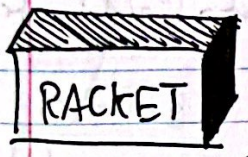


implicit in a bunch of places, incl. define

side effects

- **(begin)**: evaluates a bunch of expressions in a row
  - → value: what last one evaluates to
  - → previous ones have side effects, which affect result.
  - → basically, do these things in order

output

- **(display)**: prints value (no line break)
- **(printf)**: prints w/ formatting

mutation

- **(set!)**: mutates a value that has been defined
  - → returns void
- **(box)**: points system
  - → box contains a value (retrieved w/ **unbox**)
    - → it is basically an address to the value
  - → when we mutate the value inside (using **set-box!**), the address remains the same, which creates continuity

input

- (read-line): reads in the next line from input
- (read): more advanced kind of input that reads an S-expression from the input (may cover multiple lines)

Structs as Closures: (struct closure (var body env.) #:transparent)

- → A closure of a function is a data structure (?) that contains a function's parameters, body, and the environment at the function when it was defined
  - → Environment is included since the outside environment may have entries that mirror the same IDs as a parameter, which would cause a scope error if the function were simply evaluated as normal

Hilroy

Commands

(set! struct1 struct2)

(set-struct-field! struct1 v)

Mutable Structs

-> if #:mutable is added to a structure, it becomes mutable
-> This implies a box-like structure is used somewhere
  -> Specifically, a struct acts like a box for its fields
  -> When a field is accessed, it looks up the value
  -> When a field is mutated, the specific value is mutated (but the struct is not (i.e. the struct is not itself boxed))

Vectors

-> A slice of consecutive memory that can be accessed by index in O(1)
  -> Much like an array in C
-> Vectors have fixed sizes that are declared on creation
-> Unlike C, Racket vectors can have any type (or a mix)
  -> This because they contain boxes, not the values themselves
-> Retrieval : (vector-ref vectorName index)
-> Mutation : (vector-set! vectorName index value)
-> Looping : (for/vector ([i n]) (f i))
  -> n is length of vector, i is current index, item at index is declared as (f i)

Correctness Proofs

-> Proof that a program always acts "correctly"
-> Structurally recursive program => inductive correctness proof
  -> These usually follow the structure of the program
-> Accumulative Recursion => proving an invariant (invariant)
  -> prove that ∀ L, acc (sum-list-help L acc) = acc + (sum-list L)
  -> Case 1: L is empty => (sum-list L acc) = acc = sum of list + acc
  -> Case 2: L = cons (x L'). Assume (sum-list-help L' acc) produces sum of L' + acc. Then (sum-list-help L acc) = (sum-list-help (cons x L') acc) = (sum-list-help (+ x acc) L') = (+ (sum-list L') x) + acc = (sum-list L) + acc
-> Then, let acc = 0; then (sum-list-help L 0) = (sum-list L)

stdio.h : standard in/out
ctype.h : types
math.h : math

Header Files

-> Contains a list of headers that are related to a given task
  -> Headers need to be included / stated before they are used; header files save us from having to paste headers in all the time
-> Usage: #include <headernames.h> // at start of program
  -> Tells the preprocessor to put the contents of the file into the main program
-> However, the header file only contains headers, not definitions
  -> These are stored in a common location and linked by the linker

Printf / Scanf — requires #include <stdio.h>

-> printf("format", var1, var2...) // prints var1, var2 based on format
->scanf("format", &var1, &var2...)
  -> Scans in the specified type and stores them in the addresses provided
  -> " %d" strips whitespace and just gives the next number

Declaration / Definition

-> Variables, functions can be declared before they are defined

```
int a;          // a is declared, not given a value
a = 2;          // a is given value 2
int f(int x, int y)  // function declared (header)
```

-> Declarations alert the compiler that such a variable / function will exist, but it won't need to be defined now

Hilroy

lifetime: whole program
scope: function where it is declared

Static Types

-> Static variables declared in functions are global variables that is only visible to the function that calls it
  -> It outlives the function scope
-> Prevents tampering with global variables used by functions
-> static int a = 23 // declaration and defn of static int

Global Types

-> variable declared in scope of whole program (top-level) is accessible by whole program
-> mutation of global variables leads to unpredictable behavior

Memory Model

-> Memory: sequence of cells that contain a fixed-size number
  -> Code: piece of memory where code is stored
  -> Static: holds static and global variables
  -> Heap: data that must outlive its scope
    -> Space here can be arbitrarily requested w/ malloc
  -> Stack:
    -> each function call allocates a part of the stack (frame)
    -> contains all the local variables being used at the time
    -> also contains return address
    -> when function call ends, stack frame is popped from the stack (stack pointer points to next highest frame)
    -> DO NOT RETURN POINTERS TO THE STACK!
      -> [redacted] during this creates undefined behavior

Malloc / Free

-> int *p = malloc(sizeof(int)) // an int-sized piece of memory
  -> allocates memory in the heap for p
-> free(p) // releases this memory back to the heap
-> dereferencing p is undefined behavior since the memory is freed
  -> setting p = NULL; prevents possible aliasing since NULL is always an invalid location

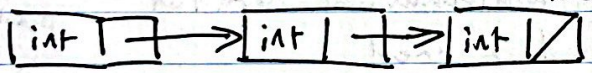
Function Pointers

-> Functions are not first-class values in C so they can't be stored as values / passed as parameters
-> However, pointers to functions are, so those can be passed as parameters and stored
  -> Type: (*f)(int) points to a function f(int...)
  -> int (*f)(int) = f // where f is a function that accepts an int

Linked Lists

-> contain data and a pointer to the next node
-> Traversal: declare a pointer, while next ≠ NULL, pointer becomes the *next pointer.

```
struct Node {
  int data = ...;
  struct Node * next = ...;
}
```



Static Functions

-> Functions defined w/ static can only be accessed from inside the function where they are defined
-> This can prevent tampering / forgery

Realloc

-> This increases the amount of memory allocated
-> *p = realloc(*p, newSize)
-> This implies copying all the data to a new location, which has a linear time cost

Hilroy

Other

δ : defining (environment)
π : program (reduced by substitution)
ω : output stream
τ : input stream

-> Modeling (δ, π, ω, τ)
-> Currying : modeling functions as returning functions
  -> An idiomatic feature of Haskell
-> Referential transparency : functions produce same result each time
  -> Not guaranteed w/ mutation, side effects
  -> Happens in pure racket
-> Memoization : caching previously computed results and looking them up instead of computing them again
  -> Application of mutation that improves efficiency
-> Aliasing : two or more pointers point to same location
-> ADT : Abstract data type
  -> Data type defined by mathematical notation
  -> Set of operations defined as well
  -> Sequence : a stack, basically (stored in an array)
    -> Sequences need to resize in order to be unbounded
    -> Resize to double the size each time => approx. constant time add

amortized analysis : stretch out behavior, notice patterns, infer time

-> Dictionary : key-value pairs (w/ search operation)
-> Tampering : changing the internals of a data structure
  -> If this can happen, data structure may not behave as expected
-> Forgery : creation of a false data structure (using something other than the intended creation method)
-> Solution : write the functions to work w/ a pointer to the structure, then just include the headers (???)
-> Encapsulation : hiding information so that data structures can only be used through an "interface" (thus confining behavior)