More Procedures

CS152

Chris Pollett

Dec. 1, 2008.

Outline

- Procedure Environments, Activations, and Allocation
- Dynamic Memory Management
- Exception Handling

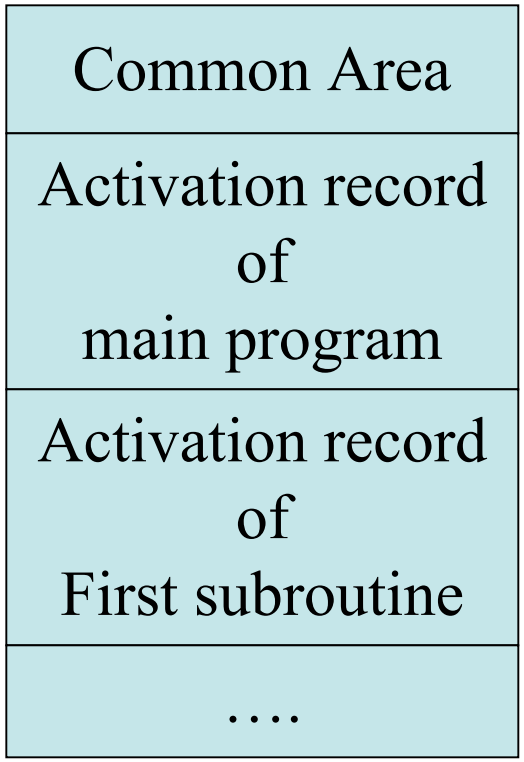
Introduction

- Last Wednesday we were talking about activation records and environments.
- We distinguished between the defining environment (the closure) of a record B versus its calling environment.
- We then looked at different parameter passing mechanisms: pass by value, pass by reference, pass by value result, and pass by name.
- Today, we are going to look at the structure activation records in more detail depending on the run-time environment of the language in question.

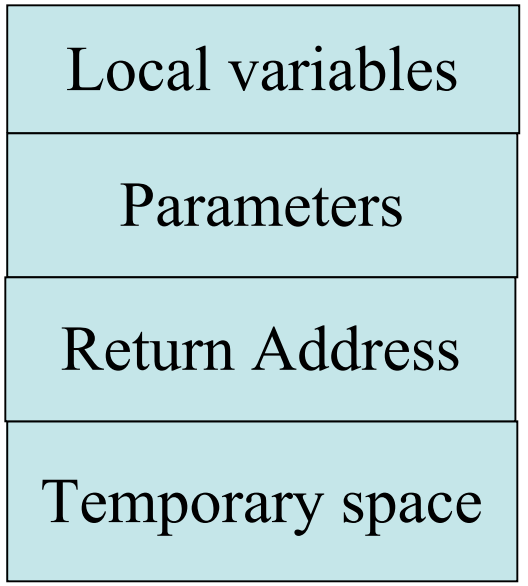
Fully Static Environment

- In Fortran 77 all memory allocation can be performed at load time.
- Thereafter, the locations of all variables are fixed for the duration of program execution.
- Functions and procedure definitions cannot be nested and recursion is not allowed. Thus, everything can be statically allocated.
- Each procedure or function has a fixed activation record, which contains space for the local variables and parameters, and possibly the return address for proper return from calls.
- Global variables are defined by COMMON statements, and are determined by pointers to a common area.

Fully Static Environment Illustrated.



Basic memory layout



Memory layout of an activation record

Stack-based Runtime Environments

- For block-structured languages like C, a procedure may be called again before its previous activation has exited. (So we can do recursion.)
- Thus, a new activation record (**stack frame**) must be created on each procedure entry.
- This is usually done using a stack. The exact format of a stack entry is implementation specific, we mentioned a different version earlier in this semester than the book uses. However, the book's version is useful for when we consider slightly more complicated languages like Ada.
- Each activation record on the stack, contains much the same information as in the Fortran case: local variables, parameters, return address, temporaries.
- There is an **environment pointer** which points to the start of the current activation record on the stack.
- Each activation record also has a pointer called a **control link**, which points to the start of the activation record that control will return to after the given procedure ends.
- The location of variables in a record is given as an **offset** from the start of the record (I.e., where the environment pointer point to).

Runtimes with Nested Procedures

- Some block-structured languages like Ada and Pascal, allow you to define one procedure inside of another.
- If P2 is defined within P1 it might make use of variables within P1 in its definition.
- In order to be able to find these variables' values, in addition to a control link, there must also be an access link, which point to the activation record in which the function was defined (the defining environment).
- Since you can nest P3 inside P2 inside P1, you might need to follow several such links (**access chaining**) to actually look up a variable.
- Ada does not allow the use of P1 to create a P2 that persists after P1 is done. So we can't create the make-balance closure example we had in Scheme.
- To implement that we need to move away from using a stack and instead to allocate activation records in a more heap like way and use garbage collection.

Dynamic Memory Management

- So what methods can we use to determine when an activation is no longer needed in a language like Scheme?
- We are allocating on a heap. We keep a free space list. We coalesce contiguous free blocks, and we might compact the heap occasionally.
- One way to keep track of who is referencing an activation is to use a reference count.
- One problem with this is circular references.
- Another technique is to use a **mark and sweep** approach. First mark each record reachable from the global environment. Then iterate and mark each record reachable from those. Do until no change. Anything not referenced at end can be garbage collected.

More Garbage Collection

- One improvement to this is called **stop and copy**:
  - Split heap memory into two halves; only use one halve at a time.
  - To garbage collect have a mark phase as in mark and sweep, but copy marked items to unused half.
  - Then make that half active and garbage collect the whole previously used half.
- Another is called **generational garbage collection**:
  - Have several heaps which are garbage collected with different frequencies.
  - If an item is not garbage collected in a high frequency collected heap a couple times, it is moved to a more slowly garbage collected heap.