CS 146 Notes

velo.x

Instructor: Robert Hackman
Section 002

Contents

1 Course Info

1.1 Main Theme

- Side-effects(impurity)
- Programs that "do things"
- Imperative programming
- Impure Racket
- C
- Low Level Machine

1.2 Review

Review all material from CS 145,

- proofs
- Big-O
- lambda calculus
-

1.3 Programming Language

- Full Racket, C
- Use linux.student.cs environment

2 Lecture January 7th

Imperative Programming is **Harder**.

2.1 Side Effect:

Example:

- text being printed to the screen
- reading input from the user
- values of variables change

All change the state of the world. The state of the world affects the program.

For example,

```
(define (f x) (+ x y))
```

depends on the current value of y. Thus the semantics of an imperative program must take into account the current state of the world when changing The state of the program

$\Rightarrow$ Temporal component inherant in the analysis of imperative programs. Ask not "what does function do?", but "what does this function do at this point of time?".

"Why study imperative programming?"

"Because the world is imperative...

- machines work by manipulating memory
- even functional programs are eventually executed imperatively

....or is it?"

Is the world constantly mutating or is it constantly being remutated? When a character appears on the screen does that change the world or create a new one?

Either way, the imperative programming more closely reflex the real world experience.

But a functional world does offer a unique view on side-effects.

2.2 Structural Recursion

The structure of the program matches test of the data.

```
(define (fact n) (if (= n 0) (* n (fact (- n 1)))))
```

Characteristics:

- The cases in the function match those in data
- The recursive call uses arguments that either stay the same, or get 1 step closer to the base case.
- example:

```
(define (length l) (cond [(empty? l) 0] [(else (+ 1 (length (rest l))))]))
```

a (list of X) is either empty or (cons x y) where x is an X and y is a (listof X)

If recursion is structural than the structure of the program matches its proof of correctness by induction claim (length l).

Proof.
Base Case: l is empty, its length is 0 and our function immediately returns 0.

Prove (length (cons x l)) is correct.

Our program would add1 to (length l) which is the length of list l by I.A., so (length l) is correct. □

A correctness proof is just a restatement of the program itself.

2.3 Accumulative Resursion

Characteristic: One of more extra parameters that "grow" while the other parameters "shrink".

```
(define (sum-list L)
    (define (sum-list-help l acc)
        (cond [(empty? L) acc]
            [else (sum-list-help (rest L) (+ (first l) acc))]
        (sum-list-help L))))
```

proof method: induction on invariant

example: to prove that (sum-list L) sums L, suffices to prove (sum-list-help L 0) produces the sum of L.

Attempt to prove by structural induction on L, Assume (sum-list-help L' 0). fails.

Need a stronger statement about the relationship between L and the acc that holds through out the recursion - an invariant.

Proof. we prove the invariant $\forall L, acc$, (sum-list-help L') produces acc + (the sum of L')

Structural induction on L.

Case 1: L is empty, then

(sum-list-help L acc) = (sum-list-help empty acc) $\rightarrow$ acc = acc + (sumoflist L).

Case 2: L = (cons x L')

Inductive Assumption: assume (sum-list-help L' acc) produces acc + (sumoflist L'),

(sum-list-help L acc) $\rightarrow$ (sum-list-help (cons x L') acc) $\rightarrow$ (sum-list-help L' (+ x acc))
$\rightarrow$ (sumof L') + (+ x acc) = X + (sumof L') + acc = sumof L + acc

Then in our function we have

(sum-list-help L 0) $\rightarrow$ (sumof L) + 0 = (sumof L)

□

2.4 Generative Recursion

Characteristic: the program does not fikkiw tge structure of the data.

```
(define (foo x)
    (cond [(= x 1) 1]
          [cond [(even? x) (+ 1 (foo (/ x 2)))]
                [else (+ 1 (foo (+ (* 3 x) 1)))]]))
```

How do we reason about imperative programs?

2.5 Impure Racket

(begin exp1 ... expn)

- evaluates all from left-to-right
- produces the value if expn(the last one).
- useless in a pure functional setting
- useful if these expressions are being evaluated for their side effects, not their produced value.
- implicit begin in the bodies of functions, lambdas, local, answers of cond/match

3 Tutorial Jan 8

- Millen 1956 - 7 Chunks in working memory
- Cowen, Chen. Rouder, 2010s
- Peterson and Peterson 1959 - working memory lasts for 20 seconds
- Ebbinghaus, 1885, forgetting curve

CS 145 $\rightarrow$ CS 146 $\rightarrow$ CS 240 E , CS 246E, CS 241E, cs 251 CS 245E

3.1

MAC, Linux -open terminal

commands

- pwd
- ls
- ls -a
- ls [dir]
- ls -l
- cd
- mkdir
- exit
- echo
- control C
- control D
-

Text Editors:

- nano
- vim
- emacs

4 Lecture January 9

Reasoning about side-effects for pure functional programming-substitution model

Can the substitution model be adapted?

- "State of the world" is an extra input + extra output at each step
- each reduction step transforms both the program and the state of the world

How to model the "state of the world"?

- simple case: list of defines
- more complex cases: Memory Model (RAM)

For now conceptualization of the machine, memory is a sequence of "boxes".

- indexed by natural numbers "addresses"
- containing a fixed-size number (say 32-bits)
- any box's contents can be fetched in O(1) time

4.1 Modelling Output

"simplest kind of side-effects"

- "state of the world" is the "sequence of charts that have been printed to the screen"
- each step of computation potentially adds characters to this sequence
- every string is just a sequence of chars:

$$\text{(string} \rightarrow \text{list "abcd")} \Rightarrow \text{(list } \#\backslash a\ \#\backslash b\ \#\backslash c\ \#\backslash d\text{)}$$

4.2 Substitution Model

$\pi_0 \Rightarrow \pi_1 \Rightarrow \pi_2 \Rightarrow \cdots \Rightarrow \pi_n$.

