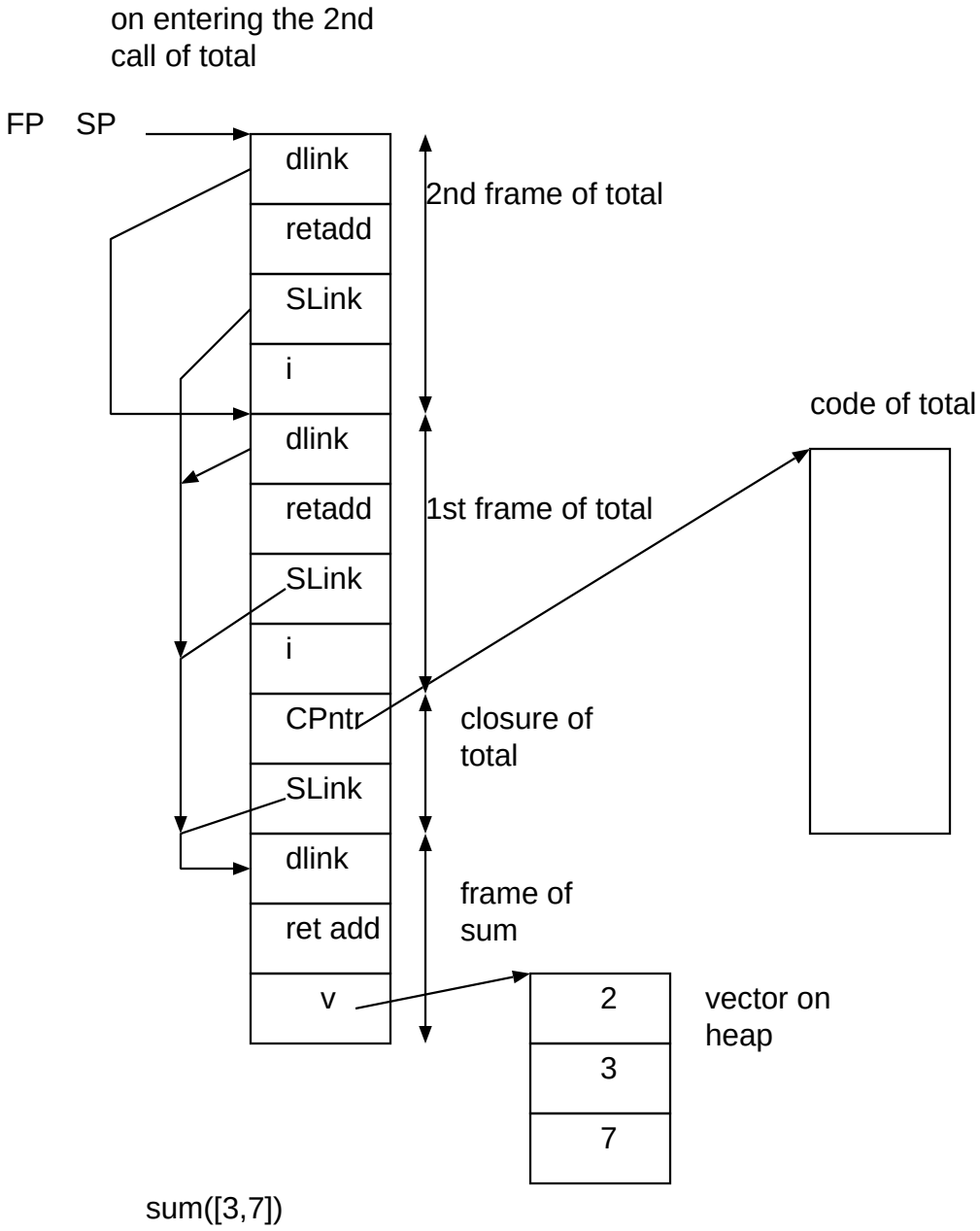
We then push the closure of total back on the stack for the next call. Note that total is a *local variable of sum*. As such it it accessed via the static link as

```
mem[slink-8]
```

so we can push it on the stack and call it with the following

```
mov eax,[fp+8]  ; get the slink
lea eax,[eax-8] ; get the address of totals closur
push [eax+4]    ; push the first word
push [eax]      ; push the second word
ret             ; enter total
```

This results in the following stack configuration



Note that the address of v remains

mem[slink+8]=mem[mem[fp+8]+8]

Lambda Lifting

The idea here is to make all functions global but to add additional parameters to get at local variables.

