CS 381 — Notes

Philip Warton

April 5, 2022

1 Elm

Trace for the evaluation of

$$[1,2] + +[5]$$

.

```
[1,2] ++ [5]
1::([2] ++ [5])
1::(2::[5])
1::[2,5]
[1,2,5]
```

For the function definition

```
(++) : List Int -> List Int -> List Int
l1 ++ l2 = case l1 of
        []      -> l2
        x::xs   -> x::(xs ++ l2)
```

First quiz is on Thursday in person, written on paper. We are allowed one sheet of notes, single sided. A function is a **parameterized expression**. Take something such as $3 + 5$ and replace these with $x + y$ and now it is a function on two variables. Notice how in "MAD LIBS" the parameters are given types that say what kind of argument should be placed into the function. A **function definition** is simply giving a name to a function. We make a simple sign function as follows

```
sign : Int -> Int
sign x = if x==0 then 0 else x//abs x
```

Assume that a min function already exists that returns the smallest integer given two arguments. Then we can define min3 as

```
min3 : Int -> Int -> Int -> Int
min3 x y z = min x (min y z)
```

We can also create a while loop recursively:

```
loop(state) = if cond(state) then
                loop(update(state))
              else
                result(state)
loop(initValues)
```

We must remember to keep in mind the mindshift about processing values. In programming with pattern matching

- A **value** v is a term built out of constructors
- A **pattern** p is a term build out of constructors and variables
- A **rule** $p \to e$ consists of a patter p and an expression e

- **Matching** v agains p creates a set of **variable bindings**
- **Evaluating** a rule for v means to evaluate e in the context of bindings produced by matching v against p

For elm, the following syntax will give us pattern matching

```
case v of
    ...
    p -> e
```

General recursion template:

```
type T = Val | Con S T

f : T -> U
f x = case x of
        Val         -> e1{Val}      -- base case
        Con s t     -> e2{s, f t}   -- inductive case
```

We practice this by constructing a recursive isEven function:

```
isEven : Int -> Bool
isEven x = case x of
        0 -> True
        1 -> False
        n -> isEven (n-2)
```

1.1 Lists

Lists are built with two constructors, [] and $v :: l$. That is,

```
[]      : List a
(::)    : a -> List a -> List a
```

We have some accessors as well

```
isEmpty : List a -> Bool
head    : List a -> Maybe a
tail    : List a -> Maybe (List a)
```

If you apply head to an empty list then **Maybe** becomes important. You can raise a runtime error like Haskell does, or we can do it like elm. If head finds some element then it will return the value, otherwise some error will be thrown. We have a template for list processing using pattern matching:

```
f: List T -> U
f l = case l of
    []      -> e1           -- result for empty list
    x::xs   -> e2{x, xs}    -- result for non-empty list
```