1 Declarative Programming

Does mercury use lazy or eager evaluation?

Some basic syntax boolean definitions are:

- && for AND
- || for OR
- *not* for not
- to test for equality use ==
- to test for inequality use / =

1.1 infix functions

- An infix function is a sandwhiched function like *.
- If a function takes two parameters, we can also call it as an infix function by surrounding it with backticks. For instance, the div function takes two integers and does integral division between them. Doing div 92 10 can also be called like 92 ‘div‘ 10.
- The function scope is limited by the functions of which it uses.

1.1.0.1 examples

- *succ x* is a funciton which returns $x + 1$
- *min* and *max* do the usual
- *sort variable* sorts the list from lowest to highest. The variable can be $[a_1, a_2, a_3, ...]$: a list, or a string of characters: anything that has order.
- fst tuple returns the first element.
- snd tuple returns the second element.
- cons operator : is the name of a function which joins together multiple items and places them in a list (which may already have elements) $item_0 : item_1 : item_2 : \cdots : some\_list\_to\_prepend\_to$.
- the map function passes a list of values through a function and returns the output values in place: map (function) list. That is, map takes a function as a parameter.
- filter (condition) list, filters the list for only items that satisfy the condition.
- toUpper var changes the case of a character to upper case.
- words ”sentance” returns a list of the words in that sentance.

1.1.0.2 definitions

- let function var = var_expr in function var_instance,
- let square x = x * x in map square [1..10],
- important in writing funciton is the concept of pattern matching.
- You can use parentheses to use more than one function. You want to double all the numbers over five? Psch! $map(*2)(filter(> 5)[10, 2, 16, 9, 4])$

1.2 Pattern Matching

A pattern always matches the way the value was originally constructed. Remember that ”abc” is syntactic sugar for ’a’ : ’b’ : ’c’ : []. If you just want some of the values, you can ignore the others with _ (underscore) like this: let (a:_:_:_) = ”xyz” in a. In fact, (a:b:c:d) is short-hand for (a:(b:(c:d))), so you can just ignore the rest in one go: let (a:_) = ”xyz” in a.

You can pattern match and keep the orignal value also. let abc@(a,b,c) = (10,20,30) in (abc,a,b,c) gives ((10,20,30),10,20,30)

1.3 If then else

```
doubleSmallNumber x = if x > 100
                      then x
                      else x*2
The else statement is mandatory within haskell
The if else statement in Haskell is an expression.
```

1.3.0.3 Lists

- strings are lists, for example ”hello” is really [’h’,’e’,’l’,’l’,’o’]
- lists are homogenous data structures: stores elements of a similar type.
- Watch out when repeatedly using the ++ operator on long strings. When you put together two lists (even if you append a singleton list to a list, for instance: [1,2,3] ++ [4]), internally, Haskell has to walk through the whole list on the left side of ++. That’s not a problem when dealing with lists that aren’t too big. But putting something at the end of a list that’s fifty million entries long is going to take a while. However, putting something at the beginning of a list using the : operator (also called the cons operator) is instantaneous. such as 5:[1,2,3,4,5]
- To obtain the item at index n of list A: $nthitem = A!!n$
- Lists have order if the stuff they contain is ordered. When using ¡, ¡=, ¿ and ¿= to compare lists, they are compared in lexicographical order.
- head list returns the first element.
- tail list returns everything but the first element
- last list returns the last element of a list
- init list returns everything but the last item
- length x takes a list x and returns its length, obviously.
- null x checks if a list x is empty and returns boolean True or False.
- reverse x reverses a list x.
- take x y takes number x and a list y. It extracts that many elements from the beginning of the list up to a max number of a list.
- drop works in a similar way, only it drops the number of elements from the beginning of a list.
- maximum takes a list of stuff that can be put in some kind of order and returns the biggest element.
- minimum returns the smallest.
- sum takes a list of numbers and returns their sum.
- product takes a list of numbers and returns their product.
- elem takes a thing and a list of things and tells us if that thing is an element of the list. It’s usually called as an infix function because it’s easier to read that way.
- To make a range of numbers do [x..y] = [x, x+1, ..., y-1, y] Characters can also be enumerated: anything that has order can be enumerated. A step can also be specified [x,y..z] = [x, x+(y-x),x+2*(y-x), ..., z-(y-x), z]. To make a list with all the numbers from 20 to 1, you can’t just do [20..1], you have to do [20,19..1]. You can also use ranges to make infinite lists by just not specifying an upper limit. Later we’ll go into more detail on infinite lists. For now, let’s examine how you would get the first 24 multiples of 13. Sure, you could do [13,26..24*13]. But there’s a better way: take 24 [13,26..]. Because Haskell is lazy, it won’t try to evaluate the infinite list immediately because it would never finish. It’ll wait to see what you want to get out of that infinite lists. And here it sees you just want the first 24 elements and it gladly obliges.
- cycle x takes a list x and cycles it into an infinite list.
- repeat x takes an element x and produces an infinite list of just that element [x, x, x, ...].
- replicate x y takes an element y and produces a list of x entries of that element.

1.3.1 List comprehensions

List comprehensions are very similar to set comprehensions. We’ll stick to getting the first 10 even numbers for now. The list comprehension we could use is $[x * 2|x < -[1..10]]$. x is drawn from [1..10] and for every element in [1..10] (which we have bound to x), we get that element, only doubled. Here’s that comprehension in action. Now with the added predicate that x times 2 is greater than 12: $[x * 2|x < -[1..10], x * 2 >= 12]$

When drawing from several lists, comprehensions produce all combinations of the given lists and then join them by the output function we supply: $[x * y|x < -[2, 5, 10], y < -[8, 10, 11]]$

1.4 Introduction

Bryan O’Sullivan, John Goerzen and Don Stewart: Real world Haskell. O’Reilly Media, 2009. ISBN 978-0-596-51498-3. Available on- line at http://book.realworldhaskell.org/read.

