Lecture 04: Basic Haskell Continued
  o Polymorphic Types
  o Type Inference with Polymorphism
  o Standard Types: Bools, Integers
  o Function definitions in more detail:
    if-then-else, guards, where

Reading: Hutton Chapter 3, 4.1 – 4.4
Polymorphic Types

Recall: Many functions (and data types) do not need to know everything about the types of the arguments and results.

Many data types and most list-processing functions are of this kind:

```haskell
data List a = Nil | Cons a (List a)
data Pair a b = P a b
data Triple a b c = T a b c
```

```haskell
append :: List a -> List a -> List a
reverse :: List a -> List a
```

Check: What is the type of

- `P`?
- `head`?
- `tail`?
Polymorphic Types

Recall: Many functions (and data types) do not need to know everything about the types of the arguments and results.

Many data types and most list-processing functions are of this kind:

data List a = Nil | Cons a (List a)
data Pair a b = P a b
data Triple a b c = T a b c

append :: List a -> List a -> List a

reverse :: List a -> List a

head (Cons x _) = x
tail (Cons x xs) = xs

Check: What is the type of
P :: a -> b -> Pair a b
head :: List a -> a
tail :: List a -> List a
Polymorphic Types

Polymorphic type inference, based on unifying type expressions, determines the types of all expressions by looking at all the places where types must be the same:

pred :: Nat -> Nat

-- return the first n elements of a list

take Zero xs = xs
take n Nil = Nil
take n (Cons x xs) = Cons x (take (pred n) xs)

- Same variable in a rule must be same type.
- Arguments each each position and result types must be the same.
- Inputs to function and type of arguments must be same.

Type of function must be:
take :: Nat -> List a -> List a
Polymorphic Types

Polymorphic type inference, based on unifying type expressions, determines the types of all expressions by looking at all the places where types must be the same:

\[
pred :: \text{Nat} \rightarrow \text{Nat}
\]

-- return the first \(n\) elements of a list

\[
take :: \text{Nat} \rightarrow \text{List}\;a \rightarrow \text{List}\;a
\]

\[
\begin{align*}
take\;\text{Zero}\;xs &= xs \\
take\;n\;\text{Nil} &= \text{Nil} \\
take\;n\;(\text{Cons}\;x\;\text{xs}) &= \text{Cons}\;x\;(\text{take}\;(\text{pred}\;n)\;\text{xs})
\end{align*}
\]

But **ALL** expressions must have appropriate types, using rule:

\[
\begin{array}{c}
f :: \text{A} \rightarrow \text{B} \\
e :: \text{A}
\end{array} \\
(f\;e) :: \text{B}
\]
Polymorphic Types

Polymorphic type inference determines the most general type that a function can have. This involves accounting for all the type constraints implied when you examine two type expressions that must apply to a single context (say an argument to a function):

data Bool = False | True
data Nat = Zero | Succ Nat
data Triple a b c = T a b c -- ex: (Zero, Zero, True)

T :: a -> b -> c -> Triple a b c

Example 1:

Let \( s = (T \text{ Zero } x \ y) \) -- \( x, y, z, w \) can have any types
Let \( t = (T z \ False \ w) \)

If \( s \) and \( t \) have to have the same type, what would that type be?

\( (\text{Triple Nat Bool } a) \)

and furthermore, we must have \( z :: \text{Nat}, x :: \text{Bool} \) but \( y, w \) can be anything as long as they are the same type \( a \)!
Polymorphic Types

data Bool  =  False | True
data Nat   =  Zero  |  Succ Nat
data Triple a b c  =  T a b c  --  ex: (A, C, B)

Example 2:

Let  s  =  (T True x     False)  --  x, y, z, w  have unknown types
Let  t  =  (T z     False x    )

If  s  and  t  have to have the same type, what would that type be?

For  s  we have  (Triple Bool a     Bool)
For  t  we have  (Triple b     Bool c    )

For the types to be the same we would have to have  a  =  b  =  c  =  Bool:

   (Triple Bool Bool Bool)

This process is like “two-sided matching” and is called Unification:

   (Triple Bool b Bool)  \rightarrow  (Triple Bool Bool Bool)  \leftarrow  (Triple a Bool c)
Polymorphic Types

data Bool = False | True
data Nat = Zero | Succ Nat
data Pair a b = P a b -- ex: (A, B)
data Triple a b c = T a b c -- ex: (A, C, B)

Example 3:

Let s = (T True x x) -- x, y can have any types
Let t = (T y False Zero)

If s and t have to have the same type, what would the type of T be?

