Algorithm Design and Data Abstraction

Brad Lushman

Preface

Disclaimer Much of the information on this set of notes is transcribed directly/indirectly from the lectures of CS 146 during Winter 2021 as well as other related resources. I do not make any warranties about the completeness, reliability and accuracy of this set of notes. Use at your own risk.

I have missed the tutorials held by TA/ISA's which established the basics of Haskell interpreter. Then in Section 14.2, the instructor continued the discussion on interpreter... I will add these Haskell intro parts if I have chance... Check http://learnyouahaskell.com.

Note that online version of this course tends to go much faster than in-person lectures...

For any questions, send me an email via https://notes.sibeliusp.com/contact.

You can find my notes for other courses on https://notes.sibeliusp.com/.

Sibelius Peng

Contents

Module I:

Impurity in Racket

Full Racket permits several expressions to appear, instead of just one in the teaching languages. They are all evaluated, but only the value of the last one is used. The others are evaluated for their side effects, and any value they produce is discarded.

Jan 12

1.1 Major themes

Major theme of CS 146

- side-effect ("impurity")

- programs that *do* things

- imperative programming

General outline

- impure Racket

- C

- low-level machine

Why functional programming first? Why not imperative first?

Imperative programming is harder. Side-effects are not easy things to deal with. For example, text is printed to the screen, keystrokes extracted from the keyboard, values of variables change. All these things change the state of the world. Also, the state of the world affects the program.

If we write a racket program like this one,

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

That depends on the value of y. However, if the value of y can change because of the side effects, we have to add a word: it depends on *current value* of y.

Thus the semantics of an imperative program must take into account the current state of the world, even while changing the state of the world.

So there is then a temporal component inherent in analysis of imperative programs. It is not "what does this do?", but "what does this do at this point in time?"

Why study imperative programming at all? It seems it doesn't worth it. "The world is imperative". For example, machines work by mutating memory. Even functional programs are eventually executed imperatively.

. . . "or is it?" Is the world constantly mutating, or is is constantly being reinvented? When a character

appears on the screen, does that change the world or create a new one?

Either way, imperative programming matches up with real-world experience, but a functional world view may offer a unique take on side-effects.

1.2 Recursion

Recall from CS 145:

Structural recursion: the structure of the program matches the structure of data.

For example, natural numbers.

```
(define (fact n)                ; A Nat is either
    (if (= n 0) 1               ; 0 or
        (* n (fact (- n 1)))))) ; (+ 1 n) where n is a Nat
```

The cases in the function match the cases in the data definition. The recursive call uses arguments that either stay the same or get one step closer to the base of the data type.

Here is another example on the length of the list.

```
(define (length l)                     ; A (list of X) is empty
    (cond [(empty? l) 0]               ; or (cons x y) where x
          [else (+ 1 (length (rest l)))])) ; is an X and y is a (list of X)
```

If the recursion is structural, the structure of the program matches the structure of its correctness by induction.

Claim (length L) produces the length of the list L.

> Proof:
> Structural induction on L.
>
> **Case 1** L is empty. Then (length L) produces 0, which is the length of the empty list.
>
> **Case 2** L is (cons x L'). Assume that (length L') produces n, which is the length of L'. Then (length L) produces (+ 1 n), which is the length of (cons x L'). □

Correctness proof just looks like a restatement of the program itself.

Accumulative recursion one ore more extra parameters that "grow" while the other parameters "shrink".

For example,

```
(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 0))
```

Proof by induction on an invariant. For example, to prove that (sum-list L) sums L, suffices to prove (sum-list-help L 0) produces the sum of L. Let's try to prove by structural induction on L.

Case 1 L is empty. Then (sum-list-help L 0) is (sum-list-help empty 0) which gives 0.

Case 2 L = (cons x L'). Assume (sum-list-help L' 0) $\Rightarrow$ the sum of L'. Then (sum-list-help L 0) is (sum-list-help (cons x L') 0) which reduces to (sum-list-help L' (+ x 0)) which is then equal to (sum-list-help L' x). Then we are in trouble, because this does not match inductive hypothesis. Proof fails.

So we need a stronger statement about the relationship between L + acc that holds throughout the recursion - an invariant.

> Proof:
> We prove the invariant $\forall$ L, $\forall$ acc (sum-list-help L acc) produces acc + (sum-list L) by structural induction on L.
>
> **Case 1** L is empty. Then (sum-list-help L acc) is (sum-list-help empty acc) which gives acc, which is equal to the sum of the list + acc.
>
> **Case 2** L is (cons x L'). Assume (sum-list-help L' acc) produces the sum of L' + acc. Then (sum-list-help L acc) = (sum-list-help (cons x L') acc) $\rightsquigarrow$ (sum-list-help L' (+ x acc)) which is equal to (sum-list L') + (x + acc) = (+ (sum-list L') x) + acc = (sum-list L) + acc
>
> Then let acc = 0: (sum-list-help L 0) = (sum-list L). □

General recursion: does not follow the structure of the data. Proofs require more creativity.

How do we reason about imperative programs?

1.3 Impure Racket

```
1 (begin exp_1 ... exp_n)
```

evaluates all of exp_1, ..., exp_n in left-to-right order and produces the value of exp_n. This is useless in a pure functional setting, but it is useful if exp_1, ..., exp_(n-1) are evaluated for their side-effects.

There is an implicit begin in the bodies of functions, lambdas, local, answers of cond/match. For example,

```
1 (define (f x)
2     ... ; side-effect 1
3     ... ; side-effect 2
4     ... ; side-effect 3
5     ans
6 )
```

Reasoning about side-effects: for pure functional programming, we have the substitution model, so-called "stepping rules". Can the substitution model be adapted? we can have the "state of the world" an extra input & extra output at each step. So each reduction step transforms the program & also the "state of the world".

How do we model the "state of the world"? For the simple case, it is just a list of definitions. For more complex cases, we need some kind of memory model (RAM) (won't use yet).

<div style="text-align: right;">

2

</div>

Jan 14

2.1 RAM

For now: conceptualization of a RAM (random access machine). Memory is a sequence of "boxes", which are indexed by natural numbers ("addresses"). It contains a fixed size number (say 8 bits or 32 bits). Any box's contains can be fetched $O(1)$ time.

For example, 32-bit RAM:

Address	Contents
4	1100111010111...
8	0011101011101...
12	
$\cdots$	
256	
$\cdots$	

$\longleftarrow$ 32 bits $\longrightarrow$

(binary digits)
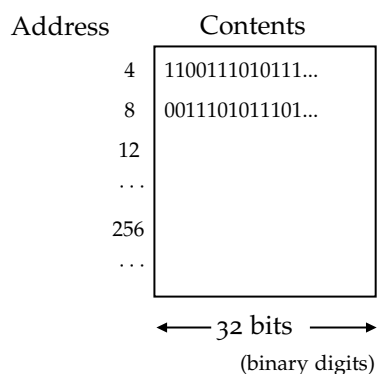
Will use in a later module, but keep it in mind.

2.2 Modelling Output

It is the simplest kind of side-effect. The "state of the world" here is the sequence of characters that have been printed to the screen. So each step of computation potentially adds characters to this sequence.

> **Note:**
> Every string is just a sequence of characters. Indeed, there is a racket function:
>
> ```
> (string->list "abcd") ===> (list #\a #\b #\c #\d)
> ```

Substitution model $\pi_0 \Rightarrow \pi_1 \Rightarrow \pi_2 \Rightarrow \cdots \Rightarrow \pi_n$ where each π_i is a version of the program obtained by applying one reduction step to π_{i-1}.

In addition to this sequence of programs, now also: $\omega_0 \Rightarrow \omega_1 \Rightarrow \omega_1 \Rightarrow \cdots \Rightarrow \omega_n$ where each ω_i is a

version of the output sequence. Because the sequence of characters can only grow, each ω_i is a *prefix* of ω_{i+1} (can't "unprint" characters).

Therefore, we have a combined version: $(\pi_0, \omega_0) \Rightarrow (\pi_1, \omega_1) \Rightarrow \cdots \Rightarrow (\pi_n, \omega_n)$.

Some program reductions will create definitions, (e.g., `local`), and these defined values will eventually change. So let's separate out the sequence of definitions δ.

So we got a triple now: $(\pi_0, \delta_0, \omega_0) \Rightarrow (\pi_1, \delta_1, \omega_1) \Rightarrow \cdots \Rightarrow (\pi_n, \delta_n, \omega_n)$ where δ_0, ω_0, representing the beginning of the program, are empty.

If $\pi_0 = $ (define id exp) ..., then we reduce exp according to the usual CS 145 (& new CS 146) rules. This may cause characters to be sent to ω. Now exp is reduced to val. Then remove (define id val) from π and add to δ.

If $\pi_0 = $ exp ..., then we reduce exp by the usual rules, which may cause characters to be sent to ω. Now exp is reduced to val which is removed from π. So the characters that make up val added to ω.

When π is empty, then we are done. So δ, ω is the *state*, that which changes, other than the program itself. ω here is relatively harmless because changes to ω don't affect the running of the program. What about δ? δ is not a problem yet, because variables are not yet changing. All we are doing now is adding new definitions, which is not really a change of state.

How can we affect ω? In Racket, we can do

- (display x) which outputs the value of x with no line break

- (newline) gives the line break.

- (printf "The answer is ~a.\n" x) which is formatted print. The value of x replace ~a. And \n is the new line character. As a Racket character on its own: #\newline.

In Racket,

```
1  > (display "Hello")
2  Hello
3  > "Hello"
4  "Hello"
5  > (begin (display "Hello") 5)
6  Hello5
7  > (define x (begin (display "Hello") 5))
8  Hello
9  > x
10 5
```

But then, what do display, newline, printf return? It looks that they don't return anything. We can try following:

```
1  > (define y (display "Hello"))
2  Hello
3  > y            ; y does have a value
4  > (list y)  ; by a trick
5  '(#<void>)
```

They return special value #<void> which is not displayed in DrRacket. Basically, for functions, that essentially return nothing, and also the result of evaluating (void). Functions that return void are called *statements* or *commands* and that's where imperative programming gets its name.

Recall: a Racket function map. (map f (list l1 ... ln)) produces (list (f l1) ... (f ln)). It's reasonable to ask what if f is a statement? The idea: it is needed for side-effects and produces #<void>. Then (map f (list l1 ... ln)) produces (list #<void> ... #<void>) which is not useful.

Instead, now consider for-each: (for-each f (list l1 l2 ... ln)) *performs* (f l1), (f l2)... (f ln) and *produces* #<void>. For example, we can use it as follows:

```
1  (define (print-with-spaces lst)
2      (for-each (lambda (x) (printf "~a    " x)) lst))
```

This will print out each item in the list with spaces in between and will produce void at the end rather than a list of void's. Let's write for-each:

```
1  (define (for-each f lst)
2      (cond [(empty? lst) (void)]
3            [else (f (first lst)) ; implicit begin
4                  (for-each f (rest lst))]))
```

or using if:

```
1  (define (for-each f lst)
2      (if (empty? lst)
3          (void)
4          (begin (f (first lst)) (for-each f (rest lst)))))
```

Doing nothing in one case of an if condition is common enough that there is a specialized form:

```
1  (define (for-each f lst)
2      (unless (empty? lst) (f (first lst)) (for-each f (rest lst)))) ; implicit begin
```

It evaluates body expressions if the test is false. Similarly, (when ...) evaluates body expressions if test is true.

Before we had output, the order of operations didn't matter (assuming no crashes/non-terminations), but now, the order of evaluation may affect the order of output. Also, before we had output, all non-terminating programs could be considered equivalent (not meaningful), but now non-terminating programs can do interesting things (e.g., print the digits of π).

Semantic model should include the possibility of non-terminating programs. What will be the meaning of the non-terminating programs be? It is what the program would produce "in the limit". Here we let Ω to denote the set of possible values of ω, which would include finite & infinite sequences of characters.

But why do we need output? We never used it in CS 145, and Racket has a REPL (Read-Eval-Print-Loop). We can just call functions and see the result. That's what Racket has, but many languages don't have this. Instead, they have compile/link/execute cycle. Under this cycle, the program is translated (by a *compiler*) to a native machine code and then executed from the command line. Then we will only see output if the program prints it. Below is an example of C program.

```
1  #include <stdio.h>
2  int main (void) {
3      printf("Hello, world!\n");
4      return 0;
5  }
```

Here we have to ask for it if we want something to show up in the screen (line 3).

What about Racket? Here is a use in Racket: tracing program.

```
1  (define (fact n)
2      (printf "fact applied to argument ~a\n" n) ; implicit begin
3      (if (= n 0) 1 (* n (fact (- n 1))))))
```

This can aid debugging.

2.3   Modelling input

Let's now talk about the input. We can imagine an infinite sequence consisting of all characters the user will ever press ι. So the model now is $(\pi, \delta, \omega, \iota)$. Every time we need to accept an input character, is the same as removing a character ι.

Here is a small problem: the sequence may *depend* on the output, so the users decide what to input *in response to* what is displayed on the screen. So a more realistic model of input would perhaps not assume all input is available at one.

The alternative: a request for input yields a function consuming one or more characters and producing the next program π, with the user's characters substituted for the read request. For example, a function (read-line), might be modeled as λ (line) line. So if user types "abc", as a result of this, we get "abc". Then the entire program reduces to a big "nesting" of input request functions, basically, one function per "prompt". If we supply user input for each prompt, it yields the final result.

Proof techniques for imperative programs will come much later.

2.4   Input in Racket

(read-line) produces a string consisting of all characters pressed until the first newline and the string we get does *not* contain the newline.

```
1  (read-line) ; pops up a little box and lets us to type
2  Test.
3  "Test."      ; and get back the string as the result.
4  > (string->list (read-line)) ; if we type Test.
5  (list #\T #\e #\s #\t #\.)
```

To read a list of lines, the question then is how do we know when to stop reading? If we look carefully at the box popped up by (read-line), at the end of the box, there is a yellow button, which says "eof" (end of file). When we press that button, it also ends the search for input. "eof" means there is no more input.

```
1  (define (read-input)
2      (define nl (read-line)) ; nl stands for next line
3      (cond [(eof-object? nl) empty]
4            [else (cons nl (read-input))]))
```

Note that this implementation of (read-input) is not tail-recursive.

A more primitive form of input would be (read-char) which extracts one character from the input sequence.

3

Jan 19

3.1 More primitive input reading

read-char reads one char from the input sequence. Here is a quick demo.

```
1  > (read-char)
2  abcde ; type in the box and press enter
3  #\a
4  > (read-char)
5  #\b
6  > (read-char)
7  #\c
8  > (read-char)
9  #\d
10 > (read-char)
11 #\e
12 > (read-char)
13 #\newline
14 > (read-char) ; now ask for new inputs
```

peek-char examines the next char in the sequence, without removing it from the sequence. It does read the character, but does not take that from io, or the input stream.

```
1  (define (my-read-line)
2      (define (mrl-h acc)
3          (define ch (read-char))
4          (cond [(or (eof-object? ch) (char=? ch #\newline)) (list->string (reverse
   acc))]
5                [else (mrl-h (cons ch acc))]))
6      (mrl-h empty))
7
8  ; call it by
9  (my-read-line)
```

Less primitive input: read consumes from input (and produces) an S-expression (no matter how many chars or lines it occupies)

```
1  > (read) ; type abc
```

```
2   'abc      ; symbol
3   > (read)
4   (a b c    ; not closed
5   de f ghi  ; racket not satisfied
6   )         ; bracket closed
7   '(a b c de f ghi)
8   > (read)
9   (a b (c d e (f)) g)
10  '(a b (c d e (f)) g)
```

3.2   Writing DrRacket

The next example is that we write DrRacket: Implementing a Racket REPL

```
1   (define (repl)
2       (define exp (read))
3       (cond [(eof-object? exp) (void)]
4             [else (display (interp (parse exp)))
5                   (newline)
6                   (repl)]))
7   (repl)
```

`parse` figures out what that S-expression means: function/if... `interp` is do it.

Let's write our own version of `read`. Process typically happens in two steps. The first step is **Tokenization**, which converts sequence of raw characters to a sequence of *tokens* (meaningful "words"). For example, left paren, right paren, id, number... Typically, id's start with a letter, nums start with a digit. Because of that, here is a key observation: peeking at the next character tells us what kind of token we will be getting, and what to look for to complete the token. So this is asking us to build the structure: (struct token (type value)) where type is the kind of token: 'lp, 'rp, 'id, 'num; and value is the "value" of the token (numeric value, name, etc).

We gonna make a couple of helpers first:

```
1   (define (token-leftpar? x) (symbol=? (token-type x) 'lp))
2   (define (token-rightpar? x) (symbol=? (token-type x) 'rp))
```

```
1   ; read-id: -> (listof char)
2   (define (read-id)
3       (define nc (peek-char))
4       (if (or (char-alphabetic? nc) (char-numeric? nc))
5           (cons (read-char) (read-id))
6           empty))
```

```
1   ; read-number: -> (listof char)
2   (define (read-number)
3       (define nc (peek-char))
4       (if (char-numeric? nc)
5           (cons (read-char) (read-number))
6           empty))
```

Here is our main tokenizer:

```
1  ; read-token: -> token
2  (define (read-token)
3      (define fc (read-char))
4      (cond
5          [(char-whitespace? fc) (read-token)]
6          [(char=? fc #\() (token 'lp fc)]
7          [(char=? fc #\)) (token 'rp fc)]
8          [(char-alphabetic? fc) (token 'id (list->symbol (cons fc (read-id))))]
9          [(char-numeric? fc) (token 'id (list->number (cons fc (read-number))))]
10         [else (error "lexical error")]))
```

Note that `list->symbol`, `list->number` don't exist, but it's easy to build them.

Step 2 is **parsing**: are the tokens arranged into a sequence that has the structure of an s-exp? if so, then produce the s-exp. Let's first make a helper.

```
1  ; read-list: -> (listof s-exp)
2  (define (read-list) ; assumes left-par has already been read
3      (define tk (read-token))
4      (cond
5          [(token-rightpar? tk) empty]
6          [(token-leftpar? tk) (cons (read-list) (read-list))]
7          [else (cons (token-value tk) (read-list))]))
```

All left is to build `read`:

```
1  ; my-read: -> s-exp
2  (define (my-read)
3      (define tk (read-token))
4      (if (token-leftpar? tk) (read-list) (token-value tk)))
```

There are some good exercises:

- expand the set of token types, e.g., strings.

- handle other kinds of brackets, [ ], { } which have to match.

What have we lost by accepting input? We lost *referential transparency*: the same expression has the same value whenever it is evaluated. For example, (f t) always produces the same value. If we do (let ((z (f 4))) body), then every (free) z in body can be replaced by (f 4) and vice versa. "equal can be substituted for equals". It is not true anymore! because (read) doesn't produce the same value. That makes it harder to reason about programs, where simple algebraic manipulation is no longer possible.

3.3   Intro to C

C is built from expressions, statements, blocks, functions, program.

3.3.1   Expressions

Example of expressions: 1 + 2 uses infix operators. There is a notion of precedence in C unlike racket. Also a function call, f(7), the name comes first. printf("%d\n", 5) is also a function call.

Operator precedence follows usual mathematical conventions. For example, 1 + x * y, multiplication is done first. If we want plus to do first, then we do (1 + x) * y.

We can take function call in a larger expression: `3 + f(x, y, z)`. `printf("%d\n", 5)` is a function call, and C substitutes 5 in place of `%d`, which means display as a decimal number. It's natural to ask, what does `printf` produce? It produces the number of characters printed.

3.3.2 Statements

The easiest way to make a statement (command) is to take an expression and put a semicolon at the end. For example, `printf("%d\n", x);`. Here the value produced by the expression is ignored, so expression is used only for its side-effects. Thus we could do `1 + 2;`, which is legal, but useless. Also, in previous lectures, we have seen `return 0;`, which produces the value 0 as the result of this function and control returns immediately to the caller. `;` is an empty statement, which does nothing. Other statement forms to come.

3.3.3 Blocks

Block is a group of statements treated as one statement.

```
{
    stmt 1
    stmt 2
    ...
    stmt n
}
```

We can think this, sort of,

```
(begin stmt 1 ... stmt n)
```

Note that the difference `begin` has a value, which is the value of `stmt n`, and this is not the case in C. Thus this is not a perfect analogy. A better analogy is that we replace `begin` by `void`, then they will get evaluated but the entire thing has `void` value.

3.3.4 Functions

Here is a function.

