LAMBDA NOTATION

- Recall that a function in Haskell accepts one argument and returns one result.

peanuts	$\rightarrow$	chocolate-covered peanuts
raisins	$\rightarrow$	chocolate-covered raisins
ants	$\rightarrow$	chocolate-covered ants

  Using the lambda calculus, a general "chocolate-covering" function (or rather λ-expression) is described as follows:

  $$\lambda x.\text{chocolate-covered } x$$

  ... and then we can get chocolate-covered ants by applying this function:

  $$(\lambda x.\text{chocolate-covered } x) \text{ ants} \quad \rightarrow \quad \text{chocolate-covered ants}$$

Lambda Notation (cont'd)

- A general covering function:

$$\lambda y.\lambda x.y\text{-covered } x$$

The result of the application of such a function is itself a function:

$$(\lambda y.\lambda x.y\text{-covered } x) \text{ caramel} \quad \rightarrow \quad \lambda x.\text{caramel-covered } x$$

$$((\lambda y.\lambda x.y\text{-covered } x) \text{ caramel}) \text{ ants} \quad \rightarrow \quad (\lambda x.\text{caramel-covered } x) \text{ ants}$$
$$\rightarrow \quad \text{caramel-covered ants}$$

- Functions can also be parameters to other functions:

$$\lambda f.(f) \text{ ants}$$

$$((\lambda f.(f) \text{ ants}) \; \lambda x.\text{chocolate-covered}) \; x \quad \rightarrow \quad (\lambda x.\text{chocolate-covered } x) \text{ ants}$$
$$\rightarrow \quad \text{chocolate-covered ants}$$

- In a Haskell program, we write functions and then apply them.
  - Haskell programs are nothing more than collections of λ-expressions, with added sugar for convenience (and diabetes).
  - We write a Haskell program by writing λ-expressions and giving names to them.

```
succ x = x + 1                      succ = \ x -> x + 1

length = foldr oneplus 0            length = foldr (\ x -> \ n -> 1+n) 0
    where oneplus x n = 1+n         -- shorthand:  (\ x n -> 1+n)

Main> succ 10                       Main> (\ x -> x + 1) 10
11                                  11
```

  - Another example: `map (\ x -> x+1) [1,2,3]` maps (i.e., applies) the λ-expression $\lambda x.x + 1$ to all the elements of the list, thus producing `[2,3,4]`.
  - In general, for some expression E, $\lambda x.E$ (in Haskell-speak: `\ x -> E`) denotes the function that maps x to the (value of) E.

LAMBDA CALCULUS

- The lambda calculus is a formal system designed to investigate function definition, function application and recursion. It was introduced by Alonzo Church and Stephen Kleene in the 1930s.

- We start with a countable set of identifiers, e.g., $\{a, b, c, \ldots, x, y, z, x1, x2, \ldots\}$ and we build expressions using the following rules:

LEXPRESSION	$\rightarrow$	IDENTIFIER	
LEXPRESSION	$\rightarrow$	λIDENTIFIER.LEXPRESSION	(abstraction)
LEXPRESSION	$\rightarrow$	(LEXPRESSION)LEXPRESSION	(combination)
LEXPRESSION	$\rightarrow$	(LEXPRESSION)	

  – In an expression $\lambda x.E$, x is called a bound variable. A variable that is not bound is a free variable.

- Syntactical sugar: Normally, no literal constants exist in lambda calculus. We use, however, literals for clarity.

  – Further sugar: HASKELL!!

Reductions

- In lambda calculus, an expression $(\lambda x.E)F$ can be reduced to $E[F/x]$ ($E[x := F]$ in the textbook). $E[F/x]$ stands for the expression E, where F is substituted for all the bound occurrences of x.

- In fact, there are three reduction rules:

  α: $\lambda x.E$ reduces to $\lambda y.E[y/x]$ if y is not free in E (change of variable).

  β: $(\lambda x.E)F$ reduces to $E[F/x]$ (functional application).

  γ: $\lambda x.(Fx)$ reduces to F if x is not free in F (extensionality).

- The purpose in life of a Haskell program, given some expression, is to repeatedly apply these reduction rules in order to bring that expression to its “irreducible” form (formally, normal form).