1.4.0.1 The Blub paradox

Consider Blub, a hypothetical average program- ming language right in the middle of the power continuum. When a Blub programmer looks down the power continuum, he knows he is looking down. Lan- guages below Blub are obviously less powerful, be- cause they are missing some features he is used to. But when a Blub programmer looks up the power continuum, he does not realize he is looking up. What he sees are merely weird languages. He thinks they are about equivalent in power to Blub, but with some extra hairy stuff. Blub is good enough for him, since he thinks in Blub. When we switch to the point of view of a program- mer using a language higher up the power contin- uum, however, we find that she in turn looks down upon Blub, because it is missing some things she is used to. Therefore understanding the differences in power between languages requires understanding the most powerful ones.

Levels of abstraction and power range from assembler to machine code to low level imperative languages to object oriented languages, scripting languages and multi paradigm languages to declarative languages.

1.5 Functional programming

- Imperative languages are based on commands, in the form of instructions and statements. Commands are executed. Commands have an effect.
- Functional languages are based on expressions. Expressions are evaluated. Expressions have no effect.
- The basis of functional programming is equational reasoning: if two expressions have equal values, then one can be replaced by the other.
- Haskell can be run compiled with ghc or can be interpreted with ghci. The prelude is Haskell's standard library. ghci uses its name as the prompt to remind users that they can call its functions.
- NOTE Once you invoke ghci,
  - you type an expression on a line;
  - it typechecks the expression;
  - it evaluates the expression (if it is type cor- rect);
  - it prints the resulting value. You can also load Haskell code into ghci with :load filename.hs. The suffix .hs, the standard suffix for Haskell source files, can be omitted.

1.6 Lists

The notation [] means the empty list, while x:xs means a nonempty list whose head (first element) is represented by the variable x, and whose tail (all the remaining elements) is represented by the vari- able xs. The notation ["a", "b"] is syntactic sugar for "a":"b":[] or for ["a",["b"]].

1.7 Syntax introduction

Function definitions should be exhaustive (one pattern should apply for every call) and one-to-one aka exclusive. The option -fwarn-incomplete-patterns, ghci will give you a warning if the patterns are not exhaustive; if given the option -fwarn-overlapping-patterns, ghci will give you a warning if the patterns are not exclusive. Program abortion will occur if not exhaustive. If not mutually exclusive, the first declaration found (from the top of the script downwards that applies will occur. Basic function call is:

```
f fa1 fa2 fa3.
```

Comments start with two minus signs and continue to the end of the line.

The names of functions and variables are sequences of letters, numbers and/or underscores that must start with a lower case letter. Actually, it is also ok for function names to consist wholly of graphic characters like +, but only builtin functions should have such names.

Indentation is important.

```
-- vim: syntax=haskell
```

This specifies the set of rules vim should use for syntax highlighting.

A function or expression is said to have a side effect if, in addition to producing a value, it also modifies some state or has an observable interaction with calling functions or the outside world. What really distinguishes pure declarative lan- guages from imperative languages is that they do not allow side effects. There is only one benign exception to that: they do allow programs to generate exceptions.

The absence of side effects allows pure functional languages to achieve referential transparency, which means that an expression can be replaced with its value.

In Haskell, functions never have side effects other than potentially throwing an exception. For func- tions that do input or output, the I/O action is part of the function's main effect, not a side effect, and (as we will see later) the function's type will show this fact.

1.7.1 Assignment

In conventional, imperative languages, even object-oriented ones (including C, Java, and Python), each variable has a current value (a garbage value if not yet initialized), and assignment statements can change the current value of a variable. In functional languages you can define a variable's value, but you cannot redefine it. Once a variable has a value, it has that value until the end of its lifetime.

Haskell programs can give a variable a value in one of two ways. The explicit way is to use a let clause: let pi = 3.14159 in ... This defines pi to be the given value in the expression represented by the dots. It does not define pi anywhere else. The implicit assignemnt is to put the variable in a pattern on the left hand side of an equation: len (x:xs) = 1 + len xs.

1.7.2 Builtin Haskell types

Haskell has a strong, safe and static type system. The strong part means that the system has no loop- holes; one cannot tell Haskell to e.g. consider an integer to be a pointer, as one can in C with (char *) 42. The safe part means that a running program is guaranteed never to crash due to a type error. The static part means that types are checked when the program is compiled, not when the program is run.

The command :t x returns the type of x, which can be useful to work out the type of a function. The command :set +t tells ghci to print the type as well as the value of every expression it evaluates.

- Int stands for integer which is bounded, on 32-bit machines the maximum possible Int is 2147483647 and the minimum is -2147483648.
- Integer stands for integer. The main difference is that it's not bounded so it can be used to represent really really big numbers. I mean like really big. Int, however, is more efficient.
- Float is a real floating point with single precision.
- Double is a real floating point with double the precision!
- Bool is a boolean type.
- Char represents a character. It's denoted by single quotes. A list of characters is a string.
- Tuples are types but they are dependent on their length as well as the types of their components. Note that the empty tuple () is also a type which can only have a single value: ().
- When writing our own functions, we can choose to give them an explicit type declaration with

  ```
  function_name :: Targ1 -> Targ2 -> ... -> Targn -> Treturn_value
  ```

  Here the T stands for the type of the argumetn or the return value.
- Functions that have type variables (type variables must begin with a lower case) are called polymorphic functions.

1.7.2.1 Tuple Tuples are non-homogenous unlik lists, fixed in length, size helps to determine type. Some tuple functions are:

- fst (x,y) takes a pair (x,y) and returns its first component x.
- snd (x,y) takes a pair (x,y) and returns its second component y.
- zip [x1, x2, ...xn] [y1, y2, ...,yn] takes two lists and returns them paired up as tuples and as such would give [(x1,y1),(x2,y2),...,(xn,yn)]

I the lenght of the lists don't match the longer list gets cut.