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

In racket, this would be roughly equivalent to

```
; f: Num Num -> Num
(define (f x y)
    (printf "x = ~a, y = ~a\n" x y)
    (+ x y))
```

Function call: `f(4, 3)` is an expression, produces 7. `f(4, 3);` is a statement. Thus in racket, it can be viewed as, `(f 4 3)` and `(void (f 4 3))`.

Note that contracts (type signatures) are required and enforced.

3.3.5 Programs

Program itself is a sequence of functions. The starting point is the special function, known as `main`, and it looks like this

```
1   int main() { // int main(void) {
2       ...
3       ...
4   }
```

For example,

```
1   int main() {
2       f(4, 3);
3       return 0;
4   }
5   // and we got our f defined before
6   int f(int x, int y) {
7       printf("x = %d, y = %d\n", x, y);
8       return x + y;
9   }
```

If we give it to compiler, it won't compile. Why?

4

Jan 21

4.1   Compile C programs

Recall from last lecture:

```c
int main() {
    f(4, 3);
    return 0;
}
// and we got our f defined before
int f(int x, int y) {
    printf("x = %d, y = %d\n", x, y);
    return x + y;
}
```

won't compile. A C program compiled: there is a program called the compiler that translates the program into the binary which is the only language the computer actually speaks and the computer execute this binary code directly: not through "DrC" like in DrRacket, the program runs natively on the machine on its own.

The way to compile: `gcc myfile.c -Wall -o myfile`. Here `-o myfile` is what we want the output program to be called, name of the output. If we don't do this, the default is `a.out`. `-Wall` stands for "Warn all". To run it, `./myfile` where `.` means the current directory. Without specifying the current directory, it won't know where to find the program to run.

Now back to our problem. The compiler will complain: `main` doesn't know what `f` is. C enforces the rule: declaration-before-use: can't use a function/variable/etc... until we tell C about it. C has this rule because C is old, and it uses one-pass compiler.

Solution 1   Put `f` first.

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

int main() {
    f(4, 3);
```

```
8      return 0;
9  }
```

Ok, but... this doesn't always work. We may want a different order just for the aesthetic of the program. Moreover, reordering the programs does more than C asks.

4.2   Declaration vs. Definitions

```
1  int f(int x, int y) {
2      // ...
3  }
```

is both *declaration* (tells C the function exists) and *definition* (completely constructs the function).

C only requires *declaration* before use. So what we can do instead is

```
1  int f(int x, int y); // - function prototype or header
2                       // - declaration only.
3  int main() {
4      f(4, 3);
5      return 0;
6  }
7
8  int f(int x, int y) {                  // this is the function definition
9      printf("x = %d, y = %d\n", x, y);  // also solves the mutual recursion problem
10     return x + y;
11 }
```

However, this still doesn't compile. What is `printf`? no declaration for `printf`. If we knew what it was, in theory we could do

```
1  int printf(---???---);
2  int f(int x, int y);
3
4  int main() {
5      f(4, 3);
6      return 0;
7  }
8
9  int f(int x, int y) {
10     // ...
11 }
```

Rather than declare every standard library function header before we use it, C provides "header files". So we write

```
1  #include <stdio.h>
2
3  int f(int x, int y);
4  int main() {
5      f(4, 3);
6      return 0;
7  }
```

```
8   int f(int x, int y) {
9       // ...
10  }
```

`#include` is not part of the C language. Rather it is a directive to the C preprocessor (which runs before the compiler). It's sort of like macro expansion in Racket. `#include <file.h>` means "drop the contents of `file.h` right here". `stdio.h` contains declarations for `printf`/other IO (input output) functions, and it is located in a "standard place". For example, `/usr/include` directory.

Now until this point, the compiler is satisfied. However, still technically incomplete: where is the code that implements `printf`? `printf` was written once, compiled once, and put in a "standard place", for example, `/usr/lib`.

Code for `printf` must be combined with our code. This step is known as "linking", which is done by a linker, and linker runs automatically. It "knows" to link the code for `printf`. If we write our own modules, then we need to tell the linker about them (later).

Let's go back to `main`: we have the returned value 0. To whom am I returning the zero? The operating system. We can type `echo $?` to check the returned value. Typically, 0 usually means OK. Anything > 0 is some kind of error.

Only in the case of `main`, `return` maybe left out, and in that case, 0 is assumed.

4.3 Variables and input in C

Let's talk about variables.

```
1   int f(int x, int y) {
2       int z = x + y;
3       int w = 2;
4       return z / w;
5   }
```

Input:

```
1   #include <stdio.h>
2   int main() {
3       char c = getchar();
4       return c;
5   }
```

Let's try to read in a number.

```
1   #include <stdio.h>
2   // like before, we don't care about the negatives at this point.
3   int getIntHelper(int acc) {
4       char c = getchar();
5       if (c >= '0' && c <= '9') return getIntHelper(acc * 10 + c - '0');
6       else return acc; // "else" keyword is technically not needed here.
7   }
8
9   // An alternative way: ternary operator
10  int getIntHelper(int acc) {
11      char c = getchar();
```

```
12      return (c >= '0' && c <= '9') ? getIntHelper(acc * 10 + c - '0') : acc;
13  }
14
15  int getInt() {
16      return getIntHelper(0);
17  }
```

We got boolean conditions, like c >= '0', && means "and".

```
1  if (test) stmt
2  else stmt // only needed if there is sth to do in the false case.
```

Typically here, stmt will be a block:

```
1  if (test) {
2      stmt 1
3      ...
4      stmt n
5  }
6  else {
7      ...
8  }
```

It is recommended to put curly brace for the statement(s). Consider the dangling else problem:

```
1  if (condition 1)
2      if (condition 2)
3          stmt 1
4  else // this ``else'' actually goes to the second ``if''
5      stmt 2
```

Don't fool by the indentation. This is actually

```
1  if (condition 1) {
2      if (condition 2) {
3          stmt 1
4      }
5      else { stmt 2 }
6  }
```

Conditional operator ? : (also called the ternary operator). if else is a statement while ? : creates an expression: a ? b : c has value b if a is true, has value c if a is false.

Also note that there is no built-in boolean type in C. 0 means false, and non-zero (often 1) means true. We have boolean type, constants true, false in stdbool.h.

4.4 Characters

are just restricted form of integer.

int varies, but typically occupies 32 bits ($\sim 4 \times 10^9$ distinct values). char occupies always 8 bits (256 distinct values). '0' is the character 0, numerically it is 48. Similarly, '9', numerically 57.

char c = '0'; is identical to char c = 48; etc. Everything in memory is numbers, so each character must have a numerical code that represents it. The code here is known as ASCII code.

To convert a char c to its numeric value: c – `'0'` (c – 48). Convert a number (0 - 9) to ASCII: c + `'0'`.

Let's take a second look at getchar: char c = getchar(); not match the prototype: int getchar (); Why int if it's supposed to produce a char? What if there are no chars? (EOF?) If getchar returned any character in this case, there would be no way to indicate EOF (every possible returned value denotes a valid character).

If there are no chars (EOF), getchar produces an int can't possibly be a char (not in the range 0..255). The constant EOF denotes the value getchar produces an eof (often, EOF = -1).

Next question: getInt burns a character after reading an int. Does C has a function like Racket's peek-char? No, but it has ungetc which stuffs a char back into the input stream.

```
int peekchar() {
    int c = getchar();
    return c == EOF ? EOF : ungetc(c, stdin);
}
```

Here we have equality operator ==. The reason we return EOF here is because we don't want to stuff a char if we didn't receive a char. stdin is the keyboard stream (or redirected). ungetc returns the char that was stuffed.

Jan 26

5.1 Improved `getInt`

An improved getInt, one doesn't burn a character.

```c
#include <stdio.h>
#include <ctype.h> // character predicates

int getIntHelper(int acc) {
    int c = peekchar();
    return (isdigit(c)) ? getIntHelper(10 * acc + getchar() - '0') : acc; //
    predicate here will be false for non-digits, EOF.
}
```

This is simpler, but not efficient because we call peekchar and getchar. To be more efficient, we don't need to call getchar twice per character:

```c
int getIntHelper(int acc) {
    int c = getchar();
                                                           comma operator
    return (isdigit(c)) ? getIntHelper(10 * acc + c - '0') : (ungetc(c, stdin), acc
    );
}
```

a, b evaluates a, then evaluates b, result is the value of b. It is equivalent (begin a b) in racket. Use is sparingly, otherwise it will affect the readability of the code.

What if there is whitespace before we reach the int? So we can write a function skip the whitespace, but we don't want it return anything.

```c
void skipws() {
    int c = getchar();
    if (isspace(c)) {
        skipws();
    }
    else ungetc(c, stdin);
}
```

Here is our first example of a void function. The idea is that it returns nothing, therefore, cannot

be used in an expression. For example, `void x = skipws();` is illegal. There are no `void` variables, thus only good for side-effects. To return from `void` functions, either reach the end like in `skipws`, or `return`; with no expression.

With that in place, `getInt` becomes simpler:

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

5.2 Mutation (in Racket)

Basic mutation: `set!` which is pronounced as "set bang", instead of impolite way "set" (with the extremely high volume).

```
(define x 3)
(set! x 4)  ; produces (void), changes delta
```

Now x is 4. Note x must have been previously defined. So we can now change the value of a variable. What can we do with that? For example,

```
> (lookup 'Brad)
false
> (add 'Brad 36484)
> (lookup 'Brad)
36484
```

This is not possible in pure Racket because same expression can't produces different results. How do we implement this in impure Racket:

```
(define address-book empty) ; global variable, and is visible throughout the entire
    program
(define (add name number)
    (set! address-book (cons (list name number) address-book)))
```

Global data is good for defining constants to be used repeatedly. *But* not good with mutation because any part of the program could change a global variable, thus it affects the entire program. So we got hidden dependencies between different parts of the program, and therefore, it's harder to reason about the program.

5.2.1 Application: Memoization

Caching: saving the result of a computation to avoid repeating it.
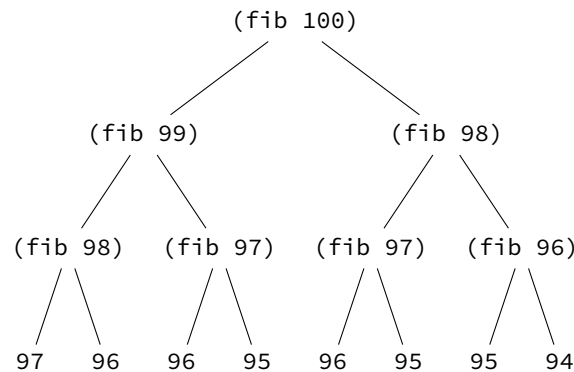
Memoization: maintaining a list or table of cached values.

Consider

```
(define (fib n)
    (cond [(= n 0) 0]
          [(= n 1) 1]
          [else (+ (fib (- n 1))
                   (fib (- n 2)))]))
```

Note that this is inefficient because recursive calls are repeated.

So if want `fib (100)`, it will be expanded as follows:

```
                          (fib 100)
                  /                    \
            (fib 99)                  (fib 98)
            /      \                  /      \
     (fib 98)  (fib 97)      (fib 97)  (fib 96)
      /   \      /   \        /   \      /   \
    97    96   96    95     96    95   95    94
```

Note that (`fib 98`) called twice, (`fib 97`) called 3 times, (`fib 96`) called 5 times ... Thus (`fib n`) is $\Theta(F_n) \approx \varphi^n$ where $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$. So we can avoid repetition by keeping an association list of paris (n, F_n).

```
1   (define fib-table empty)
2
3   (define (memo-fib n)
4       (define result (assoc n fib-table))
5       (cond [result => second]
6             [else (define fib-n
7                       (cond [(<= n 1) n]
8                             [else (+ (memo-fib (- n 1))
9                                      (memo-fib (- n 2)))])])
10                  (set! fib-table (cons (list n fib-n) fib-table)) fib-n]))
```

Few notes here.

- assoc is a builtin function for association list lookup. In particular, (`assoc x lst`) returns the pair (`x y`) from lst or false if it fails.

- Any value can be used as a test. In racket, `false` is false, anything else is true.

- (`cond [x => f]`) ... if x passes (i.e., is not false), then produces (`f x`).

  (`cond [result => second] ...`) is equivalent to (`cond [(list? result) (second result)] ...`)

Now calls to (`fib n`) now happen only once. But our global variable `fib-table` is accessed by anyone. Can we hide it? Can we arrange that only `memo-fib` has access to this global variable? Here is a way to do it:

```
1   (define memo-fib
2       (local [(define fib-table empty)
3               (define (memo-fib n) ...)]
4           memo-fib))
```

Equivalently, use `let`:

```
1   (define memo-fib
2       (let ((fib-table empty))
3           (lambda (n) ...)))
```

This doesn't quite work for the address-book because we have two functions which need access to it.

5.3   Mutation in C

In C, we have an operator = performing mutation ("assignment operator"). For example,

```
1  int main() {
2      int x = 3;
3      printf("%d\n", x); // 3
4      x = 4;
5      printf("%d\n", x); // 4
6  }
```

Note that = is an operator. x = y is an expression, thus it has a value as well as an effect: its value is the value assigned. Then x = 4 sets x to 4, and has value 4. For example,

```
1  int main() {
2      int x = 3;
3      printf("%d\n", x);     // 3
4      printf("%d\n", x = 4); // 4
5  }
```

It really has no advantages... and it has many disadvantages. Because assignment is an expression, C allows us to do x = y = z = 7; which sets all of x, y, z to 7. Now consider the following,

```
1  int main() {
2      int x = 5;
3      if (x = 4) { // this assigns x to 4, and has value 4, non-zero, true.
4          printf("x is 4\n"); // Thus always prints x is 4
5      }
6      x = 0;
7      if (x = 0) { // this assigns x to 0, and has value 0, false.
8          printf("x is 0\n"); // Thus never prints x is 0
9      }
10 }
```

It is easy to confuse assignment with equality check: if (x == 4) ... Thus usually best to use assignment only as a statement.

One thing we can do is that we can leave variables uninitialized and assign them later. For example,

```
1  int main() {
2      int x; // uninitialized
3      x = 4;
4      ...
5  }
```

This is actually not a good idea. Do only with a good reason. For example,

```
1  int x;
2  if (x == 0) {
3      ... // will this run or not?
4  }
```

The answer to the question above is *we don't know*, because x's value is not known! The value of an uninitialized variable is undefined. Typically, it's whatever value was in the memory from before.

6

Jan 28

6.1 Global variables in C

```c
int c = 0; // global variable

int f() {        // returns 0, then 1, then 2, etc.
    int d = c;
    c = c + 1;
    return d;
}

int main() {
    printf("%d\n", f()); // 0
    printf("%d\n", f()); // 1
    printf("%d\n", f()); // 2
}
```

Be careful:

```c
int main() {
    printf("%d\n%d\n%d\n", f(), f(), f());
}
```

This could produce $\begin{matrix} 0 \\ 1 \\ 2 \end{matrix}$ or $\begin{matrix} 2 \\ 1 \\ 0 \end{matrix}$ or others! Order of argument evaluation is *unspecified*.

As with the Racket fib example, we can interfere with f by mutating c.

Can we protect c from access by functions other than f? Yes by using a magic keyword: static.

```c
int f() {
    static int c = 0;
    int d = c;
    c = c + 1;
    return d;
}
```

Here `c` is still a global variable, but it's a global variable that only `f` can see. In terms of variables, there are two notions which we tend to group them together because they are often the same. One is scope, one is extent or lifetime. A traditional global variable has global scope, thus everyone can see it. But more importantly here, its extent: how long it is alive, so it has a global extent. Static variable `c` has a local scope: only `f` can see it, but a global extent: it does not go away when `f` goes away.

6.2 Repetition

Let's say I write a function like this:

```
void sayHiNTimes(int n) {
    if (n > 0) {
        printf("Hi\n");
        sayHiNTimes(n-1);
    }
}
```

This is tail recursion: the recursion call is the last thing the function does. In C, with mutation, we can express this more idiomatically as

```
void sayHiNTimes(int n) {
    while (n > 0) {
        printf("Hi\n");
        n = n - 1;
    }
}
```

This is known as a *loop*, basically shorthand for tail-recursive computation. The body of the loop is executed repeatedly, as long as the condition remains true. In general, if we have

```
void f(int c) {
    if (cont(c)) { // continuation condition
        body(c);
        f(update(c));
    }
}
```

then it becomes

```
void f(int c) {
    while (cont(c)) {
        body(c);
        c = update(c);
    }
}
```

So in the latter version, `f` is not needed, i.e., the things inside may not need to be its own function anymore, if used only once. If we have accumulators, we can still do that.

```
int f(int c, int acc) {
    if (cont(c)) {
        body(c);
        return f(update1(c), update2(c, acc));
    }
```

```
6      return g(acc);
7  }
8
9  f(acc, 0);
```

Then it becomes

```
1  int acc = acc0;
2  while (cont(c)) {
3      body(c);
4      acc= update2(c, acc);
5      c = update1(c);
6  }
7  acc = g(acc);
```

Let's do a concrete example.

```
1  int getIntHelper(int acc) {
2      char c = getchar();
3      if (isdigit(c)) {
4          return getIntHelper(10*acc+c-'0');
5      }
6      return acc;
7  }
8  int getInt() {
9      return getIntHelper(0);
10 }
```

How might we change it? We can do:

```
1  int acc = 0;
2  char c = getchar();
3  while (isdigit(c)) {
4      acc = 10 * acc + c - '0';
5      c = getchar();
6  }
```

which is also shorter. We notice some common patterns, which we can emerge:

```
1  (initialize variables)
2  while (condition) {
3      (body)
4      (update variables)
5  }
```

It's common to forget the "update step", then we might have infinite loop. There is an alternative format which forces you to do all important things upfront, then much harder to forget them.

```
1  for (init; condition; update) {
2      (body)
3  }
```

With for loop, we might do

```
1  int acc = 0;
2  char c;
3  for (c = getchar(); isdigit(c); c = getchar()) {
4      acc = 10 * acc + c - '0';
5  }
```

or even

```
1  int acc = 0;
2  for (char c = getchar(); isdigit(c); c = getchar()) {
3      acc = 10 * acc + c - '0';
4  }
```

So in the latter version, we put the initialization in the part of the loop. Is there a difference between doing these two things? And a related question to that: couldn't I also put initialization of `acc` in the loop as well? The answers to both questions have to do with the scope. When I declare the variable outside the loop, the scope is outside the loop. By putting the definition of `c` right in the loop, the scope of `c` is confined to the loop. Once the loop is done, there is no such `c` anymore. Or even, if we are inclined, we can write the loop as so:

```
1  int acc = 0;
2  for (char c = getchar(); isdigit(c); acc=10*acc+c-'0', c=getchar());
```

Note the usage of comma operator here, which makes this legible. Also, the loop body is empty, which is indicated by `;`, an empty statement, or `{ }`.

What about the `peekchar` version?

```
1  int acc = 0;
2  for (char c = peekchar(); isdigit(c); acc=10*acc+getchar()-'0', c=peekchar());
```

or even

```
1  int acc = 0;
2  for (char c = peekchar(); isdigit(c);c=(getchar(),peekchar())) {
3      acc = 10 * acc + getchar() - '0';
4  }
```

Often loop is controlled by counters, then we update counters. For example, here are some very common patterns:

```
1  c = c + 1;
2  c = c - 2;
3  c = 10 * c;
4  c = c / 2;
5  c = c + d;
```

C has some specialized syntax, which is equivalent to above:

```
1  c += 1;
2  c -= 2;
3  c *= 10;
4  c /= 2;
5  c += d;
```

If we want to increment/decrement by 1, then ++c increments c; --i decrements i.

These are expressions, thus they have a value as well as an effect. ++c increments c and produces the value of c and --i similarly. Which value? the old one or the new one? We can try these out.

```
int i = 1;
printf("%d\n", ++i); // 2
```

Thus the new value. There is also a postfix versions i++, i--, which people seem to like better. These postfix versions increment/decrement i, but produce the old value of i. So it implies the old value must be remembered.

For the most cases, prefix is simpler. The one possible reason for people prefer postfix is because the name of C++, which is not called ++C. If we use increment/decrement operators, and there is no good reason for postfix, we should use prefix version.

6.3   More on Global Data

Global variables likes `int i = 0;`, we should avoid where possible because it creates hidden dependencies. However, Global *constants* are still useful. We can force a variable to remain constant in C. We can say

```
const int passingGrade = 50; // cannot be mutated.
```

6.4   Intermediate Mutation (Racket)

What if we want to work with multiple address books?

