

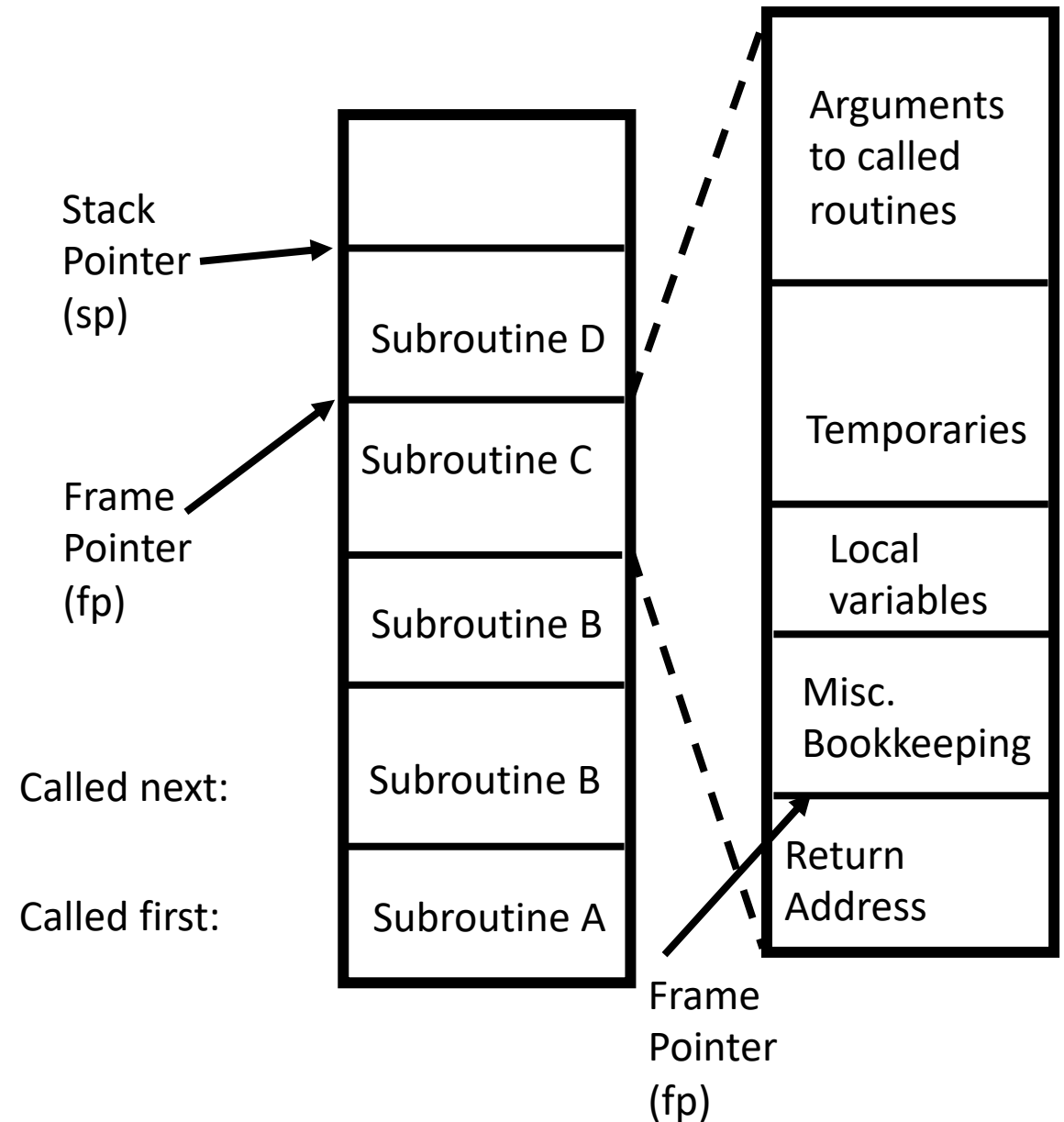
Subroutines

Overview

- How can subroutines access each other's environment (We already saw this, but some things will make more sense now)
- How to pass parameters:
 - Pass by reference
 - Pass by value
 - Pass by name
 - Others
- How are results returned

