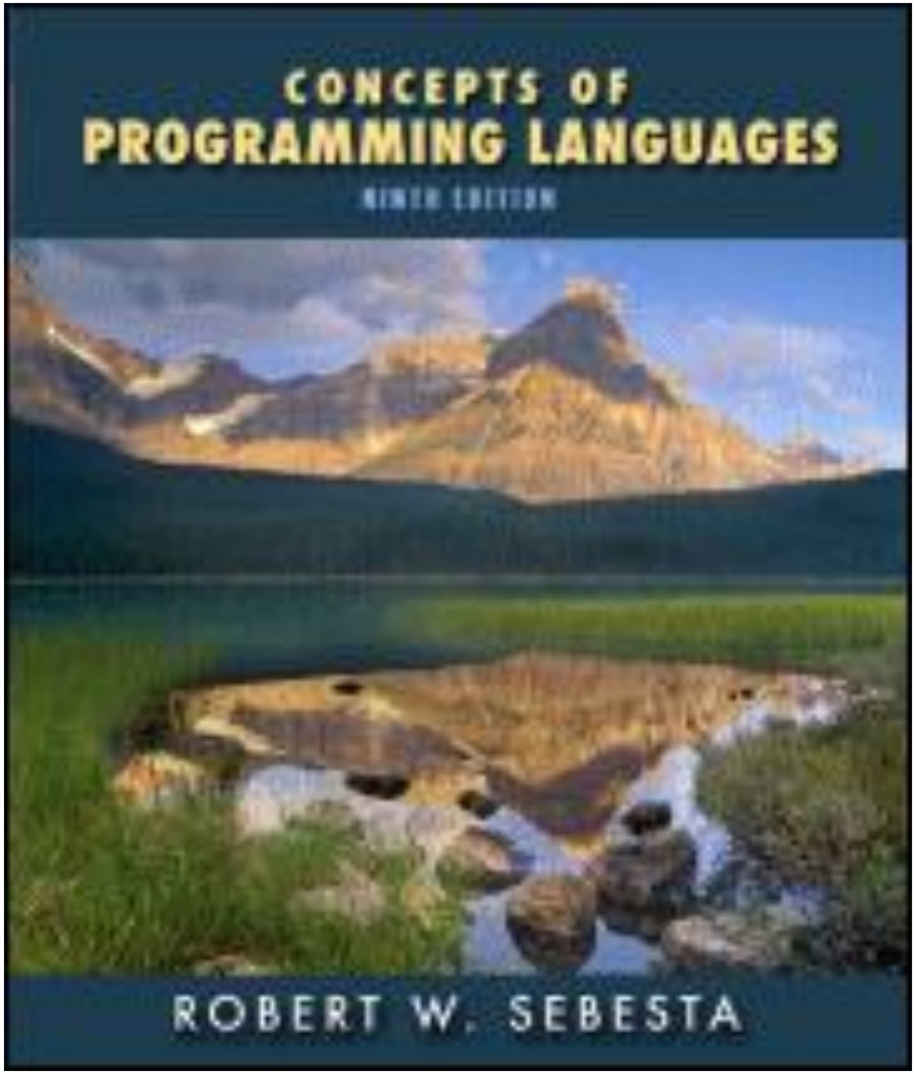
Chapter 10

Implementing Subprograms



Ch10 – Implementing Subprograms

10.1 The General Semantics of Calls and Returns

10.2 Implementing "Simple" Subprograms

10.3 Implementing Subprograms with Stack-Dynamic Local Variables

10.4 Nested Subprograms

10.5 Blocks

10.6 Implementing Dynamic Scoping

10.1 The General Semantics of Calls and Returns

- Subprogram linkage
  - The subprogram call and return operations of a language are together called its *subprogram linkage*
- Actions associated with subprogram calls
  - Parameter passing methods
  - Non-static local variables
  - Execution status of calling program
  - Transfer of control
  - Subprogram nesting (accessing nonlocal variables)

10.2 Implementing "Simple" Subprograms

- "Simple" subprograms
  - Subprograms cannot be nested.
  - All variables are static (thus, recursion isn’t allowed).
  - E.g. Fortran 77
- Activation record
  - The format, or layout, of the noncode part of an executing subprogram is called an **activation record**.
  - An **activation record instance** is a concrete example of an activation record (the collection of data for a particular subprogram activation).
  - When recursion is allowed, an AR may have many ARI’s.
    I shall usually use AR, meaning ARI.