```
(define work '(("Manager" 12345)
               ("Director" 23456)))
(define home '())

(define (add-entry abook name number)
    (set! abook (cons (list name number) abook)))

(add-entry home "Neighbour" 34567)

> home
'()
```

home is still empty, no change! Code doesn't work! Not clear how to make it work. What does substitution model say? (add-entry home "Neighbour" 34567) says substitute '() for abook in body. Then it becomes (set! '() (cons (list name number) '())). The latter part makes sense. However, (set! '() ...) doesn't makes sense: we are mutating an empty list; also based on this statement, Racket has no idea we are mutating home.

To make this work... Recall from CS 145 (??), simulation of structs using lambda. Do the same thing to create a struct with one field, called a box. A box has two operations: get the value in the box; set the value to a new value.

```
(define (make-box v)
    (lambda (msg)
        (cond [(equal? msg 'get) v])))
```

```
5  (define (get b) (b 'get))
6
7  (define b1 (make-box 7))
8  (get b1)
9  ; becomes
10 (define b1 (lambda msg) (cond [(equal? msg 'get) 7]))
11 (get b1)
12 ; becomes
13 (get (lambda (msg (cond [(equal? msg 'get) 7]))))
14 ; becomes
15 ((lambda (msg) (cond [(equal? msg 'get) 7])) 'get)
16 ; becomes
17 (cond [(equal? 'get 'get) 7])
18 ; becomes
19 7
```

To support set, we can introduce a local copy of v.

```
1  (define (make-box v)
2      (define val v)
3      (lambda (msg)
4          (cond [(equal? msg 'get) val])))
5
6  (define (get b) (b 'get))
7
8  (define b1 (make-box 7))
9  (get b1)
10 ; becomes
11 (define val_1 7)
12 (define b1 (lambda (msg)
13     (cond [(equal? msg 'get) val_1])))
14 ; and eventually it will give us
15 7
```

Now how do we add set? It requires an extra parameter. We can achieve this by having the box return a function.

```
1  (define (make-box v)
2      (define val v)
3      (lambda (msg)
4          (cond [(equal? msg 'get) val]
5                [(equal? msg 'set) (lambda (newv) (set! val newv))])))
6
7  (define (get b) (b 'get))
8  (define (set b v) ((b 'set) v))
9
10 (define b1 (make-box 7))
11 (set b1 4)
12 ; becomes
13 (define val_1 7)
14 (define b1 (lambda (msg) ... val_1 ...))
15 (set b1 4)
```

```
16  ; becomes
17  (define val_1 7)
18  ...
19  (set (lambda (msg) ... val_1 ...) 4)
20
21  ; becomes
22  (define val_1 7)
23  ...
24  (((lambda (msg) ... val_1 ...) 'set) 4)
25
26  ; becomes
27  (define val_1 7)
28  ((cond [(equal? 'set 'get) val_1]
29          [(equal? 'set 'set) (lambda (newv) (set! val_1 newv))] 4 )
30
31  ; becomes
32  (define val_1 7)
33  ((lambda (newv) (set! val_1 newv)) 4)
34
35  ; becomes
36  (define val_1 7)
37  (set! val_1 4)
38
39  ; becomes
40  (define val_1 4)
41  (void)
```

7

Feb 3

7.1 Intermediate Mutation (Racket) cont'd

Why (now) does this fix the problem?

Before we had this:

```
(define home '())
(add home ... ...)

; reduces to
(add '() ... ...) ; value of the home substituted

; reduces to
(set! '() ... ...) ; can't update home, how do we know this empty list is home not
    sth else?
```

Now we have

```
(define home (make-box '())) ; this creates a variable, local define

(define (add abook name num)
    (set abook (cons (list name num) (get abook)))) ; `get' fetch the var

(add home ... ...) ; this is a function that can update the created variable
```

Boxes are actually built into Racket. The syntax for boxes:

```
exp :: = ... ; anything before
        | (box exp)
        | (unbox exp)
        | (set-box! exp exp) ; first exp is a box, but doesn't have to be an id;
    second exp is a value.
```

Then address book example using built-in box could have been written:

```
(define home (box '(("Neighbour" 34567))))
(define work (box '(("Manager" 12345) ("Director" 23456))))
```

```
3
4   (define (add abook name num)
5       (set-box! abook (cons (list name num) (unbox abook)))))
```

The semantics for box:

```
1   (box v)          ; v is a value
2   ; becomes
3   (define _u v)    ; u is a fresh name
4   _u               ; then (box v) becomes _u
```

Convention: When we write an underscore before a variable name it means the variable's value is not looked up during the expression evaluation, unless (unbox __) is called on it. (Note that this is for the stepping rule, no particular meaning in Racket).

If we have (unbox _n), then we want to find (define _n v), then (unbox _n) produces v. If we have (set-box! _n v), then find (define _n ...) and replace that with (define _n v), then (set -box! _n v) produces (void).

Let's see an example on how we step on boxes.

```
1   (define box1 (box 4))
2   (unbox box1)
3   (set-box! box1 true)
4   (unbox box1)
5
6   ; becomes
7   (define _u1 4)
8   (define box1 _u1)
9   (unbox box1)
10  (set-box! box1 true)
11  (unbox box1)
12
13  ; becomes
14  (define _u1 4)
15  (define box1 _u1)
16  (unbox _u1)
17  (set-box! box1 true)
18  (unbox box1)
19
20  ; becomes
21  (define _u1 4)
22  (define box1 _u1)
23  4
24  (set-box! _u1 true)
25  (unbox box1)
26
27  ; becomes
28  ; (define _u1 4) no longer here
29  (define _u1 true)
30  (define box1 _u1)
31  4
32  (void)
```

```
33  (unbox box1)
34  ; becomes
35  (define _u1 true)
36  (define box1 _u1)
37  4
38  (unbox _u1)
39
40  ; becomes
41  (define _u1 true)
42  (define box1 _u1)
43  4
44  true
```

This is a bit messy, and it is one of the challenges of mutation.

7.2   The same problem in C

Suppose we want to write a function,

```
1  void inc(int x) {
2      x = x + 1;
3  }
4  int main() {
5      int x = 1;
6      inc(x);
7      printf("%d\n", x);
8  }
```

What we want is 2, but what we get is 1. Racket solution is putting the variable in a box. What is the C equivalent? one-field structure?

Structures in C:

```
1  struct Posn {
2      int x;
3      int y
4  }; // we got this cute/weird/curious semicolon here.
```

This semicolon is not optional, *needed*. The "reason" they designed this cuteness/shortcut is for the following:

```
1  struct Posn {
2      int x;
3      int y
4  } p1, p2, p3;
```

where we can define the `struct` and declare the variables at the same time. However, we tend to use `struct` variables locally, and `struct` definitions tend to be global...

Then we can use `struct` as follows:

```
1  int main() {
2      struct Posn p;
3      p.x = 3;
```

```
4        p.y = 4;
5        printf("p=(%d, %d)\n", p.x, p.y);
6    }
```

or we can initialize all at once:

```
1    int main() {
2        struct Posn p = {3, 4};
3        printf("p = (%d, %d)\n", p.x, p.y);
4    }
```

But watch out the following, which is not allowed:

```
1    int main() {
2        struct Posn p;
3        p = {3, 4};
4        printf("p = (%d, %d)\n", p.x, p.y);
5    }
```

Let's try to write a function mutate the `struct`:

```
1    void swap (struct Posn p) {
2        int temp = p.x;
3        p.x = p.y;
4        p.y = temp;
5    }
6
7    int main() {
8        struct Posn p = {3, 4};
9        swap(p);
10       printf("p = (%d, %d)\n", p.x, p.y);
11   }
```

What we want is `p = (4, 3)`, but what we get is `p = (3, 4)`. Still doesn't work.

The problem: C (and also Racket) passes parameters by a mechanism called *call-by-value*. The idea is that the function operates on a *copy* of the argument, not the argument itself. Note that the Racket substitution model naturally implements call-by-value. For example, if we do,

```
1    (define x 3)
2    (f x) => (f 3)  ; the value of x, not x itself
```

In C, what we have is

```
1    void inc(int x) {
2        ++x;
3    }
```

Here x really does get mutated, but it's a copy of x, not the original from the caller. Therefore, the original remains the same. Similarly, in swap we wrote, the entire structure is copied into the function, thus the original structure does not change.

So there is something special about boxes. They are not equal to the value they hold, but they tell (know) you how to *find* the value: `unbox`/`get` to find the value. What does that look like in C? To find a value, we are asking where it is located: it is in memory (RAM). As we said, every value in memory

has an address. If given the address, we can "find" the value. Thus addresses could function as boxes. Instead of passing a value to a function, we can pass an address. For example,

```
int main() {
    int x = 1;
    inc(&x);  // & is known as ``address-of'' operator, which passes x's address,
    not its value.
    printf("%d\n", x);
}

void inc(int x) {   // this is wrong now. We didn't
    x = x + 1;      // get an int, we get an address.
}
```

Maybe: `void inc(address x) {...}` which is also wrong. We need more info than address: what type of data is stored at that address? We need to say x is the address of an `int`.

```
void inc(int *x) {  // x is called a pointer to an int
    x = x + 1;      // (i.e., the address of an int)
}
```

This is still wrong because of its body: we don't want to add 1 to the address, but we want to add 1 to the value stored at the address. So what we want is:

```
void inc(int *x) {
    *x = *x + 1;  // * = dereference operator
}
```

and the dereference operator = fetch the value stored at this address (`unbox` in Racket). LHS of assignment: `*x = expr` = store the value of expr at address x (`set-box!` in Racket). Thus in Racket, it's equivalent to (`set-box! x (+ (unbox 1) 1)`).

When we see `int *x`, it's like x is a pointer to an `int`, but it's intended to be read is "`*x` is an `int`".

Alternatively we can write:

```
void inc(int *x) {
    *x += 1;
}
// OR
void inc(int *x) {
    ++*x;
}
// OR?
void inc(int *x) {
    *x++; // WRONG - why?
}
```

When we say `*x++`, we got operators on both left and right. Which of these two actually happens first? the `*` or the `++`? In C, postfix always takes precedence over prefix. So `*x++` means `*(x++)`. The address is incremented and the old value of the address is fetched (and thrown away). Thus no change to the original variable. If we cannot give up the postfix habit, we can do:

```
void inc(int *x) {
    (*x)++;
}
```

Now consider swap. The first version didn't work. Just like `inc`, we can fix this by passing a pointer:

```
void swap(struct Posn *p) {
    int temp = *p.x;
    *p.x = *p.y;
    *p.y = temp;
}
```

which is wrong, and won't even compile. Same problem as before: postfix before prefix. `*p.x = *p.y` means `*(p.x) = *(p.y)` and `p.x, p.y` aren't pointers. We need parentheses:

```
void swap(struct Posn *p) {
    int temp = (*p).x;
    (*p).x = (*p).y;
    (*p).y = temp;
}
```

However, this is clunky. `(*p).x` is common enough that it has its own notation: `p->x`. Thus the previous code can be written as:

```
void swap(struct Posn *p) {
    int temp = p->x;
    p->x = p->y;
    p->y = temp;
}
```

Thus we have more sophisticated user input: `scanf`. Note that `scanf("%d",x)` is wrong. It should read x as a decimal integer and skip leading whitespace. Here `scanf` is a function which can't modify x. Instead, we can do `scanf("%d", &x)`. We can also do `scanf("%d %d", &x, &y)`. The space between `%d`'s means to skip *any* amount of whitespace between the two `int`s (including zero). Zero space is possible if the second `int` is negative.

Note that `scanf` returns the number of arguments actually read. `scanf` has lots of options, very complicated.

8

Feb 4

8.1 Advanced Mutation

It means mutating structures and lists.

In Scheme, we can mutate parts of a cons with `set-car!` and `set-cdr!`.

In Racket, cons fields are immutable, cannot be mutated. For mutable paris, Racket provides `mcons`. To mutate fields, `mset-car!`, `mset-cdr!`. For `struct`s, it is also immutable. But Racket provides the option `#:mutable`. It will look like this:

```
1  (struct pos (x y) #:mutable)
2  (define p (posn 3 4))
3  (set-posn-x! p 5)
4  (posn-x p) ; => 5
```

This has an impact on our semantics: from CS 145, we said (`make-posn v1 v2`) is a value. However, now, (`posn v1 v2`) cannot be a simple value if it is mutable. It has to behave more like a `box`. How would a `struct` behave like a box? Here are the two ways. Is a `struct` automatically boxed? or is a `struct` a box? We can find that out by some experiments.

```
1  (struct posn (x y) #:mutable #:transparent)
2  (define (mutate-posn p)
3      (set-posn-x! p (+ 1 (posn-x p))))
4
5  (define (mutate-posn2 p)
6      (set! p (posn (+ 1 (posn-x p)) (posn-y p))))
7
8  (define p (posn 1 2))
9  (define q (posn 1 2))
10
11 (mutate-posn p)
12 (mutate-posn2 q)
13 p ; (posn 2 2)
14 q ; (posn 1 2)
```

So a `struct` is not automatically boxed, but it does box its contents. So we can write (`posn v1 v2`) as

```
1  (define _val1 v1)    ; recall, no expansion
```

```
2   (define _val2 v2)
3   (posn _val1 _val2)
```

(posn-x p) where p is (posn _val1 val2), we find the definition for _val1, fetch the value.

(set-posn-x! p v) where p is (posn _val1 val2), we find (define _val1 ...), replace it with (define _val1 v).

So now we are ready to do some stepping. For example,

```
1    (define p1 (posn 3 4))
2    (set-posn-x! p1 5)
3
4    ; becomes
5    (define _v1 3)
6    (define _v2 4)
7    (define p1 (posn _v1 _v2))
8    (set-posn-x! p1 5)
9
10   ; becomes
11   (define _v1 3)
12   (define _v2 4)
13   (define p1 (posn _v1 _v2))
14   (set-posn-x! (posn _v1 _v2) 5)
15
16   ; becomes
17   (define _v1 5)
18   (define _v2 4)
19   (define p1 (posn _v1 _v2))
20   (void)
```

These rules generalize to any mutable struct, mcons.

Now consider

```
1    (define lst1 (cons (box 1) empty)) ; note that we are using cons, not mutable
2    (define lst2 (cons 2 lst1))
3    (define lst3 (cons 3 lst1))
4    (set-box! (first (rest lst2)) 4)
5    (unbox (first (rest lst3))) ; gives 4
```
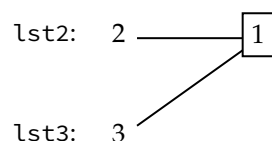
By the CS 145 understanding, this should produce 1. But in fact, it gives 4. Why?

$$lst2 = (cons \ 2 \ (cons \ (box1) \ empty))$$
$$lst3 = (cons \ 3 \ (cons \ (box1) \ empty))$$

In this case, the two (box 1)'s are actually the same object. When we define lst2 and lst3, these two lists actually share the same tail. What we actually have is

We could never tell that this was true in CS 145 because there is no way to observe a difference in

terms of whether these two lists are completely distinct lists or actually they are sharing the tail unless we can perform mutation. Thus we do need mutation to observe this.

Nevertheless, we could actually deduce they are actually sharing the tail. We can do

```racket
(define lst1 '(1 2 ... 100000000)) ; which takes O(n) time to build
(define lst2 (cons 0 (rest lst1))) ; O(1) time
```

If the second line is fast, it's not possible for Racket to recreate an entire list, thus it must be reusing that list. From that perspective, lst2 is sharing the tail with lst1.

Under the old substitution rules, we will get the wrong answer 1. Under the new substitution rules, boxes are rewritten as a separate define with deferred lookup. We then end up with

```racket
(define _val 1)
lst2 = (cons 2 (cons _val1 empty))
lst3 = (cons 3 (cons _val1 empty))
```

Here the shared item reflected in the rewrite.

All these force us to rethink what we mean by define. If we have

```racket
(define x 3)
(set! x 7)
x
; becomes
(define x 3)
(void)
x
; becomes
7
```

We cannot replace all occurrence of x with 3, otherwise we could have gotten 3 at the end. So x is not just a value, but something we can mutate, it's an entity we can access. Therefore x must denote a *location*, and the location contains the value. So we don't just have one lookup $\delta : \text{var} \to \text{value}$, instead we have *two* lookups: $\text{var} \to \text{location}$, $\text{location} \to \text{value}$ where the second lookup is carried out by RAM.

set! changes the $\text{location} \to \text{value}$ map, but not the $\text{var} \to \text{location}$ map (nothing changes that, at least for now). Similarly, set-box! changes the $\text{location} \to \text{value}$ mapping. (define ...) creates a *location*, fills it with a value. Keep this in mind.

8.2 Aliasing in C

Does this happen in C as well? Yes. Consider

```c
int main() {
    int x = 1;
    int *y = &x;
    int *z = &x; // or int *z = y;
    *y = 2;
    *z = 3;
    printf("%d %d %d\n", x, *y, *z);
}
```

The output is 3 3 3. Why? y is initialized to x's address, the y points to the location where x resides, and z is initialized to y (or &x), thus z also equals to x's address. Therefore, *y = 2 stores 2 at x's location: x == *y == *z == 2. Similarly, *z = 3 stores 3 at x's location. We can picture this visually:
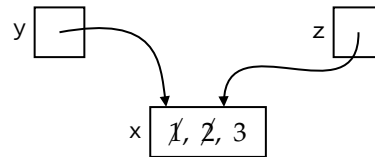
Therefore, x, *y, *z are three different names for the same data. This phenomenon is called *aliasing*: accessing the same data by different names. Aliasing is tricking business, and it can be subtle. Consider the following :
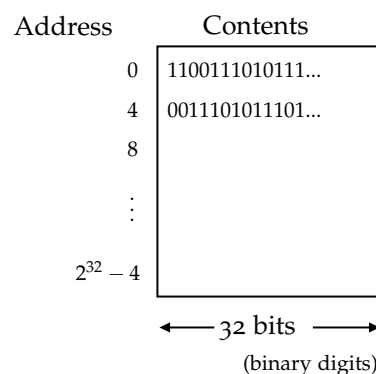
```
void f(int *x, int *y) {
    *y = *x + 1;
    if (*y == *x) {
        printf("How could this every print?\n"); // (*)
    }
}

int main() {
    int z = 1;
    f(&z, &z);          // makes x and y aliases, thus (*) DOES print
    printf("%d\n", z);  // print 2
}
```

Hence it makes programs very difficult to understand.

8.3 Memory and vectors

Recall from the beginning of the course: Memory is a set of numbered "slots":

Each box is 8 bits (one byte), but they are usually treated in groups of 4-byte *words*.

Primitive data structure: the *array*, it's a "slice" of memory, and a sequence of consecutive memory locations. We will discuss at length when we return to C.

In Racket (also Scheme), it's known as the vector. It is used much like a traditional array. Unlike arrays, the slot of the vector can hold items of any size. Thus we can have unlimited integers, strings, whatever.
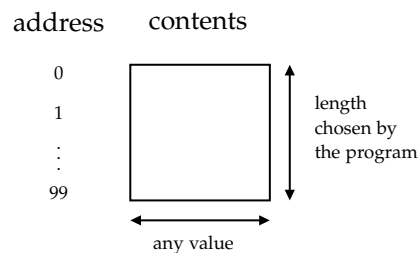
8.4   Vectors in Racket

```
1  (define x (vector 'blue true "you"))
```

is a 3-item vector. From here we know that we can put whatever we want in `vector`. We can also do

```
1  (define y (make-vector 100))
2  y = '#(0 0 0 ... 0 0 0)
```

which creates a `vector` of length 100.

Unlike a `list`, it's like a many slice of memory. So it looks like,

We can also create a vector like

```
1  (define z (build-vector 100 sqr))
2  => 0 1 4 9 ... 99^2
3  (define y (make-vector 100 5))
4  => 5 5 5 5 ... 5 ; a hundred 5's
```

What is different about `vectors` versus `lists` is the way we work with them. We already know we work with a `list` by taking the first item and taking the rest. We process the list recursively. Vectors are quite different. They are not accessed by `first` and `rest`. Vector says which item do you want?