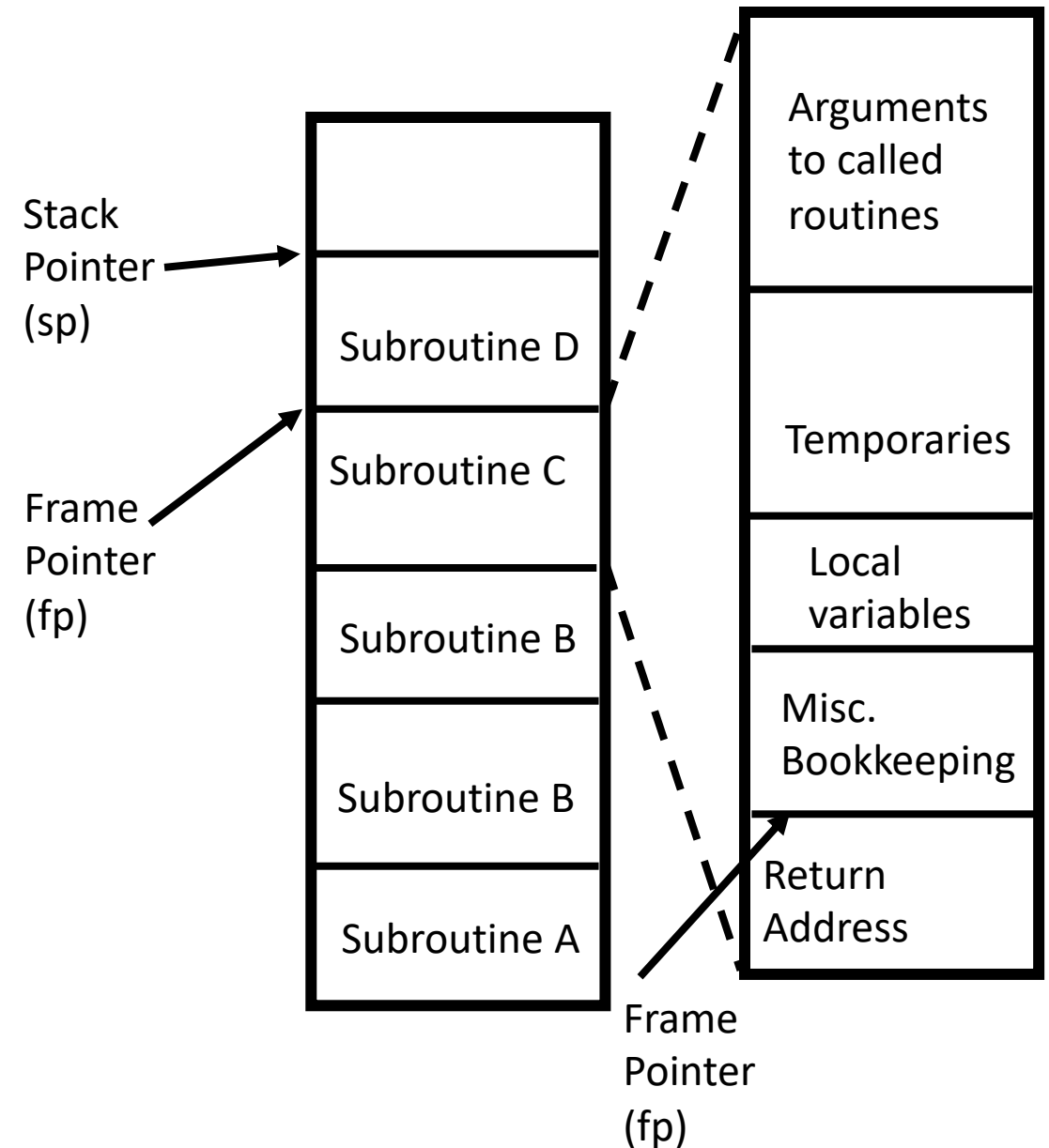
Stack-based Allocation

- Recursion complicates static allocation of variables
- Number of instances of a variable (e.g., a variable named “count” in a function “sum”) is, in theory, unbounded
- Natural nesting of functions allows to allocate memory on the stack
- Each instance (call) of a subroutine assigns memory from the stack for the various variables and constants used in the subroutine