Answer: No type exists, as x would have to simultaneously be Bool and Nat, so it is contradictory and is a type error! The type expressions

(Triple Bool a a) and (Triple b Bool Nat)

can NOT be unified!
Polymorphic Types

Example:

\[ f :: (\text{Pair } a \ b) \rightarrow (\text{Triple } a \ b \ b) \]
\[ f \ (P \ x \ y) = (T \ x \ y \ y) \]

\[ k :: (\text{Pair } \text{Bool } a) \rightarrow (\text{Pair } a \ \text{Bool}) \]
\[ k \ (P \ x \ y) = (P \ y \ x) \]

\[ \text{test } x = (f \ (k \ x)) \]

\[ c = (P \ \text{Pair } \text{Bool } a) \]

Reading: Hutton Ch. 3.7

data Bool = False | True
data Nat = Zero | Succ Nat
data Pair a b = P a b
data Triple a b c = T a b c

unify these two

\[ f :: A \rightarrow B \quad e :: A' \]
\[ (f \ e) :: B' \]

\[ k :: (\text{Pair } \text{Bool } a) \rightarrow (\text{Pair } a \ \text{Bool}) \quad x :: c \]

\[ (k \ x) :: (\text{Pair } a \ \text{Bool}) \]
Polymorphic Types

Unification determines what type a function must have:

\[ f :: (\text{Pair}\ a\ b) \rightarrow (\text{Triple}\ a\ b\ b) \]
\[ f\ (\text{P}\ x\ y) = (\text{T}\ x\ y\ y) \]

\[ k :: (\text{Pair}\ \text{Bool}\ a) \rightarrow (\text{Pair}\ a\ \text{Bool}) \]
\[ k\ (\text{P}\ x\ y) = (\text{P}\ y\ x) \]

test\ x = (f (k x))

Unify:

\[ (\text{Pair}\ a'\ b') \]
\[ (\text{Pair}\ a\ \text{Bool}) \]

\[ c = (\text{Pair}\ \text{Bool}\ a) \quad a = a' \quad b' = \text{Bool} \]
Polymorphic Types

Unification determines what type a function must have:

\[
f :: (\text{Pair } a \ b) \rightarrow (\text{Triple } a \ b \ b) \\
f \ (P \ x \ y) = (T \ x \ y \ y)
\]

\[
k :: (\text{Pair } \text{Bool } a) \rightarrow (\text{Pair } a \ \text{Bool}) \\
k \ (P \ x \ y) = (P \ y \ x)
\]

test \ x = (f \ (k \ x))

Unify these two

\[
f :: A \rightarrow B \\
e :: A' \\
(f \ e) :: B'
\]

\[
k :: (\text{Pair } \text{Bool } a) \rightarrow (\text{Pair } a \ \text{Bool}) \\
x :: c \\
(k \ x) :: (\text{Pair } a \ \text{Bool})
\]

\[
f :: (\text{Pair } a' \ b') \rightarrow (\text{Triple } a' \ b' \ b') \\
(f \ (k \ x)) :: (\text{Triple } a' \ \text{Bool } \text{Bool})
\]

c = (\text{Pair } \text{Bool } a) \\
a = a' \\
b' = \text{Bool}

test :: ??

Reading: Hutton Ch. 3.7

data \text{Bool} = \text{False} | \text{True} \\
data \text{Nat} = \text{Zero} | \text{Succ } \text{Nat} \\
data \text{Pair } a \ b = P \ a \ b \\
data \text{Triple } a \ b \ c = T \ a \ b \ c
Polymorphic Types

Unification determines what type a function must have:

\[ f :: (\text{Pair } a \ b) \rightarrow (\text{Triple } a \ b \ b) \]
\[ f \ (\text{P } x \ y) = (\text{T } x \ y \ y) \]

\[ k :: (\text{Pair } \text{Bool} \ a) \rightarrow (\text{Pair } a \ \text{Bool}) \]
\[ k \ (\text{P } x \ y) = (\text{P } y \ x) \]

\[ \text{test } x = (f \ (k \ x)) \]

Unify these two:

\[ f :: A \rightarrow B \quad e :: A' \]
\[ (f \ e) :: B' \]

Another example at end of the slides.....
Adding Numbers to Bare Bones Haskell: Built-in Numeric Types

Int -- fixed-precision integers
Integer – arbitrary-precision integers
Float – 32-bit floating-point
Double – 64-bit float-point
Rational

