CS322 Languages and Compiler Design II

Spring 2012

Lecture 8

RUNTIME ENVIRONMENTS

- Data Representation (mostly covered in CS321)
- Storage Organization
- Procedures (& Stacks)

- Activation Records

- Access to non-local names

- Procedures as first-class values

- Object-oriented dispatch

- Parameter Passing

- Storage Allocation

- Static/Stack/Heap

- Garbage Collection

Typical Storage Organization (Unix)

- Subdivide machine address space by function, access, allocation.

```
∞  ---------------
   |             |
   |   STACK     |
   |             |
   ---------------   ‘‘top’’ of stack
   |      |      |      (but stacks nearly always
   |      |      |           grow DOWN!)
   |      ↓      |
   |             |   (in VM systems, these pages
   |             |      don’t actually exist)
   |      ↑      |
   |      |      |
   |      |      |
   ---------------   (controlled by  sbrk system call)
   |             |
   |    HEAP     |   Managed by allocator/collector.
   |             |
   ---------------------
   |             |   Uninitialized (‘‘b[lock] s[tatic] s[torage]’’)
   | STATIC DATA |------
   |             |   Initialized (in object file)
   ---------------------
   |             |
   |   CODE      |   Read-only (in object-file)
   |             |
0  ---------------
```

Storage Classes

Static Data : Permanent Lifetimes

• Global variables and constants.

• Allows fixed address to be compiled/linked into code.

• No runtime management costs.

• Original FORTRAN (no recursion) used static activation records.

Stack Data : Nested Lifetimes

• Allocation/deallocation is cheap (just adjust stack pointer).

• Most architectures support cheap `sp`-based addressing.

• Good **locality** for VM systems, caches.

• Algol, C family use stack for activation records.

Heap Data : Arbitrary Lifetimes

• Requires explicit allocation and (dangerous) explicit deallocation or garbage collection.

• Languages (e.g. Lisp, fab) with first-class functions need heap for some activation-record data, which have non-nested lifetimes.

PROCEDURES AND ACTIVATIONS

• A procedure **definition** associates a **name** with a procedure `body` and associated **formal parameters**.

• A procedure **activation** is created during execution when the procedure is called (with **actual parameters**).

• Activations have **lifetimes**: the time between execution of the first and last statements in the procedure.

• Activations are either **nested** (e.g., a,b) or **non-overlapping** (e.g., b,c):

```
a() {                          a
  b();                        / \
  c();                       /   \
}                           b     c
                                  |
c() {                             |
  d();                            d
}
```

• Procedure `f` is **recursive** if two or more activations of `f` are **nested**. (Note that `f` need not call itself directly.)