10.2 Implementing "Simple" Subprograms

- Activation records for "simple" subprograms

Figure 10.1

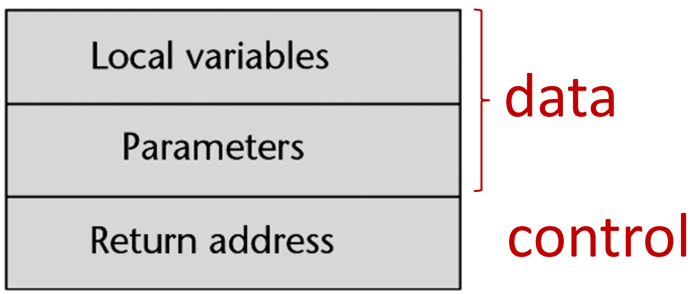
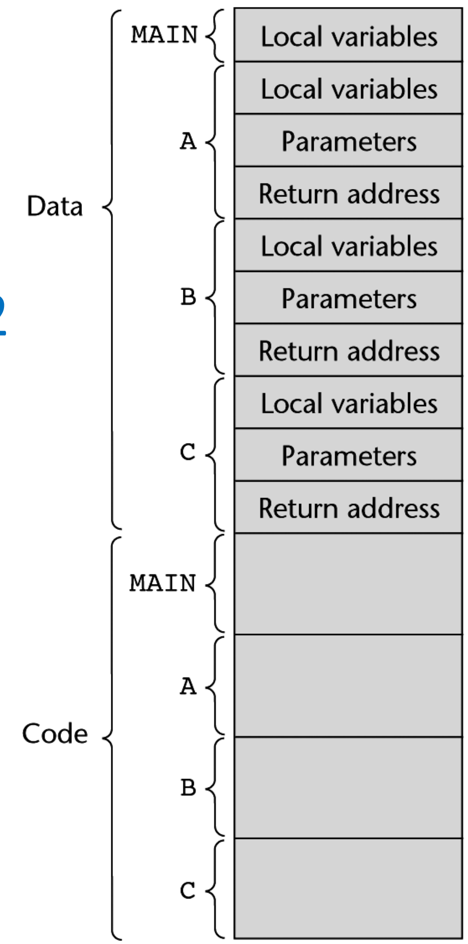


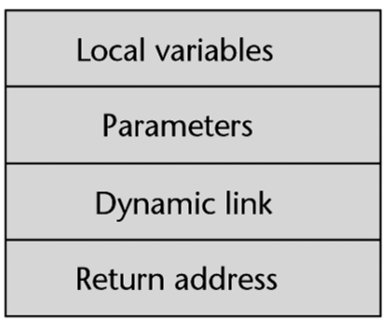
Figure 10.2



- Code and ARs of a program with simple subprograms

  A function’s code and AR are usually attached together.

10.3 Implementing Subprograms with Stack-Dynamic Local Variables

- Subprograms with stack-dynamic local variables
  - Subprograms cannot be nested
  - Variables are non-static (thus, recursion is supported)
  - E.g. C/C++
- More complex activation record
  - 

    Dynamic link (pointer)

    1 point to the caller's AR

    2 for maintaining the run time stack
  - The compiler generates code to cause implicit allocation and deallocation of local variables from the runtime stack.