1.7.3 Type constructors

In Haskell, list is not a type; it is a type constructor. Given any type t, it constructs a type for lists whose elements are all of type t. This type is written as [t], and it is pronounced as "list of t". Haskell considers strings to be lists of characters, whose type is [Char]; String is a synonym for [Char].

The names of types and type constructors should be identifiers starting with an upper case letter (unlike functions); the list type constructor is an exception.

The notation x::y says that expression x is of type y.

Num is the class of numeric types, including the four types above. The notation 3 :: (Num t) =¿ t means that "if t is a numeric type, then 3 is a value of that type".

Normally, + is a binary infix operator, but you can tell Haskell you want to use it as an ordinary func- tion name by wrapping it in parentheses. (You can do the same with any other operator.) This means that (+) 1 2 means the same as 1 + 2.

Declaring the type of functions is required only by good programming style. It is not required by the Haskell implementation, which can infer the types of functions.

```
iota n = if n == 0 then [] else iota (n-1) ++ [n]
```

```
let iota n | n == 0 = [] | n > 0 = iota (n-1) ++ [n]
```

The offside rule says that the keywords then and else, if they start a line, must be at the same level of indentation as the corresponding if, and if the then and else expressions are on their own lines, these must be more indented than those keywords.

1.7.4 Parametric polymorphism

Here is a version of the code of len complete with type declaration:

```
len :: [t] -> Int
len [] = 0
len (_:xs) = 1 + len xs
```

This function, like many others in Haskell, is polymorphic. This is called parametric polymorphism because the type variable t is effectively a type parameter. Since the underscore matches all values in its position, it is often called a wild card.

1.7.5 Discriminated union types

Haskell allows programmers to define their own types *dataTypeConstructor = DataConstructor1|DataConstructor2*. TypeConstructor is considered an arity-0 type con- structors; given zero argument types, they each construct a type. Data constructors are constructs for values, and data types are constructs for types.

NOTE Arity means the number of arguments. A function of arity 0 takes 0 arguments, a function of arity 1 takes 1 argument, a function of arity 2 takes 2 arguments, and so on.

You do not have to use such types. If you wish, you can use the standard boolean type instead with multiple function definitions.

Here is one way to represent standard western play- ing cards:

```
data Suit = Club | Diamond | Heart | Spade
data Rank
= R2 | R3 | R4 | R5 | R6 | R7 | R8
| R9 | R10 | Jack | Queen | King | Ace
data Card = Card Suit Rank
```

On the right hand side of the definition of the type Card, Card is the name of the data constructor, while Suit and Rank are the types of its two argu- ments.

In this definition, Card is not just the name of the type (from its first appearance), but (from its sec- ond appearance) also the name of the data con- structor which constructs the "structure" from its arguments.

The function show is used for pretty printing of functions.

A disjunction (choice between) values corresponds to an enumerated type. A conjuntion of values corresponds to a structure type.

1.7.6 Discriminated union types

Haskell has discriminated union types, which can include both disjunction and conjunction at once. Since disjunction and conjunction are operations in Boolean algebra, type systems that allows them to be combined in this way are often called algebraic type systems, and their types algebraic types.

dataTypeConstructor = DataConstructorType1Type2|DataCOnstructor2Type2

Thus the type constructor is a discriminated union type and is constructed using either of the DataConstructor constructors.

1.7.7 Data constructors and DataTypes

"""1 Type constructor A type constructor is used to construct new types from given ones.

data Tree a = Tip — Node a (Tree a) (Tree a) illustrates how to define a data type with type constructors (and data constructors at the same time). The type constructor is named Tree, but a tree of what? Of any specific type a, be it Integer, Maybe String, or even Tree b, in which case it will be a tree of tree of b. The data type is polymorphic (and a is a type variable that is to be substituted by a specific type). So when used, the values will have types like Tree Int or Tree (Tree Boolean). 2 Data constructor A data constructor groups values together and tags alternatives in an algebraic data type,

data Tree a = Tip — Node a (Tree a) (Tree a) where there are two data constructors, Tip and Node. Any value that belongs to the type Tree a (I'm happy leaving the type parameter unspecified) will be a constructed by either Tip or Node. Tip is a constructor alright, but it groups no value whatsoever, that is, it's a nullary constructor. There can be only one value that will have this constructor, also conveniently denoted Tip. So nullary constructors contain no data apart from its name! For example, the Bool data type is defined to be data Bool = True — False and for all practical purposes you can just think of them as constants belonging to a type.

On the other hand, Node contains other data. The types of those data are its parameters. The first one has type a, so it's just a value of the parameter type a. This one is the value the tree node holds in it. The remaining two are the branches. Each of them have type Tree a, naturally."""

FROM http://www.haskell.org/haskellwiki/Constructor

1.7.8 Representing expressions in Haskell

```
data Expr
= Number Int
| Variable String
| Binop Binop Expr Expr
| Unop Unop Expr
```

In this example, Binop is the name of a type as well as the name of a data constructor, and the same is true for Unop. Having a type and a data constructor with the same name occurs fairly frequently in Haskell programs.

1.7.9 Case expressions

```
aaa x = case x of
          1 -> "A"
          2 -> "B"
          3 -> "C"
```

aaa x returns the corresponding letter to the number x

Optional values can be indicated by using the maybe type defined in the prelude: data Maybe t = Nothing — Just t. For any type t, a value of type Maybe t is either Nothing, or Just x, where x is a value of type t.

A recursive type have types which can themselves be of the same type or be composed of the same type. A recursive type needs a nonrecursive alternative, because without one, all values of the type would have infinite size.

Functional languages do not have language con- structs for iteration. What imperative language programs do with iteration, functional language programs do with recursion.

1.7.10 Polymorphic functions

We could also define trees like this:

```
data Tree k v
= Leaf
| Node k v (Tree k v) (Tree k v)
```

In this case, k and v are type variables, variables standing in for the types of keys and values, and Tree is a type constructor, which constructs a new type from two other types.