Activation Trees - Example

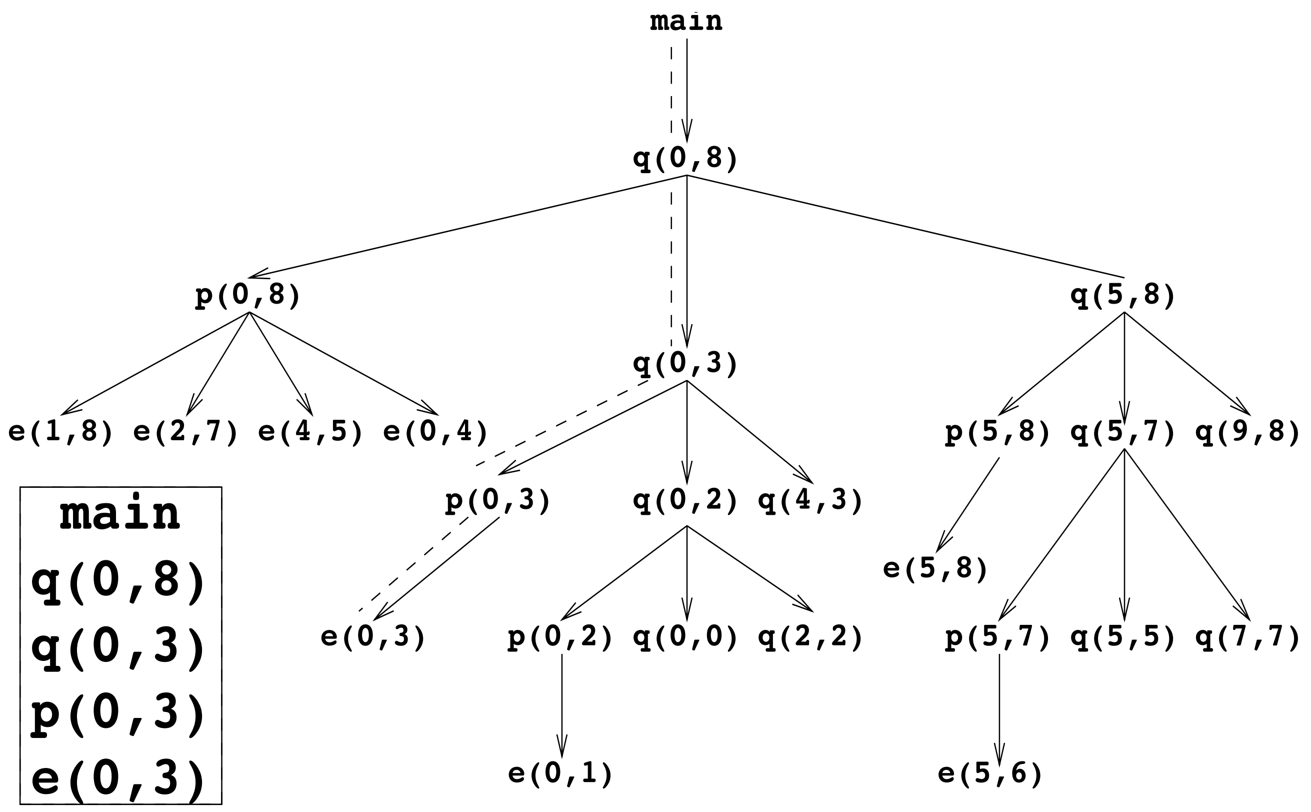
```
int a[9] = {10,32,567,-1,789,3,18,0,-51};

main() {
  quicksort(0,8);
}

void exchange(int i, int j) {
  int x = a[i]; a[i] = a[j]; a[j] = x;
}

int partition(int y, int z) {
  int i = y, j = z + 1;
  while (i < j) {
    i++; while (a[i] < a[y]) i++;
    j--; while (a[j] > a[y]) j--;
    if (i < j) exchange(i,j);
  };
  exchange(y,j);
  return j;
}

void quicksort(int m, int n) {
  if (n > m) {
    int i = partition(m,n);
    quicksort(m,i-1);
    quicksort(i+1,n);
  };
}
```

Activation Trees - Example (Cont.)

(also known as '**call tree**")

Execution corresponds to depth-first traversal of tree.

Identify activations by function name and actual argument values.



Control stack keeps track of live activations; it contains activations along the path from root to "current" activation (see example above).

ACTIVATION RECORDS (A.K.A. “FRAMES”)

Contain **data** associated with a particular activation of a procedure:

• Actual parameters (maybe in registers).

• Return value (maybe in register).

• Local variables, including temporaries (maybe also saved registers).

• Perhaps a closure pointer or “static link” (= “access link”) to access non-local variables.

Also includes **control** information about the calling procedure:

• Return address in caller.

• Perhaps a “control link” (= “dynamic link”) that points to **caller**’s activation record.

Use **fixed layout** (as far as possible) for activation records, so each frame item can be referenced as:

```
    (item address) = (frame pointer) + (statically-known offset)
```

Most architectures perform such references efficiently.

Activation Record Lifetimes

The **lifetime** of an activation record corresponds to the longest lifetime of anything contained in it.

The lifetimes of all contents begin when the activation begins (i.e., when the procedure is called).

The lifetime of **control** information ends when the activation's lifetime ends (i.e., when the procedure returns).

For most conventional languages, including C, Java, Pascal, etc., the lifetimes of local **data** are also contained within the **activation**'s lifetime.

