CS 381 — Notes

Philip Warton

May 10, 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
```

1.2 Higher order functions and polymorphism

Every elm value has a type, as do functions, with some small exceptions ex. 3 (Number). If we recall our function 'length', it took a type of 'List a' to 'Int'. We are using a variable for the type. This is called parametric polymorphism instead of sub-type polymorphism. That is, 'a' is a type parameter that will work for any different type. Another example is our function 'Tuple.first' which maps a tuple $(a, b) \to a$. In this example we can have a tuple with two different type variables to the type variable of the first. If you see the same type variable name twice, they must inherit the same type. Often with these type parameters the type of a function dictates much of the implementation of that function. For example see this zip function that can be implemented in elm:

```
zip : List a -> List b -> List (a,b)
zip l1 l2 = case l1 of
    [] -> []
    x::xs -> case l2 of
```

```
        [] -> []
        y::ys -> (x,y)::zip xs ys
```

The map function will apply a function to each element of a list. It is defined as

```
map : (a -> b) -> List a -> List b
map f l = case l of
    [] -> []
    x::xs -> f x::map f xs
```

One can notice that the type of the function almost entirely defines how we can possibly define the function.

```
andThen : Maybe a -> (a -> Maybe b) -> Maybe b
andThen m f = case m of
    Nothing -> Nothing
    Just x -> f x

reverse : List a -> List a
reverse l = case l of
    [] -> []
    x::xs -> reverse xs ++ [x]

last : List a -> Maybe a
last = head << reverse
```

We think of $f << g$ as 'f composed with g'.

2 Grammar

I am missing a lot of stuff on here to be honest.

2.1 Parse Trees

2.2 Abstract Syntax

All of the following were correct

- type Bin = A | B | Conc Bin Bin
- type Bin = Single Bool | Many Bool Bin
- type Bin = OnlyZero | OnlyOne | ZeroAnd Bin | OneAnd Bin
- type Bin = Seq Bool [Bool]

Binary predicate example: factorial!

$$\frac{}{fac(1,1)} \qquad \frac{fac(n-1,m)}{fac(n,n\cdot m)}$$

We can also come up with other ways to formulate the second rule, which may be useful for readability or workability. It is also possible to define these things using different syntax. As an exercise, we define fibonacci numbers using inference rules:

$$\frac{}{fib(1,1)} \quad \frac{}{fib(2,1)} \quad \frac{fib(n-1,a), fib(n-2,b)}{fib(n,a+b)}.$$

A derivation can be though of as a proof trees. Leaves are axioms, and internal nodes are instances of rules.

$$\frac{}{n \Downarrow \underline{n}}$$

$$\frac{e \Downarrow n}{-e \Downarrow -n}$$

$$\frac{e_1 \Downarrow n \quad e_2 \Downarrow m}{e_1 + e_2 \Downarrow n + m}$$

$$\frac{e_1 \Downarrow n \quad e_2 \Downarrow m}{e_1 * e_2 \Downarrow n \cdot m}$$

For creating a type system we need the following components:

$$v \in var$$
$$e \in expr$$
$$T \in type$$

Type system in 4 steps:

1. Define object language
2. Define type language
3. Define typing rules
4. Finally we need a type checker

$$e \in expr ::= n \mid e + e \mid e = e \mid not\ e$$
$$T \in type ::= Int \mid Bool$$

$$r_1 = \frac{}{n : Int}$$

$$r_2 = \frac{e_1 : Int \quad e_2 : Int}{e_1 + e2 : Int}$$

$$r_3 = \frac{e_1 : Int \quad e_2 : Int}{e_1 = e_2 : Bool}$$

$$r_4 = \frac{e : Bool}{not\ e : Bool}$$