Extra care needs to be taken when constructing polymorphic types when dealing with order etc inside functions.

1.7.11 Comparing values for equality and order

In Haskell, comparison for equality can only be done on values of types that belong to the type class Eq, while comparison for order can only be done on val- ues of types that belong to the type class Ord. Membership of Ord implies membership of Eq, but not vice versa. The declaration of search bst should be this:

```
search_bst ::
Ord k => Tree k v -> k -> Maybe v
```

The construct Ord k =¿ is a type class constraint; it says search bst requires whatever type k stands for to be in Ord. This guarantees its membership of Eq as well. You can derive membership automatically:

```
data Type = ...
deriving (Ord, Show)
```

The automatically created comparison function takes the order of data constructors from the or- der in the declaration itself.

If the two values being compared have the same top level data constructor, the automatically created comparison function compares their arguments in turn, from left to right.

1.7.12 Data Structures

In declarative languages, data structures are immutable: once created, they cannot be changed. To update you create another version of the data structure, one which has the change you want to make, and use that version from then on.

A where clause mainexpr where name = expr has the same meaning, but has the definition of the name after the main expression.

1.8 More advanced Haskell

First order values are data. Second order values are functions whose arguments and results are first order values. In general, nth order values are functions whose arguments and results are values of any order from first up to n - 1.

In Haskell, anonymous functions are defined by lambda expressions, and you use them like this.

```
filter (\x -> (mod x 2) == 0) [0..10]
```

When defining a discriminated union type, you may give a name to each field of a data constructor, using the syntax shown by this example:

```
data Customer = Customer {
customer_id :: Int,
customer_name :: String,
customer_address :: [String]
} deriving (Eq, Show)
```

When you give names to the fields of a data con- structor this way, Haskell defines a function for each field. These functions are known as accessor or record selector functions. Each of these functions has the name of the field and returns that field. In this case, the accessor functions are

```
customer_id :: Customer -> Int
customer_name :: Customer -> String
customer_address :: Customer -> [String]
```

If any of the functions customer id, customer name or customer address is given a customer constructed with AnonCustomer, it will raise an exception. Likewise if anon id or anon po box is given a customer constructed with NamedCustomer.

User code can also raise exceptions by calling the predefined function error, whose type is String -¿ a. The argument is a string which should be a descrip- tion of the problem. It does not actually return, though formally, the return type is a type variable, so that it can match whatever its caller expects.

1.9 Partial application

Given a function with n arguments, partially applying that function means giving it its first k arguments, where $k < n$. The result of the partial application is a closure that records the identity of the function and the values of those k arguments. There is no way to partially apply a function (as opposed to an operator, see below) by supplying it with k arguments if those not are not the first k arguments.

Operators and sections If you enclose an infix operator in parentheses, you can partially apply it by enclosing its left or right operand with it; this is called a section.

```
Prelude> map (*3) [1, 2, 3]
[3,6,9]
```

You can turn an arity-2 function into an infix operator by putting back-quotes around it. You can then use section notation to partially apply either of its arguments.

```
Prelude> map (5 'mod') [3, 4, 5, 6, 7]
[2,1,0,5,5]
Prelude> map ('mod' 3) [3, 4, 5, 6, 7]
[0,1,2,0,1]
```

1.9.1 Types for partial application

In most languages, the type of a function with n arguments would be something like: f :: (at1, at2, ... atn) -¿ rt where at1, at2 etc are the argument types, (at1, at2, ... atn) is the type of a tuple containing all the arguments, and rt is the result type. To allow the function to be partially applied by supplying the first argument, you need a function with a different type: f :: at1 -¿ ((at2, ... atn) -¿ rt) This function takes a single value of type at1, and returns as its result another function, which is of type (at2, ... atn) -¿ rt.

You can keep transforming the function type until every single argument is supplied separately: f :: at1 -¿ (at2 -¿ (at3 -¿ ... (atn -¿ rt))) The transformation from a function type in which all arguments are supplied together to a function type in which the arguments are supplied one by one is called currying. In Haskell, all function types are curried.

The arrow that makes function types is right associative, so the second declaration below just shows explicitly the parenthesize implicit in the first:

```
is_longer :: Int -> String -> Bool
is_longer :: Int -> (String -> Bool)
```

What happens when you have supplied all the arguments?

There are two things you can get:

- a closure that contains all the function's argu- ments, or
- the result of the evaluation of the function.

In C and in most other languages, these would be very different, but in Haskell, as we will see later, they are equivalent.

Any function that makes a higher order function call or creates a closure (e.g. by partially applying another function) is a second order function. This means that both filter and its callers are second order functions. filter has a piece of data as an argument (the list to filter) as well as a function (the filtering func- tion). Some functions do not take any piece of data as arguments; all their arguments are functions. The builtin operator '.' composes two functions. The expression f . g represents a function which first calls g, and then invokes f on the result: $(f.g)x = f(gx)$ If the type of x is represented by the type variable a, then the type of g must be a -¿ b for some b, and the type of f must be b -¿ c for some c. The type of . itself is therefore (b -¿ c) -¿ (a -¿ b) -¿ (a -¿ c).

This style of programming is sometimes called point-free style, though value-free style would be a more accurate description, since its distinguishing characteristic is the absence of variables representing actual values.

NOTE The . operator is right associative, so head . reverse . sort parenthesizes as head . (reverse . sort), not as (head . reverse) . sort, even though the two parenthesizations in fact yield functions that compute the same answers for all possible argument values.

Given the above definition, max xs is equivalent to (head . reverse . sort) xs, which in turn is equivalent to head (reverse (sort xs)).

1.9.2 Monads

A monad is a type constructor M that supports two operations:

- A sequencing operation, denoted ¿¿=, whose type is M a -¿ (a -¿ M b) -¿ M b.
- An identity operation, denoted return, whose type is a -¿ M a.

