Demo Session Notes

Rachel Samuelsson

Week 1

If you want to play with this haskell code yourself can load the source for this document (the `.lhs` file, not the `.pdf`) in ghci, just like any other haskell file.

Exercise 1.1

Recall the datatype from the exercise

```
data Exp = Con Integer
         | Plus Exp Exp
         | Minus Exp Exp
         | Times Exp Exp
         deriving (Eq, Show)
```

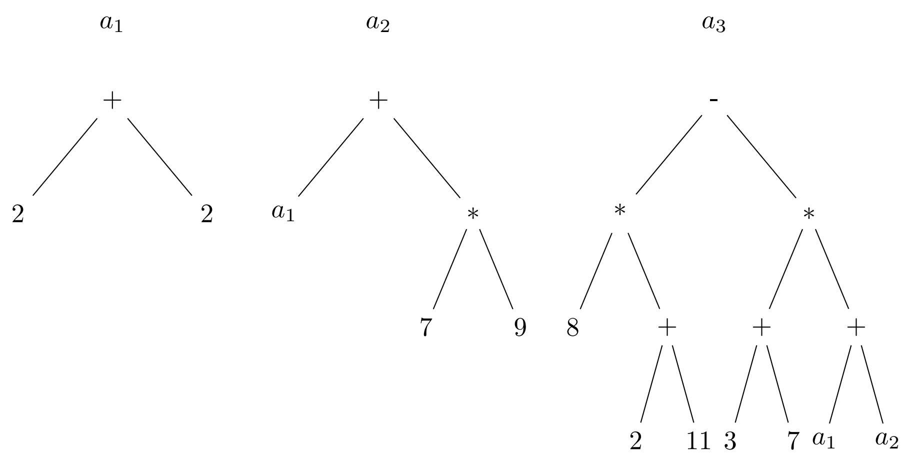
Part 1

It's benificial to start out with thinking about what these expressions really mean.

One way of doing this is by adding more parethesis to the expressions.

$$a_1 = 2 + 2$$
$$a_2 = a_1 + (7 * 9)$$
$$a_3 = (8 * (2 + 11)) - ((3 + 7) * (a_1 + a_2))$$

Another approach is to draw the expressions as trees, where a node is an operation, and its children are its operands.



This is nice cause it lets us easily see what the outermost operation is at each point, which is a bit harder in infix expressions. Generally it can be benifical to think about expressions as trees, rather than strings of symbols/text.

We can now translate these into Haskell by replacing the operands with their corresponding constructor by taking the left tree as the first argument and the right as the second. Numbers are translated using the `Con` constructor.

```haskell
a1, a2, a3 :: Exp
a1 = Plus (Con 2) (Con 2)
a2 = Plus a1 (Times (Con 7) (Con 9))
a3 = Minus (Times (Con 8) (Plus (Con 2) (Con 11)))
           (Times (Plus (Con 3) (Con 7)) (Plus a1 a2))
```

Part 2

For this part we will pattern watch on the `Exp` type and use a lot of recursion. Each operation is evaluated by first evaluating its operands, which are also of the type `Exp`, yielding the integers corresponding to their value, and then applying the proper integer operator to these values.

```haskell
eval :: Exp -> Integer
eval (Con i) = i
eval (Plus e1 e2) = eval e1 + eval e2
eval (Minus e1 e2) = eval e1 - eval e2
eval (Times e1 e2) = eval e1 * eval e2
```

This is a very general pattern which you will see a lot in the course. You have some data structure, where each constructor has some corresponding function in another type. Then evaluation can be done by evaluating all the constructors arguments and applying the corresponding function to them. In this case `Plus` corresponds to +, `Minus` to −, etc.

Part 3

First we add a constructor to the `Exp` type to allow us to represent variables.

```
data Exp = Con Integer
         | Plus Exp Exp
         | Minus Exp Exp
         | Times Exp Exp
         | Var String
         deriving (Eq, Show)
```

We can now define c_1 as an `Exp`.

```
c1 :: Exp
c1 = Times (Times (Minus (Var "x") (Con 15))
                  (Plus (Var "y") (Con 12)))
           (Var "z")
```

Note: we could have put the parenthesis on the multiplication the other way around and ended up with a different Haskell expression. This expression would be equally valid as multiplication is associative, meaning that $a*(b*c) = (a*b)*c$.

In order to evaluate c_1 we need to assign values to variables, we do this by defining a function `varVal`, which pattern matches on our strings.

```
varVal :: String -> Integer
varVal "x" = 5
varVal "y" = 8
varVal "z" = 13
```

This function then lets us extend eval to handle c_1, and other expressions including variables.

```
eval :: Exp -> Integer
eval (Con i) = i
eval (Plus e1 e2) = eval e1 + eval e2
eval (Minus e1 e2) = eval e1 - eval e2
eval (Times e1 e2) = eval e1 * eval e2
eval (Var s) = varVal s
```

We can then evaluate c_1 and see that it evaluates to -2600.