```
1  (define y (make-vector 100 5))
2  (vector-ref y 7)      ; gives 5
3  (vector-set! y 7 42)
4  (vector-ref y 42)     ; gives 42
```

Thus we access/mutate items by index. Hence the main advantage of `vectors` over `lists` is `vector-ref` and `vector-set!` run in $O(1)$ time. This is a consequence of the way `vector` is stored versus the way `list` is stored. With a `list`, we can't get to a second item until we have been through the first item. If we want the 100th item in the list, we have to `rest` 99 times and `first`. On the other hand, with `vectors`, which is a slice of memory, if we want 100th item of the `vector`, all we need to say is where it's start and add 100 × the slot size, which gives us the exact address where the item is located. Thus we can fetch any item in the `vector` in constant time.

It turns out these things are cranky to work with. What's wrong with `vectors`? It has several disadvantages:

1. *The size is fixed.* A `list` can very easily grow: all we need to do is to put `cons` at the front. However, with a `vector`, we have made a choice to designate a particular slice of memory of being part of that `vector`. The memory before/after it could very well be used for other things. Thus we cannot just take a `vector` and make it bigger.

2. *It's difficult to add or remove elements.* Everything is stored consecutively. If we want to take something out of the middle, that leaves a gap, we then need to shuffle everything down to close the gap. Similar situation if we want to add something into the middle.

3. `vector-set!` *tends to force an imperative style.* Once we start to work with `vectors` instead of `lists`, we will find our Racket code doesn't work so well functionally anymore, it kinda force us to the imperative Racket.

Feb 9

9.1 Vectors in Racket cont'd

Let's first write `build-vector`:

```
1  (define (my-build-vector n f)
2      (define res (make-vector n))
3      (define mbv-h i) ; my build vector helper
4          (cond [(= i n) res]
5                [else (vector-set! res i (f i)) (mbv-h (+ i 1))])
6      (mbv-h 0))
```

Vectors work well with imperative-style algorithms. Racket provides macros `for`, `for`/`vector` that facilitate this. Thus we could write

```
1  (define (my-build-vector n f)
2      (define res (make-vector n))
3      (for ([i n]) ; i goes from 0 to n
4          (vector-set! res i (f i)))
5      res)
```

Or in `for`/`vector` form,

```
1  (define (my-build-vector n f)
2      (for/vector ([i n]) (f i)))
```

Let's do another example with `vectors`. For example, sum the elements of a `vector`,

```
1  (define (sum-vector vec)
2      (define (sv-h i acc)
3          (cond [(= i (vector-length vec)) acc]
4                [else (sv-h (+ i 1) (+ acc (vector-ref vec i)))]))
5      (sv-h 0 0))
```

It's very look-like, then we can use `for`:

```
1  (define (sum-vector vec)
2      (define sum 0)
3      (for [(i (vector-length vec))]
```

```
4          (set! sum (+ sum (vector-ref vec i)))))
5      sum)
```

It's not pure functional because it uses mutation. But in some sense, it *looks* pure functional. And the use of mutation is confined to the internals of sum-vector. It can't be detected outside the function. Thus outsiders could consider it pure functional.

This provides a strategy for keeping the problems with mutation under control: hide it behind a pure functional interface.

Recall our model for computer's memory: huge lookup table,
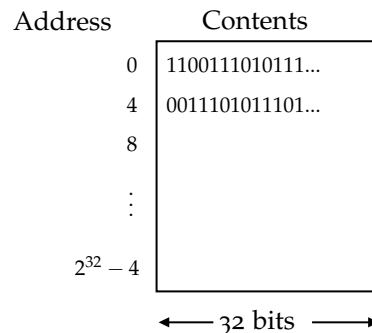
Racket vectors model that
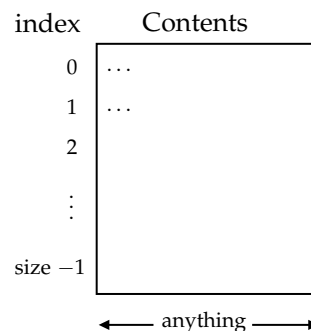
How does this work? Remember that memory slots only hold fixed size data, and yet Racket has unlimited numbers? strings?

Let's go back to struct. Recall

```
1  (define (mutate-posn p)
2      (set-posn-x! p (+ 1 (posn-x p))))
3
4  (define p (posn 3 5))
5  (mutate-posn p)
6  (posn-x p) ; gives 4
```

Does that happen in C? The equivalent in C would be

```
1  void mutate (struct Posn p) { p.x += 1; }
2  int main() {
3      struct Posn p = {3, 5};
4      mutate(p);
5      printf("%d\n", p.x); // gives 3
6  }
```

So from here, we learnt that Racket `struct`s aren't like C `struct`s: the `struct` is coped in Racket, but changes to the field still persists. As we conclude last time, the fields of a Racket `struct` are boxed, i.e., they are pointers.

Similarly, the items in Racket `vector` are *addresses* that point to the actual contents (which can then be of any size). Similarly, the fields of a cons are pointers. When we say

```
(cons 1 (cons 'blue (cons true empty)))
```

we can represent this as a box-and-pointer diagram:
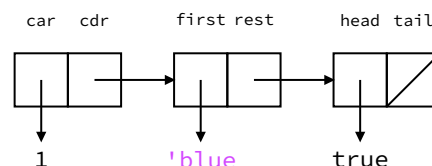
Also, since Racket is dynamically typed, the values 1, `'blue`, true must include type information. More later.

9.2 "Vectors" in C: Arrays

An `array` is a sequence of consecutive memory locations. For example,

```
int main() {
    int grades [10]; // Array of 10 ints
    for (int i = 0; i < 10; ++i) {
        scanf("%d", &grades[i]);
    }
    int acc = 0;
    for (int i = 0; i < 10; ++i) {
        acc += grades[i];
    }
    printf("%d\n", acc/10);
}
```

When we see `a[i]`, this accesses the *i*-th element of array a.

`int grades[10];` tells us valid entries are `grades[0]`, ..., `grades[9]`.

What happens if we go out of bounds? It's undefined behaviour.

Will it stop us? No. The program may or may not crash; no way to detect that if it didn't crash.

We can also give the bound implicitly:

```
int main() {
    int grades [] = {0, 0, 0, 0, 0};
    printf("%zd\n", sizeof(grades)/sizeof(int)); // gives 5
}
```

`sizeof` operator tells us the amount of memory `grades`/`int` occupy, 20 and 4 bytes respectively.

`int` in our implementation of C occupies 32 bits but we are running on 64 bits machine. So the amount of memory available might well be larger than what 32 bits can hold. So what `sizeof` creates is a value of type, not `int`, but `size_t`, which is a type that the compiler supplies. First, it is unsigned, i.e., no

negative `size_t`. Second, it's large enough to hold any amount of memory. `z` in `"%zd"` here ensures that when printing out, it is not interpreted as `int`, indicates `size_t`.

Now let's talk about functions on `arrays`.

```
int sum(int array[], int size) { // size is not part of the type of array, works on
     arrays of any size
    int res = 0;
    for (int i = 0; i < size; ++i) res += arr[i];
    return res;
}
```

If we pass `arrays` by value, we copy the whole array, which is expensive. So C will not pass `array` by value. Consider

```
int main() {
    int myArray [100];
    // ...
    int total = sum(myArray, 100); // looks like a copy, how is this not copy?
    // ...
}
```

This leads to the most confusing rule in all of C: The name of an `array` is shorthand for a pointer to its first element. So in fact, `myArray` is shorthand for `&myArray[0]`. Therefore, `sum(myArray, 100)` passes a pointer, not a whole `array`, into the function. But `sum` was expecting an `array`, not a pointer.

Why not `int sum(int *arr, int size)` then? The answer is we could have. `int *arr`, `int arr[]` are identical in meaning, in parameter declarations. Now let's use pointers to write sum this time:

```
int sum(int *arr, int size) {
    int res = 0;
    for (int i = 0; i < size; ++i) res += arr[i]; // is that OK to do this to a
    pointer?
    return res;
}
```

Yes, it actually does work.

9.3 Pointer Arithmetic

Let t ne a type. If we declare an array `t arr[10];`, then we know `sizeof(arr) = 10 * sizeof(t)`, and `arr` is shorthand for `&arr[0]`. Therefore, `*arr` is equivalent to `arr[0]`. Then what expressions produces a pointer to `arr[1]`?

> `arr + i` is shorthand for `&arr[i]` for i = 1, 2, ...

Numerically, `arr + n` produces the address equal to `arr + n * sizeof(t)`. If `arr + k` is shorthand for `&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). Actually, to confuse other people, we can do

$$a[k] \equiv *(a + k) \equiv *(k + a) \equiv k[a]$$

We can push the pointer version of sum slightly further:

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

Note that the usage of arr+size, which is a pointer outside the array: it is valid to construct a pointer that goes one slot past the end of the array, but it is not valid to dereference that pointer (UB).

cur < arr + size is a pointer comparison: return true if cur points to an earlier element in the same array than arr + size. Comparing pointers in different arrays or not in arrays at all, it UB.

Any pointer can be thought of as pointing to the beginning of an array. We have the same syntax for accessing items through an array as through a pointer. So are arrays and pointers the same thing? No! Let's see an evidence:
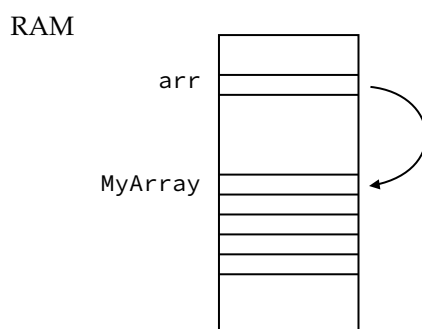
```
void f(int arr[]) {
    printf("%zd\n", sizeof(arr));
}
int main() {
    int myArray[10];
    printf("%d\n", sizeof(myArray));
    f(myArray);
}
```

Outputs: 40 (size of array), 8 (size of a pointer to the array's first item). If we take a look at RAM, they are quite different:
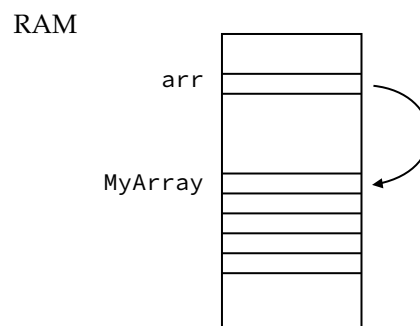
Feb 11

Continue from last time:

What does the compiler do? When we say `myArray[i]`, it will fetch `myArray` location from environment, and add `i*sizeof(int)`, and then fetch this address from the store (RAM).

What happens when you call `arr[i]`? It will fetch `arr` location from environment, then fetch `myArray` address from store, then add `i*sizeof(int)` to address, then fetch the value from this address in store.

These two operations, `myArray[i]` and `arr[i]` may look the same, but do slightly different things.

We saw that a Racket `struct`, e.g., `(struct posn (x y))`, is like a C `struct` whose fields are pointers. How can we achieve this in C?

> Aside:
>
> ```
> int *x;
> int* x;
> int * x;
> ```
>
> They all mean the same thing. The first one is more idiomatic C, the second one is often favored in C++, and the third one, the instructor is not sure there's too many people favor the third one, but it exists.
>
> But there's one technical thing, if we do
>
> ```
> int *x, y;
> ```
>
> to declare two pointers. Here x is a pointer, y is not a pointer, just an `int`. If we want y also to be a

> pointer, we have to do
>
> ```
> int *x, *y;
> ```

```
1  struct Posn {
2      int *x;
3      int *y;
4  };
5
6  int main() {
7      struct Posn p;
8      // what are p.x, p.y pointing at?
9      *p.x = 3;
10     *p.y = 4;
11 }
```

Very likely, this is gonna crash. `p.x` and `p.y` are uninitialized pointers, thus they point at arbitrary locations, dictated by whatever value they happen to hold.

Racket must do something more than this: `(posn 3 4)` must also reserve memory for x and y to point at, to hold the 3 and 4. Therefore, we need to do the same in C:

```
1  #include <stdlib.h>
2  struct Posn makePosn(int x, int y) {
3      struct Posn p;
4      p.x = malloc(sizeof(int));
5      p.y = malloc(sizeof(int));
6      *p.x = x;
7      *p.y = y;
8      return p;
9  }
```
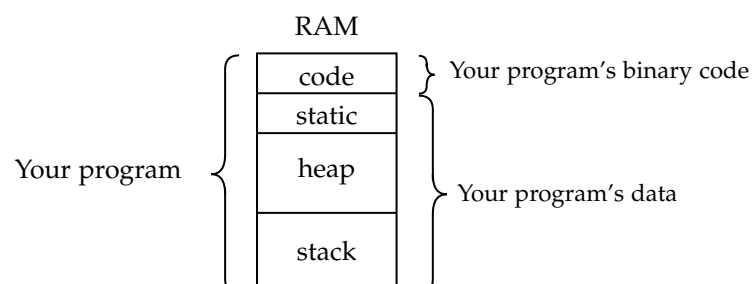
`malloc(n)` says request n bytes of memory. Then in `main`, we can do

```
1  int main() {
2      struct Posn p = makePosn(3, 4); // OK
3      ...
4  }
```

We need to understand exactly what's happening.

10.1 Memory Management

Memory layout (applies to C and Racket)

Static area is where global/static variables are stored. The lifetime of these variables is the entire program.

What is a stack? It is abstract data type (ADT) with LIFO (last in first out) semantics. In LIFO, we can only remove the most recently-inserted item. For stacks, operations are

- 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?

Racket lists are stacks: Push = cons, Pop = rest, Top = first.

Program stack stores local variables. Let's see an example.

```
int fact(int n) {
    int rec = 0;
    if (n == 0) return 1;
    rec = fact(n-1);
    return n * rec;
}

int main() {
    int f = fact(3);
}
```

`ret` here denotes what do I return to.

fact	rec: 0 n: 0 ret: fact
fact	rec: 0 n: 3 ret: fact
fact	rec: 0 n: 2 ret: fact
fact	rec: 0 n: 3 ret: main
main	f: ?

Then after all recursive calls are done, we have
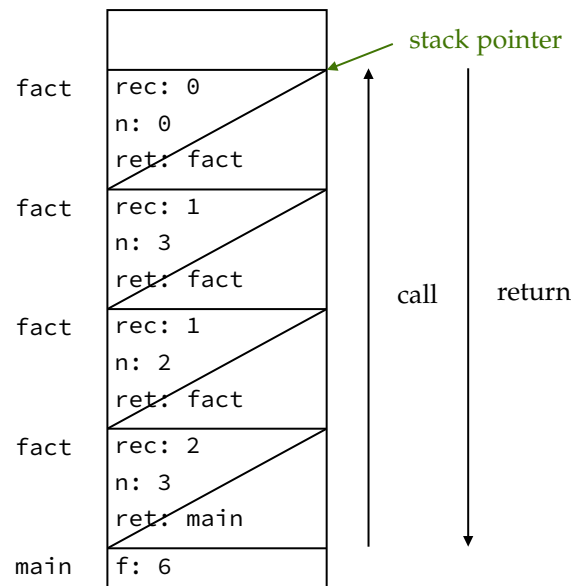
Each function call gets a *stack frame*:

- local variables are pushed onto the stack,

- also the return address: where to go when the function returns.

- each invocation of the function gets its own version of local variables.

When a function returns, its stack frame is popped. This means all local variables in that frame are released. They are not typically erased: the program keeps track of where the top of the stack is. This is what's known as *stack pointer*. The "top-of-stack" pointer moved to top of the next frame, then the old frame will be overwritten next time a frame is pushed onto the stack.

The stack holds local variables. The lifetime of variables on the stack will be scope-based.

So what if you have data that must persist after a function returns. We might try the following. What's wrong with this?

```
struct Posn makePosn(int x, int y) {
    struct Posn p;
    int a = x;
    int b = y;
    p.x = &a;
    p.y = &b;
    return p;
}
```

Here we have initialized `p.x` and `p.y`, but we have initialized them to these local variables `a` and `b`'s address. Wait no... This is bad. If in `main`, we do

```
int main() {
    struct Posn p = makePosn(3, 4);
}
```

When the function returns, local variables `a` and `b` are released; pointers `p.x` and `p.y` are thus pointing to dead memory.

Never return a pointer to data stored on local stack. If a function is to return a pointer, the pointer should point to either static, heap, or non-local stack data.

11

Feb 23

11.1 Memory Management cont'd

Some of the material is repeated here because the twitch stream was broken last time...

So what if you have data that must persist after a function returns. We might try the following. What's wrong with this?

```
1  struct Posn makePosn(int x, int y) {
2      struct Posn p;
3      int a = x;
4      int b = y;
5      p.x = &a;
6      p.y = &b;
7      return p;
8  }
```
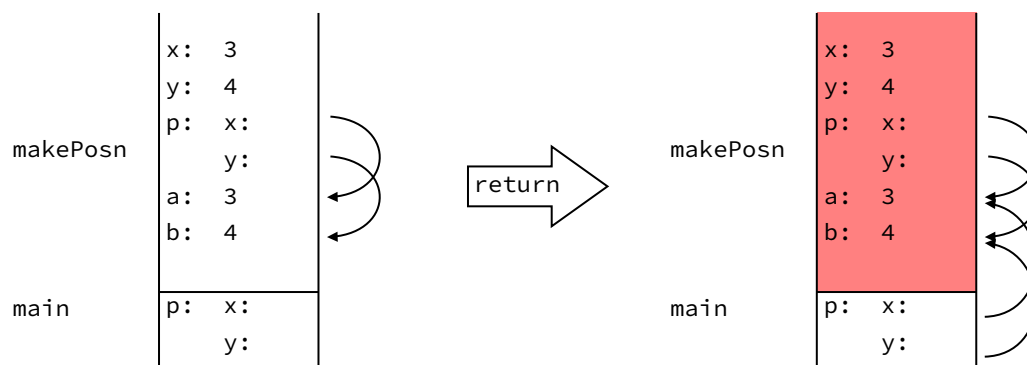
This is very BAD!! If in main, we do

```
1  int main() {
2      struct Posn p = makePosn(3, 4);
3  }
```

Let's imagine what's happen in stack:

When the function returns, local variables a and b are released; pointers p.x and p.y are thus pointing to dead memory. Returned p contains pointers to local stack-allocated data. Don't do this! x + y (or

`a + b`) won't survive past the end of `makePosn`.

This is then the difference between pointers to stack memory and what `malloc` does. `malloc` requests memory from the *heap*. Heap is a pool of memory from which we can explicitly request "chunks".
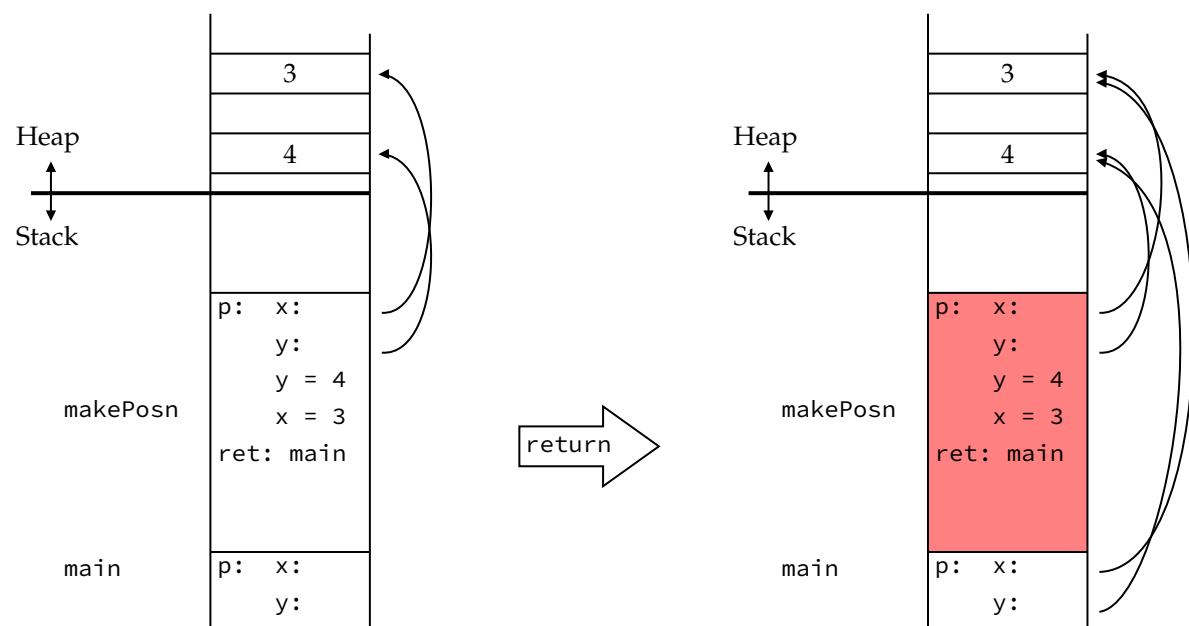
The lifetime of stack memory is until the variable's scope ends. (e.g., end of the function). The life of heap memory is arbitrary. Let's recall