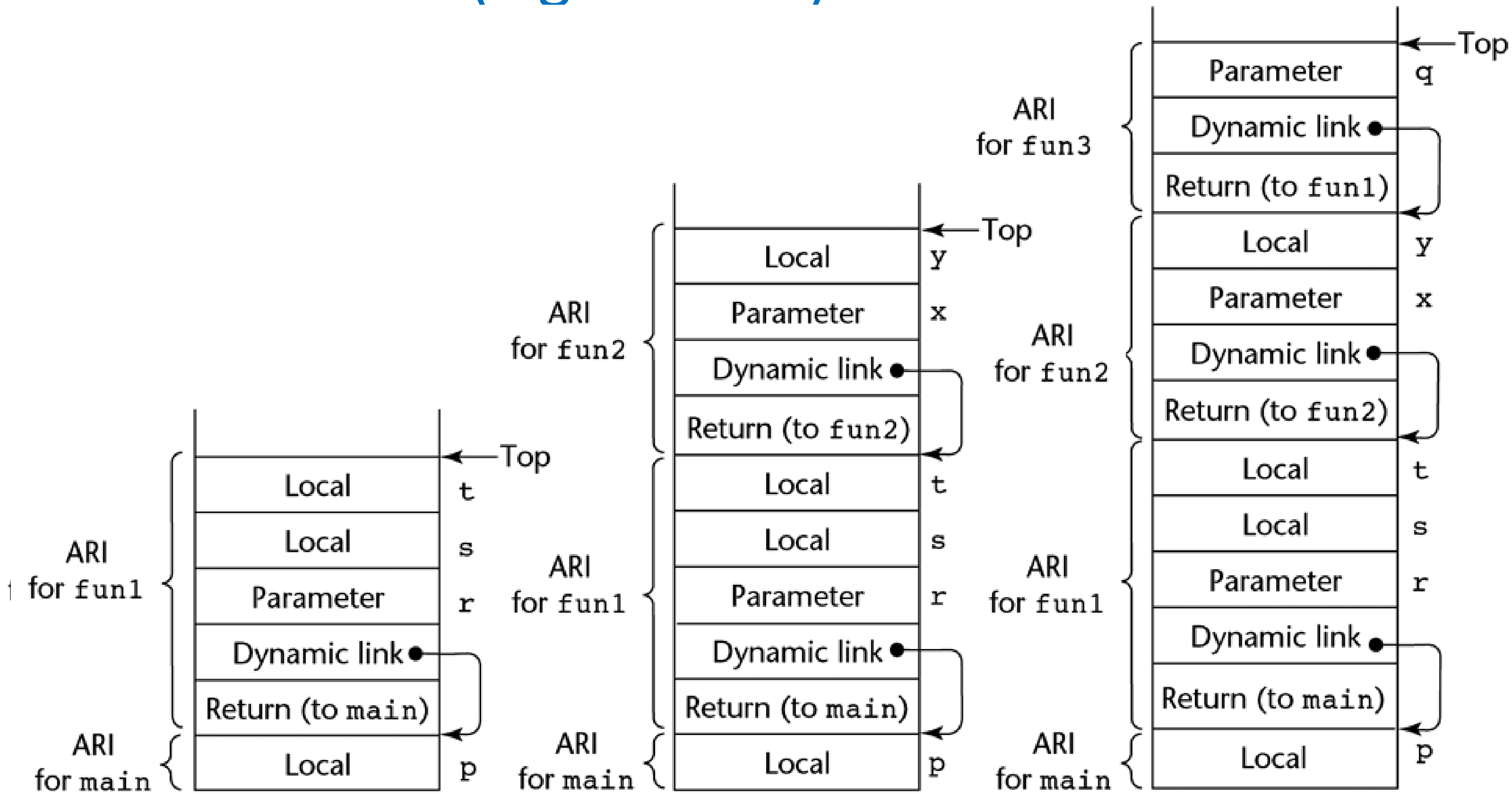
10.3 Implementing Subprograms with Stack-Dynamic Local Variables

- An example (10.3.2)

```
void fun1(float r)
{
   int s,t; fun2(s);
}
void fun2(int x)
{
   int y; fun3(y);
}
void fun3(int q) {}
void main()
{
   float p; fun1(p);
}
```

10.3 Implementing Subprograms with Stack-Dynamic Local Variables

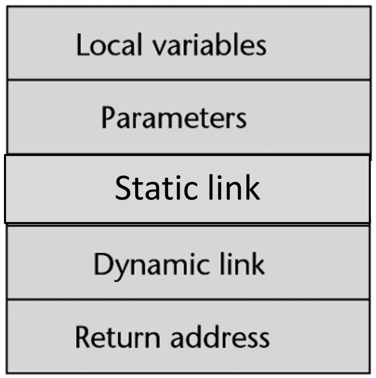
- Runtime stack (Figure 10.5)



10.3 Implementing Subprograms with Stack-Dynamic Local Variables

- Dynamic chain
  - A chain of dynamic pointers
  - The dynamic chain records the calling sequence.
- Accessing local variables
  - Local variables can be accessed by their offset from the beginning of the activation record. (base address+offset) This offset is called the local_offset.
  - The local_offset of a local variable can be determined by the compiler at compile time.
  - Variables declared inside block statements will be discussed later on.

10.4 Nested Subprograms

- Even more complex activation record
  - 

    Static link (pointer)

    1 point to the immediately enclosing block’s AR

    2 for accessing nonlocal variables

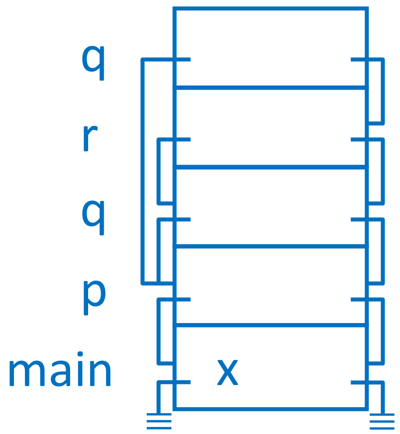
- Static chain
  - A chain of static pointers
  - A static chain records a blocking sequence.