Intermission: No Need for Constants

- Of course, one would not want to program without constants, but this is in fact possible:

$$\begin{array}{lcl} true & = & \lambda x.\lambda y.x \\ false & = & \lambda x.\lambda y.y \\ \text{if-then-else} & = & \lambda a.\lambda b.\lambda c.((a)b)c \end{array}$$

$$\begin{array}{ll} (((\text{if-then-else})false)caramel)chocolate & \\ \overset{\beta}{\Rightarrow} & (((\lambda a.\lambda b.\lambda c.((a)b)c)\lambda x.\lambda y.y)caramel)chocolate \\ \overset{\beta}{\Rightarrow} & ((\lambda b.\lambda c.((\lambda x.\lambda y.y)b)c)caramel)chocolate \\ \overset{\beta}{\Rightarrow} & (\lambda c.((\lambda x.\lambda y.y)caramel)c)chocolate \\ \overset{\beta}{\Rightarrow} & ((\lambda x.\lambda y.y)caramel)chocolate \\ \overset{\beta}{\Rightarrow} & (\lambda y.y)chocolate \\ \overset{\beta}{\Rightarrow} & chocolate \end{array}$$

WHICH ONE?

- You may have noticed that more than one order of reduction is possible in lambda calculus (and thus in Haskell):

```
square   :: Integer -> Integer
square x = x * x

smaller  :: (Integer, Integer) -> Integer
smaller (x,y) = if x<=y then x else y
```

$square\,(smaller\,(5,78))$ $\Rightarrow$ (def. $smaller$) $square\,5$ $\Rightarrow$ (def. $square$) 5×5 $\Rightarrow$ (def. $\times$) 25	$square\,(smaller\,(5,78))$ $\Rightarrow$ (def. $square$) $(smaller\,(5,78)) \times (smaller\,(5,78))$ $\Rightarrow$ (def. $smaller$) $5 \times (smaller\,(5,78))$ $\Rightarrow$ (def. $smaller$) 5×5 $\Rightarrow$ (def. $\times$) 25

Which one? (cont'd)

- Sometimes it even matters...

```
three   :: Integer -> Integer
three x = 3

infty :: Integer
infty = infty + 1
```

$three\ infty$ $\Rightarrow$ (def. $infty$) $three\,(infty + 1)$ $\Rightarrow$ (def. $infty$) $three\,((infty + 1) + 1)$ $\Rightarrow$ (def. $infty$) $three\,(((infty + 1) + 1) + 1)$ $\vdots$	$three\ infty$ $\Rightarrow$ (def. $three$) 3

LAZY HASKELL

- Haskell uses the second variant, called lazy evaluation (normal order, outermost reduction), as opposed to eager evaluation (applicative order, innermost reduction):

```
Main> three infty
3
```

- Why is good to be lazy:
  - Doesn't hurt: If an irreducible form can be obtained by both kinds of reduction, then the results are guaranteed to be the same.
  - More robust: If an irreducible form can be obtained, then lazy evaluation is guaranteed to obtain it.
  - Even useful: It is sometimes useful (and, given the lazy evaluation, possible) to work with infinite objects.

- `[1 .. 100]` produces the list of numbers between 1 and 100.
  - What is produced by `[1 .. ]`?

```
Prelude> [1 ..] !! 10
11
Prelude> [1 .. ] !! 12345
12346
Prelude> zip ['a' .. 'g'] [1 ..]
[('a',1),('b',2),('c',3),('d',4),('e',5),('f',6),('g',7)]
```

  - A useful example: the good ol' prime numbers, this time as a stream:

```
primes :: [Integer]
primes = sieve [2 .. ]
  where sieve (x:xs) = x : [n | n <- sieve xs, mod n x /= 0]
     -- alternate:
     -- sieve (x:xs) = x : sieve (filter (\ n -> mod n x /= 0) xs)

Main> take 20 primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
```

- One can implement functions that operate on streams of objects instead of objects.
  - An example of generating a stream (and filtering it) is the function `sieve`.
- A function that operates on streams of characters:

```
splitLines :: String -> [String]
splitLines str
    | line == "bye" = []
    | True          = line : splitLines rest
    where line =  takeWhile (/= '\n') str
          rest = tail (dropWhile (/= '\n') str)

joinLines :: [String] -> String
joinLines = concat.(map (++"\n"))

answer :: String -> String
answer str
    | take 2 str == "hi" = ">Pleased to meet you"
    | True               = ">You typed " ++ show (length str) ++
                           " characters. Good work!"

count :: String -> String
count = joinLines.(map answer).splitLines
```

Streams and I/O (cont'd)

```
Main> count "hi there\nhow are you\nbye"
">Pleased to meet you\n>You typed 11 characters. Good work!\n"
```