```
struct Posn makePosn(x, y) {
    struct Posn p; // one the stack
    p.x = malloc(sizeof(int)); // points to the heap
    p.y = malloc(sizeof(int)); // points to the heap
    *p.x = x;
    *p.y = y;
    return p;
}

int main() {
    struct Posn p = makePosn(3, 4);
    ...
}
```

Now let's consider the memory layout.

When `makePosn` returns, `p` (including `p.x`, `p.y`) is popped off the stack, which is then no longer live. However 3 and 4 are on the heap, which are still live. So `p` from `makePosn` is copied back to `main`'s frame. `main` then has access to 3 and 4 on the heap, and these outlive `makePosn`. Note that `make-posn` (or `posn`) in Racket would do the same thing.

What is the lifetime of heap-allocated data? As we discussed, it is arbitrarily long. If heap-allocated data *never* gets away, the program will eventually run out of memory, even if most of the data in memory is no longer in use. Racket solution to this problem: there is a run-time process detects memory that is no longer accessible. For example,

```
(define (f x)
    (define p (posn 3 4)) ; certainly is not needed after f returns
```

```
3      ...
4      (+ x 1))
```

and automatically reclaims it. This is a process known as *garbage collection*.

On the other hand, C solution: Heap memory is `freed` when we `free` it. For example,

```
1  int *p = malloc(...);
2  ...
3  free(p); // release p's memory back to the heap.
```

What happens if we don't call `free`? Failing to `free` all allocated memory is called a *memory leak*. Programs that leak memory will eventually fail, if they run long enough.

Consider the following program:

```
1  int *p = malloc(sizeof(int));
2  free(p);
3  *p = 7;
```

Will this program crash? Almost certainly, w.h.p. not. `free(p)` doesn't change `p`. `p` still points to that memory, therefore, storing something at that memory probably still works, but `p` is not pointing at a valid location. And that location may be assigned to another pointer by another `malloc` call. This is called a *dangling pointer*, which is bad.

So a better solution would be: after `free(p)`, we assign `p` to point to a guaranteed-invalid location:

```
1  int *p = malloc(sizeof(int));
2  free(p);
3  p = NULL;
```

This is called *null pointer*, which points to nothing. `NULL` us bit really part of the C language. There are certain headers that define `NULL` as constant equal to `0`. We could equally well say `p = 0;` Now what happens if we deference `NULL`? It's an undefined behavior. The program *may* crash.

If `malloc` fails to allocate memory, it returns `NULL`. Moreover, `freeing` a `NULL` pointer is guaranteed to be safe, which does nothing.

Let's consider a situation we discussed before, but in a slightly different way:
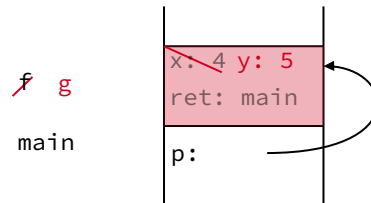
```
1  int *f() {
2      int x = 4;
3      return &x;
4  }
5
6  int g() {
7      int y = 5;
8      return y;
9  }
10
11 int main() {
12     int *p = f();
13     g();
14     printf("%d\n", *p);
15 }
```

Let's compile it with `clang`. Then we do get a warning:

```
1  warning: address of stack memory associated with local variable returned [-Wreturn-
      stack-address]
2      return &g;
3             ^
```

If we run the program, we get 5, not 4. Let's take a look at memory:

This is precisely why the program is so dangerous. This is not guaranteed, but this could happen. Let's compile with `gcc`. We then get another warning of the same kind of thing:

```
1  warning: function returns address of local variable [-Wreturn-local-addr]
2      return &g;
3             ^~
```

This time when we run it, we see the program crashes with "Segmentation fault (core dumped)". `gcc` is exercising an amazing amount of caution here , which saves us from ourselves. `gcc` is taking an approach which says: anytime I think you are returning a variable that is a pointer to a local, I am going to instead return `NULL`.

`p` points to a dead memory. By the time `f` returns, `x` is no longer a valid location. This is another instance of *dangling pointer*. The program (probably? depending on the compiler) will not crash, but will behave badly. When `g` is called, it occupies `f`'s old stack frame. `y` now occupies `x`'s old spot. `*p` now is 5 (still a dangling pointer).

The lesson we learnt from this: NEVER return a pointer to a local variable.

If we want to return a pointer, it should point to static, heap, or non-local stack data. For example,

```
1  int *pickOne(int *x, int *y) {
2      return ... ? x : y;
3  }
```

This is fine because `x` and `y` are not pointing to my local stack. It's fine to point to other's stack. Let's also show an example of pointing to heap:

```
1  struct Posn *getMeAPtr() {
2      struct Posn *p = malloc(sizeof(struct Posn));
3      return p;
4  }
```

Finally, let's consider a pointer to static:

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

This is also fine because the lifetime of global variable is the entire program, thus it is not pointing to a dead memory.

In general, when should we use heap? Three situations come into mind:

1. For data that should outlive the function that created it.

2. For data whose size is not known at compile-time.

3. For large local arrays.

Let's see some examples. The first situation is as above. For the second situation,

```
int *p = malloc(sizeof(int));
```

Is that useful? Why not just `int` n? But what if we ask for more memory? Consider

```
int numSlotsNeeded;
scanf("%d", &numSlotsNeeded);
int *p = malloc(numSlotsNeeded * sizeof(int));
...
free(p);
```

Now we can access `p[0]`, `p[1]`, `...`, `p[numSlotsNeeded-1]`, which is dynamic array (heap-allocated). Let's take a look at the memory layout.
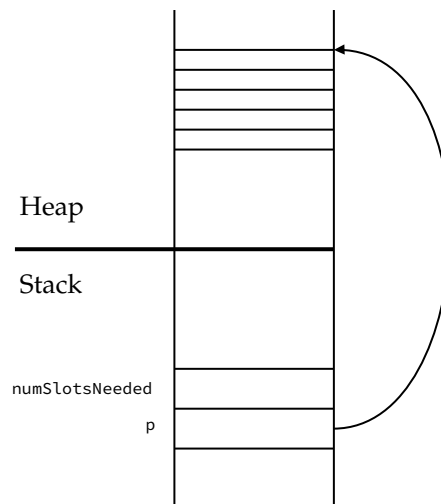
Finally, let's consider the third case. Programs typically have more heap memory available than stack memory. Consider

```
int recursiveFunction(int n) {
    ...
    int HeapArray[10000]; // this eats up stack space for each recursive call
    ...
    recursiveFunctionFunction(n-1);
}
```

11.2   Linked list

C arrays mimic Racket vectors. Can we get the behavior of a Racket list? We know that (cons x y) produces a pair

Recall that Racket is dynamically typed, then we could have
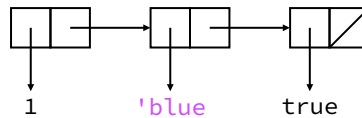
List items can have different types. On the other hand, C is statically typed, which means list items would need to have the same type. If not the same type, then it leads to headaches. So there's no real need for pointers to the data fields. We could write like this:
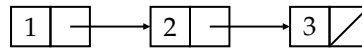
This suggests us a `struct` definition like

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

Also, we can write cons:

```
struct Node *cons(int data, struct Node *next) {
    struct Node *result = malloc(sizeof(struct Node));
    result->data = data;
    result->next = next;
    return result;
}
```

Then in `main`:

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

This is called a *linked list*.

Feb 24

12.1 More linked list

Now let's process a linked list.

```
int length(struct Node *lst) {
    if (!lst) return 0;
    return 1 + length(lst->next);
}
```

Or using iteration/loops, we can write like this

```
int length(struct Node *lst) {
    int res = 0;
    for (struct Node *cur = lst; cur; cur=cur->next) {
        ++res;
    }
    return res;
}
```

Can we write map? The idea would be

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

Well, are we allowed to pass a function as an argument to another function in C? Technically no, because functions in C are not first class values, thus we cannot take them and store them in data structures and so on. However, we can pass a pointer to a function in C. In map(f,lst), the name of a function is shorthand for a pointer to its code. This is how we use map, how do we write it? What type do we use for f? We know it's a pointer to a function.

```
struct Node *map(int *f(int), struct Node *lst);
// this is wrong, because postfix before prefix. This is a function returning a
    pointer to an int.

```

```
4    // correct way
5    struct Node *map(int (*f)(int), struct Node *lst) {
6        if (!lst) return 0;
7        return cons(f(lst->data), map(f, lst->next));
8    }
```

Now it's time to `free` the list. Say we have created a list:

```
1    int main() {
2        struct Node *lst = cons(1, cons(2, cons(3, 0)));
3        ...
4        free(lst);
5    }
```

This will create memory leak because 2 and 3 get leaked. Our second attempt:

```
1    int main() {
2        ...
3        for (struct Node *cur=lst; cur; cur=cur->next) free(cur);
4    }
```

This is still wrong, because `cur=cur->next` happens after `free(cur)`, then `cur` is dangling. To fix this, we need to grab the next pointer before we `free`. Now let's write a loop doing properly:

```
1    for (struct Node *cur=lst; cur;) {
2        struct Node *tmp = cur;
3        cur = cur->next;
4        free(tmp);
5    }
```

This can be also done recursively:

```
1    void freeList(struct Node *lst) {
2        if (lst) {
3            freeList(lst->next);
4            free(lst);
5        }
6    }
```

12.2 Application of Vectors

12.2.1 ADT Map/Dictionary (Mutable version)

Here are several operations it has. When we are specifying these operations, we should give *preconditions* (what do I need to satisfy in order to be valid when I call this function) and *postconditions* (if we call this function, having met the preconditions, what then does the function guarantee to us) for these operations.

`make-map`: it has no parameters. Pre: true[1]. It produces a new `map`.

`map`: params: map M, key k, value v. Pre: true. It produces no value. Post: if $\exists v'$ such that $(k, v') \in M$, then $M \leftarrow M \setminus \{(k, v')\} \cup \{(k, v)\}$. else $M \leftarrow M \cup \{(k, v)\}$.

[1]this means we can always call this function because true is `true`

remove: 2 params: map M, key k. Pre true. It produces no value. Post: if $\exists v$ such that $(k, v) \in M$, $M \leftarrow M \setminus \{(k, v)\}$. Otherwise M is unchanged.

search: params: map M, key k. Pre: true. Value produced is v such that $(k, v) \in M$, otherwise something outside the value domain.

To implement these, we start by assuming keys are integers (for simplicity, we omit values).

If we use an association list, then accessing an item takes time proportional to its position in the list ($O(\texttt{len}(L))$ worst case). If we use a BST, then we do have the same worst case running time because there is no guarantee the BST will be shaped well. If we use a balanced BST (e.g., AVL tree), then $O(\log n)$ is the worst case time, where $n = |M|$, we pay for a difficult implementation.

If we use vectors instead, we have the advantage of $O(1)$ for any index-based access, but how big should the vector be? To have size of vector equal to maximum key? This will waste a lot of space. We can instead combine these two: vector of association lists, which is called a *hash table*.
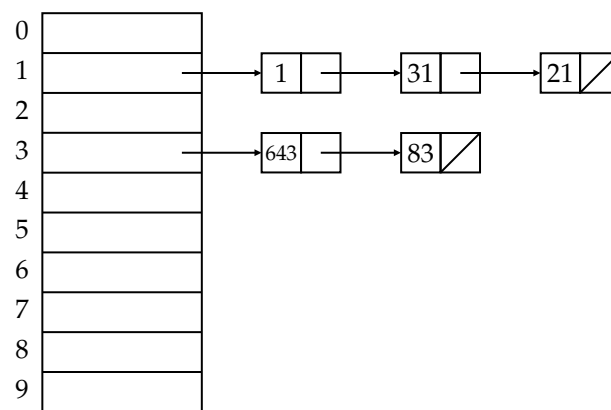

```
1   (define (create-hashtable size) (make-vector size empty))
```

To which association list should we add (k, v)? We need to map k to a vector index. Mapping called a *hash function*. For simplicity, we use remainder of k by the length of the vector. For this idea to work well, the hash function must distribute keys evenly over the indices.

```
1   (define (ht-search table key)
2       (define index (modulo key (vector-length table)))
3       (define hashlist (vector-ref table index))
4       (define lookup (assoc key hashlist))
5       (if lookup (second lookup) false))
```

Feb 25

13.1 Hash tables cont'd

Now let's do add.

```
(define (ht-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? (second lookup) val))
            (vector-set! table index
                (cons (list key val) (remove lookup hashlist))))
        (vector-set! table index (cons (list key val) hashlist))))
```

Quick note: if the keys are not numbers, then we need a hash function that maps them to numbers. Then we end up with a structure like this:
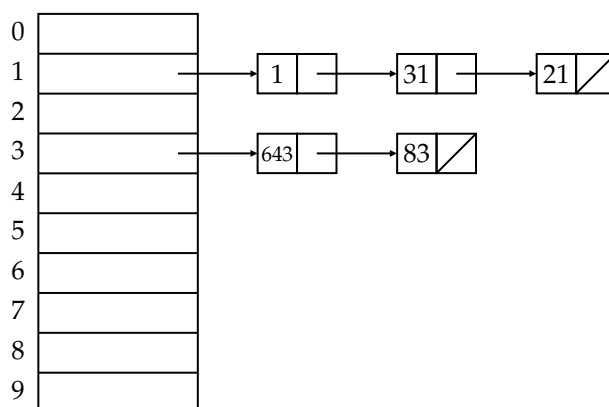
The running time to fetch values from one of these slots is going to be $O(n/\ell)$ where n is the number of items in the collection and ℓ is the length of the vector. The hope is that ℓ is somehow chosen based on the number of items that we expect to have in our table. For example, if $\ell = kn$, then the running time is $O(n/\ell) = O(n/kn) = O(1/k) = O(1)$.

13.2   ADT's in C: Sequence

First thing to note is that C doesn't have modules. C has files. We are going to implement an ADT in C, sequence. The operations:

- empty sequence

- `insert(s, i, e)`: insert e at index i in s. Pre: $0 \leq i \leq \texttt{size}(s)$.

- `size(s)`: number of elements in s. Pre: true.

- `remove(s, i)`: remove item from index i in s. Pre: $0 \leq i \leq \texttt{size}(s) - 1$.

- `index(s, i)`: return ith element of s. Pre: $0 \leq i \leq \texttt{size}(s) - 1$.

We want no limits on size. The sequence can grow as needed. Now let's talk about implementation options: *linked list*, which is easy to grow, but slow to index. The other option is *array*, which is fast to index, but hard to grow.

Our approach would be partially-filled heap array.

```
1  struct Sequence {
2      int size; // how many items in use?
3      int cap;  // how much can we hold?
4      int *theArray;
5  };
```

How do we structure this as a C module? So we would have a header file, interface, `sequence.h`:

```
1  struct Sequence {
2      int size, cap, *theArray;
3  }; // bad style, but to save space
4
5  struct Sequence emptySeq();
6  int seqSize(struct Sequence s);
7  void add(struct Sequence *s, int i, int e);
8  void remove(struct Sequence *s, int i);
9  int index(struct Sequence s, int i);
10 void freeSeq(struct Sequence s);
```

We have `freeSeq` because our `Sequence` is heap allocated. Rather than forcing the user to understand they need to `free theArray`, we provide them with `freeSeq`. Also we can see that some functions take a pointer to `struct Sequence` and the others are not. This is because when we `add` or `remove` from a `Sequence`, the size will change, then we will need to change `s`.

The implementation will go into `sequence.c` file:

```
1  #include "sequence.h"
2  /*
3      First, this includes struct definition, we then have context for anything else.
4      Second, the use of quotes (instead of <>) told us that the file is located in
5      this directory. <> standard libraries.
6  */
7
8  struct Sequence emptySeq() {
9      struct Sequence res;
10     res.size = 0;
```

```
11      res.theArray = malloc(10 * sizeof(int));
12      res.cap = 10;
13      return res;
14  }
15
16  int seqSize(struct Sequence s) {
17      return s.size;
18  }
19
20  void add(struct Sequence *s, int i, int e) {
21      for (int n = s->size; n > i; --n) {
22          s->theArray[n] = s->theArray[n-1];
23      }
24      ++s->size;
25      s->theArray[i] = e;
26  }
27
28  void remove(struct Sequence *s, int i) {
29      for (int n = i; n < s->size-1; ++n) {
30          s->theArray[n] = s->theArray[n+1];
31      }
32      --s->size;
33  }
34
35  int index(struct Sequence s, int i) {
36      return s.theArray[i];
37  }
38
39  void freeSeq(struct Sequence s) {
40      free(s.theArray);
41  }
```

Now let's see `main.c`:

```
1  int main() {
2      struct Sequence s = emptySeq();
3      add(s, 0, 4);
4      add(s, 1, 7);
5      ...
6  }
```

This is ok, but not immune to tampering and forgery. Tampering is accessing the internals of the ADT without going through the functions that provided. Forgery is building an instance of the ADT without using a constructor function that we give. Client then can do

```
1  s.size = 8; // tampering!
2
3  // forgery
4  struct Sequence t;
5  t.size = 10;
6  t.cap = 20;
7  ...
```

Can we prevent this? Qualified yes, but C is not really designed for this kind of protection like some modern languages. The idea is somehow keeping the the details of struct Sequence hidden. Can we declare, but not define, the `struct`? So in sequence.h, we just declare the `struct`:

```
1  struct Sequence;
2
3  struct Sequence emptySeq();
4  int seqSize(struct Sequence s);
5  void add(struct Sequence *s, int i, int e);
6  void remove(struct Sequence *s, int i);
7  int index(struct Sequence s, int i);
8  void freeSeq(struct Sequence s);
```

Then in sequence.c, we define `struct`:

```
1  #include "sequence.h"
2  struct Sequence {
3      int size, cap, *theArray;
4  };
5
6  // as before
```

In main.c,

```
1  #include "sequence.h"
2
3  int main() {
4      struct Sequence s = emptySeq();
5      ...
6  }
```

This won't compile. The compiler here doesn't know enough about Sequence: the compiler needs to know how big this variable it is before creating on the stack. The compiler only knows Sequence exists. However, we can provide pointers. In sequence.h,

```
1  struct SeqImpl;
2
3  typedef struct SeqImpl *Sequence; // Sequence = struct SeqImpl*
4
5  Sequence emptySeq();
6
7  void add(Sequence s, int i, int e);
8  ...
```

In main.c,

```
1  #include "sequence.h"
2
3  int main() {
4      Sequence s = emptySeq(); // s is a pointer -- OK!
5      ...
6  }
```

13.2.1   Doubling Strategy

Now what happens if the array is full?