Basically, the idea is that M a is a value of type a plus possibly something extra. For example, if M is the MaybeOK type constructor, that something extra is an indication whether an error has occurred so far. • You can take a value of type a and use the (misnamed) identity operation to wrap it in the monad's type constructor. • Once you have such a wrapped value, you can use the sequencing operation to perform an op- eration on it. The ¿¿= operation will unwrap its first argument, and then typically it will invoke the function given to it as its second argument, which will return a wrapped up re- sult. NOTE You can apply the sequencing operation to any value wrapped up in the monad's type con- structor; it does not have to have been generated by the monad's identity function.

1.9.2.1 The Maybe and MaybeOK monads

The obvious ways to define the monad operations for the Maybe and MaybeOK type constructors are these:

```
-- monad ops for Maybe
return x = Just x
(Just x) >>= f = f x
Nothing >>= _ = Nothing
-- monad ops for MaybeOK
return x = OK x
(OK x) >>= f = f x
(Error m) >>= _ = Error m
```

In a sequence of calls to >>=, as long as all invoca- tions of f succeed, returning Just x for the Maybe monad and OK x for the MaybeOK monad, you keep going. Once you get a failure indication, which is Nothing for the Maybe monad and Error m for the MaybeOK monad, you keep that failure indication and per- form no further operations.

1.9.3 Why you may want these monads

Suppose you want to encode a sequence of opera- tions that each may fail. Here are two such opera- tions:

```
maybe_head :: [a] -> MaybeOK a
maybe_head [] = Error "head of empty list"
maybe_head (x:_) = OK x
maybe_sqrt :: Int -> MaybeOK Double
maybe_sqrt x =
  if x >= 0 then
    OK (sqrt (fromIntegral x))
  else
    Error "sqrt of negative number"
```

How can you encode a sequence of operations such as taking the head of a list and computing its square root?

Simplifying code with monads

```
maybe_head :: [a] -> MaybeOK a
maybe_sqrt :: Int -> MaybeOK Double
maybe_sqrt_of_head ::
  [Int] -> MaybeOK Double
-- definition not using monads
maybe_sqrt_of_head l =
  let mh = maybe_head l in
    case mh of
      Error msg -> Error msg
```

```
    OK h -> maybe_sqrt h
-- simpler definition using monads
maybe_sqrt_of_head l =
  maybe_head l >>= maybe_sqrt
```

NOTE The monadic version is simpler because in that version, the work of checking for failure and handling it if found is done once, in the MaybeOK monad's sequence operator.

Note that the two occurrences of Error m in the first definition are of different types; the first is type MaybeOK Int, while the second is of type MaybeOK Double. The two occurrences of Error m in the definition of the sequence operation for the MaybeOK monad are similarly of different types, MaybeOK a for the first and MaybeOK b for the second.

1.10 I/O actions in Haskell

Haskell has a type constructor called IO. A function that returns a value of type IO t for some t will return a value of type t, but can also do input and/or output. Such functions can be called I/O functions or I/O actions. Haskell has several functions for reading input, including:

```
getChar :: IO Char -- returns next character in input
getLine :: IO Strin -- returns next line in input
putChar :: Char -> IO () -- rints given char
putStr :: String -> IO () -- prints given string
putStrLn :: String -> IO () -- prints given string plus newline char
print :: (Show a) => a -> IO () -- uses the show function
```

The type () is the type of 0-tuples (tuples contain- ing zero values). There is only one value of this type, the empty tuple, which is also denoted (). NOTE The notion that a function that returns a value of type IO t for some t actually does input and/or output is only approximately correct; we will get to the fully correct notion in a few slides. Since there is only one value of type (), variables of this type carry no information. Haskell represents a computation that takes a value of type a and computes a value of type b as a function of $type a -> b$. Haskell represents a computation that takes a value of type a and computes a value of type b while also possibly performing input and/or output (subject to the caveat above) as a function of type $a -> IO b$.

1.10.1 IO monad

The type constructor IO is a monad.

- The identity operation: return val just returns val (inside IO) without doing any I/O.
- The sequencing operation: $f >>= g$
  - calls f, which may do I/O, and which will re- turn a value rf that may be meaningful or may be (),
  - calls g rf (passing the return value of f to g), which may do I/O, and which will return a value rg that also may be meaningful or may be (),
  - returns rg as the result of $f >>= g$.

  You can use the sequencing operation to create a chain of any number of I/O actions.

Example of monadic I/O: hello world

```
1 hello :: IO ()
2 hello = putStr "Hello, " >>= \_ -> putStrLn "world!"
3 main :: IO ()
4 main = hello
```

- This code has two I/O actions connected with >>=.
- The first is a call to putStr. This prints the first half of the message, and returns ().
- The second is an anonymous function. It takes as argument and ignores the result of the first action, and then calls putStrLn to print the second half of the message, adding a newline at the end. The result of the action sequence is the result of the last action. NOTE In this case, the resulf of the last action will always be (). Actually, there is a third monad operation besides return and ¿¿=: the operation ¿¿. This is identical to ¿¿=, with the exception that it ignores the return value of its first operand, and does not pass it to its second operand. This is useful when the second operand would otherwise have to explicitly ignore its argument, as in this case. With ¿¿, the code of this function could be slightly

```
1 hello :: IO ()
2 hello = putStr "Hello, " >> putStrLn "world!"
3 main :: IO ()
4 main = hello
```

```
1 greet :: IO ()
2 greet = putStr "Greetings! What is your name?"
3       >>=
4       \_ -> getLine
5       >>= \name -> (
6         putStr "Where are you from? "
7         >> getLine
8         >>= \town ->
9           let msg = "Welcome, " ++ name ++ " from " ++ town
    in
10          putStrLn msg
11      )
12
13 main :: IO ()
14 main = greet
```

1.10.2 do blocks

Code written using monad operations is often ugly, and writing it is usually tedious.

