Logic and Functional Programming

Labwork 2

March 3, 2021

The purpose of these labworks is to practice recursive thinking when defining functions and recursive data types.

Recap

In functional programming, we use the special form

```
(define id expr)
```

to give name *id* to the result of evaluating *expr*. Typical examples are:

```
> ; r-Earth is the numeric value of Earth's radius, in km
  (define r-Earth 6371)

> ; Give name equator to the length of Earth's equator, in km
   (define equator (* 2 pi r-Earth))

> equator
40030.173592041145
```

The `define`-special form can also be used to define functions. For example

```
> (define sphere-volume (lambda (r) (* 4 pi (/ (* r r r) 3))))
```

gives name `sphere-volume` to the function which takes input argument `r` and computes the value of $4\pi\mathtt{r}^3/3$, which is the volume of a sphere with radius `r`.

In general, the special form

```
(lambda (x1 ... xn) body)
```

is used to define functions; it has the intended reading "the function which, for input arguments $x_1, \ldots, x_n$, computes and returns the value of *body*."

After we give names to values (including functions, which are also values), we can use them to compute more interesting things. For example, to compute the volume of the Earth (in km^3), we can call

```
> (sphere-volume r-Earth)
1083206916845.7537
```

REMARK: The explicit defininition of a function `f` in RACKET is

`(define f (lambda (`x_1` ... `x_n`) `*body*`))`

Alternatively, we can also write

`(define (f `x_1` ... `x_n`) `*body*`)`

The role of naming in programming

A fundamental programming principle is the **Principle of Abstraction**:

> *"Each significant piece of functionality in a program should be implemented* **in just one place** *in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts."*

In functional programming, the Principle of Abstraction is achieved by naming of all kinds of values (including functions) with `define`, and using these names later to write more compact code. Another useful programming capability is the usage of function values as **first-class programming citizens**. A first-class programming citizen is something that can be

- stored in a composite value (e.g., pair, list, or vector),
- passed as arguments to function calls,
- returned as results of function calls.

The following example illustrates how to respect the Principle of Abstraction by naming values with `define`.

1. Suppose we want to compute the value of $(\mathtt{a}-\mathtt{b}+2)^2+(\mathtt{a}-\mathtt{b}+2)/4$, where a, b are names already assigned to some values. The computation

   ```
   > (+ (expt (+ (+ (- a b) 2)) 2) (/  (+ (+ (- a b) 2)) 4))
   ```

   is against the Principle of Abstraction because we compute the twice value of `(+ (+ (- a b) 2))`. We can avoid this repeated computation by naming the value of the subexpression which occurs twice:

   ```
   > (define c (+ (+ (- a b) 2)))
   > (+ (expt c 2) (/ c 4))
   ```

   If `c` is used only to compute the value of `(+ (expt c 2) (/ c 4))`, we can use a `let`-form to make `c` visible only during this computation:

   ```
   > (let ([c (+ (+ (- a b) 2))])
         (+ (expt c 2) (/ c 4)))
   ```

2. Suppose we want to compute the sum of areas of three circles with radii `r1`, `r2` and `r3`. We know that the area of a circle with radius r is πr^2, thus we wish to evaluate[1]

   ```
   (+ (* pi (expt r1 2))  (* pi (expt r2 2)) (* pi (expt r3 2)))
   ```

   In this case there is a repeated pattern of operations that can be abstracted away, namely the computation of the area of a circle. According to the Principle of Abstraction, we should abstract this repeated pattern of operations in a function definition, and reuse it wherever it is needed:

   ```
   > (define (circle-area r) (* pi (expt r 2)))
   > (+ (circle-area r1)  (circle-area r2) (circle-area r3))
   ```

3. Suppose we wish to compute the sum and product of a list of numbers. If the list is empty, the sum of it's elements is assumed to be 0, and the product of its elements is assumed to be 1.

   It is easy to define recursively the sum and product of a list of numbers `l`:

   (a) If `l` is empty, the sum is 0 and the product is 1.

   (b) Otherwise:

      - The sum is the result of adding the first element of `l` with the sum of elements of the rest of `l`,
      - The sum is the result of multiplying the first element of `l` with the product of elements of the rest of `l`,

   ```
   (define (sum-list l)
      (if (null? l) 0 (+ (car l) (sum-list (cdr l)))))

   (define (prod-list l)
      (if (null? l) 1 (* (car l) (prod-list (cdr l)))))
   ```

   The definitions of $f \in \{$`sum-list`, `prod-list`$\}$ are very similar: Their bodies are of the form

   ```
   (if (null? l) v (op (car l) (f (cdr l)))
   ```

   where `v` and `op` are the things that differ between their definitions. According to the Principle of Abstraction, we should try to abstract away this common pattern of computation. We can do so by defining a function `(fold op v l)` which behaves like `(sum-list l)` when `op` is `+` and `v` is `0`, and behaves like `(prod-list l)` when `op` is `*` and `v` is `1`:

---

[1] In RACKET, `pi` is a predefined name for the numeric value of π.

```
(define (fold op v l)
   (if (null? l) v (op (car l) (fold op v (cdr l)))))
(define (sum-list l)  (fold + 0 l))
(define (prod-list l) (fold * 1 l))
```

Note that, if `l` is a list of elements $\mathtt{v}_1, \mathtt{v}_2, \ldots, \mathtt{v}_\mathtt{n}$ in this order, then

$(\mathtt{fold\ op\ v}_0\ \mathtt{l})$

computes the value of the expression

$(\mathtt{op\ v}_1\ (\mathtt{op\ v}_2\ \ldots\ (\mathtt{op\ v}_\mathtt{n}\ \mathtt{v}_0)\ldots))$

In the special case when `l` is the empty list `null`

$(\mathtt{fold\ op\ v}_0\ \mathtt{l})$

returns $\mathtt{v}_0$.

The role of recursion in functional programming

1) Repetitive computations

In pure functional programming, we can not change the value assigned to a name. This means that:

- A variable defined in a scope has always the same value
- We can not work with repetitive instructions specific to imperative programming, such as `for` and `while`, because these instructions usually change the value of some variables. In pure functional programming, we can not change the value of a variable.

In pure functional programming, all repetitive computations are performed by calling functions defined recursively.

For example, suppose we want to compute the factorial value $\mathtt{1} \cdot \mathtt{2} \cdot \ldots \cdot \mathtt{n}$ for all integers $\mathtt{n} > \mathtt{0}$. In an imperative programming language (e.g., `C` or `Pascal`), we could implement the pseudocode for the procedure

```
int procedure fact(int n)
int result=1;
for (int i=1;i<=n,i++)
   result *=i;
return result
```

In functional programming, we observe that the factorial value is computed by the recursive (mathematical) function

$$\mathtt{fact} : \mathbb{N} \to \mathbb{N}, \quad \mathtt{fact}(\mathtt{n}) := \begin{cases} 1 & \text{if } n = 1, \\ \mathtt{n} \cdot \mathtt{fact}(\mathtt{n} - 1) & \text{if } n > 1. \end{cases}$$

The encoding in RACKET of this recursive function is

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

2) Recursive data types

Another place where recursion appears in computer science is in the definition of recursive data structures. Like recursive functions, recursive data structures (also known as *recursive data types*) are defined by cases, and we distinguish

- One or more **base cases**, which indicate the most elementary values of the recursive data type.
- One or more **recursive cases**, which indicate how to build composite values from smaller values, including values of the same type.

A popular way to define the syntax of recursive data types and programming constructs is with context-free grammars in Backus-Naur form (BNF). In general, the BNF definition of a recursive datatype *type* looks as follows:

$type$ `::=` $case_1 \mid \ldots \mid case_n$

where '|' is a separator between different alternatives, and every alternative $case_i$ indicates how to build a value of *type* from other values, including smaller values of type *type*. Te alternatives which indicate the construction of a value from smaller values of type *type* are the recursive cases, and the remaining alternatives are the base cases.

The following are typical examples of recursive datatypes:

▶ Lists of arbitrary values

```
lst ::= null          ; base case
      | (cons v lst)  ; recursive case
```

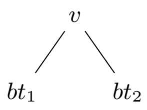
where v is any value (e.g., an integer, string, boolean, symbol, etc.).

Defining a recognizer `lst?` for values of type `lst` is straightforward: We just have to check that one of the cases holds:

```
(define (lst? l)
    (or (null? l)
        (and (cons? l) (lst? (cdr l)))))
```

▶ The type `BT` of binary trees of integers or symbols can be defined recursively as follows:

  - Every integer or symbol is of type `BT`.
  - Every other binary tree *bt* of integers or symbols is of the form



where v is an integer or symbol, and bt_1, bt_2 are values of type `BT`. We decide to represent such a binary tree with the list `(list` v bt_1 bt_2`)`. The BNF definition of this encoding of values of type `BT` is

```
BT ::= v                    ; base case
     | (list v BT BT)       ; recursive case
```

where v is an integer or symbol. Defining a recognizer `BT?` for values of type `BT` is straightforward:

```
(define (BT? bt)
   (or (number? bt)
      (symbol? bt)
      (and (list? bt)
           (= 3 (length bt))
           (or (number? (car bt)) (symbol? (car? bt)))
           (BT? (cadr bt))
           (BT? (caddr bt)))))
```

3) Defining recursive functions by structural induction

Several functions take one or more arguments of a recursively defined type. In such situations, we should try to define them by induction on the structure of that argument. Typical examples of such functions are the type recognisers of recursively defined datatypes (e.g., `lst?` and `BT?` which we have already defined). Below are more examples.