In the compiler we do a preprocessing pas which transfroms:

```
sum(vec1 v->scalar)
{
 total(scalar i->scalar)
  if i<1 then 0 else v[i]+total(i-1);
  total(length(v))
}
```

to

```
sum(vec1 v->scalar) sum$total(v,length(v));
sum$total(vec1  v,scalar i->scalar)
  if i<1 then 0
  else v[i]+sum$total(v,i-1);
```

and then compile as normal. We use the symbol $ to create variable names which can not clash with any declared by the programmer. (first pass would throw out any such names).

The effect is to remove any nested variable accesses.

For imperative languages we pass the addresses of the variables instead of the values of the variables.

Functions as parameters

The three methods described above are all equally good for the purposes of handling nesting. Only one of them works OK for function parameters.

Let us look at a toy example in Hi. Consider a function that will integrate another function over a range $a..b$. i.e we want

$$f(g,a,b) = \int_a^b g(x)dx$$

if we allow function parameter we could approximate this with

```
f((scalar->scalar)g,scalar a,scalar b->scalar)
  sum(g(iota(b-a)+a-1));
```

Suppose we now use f in the following:

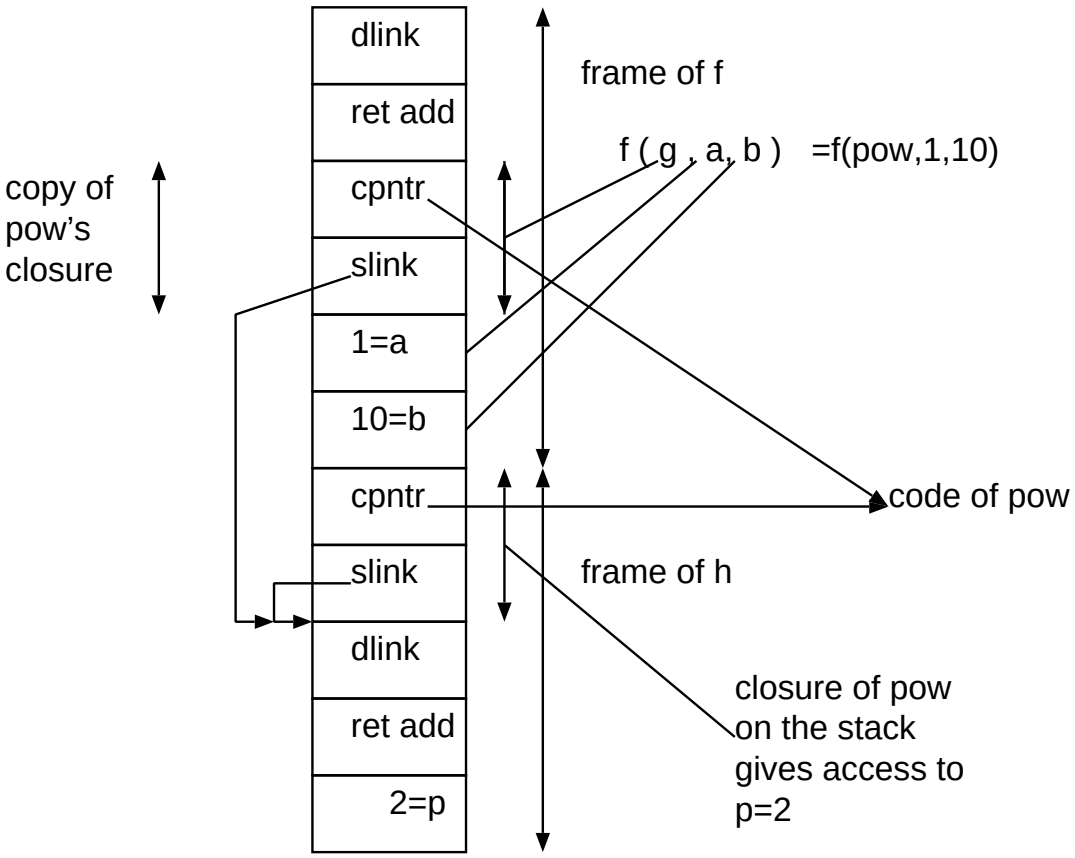
```
h(scalar p->scalar)
{
  pow(scalar x->scalar)
  {
    pwr(scalar i->scalar)
    if i=0 then 1 else x*pwr(i-1);
    pwr(p)
  }
  f(pow,1,10)
}
```