```
void add(Sequence s, int i, int e) {
    if (s->size == s->cap) {
        // make the array bigger
        s->theArray = realloc(s->theArray, /* new size */ );
        ...
    }
    ...
}
```

`realloc` increase a block of memory to a new size. If necessary, it allocates a new, larger block and frees the old block (data copied over). The question is then how big should we make it? one larger? We must assume that ech call to `realloc` causes a copy $O(n)$. If we have a sequence of adds (at then end, so no shuffling cost), the number of steps would be

$$n \ + \ n+1 \ + \ n+2 \ + \ \cdots \ + \ n+k \ + \ \cdots$$

If done n times, $O(n^2)$ total cost, $O(n)$ per add. Similarly, two larger, or three larger don't save us much. What if, instead, we double the size? Each add still $O(n)$ worst case. But we can do *amortized analysis*: we place a bound in a sequence of operations, even if an individual operation may be expensive.

If an array has a cap of k and is empty,

- k inserts at a cost of 1 each (k steps taken).

- 1 insert cost $k+1$ - cap now $2k$ ($k+1$ steps)

- $k-1$ inserts cost 1 each ($k-1$ steps)

- 1 insert costs $2k+1$ - cap now $4k$ ($2k+1$ steps)

- $2k-1$ inserts cost 1 each ($2k-1$ steps)

- 1 insert costs $4k+1$ - cap now $8k$ ($4k+1$ steps)

- …

- 2^{j-1} inserts cost 1 each ($2^{j-1}k-1$ steps)

- 1 insert cost 2^jk+1 - cap now $2^{j+1}k$ (2^jk+1 steps taken)

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

Total steps:

$$k+(k+1)+(k-1)+(2k+1)+(2k-1)+(4k+1)+\cdots+(2^{j-1}k-1)+(2^jk+1) = 3\cdot 2^jk-k+1$$

Then number of steps per insertion:
$$\frac{3\cdot 2^jk-k+1}{2^jk+1} \approx 3$$

Therefore, doubling capacity provides for $O(1)$ amortized time insertions (at the end)

14

Mar 2

14.1 Sequence cont'd

```
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 e, int i) {
    increaseCap(s);
    ...
}
```

Another way to look at amortized situation is the cost vs. the number of operations:
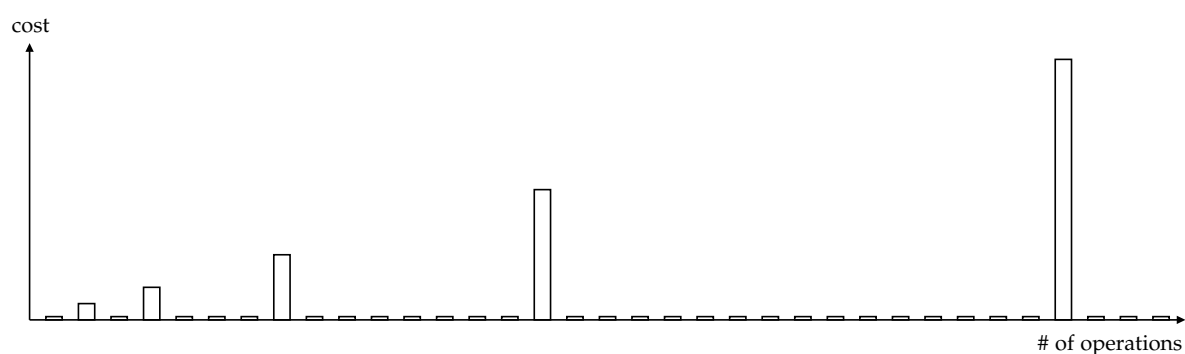
We are get exponentially worse, but also less exponentially frequent.

`increaseCap` is a helper function, and should not be called by `main`. How do we prevent it? We can just leave it out of the `.h` file. However, someone could write `main` if they know it exists:

```
#include "sequence.h"
void increaseCap(Sequence s);
...
```

What if `main` declares its own header? We can do something in `sequence.c`:

```
1  static void increaseCap(Sequence s) { ... }
```

In this context, `static` means only visible in *this* file. `static` functions/variables prevent other files from having access even if they write their own header.

14.2 Interpreting Mutation

Recall from the tutorial the deferred substitution interpreter in Haskell for Faux Racket:

$$\begin{aligned}
\mathtt{exp} = &\ \mathtt{number} \\
&|\ \mathtt{(op\ exp\ exp)} \\
&|\ \mathtt{(fun\ (id)\ exp)} \\
&|\ \mathtt{(with\ ((id\ exp))\ exp)} \\
&|\ \mathtt{(exp\ exp)} \\
&|\ \mathtt{id} \\
\mathtt{op} = &\ \mathtt{+}\ |\ \mathtt{\star}
\end{aligned}$$

Then we turn this into abstract syntax, written in Haskell:

```
1  data Op = Plus | Times
2  OpTrans Plus = (+)
3  OpTrans Times = (*)
4
5  -- An AST is either an integer, binary expression with three arguments, function
       application with two arguments, a string, or a function.
6  data Ast = Number Integer | Bin Op Ast Ast
7           | Fun String Ast | App Ast Ast | Var String
```

Note that `with` is already built with `App` and `Fun` because

$$\mathtt{(with\ ((id\ exp1))\ exp2)} \equiv \mathtt{((fun\ (id)\ exp2)\ exp1)}$$

```
1  data Val = Numb Integer
2           | Closure String Ast Env
3  type Env = [(String, Val)]
```

```
1  interp :: Ast -> Env -> Val
2  interp (Number v)_ = Numb v -- in any environment
3  interp (Fun p b) e = Closure p b e
4  interp (Bin op x y) e = Numb (opTrans op v w)
5      where
6          (Numb v) = interp x e
7          (Numb w) = interp y e
8  interp (App f x) e = interp fb ((fp, y):fe)
9      where
10          Closure fp fb fe = interp f e
11          y                = interp x e
12  interp (Var x) e = fromMaybe undefined (lookup x e)
```

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

$$
\begin{aligned}
\text{exp} = \ &\text{number} \\
&| \ \text{(op exp exp)} \\
&| \ \text{(fun (id) exp)} \\
&| \ \text{(with ((id exp)) exp)} \\
&| \ \text{(exp exp)} \\
&| \ \text{id} \\
&| \ \text{(set id exp)} \\
&| \ \text{(seq exp exp)}
\end{aligned}
$$

and

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

Note that we will implement mutation without actually using mutation. There's no mutation in Haskell.

To implement Set, basic idea would be to change the name-value biding in the environment. This needs to be done carefully. Consider

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

Should produce 11, but how? `(set x 5)` changes the environment so that x maps to 5. Then we use that environment when evaluating all that follows x, `(seq (set x 6) x)`. Then `(set x 6)` changes the environment so that x maps to 6, then use that environment when evaluating all that follows.

So each expression should return the environment that results after it is finished, so that the updated environment can be used in what follows.

But what about:

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

So need to be careful about returning environments, don't want 15!

But how would we accommodate boxes? We need a model that permits aliasing.

Idea: (Recall previous topics) Environment (*Env*) doesn't map variables to located values, but maps variables to locations. It is not threaded through the program. And they are never updated, only added to.

Store maps locations to values: values are updated; locations are not.

Then we can model aliasing: two variables map to the same location.

`interp` takes a store as an additional parameter, returns an updated store as a result.

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

```
 1  interp (Number v) _ s = (Numb v, s)
 2  interp (Fun p b) e s = (Closure p b e, s)
 3  interp (Bin op x y) e s = (Numb (opTrans op v w), s'')
 4      where
 5          (Numb v, s')  = interp x e s
 6          (Numb w, s'') = interp y e s'
 7
 8  interp (seq x y) e s = (v, s'')
 9      where
10          (_, s')  = interp x e s
11          (v,  s'') = interp y e s'
12
13  interp (Var x) e s = (fromMaybe undefined (lookup loc s), s)
14      where loc = fromMaybe undefined (lookup x e)
15
16  interp (App f x) e s = interp fb ne ns
17      where
18          (Closure fp fb fe, s') = interp f e s
19          (y, s'') = interp x e s'
20          nl = newloc s''
21          ne = (fp, nl):fe  -- new env entry, and
22          ns = (nl, y):s''  -- new store entry
23
24  interp (Set x y) e s = (Void, ns)
25      where
26          l x = fromMaybe undefined (lookup x e)
27          (nv, s') = interp y e s
28          ns = (lx, nv):s'
29
30  newloc = length -- works because we never remove a loc from the store.
```

This is impractical for long computations. So it's better to reuse old locations ("garbage collections" - later).
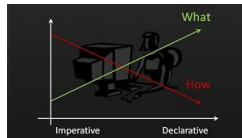
Module II:

SIMPL

This module discusses an artificial imperative language that has enough features to be able to illustrate concepts and difficulties that arise, while being simple enough to facilitate precise specification an implementation in the manner we are used to from CS 145.

15

Mar 3

15.1 Syntax

Let's build our own imperative language: SIMP (simple imperative language). It will have statements, sequencing, conditional execution, repetition. A SIMP program is a sequence of `var` declarations, initialized to integers, followed by statements.

For example,

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

The grammar will be

```
program = (vars [(id number)]
            stmt ...)

  stmt = (print aexp)
       | (print string)
       | (set id aexp)
       | (seq stmt ...)
       | (iif bexp stmt stmt) ; different from racket if, this is imperative if
       | (skip)
       | (while bexp stmt ...)

  ;; arithmetic expression
  aexp = (+|*|-|div|mod aexp aexp)
       | number
       | id

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

An easy way to implement is to use Racket macros:

```
1  (define-syntax-rule
2      (vars [(id init) ...] exp ...)
3      (let  [(id init) ...]
4          exp ...)) ;; the dots here are literal
5
6  (define-syntax-rule
7      (iif test texp fexp)
8      (if  test texp fexp))
9
10 (define-syntax-rule
11     (while test exp ...)
12         (let loop ()                  ;; named let, which sets up loop as
13             (when test                ;; a f'n with () args (i.e., no args)
14                 exp ... (loop))))) ;; with body (when ...) & invokes (loop)
```

The rest are renames of existing Racket functions.

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

Here is an example on computing all perfect numbers up to 10000. Perfect number is a number which is equal to the sum of its divisors (other than itself). To achieve this in SIMP,

```
1  (vars [(i 1) (j 0) (acc 0)]
2      (while (<= i 10000)
3          (set j 1)
4          (set acc 0)
5          (while (< j i)
6              (iif (= (mod i j) 0)
7                  (set acc (+ acc j))
8                  (skip))
9              (set j (+ j 1)))
10         (iif (= acc i)
11             (seq (print i) (print "\n"))
12             (skip))
13         (set i (+ i 1))))
```

Now consider the implementation in C:

```
1  int main() {
2      for (int i = 1; i < 10000; ++i) {
3          int acc = 0;
4          for (int j = 1; j < i; ++i) {
5              if (i % j == 0) acc += j;
6          }
7          if (acc == i) printf("%d\n", i);
8      }
9  }
```

In Racket:

```
1  (do ((i 1 (add 1 i)))
2      ((> i 10000))
3      (do ((j 1 (add1 j)
4           (acc 0 (if (zero? (remainder i j))
5                       (+ j acc)
6                       (acc)))
7          ((>= j i)))
8      (when (= acc i) (printf "~a\n" i)))))
```

In Haskell:

```
1  divisors :: Int -> [Int]
2  divisors i = [j | j <- [1..i-1], i `rem` j == 0]
3  main = print [i | i <- [1..10000], i == sum (divisors i)]
```

In Python:

```
1  print[i for i in range(1, 10000) if i==sum([j for j in range(1, i) if i%j==0])]
```

Let's then have a race:

SIMP	Racket	C	Haskell	Python
1.834s	1.361s	0.529s	2.652s	7.481s

This is purely for entertainment.

15.2   Semantics of SIMP

An imperative program is a map from state to states, where the state σ is the values of variables (ignore output).

Notation: $[x \mapsto i]\sigma$ is the state that maps x to i and any other y to $\sigma(y)$.

We then rewrite rules for pairs (π, σ) (program, state) and we will take *left innermost rule understood*, and we omit rules that leave σ unchanged, e.g., exprs, skip, (seq).

$$((\texttt{set } x \ n), \sigma) \Rightarrow (, [x \mapsto n]\sigma)$$
$$((\texttt{iif true } s_1 \ s_2), \sigma) \Rightarrow (s_1, \sigma)$$
$$((\texttt{iif false } s_1 \ s_2), \sigma) \Rightarrow (s_2, \sigma)$$
$$((\texttt{while } t \ s_1 \ \dots \ s_i), \sigma) \Rightarrow ((\texttt{iif } t \ (\texttt{seq } s_1 \ \dots \ s_i \ (\texttt{while } t \ s_1 \ \dots \ s_i)) \ (\texttt{skip})), \sigma)$$

Initial pair: (π_0, σ_0). If the program is (vars [(x1 n1) ... (xi ni)] s1 ... sj), then $\pi_0 = $ (seq s1 ... sj) and σ_0 is the function σ such that for $k = 1, \dots, i$, $\sigma(x_k) = n_k$.

16

Mar 4

16.1 SIMP Interpreter (Haskell)

```haskell
data Aop = Plus | Times | Minus | Mod | Div
aopTrans Plus  = (+)
aopTrans Times = (*)
aopTrans Minus = (-)
aopTrans Div   = div
aopTrans Mod   = mod

data Aexp = Number Integer | Var String | ABin Aop Aexp Aexp

data Bop = Lt | Gt | Le | Ge | Eq
bopTrans Lt = (<)
bopTrans Gt = (>)
bopTrans Le = (<=)
bopTrans Ge = (>=)
bopTrans Eq = (==)

data Bexp = BBin Bop Aexp Aexp
          | Not Bexp | And Bexp Bexp | Or Bexp Bexp
          | Bval Bool

data Stmt = Skip | Set String Aexp | Iif Bexp Stmt Stmt
          | Seq Stmt Stmt | While Bexp Stmt
```

We omit print for now.

```haskell
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 aes 2)

```

```
9    beval :: Bexp -> State -> Bool
10   beval (Bval b) _ = b
11   beval (Not be) s = not (beval be s)
12   beval (And be1 be2) s = (beval be1 s) && (beval be2 s)
13   beval (Or be1 be2) s  = (beval be1 s) || (beval be2 s)
14   beval (Bbin op ae1 ae2) s = bopTrans op (aeval ae1 s) (aeval ae2 s)
15
16   interp :: Stmt -> State -> State
17   interp Skip s = s
18   interp (Set x ae) s = (M.insert x $!(aeval ae s)) s
19   interp (Iif be ts fs) s = if (beval be s) then interp ts s else interp fs s
20   interp (Seq s1 s2) s = let s' = interp s 1 s in interp s2 s'
21   interp loop@(While bt body) s = interp (Iif bt (Seq body loop) Skip) s
```

16.1.1   Printing

Now to add printing: recall from module 1, we can model output as a list of chars that *would* be printed (ω - part of the state)

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

Actually use:

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

Now add print stmts to the AST:

```
1    data Stmt = ...
2              | IPrint Aexp
3              | SPrint String
4
5    interp Skip st om = (st, om)
6    interp (Set x ae) st om = ((M.insert x $!(aeval ae st)) st, om)
7    interp (Iif be ts fs) st om = if beval be st then interp ts st om else interp fs st
         om
8    interp loop@(While bt body) st om = interp (Iif bt (Seq body loop) Skip) st om
9
10   -- now sequencing
11   interp (Seq s1 s2) st om = let (st', om') = interp s1 st om in interp s2 st' om'
12
13   -- now printing
14   interp (IPrint ae) st om = (st, om ++ (show (aeval ae st)))
15   interp (Sprint s)  st om = (st, om ++ s)
```

Consider the following program:

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

If we call run = `let(_,om) interp p1 st "" in om`, the result is `"2\n4\n8\n16\n32\n64\n128\n256\n512\n1024\n"`. It is not pretty, but all information is there, the interpreter's job is done, which tells us what will be printed on the screen.

```
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 av = \om -> (av, om)

miprint :: String -> Mio ()
miprint s = \om -> ((), om ++ s)
```

Abstract the behavior of Seq:

```
interp (Seq s1 s2) st om =
    let (st', om') = interp s1 st om in interp s2 st' om' -- what we had
                    --Mio State-|

= (\t -> let (st', om') = t om in interp s2 st' om') (interp s1 st)
                              -- can't ---
                              -- pull this out
= (\t -> \u -> let (st', om') = t om in u st' om') (interp s1 st) (interp s2)
  -- give this a name --------------------------
```

New operator "bind"

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

There is a simpler version for when g doesn't need the value av

```
(>>>) :: Mio a -> Mio b -> Mio b
f >>> k = f >>> \_ -> k
```

```
interp :: Stmt -> State -> Mio State
interp Skip st = inject st
interp (Set x ae) st = inject ((M.insert x $!(aeval ae st)) st)
interp (Iif be ts fs) st
    = if (beval be st) then (interp ts st) else (interp fs st)
interp loop@(While bt body) st = interp (Iif bt (Seq body loop) Skip) st
interp (Seq s1 s2) st = interp s1 st >>>= \st' -> interp s2 st'
interp (IPrint ae) st = miprint (show (aeval ae st)) >>> inject st
interp (SPrint ae) st = miprint s >>> inject st
```

We have native Haskell approach:

$$
\begin{array}{rcl}
\texttt{Mio a} & \longrightarrow & \texttt{IO a} \\
\texttt{miprint} & \longrightarrow & \texttt{putStr} \\
\texttt{>>>=} & \longrightarrow & \texttt{>>=} \\
\texttt{>>>} & \longrightarrow & \texttt{>>} \\
\texttt{inject} & \longrightarrow & \texttt{return}
\end{array}
$$

We then change all with built-in operators. If this time we call `run = interp p1 s`, unlike before it told us what should be printed, it actually prints. This goes back to what we are talking at the beginning of the term: if a program doesn't have any side effects, what could it be? A program has to communicate the answer to the user, which is a side effect. Haskell has to deal with the question: as much as you want to be pure, if you can't talk to the user, then whatever the computing we have done, it would be lost. The way that Haskell works when it comes to output is that it builds up this specification of what the output should be and then during the runtime, it takes that specification and actually does it. This is an example of a more general concept, known as monad.

Mar 9

17.1 Monad cont'd

There's a slightly cleaner way in Haskell: "do" notation

$$e1 \texttt{ >> } e2 \longrightarrow \texttt{do } e1; \ e2$$

$$e1 \texttt{ >> } \backslash p \texttt{ -> } e2 \longrightarrow \texttt{do } p \texttt{ <- } e1 \ ; \ e2$$

For 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 s ; return st
```

This IO type is an example of a *monad*, which extends to any datatype with >>= (bind), return operators. It turns out that monads help to model otherwise impure effects purely, which abstracts away the "plumbing" that ensures effects are properly sequenced.

17.2 Proofs for Imperative Programs

Recall the Fibonacci numbers: $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2} \ (n > 1)$. In Racket,

```
(define (fib n)
    (if (<= n 1) n
        (+ (fib (- n 1)) (fib (- n 2)))))
```

Proving (fib n) = F_n: easy induction. But fib is inefficient. Better to use 2 accumulators:

```
(define (fib-h n fn fnm1)
    (cond [(zero? n) fnm1]
          [(= n 1)   fn]
          [else (fib-h (sub1 n) (+ fn fnm1) fn)]))
;; linear time (assuming constant-time arithmetic)
(define (fib n) (fib-h n 1 0))
```

To prove (fib n) = F_n, we want to show (fib-h n 1 0) = F_n. However, now consider the inductive step: (fib-h n 1 0). For $n > 1$, (fib-h (sub1 n) 1 1), which is not the same as the inductive

hypothesis. Thus we need to state and prove a more general hypothesis. For example:

(fib-h 5 1 0) $\Rightarrow$ (fib-h 4 1 1) $\Rightarrow$ (fib-h 3 2 1) $\Rightarrow$ (fib-h 2 3 2) $\Rightarrow$ (fib-h 1 5 3) $\Rightarrow$ 5

So prove: $\forall j, \forall i > 1$, if f_{jp1} is F_{j+1} and f_j is F_j, then (fib-h i f_{jp1} f_j) = F_{i+j}. This is straightforward induction on i.

What would this look like in SIMP? We set n initially and mutate (no functions)

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

It is wrong because fjm1 updated prematurely. If we swap (set fj (+ fj fjm1)) and set fjm1 fj, it is still wrong, fj updated prematurely.

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

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

The statement ans = F_n: true or false? Depends on the state at the time the statement is evaluated.

17.3   Hoare Logic

Prove triples of the form $\{P\}$ statement $\{Q\}$, known as "Hoare triples".[1]

- P: precondition. Logical statement about the state of the program before "statement" runs.

- Q: postcondition. Logical statement about the state of the program after "statement" runs.

The interpretation: "If P is true before the statement runs, then Q is true after statement runs."

To conclude $\{P\}$ (vars [(x1 v1) ...   (xn vn)] stmt ...) $\{Q\}$, it suffices to show

$$\{P \wedge x_1 = v_1 \wedge x_2 = v_2 \wedge \cdots \wedge x_n = v_n\} \text{ (seq stmt ...) } \{Q\}$$

To conclude $\{P\}$ (seq stmt1 stmt2) $\{Q\}$, it suffices to find an stmt R such that $\{P\}$ stmt1 $\{R\}$ and $\{R\}$ stmt2 $\{Q\}$.

---

[1] Note that the notation is different from CS 245: $(\!|P|\!)\,C\,(\!|Q|\!)$

Finding R can be tricky, may need to adjust P and Q to get an R that works.

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

To conclude $\{P\}$ (set x exp) $\{Q\}$, note that P and 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 be true about exp before the stmt. So

$$\{Q[\text{exp}/\text{x}]\} \text{ (set x exp) } \{Q\}$$

where $Q[\text{exp}/\text{x}]$ means Q, with exp substituted for x.

To conclude $\{P\}$ (iif B stmt1 stmt2) $\{Q\}$, suffices to show $\{P \wedge B\}$ stmt1 $\{Q\}$ or $\{P \wedge \neg B\}$ stmt2 $\{Q\}$.

(While B stmt ...) is the trickiest. If the loop doesn't run: $\{P\}$ (While B stmt ...) $\{P\}$. But wether the loop runs or not, B must be false: $\{P\}$ (While B stmt ...) $\{P \wedge \neg B\}$. If the loop repeats, whether was true at the end is true at the beginning and B is true. The body of the loop should look like this (in the triple):

$$\{P \wedge B\} \text{ (seq s1 ... sn) } \{P\}$$

So P is preserved by the body of the loop, known as a *loop invariant*. Finding the right loop invariant can be tricky. Our task now is to prove the the fib program (simplified) works.