Each element of a do block can be

- an I/O action that returns an ignored value, usually of type (), such as the calls to putStr and putStrLn below;
- an I/O action whose return value is used to define a variable, such as the calls to getLine below;
- a let clause whose scope extends to the end of the block.

Working around the operator priority problem There are two main ways to fix this problem:

```
putStrLn ("Welcome, " ++ name ++
" from " ++ town)
putStrLn $ "Welcome, " ++ name ++
" from " ++ town
```

The first simply uses parentheses to delimit the pos- sible scope of the ++ operator. The second uses another operator, $, which has lower priority than ++, and thus binds less tightly. The main function invoked on the line is thus.

Haskell programmers usually think of functions that return values of type IO t as doing I/O as well as returning a value of type t.

The correct way to think about such functions is that they return two things: • a value of type t, and • a description of an I/O operation. The monadic operator ¿¿= can then be understood as taking descriptions of two I/O operations, and returning a description of those two operations be- ing executed in order. The monadic operator return simply associates a description of a do-nothing I/O operation with a value.

```
1 main :: IO ()
2 main = do
3   putStrLn "Table of squares:"
4   print_table 1 10
5
6 print_table :: Int -> Int -> IO ()
7 print_table cur max
8   | cur > max = return ()
9   | otherwise = do
10    putStrLn (table_entry cur)
11    print_table (cur+1) max
12
13 table_entry :: Int -> String
14 table_entry n = (show n) ++ "^2 = " ++ (show (n*n))
```

NOTE The definition of print table uses guards. If cur ¿ max, then the applicable right hand side is the one that follows that guard expression; if cur ¡= max, then the applicable right hand side is the one that follows the keyword otherwise.

1.10.3 Non-immediate execution of I/O actions

Just because you have created a description of an I/O action, does not mean that this I/O action will eventually be executed. Haskell programs can pass around descriptions of I/O operations.

1.10.4 Debuggin in Haskell

- If f does I/O, which means that it returns a description of an I/O operation that you know will be executed, then you can simply print the diagnostics you need as part of that operation.
- If f does not do I/O, or if the I/O operation whose description it returns is not executed, this will not work.
- The type of unsafePerformIO is IO t -¿ t. You give it as argument an I/O operation, which means a function of type IO t. unsafePerformIO calls this function. The function will return a value of type t and a description of an I/O operation. unsafePerformIO executes the described I/O operation and returns the value. Here is an example:

```
sum :: Int -> Int -> Int
sum x y = unsafePerformIO $ do
putStrLn ("summing " ++
(show x) ++ " and " ++ (show y))
return (x + y)
```

1.10.5 The State Monad

Allows state to be passed around. The constructor State holds a function, not just a simple value like Maybe's Just. First of all the State monad is just an abstraction for a function that takes a state and returns an intermediate value and some new state value.

Just as you can chain together functions using (.) as in (+1) . (*3) . head :: (Num a) =¿ [a] -¿ a, the state monad gives you (¿¿=) to chain together functions that look essentially like :: a -¿ s -¿ (a,s)into a single function :: s -¿ (a,s).

1.10.6 THe Haskell Module System

- Every module is stored in it's own source file.
- The module must have a name, which must be an identifier starting with an upper case letter, and it should match the name of the file containing the module, without the .hs suffix.
- Every module defines one or more items. These can be types, functions or some other things we will introduce later.
- Every module specifies what subset of the items it defines it also exports to other mod- ules.
- Every module specifies what other modules it imports items from. It can also specify exactly what items it imports from each of those mod- ules.
- Every module must start with a module declaration module ModuleName where. In real programs, the module declaration should contain a parenthesized list of the names of items that this module exports to other modules:

```
module ModuleName
(TypeNameA, functionNameB)
where
```

  The where keyword is followed by the body of the module.
- The set of items exported by a module is its interface to the rest of the program.

SKIP section 2 slide 51 to

2 Logic Programming

While functional programming languages are based on the lambda calculus, logic programming languages are based on another mathematical formalism: the predicate calculus.

Syntax Rules

- The names of variables are identifiers starting with upper case letters. (Opposite to haskell)
- The names of type constructors and function symbols are identifiers starting with lower case letters. (Logic programming calls data con- structors "function symbols".) (Type constructors are opposite, what are function symbols?)
- The names of functions and predicates are identifiers starting with lower case letters. (same)
- The names of modules are identifiers starting with lower case letters. (opposite?)
- unlike haskell, mercury does not permit quotes in identifiers.
- In mercury each term is a function symbol followed by zero of more arguments (shown in parenthesis , sperated by commas).
- [] for the empty list
- Rather than x:xs mercury uses $[X|XS]$
- A term is a ground term if it contains no variables, and it is a nonground term if it contains at least one variable.
- A substitution is a mapping from variables to terms. The only relevant mapped-to-terms are ground terms in Mercury.
- Applying the substitution to a term means consis- tently replacing all occurrences of each variable in the map with the term it is mapped to.
- Applying a substitution to a term gives you an instance of that term.
- The term that results from applying a substitution θ to a term t is denoted $t\theta$. A term u is therefore an instance of a term t if there is some substitution θ s.t. $u = t\theta$. A substitution θ unifies two terms t and u if $t\theta = u\theta$.
- An atom is a predicate symbol followed by zero or more arguments. Atoms can be ground or non ground.
- In the mind of the person writing a logic program,
  - each constant (function symbol of zero arity) stands for an entity in the world of the pro- gram;
  - each function (function symbol of arity n where n ¿ 0) stands for a function from n entities to one entity in the world of the program; and
  - each predicate of arity n stands for a particular relationship between n entities in the world of the program.