1. The function `(map f l)` which takes as input arguments a unary function `f` and a list `l`, and returns the list of values obtained by applying function`f` to each element of `l`. This function can be defined by induction on the structure of `l`, which is a value of the recursive type `list`:

   ```
   (define (map f l)
      (if (null? l) l (cons (car l) (map f (cdr l)))))
   ```

2. The function `(filter pred l)` which takes as input arguments a predicate `pred` and a list `l`, and returns the list of elements of `l` which satisfy predicate `pred`. For example, `even?` is a predicate that recognises even integers, and the function call `(filter even?'(1 2 3 4 5))` should return the list `'(2 4)`.

   ```
   (define (filter pred l)
      (cond
          [(null? l) l]
          [(list? l)
             (if (pred (car l))
                 (cons (car l) (filter pred (cdr l)))
                 (filter pred (cdr l)))]))
   ```

3. The function `(join l1 l2)` which joins two lists `l1` and `l2`. For example `(join '(1 2 3) '(4 5))` should return `'(1 2 3 4 5)`.

   ```
   (define (join l1 l2)
      (if (null? l1) l2 (cons (car l1) (join (cdr l1) l2))))
   ```

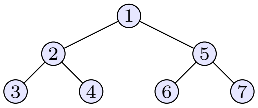
   Note that both arguments of `join` are of the recursive type `list`, and we could try to define `join` by induction on the structure of `l2`. However, the attempt to define `join` by induction on the structure of `l2` fails.

Labworks

LW1 Consider binary trees of integers defined by the BNF

```
BTI ::= n
     | (list n BTI BTI)
```

where n is an integer. For example, the binary trees of integers



is represented by the list `'(1 (2 3 4) (5 6 7))`. Also, consider the following tree traversal strategies:

preorder: visit root, then left subtree, then right subtree.

inorder: visit left subtree, then root, then right subtree.

postorder: visit left subtree, then right subtree, then root.

Define recursively the functions `(preorder bti)`, `(inorder bti)`, and `(postorder bti)` which return the list of nodes in the binary treee of integers `bti` in the order in which they are visited. For example:

```
> (preorder '(1 (2 3 4) (5 6 7)))
'(1 2 3 4 5 6 7)
> (inorder '(1 (2 3 4) (5 6 7)))
'(3 2 4 1 6 5 7)
> (postorder '(1 (2 3 4) (5 6 7)))
'(3 4 2 6 7 5 1)
```

LW2 A nested list of numbers is either the empty list, or a list whose elements are either numbers, or nested lists of numbers.

   (a) Write down the BNF definition for the recursive type `nlist` of nested lists of numbers.

(b) Define recursively the recogniser `(nlist? l)` for values of type `nlist`. For example:

```
> (nlist? null)     > (nlist? '(((1) 2) 3.2 ((4))))
#t                  #t
> (nlist? 1)        > (nlist? '(4 ((-5) a)))
#f                  #f
```

LW3. The decimal representation of a non-negative integer `N` is $d_n d_{n-1} \dots d_1 d_0$ where $n \geq 0$, and the sum of its digits is $d_0 + d_1 + \dots + d_{n-1} + d_n$.

Suppose we wish to define the function

`(digit-sum N)`

which computes the sum of digits of the non-negative integer `N`.

Note, again, that `N` does not have an explicitly defined recursive structure. However, we observe that non-negative integers **do have** a recursively defined structure, but we have to define our own recognisers and selectors for it:

Base case: `N` consists of a single decimal digit. In this case `(digit-sum N)` coincides with `N`.

Recursive case: `N` is of the form $\mathtt{10} \cdot \mathtt{M} + \mathtt{D}$ where $\mathtt{M} < \mathtt{N}$ is a positive integer, and `D` is the last decimal digit of `N`. In this case, we must add `D` with the value of `(digit-sum M)`.

To take advantage of this structure of non-negative integers, we must define the recogniser

- `(is-digit? N)` which recognises if `N` is a decimal digit

and the selectors

- `(drop-last-digit N)` which returns the number `M` obtained by dropping the last digit of `N`, and
- `(last-digit N)` which returns the value of last digit `D` of `N`.

when $\mathtt{N} > \mathtt{10}$.

LW4. Define the function `(flatten sl)` which takes as input a nested list of symbols, and returns the list of symbols contained in `sl` in the order in which they occur when `sl` is printed. Intuitively, `flatten` removes all the inner parentheses form its argument. For example:

```
> (flatten '(a b c))
'(a b c)
> (flatten '((a b) c (((d)) e)))
'(a b c d e)
> (flatten '((a) () (b ()) () (c)))
'(a b c)
```

Suggestion: First, write a recursive definition (BNF) for the nested lists of symbols.

LW5. Define the function `(swapper s1 s2 sl)` which takes as input the symbols `s1` and `s2` and the list of symbols `sl`, and returns the list of symbols which is the same as `sl`, but with all occurrences of `s1` replaced with `s2` and all occurrences of `s2` replaced by `s1`.

For example, `(swapper 'a 'b '(a b r a c a d a b r a))` should produce the list `'(b a r b c b d b a r b)`