```
1   (vars [(n 10) (fj 1) (fjm1 0) (t 0)]
2       (while (not (= n 1))
3               (set t fj)
4               (set fj (+ fj fjm1))
5               (set fjm1 t)
6               (set n (- n 1))))
7               ; fj = F(10) -- goal
```

Now let's do the proof!

```
1   (vars [(n 10) (fj 1) (fjm1 0) (t 0)]
2       ; n = 10, fj = F(1), fjm1 = F(0)
3       ; fj = F(11-n), fjm1 = F(10-n)
4       (while (not (= n 1))
5               ; fj + fjm1 = F(12-n), fj = F(11-n)
6                   ; -- not quite true invariant, but is implied by it (*)
7               (set t fj)
8               ; fj + fjm1 = F(12-n), t = F(11-n)
9               (set fj (+ fj fjm1))
10              ; fj = F(12-n), t = F(11-n)
11              (set fjm1 t)
12              ; fj = F(12-n), fjm1 = F(11-n)
13              (set n (- n 1)))
14              ; fj = F(11-n), fjm1 = F(10-n)
15      )
16      ; fj = F(11-n), fjm1 = F(10-n), n = 1
17      ; fj = F(10) -- goal
```

What's the right invariant? $f_j = F(11 - n), f_{jm1} = F(10 - n)$

(*): $f_j = F(11 - n), f_{jm1} = F(10 - n) \implies f_j + f_{jm1} = F(11 - n) + F(10 - n) = F(12 - n)$

So we are done! We have proved *partial correctness*: correctness under the assumption of termination. *Total correctness* includes a proof of termination.
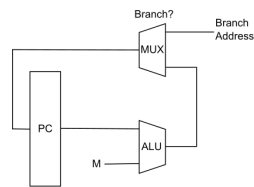
Module III:

PRIMPL

PRIMP Machine Language, A-PRIMP Assembly Language, Assembler from A-PRIMP to PRIMP, Compiler from SIMP to A-PRIMP, Variation of SIMP. The subtitle of this module could be "SIMPL is all lies".

Mar 10

18.1 PRIMPL basics

So far, the abstract syntax for SIMPL is stored in s-expressions, basically tree-structured data. But trees are an abstraction, basic machine has only memory, not tress. Let us now store our program as a *vector*. Now a program is a sequence of instructions. The RAM used for data will be the same RAM that holds the program.

This is going to change how we reason about the program: no more rewriting, because program remains fixed. As we are walking through what the program is doing, we can't rely on the program changes with every step, instead, we need to keep track of where we are in the program.

We have index into the program, called the *Program Counter* (PC). It holds the location of the next statement (instruction) to be executed.

If we build a simulator for this, it runs a *fetch-execute cycle*, which repeats the following things forever:

- fetch the instruction at the location given by PC;

- increment PC;

- execute the fetched instruction.

For the sake of clarity, we imagine PC is a separate variable, outside the RAM that holds our program.

We will have no named variables, because locations in RAM don't have names. Variables are referenced by location; for our purposes, variables start right after the program. We are still going to allow unbounded inters (will fix later).

In SIMPL, we can have arbitrarily complex expressions. In PRIMPL (primitive imperative language), were are only allowed to do one operation at a time, so we are looking for constant-space instructions.

An arithmetic statement has 3 arguments: 2 operands and a destination. For operands, we want to allow both numbers 2 and locations (2), where 2 is the number 2, and (2) is the value at location with index 2.

For example, (add (5) (1) (2)) means "$M[5] \leftarrow M[1] + M[2]$" where M is the big array of memory. For (add (7) (7) 1), it means "$M[7] \leftarrow M[7] + 1$".

Operations allowed: add sub mul div mod equal not-equal gt ge le and logical operators: land lor lnot.

PRIMPL vector entries can be either integer or boolean.

We will also have move: for example, (move (10) (12)): "$M[10] \leftarrow M[12]$"

Let's talk about *loops & ifs*. Programs are no longer tree-structured, then how will we know where to go? We then need to replace all loops and ifs with unconditional jump and conditional branch. The unconditional jump: (jump 12) means "$PC \leftarrow 12$". The conditional branch: (branch (20) 12) means "if $M[20]$ then $PC \leftarrow 12$".

In terms of *printing*, we have (print-val 21) prints 21. (print-val (15)) prints $M[15]$. We also allow (print-string "\n"), which prints "\n".

And that's it! Let's consider an example what programming in this language would look like. We start with a program in SIMPL:

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

The equivalent PRIMPL program:

```
0  (gt (11) (9) 0)       ; tmp1 <- (x > 0)
1  (branch (11) 3)       ; if tmp1 goto 3
2  (jump 8)              ; goto 8
3  (mul (10) 2 (10))     ; y <- 2 * y
4  (sub (9) (9) 1)       ; x <- x - 1
5  (print-val (10))      ; print y
6  (print-string "\n")   ; print "\n"
7  (jump 0)              ; goto 0
8  0                     ; 0 [number halts program]
9  10                    ; x
10 1                     ; y
11 0                     ; tmp1
```

18.2 PRIMPL Simulator

So we want this then we can run PRIMPL programs. The first we need is a memory.

```
1  (define mem (make-vector MEM_SIZE))
2  (define pc 0)
3  (define halted? #f)
4
5  (define (mem-get i) (vector-ref mem i))
6  (define (mem-set! i new) (vector-set! mem i new))
7
8  (define (load-primpl prog-list)
9      (set! halted? #f)
10     (vector-fill! mem 0)
11     (for [(i MEM_SIZE) (c (in-list prog-list))]
12         (mem-set! i c)))
13
```

```
14    (define (run-primpl)
15        (let loop ()
16            (unless halted?
17                (fetch-executed-once)
18                (loop))))
19
20    (define (fetch-execute-once)
21        (define inst (mem-get pc))
22        (cond [(list? inst) (set! pc (+ pc 1)) (dispatch-inst inst)]
23              [else (set! halted? #t)]))
```

Now let's implement various instructions.

```
1    (define (print-string s) (printf "~a" s))
2
3    (define (print-val op)
4        (define val (get-op-imm-or-mem op))
5        (printf "~a" val))
6
7    ;; a helper function
8    (define (get-op-imm-or-mem op)
9        (match op
10           [(? number? v) v]
11           [`(,i) (mem-get i)]
12           [x (primpl-error "Bad operand: ~a\n" op)]))
```

Similar for `get-op-mem`, `set-dest!`

```
1    (define (get-op-mem op)
2        (match op
3            [`(,i) (mem-get i)]
4            [`(,i (,j)) (mem-get (+ i (mem-get j)))]
5            [x (primp-error "Bad operand: ~a\n" op)]))
6
7    (define (set-dest! op v)
8        (match op
9            [`(,i) (mem-set! i v)]
10           [`(,i (,j)) (mem-set! (+ i (mem-get j)) v)]
11           [x (primp-error "Bad destination: ~a\n" op)]))
```

For arithmetic operations,

```
1    (define ((bin-num-op op) dest src1 src2)
2        (define opnd1 (get-op-imm-or-mem src1))
3        (define opnd2 (get-op-imm-or-mem src2))
4        (unless (number? opnd1)
5            (error 'PRIMP "First arithmetic operand not number: ~a ~a" opnd1 opnd2))
6        (unless (number? opnd2)
7            (error 'PRIMP "Second arithmetic operand not number: ~a ~a" opnd1 opnd2))
8        (set-dest! dest (op opnd1 opnd2)))
9
10   (define add (bin-num-op +))
11   (define sub (bin-num-op -))
```

19

Mar 11

19.1 Simulator cont'd

```
1  (define (move dest src)
2      (set-dest! dest (get-op-imm-or-mem src)))
3
4  (define (jump val) (set! pc val))
5
6  (define (branch opnd loc)
7      (define tested (get-op-mem opnd))
8      (when tested (set! pc loc)))
```

Now we have implementations for every operation. Then let's write a dispatch table:

```
1   (define dispatch-table
2       (hash 'print-val printval
3             'print-string print-string
4             'add add
5             'sub sub
6             'mul mul
7             'div div
8             'mod mod
9             'equal equal
10            'not-equal not-equal
11            'gt gt
12            'ge ge
13            'lt lt
14            'le le
15            'land land
16            'lor lor
17            'lnot lnot
18            'move move
19            'jump jump
20            'branch branch))
```

Now let's dispatch:

```
1  (define (dispatch-inst inst)
2      (apply (hash-ref dispatch-table (first inst))
3             (rest inst)))
```

That's the entirety of the PRIMPL simulator.

Now let's see an example program:

```
1   (define test-prog '((gt (11) (9) 0)
2                       (branch (11) 3)
3                       (jump 8)
4                       (mul (10) 2 (10))
5                       (sub (9) (9) 1)
6                       (print-val (10))
7                       (print-string "\n")
8                       (jump 0)
9                       0
10                      10
11                      1
12                      0))
13
14  (load-primpl test-prog)
15  (run-primpl)
```

How can we make PRIMPL programming easier? Having loop back? We can't invent new instructions: what we have got is what we are stuck with. What can we do? We would like to add some "nice features" to PRIMPL: we can create a higher-level language, A-PRIMPL. Then we create a translator from A-PRIMPL to PRIMPL, known as *assembler*[1]. A-PRIMPL adds *pseudo-instructions*, which are shorthand for ordinary PRIMPL.

Here are some examples of pseudo-instructions:

- (lit 4): insert the value 4 here, just for readability.

- (halt): stop the program, just produces 0.

- (const NAME 6): creates a symbol NAME with value 6. This does not generate an entry in the PRIMPL array. It says: "replace all occurrences of NAME with 6".

Here let's take a brief aside and talk about C:

```
1  #define SIZE 100    // preprocessor directive
2  int a[SIZE];        // replace with 100
```

And we can do very crazy things with #define (not advocated), because it's really text substitution:

```
1  #define ever (;;)
2  for ever { ... }
```

It's fairly popular among C programmers to use #define, but the idea behind is that C is old, and everything was expensive. To have a constant value, the lookup of constant value as program runs would be expansive. So instead they have this directive: every time we see SIZE, just substitute it now. Generally speaking now, the use of #define like this is considered to be a bad idea, because this is not an actual variable, then we can't take its address and so on. Modern C has const declarations which

---

[1]this is a slightly different terminology from the one in the real world. A real world assembler will take a higher-level of machine language, make it more primitive, and take a *second step* to turn it into binary code.

cover most of its situations, and it has compiler smart enough to make it still fast. Apart from few uses, this is largely obsolete.

Now let's go back to A-PRIMPL.

- (const A 5)
  (const B A)

- (label NAME): generates no entry in the PRIMPL array. Symbol NAME is the location in the vector when the next instruction would be placed. Then we can use as target for branch/jump.

- (data NAME 1 2 3 4), fills a sequence of array locations with 1 2 3 4. NAME represents the index of the first location.

- (data NAME (10 1)), is equivalent to (data NAME 1 1 1 1 1 1 1 1 1 1).

NAME can be used as an indirect operand. For example, assume NAME is 40, then (add NAME NAME 1) is translated to (add (40) (40) 1).

```
50  #| 1  |# (data X 1) ; Assume stored at location 50
51  #| 50 |# (data Y X) ; Assume stored at location 51 -- No deref!
```

Now let's write the previous program in A-PRIMPL.

```
0   (label LOOP_TOP)              ; loop-top
1   (gt TMP1 X 0)                 ; tmp1 <- (x > 0)
2   (branch TMP1 LOOP_CONT)       ;     if tmp 1 goto loop_cont
3   (jump LOOP_DONE)              ;     goto loop_done
4   (label LOOP_CONT)             ; loop-cont:
5   (mul Y 2 Y)                   ;     y <- 2 * y
6   (sub X X 1)                   ;     x <- x - 1
7   (print-val Y)                 ;     print y
8   (print-string "\n")           ;     print "\n"
9   (jump LOOP_TOP)               ;     goto loop_top
10  (label LOOP_DONE)             ; loop-done:
11  (halt)                        ;     STOP
12  (data X 10)
13  (data Y 1)
14  (data TMP 0)
```

Now this has to be translated out by an assembler, which generally is in 2 phases:

1. Build a hash table of all symbols as keys. Value may not ye be known. For example,

```
(const A B)
(const B 10)
```

2. Fill in the values. Watch out for circular references, which is an error:

```
(const A B)
(const B C)
(const C A)
```

Then produce the PRIMPL program.

19.2 Converting SIMPL into A-PRIMPL

is a tool commonly known as a *compiler*. Consider a SIMPL program (vars [(x 3)] (print x)),
A-PRIMPL program would be

```
0  (print-val x)
1  (halt)
2  (data x 3)
```

20

Mar 17

20.1 Converting SIMPL into A-PRIMPL

is a tool commonly known as a *compiler*. Consider a SIMPL program (vars [(x 3)] (print x)),
A-PRIMPL program would be

```
0  (print-val x)
1  (halt)
2  (data x 3)
```

We want to make sure we never invent a var/label called x. As a convention, we will *prefix* SIMPL
vars with _:

```
1  (vars [(x 3)]
2      (print x))
```

will become

```
0  (print-val _x)
1  (halt)
2  (data _x 3)
```

In general, we have the SIMPL program

```
1  (vars [(x1 n1) ... ]
2      stmt ...)
```

and the result of compilation:

```
0  ;
1  ; code for
2  ;   stmt ...
3  (halt)
4  ...
5  (data xi ni)
6  ...
7  ;                   ____
8  ; temp storage      || stack
9  ;                   \/
```

Why temp storage? Consider `(+ exp1 exp2)`, how do we turn it into PRIMPL code? We would recursively emit code to compute `exp1` and `exp2`, then add. We need to store the first value somewhere while computing the second. After summing, temp storage is no longer needed.

There are several ways to achieve that. We could just create a separate location for every possible temporary, but that's a waste of space. A better option would be to use a *stack*. We use a data stmt to create a var SP (stack pointer) that holds the location of the first unused spot in the stack.

How do we put something on the stack? We might try something like `(move SP 5)`, but this doesn't work: it stores 5 in SP, not on the stack. Or we might want `(move (SP) 5)`? This is not (currently) legal.

What if we want to fetch a value? We might do

```
(move A (-1 SP))
(move B (-2 SP))
```

which are currently legal.

Let's add to PRIMPL (cheating a bit).

$$(\texttt{offset Label}) \equiv M[\texttt{offset} + M[\texttt{Label}]]$$

For example,

$$(\texttt{add (10) (-1 (59)) 1}) \quad \equiv \quad M[10] \leftarrow M[-1 + M[\underset{\underset{\text{top item on stack if SP = 59}}{\uparrow}}{59}]] + 1$$

Or we could do `(move (-1 (59)) 0)`, which replaces top stack item with 0. To push 0, we can do

```
(move (0 (59)) 0)
(add (59) (59) 1)
```

Using SP:

```
(move (0 (SP)) 0)
(add (SP) (SP) 1)
```

To compile `(set var exp)`, first compile the code to compute `exp`, then we need to code that pops the value on top of the stack, and moves it to the location associated with var.

Now consider `(iff exp stmt1 stmt2)`. We need code that evaluates `exp`, depending what we get from this, we need `branch`: `(branch dest LABEL0)`, where `dest` is the location where the value computed by `exp-code` is stored. Then it is followed by `jumps` and code for `stmts`. The pseudocode is as follows:

```
exp-code
(branch dest LABEL0)
(jump LABEL1)
(label LABEL0)
stmt1-code
(jump LABEL2)
(label LABEL1)
stmt2-code
(label LABEL2)
```

How about `(while exp stmt ...)`? For `while`, of course, we have to potentially do the entire piece of code over and over again. Let's first label the top, the code for `exp`, then `branch`, then code for `stmts`, then `jump` to the top. The pseudocode is as follows:

```
0  (label LABEL0)
1  exp-code
2  (branch dest LABEL1)
3  (jump LABEL2)
4  (label LABEL1)
5  stmts-code
6  (jump LABEL0)
7  (label LABEL2)
```

So these are the basic ideas of how we could turn a SIMPL program into a A-PRIMPL program, and we also have the assembler which will turn A-PRIMPL program into a PRIMPL program. By putting these two together, we could go from SIMPL to PRIMPL.

Now let's talk about how we might add more interesting things to SIMPL and how those things get translated, then we can have more sophisticated programming languages that still turn into PRIMPL, which gives us the ability to write code in better ways.

20.2   Adding Arrays to SIMPL

Let's start with

```
1  (vars [(sum 0) (A (array 1 2 3 4 5))]
2      (while (< i 5)
3          (set sum (+ sum (array-ref A i)))
4          (set i (+ i 1)))
5      (print sum))
```

How do we translate (array-ref A i) into PRIMPL? We need to add i to the address of the zeroth element of A, then fetch from that address in memory. This should remind us in C, we have *(A+i). Similarly for array-set. So we will do it like

```
0   (label LOOP_TOP)
1   (lt TMP1 _i 5)
2   (branch TMP1 LOOP_CONT)
3   (jump LOOP_DONE)
4   (label LOOP_CONT)
5   (add _sum _sum (_A _i))
6   (add _i _i 1)
7   (jump LOOP_TOP)
8   (label LOOP_DONE)
9   (print_val _sum)
10  (halt)
11  (data _sum 0)
12  (data _i 0)
13  (data TMP1 0)
14  (data _A 1 2 3 4 5)
```

Can we signal an error if an array reference is out of bounds?

$$(A \ (array \ 1 \ 2 \ 3 \ 4 \ 5)) \quad \Longrightarrow \quad \begin{array}{l} \text{(data SIZE\_A 5)} \\ \text{(data \_A 1 2 3 4 5)} \end{array}$$

Then (array-ref A I) becomes a bit more complicated:

```
0  (lt TMP0 _i 0)
1  (branch TMP0 INDEX_ERROR)
2  (ge TMP0 _i SIZE_A)
3  (branch TMP0 INDEX_ERROR)
4  (move TMP0 (_A _i))
```

Similar for (`array-set A i j`). The code is safer, but less efficient. The use of checks for every single time of the loop would be superfluous, here (in our example) because the loop bounds guarantee that all array accesses are safe.

21

Mar 18

21.1 Strings in C

Unlike Racket, C has no string type. Strings in C are arrays of characters, which might like

```
char myString [] = "Hello";
```

Recall in C, the size of an array is not stored as part of the array. Again, if this array is passed as a parameter to some function,

```
int f (char arr [] { ... })
```

it's passed as pointer, so there's no indication of size.

The convention is `char` arrays that denote strings end with the character `'\0'`, which is ASCII 0. It signals the end of a string. So what's behind the last example is

```
char myString [] = "Hello"; // 'H', 'e', 'l', 'l', 'o', '\0'
printf("%d\n", sizeof(myString)); // 6
```

If we look at this on the stack,

Stack:

myString	H
	e
	l
	l
	o
	\0

Be careful:

```
char *yourString = "Hello";
printf("%d\n", sizeof(yourString)); // 8, which is a pointer
```

In the stack, it looks like
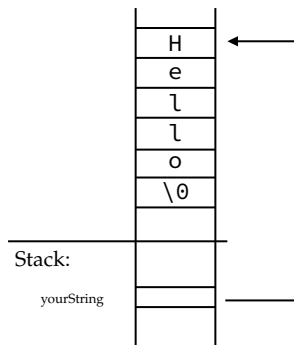
Where is `"Hello"` stored in this case? In static area, specifically "literal pool". It's often the case that they are stored in the read-only memory. Therefore,

```
myString[0] = 'h';   // OK
yourString[0] = 'h'; // UB
```

So a good practice is to have