Each π_i is a version of the program, obtained by applying one reduction step to π_{i-1}.

Now also: $\omega_0 \Rightarrow \omega_1 \Rightarrow \omega_2 \Rightarrow \cdots \Rightarrow \omega_n$.

- each ω_i is a version of the ouput sequence.
- each ω_i is a prefix of ω_{i+1} (can't unprint chars)

4.3 Combined

$(\pi_0, \omega_0) \Rightarrow (\pi_1, \omega_1) \Rightarrow \cdots \Rightarrow (\pi_n, \omega_n)$.

But some program reductions create definitions, e.g. local

-defined values will eventually change

-better to Separate out the sequence of defines δ

$$(\pi_0, \omega_0, \delta_0) \Rightarrow (\pi_1, \omega_1, \delta_1) \Rightarrow \cdots \Rightarrow (\pi_n, \omega_n, \delta_n)$$

δ_0, ω_0 empty

Example 1: If π_0 =(define id exp) $\cdots$

- reduce exp according to the usual CS 135/145 rules: may cause chars to be sent to ω
- exp will now be reduced to val
- remove (define id val) from a and add it to δ

Example 2: If $\pi_0 =$ exp $\cdots$

- reduces exp by the usual rules: may cause chars to be sent to ω
- exp is now reduced to val remove from π
- chars that make up val to ω
- When π is empty, done.
- δ, ω, that which changes, other than the program itself.
- ω relatively harmless: changes to ω don't affect the running of the program
- δ: not a program yet because variables are not carrying (adding new definitions is not really a state change)

4.4 Functions Affecting ω

- (display x) : outputs the value of x, no line break following it
- (newline): line break
- (printf "The answer is $\sim a.\backslash n$" x)

  -formatted print - value of x replaces a

  -newline character (as a Racket character #\newline)

Question: but then what do display newline, printf return?

Answer: special value $\#\langle void \rangle$

- not displayed in Dr Racket, also the result of evaluating (void).
- Functions that return $\#\langle void \rangle$ are often called statement or command (where imperative programming gets its name).

Recall: map (map f (list l_1, ..., l_n)) produces (list (f l_1) (f l_2) (f l_n))

If f is a statement, f is needed for side-effects, produces $\#\langle void \rangle$ then (map f (list l1,...,ln)) produces (list$\#\langle void \rangle$, $\#\langle void \rangle$, ...,$\#\langle void \rangle$) not useful.

(for-each f (list l1, ... ln)) performs

Note:

- (display 542) and (printf "$\sim a$" 542) are the same
- (void) is a function, $\#\langle void \rangle$ is a value.

5 Jan 14

Reasoning about ouput ctd:

- before we had output:
- order of operations didn't matter (assuming no crashes/non-termination)
- now order of evaluation may effect order of output
- all non-terminating programs could be considered equivalent and not meaningful.
  -now, non-terminating programs can do interesting things e.g. printing digits of π.
- semantic made I should include the prossibility of non-terminating porgrams
  -meaning: what the program would product "in the limit"
  -Ω (set of possible values of ω) : would include finite + infinite sequences pf chars

What if you want to save the output? (with-output-to-file "file.txt" (lambda () (printf "Test\n")))

(with-output-to-file): invokes the thunk and send the output to the file.

Better: -user decides where output goes -linux shell -output redirection

Why do we need output?

- many languages: compile, link, evaluate, cycle.
  - program is translated (by a compiler) to native machine code
- only see output if the program prints it
- e.g.
  #include <studio>
  int main(void){
  printf ("hello.world! \n");
  return 0;
  }
- a use in Racket: tracing programs
  e.g.
  (define (fact n)
  (printf "fact applied to argument ~n" n)
  (if (zero? n) 1 (+ n (fact (- n 1)))))

5.1 Modelling Input

- infinite sequence consisting of all chars the user will ever press ι (*** model now $\pi, \delta, \omega, \iota$)
- accepting an input char = removing a char from ι

Small Problem: the sequence may depend on the output

-the users decide what to input in response to what is displayed on the screen -so a more realistic model of input would prehaps not assume all input is available at once

Alternative: -a request for input yields a function consuming one or more characters and producing the next program πm with input chars substituted for the read request

e.g. (read-line) $\rightarrow \lambda(line)line$ (use types abc) $\rightarrow$ "abc"

-entire program reduces to big "nesting" of input request functions -one function per prompt -supplying user input for each prompt yields final result

Input in Racket: (read-line) -produces a string consisting of all chars pressed until the first newline

(string does not contain the newline)

e.g. (string→list (read-line)) (if we type Test) $\Rightarrow (\#\backslash T\ \ \#\backslash e\ \ \#\backslash s\ \ \#\backslash t)$

Less primitive input:

- read: consumes from input, and produces, an S-expression

6 Jan 16

Let's work out our own read:
process usually happens in 2 steps (lots more details in 241)

6.1 Tokenization

Step 1 of read: convert the sequence of raw chacracters to a sequence of tokens (meaningful "words")
e.g. left paren, right paren, identifier, number
-"id" start with a letter
-"numbers" start with a number

Key Observation: peeting at the next char tells us what kind of token we will be getting and what to look for to complete the token

```
(struct token (type value))
 token: kind of token, 'lp, 'rp, 'id, 'num
value: "value" of the token(numeric value, name, etc)

(define (token-leftpar? x) (symbol=? (token-type x) 'lp))
(define (token-rightpar? x) (symbol=? (token-type x) 'lp)) ; read id: → (listof char)

(define (read-id)
      (define nc (peek-char))
      (if (or (char-akphabetic? nc) (char-numeric? nc))
            (cons (read-char) (read-id))
            empty))

(define (read-number)
      (define nc (peek-char))
      (if (char-numeric? nc)
            (cons (read-char) (read-number))
            empty))

Main Tokenizer: read-token → token
(define (read-token)
      (define fc (peek-char))
      (cond [((char-whilespace? fc) (read-token))]
            [(char=? fc # \c) (token 'lp fc)]
            [(char=? fc #\) ) (token 'rp fc)]
            [(char-alphabetic? fc) (token 'id (list->symbol (cons fc (read-id))))]
            [(char-alphabetic? fc) (token 'num (list->number (cons fc (read-number))))]
            [else (error "lexical error")]))
```

Note: (list->symbol, list->number) do not exist, but are easy to build.