This will approximate

$$h(p) = \int_1^{10} x^p dx$$

- Can we use Lambda Lifting?
  - No because different functions g, passed to f, may have different numbers of additional parameters.
- Can we use displays?
  - No, because the display of f will not contain an entry corresponding to the frame of h which calls it. Thus the display set up by pow would be wrong and could not allow access to p.
- Can we use closures?
  - Yes, provided we pass both words of the closure to f, the static chain will work.

- Here is a stack snapshot for h(2)



Note that applytoall would have to be modified to take closures as parameters instead of simple machine addresses.

Functions as results

The next level of complexity comes when we create functions that return functions. I will show how to modify the implementation of closures and activation records to deal with this.

Given the definition of f previously , let us define a function that will allow g to be integrated within a specified range.

```
k(scalar l->((scalar ->scalar)->scalar))
{ i((scalar->scalar)g->scalar)f(g,1,l);i}
```

such that

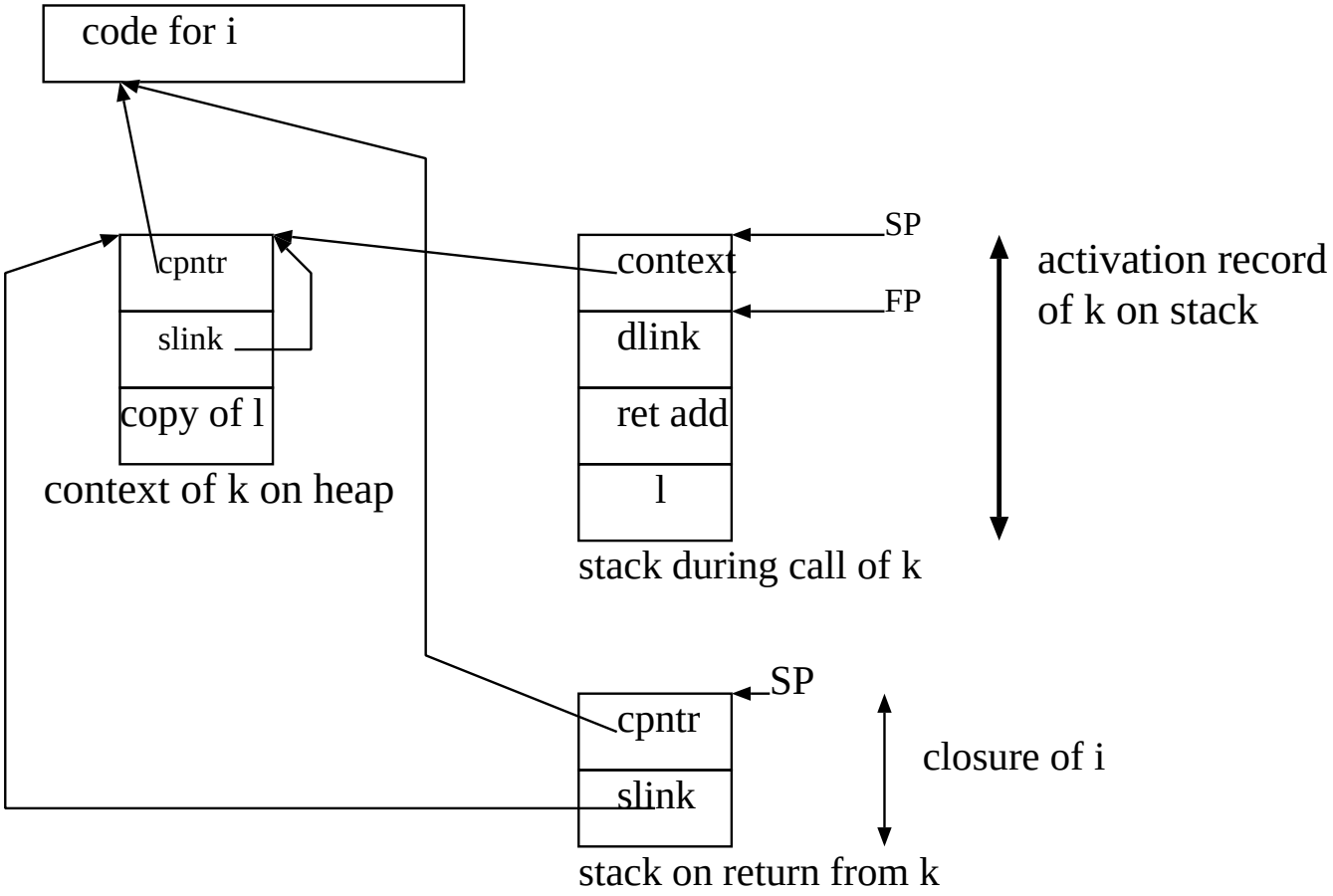
$$k(l)(g) = \int_1^l g(x)dx$$

when *k(l)* is called it returns *i*, and when *i* runs it has to have access to the parameter *l* of *k*,

But parameter l will have been on the stack when k was called but will have been poped from the stack by the time i is called.

This means that the static link will be pointing at a part of the stack that is no longer valid. How to deal with this?

