# CS143 Compilers

Lecture 10

August 6, 2001

## Run-Time Environments

• Before we translate our intermediate code to target code, we must be able to associate the objects in the source program with objects in the environment in which the corresponding target program will execute.

• In particular, provisions must be made to allocate space for data, provide access to this data, and process procedure calls as intended in the source program. Often, a run-time environment must be able to support different conventions from source languages for similar tasks, such as passing parameters to procedures.

## Subdivisions of Run-Time Memory

• At run time, a target program is allocated a block of storage in which to run.

• This storage space may be divided to include the following:

– target code

– static data

– a stack to hold blocks associated with procedure calls, called *activation records*

– space for dynamically allocated blocks of memory, known as a *heap*.

## Activation Records

• An *activation record* is a block of memory containing information relevant to a single invocation of a procedure.

• Activation records are typically maintained on a stack by the run-time system.

• When a procedure is called, an activation record is created and pushed onto the stack, and control is passed to the code for the procedure.

• Upon return, the activation record is popped and de-allocated, and control returns to the calling procedure.

## Compile-Time Layout of Local Data

• Data is typically assigned at compile time, with each object in the source program assigned a block whose size is determined by the object's type.

• Each declaration of a local variable is assigned an *offset* value at compile time. The memory for this variable is allocated at *offset* bytes from a base address that is determined at run time.

• The actual location may be adjusted due to addressing constraints of the target machine, such as alignment considerations.

## Storage-Allocation Strategies

• Different data areas are managed using different storage-allocation strategies.

• Static allocation is used to allocate static data such as global variables and code.

• Stack allocation is used to allocate activation records for procedures.

• Dynamic allocation is used to assign variable-length blocks of memory, the size and lifetime of which cannot be determined at compile time.

## Static Allocation

- When using static allocation, addresses within the program's assigned memory are determined at compile time.
- This strategy allows values of local variables to be retained across procedure calls, since local names are consistently bound to the same location in memory.
- Some restrictions are:
  - – Recursion is problematic, since different instances of local variables share the same memory location
  - – Objects cannot be allocated dynamically.

## Stack Allocation

- Using stack allocation, a block is allocated at run time, of size determined at compile time.
- Objects local to the block are allocated space within that block at offsets determined at compile time.
- Each block, upon allocation, is pushed onto a stack. A pointer is maintained to the previous block on the stack, to provide access to data within that block, and other blocks.
- The data is de-allocated when the block is popped off the stack.

## Calling and Return Sequences

- When procedure calls are handled using stack allocation, with each block playing the role of an activation record, a compiler must generate segments of code to implement the invocation of, and return from, a procedure. These segments are *calling sequences* and *return sequences*.
- A calling sequence allocates an activation record and enters information into its fields, such as parameter values and return addresses. A return sequence restores the state of the machine so that the calling procedure can continue execution.

## Placement of Data within an Activation Record

- As a rule of thumb, data whose size is fixed is best placed in the middle of an activation record.
- Actual parameters and a return value should be placed at the beginning (which is also the end of the caller's activation record), since the procedure may accept a variable number of arguments or return variable-length data, and the caller must know where to start placing this data, without necessarily knowing the size of the callee's activation record.
- Space for temporary variables is unknown until the last stages of compilation, since it may change during optimization.

## Heap Allocation

- Heap allocation is necessary if values of names that are local to a procedure must persist beyond the deallocation of an activation record.
- Heap management can be difficult because much bookkeeping is required to keep track of blocks of arbitrary length being allocated and deallocated during execution, resulting in memory becoming fragmented.
- It is helpful to handle small blocks as a special case, maintaining a linked list of free blocks of a small size. The goal is to keep heap management time proportional to the amount of data being allocated, as that is often proportional to the amount of computational effort devoted to that data.

## Parameter Passing

Different languages use different conventions for passing parameters, which must be considered during compilation:

- – Call-by-value: the values of the parameters are copied to the activation record
- – Call-by-reference: pointers to the actual parameter locations are copied to the activation record
- – Copy-restore: a hybrid between call-by-value and call-by-reference
- – Call-by-name: procedure bodies are textually substituted for the call, replacing formal parameter names with actual parameter names.