6.2 Parcing

-are the tokens arranged into a sequence that has the structure of an $S - exp$?

-if so, produce the S-exp

```
Helper: read-list → (listof sexp)
(define (read-list)
    (define tk (read-token)
    (cond [(token-rightpar? tk) emtpy]
          [(token-leftpar? tk) (cons (read-list) (read-list))]
          [else (cons (token-value tk) (read-list))])))


Helper: read-list → (listof sexp)
(define (my-read)
    (define tk (read-token))
    (if (token-leftpar? tk)
        (read-list)
        (token-value tk)))
```

6.3 Discussion About Input

What have we lost by accepting input?

- Referential Transparency: the same expression has the same value whenever it is evaluated
  e.g. (f 4) always produces the same value

  (let (z (f 4)) body)

  -every (free) z in body can be replaced by (f 4) and vice versa, equals can be suibstituted for equals?.

  -not true anymore (read) doesnt always produce the same value.

- makes it harder to reason about programs

  - simple algebraic manipulation no longer possible

7 Intro to C

-expressions, statements, blocks, functions, programs

Operation Precedence: 1+2 - infix, operators, preceduence

usual mathematical conventions

function calls 3+ g(x, y, z)

Statements:

- printf ("%d \n ", 5) function call
  value produced y expression is ignored
  expression evaluated only for its side-effects
- 1+2; legal but useless
- return 0; produces the value 0 as the result of this function (contol returns immediately to the caller)
- ; empty statement (do nothing), either state forms to come

Blocks: group of-statements treated as one
{
statement 1
statement 2
statement 3
⋮
} think as:

(begin statement 1
statement 2
statement 3
⋮
)

Functions:

C Code:

```
 int f(int x, int y) {
       printf("x = %d, y = %f \n", x, y);
       return x + y;
}
```

Racket Code:

```
 ; f: num num -> num
(define (f x y)
       (printf "x = ã, y = ã \n" x y)        (f x y))
```

think:

```
 (define (f x y)
       (printf "x = ã, y = ã \n" x y)        (f x y))
```

function call:

```
 f (4,3)
```

```
f (4,3);
```

think: `f (4,3)`
`f (4,3);`

Note: contracts (type signatures) are required and enforced

C enforces declaration before use -cannot use a function, variable, etc, until you tell c about it.

solution 1: put f first, not necessary

solution 2: put the function prototype or header first, declaration only, not definition.

include

- not part of the program, direction to the *C* processor, which runs before the compiler, the macro expansion in Racket.
- "drop the contents of the file right here"
- stdio.h
  - contains declarations for printf and other I/O fns.
  - located in a standard place

printf

- was written once, compiled once and put in a "standard place" e.g. \usr\lib
- code for printf must be combined with its code "linking"
  - a linker takes care of this
  - linkder runs automatically
  - knows to link in the code for printf
- If you write your own modules, you need to tell the linker about them

Variables:

```
int f (int x, int y) {
int z = x + y;
int w = 2;
return z/w;
}
```

Input:

```
 #include<stdio.h>
int main()
{
      char c = getchar();
      return c;
}
```
-read in a number

```
#include<stdio.h>
int getInt Helper(int acc)
{
```

```
        char c = getchar();
        if (c >= '0' && c<= '1')
                return getIntHelper(acc*10+c-'0');
        else return acc;
}

int getInt()
{
        return getIntHelper(0);
}
```

Conditional Operator: also called the ternary operator

if else is a statement

?? creates an expression

a?b?c has value b if a is true

has value c if a is false

no built-in boolean type in C

0 means false

non-zero (often 1) means true

e.g. if (o) {...} -false condition boolean type, whats in the bracket will never be run

Characters -restricted form of integers

- int - varies, but typically 32 bits, $(\tilde{4} \times 10^r)$

8 CS 146 Tutorial Jan 22

- put any 1 in front of every list in arg 2

prepend: integer $\rightarrow$ [[integer]]

9 Jan 23

9.1 ASCII code

Ones to remember:

- $'\backslash n = 10$
- '␣$= 32$
- $'0' -' 9' = 48 - 57$
- $A - Z = 65 - 90$
- $a - z = 97 - 122$

Convert a char c to its numertic value: $c -' 0'$, (c-48)

Convert a numeric digit to a ascii: $c +' 0'$

A second look at getchar: char c = getchar();

The prototype for getchar is actually not getchar().

Why int if its supposed to produce a char?

What if there are no chars (EOF)?
- if getchar is returned a char, there would be no way to indicate EOF, (every possible value returned represents a valid character)

If there are no chars, getchar produces an int that cannot possiblt be a char (i.e. not in the range 0-255)

The constant EOF denotes the value getchar produces an eof. (often EOF =-1)

getInt burns a chracter after reading an int

Does C havea function have Racket's peek-char
No, but it has ungetc, -stuffs a char back into the input stream.

Check: Jan-23.c

9.2 Basis Mutation

-for racket

- set!

```
(define x 3)
(set! x 5)
```

-hidden dependencies between different parts of the program
-harder to reason about programs

application : memorization

Caching: saving the result of compression, and repeating it

Memorization: maintaing a list or table of cached values

9.3 Feb 6

We can fix with parenthesis, grabbing the field of a structure you have a pointer to is a common enough operation.

10 Feb 12 - Wednesday

int grades [0]; -valid entries are grade[0], $\cdots$, grades[9].