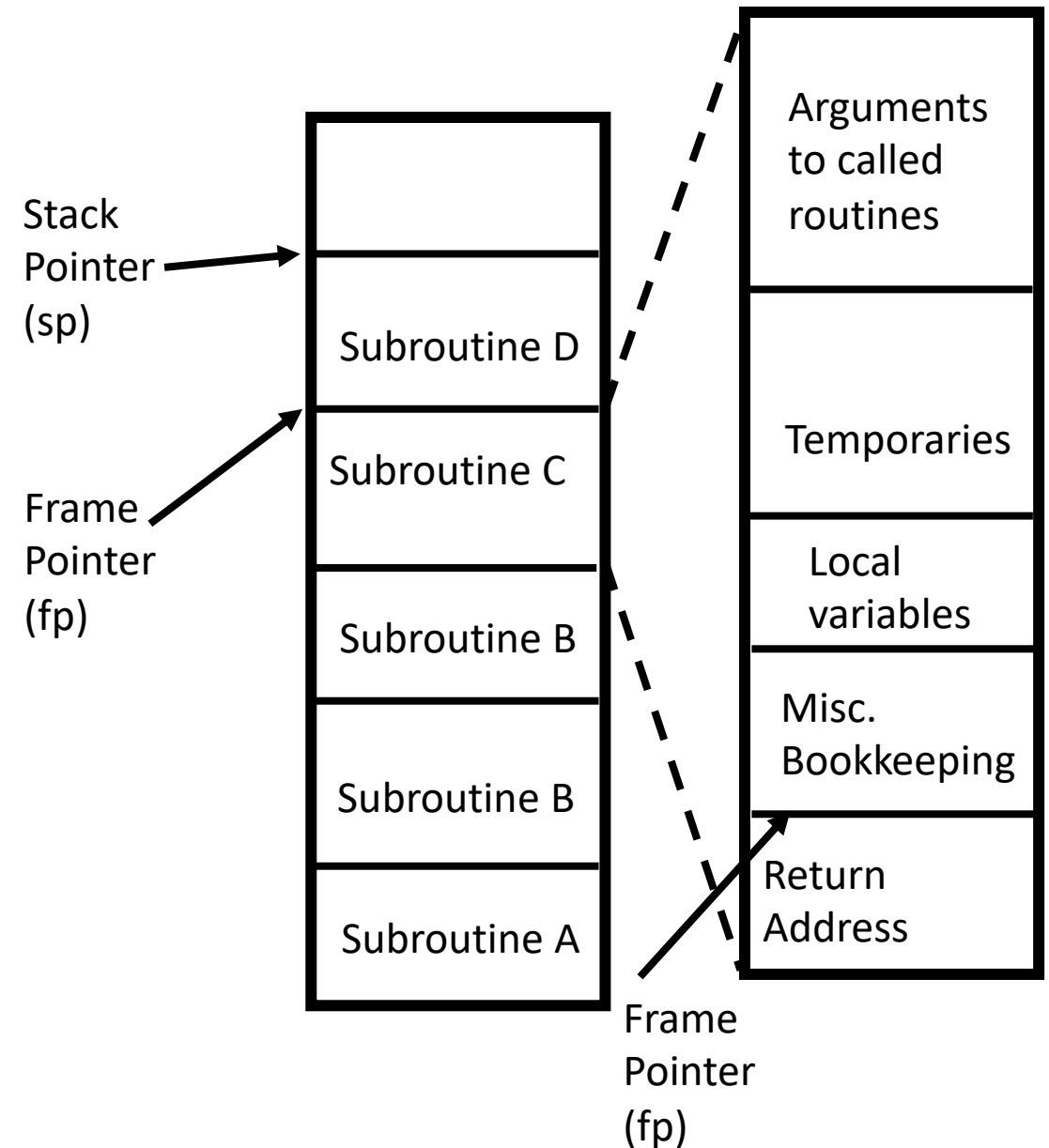
Stack-based Allocation

- Memory of subroutine allocated in a frame or activation record
- Frame also allocated memory for temporary variables (produced by compiler)
- Bookkeeping information includes: return address, reference to frame of caller (dynamic link), saved values of registers needed by caller and callee (e.g., the program counter)