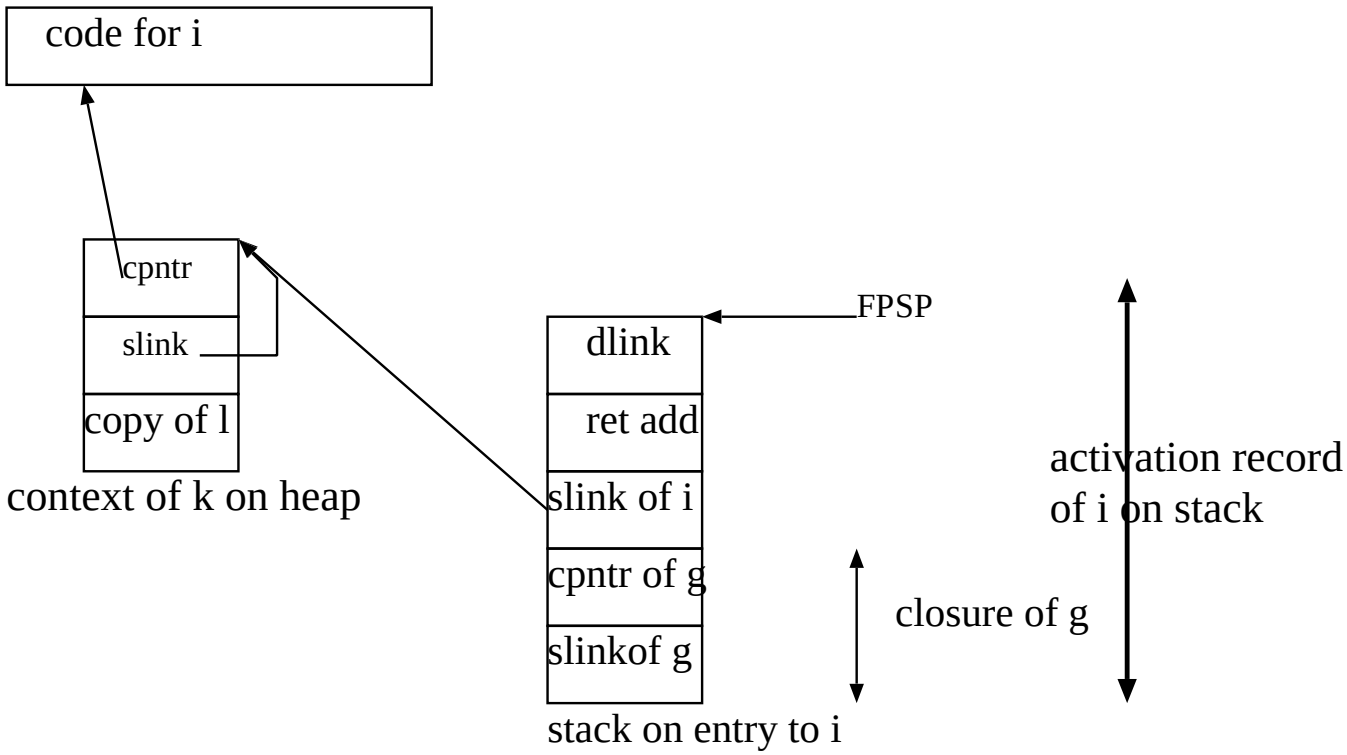
Context on the heap



- The parameters and local store of k are transfered from the stack to the heap,

- parameters have to be copied on entry

On entry to function i



When i runs it now has access to l via its static link as before, except that the block holding l is now on the heap. The address of variables relative to slink will obviously be different in this case.

curried functions

Currying also called partial application involves creating a new function as a result of partially binding the formal parameters of an existing function

for instance given

```
g(scalar->scalar)
```

and

```
f((scalar->scalar),scalar ,scalar ->scalar)
```

then f(,a,b) gives us a function of form

```
((scalar->scalar)->scalar)
```

These can be handled by translating them into appropriate nested functions:

thus

```
m( scalar a,scalar b->((scalar->scalar)->scalar))
 f(,a,b);
```

can be transformed into

```
m( scalar a,scalar b->((scalar->scalar)->scalar))

{ F((scalar ->scalar)G->scalar)f(G,a,b);F}
```

which can be handled other nested functions.