Thus, since activation lifetimes behave in a stack-like manner, we can allocate and deallocate activation records on a **stack**.

(For some languages, like Lisp, fab, etc., data lifetimes don't obey these rules; such data cannot be stack-allocated. More later.)

We don't "push" and "pop" whole activation records; instead, we build and destroy them in pieces.

Calling Sequence

= Sequence of steps that, taken together, build and destroy activation records.

Typically divide into:

• Call sequence – performed by caller.

• Entry sequence – performed by callee.

• Exit sequence – performed by either/both.

Key problem: caller and callee know very little about each other.

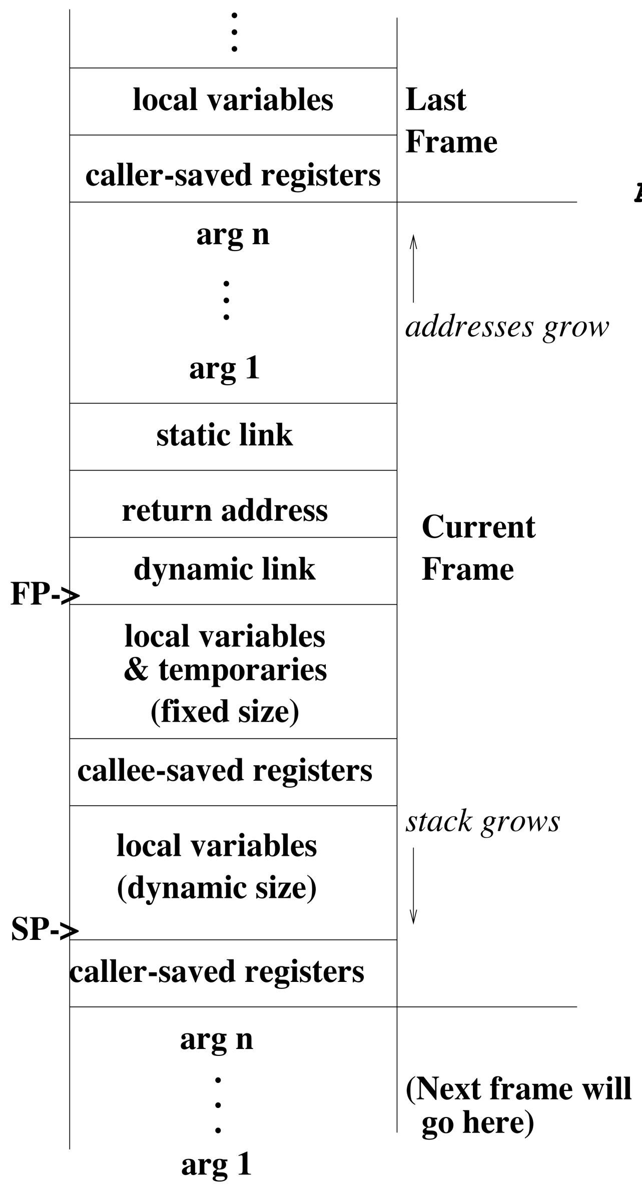
• Single callee may have many callers.

• Caller may not know callee statically (e.g., C function pointers).

• Caller shouldn't need to know details of callee's implementation (may be compiled separately).

Thus, caller and callee must blindly cooperate via a set of **conventions**.

TYPICAL ACTIVATION RECORDS



Addressing

```
arg_i = M[fp+2+i]
local_j = M[fp-j]
old fp = M[fp]
old pc = M[fp+1]
```