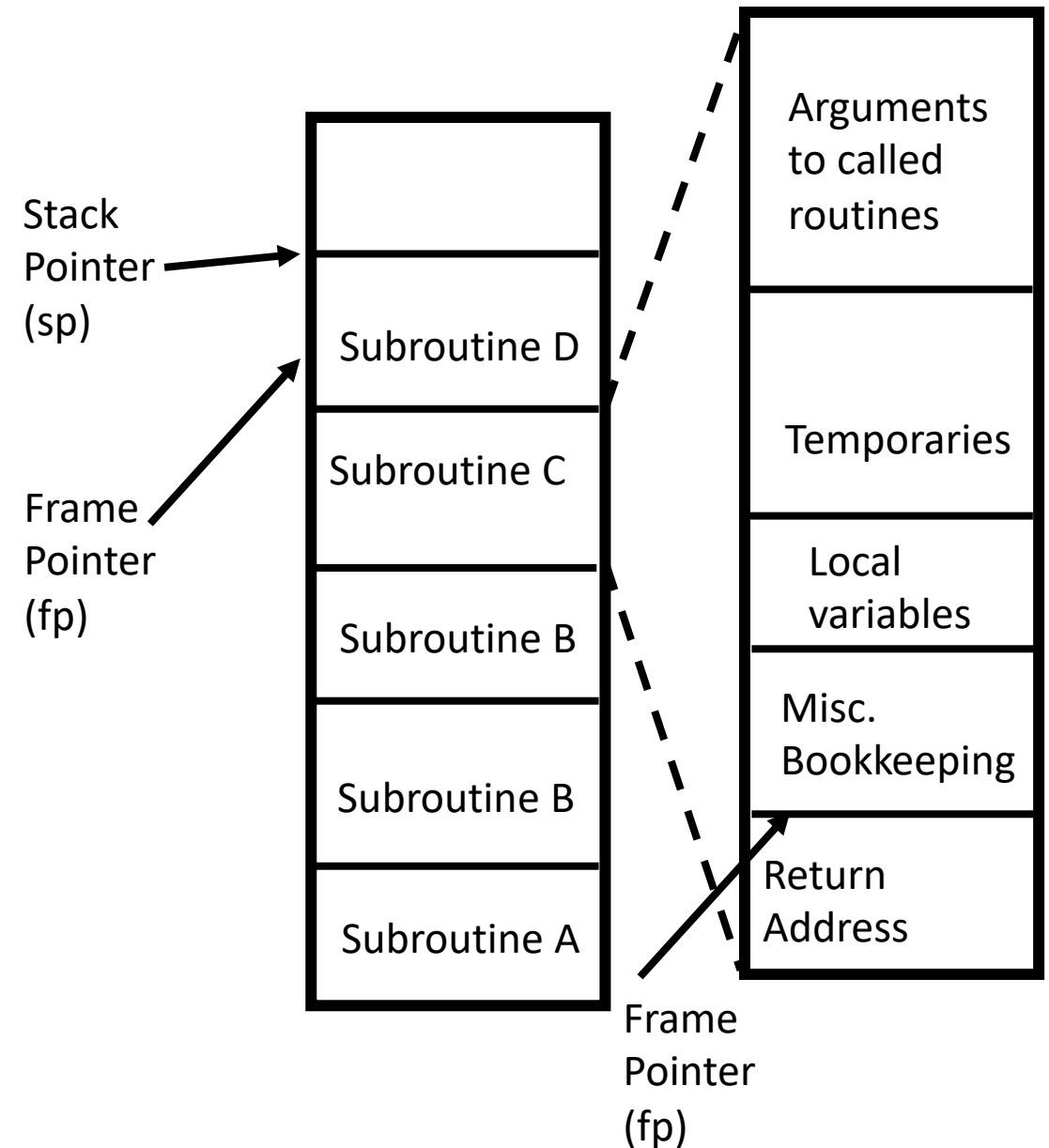
Stack-based Allocation

- Arguments passed to subroutines lie at the top of the frame (it's much more convenient)
- Subroutine actual arguments usually pushed into the stack
- Memory layout very implementation and language dependent
- Stack maintenance responsibility of the caller, before and after calling sequence
- Two parts: prologue and epilogue



Stack-based Allocation

- Location (actual address) of stack frame not determinable at compile-time, but offsets within frame are
- Frame pointer (fp) points to known location within the new activation frame; useful for external access such as copying results
- Other addresses accessed by known offsets w.r.t. fp
- FP, SP, PC usually correspond to machine registers



Typical Calling Sequence

Caller

- Save registers
- Compute values of arguments, move them into stack / registers
- (if language with nested subroutines) Compute static link and pass it as a hidden argument
- Use special instruction to jump to address start of subroutine (includes saving the return address in the stack or in a register)
- Moves returned values to wherever needed
- Restores pending registers

Callee

- Allocates a frame by deducting some constant offset from the sp
- Saves old frame pointer into the stack, update to the newly allocated frame
- Move return value (if any) to a register, or specific address given by the caller (possibly in the stack)
- Restores register values from the caller (inc. fp and sp)
- Jumps to return address

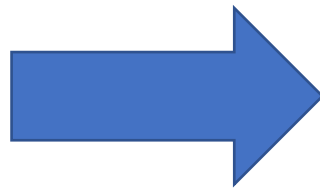
Prologue

Epilogue

In-Line Expansion

- Language implementations allow certain subroutines to be expanded [in-line](#) (function inlining) at point of call
- Essentially copy-paste called function into caller → replicates code

```
inline int func (int a, int b) {  
    return a+b;  
}  
int main () {  
    int x = 1;  
    int y = 2;  
    int z = func(x,y);  
}
```



```
inline int func (int a, int b) {  
    return a+b;  
}  
int main () {  
    int x = 1;  
    int y = 2;  
    int z = x+y;  
}
```


In-Line Expansion

- Language implementations allow certain subroutines to be expanded in-line (function inlining) at point of call
- Essentially copy-paste called function into caller → replicates code
- Advantages:
 - Avoids overheads: space allocation, branch overheads from call and return, maintenance of static chain and displays, and saving/restoring registers (potentially)
 - Gives a better view to compiler: improves potential for code optimization (principle: can do better knowing more), for instance in global register allocation, instruction scheduling and common subexpression elimination across function boundaries
 - Usually just a hint
- Disadvantages:
 - Potential increase for code size → function body replicated for every call site
 - Not viable for recursive functions

Parameter Passing

- First distinction: formal parameters vs actual parameters
 - Formal parameters: used in function definition
 - Actual parameters: what is passed during program execution (can be a temporary variable holding the value of an expression)
- Passing modes:
 - By value
 - By reference
 - By value/result
 - Read only
 - By name
 - Default and others
- Implementation-wise the main decision is: pass address of original entity or a copy to it? Copy implies extra work (the copying) and space

Parameter Passing: By Value

- Creates a fresh new copy of the actual variable being passed
- Copies all the contents of the actual parameter from the caller stack frame to the callee stack frame
- Some languages only provide this mechanism (e.g. C)
- That's why:
 - Passing a pointer as a parameter is how we can update variables passed as arguments to function calls in C
 - Modifying an actual parameter in C never changes the original value
 - Even when passing a pointer argument in C, we can change the value pointed to, but not the address stored in the pointer itself

```
void swap (int * a, int * b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Parameter Passing: By Reference

- The reference stores the address to the actual parameter
- Essentially just a new name to the variable being passed by reference
- Notice that this implies a mechanism for accessing non-local variables
- Most languages would deem illegal to pass an expression (e.g. $a+2*i$) as an actual parameter with by reference mode: only l-values (not r-values = right-side values)
- Original purpose: efficiency. In older computers, the overhead of Pass By Value could be significant (e.g not the same to pass a 4-byte variable as to pass a 10x10 matrix of an 8-byte datatype). Also program protection: disavow certain modifications to data