What happens if you go out of bounds? -undefined behaviour.

Will it stop you? No - program may or may not crush

-if not, data may be corrupted, no way to detect.

Can give the bound implicitly.

```
int main(){
    int grades[] = {0, 0, 0, 0, 0};
    printf("%zd\ n", sizeof(grades)/sizeof(int));
}
```

- $\%z$ indicates size-t
- sizeof(grades): amount of memory grades occupies(20 bytes)
- sizeof(int): amount of memory an int occupies(4 bytes)

10.1 Function on Arrays

Sum:

```
int sum (int arr[], int size[]) {
    int res = 0;
    for (int i = 0; i < size; ++i)
        res += arr[i];
    return res;
}
```

size is not part of the type - works on an arrays of any size.

int size: tell the function what the size is

passing arrays by value $\rightarrow$copy the whole array $\rightarrow$expensive

C will not do this:

```
int main(){
    int myArray[100];
    ...
    int total = sum(myArray, 100);
    ...
}
```

Most Confusing Rule in C:

The name of the array is shorthand for a pointer to its first element.
- myArray is shorthand for &myArray[0].

As a result, sum(myArray, 100) passes a pointer, not the whole array, into the function.

But sum is expecting an array, not a pointer.

Why not int sum(int *arr, int size) then?

int *arr int arr[]

are identical in *parameter deals*.

Note: not declare both x and y as pointers, int *x, *y; True in both C and C++.

Pointer Arithmetic

Let t be a type.

t arr[10]; sizeof (arr) = 10*size(t);

arr is equivalent to &arr[0];

*arr is equivalent to arr[0];

What expression produces a pointer to arr[1]?

- arr+1 is shorthand for &arr[1].
- arr+2 is shorthand for &arr[2].
- etc

Numerically, arr+n produces the address equal to arr+n*sizeof(t).

if arr+k means &arr[k], then *(arr+k) means arr[k].

Therefore, sum is equivalent to

```
int sum (int *arr, int size){
    int res = 0;
    for (int i = 0; i < size; ++i)
        res += *(arr+i);
    return res;
}
```

In fact, a[k] is just shorthand for *(a+k). a[k] ≡ *(a+k) = *(k+1) ≡ k[a]

```
int sum (int *arr, int size){
    int res = 0;
    for (int *cur = arr; cur < arr+size; t+cur)
        res += *cur;
    return res;
}
```

& means the address.

11 Feb 13

Each function call gets a stack frame.

- Local vars pushed onto stack
- also return address - where to go when the function returns
- each invocation of the funciton gets its own version fo the local variables

When a function returns, its stack frame is popped.

- all local variables in that frame are released
- not typically erased
- "top of stack" pointer moved to top of next frame
- will be overwritten the next time a frame is pushed onto the stack

What is stack? An ADT with LIFO semantics.

LIFO: last in first out.

Can only remove the most recently inserted item.

Operations:

- Push: add an item to the stack
- Top: what is the most recently inserted item?
- Pop: remove the most recently inserted item
- Empty!: is the stack empty?

So what if you have data that must persist after a function returns?

e.g. what's wrong with struct posn makeposn(int x, int y) struct posn p; int a x, a= y; p.x = &a; p.y = &b.

12 Heap - Feb 25

What is the lifetime of heap-allocated datea?
-arbitrarily long

If heap-allocated data never goes away, program will eventually run out of memory. Even if most of the data in memory is no longer in use.

- **Racket Solution:** a run-time process that detects memory that is no longer accessible.

  Example:
  (define f x)
  (local [(define p (posn 3 4))])
  ... (x+1)))
  and automatically reclaims - **Garbage Collection**

- **C Solution:** Heap memory freed when you free it.

  Example:
  int *p = malloc(...);
  ...
  free(p); A release p's memory back to the heap

Failing to free allocated memory - called a memory leak.

Program that leak will eventually fail, if they run long enough.

Example:
int *p = malloc();
free(p);
*p = 7; // will this crash?

- Probably not: free(p) does not change p,
- p still points to that memory -storing something at that memory probably still works
  - but p is not pointing at a valid location, that location may be assigned to another pointer by another malloc call
  - called a **Dangling Pointer** BAD!!!
- Better: after free(p), assign p to point to a guarenteed-invalid location: int *p = malloc();
  free(p);
  p = NULL; // null pointer: points to nothing

NULL:

- not really part of the C language
- defined as a constant equal to 0
- could equally well say p = 0;

Dereferencing NULL:

- undefined behaviour
- program may crash

If malloc fails to allocate memory, returns null.

Consider again:

```
int *f(){
        int x = 4;
        return &x;
}

int g(){
        int y = 5;
        return &x;
}

int main(){
        int *p = f();
        g();
        printf("%d \n", *p);
}
```

p points to dead memory, by the time f returns, x is no longer a valid location. Another dangling pointer.

Program probably will not crash.

When *g* is called, it occupies f's old stack frame.

y now occcupies x's old spot, *p is not 5 (still a dangling pointer). s

Lesson: NEVER return a pointer to a local variable
- If you want to return a pointer, it should point to state, heap, or non-local stack data.

```
int *pickOne(int *x, int *y){
        return ... ?x : y; }

struct posn *getMeAPtr(){
struct Posn *p = malloc (sizeof (struct Posn));
return p;
}

        int z = 5;
int *f (){return &z;}
```

Use Heap:

1. For data that should outlive the function that creates it
   As above.

2. **For data whose size is not known at compile-time**
   `int *p = malloc(sizeof(int));` not that useful, why not just int n?

   But what if we ask for more memory? `int numSlotsNeeded;`
   `scanf("%d", & numSlotsNeeded);`
   `int *p = malloc(numSlotsNeeded *sizeof(int));`

   -can access p[0]; ... ; p[numSlotsNeeded - 1];

   -Dynamic array (heap-allocated) ... free(p)

3. For large local arrays,

   Programs typical, have more heap memory available than stack memory

```
int recursiveFunction(int n){
    int tempArray [10000];
    ...;
    return 1*recursiveFunction(n-1);
}
```

C arrays that mimic Racket vectors: can we get the behaviour of a Racket list?