Operators +, -, *, == are the same in Haskell as in Python, Java, & &C except:

exponentiation: \( x^3 \) (only for integer exponents)
\( x^{**}3.1415 \) (only for floating-point exponents)
unary minus: \((-9)\) (must use parentheses)
not equals: /=
Integer division: (div 10 7) => 1
Floating-point division ( 3.4 / 4.9) => 0.693877551020408
modulus: (mod 10 7) => 3

We’ll explore types in detail next week..... for now we will only use Integers.
Built-in Numeric Types: Infix vs Prefix Functions

We have been using prefix notation up to this point and two of the new functions we have for Integers are given in this form:

- Integer division: \( \text{div } 10 \ 7 \) => 1
- modulus: \( \text{mod } 10 \ 7 \) => 3

But most (binary) arithmetic operators are infix:

\[
\begin{align*}
(4 \ * \ 3) &= 12 \\
(2 \ - \ 3) &= (-1)
\end{align*}
\]

There are also postfix (unary) functions in mathematics:

5! => 120

as well as mixfix for functions of more than 2 arguments:

\[
\begin{align*}
(3 < 4 \ ? \ 2 : 5) &= 2 \\
(\text{if } 6 < 4 \ \text{then } 2 \ \text{else } 5) &= 5
\end{align*}
\]

Remember:

\[\Rightarrow\] means “evaluates to”

The term operator generally refers to a function which is used with infix notation: \(+\ *\ ^\) etc. We’ll just call them functions.
Built-in Numeric Types: Infix vs Prefix Functions

Haskell is completely flexible about prefix and infix notation for binary (two argument) functions:

To use a function defined in **prefix form as infix** surround it by **backquotes**:

- `div 10 7 => 1`  
  `10 `div` 7` => 1
- `mod 10 7 => 3`  
  `10 `mod` 7` => 3

To use a function defined in **infix form as prefix** surround it by **parentheses**:

- `(10 + 7) => 17`  
  `((+) 10 7)` => 17
- `(10 ^ 3) => 1000`  
  `((^) 10 3)` => 1000

To define an **infix** function it must consist of special symbols (no letters) and the type declaration must use prefix (with parentheses):

```haskell
(!!) :: List a -> Integer -> a -- select the nth element
(!!) (Cons x _ ) 0 = x
(!!) (Cons x xs) n = xs !! (n-1)
```
Prelude’s Boolean Type

So Haskell defines the Bool type in the Prelude as follows (Hutton p. 281):

```haskell
data Bool = False | True

not :: Bool -> Bool
    not True = False
    not False = True

(&&) :: Bool -> Bool -> Bool
    False && _ = False
    True && b = b

(||) :: Bool -> Bool -> Bool
    False || b = b
    True || _ = True
```

So you can just use the Bool defined in Prelude from now on...
Now we will look at ways to extend BB Haskell to make it easier to use!

An important predefined function: Conditional expressions

Real Haskell

\[(\text{if } x \text{ then } y \text{ else } z)\]

Bare-Bones Haskell

\[(\text{cond } x \ y \ z)\]

\[
\begin{align*}
\text{Bool} & \quad \text{Must be same type because expressions can only return one type:} \\
(\text{if } 5 < 8 \text{ then } 6 \text{ else } 8) \times 3 & \Rightarrow 18
\end{align*}
\]

So, the type is: \text{Bool} \rightarrow a \rightarrow a \rightarrow a
Functions Definitions: Where Expressions

It is very common to need “helper functions” to define a function:

\[
\begin{align*}
\text{remDup} :: \text{List Integer} & \rightarrow \text{List Integer} \\
\text{remDup} \; \text{Nil} & = \text{Nil} \\
\text{remDup} \; (\text{Cons} \; x \; \text{Nil}) & = (\text{Cons} \; x \; \text{Nil}) \\
\text{remDup} \; (\text{Cons} \; x \; \text{xs}) & = \text{remDup}' \; x \; (\text{reDup} \; \text{xs})
\end{align*}
\]

\[
\begin{align*}
\text{remDup'} :: \text{List Integer} & \rightarrow \text{List Integer} \\
\text{remDup}' \; x \; \text{xs} & = \text{if} \; x = \text{head} \; \text{xs} \; \text{then} \; \text{xs} \; \text{else} \; (\text{Cons} \; x \; \text{xs})
\end{align*}
\]

But why should \text{remDup}' be visible anywhere but the definition of \text{remDup}?

What if you just want to call it \text{f} or \text{helper}? Then you can’t use these names anywhere again in this file!

