CS558 Programming Languages

Winter 2013

Lecture 4

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.

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.)

Procedure Activation Data

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

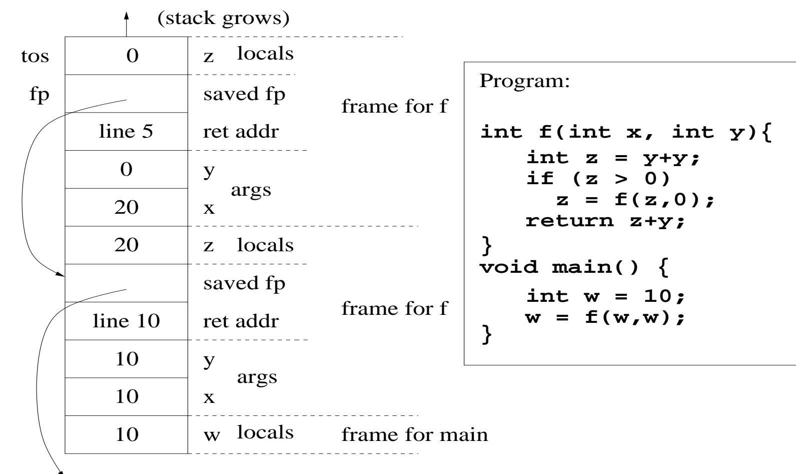
- the **return address** of the caller
- the **actual** values corresponding to the **formal** parameters of the procedure
- space for the values of **local variables** associated with the procedure.

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.



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 (CBV)
- call-by-reference (CBR)

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 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., `(x + 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 like this can be **very** confusing!

• In Python and OCaml, 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 subsitution (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 (CBN)

- 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 powerful...

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

- ...but potentially very confusing in the presence of side-effects.

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)
```

Iteration vs. Recursion

Any iteration can be written as a recursion.

For example:

while (t) do e

is equivalent to

```
void f(bool b) {
  if (b) then {
    e;
    f(t)
  }
}
f(t)
```

where we assume that the variables used by e and t are global.

When can we do the converse? It turns out that a recursion can be rewritten as an iteration (without needing any extra storage) whenever all the recursive calls are in **tail position**. To be in tail position, the call must be the **last** thing performed by the caller before it itself returns.

OCaml Tail-call Examples

List operations can often be made tail-recursive in this way:

```
let rec last xs = (* tail-recursive *)
  match xs with
  | [] -> 0
  | [x] -> x
  | x::xs' -> last xs'

let rec length xs = (* not tail-recursive *)
  match xs with
  | [] -> 0
  | x::xs -> 1 + (length xs)

let length l = (* use accumulating parameter; now is tail-recursive *)
  let rec f xs len =
    match xs with
    | [] -> len
    | x::xs' -> f xs' (len+1) in
  f l 0
```

A decent compiler can turn tail-calls into iterations, thus saving the cost of pusing an activation frame on the stack. This is essential for languages (like ML) that lack iteration, and useful even for those that have it (like C).

Systematic Removal of Recursion

(Adapted from Sedgewick, *Algorithms*, 2nd ed.. Examples in C.)

But what about general (non-tail) recursion? One way to get a better appreciation for how recursion is implemented is to see what is required to get rid of it.

Original program:

```
typedef struct tree *Tree;
struct tree {
  int value;
  Tree left, right;
};

void printtree(Tree t) {
  if (t) {
    printf("%d\n",t->value);
    printtree(t->left);
    printtree(t->right);
  }
}
```


Step 1

Remove **tail-recursion**.

```
void printtree(Tree t) {
top:
  if (t) {
    printf("%d\n",t->value);
    printtree(t-left);
    t = t->right;
    goto top;
  }
}
```

STEP 2

Use explicit stack to replace non-tail recursion. Simulate behavior of compiler by pushing local variables and return address onto the stack **before** call and popping them back off the stack **after** call.

Assume this stack interface:

```
Stack empty;
void push(Stack s,void* t);
(void*) pop(Stack s);
int isEmpty(Stack s);
```

 © 1994–2013 ANDREW TOLMACH

STEP 2 (CONT.)

Here there is only one local variable (`t`) and the return address is always the same, so there's no need to save it.

```
void printtree(Tree t) {
  Stack s = empty;
top:
  if (t) {
    printf("%d\n",t->value);
    push(s,t);
    t = t->left;
    goto top;
retaddr:
    t = t->right;
    goto top;
  }
  if (!(isEmpty(s))) {
    t = pop(s);
    goto retaddr;
  }
}
```

 © 1994–2013 ANDREW TOLMACH

STEP 3

Simplify by:

- Rearranging to avoid the `retaddr` label.
- Pushing right child instead of parent on stack.
- Replacing first `goto` with a `while` loop.

```
void printtree(Tree t) {
  Stack s = empty;
top:
  while (t) {
    printf("%d\n",t->value);
    push(s,t->right);
    t = t->left;
  }
  if (!(isEmpty(s))) {
    t = pop(s);
    goto top;
  }
}
```

 © 1994–2013 ANDREW TOLMACH

STEP 4

Rearrange some more to replace outer `goto` with another `while` loop.

(This is slightly wasteful, since an extra `push`, `stackempty` check and `pop` are performed on root node.)