```
void swap (int & a, int & b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

Parameter Passing: By Value/Result


- Hybrid between passing By Value and By Reference
- Similarity with By Value: On call, the runtime copies the value of the actual parameter into the formal parameter; local changes don't affect (mostly) the actual parameter;
- Distinction with By Value and Similarity with By Reference: the final value / result of the formal parameter is copied back into the actual parameter (caller side). So the actual parameter is in fact changed.
- Distinction with by Reference: still passing values, no actual reference or address is being used; callee doesn't really know the real address of the actual parameter; it's the runtime who does the copy-in and copy-out work

Other Types of Parameter Passing

- [Pass by Name](#): boils down to textual replacement (as in C macro expansion); implies re-evaluation of passed parameter
- Named parameters (not to be confused with [pass by Name](#)): arguments are usually positional, this requires explicitly stating the name of a formal argument in the calling sequence:

```
polynomial(1,2,3);  
polynomial(x=1,y=2,z=3);  
polynomial(y=2,z=3,x=1);
```

- Default parameters: no need to actually pass them, they have a default value (e.g. parameters with initialization in C++)
- Also see variadic macros [here](#) and [here](#) and variadic functions [here too](#)



```
#define max(x,y)    ((x) >= (y)? (x) : (y))  
...  
z = max(a+b, a-b+10);  
  
replacing/expanding:  
  
z = ((a+b) >= (a-b+10) ? (a+b) : (a-b+10));  
  
// Use with “-E” compiler option with C compilers  
  
BTW, always parenthesize arguments in macros:  
#define mysum(x,y)  x + y  
w = mysum(3,4) * 5; // will produce 3 + 4 * 5
```

Parameters in Programming Languages

Parameter mode	Representative languages	Implementation mechanism	Permissible operations	Change to actual?	Alias?
value	C/C++, Pascal, Java/C# (value types)	value	read, write	no	no
in, const	Ada, C/C++, Modula-3	value or reference	read only	no	maybe
out	Ada	value or reference	write only	yes	maybe
value/result	Algol W	value	read, write	yes	no
var, ref	Fortran, Pascal, C++	reference	read, write	yes	yes
sharing	Lisp/Scheme, ML, Java/C# (reference types)	value or reference	read, write	yes	yes
r-value ref	C++11	reference	read, write	yes*	no*
in out	Ada, Swift	value or reference	read, write	yes	maybe
name	Algol 60, Simula	closure (thunk)	read, write	yes	yes
need	Haskell, R	closure (thunk) with memoization	read, write [†]	yes [†]	yes [†]

Figure 9.3 Parameter-passing modes. Column 1 indicates common names for modes. Column 2 indicates prominent languages that use the modes, or that introduced them. Column 3 indicates implementation via passing of values, references, or closures. Column 4 indicates whether the callee can read or write the formal parameter. Column 5 indicates whether changes to the formal parameter affect the actual parameter. Column 6 indicates whether changes to the formal or actual parameter, during the execution of the subroutine, may be visible through the other. *Behavior is undefined if the program attempts to use an r-value argument after the call. [†]Changes to arguments passed by need in R will happen only on the first use; changes in Haskell are not permitted.

Function Returns

- On subroutine return a compiler must:
 - Recover the previous state (machine registers)
 - Copy/move results depending on available passing mechanisms
 - Transfer back control (jump back to some address by changing the value of the program counter, PC register)
- Copying results can be responsibility of the caller or callee:
 - Caller can know where to copy the result from (assuming that nothing else executes in between the ending of the called subroutine and the next instruction)
 - Callee can know where to copy the result to (in the activation record of the caller) ➔ that's when access links, displays and dynamic links become useful.