  This mapping from symbols in the program to the world of the program is called an interpretetaion.
- You can think of a Predicate with n arguments as a function from all possible combinations of n terms to a truth value. Or, each predicate as a set of tuples of n terms, wehere every tuple is implicityly mapped to true, and every non existant tuple mapped to false.
- A predicate definition consists of a set of clauses.
- A rule consists of a head (an atom) and a body (a goal) speerated by an implication sign which in mercury is : −. In predicate calculus, the usual from of the goal is a conjunction of atoms. A rule is a conditional assertion: it asserts that the head is true if the body is true.
- logical symbols not important.
- A logic program consists of a set of predicate definitions.
- The semantics of this program (its meaning) is the set of its logical consequences.
- Ground queries are used to find if an assertion is true or false
- nonground queries are used to find all substitution sthat make the query atom true.
- The SLD resolution algorithm: recursivly unify selected queries with the head of a clause, applying unifying substitutions to the clause and unsolved subgoals. The recursion terminates at facts.
- A static type system is desireable as it narrows down the meaning of predcates to what programmers are thinking.

3 Introduction to Mercury

- Type definitions are effictavily identical to haskell, syntax is however differn't.

```
:- type cord(T)
    ---> nil
    ; leaf(T)
    ; branch(cord(T), cord(T)).
```

- Comments use % or $/*blah*/$
- Programmers may name the fields of funciton symbols

```
:- type cd
---> cd(
artist :: string,
tracks :: list(track_info)
).
..., CD = cd(Artist, _), ...
..., Artist = CD ^ artist, ...
```

  Both of the above lines unify Artist with the first field of CD, but the second one doesn't have to be updated if a later change adds some new fields to the cd function symbol.

```
NewCD = CD ^ artist := "Beatles" unifies
```

  NewCD with a term that is the same as CD, except for the artist field containing "Beatles".
- Programmers can declare the types of the arguments of predicates like this:

```
:- pred append(list(T), list(T), list(T))
```

- Besides predicates, Mercury also supports functions, whose types can also be declared, like this:

```
:- func merge(list(T), list(T)) = list(T)
```

- You have to provide type declarations for predicates and functions exported from their defining module.
- Mercury lets programmers declare one or more modes for each predicate.

```
:- mode append(in, in, out).
:- mode append(out, out, in).
```

  Each corresponds to one way in which the predicate can be used (some form of queries might not make sense). Arguments with mode in are input arguments, it is the responsibility of the caller to decide what the value of that argument should be. Arguments with mode out are output arguments: it is the responsibility of this predicate (the callee) to decide what the value of that argument should be. Function don't require mode declarations as they

are implicit. Each mode of a predicate or funciton is called a procedure. A procedure consumes its input arguments and produces or generates its output arguments. Mercury allows the caller to override the modes set by the callee, thus apredicate can be called with only it's out variables.
If a predicate has only one mode, the programmer may use a predmode declaration to declare the predicate's type, mode and determinism all at once.
The mode analysis pass of the Mercury compiler works on each procedure separately. Amongst other tasks,
  - it identifies the modes of all the calls and builtins in the procedure body;
  - it reorders conjunctions to ensure that the goal that produces a variable comes before all the goals that consume it; and
  - it checks that at the end of the procedure body, the value of every output argument is known.

  Mercury thus supports reversible code; very few other languages do.
- Types of unifications (assertions of equality of terms):
  - A unification of the form $X = f(Y1, ..., Yn)$ is a construction unification if $Y1..Yn$ are input and X is output.
  - A unification of the form $X = f(Y1, ..., Yn)$ is a deconstruction unification if X is input and $Y1..Yn$ are output.
  - A unification of the form $X = Y$ is an assignment unification if X is input and Y is output, or vice versa.
  - A unification of the form $X = Y$ is a test unification if X and Y are both input.
- Mercury always generates separate code for each procedure, even when they are different modes of the same predicate.
- The determinism of a procedure gives lower and upper bounds on the number of solutions it may have.

determinism	min num	max num
det	1	1
semidet	0	1
multi	1	∞
nondet	0	∞

- If the procedure has no output arguments, it corresponds to a Haskell function that returns a boolean: True for success, False for failure.

3.1 Mercury Goals

The bodies of Mercury clauses cna contain any of these constructs
- unification: $Term1 = Term2$
- call: $p(Term1, \cdots, Termn)$
- conjunction: $G1, \cdots, Gn$. Conjunciton requires all these things to happen for one solution.
- disjunction: $(G1; ...; Gn)$. Disjunctions introduce the possibility of generating more than one solution, since each disjunct provides a different way to prove a query. Disjunction is recognised as or and is the only way to provide multiple solutions. Disjuncts are seperated by ";". When two disjuncts unify with differn't function symbols, at most one of these unifications can suceed: turning the disjunciton into a switch. If disjuncts together make all the function symbols of a type, this is called a complete switch. If one arm generates a value for a variable then all arms must clearly. And they must be of the same type?
- negation: not G.
- if-then-else: $(C- > T; E)$ or (if C then T else E). If then else look like disjunctions with an extra condition on the first disjunct. A variable generated by the condition can only be used in the then part. $(C- > T; E)$ is a goal if T and E are goals, otherwise it is an expression. C must always be a goal.
- existential quantification: some [V1,...,Vn] G is true if for some values of [V1,...,Vn], G is true.
- universal quantification: all [V1,...,Vn]
- implications: $G1 => G2, G1 <= G2, G1 <=> G2$. $G1 => G2$ is true if whenever G1 is true, G2 is aswell.
- higher order predicate calls: P(Term 1, ..., Term n)
- higher order function calls: Term = F(Term 1, ..., Term n)

3.2 Mercury modules

-
```
:- module <module name>
:- interface
<interface section: declares types, predicates, functions to
     export, documentation of the functionality of a program>
:- implementation
<implementation section: contains declarations of private
     variables, clauses of predicates and functions, and bothe
     the declarations and clauses of the private predicates and
     functions>
```
- You can export a type as an abstract type by declaring only its name in the niterface section and keeping its definition in the implementation section.
- An import declaration lists one or more modules that this module wants to import from. An imported module that defines a type that is used in the interface of this module must be imported in the interface section. An imported module that does not define any type used in the interface of this module must be imported in the implementation section.
- If the name of a type, predicate or function is ambiguous because the name is defined by more than one of these modules, you must qualify the name: the name is preceded by the module name and a dot.