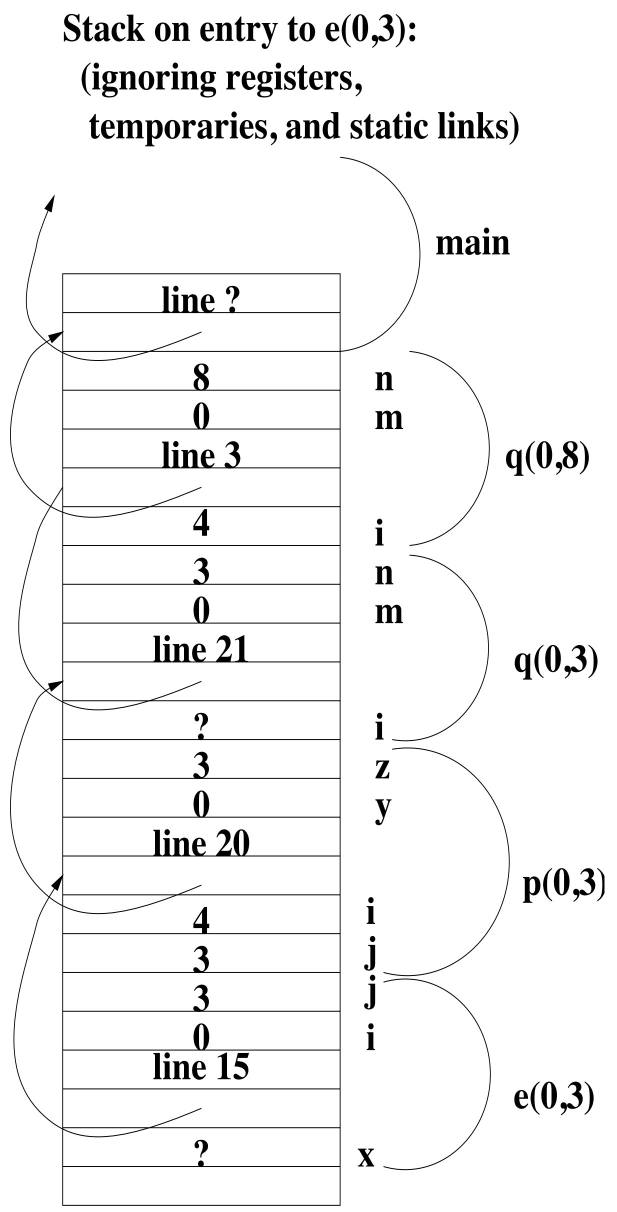
Common variations:

- order of info
- defn. of "frame"
- use of dynamic sizing

Depends on:

- hardware
- convention

Typical Sequence - Example (Quicksort)



Register Management

One crucial aspect of calling convention is how caller and callee manage the registers, which they must share.

In order for caller and callee to both use a register, the caller's value for that register must be saved in memory (usually on the stack) before the callee uses the register, and then restored afterwards.

• Who should do that, the caller or the callee?

• Advantage of a **caller-save** register is that the caller doesn't need to save its value if that isn't needed after the call.

• Advantage of a **callee-save** register is that the callee doesn't need to save its value if the caller doesn't need to use the register.

• Typically, the calling convention partitions register set into some of each. Then compilers try to put short-lived temporaries into caller-save registers, and longer-lived values that must survive calls into callee-save registers.

TYPICAL CALLING SEQUENCE

1. Caller pushes caller-save registers.
2. Caller pushes arguments (in reverse order) and closure pointer or static link (if any).
3. Caller executes `call` instruction, which pushes `pc` (details vary according to machine architecture).
4. Callee pushes `fp` as dynamic link and sets `fp = sp`.
5. Callee adjusts `sp` to make room for fixed-size locals.
6. Callee pushes callee-save registers.
7. Callee can adjust `sp` dynamically during procedure execution to allocate dynamically-sized data on the stack.

Typical Return Sequence

1. Callee restores callee-save registers.
2. Callee resets `sp = fp`, thereby popping locals and any dynamically-sized data.
3. Callee pops dynamic link into `fp`.
4. Callee does a `return`, which pops return address into `pc`.
5. Caller pops static link and args.
6. Caller restores caller-save registers.

Common variations

• Hardware instructions may do more or less implicit stack manipulation, depending on machine architectures. (Example: x86 pushes return address on stack, but many machines put it in a register instead, leaving callee to push it if necessary.)

• Arguments (up to some fixed number) may be passed in registers instead of on the stack.

• Return value is almost always returned in a register (if small enough to fit).