10.4 Nested Subprograms

- Chain offset (static distance)
  - The chain offset of a variable or subprogram reference is static depth of referencing block

    – static depth of declaring block
  - It can be determined at compile time by the compiler.

main
var x
0 p
1 q
2 r
3 $x^{3(=3-0)}$ ... $q^{2(=3-1)}$
$x^{2(=2-0)}$ ... $r^{0(=2-2)}$
$q^{0(=1-1)}$
$p^{0(=0-0)}$



10.4 Nested Subprograms

- Lexical address

  The lexical address of a variable reference is a pair (*d*,*o*)

  where *d* = its chain offset (static distance)

  o = the variable's local offset within its AR

- Using static pointers

  To access a variable, the compiler

  1 computes the lexical address of the variable reference

  2 generates the following code for execution

  a) traverse *d* steps down the static chain to find the base address *b* of the AR containing the variable

  b) access the variable at address *b*+*o*

10.4 Nested Subprograms

- Maintaining static pointers (subprogram)

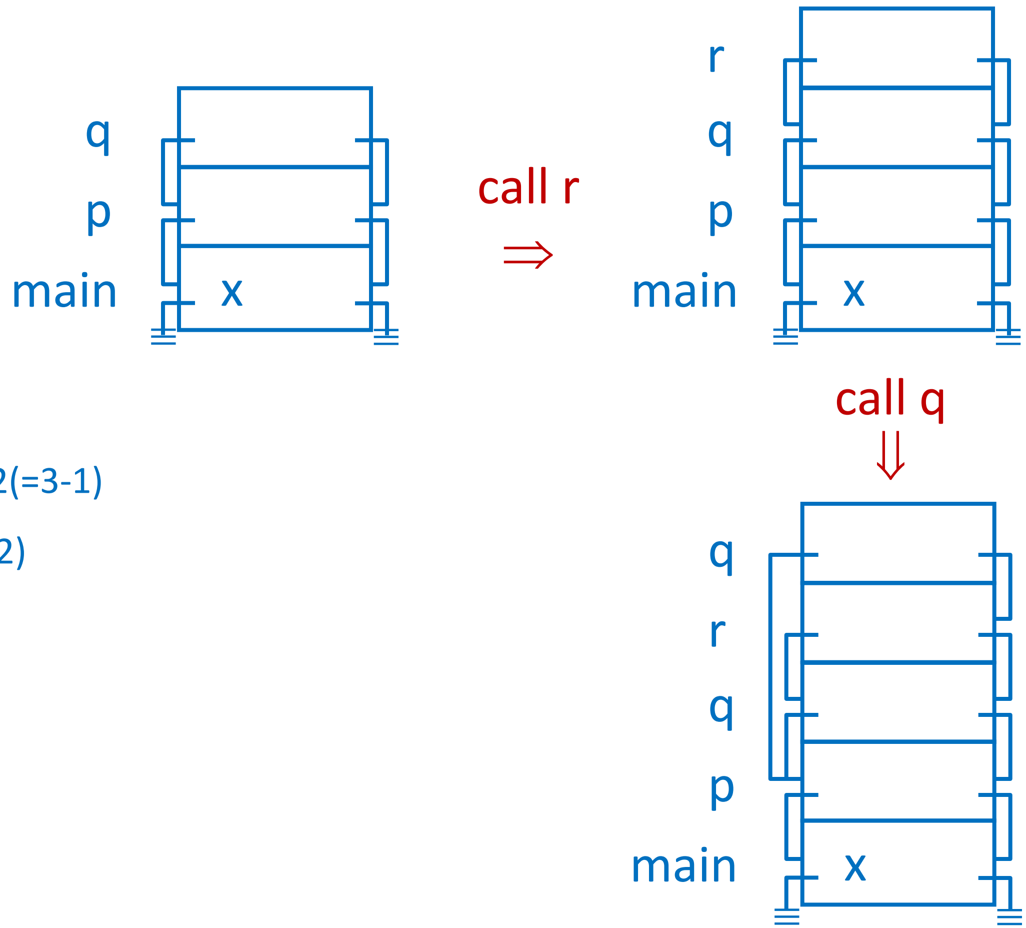
  To maintain the static pointers, the compiler

