# LEC 2018-01-16

## Input Continued

### What have we lost by accepting inputs?

- *Referential Transparency*:
  - The same expression has the same value every time whenever it is evaluated.
    - e.g. **`(f 4)`** always produce the same value.
    - e.g. **`(let ((z (f 4))) body)`**: every z in body can be replaced by (f 4) and vice versa; "equals can be substituted for equals."
  - But this is not true anymore -- for example, if I use the function **`read`**, it probably won't return the same value every time. Thus it's harder for us to reason about program, because simple algebraic manipulation is no longer possible. *This is one of the difference between functional vs. imperative programming.*

## Intro to C

### Expressions

```
// infix operator
1 + 2

// function calls
f(7)
3 + f(x, y, z)

// print function, %d stands for decimal
// produces the number of characters printed
printf("%d\n", 5)

```

- Precedence: use usual mathematical conventions.

## Statements

-
  ```
  // take an expression and add a semicolon at the end
  printf("%d\n", 5);
  ```

    - The value produced by the statement is ignored.
    - The expression is evaluated only for its side-effects.

-
  ```
  1 + 2; //legal, but useless
  ```

-
  ```
  return 0; // produce the value 0 as the result of this function
  ```

    - Control returns immediately to the caller.

-
  ```
  ; // empty statement (does nothing)
  ```

## Blocks

- Groups of statements, treated as one statement.
- C:

```
{
  stmt1
  stmt2
  .
  .
  .
  stmtN
}
```

- Racket:

```
(void
 stmt1
 stmt2
 .
 .
 .
 stmtN
 )

```

### Functions

- C:

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

- Racket:

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

- The notion of contract in Racket is an artificial thing that Racket itself doesn't care; it's our message to the user. But in C, **`int f(int x, int y) {...}`** is legit code instead of comments. We must obey it to make the function work properly.

### Program

- A program is a sequence of functions.
- Starting point: function **`main`**.

### Our first program

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

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

// Doesn't compile...
```

- C compilers, unlike Racket compilers, runs from the top to the bottom.
- Therefore **`main`** does not know what **`f`** is.
- Conclusion: **C enforces declaration before use**
    - You can't use a function/variable/etc. until you tell C about it.
    - Why? Historical reasons... But in theory, C programs can be compiled with a one-pass compiler.
- Fix it: write **`f`** before **`main`**? OK, but it's more than necessary.

### A very important note: decoration *vs.* definition

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

- This is both a **decoration** (where I tell C about the function) and a **definition** (which completely constructs the function).
    - Decoration: "*there is a function that takes two integers and produces an integer.*"
    - Definition/Creation: *creating the function.*
- **C only requires decoration before use.** *I only need to tell C that there exists such a function* **`f`**.
- Thus, what we can do is, **`int f(int x, int y);`**
    - This is called a **function prototype**, or **header**
    - This is just a decoration. We need to define the function later in the program.

- **Remark.** This also solves the problem of mutual recursion.

```c
int f(int x, int y);

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

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

// Still doesn't compile... well, what's printf?? C doesn't know what printf is..xd
```

**Solution: `#include` some stuff!!**

- Rather than declare every standard library function header before you use it, C provides ***header files***. These files came with the compiler, so you don't need to write them by yourself.

```c
#include <stdio.h>

int f(int x, int y);

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

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

// Finally works...
```

- What is **`#include`**?
  - A ***C preprocessor directive***
  - This runs before the compiler; it is very similar to the **macro expansion** in Racket.
  - Here, the C preprocessor transforms this code into something else, then compile it.
  - **`#include <file.h>`** : "drop the contents of **`<file.h>`** right here."

- Standard-IO:
  - Contains declarations for **`printf`** and other IO functions.
  - Where is it? Located in a "standard place", ie. a place where your compiler knows where to look.
- **`printf`**?
  - It is written once, compiled once, and what you get is the binary of the function.
  - It is also located at the "standard place".
- Code for **`printf`** must be combined with this code -- ***linking***
  - A ***linker*** takes care of this (runs automatically).
  - This linker "knows" to link the code for **`printf`**.
- If you write your own modules, you need to tell the linker about them (later).

### More about `main`

- Your main function also returns a value -- **`return 0;`**
- This goes to the operating system.
- What for? To indicate whether the program was successful
- *Fun fact*: type **`echo $?`** you would get the return value of the function you last ran.

---

## Variables

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

- You need to be exclusive about what type of value a variable holds.

---

## Input

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

- **`char`** are just small **`int`** 's. XD

### Read in a number

```c
#include <stdio.h>
int getIntHelper(int acc){
  char c = getchar();
  if (c >= '0' && c <= '9'){
    return getIntHelper(acc * 10 + c - '0');
  } else {
    return acc;
  }

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

Questions

1. **Linking**
2. **Preprocessor directive**.