```
const char *yourString = "Hello";
```

To print strings, we can do `printf("%s", myString)` which keeps printing characters from `myString` until it hits `\0` (not printed). `printf("myString")` is dangerous if `myString` contains `%`.

There are other string manipulations, which we get access to them by including `<string.h>`:

```
#include <string.h>
char s[] = "Hello, world!";
char t[14];
printf("%d\n", strlen(s)); // 13
strcpy(t, s);              // copy s into t
```

In fact, we can implement `strlen` if we like:

```
size_t strlen(const char *s) {
    size_t length = 0;
    while (*s++) ++length;
    return length;
}
```

We can also implement `strcpy` if we don't like:

```
char *strcpy(char *target, const char *source) {
    char *tmp = target;
    while (*tmp++ = *source++);
    return target;
}
```

Here is something we can do:

```
char s[14] = "Hello"; // more room than needed
strcat(s, " world");  // concatenate
```

Always make sure the target has enough space to hold the string.

We want also to compare strings:

```
1  char *s;
2  ...
3  if (s === "hello") { // this is pointer equality, which might not be intended.
4      ...
5  }
6
7  ...
8
9  if (!strcmp(s, "hello")) { // correct
10     ...
11 }
```

Here

$$\texttt{strcmp(s,t) returns} \begin{cases} 0 & \text{if } s = t \\ \text{a negative number} & \text{if } s < t \\ \text{a positive number} & \text{if } s > t \end{cases}$$

where the string comparison is by lexicographic order.

Once again, for strcpy, strcat, make sure the target string holds enough space! these do not allocate space.

Consider

```
1  char s[7] = "abc"; // 'a', 'b', 'c', '\0'
2  // room for two 'abc'.
3  strcpy(s+3, s); // what happens?
```

This is wrong. s[3] used to be \0, but gets overwritten by a. Then s keeps growing, strcpy won't stop until program crashes.

We also want to read strings:

```
1  char name[20];
2  printf("What is your name? ");
3  scanf("%s", name); // no & here because already a pointer
```

OK, but what if we type more than 19 letters? This is a big source of *buffer overflows*.

Consider a classic example of banking:

```
1  int main(void) {
2      struct {
3          char command[8];
4          int balance;
5      } s;
6      /* Now compiler is smarter. If we declare command first and balance second
         outside the struct, the compiler would put balance first and command second. So
         when the program overruns, it won't touch the balance. The struct forces the
         order here. */
7      s.balance = 0;
8
9      while (1) {
10         printf("Command? ('balance', 'deposit', or 'q' to quit): ");
11         scanf("%s", s.command);
```

```
12          if (!strcmp(s.command, "balance")) {
13              printf("Your balance is: %d\n", s.balance);
14          } else if (!strcmp(s.command, "deposit")) {
15              printf("Enter your deposit amount: ");
16              int dep;
17              scanf("%d", &dep);
18              s.balance += dep;
19          } else if (!strcmp(s.command, "q")) {
20              printf("Bye!\n");
21              break;
22          } else {
23              printf("Invalid command. Please try again.\n");
24          }
25      }
26  }
```

If we do a sequence of commands like:

```
1   > balance
2   0
3   > deposit, 100
4   > balance
5   100
6   > bradwantsmoney
7   Invalid command. Please try again.
8   > balance
9   1852796275
10  > q
11  Bye!
12  *** stack smashing detected ***: terminated
13  Aborted (core dumped)
```

The big lesson here: Never use scanf with %s. Instead we could do scanf("%19s", name), which will not read more than 19 characters, safer. Similarly, there are safer versions of strcpy, strcat: strncpy, strncat, strncmp. strncpy(target, source, n) copies source into target until \0 is encountered, or n characters have been copied. Note that if strncpy stops because n characters have been copied, i.e., source is longer than n, it does not add a null terminator, so we have to add it ourselves. For example,

```
1   strncpy(target, source, 10);
2   target[10] = 0;
```

Careful that

```
1   char msg[6] = "hello";
2   char msg[15] = "hello"; // ok, filled with \0 's
3   char msg[5] = "hello";  // no null terminator! Trouble awaits.
4   char msg[4] = "hello";  // truncated (probably a warning). No null terminator.
```

22

Mar 23

22.1 Adding Functions to SIMPL

The first thing to do is to invent new syntax:

```
1  (fun (f x y)
2      (vars [(i 10) (j 10)]
3          (set i (+ (* j x) y))
4          (return (* i i)))))
```

and we gonna call this language SIMPL-F.

Now we are going to follow C model, distinguished function `main` (no parameters) starts the program. The question is how do we turn it into PRIMPL? Or how do we compile this to PRIMPL?

A big problem is function return: where do we go back? We need to return to where we left off. In other words, we need to save PC somewhere and restore it when it's time to return.

However, no current PRIMPL instruction lets us save PC. We need something new. We sort of cheating again, as the instruction is already there, but we don't know about it. The instruction: (`jsr (50) 10`) means "jump to subroutine", which is equivalent to $M[50] \leftarrow PC; PC \leftarrow 10$.

Quick note on a reminder of how fetch-execute cycle works: PC has already been incremented, so this saves the address of the *next* instruction, which is what we want. For function return: (`jump (66)`) $\equiv PC \leftarrow M[50]$.

What about arguments and results? Well, there are a couple of ways that this can be done. We will start a simple solution, which isn't the best. A simple method would be to have dedicated location for every function & result. For example, if we have a function

```
1  (fun (f x y)
2      (var [] return (+ x y)))
```

then we might turn it into

```
0  (label START_f)
1  (add RETURN_VALUE_f ARG_f_x ARG_f_y)
2  (jump RETURN_ADDR_f)
3  (data RETURN_ADDR_f 0)
4  (data RETURN_VALUE_f 0)
```

```
5  (data ARG_f_x 0)
6  (data ARG_f_y 0)
```

For the function call (f exp1 exp2), we do

```
0  exp1_code
1  (move ARG_f_x TMP1)
2  exp2_code
3  (move ARG_f_y TMP2)
4  (jsr RETURN_ADDR_f START_f)
5  (move TMP3 RETURN_VALUE_f)
```

This is the model followed by old FORTRAN & COBOL compilers. This does not permit recursion: direct or mutual, because the dedicated spaces would be overwritten. Umm... Recursion would be nice to have... Each active function *call* (not just function) needs space to store arguments, locals, return address, temporaries, which leads us to use stack, and all these stuff constitutes the stack frame.

As temporaries come and go (i.e., are pushed and popped), SP changes. As a result, accessing arguments and locals by offset from SP (as we have done) becomes difficult, as offset constantly change.

Here is a common solution to this problem: *frame pointer*,[1] which points to the other end of the stack frame. It has a nice property that it doesn't move. Now let's take a look at how our memory would diagram out then.
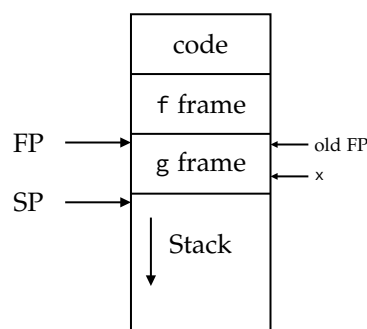
Offsets from FP will not change, even if SP moves. So previous value of FP must be also saved on the stack, which is part of g's frame.

To *allocate* a stack frame, we add the amount of space needed to the stack pointer. To *deallocate*, we subtract this amount from the stack pointer. One of the lesson we learned from this: function calls do have a cost, function calls do incur overhead.

Let's consider the following function:

```
1  (fun (fact-h n acc)
2      (vars []
3          (iif (= n 0)
4              (return acc)
5              (return (fact-h (- n 1) (* n acc))))))
```

which is tail-recursive.

With a tail recursive function, the recursion is the last thing that the function is going to do. Hence the current stack frame is longer needed, thus it can be reused, rather than create a new one. This leaves us no space overhead.

---

[1] same thing as base pointer in ⓘⓃⓣⒺⓛ assembly

We can have similar optimization for any tail call (any function call in tail position).

Many imperative language compilers don't do this optimization, while functional language compilers largely do.

We can also do some further enhancements: Consider *global mutable variables*. They are not difficult, but how do we initialize them? It's easy if by constants. If initialized by expressions, we need to consider the case if these expressions have side effects, then the order of initialization matters. Also declaration before use is something to consider.

22.2   Adding both Arrays & Functions to SIMPL

This language is called SIMPL-FA. This leads us to

- arrays as local functions variables, which we can store on the stack.

- return a local array (as a pointer) from a function, which would be a dangling pointer.

- pass arrays as arguments, where we can pass pointer to first item.

All of them, we can use C as an example for how we would make this work.
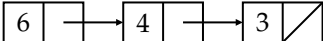
Consider an array (A (make-array 5 0)), we can turn this into

```
500   (data _A _CONTENTS_A)    ; contains 502
501   (data SIZE_A 5)
502   (data CONTENTS_A (5 0))
```

So passing array A as a parameter P, P will be 502, which just like passing anything else. Can we return an entire array from a function? Later.

Mar 24

23.1 Lists in PRIMPL

For the sake of simplicity, let's talk about lists of integers: 6 → 4 → 3

To build something like this in A-PRIMPL is quite simple:

```
(const EMPTY 0)
(data NODE3 6 NODE2) ; don't need
(data NODE2 4 NODE1) ; to be consecutive
(data NODE1 3 EMPTY) ;
```

Let's sum the numbers in a list:

```
(move P LIST)
(move SUM 0)
(label LOOP)
(eq TMP0 P EMPTY)  ; or equal
(branch TMP0 DONE)
(add SUM SUM P)
(move P (1 P))
(jump LOOP)
(label DONE)
```

Equivalently in C, we have

```
struct Node *p = lst;
int sum = 0;

while (p) {
    sum t = p->data;
    p = p->next;
}
```

How do we create new lists at run-time in PRIMPL? How do we write cons? We need to implement the heap.

A simple strategy would be that we can set aside a large chunk of memory, combined with a pointer

to the next unallocated location, called "allocation pointer". Then to do `cons`, we advance allocation pointer by 2 locations, return the first location in the pair, and fill it with first & rest. What happens when we run out of heap space? maybe some old `cons` cells are no longer needed. For example, imagine we have a list `'(6 4 3)`, but we want the `'(6 3)`,
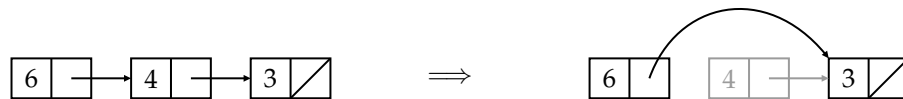
In C, it would look like `lst->next = lst->next->next;`

What happens to the 4 node? It can be freed. How to free then?

One approach is to keep a "free" list. When a `cons` is no longer needed, add to free list. So we change allocation strategy a bit: take from the free list if possible; else go to the rest of the heap.

How do we know a `cons` cell is no longer needed? The easiest way is the programmer `frees` it, but burden on the programmer, and doesn't work well with functional programming. The other way is to use garbage collection.

The idea of free list works well if we only allocate `cons` cells, i.e., all the same size. What if we want different sizes? Some searches need to be done.

What if we want lists of lists? There's no way to tell numbers from locations. Is the data field an `int`, or a pointer to `cons` cell? One way is to make a `cons` cell 3 locations: | type | first | rest |
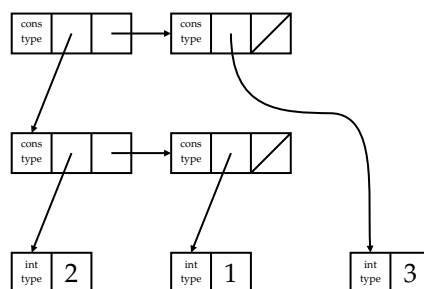For example,
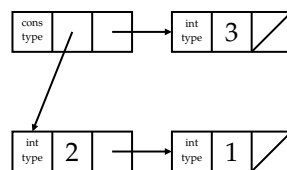
To realize, we can do

```
(const INT_TYPE 0)
(const CONS_TYPE 1)
(data NODE4 CONS_TYPE NODE2 NODE3)
(data NODE3  INT_TYPE   3   EMPTY)
(data NODE2  INT_TYPE   2   NODE1)
(data NODE1  INT_TYPE   1   EMPTY)
```

More generally, we can tag *all* values with their types (Racket). As in the previous example, we can do

MMIX

Module IV:

MMIX

MMIX is a computer intended to illustrate machine-level aspects of programming. In my books The Art of Computer Programming, it replaces MIX, the 1960s-style machine that formerly played such a role... I strove to design MMIX so that its machine language would be simple, elegant, and easy to learn. At the same time I was careful to include all of the complexities needed to achieve high performance in practice, so that MMIX could in principle be built and even perhaps be competitive with some of the fastest general-purpose computers in the marketplace.

Knuth, Donald E.

Why is PRIMPL isn't realistic? Locations are a fixed size (32 bits or 64 bits). No matter what the PRIMPL vector holds, it *must* fit in 32 bits. We need encoding schemes for numbers, for instructions, etc. These *will* overlap. The only way to tell what the word represents is by context.

Let's first introduce some terminology.
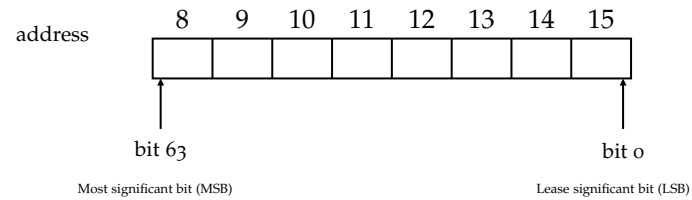
Bits are grouped into 8 *bytes* of 8 bits each:

- 7 : 0 (bits 7 - 0) is the least significant byte.

- 63 : 56 is the most significant byte.

Each byte has an address. The address of a word is the address of its most significant byte. Addresses increase left-to-right, called a big-endian architecture (other direction is little-endian).[1] Therefore, the addresses of words are always divisible by 8.

---

[1]Check https://en.wikipedia.org/wiki/Endianness#Etymology for etymology.

24

Mar 25

24.1 MMIX cont'd

Thus the numbers represented in binary is $\sum_{i=0}^{63} a_i 2^i$, which gives us numbers $[0, 2^{64} - 1]$. This only gives nonnegative numbers. There are a couple of ways to work with negative numbers. In MMIX, this is done using 2's complement notations, which we might have seen in CS 145. The idea: for positive, 0 padded on the left; for negative, 1 padded on the left. We have

$$0 \cdots 00 = 0$$
$$1 \cdots 11 = -1$$
$$1 \cdots 10 = -2$$

To negate a 2's complement number, flip the bits and add 1. For example,

$$5 = 000 \cdots 0101$$
$$-5 = 111 \cdots 1010 + 1 = 111 \cdots 1011$$

If the first bit is 1, is that a negative number or a large positive number? For our purposes, it represents a negative number. The answer in C is "we choose":

```
1  int x;         // -2^31 ... 2^31 - 1
2  unsigned int x; // 0 ... 2^32 - 1
```

Why PRIMPL isn't realistic? Large memories are slow. Most instructions typically run in 1 or 2 (clock) cycles. Memory access can take 10's or 100's of cycles. What's a cycle?

It occurs a lot in the context of computing machines. Most modern computers run in a clock architecture. When the chip is running, it routes electrons from one place to another. When we ask a certain thing to happen, the signal is sended through several certain gates and so on. Those paths these signals are taking are not of the same length. So some aspects of what we are trying to do are completed faster than others, and we don't let the faster ones start the next thing when the others haven't finished. We then want to make sure that every task is done at their destination before they start the next one. The clock is a regular signal, saying "go, go, go, go...". In between those "go"s, that's enough time for signals to reach their destination and wait to be told "go" again. Another interesting term is "overclocking". It tunes the clock of CPU to run a little faster. As long as everyone still gets job done between "go"s, everything is still fine, which might take more power. A *cycle* is time between two ticks, two time the clock says "go".

So it makes sense to have a small amount of fast memory and do computations on that. It would be fast because it's small, and no travel time because it's on CPU. And then we move to & from main memory when necessary. Specifically, it is called *registers*.[1]

24.2   MMIX Machine

First, we have general-purpose registers, and we are given 256 of them.[2] There are also special-purpose registers, and there are 32 of them. Both of them are 32 bits wide. And we have RAM as well.
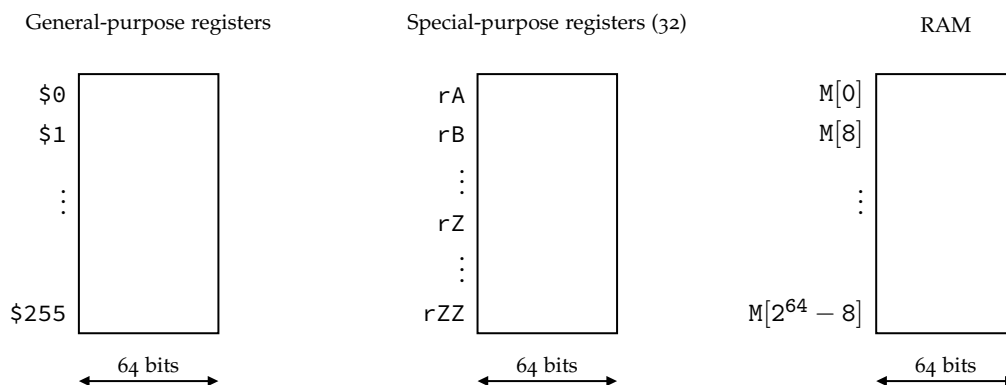
Example MMIX instruction (instructions are 32 bits wide).

```
ADD $4, $3, $2 ; set $4 to $3 + $2
```

This instruction is encoded in hex numbers. The hex encoding for this is `0x20040302`.

24.2.1   Data Processing Instructions

Now let's talk about some actual instructions. Generally speaking, instructions come in one of two formats.

- **Register format**:

  For example, `ADD $A, $B, $C`.

- **Immediate format**:

  For example, `ADD $A, $B, N`.

In general, immediate format opcode = reg format opcode + 1. For example, `ADD` (reg) opcode = `0x20`. `ADD` (imm) opcode = `0x21`. Also, `SUB` opcode is `0x24`, `MUL` opcode is `0x18`, `DIV` opcode is `0x1c`.

Note that for `DIV $X, $Y, $Z`, it sets $X \leftarrow \lfloor \$Y/\$Z \rfloor$ and sets $rR \leftarrow \$Y$ mod $\$Z$. rR is the remainder register. To fetch from rR or any special-purpose register: `GET $X, rR`. The opcode for it is `0xFE`.

For comparisons, there's a comparison instruction `CMP` with opcode `0x30`: `CMP $X, $Y, $Z`, and

$$\$X \leftarrow \begin{cases} -1 & \text{if } \$Y < \$Z \\ 0 & \text{if } \$Y = \$Z \\ 1 & \text{if } \$Y > \$Z \end{cases}$$

Once we have the result for comparison, we can use it for branches and jumps.

---

[1]Caches is the same idea, but it's between register and RAM.
[2]The number varies with chips because registers are built in to the chips. In MIPS, we have 32. In (intel), we get 4.

Branches & Jumps

To jump, the instruction is `JMP Addr`. Notice that there's only one argument, so it can occupy 24 bits. Also note this instruction stands for `@ += Addr * 4`, which is relative addressing. `@` stands for PC.

To branch, `BN $X, Addr` stands for `@ += Addr * 4` if `$X < 0` where "N" stands for negative. Also `BZ` (zero), `BP` (positive), `BOD` (odd), `BEV` (even), `BNN` (non-negative), `BNZ` (non-zero), `BNP` (non-positive).

Now let's do an example with GCD:

$$\gcd(0, a) = a$$

$$\gcd(a, b) = \begin{cases} \gcd(a - b, b) & \text{when } a \geq b \\ \gcd(a, b - a) & \text{when } b > a \end{cases}$$

First let's do it in SIMPL:

```
(seq
    (while (not (= a b))
        (iif (> a b)
            (set a (- a b))
            (set b (- b a))))
    (print a))
```

In PRIMPL:

```
(label gcd)
(eq tmp a b)
(branch tmp end)
(lt tmp a b)
(branch tmp less)
(sub a a b)
(jump gcd)
(label less)
(sub b b a)
(jump gcd)
(label end)
(print-val a)
```

In MMIXAL (assembly language), where we can use label:

```
gcd  CMP $2, $0, $1
     BZ  $2, end
     BN  $2, less
     SUB $0, $0, $1
     JMP gcd
less SUB $1, $1, $0
     JMP gcd
end  ??? ; how to print
```