

CS558 Programming Languages

Winter 2008

Lecture 7

PROCEDURES AND FUNCTIONS

Procedures have a long history as an essential tool in programming:

- Low-level view: subroutines give a way to avoid duplicating frequently used code
- Higher-level view: Procedural abstraction gives a way to divide large programs into smaller components with hidden internals

We can imagine abstracting over many aspects of a piece of code. Mainstream languages chiefly support abstraction over values and (sometimes) types.

PROCEDURE ACTIVATION DATA

Each invocation of a procedure requires associated data, such as:

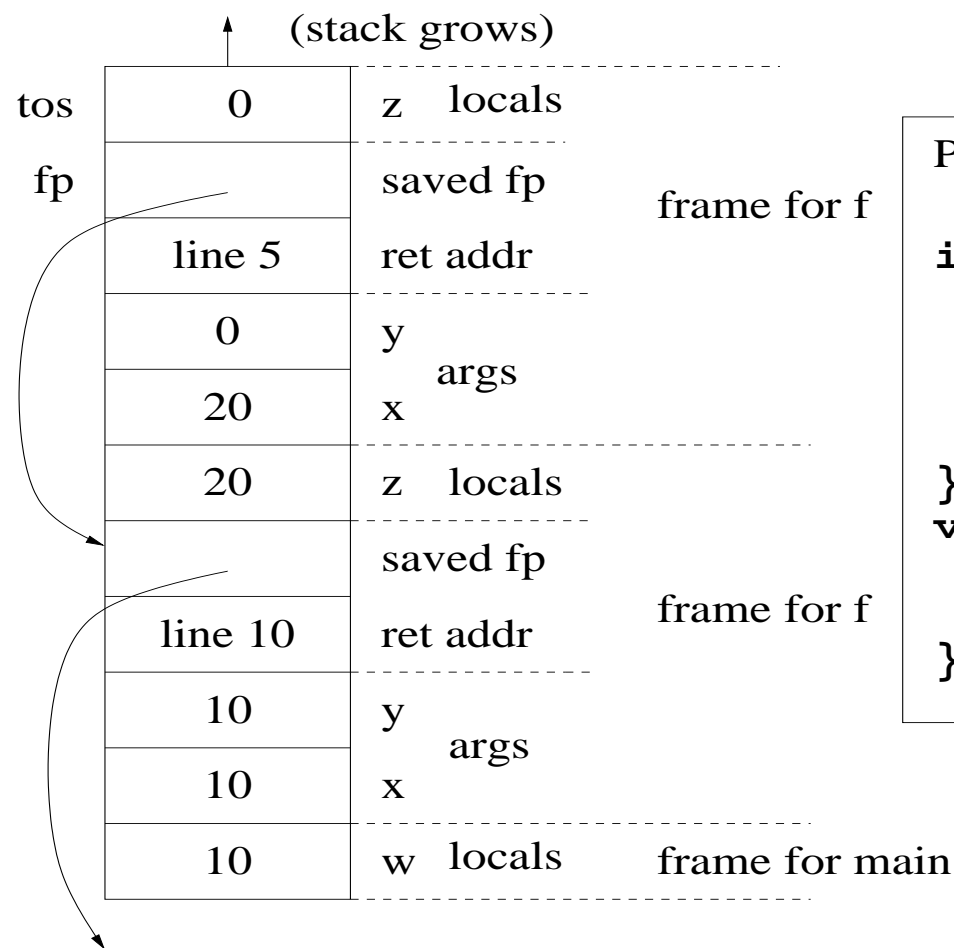
- the **return address** of the caller
- the **actual** values corresponding to the **formal** parameters of the procedure
- space for the values of **local variables** associated with the procedure.

This activation data must live from the time the procedure is invoked until the time it returns. If one procedure calls another procedure, their activation data must be kept separate, because their lifetimes overlap. In particular, the data for all invocations of a recursive procedure must be kept separate.

ACTIVATION STACKS

In most languages, activation data can be stored on a **stack**, and we speak of pushing and popping activation **frames** from the stack, which is a very efficient way of managing local data.

A typical activation stack, shown just before inner call to `f` returns.



Program:

```
int f(int x, int y){
    int z = y+y;
    if (z > 0)
        z = f(z,0);
    return z+y;
}
void main() {
    int w = 10;
    w = f(w,w);
}
```

WHAT ABOUT REGISTERS?

Although it is convenient to view all locations as memory addresses, most machines also have **registers**, which are:

- much faster to access,
- but very limited in number (e.g., 4 to 64).

So compilers try to keep variables (and pass parameters) in registers when possible, but always need memory as a backup. Using registers is fundamentally just an (important!) optimization.

Easy to have environment map each name to location that is **either** memory address or register.

- But registers don't have addresses, so they can't be accessed indirectly, and register locations can't be passed around or stored.