• If everything is the frame has statically fixed size, there's no real need for a frame pointer or a dynamic link – though they can still be handy.

x86-64 Calling Conventions

- First 6 integer arguments are passed in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`.
- First 8 float arguments are passed in SSE registers
- Any additional arguments are passed on the stack.
- Integer return value goes in `%rax`; float return value in `%xmm0`.
- Registers `%rbx,%rbp,%r12,%r13,%r14,%r15` are callee-save.
- The remaining integer registers, and all float registers, are caller-save.
- Use of a frame pointer is optional.

Access to Non-local Variables

• In Pascal, Ada, Haskell, fab, etc., we can nest procedure declarations **inside** other procedure declarations. (Cannot do this in C!)

• Parameters and local variables of outer procedures are visible within inner procedures.

• More precisely, the variables associated with the **most recent** still-live activation of the outer procedure are visible within inner procedures.

• References to these variables must be compiled to code that can locate the corresponding values at runtime.

• The lifetime of such a variable is the maximum lifetime of any inner functions that uses it. So the storage for the variable must live that long too.

Heap Storage for Activation Data

If an inner procedure is fully "first-class" it can outlive the outer procedure in which it is defined, e.g.:

```
{
  func plus (const n:integer) -> (integer -> integer) {
    func plusn (x:integer) -> integer {
      return x+n
    };
    return plusn
  };
  var f := plus(2);
  var a := f(40)
}
```

```
                main
                 /\
                /  \
               /    \
         plus(2)     f(40)   == plusn(40)
```
(requires value `n` = 2)

Activation of `plus` is no longer live when `f` is called!

Heap Storage for Activation Data

• If `n` is stored in activation record for `plus` and activation-record is stack-allocated, it will be gone at the point where `f` needs it!

• Languages supporting first-class nested procedures (e.g., Lisp, Scheme, ML, Haskell, fab, etc.) solve problem by using **heap** to store variables like `n`.

• Simple solution: Just put all activation records in the heap to begin with! (Will be garbage collected when the Garbage collection is a must!)

• More refined solution, which we are using for fab: Represent procedure values by a heap-allocated "closure" record, containing the procedure's code pointer and values of the non-local ("free") variables referenced by the procedure.

• Involves taking **copies** of the values of non-local variables, so only works properly when values are **immutable**.

• If language requires support for mutation, compiler can introduce an extra level of indirection.

Stack Storage and Access Links

In many languages, such as Pascal and its descendents, functions are not fully first-class: they cannot be treated as values that can be returned or stored.

So if procedure `f` is declared inside `g`, then `f` can only appear as descendent of `g` in the activation tree.

This allows us to stack-allocate activation records as usual, and still guarantee that non-local variables will still exist when they are needed.

A standard way of doing this is using **access links**.

For example, consider quicksort again, written using nested (but not first class) functions in fab...

```
{
  var a1 := @integer{9 OF 0};
  var a2 := @integer{99 OF 0};
  func sort (n:integer, const a:@integer) {
    func exchange (i:integer,j: integer) {
        x := a[i]; a[i] := a[j]; a[j] := x
    };
    func quicksort (m:integer, n:integer) {
      func partition (y:integer,z:integer) -> integer {
        var i := y; var j:= z+1;
        ...
        while a[i] < a[y] do i := i + 1;
        ...
        exchange(y,j);
        return j;
      };
      if n > m then {
        var i := partition(m,n);
        quicksort(m,i-1);
        quicksort(i+1,n);
      }
    };
    quicksort(0,n-1)
  }
  sort(9,a1);
  sort(99,a2)
}
```

Access Links for Non-local Variables

• Each activation record can include an **access** (or **static**) **link** pointing to the statically-enclosing activation record.

• If `p` is nested immediately inside `q`, then the access link in `p`'s activation record points to the most recent live activation record of `q`.

