Procedures and Functions

- Procedures have long history as essential programming tool.

- Low-level view: *subroutines* let us avoid duplicating frequently-used code.

- Higher-level view: *procedural abstraction* lets us divide programs into components with hidden internals.

- Procedural abstractions are *parameterized* over values and (sometimes) types.

- A *function* is just a procedure that returns a result (or, conversely, a procedure is just a function whose result we don’t care about).
Each invocation of procedure is specialized by associated activation data, such as:

- the actual values corresponding to the formal parameters of the procedure
- locations allocated for the values of local variables
- the return address in the caller

Activation data lives from the time procedure is called until the time it returns.

If one procedure calls another, directly or indirectly, their activation data must be kept separate, because lifetimes overlap.

In particular, each recursive invocation needs new activation data.
Activation Stacks

In most languages, activation data can be stored in frames, which are pushed and popped on the stack.
Calling conventions

- In compiled language implementations, we want to be able to generate the code for procedures separately from the code for their applications.
  - e.g. procedure may live in a pre-compiled library.
- Requires a calling convention between caller and callee.
  - e.g. caller places parameter values on the stack in a fixed order, and callee looks for them there.
- In an interpreter, where caller and callee are visible at the same time, it is easy to be imprecise about this, but we will try to build a careful model in the labs.
Procedure Parameter Passing

When we apply a function in an imperative language, the formal parameters get bound to locations containing values.

- How is this done and which locations are used?
- Do we pass addresses or contents of variables from the caller?
- How do we pass actual values that aren’t variables?
- What does it mean to pass a large value like an array?

Two main approaches: call-by-value (CBV) and call-by-reference (CBR).

Also call-by-name/need (CBN).
Call-by-value

- Each actual parameter is evaluated to a value before call.

- On entry to function, each formal parameter is bound to a freshly-allocated location, and the actual parameter value is copied into that location.

  - Much like processing declaration and initialization of a local variable.

- Semantics are just like assignment of actual expression to formal parameter.

- Simple; easy to understand!
Issues with call-by-value

- Updating a formal parameter doesn’t affect actuals in the caller.
- Usually a good thing!
- But sometimes not what we want

```c
void swap(int i, int j) {
    int t;
    t = i; i = j; j = t;
}
...
swap(a[p], a[q]);
```

Of course, perhaps this is usually a good thing!
More issues

- Can be inefficient for large unboxed values, e.g. C structs (records):

```c
typedef struct {double a1,a2,...,a10;}
    vector;

double dotp(vector v, vector w) {
    return v.a1 * w.a1 + v.a2 * w.a2 + ... 
        + v.a10 * w.a10;
}

vector v1,v2;
double d = dotp(v1,v2);
```

Call to `dotp` copies 20 doubles
Call-by-reference

- Pass a pointer to the existing location of each actual parameter.

- Within function, references to formal parameter are indirected through this pointer — so parameter can be dereferenced to get the value, but can also be updated.

- If actual argument doesn’t have a location (e.g. is an expression \((x+3)\)) then either:
  - evaluate it into a temporary location and pass address of temporary, or
  - treat as an error.
Issues with Call-by-reference

- Now procedures like `swap` work fine!
- Can also return values from procedure by assigning to parameters
- Lots of opportunity for aliasing problems, e.g.

```plaintext
PROCEDURE matmult(a,b,c: MATRIX)
  ...
  (* sets c := a * b *)

matmult(a,b,a) (* oops! *)
```

overwrites parts of argument as it computes result
Hybrid methods

In Pascal, Ada, and C++, programmer can specify (in the procedure header) for each parameter whether to use CBV or CBR.

C always uses CBV, but programmers can take the address of a variable explicitly, and pass that to obtain CBR-like behavior:

```c
swap(int *a, int *b) {
    int t;
    t = *a; *a = *b; *b = t;
} swap (&a[p],&a[q]);
```
Values can be References

In many modern languages, like Java or Python, both records (objects) and arrays are always boxed, so values of these types are already pointers (or references)

Thus, even if the language uses CBV, the values that are passed are actually references: calls don’t cause any actual copying of the large values

But it is a mistake (which some otherwise good authors make) to say that these languages use “call-by-reference” (If they did, they would be passing a reference to the reference!)
One simple way to give semantics to procedure calls is to say they behave “as if” the procedure body were textually substituted for the call, substituting actual parameters for formal ones.

This is very similar to macro-expansion, which really does this substitution (statically). E.g. (in C):

```c
#define swap(x,y) {int t; t = x; x = y; y = t;}
...
swap(a[p], a[q]);
```

expands to

```c
{int t; t = a[p]; a[p] = a[q]; a[q] = t;}
```
Avoiding capture

Blind substitution is dangerous!

```c
#define swap(x,y) {int t; t = x; x = y; y = t;}
```

```c
swap(a[t], a[q])
```

expands to

```c
{int t; t = a[t]; a[t] = a[q]; a[q] = t;}
```

Nonsense!

We say that \( t \) has been \textit{captured} by the declaration in the macro block.
Call-by-name (CBN)

One solution is to note that names of local variables are not important, e.g. we can rename to

```cpp
{int u; u = a[t]; a[t] = a[q]; a[q] = u;}
```

Call-by-name can be thought of as “substitution with renaming where necessary”

On real machines, CBN is implemented by passing to the function the AST for actual argument + values of its free variables

This makes CBN much less efficient to implement than CBV or CBR. (We may see more later.)
Call-by-need

A very useful feature of call-by-name is that arguments are evaluated only if needed.

\[
\text{foo } x \ y = \begin{cases} 
x & \text{if } x > 0 \\
\text{else } y 
\end{cases}
\]

\[
\text{foo } 1 \ \text{(factorial } 1000000) 
\]

- Avoids expensive computation.

As a further refinement, “pure” functional languages typically use **call-by-need** (or **lazy**) evaluation, in which arguments are evaluated at most once:

\[
\text{foo } x \ y = \begin{cases} 
\text{if } x > 0 & \text{then } x \\
\text{else } y \ast y 
\end{cases}
\]

\[
\text{foo } (-1) \ \text{(factorial } 1000000) 
\]

- Avoids expensive recomputation.