  1. computes the static distance *d* of the subprogram reference
  2. generates the following code for execution
     a) traverse *d* steps down the static chain to find the base address *b* of the immediately enclosing block's AR (i.e. the declaring block's AR)
     b) static pointer of the subprogram's AR ← *b*

10.4 Nested Subprograms

- Example

main
var x
0 p
1 q
2 r
3 $x^{3(=3-0)}$ … $q^{2(=3-1)}$
$x^{2(=2-0)}$ … $r^{0(=2-2)}$
$q^{0(=1-1)}$
$p^{0(=0-0)}$



10.4 Nested Subprograms

- Example (Figure 10.9)

Main_2

Bigsub

Sub1

Sub2

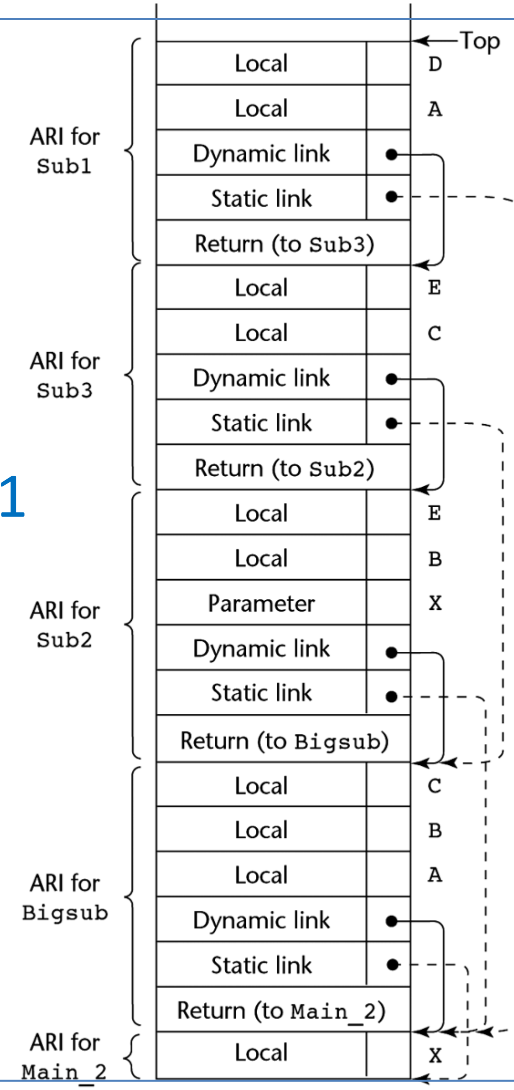
Sub3

Main_2 → Bigsub

→ Sub2 → Sub3 → Sub1

To which locations do sp and dp point are compiler-dependent.

(In the figure, sp points to AR's bottom, and dp points to AR's top.)



10.5 Blocks

- Maintaining static pointers (block statement)

  Approach 1

  - Block statements have ARs
  - E.g. Scheme, ML – trade space for assignment-free
  - Treat block statements as parameterless subprograms that are always called from the same location (static distance = 0)
  - To maintain the static pointers, the compiler simply generates the following code for execution

    static pointer of the block statement’s AR

    ← the base address of the immediately enclosing block’s AR found at stack top

10.5 Blocks

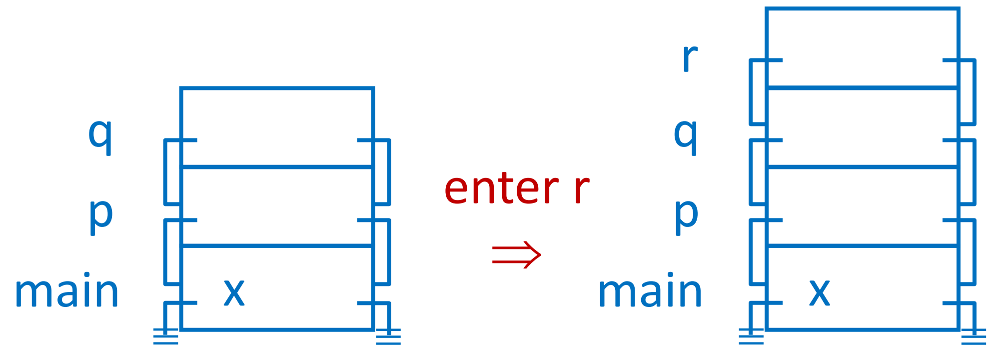
- Maintaining static pointers (block statement)
  Approach 2
  - Block statements have no ARs
  - E.g. C, C++, Java – due to assignment and iteration
  - The storage for variables declared in a block statement is allocated in the immediately enclosing subprogram's AR.
  - Pro: Save time
    - Save time to allocate and deallocate AR
    - Save time to access nonlocal variables, since the static distance of a nonlocal variable is shorten