(cons x y) ; produces a pair pointers

Recall: - Racket is dynamically typed, and the list items can have different types
- C is statically typed : the list items would need to have same type (if not - headaches)

-so not real need for pointers to data fields

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

struct Node *cons (int data, struct Node *next) {
struct Node *result=malloc(sizeof )
```

13 Feb 26

```
int main(){ struct Node *est = ...; struct Note *lst2= map(f,lst); }
```

struct Node *map(int *f (int), struct Node *lst) what type do we use for f? postfix before prefix, function returning a pointer.

struct Node *map(int (*f)(int),) if (!lst) return 0; return cons (f(lst->data), map(f, lst->next));}

Freeing the lst

```
int main(){
    struct Node *lst = cons(1, cons (2, oons(3, 0)));
    ...
    free(lst); }
```

WRONG:

```
int main(){
...
for (struct Node *cur = lst, cur; cur = cur -> next)
    free(cur);  cur=cur->next happends after free(cur),
```

-cur is dangling,
-need to grab next pointer before you free.

OK:

```
for (struct Node *cur = lst; cur;)
{
struct Node *tmp = cur;
cur = cur->next;
free(tmp);
}
```

or

```
void freeList(struct Node *lst)
{
    if(lst){
    freeList(lst->next);
    free(lst);
}
```

13.1 ADTs in C

Note: C doesnt have modules. C has files.

Implement ADT sequence in C:

Opertaions

- empty sequence
- insert(s, i, e) : insert int e at index i in s.
  precondition: $0 \leq i \leq size(s)$
  Note that you can put before the first one and after the last one
- size(s) - # elements in s
- remove(s, i) : remove item from index i
  precondition: $0 \leq i \leq size(s) - 1$
- index(s, i) : returns i^{th} element of s
  precondition: $0 \leq i \leq size(s) - 1$

Want : no limits on size

Implementation Options:

- Sequence: can grow as needed: linked int is easy to grow but has slow index
- Arrays: fast index but does not grows

Approach: partially filled heap array

```
struct sequence {
    int size; // how many items are in use?
    int *theArray;
    int cap; // how much can we hold
};
struct Sequence emptySeq();
int seqSize(struct Sequence s);
```

```
void Add(struct Sequence *s, int i, int e);


void remove (struct Sequence *s, int i);
int index(struct Sequence s, int i);
void freeSeq(struct Sequence *s);
```

sequence.c

#include

13.2 Feb 27

WRONG: **Sequence.h**

```
struct Sequence;
Sequence empty Seq();
```

Sequence.c

```
struct Sequence{
    int size, cap;
    int *theArray; };
```

main.c

```
#include "Sequence.h"
int main(){
    struct Sequence s;
    ...
}
```

Problem: compiler doesnt know enough about Sequence(needs size)

OK:

Sequence.h

```
typedef SeqImpl *Sequence;
Sequence empty Seq();
void add (Sequence s, int i, int e);        // (Sequence Int *x ; is a sequence)
```

main.c #include "Sequence.h"
Sequence.s; //pointer - OK! What happends when array is full? void add(Sequence s, int i, int c){
// make the array bigger
s->theArray = realloc(s->theArray, ???) // ???: new size }

realloc:

- increases a block of memory to a new size,
- if necessary, allocates a new larger block and frees the old block (data copied ever)

How big should we make it? - one bigger? (must assume each call to realloc causes a copy O(n)) If we have a sequence of adds (at the end, so no shuflling cost) # step is $n + n + 1 + n + 2 + \cdots + n + k$.

What if instead we double the size? - Each add still O(n) wprse case.

But... Amortized Analysis: places a bound on a sequence of operations.

even if an individual operation can be expensive.

If an array has a cap of k and is empty

- k inserts cast of 1 each (k steps)
- 1 insert costs $k+1$ - cap now 2k ($k+1$) steps
- $k-1$ inserts cast of 1 each ($k-1$) stpes
- 1 insert costs $2k+1$ - cap now 4k ($2k+1$) steps
- $2k-1$ inserts cost of 1 each ($2k-1$) steps
- 1 insert costs $4k+1$ - cap now 8k ($4k+1$) steps

Total # of insertions: $k+1+(k-1)+1+(2k-1)+1+\cdots+(2^{j-1}k-1)+1 = 2^j k+1$

Total # of steps: $k+k+1+k-1+2k+1+2k-1+4k+1+\cdots+2^{j-1}k-1+2^j k+1 = 3\cdot 2^j k-k+1$

So the # of steps per insertion is

$$\frac{3\cdot 2^j k-k+1}{2^j k+1} \approx \frac{3\cdot 2^j k}{2^j k} = 3$$

```
void increaseCap(Sequence s){
    if (s->size == s->cap){
        s->theArray = realloc(s->theArray, 2*s->cap * sizeof(int));
        s->cap * = 2;
}
void add(Sequence s, int i, int c) {
    increaseCap(s);        ...
}
```

Helper Function: increase Cap : main should not be calling this how dowe prevent it?

leave it out of the leader file.

But

13.3 Hash Table - March 3

Recall:

- make-map : no parameters, pre: true
- Add: parameters map M, key k, value v, Pre:true, produces no value

  if $\exists x'$s.t. $(k, v') \in M, then M < -(M \backslash \{(k, v)\}) \cup \{(k, v)\}$ else $M < -M \cup \{(k, v)\}$
- remove: parameters map M, key k, pre: true produces no value
- Post: if $\exists vs.t.(k, v) \subset M$, then $M < -M \backslash \{(k, v)\}$. Otherwise, M is unchanged.
- Search: parameters map M, key k, Pre: true

  Value produces is v s.t. $(k, v) \in M$, otherwise something outside the value domain.

Implementation:

Assume keys and integers (for simplicity - omit values)

- If we use an association list - accessing an item takes the proportional to its position in the list (P(length L) worst case)
- If we use a BST - same worset case running time, if we use a balanced BST, e.g. (AVL tree) O(log n) worse case time (n = |M|), difficult
- if we use vectors instead O(1), (time for any access, but how long should the vector be)? size of vector: maximum key? – will waste space

Hash Table

To which association list should I add (k,v)?

- need to map k to a vector index -mapping called a hash function -for simplicity use the remainder of k by the length of the vector
- for this idea to work well, the hash function f must distribute keys evenly over the indices

