Algorithm Design and Data Abstraction

CS 146

Brad Lushman

LaTeXed by *Sibelius Peng*

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.

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

Preface **1**

1 Jan 12 **3**
1.1 Major themes . . . 3
1.2 Recursion . . . 4
1.3 Impure Racket . . . 5

2 Jan 14 **6**
2.1 RAM . . . 6
2.2 Modelling Output . . . 6
2.3 Modelling input . . . 9
2.4 Input in Racket . . . 9

3 Jan 19 **10**
3.1 More primitive input reading . . . 10
3.2 Writing DrRacket . . . 11
3.3 Intro to C . . . 12
3.3.1 Expressions . . . 12
3.3.2 Statements . . . 12
3.3.3 Blocks . . . 13
3.3.4 Functions . . . 13
3.3.5 Programs . . . 13

4 Jan 21 **15**
4.1 Compile C programs . . . 15
4.2 Declaration vs. Definitions . . . 16
4.3 Variables and input in C . . . 17
4.4 Characters . . . 18

1

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,

```racket
(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 method: 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 ∀ `L`, ∀ `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

```
(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, `lambda`s, `local`, answers of `cond/match`. For example,

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

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).

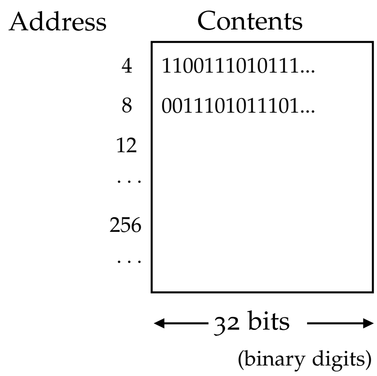
2

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:



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,

```
> (display "Hello")
Hello
> "Hello"
"Hello"
> (begin (display "Hello") 5)
Hello5
> (define x (begin (display "Hello") 5))
Hello
> x
5
```

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

```
> (define y (display "Hello"))
Hello
> y          ; y does have a value
> (list y)   ; by a trick
'(#<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:

```
(define (print-with-spaces lst)
    (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`:

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

or using `if`:

```
(define (for-each f lst)
    (if (empty? lst)
        (void)
        (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:

```
(define (for-each f lst)
    (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.

```
(define (fact n)
    (printf "fact applied to argument ~a\n" n) ; implicit begin
    (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.

```
(read-line) ; pops up a little box and lets us to type
Test.
"Test."     ; and get back the string as the result.
> (string->list (read-line)) ; if we type Test.
(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.

```
(define (read-input)
    (define nl (read-line)) ; nl stands for next line
    (cond [(eof-object? nl) empty]
          [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.

```
> (read-char)
abcde ; type in the box and press enter
#\a
> (read-char)
#\b
> (read-char)
#\c
> (read-char)
#\d
> (read-char)
#\e
> (read-char)
#\newline
> (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.

```
(define (my-read-line)
    (define (mrl-h acc)
        (define ch (read-char))
        (cond [(or (eof-object? ch) (char=? ch #\newline)) (list->string (reverse acc))]
              [else (mrl-h (cons ch acc))]))
    (mrl-h empty))

; call it by
(my-read-line)
```

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

```
> (read) ; type abc
'abc     ; symbol
> (read)
(a b c   ; not closed
```

```
de f ghi ; racket not satisfied
)        ; bracket closed
'(a b c de f ghi)
> (read)
(a b (c d e (f)) g)
'(a b (c d e (f)) g)
```

3.2 Writing DrRacket

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

```
(define (repl)
    (define exp (read))
    (cond [(eof-object? exp) (void)]
          [else (display (interp (parse exp)))
                (newline)
                (repl)]))
(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:

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

```
; read-id: -> (listof char)
(define (read-id)
    (define nc (peek-char))
    (if (or (char-alphabetic? nc) (char-numeric? nc))
        (cons (read-char) (read-id))
        empty))
```

```
; read-number: -> (listof char)
(define (read-number)
    (define nc (peek-char))
    (if (char-numeric? nc)
        (cons (read-char) (read-number))
        empty))
```

Here is our main tokenizer:

```
; read-token: -> token
(define (read-token)
    (define fc (read-char))
    (cond
        [(char-whitespace? fc) (read-token)]
        [(char=? fc #\() (token 'lp fc)]
        [(char=? fc #\)) (token 'rp fc)]
```

```
        [(char-alphabetic? fc) (token 'id (list->symbol (cons fc (read-id))))]
        [(char-numeric? fc) (token 'id (list->number (cons fc (read-number))))]
        [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.

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

All left is to build `read`:

```
; my-read: -> s-exp
(define (my-read)
    (define tk (read-token))
    (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

```
int main() { // int main(void) {
    ...
    ...
}
```

For example,

```
int main() {
    f(4, 3);
    return 0;
}
// and we got our f defined before
```

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

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);
    return 0;
}
```

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

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

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

```
int f(int x, int y); // - function prototype or header
                     // - declaration only.
int main() {
    f(4, 3);
    return 0;
}

int f(int x, int y) {                   // this is the function definition
    printf("x = %d, y = %d\n", x, y);   // also solves the mutual recursion problem
    return x + y;
}
```

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

```
int printf(---???---);
int f(int x, int y);

int main() {
    f(4, 3);
    return 0;
}

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

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

```
#include <stdio.h>

int f(int x, int y);

int main() {
    f(4, 3);
    return 0;
}

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

`#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.

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

Input:

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

Let's try to read in a number.

```
#include <stdio.h>
// like before, we don't care about the negatives at this point.
int getIntHelper(int acc) {
    char c = getchar();
    if (c >= '0' && c <= '9') return getIntHelper(acc * 10 + c - '0');
    else return acc; // "else" keyword is technically not needed here.
}

// An alternative way: ternary operator
int getIntHelper(int acc) {
    char c = getchar();
    return (c >= '0' && c <= '9') ? getIntHelper(acc * 10 + c - '0') : acc;
}

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

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

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

Typically here, stmt will be a block:

```c
if (test) {
    stmt 1
    ...
    stmt n
}
else {
    ...
}
```

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

```c
if (condition 1)
    if (condition 2)
        stmt 1
else // this ‘‘else’’ actually goes to the second ‘‘if’’
    stmt 2
```

Don't fool by the indentation. This is actually

```c
if (condition 1) {
    if (condition 2) {
        stmt 1
    }
    else { stmt 2 }
}
```

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 `char`s? (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 `char`s (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.

```c
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.