10.5 Blocks

- Example

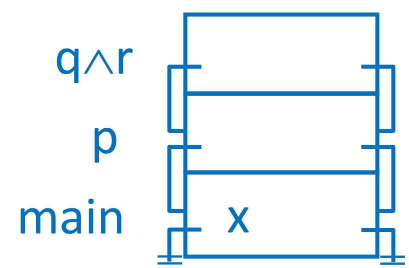
main
var x
p
0
1 q
2 block stmt r
$\cancel{3}$ $x^{3(=3-0)}$ … $q^{2(=3-1)}$
$x^{2(=2-0)}$ …
$q^{0(=1-1)}$
$p^{0(=0-0)}$

$x^{2(=2-0)}$ … $q^{1(=2-1)}$



Block statement having an AR

Block statement having no AR

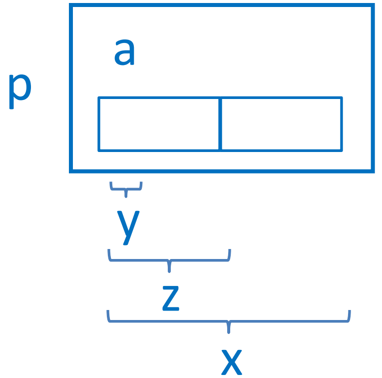


10.5 Blocks

Approach2

- Con: Waste storage
  - Storage is allocated even when the block statement is not being executed.
    (Worse still, it may not be executed at all.)
- However, it is often possible to optimize the storage usage.

```
void p(int a)
{
    while (...) { double x; ... }
    if (...) { char y; ... } else { int z; ... }
}
```



10.5 Blocks

- Final remark

  C/C++ don't have static pointers for two reasons:

  1 Functions can't be nested – nonlocal variables are all global

  2 Block statements have no ARs.

Supplementary: Tail-recursive optimization

- Non-tail-recursive function

  ```
  int f(int n) { if (n==0) return 1; else return n*f(n-1); }
  ```

  Since the recursive call isn't tail-recursive (i.e. the last thing to do), the value of n must be retained.
  Thus, on each recursive call, a new AR for f must be allocated

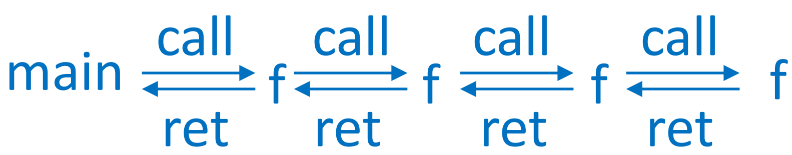
- Tail-recursive function

  ```
  int f(int n,int a) { if (n==0) return a; else return f(n-1,n*a); }
  ```

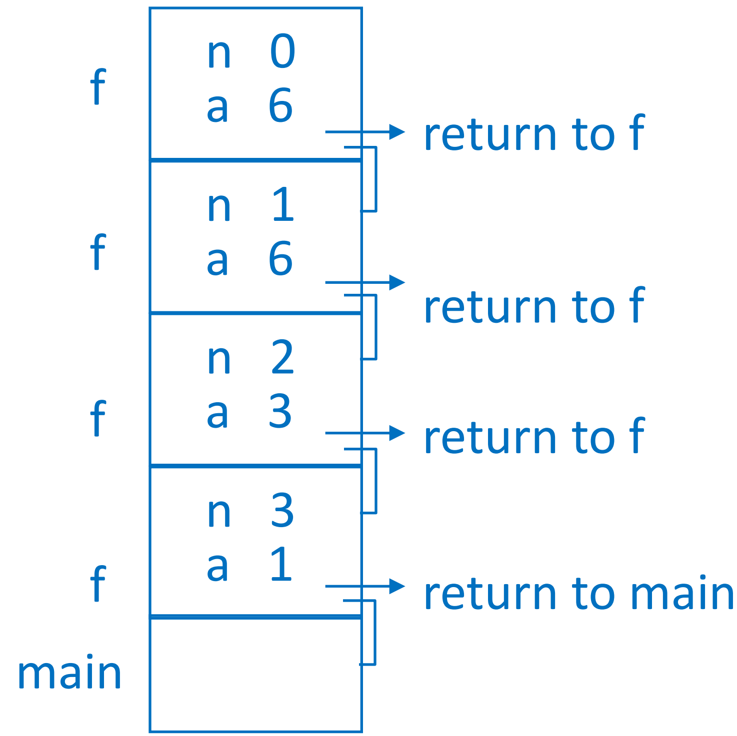
  Since the recursive call is tail-recursive, the values of n and a needn't be retained.
  Thus, a single AR for f suffices.