Would be nice to have a “local definition” of the helper functions....
Functions Definitions: Where Expressions

Is is a good idea to indent your helper functions using the keyword where:

```haskell
remDup :: List Integer -> List Integer
remDup Nil = Nil
remDup (Cons x Nil) = (Cons x Nil)
remDup (Cons x xs) = remDup' x (reDup xs)
  where remDup' :: List Integer -> List Integer
        remDup' x xs = if x == head xs then xs else (Cons x xs)

len x y = sqrt (sq x + sq y)
  where sq a = a * a

doStuff :: Int -> String
doStuff x | x < 3 = report "less than three"
           | otherwise = report "normal"
  where report y = "the input is " ++ y
```
Haskell Types

Guarded Equations

Consider the following functions to find minimum and maximum of two Integers

\[
\text{min} :: \text{Integer} \to \text{Integer} \to \text{Integer} \\
\text{min} \ x \ y = \text{if } x \leq y \text{ then } x \text{ else } y
\]

\[
\text{max} :: \text{Integer} \to \text{Integer} \to \text{Integer} \\
\text{max} \ x \ y = \text{if } x \geq y \text{ then } x \text{ else } y
\]

This is a fairly common pattern, where we test some Boolean condition on the parameters. In Haskell, this can be equivalently done using “Guarded Matching”:

\[
\text{min} :: \text{Integer} \to \text{Integer} \to \text{Integer} \\
\text{min} \ x \ y | x \leq y = x \quad \text{match succeeds only if guard true} \\
\text{min} \ x \ y = y
\]

\[
\text{max} :: \text{Integer} \to \text{Integer} \to \text{Integer} \\
\text{max} \ x \ y | x \geq y = x \quad \text{match succeeds only if guard true} \\
| \text{otherwise} = y \quad \text{otherwise is always True}
\]
Haskell Types

There are usually many different ways of defining a function, and no one way (helper functions, if-then-else, guards) is automatically better. These are available if you want to use them...

Using where:

\[
\text{remDup} :: \text{List Integer} \rightarrow \text{List Integer} \\
\text{remDup} \quad \text{Nil} \quad = \quad \text{Nil} \\
\text{remDup} \quad (\text{Cons} \ x \ \text{Nil}) \quad = \quad (\text{Cons} \ x \ \text{Nil}) \\
\text{remDup} \quad (\text{Cons} \ x \ \text{xs}) \quad = \quad \text{remDup'} \ x \ (\text{reDup} \ \text{xs}) \\
\quad \text{where} \quad \text{remDup'} :: \text{List Integer} \rightarrow \text{List Integer} \\
\quad \text{remDup'} \ x \ \text{xs} = \quad \text{if} \ x =\ &= \quad \text{head} \ \text{xs} \quad \text{then} \ \text{xs} \quad \text{else} \quad (\text{Cons} \ x \ \text{xs})
\]

Or you can do it with an if-then-else in the main function:

\[
\text{remDup} :: \text{List Integer} \rightarrow \text{List Integer} \\
\text{remDup} \quad \text{Nil} \quad = \quad \text{Nil} \\
\text{remDup} \quad (\text{Cons} \ x \ \text{Nil}) \quad = \quad (\text{Cons} \ x \ \text{Nil}) \\
\text{remDup} \quad (\text{Cons} \ x \ (\text{Cons} \ y \ \text{ys})) \quad = \quad \text{if} \ (x =\ &= \ y) \\
\quad \text{then} \ (\text{remDup} \ (\text{Cons} \ y \ \text{ys})) \\
\quad \text{else} \ (\text{Cons} \ x \ (\text{remDup} \ (\text{Cons} \ y \ \text{ys})))
\]
There are usually many different ways of defining a function, and no one way (helper functions, if-then-else, guards) is automatically better. These are available if you want to use them...

Or you can do it with an if-then-else in the main function:

```haskell
remDup :: List Integer -> List Integer
remDup  Nil                 = Nil
remDup (Cons x Nil)         = (Cons x Nil)
remDup (Cons x (Cons y ys)) = if (x == y)
                              then (remDup (Cons y ys))
                              else (Cons x (remDup (Cons y ys))
```

Or you can do it with a guard:

```haskell
remDup :: List Integer -> List Integer
remDup  Nil                 = Nil
remDup (Cons x Nil)         = (Cons x Nil)
remDup (Cons x (Cons y ys)) | x == y = (remDup (Cons y ys))
remDup (Cons x (Cons y ys)) = (Cons x (remDup (Cons y ys)))
```
Polymorphic Types (Extra Practice!)