Register Windows

1. Used on some RISC machines

2. Optimised for calling C

3. Uses stack of registers not stack in memory

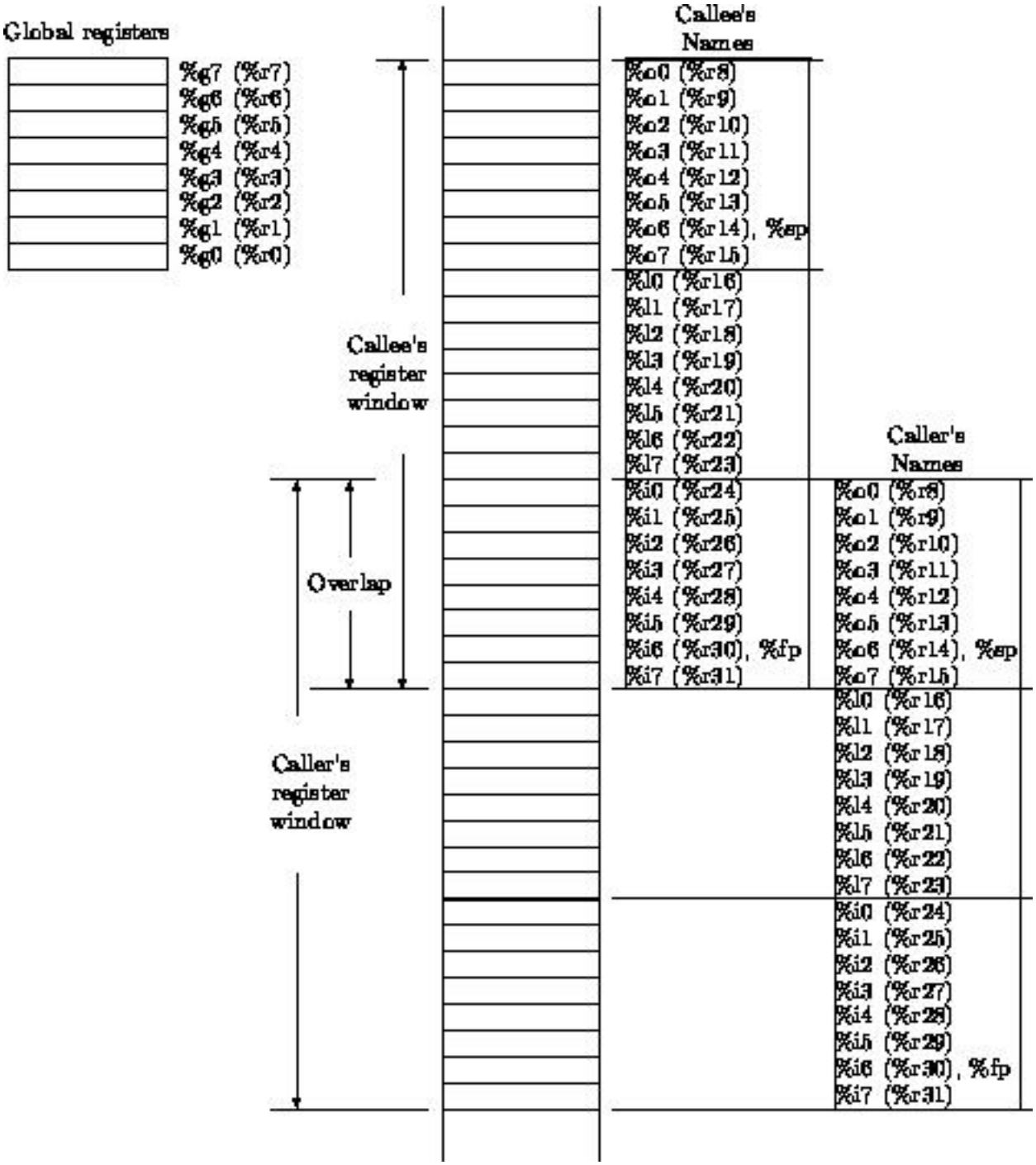
SPARC

Sparc has 32 general purpose integer registers visible to the program at any given time. Of these, 8 registers are global registers and 24 registers are in a register window. A window consists of three groups of 8 registers, the out, local, and in