Supplementary: Tail-recursive optimization

- Without tail-recursive optimization



O(n) time and O(n) stack space



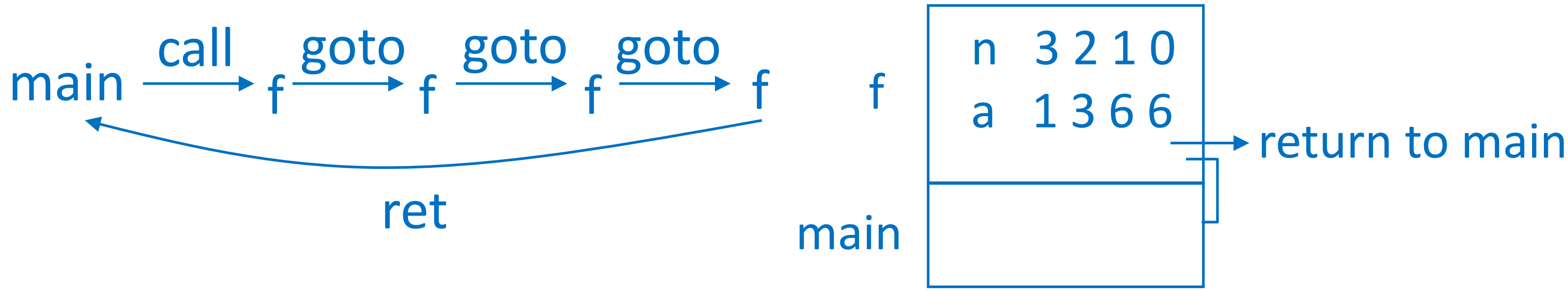
Supplementary: Tail-recursive optimization

- With tail-recursive optimization

```
int f(int n,int a)
{
entry: if (n==0) return a; else return f(n-1,n*a);
}
```

compiled to

```
n,a = n-1,n*a;
goto entry;
```



O(n) time and O(1) stack space

Supplementary: Tail-recursive optimization

- Tail-recursive optimization

  A tail-recursive call is executed in the following steps:

  1 Pass actual parameters to formal parameters by parallel assignments

  2 Goto the entry of the function being called

  The parallel assignment n,a = n-1,n*a means

  1 evaluate the expressions on the r.h.s. first (and store their values in temporary locations)

  2 assign their values in arbitrary order to the variables on the l.h.s

Supplementary: Tail-recursive optimization

- Check if the compiler optimizes tail-recursive calls
  - It suffices to test

    void p() { p(); }

    Without optimization, the run time stack will overflow.

    With optimization, it will run forever, as it is compiled to

    void p() { entry: goto entry; }
  - Scheme and ML compilers do tail-recursive optimization;

    C and C++ compilers usually don't.
- If the compiler optimizes tail-recursive calls, then
  1. Whenever possible, write a tail-recursive function

     Note: APS (Accumulator Passing Style) is useful.

Supplementary: Tail-recursive optimization

- If the compiler doesn't optimize tail-recursive call

  1 Same as above

  2 DIY (Do It Yourself)

  3 Make the code structured (i.e. without goto)

```
int f(int n,int a)
{
entry:
    if (n==0) return a; else { a=n*a; n=n-1; goto entry; }
}
```

```
while (n!=0) { a=n*a; n=n-1; }
return a;
```