Exercise 1.2

Note that in the previous exercise we chose to look only at evaluations to the integers, and the constant contstructor `Con` only took integers. In reality there are many more types for which it makes sense to talk about addition, subtraction, and multiplication, so we may want to talk about expressions over many types. To this end the exercise introduces a new type:

```
data E2 a = Con' a
          | Var' String
          | Plus'  (E2 a) (E2 a)
          | Minus' (E2 a) (E2 a)
          | Times' (E2 a) (E2 a)
```

Note: The constructors are given a name ending in `'` here, unlike the exercise, this is due to Haskell not allowing multiple constructors of the same name being defined in one file, and this file being a valid Literate Haskell file.

Part 1

Translating expressions into this type is exactly analagous to before, except now we can have `Con` values which are not integers.

Part 2

There is another, arguably bigger, issue with the first implementation: the way names are associated to values. Associating names to values with a hardcoded function requires one to change the function each time the value of a variable is supposed to be changed. It would be nicer if the evaluation function used a table which was easier to edit. In Haskell we define the type of a table to be a list of pairs of strings and values.

```
type Table a = [(String, a)]
```

We think of `x` having value `5`, if the list contains an entry `("x", 5)`.

We then define the given table in code by writing a list of all the name, value pairs.

```
vars :: Table Double
vars = [ ("a", 1.5)
       , ("b", 4.8)
       , ("c", 2.4)
       , ("d", 7.4)
       , ("e", 5.8)
       , ("f", 1.7)
       ]
```

In order to be able to retrieve the value of a variable we need a function which takes a teble, and a string, and returns the proper table. We do this by recursing on the list till we find a matching string.

```
varVal' :: Table a -> String -> a
varVal' ((str, val):xs) s
  | str == s = val
  | otherwise = varVal' xs s
```

Note: if the name is not in the list this function will error.

We are now ready to update the evaluation function. The new function will take a table, in addition to an expression, which will define the variable names. Now we decide what variables to use when we call the function, not when we write it. Most of the function will be the same, we only really need to add the new table argument, remember to pass it on when we recurse, and use the new varVal to look up variables.

```
eval' :: Num a => Table a -> E2 a -> a
eval' table (Con' i) = i
eval' table (Plus' e1 e2) = eval' table e1 + eval' table e2
eval' table (Minus' e1 e2) = eval' table e1 - eval' table e2
eval' table (Times' e1 e2) = eval' table e1 * eval' table e2
eval' table (Var' s) = varVal' table s
```

Exercise 1.7

We want to show that having a function from the type `Either b c` to `a` is the same as having a pair of functions `b -> a` and `c -> a`. Intuitively this makes sense, as when we want to define a function out of the `Either` type, we usually do so by pattern matching on the `Left` and `Right` constructors, which requires us to give a case, or a function, for both types.

The way we prove these are the same are by giving functions back and forth. The first of which is `s2p :: (Either a b -> c) -> (b -> a, c -> a)`. We know this function should look something like `s2p f = x`, where `x :: (b -> a, c -> a)`. As `x` is a pair we could then try to construct a pair ourselfs, which yields `s2p f = (g, h)`, where `g :: b -> a` and `h :: c -> a`. Continuing this pattern we realise we need functions and use lambdas, which gives us `s2p f = (\b -> a1, \c -> a2)`. We then only need to construct `a1, a2 :: a`. We can do this using the function `f :: Either a b -> c`, given as an input, by first applying `Left` to `a` and `Right` to `b`.

```
s2p :: (Either b c -> a) -> (b -> a, c -> a)
s2p f = (\a -> f (Left a), \b -> f (Right b))
```

If we wish to, we could rewrite this using function composition

```
s2p' :: (Either b c -> a) -> (b -> a, c -> a)
s2p' f = (f . Left, f . Right)
```

The other direction of the equivalence was already hinted at, with the intuiton I gave above. We can patter match on the `Either b c`, and apply one of the functions given as an input depending on if we matched on `Left` or `Right`. Note that the function arrows in Haskell are right associative (if there are no parenthesis written out, they are implicitly grouped to the right, e.g `a -> b ->`

`c -> d` means `a -> (b -> (c -> d))`), as such, we can match on the either from the returned function as well.

```
p2s :: (b -> a, c -> a) -> (Either b c -> a)
p2s (f, g) (Left a) = f a
p2s (f, g) (Right b) = g b
```

Exercise 1.11

Recall the type `ComplexSyn r` from the book:

```
data ComplexSyn r
  = ToComplexCart r r
  | ComplexSyn r :+: ComplexSyn r
  | ComplexSyn r :*: ComplexSyn r
```

Part 0

Since eval is applied to e, e must be of type `ComplexSyn r`, we also see that `e` is compared to the result of the embed function, which also indicates the type is `ComplexSyn p`.

Part 1

The equality for the syntax just tells us how things are written. Here $1 + 1 \neq 2$, since they are different syntactic representations, even if they have the same semantics (value).

Part 2

Using the syntactic equality `embed (eval s)` is equal to `s` only if `s` is of the form `ToComplexCart x y`, as that is all `embed` outputs. However, using equality up to evaluation we have

```
      embed (eval s) == s
      -- up to eval
  <=> eval (embed (eval s)) == eval s
      -- eval (embed x) == x for all x
      -- thus eval (embed (eval s)) == eval s, with x = eval s
  <=> eval s == s
```

Meaning it trivially holds up to evaluation.