```
(define (make-hashtable size))
    (define index (module key (vector-length table)))
    (define hashlist (vector-ref table index))
    (define lookup (assoc key hashlist))
    (if lookup (second (lookup false)))
```

```
(define (lst-add table key val))
    (define index (modulo key (vector-length table))
    (define hashlist (vector-ref table index))
    (define lookup (assoc key hashlist))
    (if lookup
      (when (not(equal? (seeend lookup) val))
            (vector-set! table index (cons (list key val)
                                           (remove lookup hashlist))))
      (vector-set! table index (cons (cons (list key val) hashlist)))
```

If the keys aren ot numbers, need a hash function that maps them to numeric values(no details here).

13.4 Interpreting Mutation - March 3, 4

Recall: the **Deferred Substitution Interpreters** in Haskell for faux racket.

Concrete Syntax and Abstract Syntax Full racket programmer writes code like this.

```
exp = number
    | (op exp exp)
    | (fun (id) exp)
    | (with ((iod exp)) exp) | (exp exp)
    | id
```

```
data Op = Plus|Times
 opTrans Plus = (+)
 opTrans Time = (*)

data Ast = Number Integer | Bin Op Ast | Fun String Ast
    | App Ast Ast | Var String

data Val = Numb Integer | Closure String Ast Env
```

(with ((id exp1)) exp2) ≅ ((fun id exp2) exp1)

Closure : a environment that is

```
interp :: Ast-> Env-> Val
interp (Number v) _ = Numb v
interp (Fun p b) e = Closure p b e
interp (Bin op x y) = Numb(opTrans op v w)
    where
        (Numb v) = interp x e
        (Numb w) = interp y e
interp (App f x) e = interp fb ((fp,y) : fe)
    where
        Closure fp fb fe = interp f e
        y = interp x e
interp (Var x) e = fromMaybe undefined (lookup x e)
```

if lookup fails, produce undefined.

The applications are where environment gets used.

Now let's add set (for mutation) and seq (for sequencing)

```
exp = ...
    | (set id exp)
    | (set exp exp)

data Ast = ...  | Set String Ast
                | Seq Ast Ast
```

Note we will implement mutation without actually using mutation.

Implementation Set:
basic idea: change the name-value binding in the environment.
Needs to be done carefully Consider

```
(with (x 0))
    (+ (Seq (set x 5) x)
        (Seq (set x 6) x)))
```

Should produce 11.

The interpreter should not only produce a value but also an envir.

(set x 5) changes the envir, now x maps to 5, use this envir when evaluating all that follows - x, (seq (set x 6) x).
(set x 6) changes the environment, so that x maps to 6, use that environment when evaluating all that follows - x.

But what about

```
(with ((x 0))
    (+ (seq (set x 5)x)
        (seq (with ((x 10)) 0) x)))
```

Careful about returning envs, don't want 15.

Each expression should return environment that results after it's finished so that the updated envir can be used in what follows.

Can this model-threading environments in + out of expres work? Yes, if you do it right.

Boxes? -need a model that permits alising.

Idea (recall):

- Env maps variables to locations - not threaded, never updated, only added to
- Store maps locations to values, values are updated but locations are not
- Alising: two variables map to the same location
- interp: takes a store as an additional parameter

```
interp::Ast->Env->Store->(Val, Store)
type Loc = Integer          |
type Env = [(String, Loc)]  | data Val = ... | void
type Store = [(Loc, Val)]   |

interp (Number v) _ s = (Numb v, s)
interp (Fun p b) e s = (Closure p b e, s)
interp (Bin op x y) e s = (Numb (opTrans op v w), s'')
    where
        (Numb v, s') = interp x e s
        (Numb w, s'') = interp y e s'
interp (Seq x y) e s = (v s'')
    where
        (_ , s') = interp x e s
        (v, s'') = interp y e s'
interp (Var x) e s = (fromMaybe undefined (lookup x e))
    where loc = fromMaybe undefined (lookup x e)
interp (App f x) e s = interp fb ne ns
    where
        (Closure fp fb fe, s') = interp f e s
        (y, s'') = interp x e s'
        nl = newloc s''
        ne = (fp, nl) : fe
        ns = (nl, y) : s''
interp (Set x y) e s = (void, ns)
    where
        lx = fromMaybe undefined (lookup x e)
        (nv, s') = interp y e s
        ns = (lx, nv) : s'
```

newloc = length

works because we never remove a location from the store

impractical for long computations

better - reuse old locations ("garbage collection" - maybe later)

question: with? closure? 15?

14 Module 2: SIMP

14.1 March 5

Let's build our own imperative language: SIMP

- statements
- sequencing
- conditional execution
- repetition

A SIMP program is a sequence of var deals, initialized to integers, followed by statements e.g.

```
(var [(x 0) (y 1) (z 2)]
    (print y))
```

Grammar:

```
program = (var [(id number) ...]
            start ...)
```

```
stmt = (print aexp)
    | (print string)
    | (set id aexp)
    | (seq stmt ...)
    | (iif bexp stmt stmt)
    | (skip)
    | (while bexp stmt)

aexp = (+|x|-|div|mod aexp aexp)
    | number
    | id

bexp = (=|>|<|>=|<= aexp aexp)
    | (not bexp)
    | (and|or bexp ...)
    | true | false
```

Easy implementation: Racket macros

```
(define-syntax-rule
    (var [(id init) ...] exp ...)
    (let [(id init) ...] exp ...))