registers. A Sparc implementation can have from 2 to 32 windows, thus varying the number of registers from 40 to 520. Most implentations have 7 or 8 windows. The variable number of registers is the principal reason for the Sparc being "scalable".

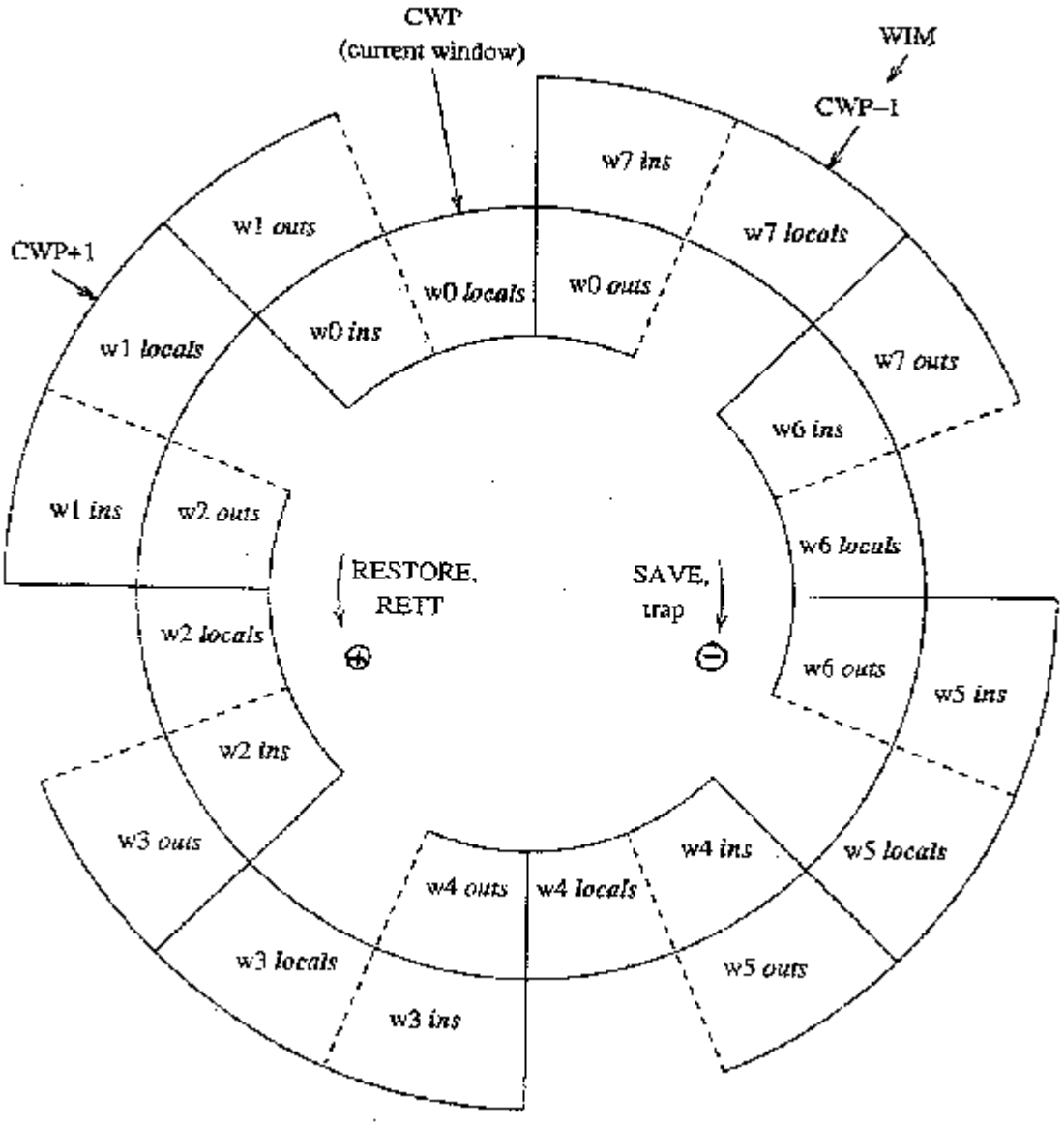
Register Group	Mnemonic	Register Address
global	%g0-%g7	r[0]-r[7]
out	%o0-%o7	r[8]-r[15]
local	%l0-%l7	r[16]-r[23]
in	%i0-%i7	r[24]-r[31]