Such a process determines what type a function must have:

\[ g :: (\text{Triple } \text{Bool } a \ b) \rightarrow (\text{Pair } (\text{Pair } \text{Nat } a) \ b) \]
\[ g \ (T \ \text{True} \ y \ z) = (P \ (P \ \text{Zero} \ y) \ z) \]

\[ h :: (\text{Pair } a \ (\text{Pair } b \ \text{Nat} )) \rightarrow (\text{Triple } a \ b \ \text{Bool}) \]
\[ h \ (P \ x \ (P \ y \ \text{Zero})) = (T \ x \ y \ \text{Bool}) \]

\[ \text{comp } x = (h \ (g \ x)) \]

\[ g :: (\text{Triple } \text{Bool } a \ b) \rightarrow (\text{Pair } (\text{Pair } \text{Nat } a) \ b) \quad x :: c \]

\[ (g \ x) :: (\text{Pair } (\text{Pair } \text{Nat } a) \ b) \]

\[ c = (\text{Triple } \text{Bool } a \ b) \]
**Polymorphic Types (Extra Practice!)**

Such a process determines what type a function must have:

\[ g :: (\text{Triple Bool} \ a \ b) \rightarrow (\text{Pair} \ (\text{Pair Nat} \ a) \ b) \]

\[ g \ (T \ True \ y \ z) = (P \ (P \ Zero \ y) \ z) \]

\[ h :: (\text{Pair} \ a \ (\text{Pair} \ b \ \text{Nat})) \rightarrow (\text{Triple} \ a \ b \ \text{Bool}) \]

\[ h \ (P \ x \ (P \ y \ \text{Zero})) = (T \ x \ y \ \text{Bool}) \]

\[ \text{comp} \ x = (h \ (g \ x)) \]

\[ g ::= (\text{Triple Bool} \ a \ b) \rightarrow (\text{Pair} \ (\text{Pair Nat} \ a) \ b) \quad x:::c \]

\[ (g \ x)::(\text{Pair} \ (\text{Pair Nat} \ a) \ b) \]

\[ h ::= (\text{Pair} \ a' \ (\text{Pair} \ b' \ \text{Nat})) \rightarrow (\text{Triple} \ a' \ b' \ \text{Bool}) \]

\[ (h \ (g \ x)) :: ?? \]

**Unify:**

\[ (\text{Pair} \ a' \ (\text{Pair} \ b' \ \text{Nat})) \quad a' = (\text{Pair} \ \text{Nat} \ a) \]

\[ (\text{Pair} \ (\text{Pair} \ \text{Nat} \ a) \ b) \quad b = (\text{Pair} \ b' \ \text{Nat}) \]
Polymorphic Types (Extra Practice!)

Such a process determines what type a function must have:

g :: (Triple Bool a b) -> (Pair (Pair Nat a) b)
g (T True y z) = (P (P Zero y) z)

h :: (Pair a (Pair b Nat)) -> (Triple a b Bool)
h (P x (P y Zero)) = (T x y Bool)

comp x = (h (g x))
Polymorphic Types (Extra Practice!)

Such a process determines what type a function must have:

\[
g :: (\text{Triple Bool } a \ b) \rightarrow (\text{Pair } (\text{Pair Nat } a) \ b)
\]
\[
g \ (T \ True \ y \ z) = (P \ (P \ Zero \ y) \ z)
\]

\[
h :: (\text{Pair } a \ (\text{Pair } b \ \text{Nat})) \rightarrow (\text{Triple } a \ b \ \text{Bool})
\]
\[
h \ (P \ x \ (P \ y \ \text{Zero})) = (T \ x \ y \ \text{Bool})
\]

\[
\text{comp } x = (h \ (g \ x))
\]

\[
g :: (\text{Triple Bool } a \ b) \rightarrow (\text{Pair } (\text{Pair Nat } a) \ b) \quad x :: c
\]
\[
(g \ x) :: (\text{Pair } (\text{Pair Nat } a) \ b)
\]
\[
h :: (\text{Pair } a' (\text{Pair } b' \ \text{Nat})) \rightarrow (\text{Triple } a' \ b' \ \text{Bool})
\]
\[
(h \ (g \ x)) :: (\text{Triple } (\text{Pair Nat } a) \ b' \ \text{Bool})
\]

\[
a' = (\text{Pair Nat } a)
\]
\[
b = (\text{Pair } b' \ \text{Nat})
\]
\[
c = (\text{Triple Bool } a \ b)
\]

\[
\text{comp} :: (\text{Triple Bool } a \ (\text{Pair } b' \ \text{Nat})) \rightarrow (\text{Triple } (\text{Pair Nat } a) \ b' \ \text{Bool})
\]