Functions, Function Implementation, and Memory Management

Procedure Definition and Activation

- Mechanism in a language for abstracting a group of actions or computations
- The group of actions is called the body of the procedure, with the body represented as a whole by the name of the procedure
- Defined by providing a specification or interface and a body
    - The specification gives the name of the procedure, a list of the types of names of its formal parameters, and the type o its returned value

Activation

- You activate, or call, a procedure by stating its name, together with arguments to the call, which correspond to its formal parameters

Callee

- Call to a procedure transfers control to beginning of the body of the called procedure (the callee)

Caller

- When executing reaches the end of the body, control is returned to the caller
- Or with something like a return statement

Non Local References

- References to variables declared outside of its own body

Semantics

- A procedure is a block whose declaration is separated from its execution

Environment

- Determines allocation of memory and maintains the meaning of names during executing

Fully Static Environments

- All memory allocation can be performed at load time, and the locations of all variables are fixes for the duration of the program execution

- Function an procedure definitions cannot be nested (it is all global)
- All the information associated with a function or subroutine can be statically allocated
- Each procedure or function has a fixed activation record, which contains space for the local variables and parameters

Stack Based Runtime Environments

- In a block-structured language with recursion, activations of procedure blocks cannot be allocated statically
- Because a procedure may be called again before its previous activation is exited, a new activation must be created on the stack every time a block is entered and released on exit
- Several additional pieces of information are required
  - Pointer to current activation must be kept
    - This is because each procedure has no fixed location for its activation record, but the location of its activation record may vary as execution proceeds
    - This pointer must be kept in a fixed location (usually a register), and is called the environment pointer or ep, since it points to the current environment
  - Pointer to activation record of the block from which the current activation was entered
    - This is the activation of the caller in the case of a procedure call

Access Chaining

- When blocks are deeply nested, it may be necessary to follow more than one access link to find nonlocal reference
- Used with deep nesting

Limitations

Dangling Reference

- Any procedure that can return a pointer to a local object, either by returned value or through a pass by reference parameter, will result in a dangling reference when the procedure is exited, since the activation record of the procedure will be deallocated from the stack

Reference Counts

Activation Record

- Activation record shows where things like x and why are allocated
- Tracing back all the way to global environment

Defining Environment

- The environment whose context the block is in
- Remains constant

Calling Environment

- Where is the block being called from
- Can change
- Dynamic environment

Forms

Closed Form

- Containing no non-local dependencies
- Block contained within itself
- Overloading is not a closed form

Closure

- Resolves all outstanding nonlocal references relative to the body of the function
- One of the major tasks of a runtime environment to compute closures for all functions in cases where they are needed

Parameters

Formal Parameters

- Parameters without value

Actual Parameters

- This is an argument
- The actual values for the parameters

Parameter Passing Mechanisms

- The way in which parameters are replaced by the arguments during a call

Pass by Value

- The most common mechanism for parameter passing
- The arguments are expressions that are evaluated at the time of the call
- The argument values become the values of the parameters during procedure execution

- Changes can still occur outside of the procedure (e.g. pointers or reference types)

Pass by Reference

- Passing variable location instead of passing the value
- An argument must be like a variable with an allocated address
- The parameter becomes an alias for the argument
- An changes made to the parameter occur to the argument as well

Multiple Aliasing

- See slides

Pass by Value-Result

- Similar result to pass by reference, except no actual alias is established
- The argument value is copied and used in the procedure
    - Then, the final value of the parameter is copied back to the location of the argument when the procedure exits
- Also known as copy-in copy-out or copy-restore

Pass by Name and Delayed Evaluation

- Argument not evaluated until its actual use as a parameter in the called procedure

Translator

- A translator can prevent many violations of these parameter specification

Type Checking

Binding

Deep Binding

- Closures allow functions to retain access to their lexical scope
- Useful for maintaining state across multiple calls

```
def  make_counter():
     count = 0 # Enclosing variable

     def counter():
          nonlocal count # Access the enclosing variable
          count += 1
          return count
     return counter

counter1 = make_counter()
print(counter1()) # Output: 1
print(counter1()) # Output: 2
```

- The function retains access to its lexical scope, meaning that it can access the variables that were in scope when the function was defined, regardless of where the function is called

Shallow Binding

- The function resolves variables based on the current calling environment
- Less control over the variable's state compared to deep binding

```
x = 10
def f():
    return x

def g():
    x = 20
    return f() # Calls f in a different scope

print(g()) # Output: 10 (x is still 10 in the global scope)
```

- Variable's most recent value is used at the time of the function call
  - Means that the function uses the current environment, which may change depending on where the function is called