```
void printtree(Tree t) {
  Stack s = empty;
  push(s,t);
  while(!(isEmpty(s))) {
    t = pop(s);
    while (t) {
      printf("%d\n",t->value);
      push(s,t->right);
      t = t->left;
    }
  }
}
```

 © 1994–2013 ANDREW TOLMACH

Step 5

A more symmetric version can be obtained by pushing and popping the left children too.

Compare this to the original recursive program.

```
void printtree(Tree t) {
  Stack s = empty;
  push(s,t);
  while(!(isEmpty(s))) {
    t = pop(s);
    if (t) {
      printf("%d\n",t->value);
      push(s,t->right);
      push(s,t->left);
    }
  }
}
```


Step 6

We can also test for empty subtrees **before** we push them on the stack rather than after popping them.

```
void printtree(Tree t) {
  Stack s = empty;
  if (t) {
    push(s,t);
    while(!(isEmpty(s))) {
      t = pop(s);
      printf("%d\n",t->value);
      if (t->right)
        push(s,t->right);
      if (t->left)
        push(s,t->left);
    }
  }
}
```


Exceptions

Programs often need to handle **exceptional** conditions, i.e., deviations from “normal” control flow.

Exceptions may arise from

- failure of built-in or library operations (e.g., division by zero, end of file)
- user-defined events (e.g., key not found in dictionary)

It can be awkward or impossible to deal with these conditions explicitly without distorting normal code.

Most recent languages (Ada, C++, Java, Python, OCaml, etc.) provide a means to **define**, **raise** (or **throw**), and **handle** exceptions.

Example: Exceptions in Python

```
class Help(Exception):  # define a new exception
    pass

try:
    ...
    if gone wrong:
        raise Help()    # raise user-defined exception
    ...
    x = a / b  # might raise a built-in exception
    ...
except Help:
   ...report problem...
except ZeroDivisionError:
   x = -99
```

What to do in an exception?

If there is a statically enclosing handler, the thrown exception behaves much like a `goto`. In previous example:

```
...
if (gone wrong) goto help_label;
...
help_label: ...report problem...
```

But what if there is no handler explicitly wrapped around the exception-throwing point?

- In most languages, uncaught exceptions **propagate** to next **dynamically** enclosing handler. E.g, caller can handle uncaught exceptions raised in callee.
- A few languages support **resumption** of the program at the point where the exception was raised.
- Many languages permit a value to be returned along with the exception itself.

Exception Handling Example

```
class BadThing(Exception):
    def __init__(self, problem):
        self.problem = problem

def foo():
    ... raise BadThing("my problem") ...

def bar():
    try:
        x = foo()
    except BadThing as exn:
        print ("oops:" + exn.problem)
```


Exceptions vs. Error Values

An alternative to user-raised exceptions is to return status values, which must be checked on return:

```
let rec find (k0:string) (env: (string * int) list)
                                            : int option =
   match env with
   | [] -> None
   | (k,v)::t ->
       if k = k0 then
         Some v
       else
         find k0 t

... match find "abc" e0 with
    | Some v -> ... v ...
    | None -> ...perform error recovery...
```


Exception vs. Error Values (2)

With exceptions, we can defer checking for (rare) error conditions to a more convenient point.

```
exception NotFound
let rec find (k0:string) (env: (string * int) list) : int =
   match env with
   | [] -> raise NotFound
   | (k,v)::t ->
       if k = k0 then
         v
       else
         find k0 t

... try
       let v = find "abc" e0 q
       in ... v ...
    with NotFound ->
       ...perform error recovery...
```

Implementing Exceptions (1)

One approach to implementing exceptions is for the runtime system to maintain a **handler stack** with an entry for each handler context currently active.

- Each entry contains a handler code address and a call stack pointer.

- When the scope of a handler is entered (e.g. by evaluating a `try...with` expression), the handler's address is paired with the current call stack pointer and pushed onto the handler stack.

- When an exception occurs, the top of the handler stack is popped, resetting the call stack pointer and passing control to the handler's code. If this handler itself raises an exception, control passes to the next handler on the stack, etc.

- Selective handlers work by simply re-raise any exception they don't want to handle (causing control to pass to the next handler on the stack).

Exceptions on Purpose

- In this execution model, raising an exception provides a way to return quickly from a deep recursion, with no need to pop stack frames one at a time.

Example:

```
exception Zero
let product l =
  let rec prod l =
    match l with
    | [] -> 1
    | h::t ->
       if h = 0 then
         raise Zero
       else
         h * (prod t) in
  try
    prod l
  with Zero -> 0
```


Implementing Exceptions (2)

The handler-stack implementation makes handling very cheap, but incurs cost each time we enter a new handler scope. If exceptions are very rare, this is a bad tradeoff.

- As an alternative, some runtime systems use a **static table** that maps each code address to the address of the statically enclosing handler (if any).

- If an exception occurs, the table is inspected to find the appropriate handler.

- If there is no handler defined in the current routine, the runtime system looks for a handler that covers the return address (in the caller), and so on up the call-stack.

- The deliberate use of exceptions in the previous example would probably be unwise if this implementation approach is used.