Supplementary: Tail-recursive optimization

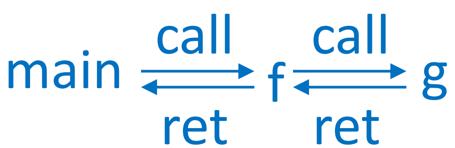
- Last-call optimization

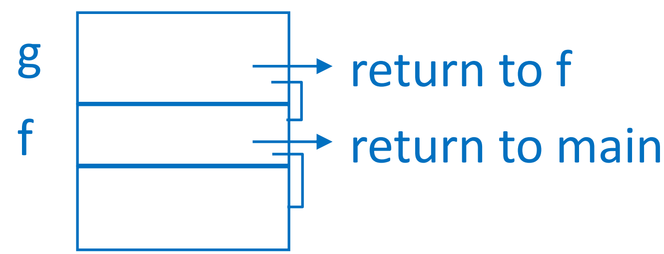
  A generalization of tail-recursive optimization.

```
void g() { … }
void f() { … g(); }
int main() { … f(); … }
```

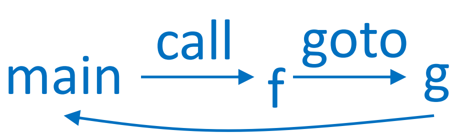
last call

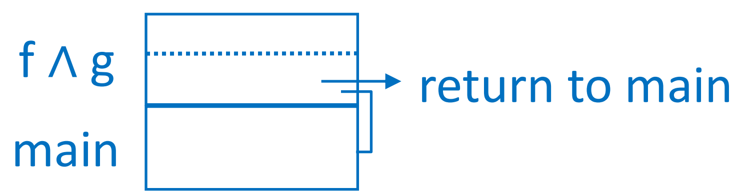
Without last-call optimization





With last-call optimization





10.6 Implementing Dynamic Scoping

- Deep access
  - Dynamic pointers have duel purposes
    1 for maintaining the run time stack
    2 for accessing the non-local variables
  - Advantage – Function entry and exit are easy
    Disadvantage – Take time to search the dynamic chain
  - The dynamic chain has to be searched.
    (cf. In static scoping, the static chain isn't searched.)
    - Names of variables must be saved in the run-time stack.
    - The length of the dynamic chain that must be searched can't be determined at compile time.

10.6 Implementing Dynamic Scoping

- Shallow access
  - Advantage – Fast access

    Disadvantage – Function entry and exit are expensive
  - Variable stack method
    - Have a separate stack for each variable name
    - On entry – push values of local variables onto their stacks
    - On exit – pop them off
    - Variable access

      1 look at the variable's stack top

      2 If the variable's stack is empty, error

10.6 Implementing Dynamic Scoping

- Shallow access
  - Variable stack method (Cont'd)

main() { int u,v; }
sub1() { int v,w; }
sub2() { int w,x; }
sub3() { int x,z; x = u+v; }

main () → sub1 → sub1 → sub2 → sub3

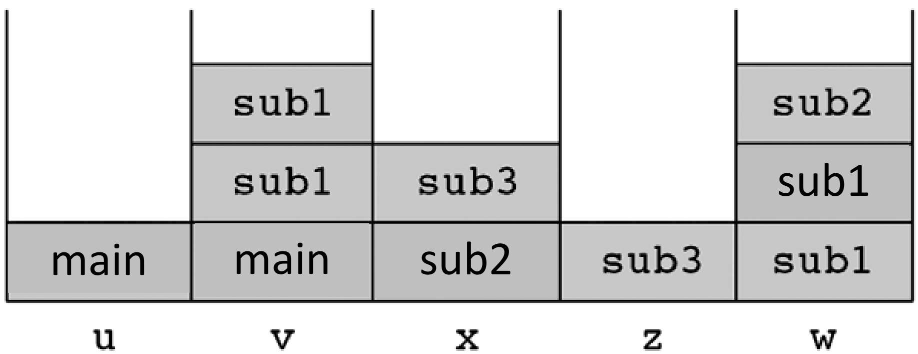
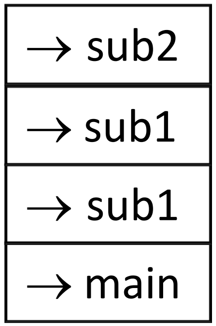


Figure 10.12



Run-time stack stores return addresses.

10.6 Implementing Dynamic Scoping

- Shallow access
  - Central table method
    - Have a central table with an entry for each variable name
    - On entry
      1. move values of active variables of the same name from table to stack
      2. store the values of local variables in the table
      3. mark the local variables active
    - On exit – undo the entry actions
    - Variable access
      1. look at the variable's entry in the table
      2. If it is inactive, error

10.6 Implementing Dynamic Scoping

- Shallow access
  - Central table method (Cont'd)

    main() → sub1

name	active	Value
u	1	main's
v	1	main's sub1's
w	0 1	sub1's
x	0	
z	0	

sub1 | main's v → main

run time stack