3.3 Handling State

- Mercury views I/O predicates as describing a relationship between two states of the world: the one before the action and the one after. A state of the world is represented by a value of the special type io.
- Since the world has only one state at any one time, we must make sure that old states of the world are never referred to again.
- The mode di (destructive input) says that the storage of this argument may be destroyed (reused) in the predicate, so the argument must be the last reference to that storage. The mode uo (unique output) says that there are no other references to the value returned for this argument.

```
:- pred hello(io::di, io::uo) is det.
hello(S0, S) :-
io.write_string("Hello, ", S0, S1),
io.write_string("world\n", S1, S).
```

- !S stands for two arguments representing the current and the next states of data structure S. The first variable can be accessed with !.S and the second with :.S.

3.4 Higher order programming

- You can easily make functions variables toother functions, just by denoting them with an uppercase letter. The mode of the function is assumed by the mercury compiler. Functions assed in as variables are lambda expressions?
- For higher order values representing predicates, the programmer has to provide the predicate's mode and determinism as well as its type.

```
:- pred list.filter(pred(T),
    list(T), list(T)).
:- mode list.filter(
    in(pred(in) is semidet),
    in, out) is det.
```

- Singleton variables, which are variables that occur only once in a clause, and should start with an underscore.
- section 4 slide 45, wtf?
- You can partially apply a function or a predicate by specifying the values of its first k arguments.

```
:- pred shorter_than(int::in, list(T)::in)
    is semidet.
shorter_than(Limit, List) :-
    list.length(List) < Limit.
    ... list.filter(shorter_than(5),
    Lists, ShortLists) ...
```

3.5 Using Mercury

- compilation: mmc prog.m
- To compile a multi-module Mercury program in which the main module is mainmodule.m can be compiled with mmc –make toplevel.
- Preparing a program for debugging is achieved with mmc –make –debug mainmodule
- The files generated by non-debug compiles are in- compatible with debugging, so before you give the above command, you should remove them all with mmc –make mainmodule.realclean
- to invoke the mercury debugger: mdb ./mainmodule. mdb will stop at each event in the program. Debugger events and commands are on page 82.
- Declarative debuggers guide investigation.

4 Declarative Programming techniques

- In a programming language that uses lazy evalua- tion, an expression is not evaluated until its value is actually needed (input to arithmetic operator, match against a patter, output the value).
- A suspension is a pointer to a function together with all the arguments that you need to give to that function. Aka a promise. Aka a thunk.
- Parametric polymorphism is the form of polymorphism where types include type varaibles.
- Lazyness ensures that programmers can create their own data structures.
- Laxyness adds two sorts of slowdowns: execution creates a lot of suspensions that will be evealuated anyway. Every access to a value must first check whether the value have been materialised yet. Programmers can eliminate som e of the lazyness using the built in operator cadeA 'seq' codeB which returns codeB but first forces the top data constructor of codeA to be known . Laxyness can also make it harder for a programmer to understand where the program is spending most of its time and what parts of the program allocate most of its memeory.
- The value of an expression whose evaluation loops infinitely or throws an exceptino bottom.
- A function is strict if it always needs the values of all its arguments. When the Haskell code generator sees a call to a strict function, instead of generating code that cre- ates a suspension, it can generate the code that an imperative language compiler would generate: code that evaluates all the arguments, and then calls the function. NOTE Eager evaluation is also called strict evalu- ation, while lazy evaluation is also called nonstrict evaluation.

4.1 Higher order programming

- Reduction operations reduce a list to a single value. fold is a reduction operation:

```
foldl :: (v -> e -> v) -> v -> [e] -> v
foldl _ base [] = base
foldl f base (x:xs) =
    let newbase = f base x in
    foldl f newbase xs

foldr :: (e -> v -> v) -> v -> [e] -> v
foldr _ base [] = base
foldr f base (x:xs) =
    let fxs = foldr f base xs in
    f x fxs

balanced_fold :: (e -> e -> e) -> e ->
    [e] -> e
balanced_fold _ b [] = b
balanced_fold _ _ (x:[]) = x
balanced_fold f b l@(_:_:_) =
    let
        len = length l
        (half1, half2) =
            divide_list (div len 2) l
        value1 = balanced_fold f b half1
        value2 = balanced_fold f b half2
    in
        f value1 value2
```

You can define sum, product and concatenation in terms of both foldl and foldr because addition and multiplication on integers, and list append, are all associative operations, however they may not run as effiecantly as each other.

- A list comprehension consists of
  - a template (an expression, which is usually a variable)
  - one or more generators (each of the form $var < -list$),
  - zero or more tests (boolean expressions),
  - zero or more let expressions defining local variables.

4.2 Memory-conscious programming

Haskell programs first begin with some data structure, traverse it createing another, traverse it creating yet another. Optimising code such that intermediate data structures are removed is called deforestation.

- Repeeated appends to the end of lsits take time that is quadratic in the final length of the list. We can avoid this situation by switching to cords instead of lsits.

```
data Cord a
    = Nil
    | Leaf a
    | Branch (Cord a) (Cord a)

append_cords :: Cord a -> Cord a -> Cord a
append_cords a b = Branch a b

cord_to_list :: Cord a -> [a]
cord_to_list Nil = []
cord_to_list (Leaf x) = [x]
cord_to_list (Branch a b) =
    (cord_to_list a) ++ (cord_to_list b)
```

- using an accumulator to store sucessive entries is much more effiecant than appending like above in cord to list.
- fat lists are lists that can contian more than one item. Functinos operating on fat lists need to consider extra cases, standard library functions thus will not be appropiate.

4.3 Design Patterns