(define-syntax-rule
    (while (est exp ...)
    (let loop ()
        (when test
            exp ... (loop))))       ;; known as named let
```

Named Let: sets up loop as a function with () args (no args) with body (when ...) + invokes (loop)

The test are renames of existing Racket Functions.

```
(provide vars iif while
    (rename-out [display print] [begin seq] [set! set]
        [void skip] [quotient div] [modulo mod]))
        > >= < <= and or not + - * true false
        #%module begin #% datum #%app #%top #%top-interaction
```

Example: compute all perfect numbers up to 10000. In SIMP

```
(var [(i 1) (j 0) (acc 0)]
    (while (<= i 10000)
        (set i 1) (set acc 0)
        (while (k j i)
            (iif (= (mod i j) 0)
                (set acc (+ acc j))
                (skip))
            (set j (+ j 1)))
            (iif (= acc i)
                (seq (print i) (print "\n"))
                (skip))
            (set i (+ i 1))))
```

In C:

```
int main(){
    for (int i = 1; i<=10000; ++i)
    {
        if (i%j == 0) acc += j;
    }
    if (acc == i) printf("%d\n", i);
}
```

Semantics of SIMP maps from states to states

state σ :values of vars(ignore output) -map from vars to integer values

notation: $[x \mapsto i]\sigma$ is the state that maps x to i and nay other y to $\sigma|y$.

Rewrite rules for pairs (π, σ) (program, state)
(omitting rules that leave σ unchanged, e.g. expr, skip, (seq))

((set x n), σ) $\Rightarrow$ $(, (x \mapsto n)\sigma)$
((iif true s1 s2), σ) $\Rightarrow$ $(s1, \sigma)$
((iif false s1 s2), σ) $\Rightarrow$ $(s2, \sigma)$
((while t s1 ... si), σ) $\Rightarrow$ ((iif t (seq s1 ... si (while t s1 si)) (skip)), σ)

Initial pair (π_0, σ_0)
If the program is (vars [(x1 n1) ... (xi ni)] s1 ... sj)

Then $\pi_0 = (seqs1...sj)$ and σ_0 is the function σ s.t. for $k = 1, ..., i$, $\sigma(x_k) = n_k$.

14.1.1 Intepreter (Haskell)

```
data AOp = Plus|Times|Minus|Mod|Div deriving Show  -- make this displayable

aopTrans Plus = (+)
aopTrans Times = (*)
aopTrans Minus = (-)
aopTrans Div = div
aopTrans Mod = mod

data Aexp = Number Integer | Var String | ABin Aop Aexp Aexp
    deriving Show
data Bop = Lt|Gt|Le|Ge|Eq deriving Show
bopTrans Lt = (<)
bopTrans Gt = (>)
bopTrans le = (<=)
bopTrans Ge = (>=)
```

```
bopTrans Eq = (==)

data Bexp = BBin BOp Aexp Aecp | Not Bexp | And Bexp Bexp
            | Or Bexp Bexp | BVal Bool  deriving Show
data Stmt = Skip | Set String Aexp | Iif Bexp Stmt Stmt
            | Seq Stmt Stmt | While Bexp Stmt deriving Show
        -- (omit print for now)
```

```
import qualified Data.Map as M
type state = M.Map String Integer

aeval :: Aexp->State->Integer
aeval (Number n) _ = n
aeval (var x) s = M.findWithDefault undefined x s
aeval (ABin aop ae1 ae2) s = aopTrans aop (aeval ae1 s)
                                          (aeval ae2 s)
beval :: Bexp->State->Bool
bevall (Bval b) _ = b
beval (Not b) _ = not (beval be st)
beval (And be1 be2) st = (beval be1 st) && (beval be2 st)
beval (Or be1 be2) st = (beval be1 st) && (beval be2 st)
beval (BBin op ae1 ae2) st = bopTrans op (aeval ae1 st) (aeval ae2 st)

interp :: Stmt ->stat -> State
interp Skip  st - st
interp (set x ae) st Mainsert x $!(aeval ae st) st
```

14.2 March 10

Recall:

```
interp::stmt->state->state
interp Skip st = st
interp (set x ae) st = (Mapinsert x $! (aeval ae st)) st
interp (Iif be ts fs) st =
    if (beval be st) then interp ts st
    else interp fs st
interp (Seq s1 s2) st = let st' = interp s1 st in interp s2 st'
interp loop @ (While bt body) st = interp (Iif bt (Seq body loop) Skip) st
```

14.2.1 Add Printing

Recall from Module 1,
-can model output as a list of chars that would be printed (ω as part of the state)

```
interp :: Stmt -> (State, String) -> (State, String)
          Stmt -> State -> String -> (State, String)
```

Add print statements to the AST:

```
data Stmt = ...
          | IPrint Aexp
          | SPrint String

interp Skip st om = (st, om)
interp (Set x ae) st om = (M.insert x $! (aeval ae st) st, om)
interp (Iif be ts fs) st on = if (beval be st) then interp ts st om
                              else interp fs st om
interp loop@(While bt body) st om =
    interp (Iif bt (Seq body loop) Skip) st on
interp (Seq s1 s2) st om =
    let (st', om') = interp s1 st om
    in interp s2 st' om'
```

```
interp (IPrint ae) st om = (st, om++(show (aeval ae st)))
interp (SPrint ae) st om = (st, om ++ s)
```

```
(vars [(x 10) (y 1)]
    (while (> x 0)
        (set y (* 2 y))
        (set x (- x 1))
        (print y)
        (print "\n")))
```

14.2.2 String Transformer

```
interp :: stmt -> State -> String -> (State, String) // string transformer
// factor this out of the type
type Mio a = String ->(a, String)
interp :: Stmt -> State -> Mio State
```

Two Helper Functions:

```
inject: a -> Mio a
inject x = \om -> (x, om)
miprint :: String -> Mio()
miprint s = lom -> ((), om ++ s)
```

Abstract the behaviour of Seq:

```
interp (Seq s1 s2) st om =
    let (st', om') = interp s1 st om in interp s2 st' om'
= (\f -> let (st', om') = f om in interp s2 st' om') (interp s1 st)
// use lambda rules
= (\f->\g->let (st', om') = f on in g st' om') (interp s1 st) (interp s2)
= (\f->\g->lom->let (st' om')=f om in g st' o')(interp s1 st)(interp s2) om
```

Now operator "bind"

```
(>>>=) :: Mio a -> (a->Mio b) -> Mio b
f>>>= g = \om->let(x, om') = f om in g x om'
```

Simplification: for when g doesn't need the value of x

```
(>>>) :: mio a-> Mio b->Mio b
f >>> g = f >>> = \_-> g
interp :: Stmt->State->Mio State
// interp Skip St = \om ->(st, om)
interp Skip St = inject st
interp (set x ae) st = inject (M.insert x $! (aeval ae st) st)
interp (Iif be ts fs) = if (aeval ae st) then interp ts st
                        else interp fa st
interp (Seq s1 s2) st om = (>>>=) (interp s1 st)(interp s2) om
                         = ((interp s1 st) >>>= (interp s2) om)
interp (Seq s1 s2) st = (interp s1 st).>>>= (interp s2)
```

```
interp loop@ (While bt body) st = interp (Iif bt (SEq body lopp) skip) st
interp (IPrint ae) st = miprint (show (aevale ae st)) >>> (inject st)
interp (Sprint s) st = (miprint s) >>> (inject s t)
```

14.3 March 11

Native Haskell Approach:

```
Mio a    ->  IO a
miprint  ->  putStr
>>>=     ->  >>=
>>>      ->  >>
inject   ->  return
```

Cleaner Haskell: "do" notation

```
e1 >> e2     ->  do e1, e2
e1 >>= \p->e2    ->do p<-e1;e2
```

Example:

```
interp (Seq s1 s2) st = do st' <- interp s1 st; interp s2 st'
interp (IPrint ae) st = do putStr (Show (aeval ae st)); return st
interp (SPrint s) st = do putStr; return st
```

IO

- example of a **monad**
  -Any data type with »=, return operators
- Needs help to model otherwise impure effects purely
  - abstract away the "plumbing" that ensures effects are properly sequenced

Proofs for Imperative Programs

Recall: Fibonacci Numbers.

In racket:

So prove $\forall j, \forall i > 1$, if f_{jp1} is F_{j+1} and f_j is F_j then $(fib - n\ i\ fjp1\ fj) = F_{i+j}$.

What would this look like in SIMP?
-set n initially + mutate (n functions)

```
(vars [(n 10) (fj 1) (fnj1 0) (ans 0)]
    (iif (= n 0)
         (set ans fjn1)
         (seq (while (> n 1)
                (set fjn1 fj)
                (set fj (+ fj fjm1))
                (set n (- n 1)))
              (set ans fj)))
    (print ans))
```

It's wrong because $fj1$ updated prematurely.

```
(vars [(n 10) (fj 1) (fjn1 0) (1 0) (ans 0)]
    (iif (= n 0)
         (set ans final)
         (seq (while (> n 1)
                (set t fj)
                (set fj (+ fj fjm1))
                (set fjm1 t)
                (set n (- n 1))
                (set ans fj)))
                (print ans)))
```

14.4 March 12

Can we prove that for given n, this program prints F_n? Equivalently, hence prove that the final value of ans is F_n?

The statement answer F_n true or false?
-depends on the sate at the time the statement is evaluated.

14.4.1 Hoare Logic

Prove triples of the form $\{P\}$ statement $\{Q\}$ "Hoare Triples",

- P : precondition - logical statement about the state before statement runs
- Q : postcondition - logical statement about the state after the statement runs
- Interpretation: If P is true before statement runs, then Q is true after statement runs.

To conclude:

{P} (vars [(x1 v1)...(xn vn)] stmt ...) {Q}

it suffices to show:

{ P and x1 = v1 and ... and xn = vn } (seq stmt ...) {Q}

It suffices to find a stmt R such that {P} s1 {R}, {R} s2 {Q},.

Finding R can be tricky - may have to adjust P + Q to get an R that works.

So to conclude {P} stmt {Q}, we can prove {P'} stmt {Q'} where $P \Rightarrow P'$, $Q' \Rightarrow Q$.

- **(set x exp)**
  To conclude {P}(set x exp) {Q},

  P+Q should be almost the same, only the value of x has changed. Q can say nothing about the old value of x, whatever Q says about x must have been true about exp before the stmt.

  so {Q [exp/x]}(set x exp) {Q} {atb = y}(set x (+ a b)){x = y}

- **(Iif B stmt1 stmt2)**
  To conclude {P}(iif B stmt1 stmt2) {Q},

  suffices to show

  {P and B} stmt1 {Q}
  {P and not B}stmt2{Q}

(while B stmt) - trickiest
If the loop doesn't run

Now prove that the fib program works.

15 Module 3 - PRIMP

so far - abstract syntax stored in S-expression, basically tree-structured data.

But trees are an abstraction - basic machine has only memory not trees.

Let us now store our program in a vector, a program is a linear sequence of instructions, the RAM used for data will be the same RAM that holds the program.

Reasoning about the program - no more rewriting -program remains fixed

so we need to keep track of where we are in the program

Index into the program -called the program counter

holds the location of the next statement (instruction to be exceeded)

Simulator runs a fetch-execute cycle

repeat: -fetch the instr at location given by PC -increment PC -execute the fetched instr

For clarity - PC is a seperate variable, outside of the RAM that holds our program