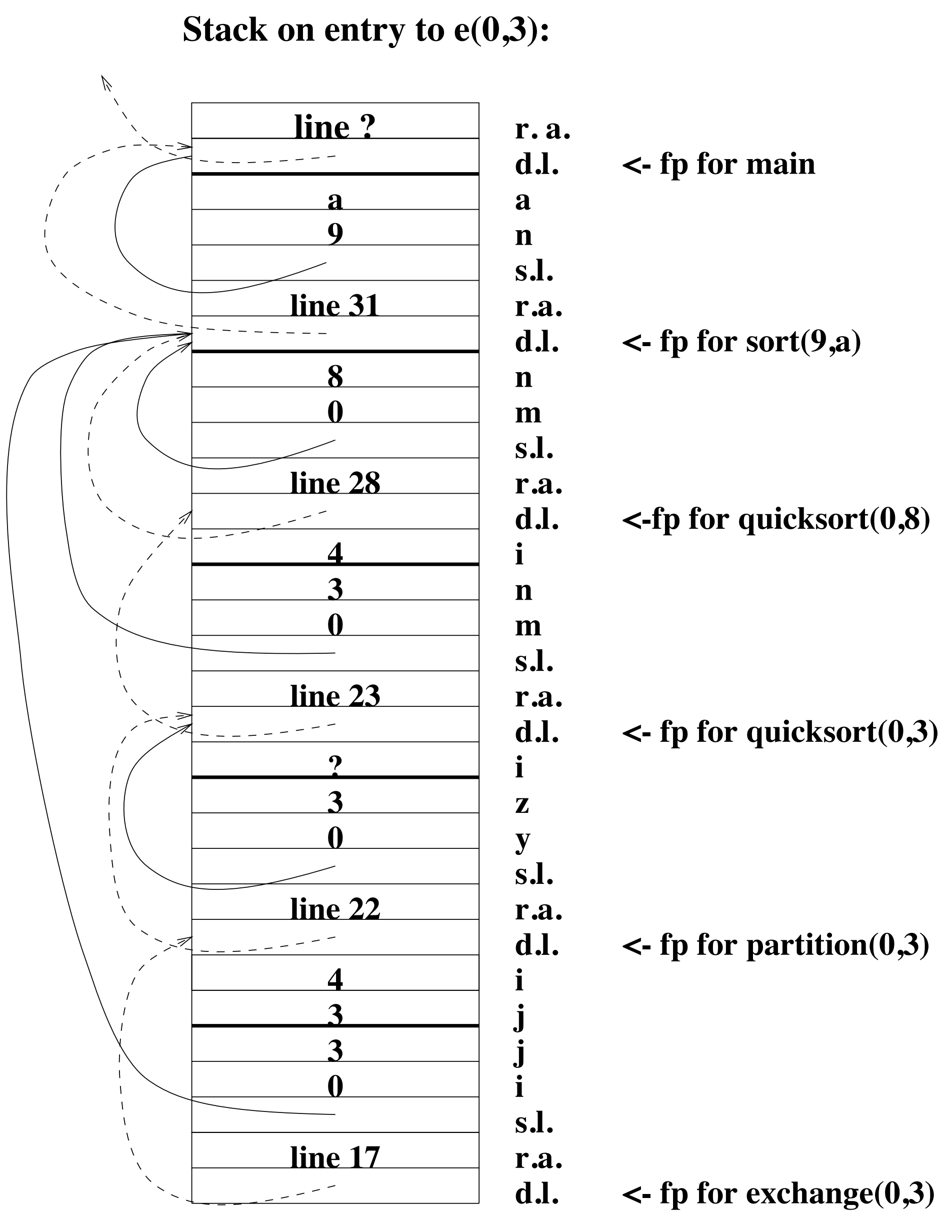
• Non-local `v` is found by following one or more access links to the activation record that contains `v`, and then taking the appropriate offset within that record.

• If `v` is declared locally at depth n_v, and accessed in `p` at depth n_p, then the number of access links to follow is just $(n_p - n_v)$.

• This number, and the offset are fully known at `compile` time.

• Access links are initialized during the calling sequence; usually calculated by the caller and passed as a "hidden" first argument.

Example with Static Links: Quicksort



[Note: return address line numbers in this diagram are incorrect.]