PROCEDURE PARAMETER PASSING

When we activate a procedure, 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
- call-by-reference

CALL-BY-VALUE

- Each actual argument is **evaluated** to a **value** before call.
- On entry, value is **bound** to formal parameter just like a local variable.
- Updating formal parameter doesn't affect actuals in **calling** procedure.

```
double hyp(double a, double b) {  
    a = a * a;  
    b = b * b;  
    return sqrt(a+b);  
}
```

- Simple; easy to understand!
- Implement by binding the formal parameters to freshly-allocated locations, and **copying** the actual values into these locations (just like **assignment**).

PROBLEMS WITH CALL-BY-VALUE (1)

- Can be inefficient for large unboxed values:

Example (C): Calls to dotp copy 20 doubles

```
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);
```

PROBLEMS WITH CALL-BY-VALUE (2)

- Cannot affect calling environment directly. (Of course, perhaps this is a **good** thing!)

Example: calls to `swap` have no effect:

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

- Can at best **return** only one result (as a value), though this might be a record.

CALL-BY-REFERENCE

- Pass the existing **location** of each actual parameter.
- On entry, the formal parameter is bound to this location, which must be dereferenced to get value, but can also be **updated**.
- If actual argument doesn't have a location (e.g., "2 + 3"), either:
 - Evaluate it into a temporary location and pass address of temporary, or
 - Treat as an error.
- Now swap, etc., work fine!
- Accesses are slower.
- Lots of opportunity for **aliasing** problems, e.g.,

```
PROCEDURE matmult(a,b,c: MATRIX)
```

```
... (* sets c := a * b *)
```

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

- **Call-by-value-result** (a.k.a. **copy-restore**) addresses this problem, but has other drawbacks.

HYBRID METHODS; RECORDS AND ARRAYS

How might we combine the simplicity of call-by-value with the efficiency of call-by-reference, especially for large unboxed values?

- In Pascal, Ada, and similar languages, where records and arrays are both unboxed, the programmer can specify (in the procedure header) for each parameter whether to use call-by-value or call-by-reference.
- In ANSI C/C++, record (`struct` or `class`) values are unboxed, but arrays are boxed. C always uses call-by-value, but programmers can take the address of a variable explicitly, and pass that to obtain cbr-like behavior:

```
swap(int *a, int *b) {  
    int t;  
    t = *a; *a = *b; *b = t; }  
swap (&a[p], &a[q]);
```

Of course, it is the programmer's responsibility to make sure that the address remains valid (especially when it is **returned** from a function).

COMPLEX AND SIMPLE SOLUTIONS

- C++ supports both cbr parameters and explicit pointers:

```
swap(int &a, int *b) {  
    int t;  
    t = a; a = *b; *b = t;  
}  
...  
swap(a[p], &a[q]);
```

Mixing explicit and implicit pointers can be **very** confusing!

- In Java and ML, values of both records (objects) and arrays are boxed. These languages have only call-by-value, but this doesn't actually cause copying, even for record or array values.
- Approach is made more feasible because programmer doesn't have to worry about lifetime of heap data, due to automatic garbage collection.
- Clever compilers can decide whether smallish objects should be heap-allocated or kept unboxed, while continuing to give the **semantic** effect of the boxed representation.

SUBSTITUTION

One simple way to give semantics to procedure calls is to say they should behave as if the procedure body was **textually substituted** for the call, with the actual parameters substituted for the formal ones.

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

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

- It even makes sense for recursive procedures (though of course it cannot be **implemented** by static substitution in this case).
- **BUT** blind substitution is dangerous because of possible “**variable capture**,” e.g.,

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

expands to

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

CALL-BY-NAME

- Here t is “captured” by the declaration in the macro, and is undefined at its first use.

- Note that name of local variable is not important: it could be renamed:

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

- **Call-by-name** (first proposed in Algol60) can be thought of as “substitution with renaming where necessary.”
- In practice, call-by-name is implemented by binding any free variables in arguments at the point of call (rather than the point of use).
- This makes CBN much less efficient to implement than CBV or CBR.

JENSEN'S DEVICE

- Call-by-name is flexible, but potentially very confusing in the presence of side-effects.

```
real procedure SIGMA(x, i, n);
  real x; integer i, n;
begin
  real s;
  s := 0;
  for i := 1 step 1 until n do
    s := s + x;
  SIGMA := s;          (sets return value)
end
```

```
SIGMA(a(j), j, 10);      (computes  $\sum_{j=1}^{10} a_j$ )
SIGMA(a(k)*b(k), k, 10); (computes  $\sum_{k=1}^{10} a_k b_k$ )
```

CALL-BY-NEED

- If language has no mutable variables (as in “pure” functional languages), call-by-name gives a substitution gives a beautifully simple semantics for procedure calls.
- Arguments are evaluated only if needed.

```
foo x y = if x > 0 then x else y
```

```
foo 1 (factorial 1000000)
```

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

```
foo x y = if x > 0 then x else y * y
```

```
foo (-1) (factorial 1000000)
```