At any one point the registers in use are selected by a current register bank register in the cpu.

This is basically a stack within the cpu.


Global registers
%g7 (%r7)
%g6 (%r6)
%g5 (%r5)
%g4 (%r4)
%g3 (%r3)
%g2 (%r2)
%g1 (%r1)
%g0 (%r0)
Callee's Names
%o0 (%r8)
%o1 (%r9)
%o2 (%r10)
%o3 (%r11)
%o4 (%r12)
%o5 (%r13)
%o6 (%r14), %sp
%o7 (%r15)
%l0 (%r16)
%l1 (%r17)
%l2 (%r18)
%l3 (%r19)
%l4 (%r20)
%l5 (%r21)
%l6 (%r22)
%l7 (%r23)
%i0 (%r24)
%i1 (%r25)
%i2 (%r26)
%i3 (%r27)
%i4 (%r28)
%i5 (%r29)
%i6 (%r30), %fp
%i7 (%r31)
Callee's register window
Overlap
Caller's register window
Caller's Names
%o0 (%r8)
%o1 (%r9)
%o2 (%r10)
%o3 (%r11)
%o4 (%r12)
%o5 (%r13)
%o6 (%r14), %sp
%o7 (%r15)
%l0 (%r16)
%l1 (%r17)
%l2 (%r18)
%l3 (%r19)
%l4 (%r20)
%l5 (%r21)
%l6 (%r22)
%l7 (%r23)
%i0 (%r24)
%i1 (%r25)
%i2 (%r26)
%i3 (%r27)
%i4 (%r28)
%i5 (%r29)
%i6 (%r30), %fp
%i7 (%r31)

It is actually implemented as a circular buffer



1. The SPARC approach is only suitable for languages like C that do not allow nesting of functions since the locals of a procedure are invisible to any called procedure.

2. Furthermore it does not support passing the addresses of parameters (Pascal Var Parameters)

3. Finally it costs a huge amount on process swap as all of the register windows have to be stored and reloaded each time there is a context switch.

It is a typical RISC bad idea arising from over concentration on one problem - getting fast uni-process C code.