Ad Hoc Binding

- Binding can change based on how methods or functions are called
- Flexible but can lead to confusion if not managed carefully

```
class A:
    def method(self):
        return self.value

class B:
     def __init__(self):
         self.value = 5

b = B()
a = A()
a.method = b.method # Ad hoc binding of method

print(a.method()) # Output: 5
```

demonstrates how you can bind a method from one class to an instance of another class

- Variables are bound in a context-dependent manner, often determined by the specific function or usage rather than a strict lexical or dynamic scope

Memory Management

Dynamic Memory Management

- In a typical imperative language, automatic allocation and deallocation of storage occurs only for activation records on a stack
- Space is allocated for an activation record when a procedure is called and deallocated when the procedure is exited
- Explicit dynamic allocation and use of pointers is also available under manual programmer control using a heap of memory separate from the stack
- Manual memory management on the heap suffers from a number of potential problems, including the creation of garbage and dangling references
- Languages with significant needs for heap storage are better off leaving nonstack dynamic storage to a memory manager that provides automatic garbage collection
- Any language that does not apply significant restrictions to the use of procedures and functions must provide automatic garbage collection, since the stack-based system of procedure call and return is no longer correct
- Functional languages which have first-class function values and object oriented programs fall into this category
- A very simple solution for dynamic memory is to no deallocate memory once it has been allocated
    - Every call creates a new activation record
    - Advantages
        - Correct
        - Easy to implement
        - Can work for small programs
    - Disadvantages

- Not option for most programs written in functional and object-oriented languages
- Memory quickly exhausted if deallocation does not occur
    - Internal representation of data uses pointers and requires a substantial amount of indirection and memory overhead
    - Has been used in conjunction with virtual memory
    - Causes swap space to become exhausted and can cause serious performance due to page faults

Maintenance of Free Space

- A contiguous block of memory is provided by the OS for the use of an executing program
- Free space within this block is maintained by a list of free blocks
- Linked lists can implement maintenance of free space
    - Allocated blocks shaded and free blocks blank
    - Circular list
- When a block needs to be allocated, the memory manager searches for a free block with enough space, then removes that block from the free list

Coalescing

- When memory is reclaimed, blocks are returned to the free list
    - Must be joined with adjacent blocks to form the largest contiguous block of free memory

Fragmentation

- A free list can become fragmented

Storage Compaction

- Memory must occasionally be compacted to put all the free blocks together and coalesce them
- Swap space is the intermediary for this

Reclamation of Storage

Reference Counting

- Considered an eager method of storage reclamation
    - Tries to reclaim space as soon as it is no longer referenced
- Each allocated block contains count field to count references
    - Deallocates block when count reaches 0

Disadvantages

- Extra memory for count field
- Effort to maintain the counts
- Circular references can cause unreferenced memory to never be deallocated

Mark and Sweep

- Lazy approach
  - Will put off reclaiming storage until allocator runs out of space
    - At which point it looks for referenceable storage and moves all unreferenced storage back to the free list
- Two passes
  - First pass follows all pointers recursively, starting with current environment or symbol table
  - Second pass sweeps linearly through memory, returning unmarked blocks to the free list

Disadvantages

- Requires extra storage
- Double pass through memory causes significant delay
- Not good for interactive applications at scale
- Solutions
  - Stop and Copy
    - Split storage into two halves and allocated storage from half at a time
    - During marking pass, all reached blocks are immediately copied to the second half of storage not in use
    - No extra markbit required
    - One pass required
    - Once all reachable blocks in the used area have been copied, the used and unused halves of memory are interchanged
    - Doesn't improve time, only storage reclamation

Generational Garbage Collection

- Adds permanent storage area to the reclamation scheme
- Allocated objects that survive long enough are copied into permanent space and are never deallocated
- Only a small amount of memory needs to be searched
  - Very quick
- Permanent memory could still become exhausted, but this is a much smaller problem
- Process has been shown to work well

Scoping

Phases

Design Phase

- Bindings between primitive constants, types, and operations of the language are designed

Program Writing

- Binding of an identifier to a variable

Compile Time

- Allocates memory space for some of the data structures that can be statically processed
- Global variables
  - The connection between the identifier and the corresponding memory location

Run Time

- Entire period of time between program start and program end
- All associations that have not been created must be formed at runtime
  - Binding of identifiers to memory locations for the local variables in a recursive procedure

Declaration

Aliasing

Rules

Dynamic Scoping

- Looks sequentially back through the activation record for latest binding for what it is looking for
- Essentially, looking at the most recent environment

Static Scoping (Lexical Scoping)

- First look at the block itself
- If you can't find it at the block check at the outer block (structurally) and draw from that block
- Resolving with block body rather than sequential activation record
- Essentially, a variable always refers to its top level environment

- Unrelated to call stack