STREAMS AND I/O (CONT'D)

```
Main> count "hi there\nhow are you\nbye"
">Pleased to meet you\n>You typed 11 characters. Good work!\n"

Main> putStr (count "hi there\nhow are you\nbye")
>Pleased to meet you
>You typed 11 characters. Good work!
```

Streams and I/O (cont'd)

```
Main> count "hi there\nhow are you\nbye"
">Pleased to meet you\n>You typed 11 characters. Good work!\n"

Main> putStr (count "hi there\nhow are you\nbye")
>Pleased to meet you
>You typed 11 characters. Good work!

Main> interact count
hi there
>Pleased to meet you
how are you
>You typed 11 characters. Good work!
bye
```

- Input (from keyboard) and output (to terminal) are themselves streams.

```
interact :: (String -> String) -> IO ()
```

Memo functions

- Streams can also be used to improve efficiency (dramatically!)
- Take the Fibonacci numbers:

```
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

  – Complexity? $O(2^n)$

- Now take them again, using a memo stream:

```
fastfib :: Integer -> Integer
fastfib n = fibList %% n
    where fibList = 1 : 1 : zipWith (+) fibList (tail fibList)
          (x:xs) %% 0 = x
          (x:xs) %% n = xs %% (n - 1)
```

  – Complexity? $O(n)$

STRUCTURAL INDUCTION

- The induction principle is not limited to functions defined over the integers. Rather, mathematical induction over the natural numbers is an instance of the more general notion of structural induction over values of an inductively (or recursively) defined type:

  To prove property $\mathcal{P}$ on some inductively defined (algebraic) type T, we prove that

  Base case. $\mathcal{P}$ holds for the base cases of the definition of T, and

  Inductive case. $\mathcal{P}$ holds for the inductive cases of the definition of T, assuming that $\mathcal{P}$ holds for the components of type T of the inductive definitions.

- Example of inductively defined type:

```
data Nat = Zero | Succ Nat
```

  `Zero` is the base case, and `Succ Nat` is the inductive case. Thus, to prove that $\mathcal{P}$(n) holds for any `n::Nat`, we show that **(a)** $\mathcal{P}$(`Zero`) holds (base), and **(b)** if $\mathcal{P}$(n) holds, then so does $\mathcal{P}$(`Succ n`) (inductive step).

Examples of Structural Induction

- Recall that we defined arithmetic operators over `Nat`:

```
instance Num Nat where
    m + Zero = m                     -- (1)
    m + (Succ n) = Succ (m + n)      -- (2)
    m * Zero = Zero
    m * (Succ n) = (m * n) + m
```

- Let us prove that `Zero + n = n` for all `n::Nat`.

  Base: For n=`Zero` we have `Zero + Zero = Zero` by case (1).

  Inductive step: We assume that `Zero + n = n` (inductive assumption) and we prove that `Zero + Succ n = Succ n`:

```
Zero + Succ n  =  Succ (Zero + n)
                  (by (2))
               =  Succ n
                  (Zero + n = n by inductive assumption).
```

- A tree is another inductively defined type:

```
data BTree a = Null | Node a (BTree a) (BTree a)
```

- To prove that a property $\mathcal{P}$(`t`) holds for any `t::BTree a`, we show that **(a)** $\mathcal{P}$(`Null`) holds (base), and **(b)** if, for any `x::a`, both $\mathcal{P}$(`lt`) and $\mathcal{P}$(`rt`) hold, then so does $\mathcal{P}$(`Node x lt rt`) (inductive step).

- Example: the height and size of a tree are computed by the following functions:

```
height :: BTree a -> Int
height Null = 0
height (Node x lt rt) = 1 +
      max (height lt) (height rt)
```

```
size :: BTree a -> Int
size Null = 0
size (Node x lt rt) = 1 +
      size lt + size rt
```

  - We want to show that $\texttt{size t} \leq 2^{\texttt{height t}}$ for all `t::BTree a`.

- Direct proof of

$$\texttt{size t} \leq 2^{\texttt{height t}} \tag{1}$$

left as exercise.

- I will instead prove the stronger relation:

$$\texttt{size t} \leq 2^{\texttt{height t}} - 1. \tag{2}$$

- Relation (1) follows immediately from Relation (2).

Base: $\texttt{size Null} = 0 = 1 - 1 = 2^0 - 1 = 2^{\texttt{height Null}} - 1$.

Inductive step:

$$
\begin{aligned}
\texttt{size (Node x lt rt)} &= \texttt{1 + size lt + size rt} \\
&\quad \text{(by def. } \texttt{size}\text{)} \\
&\leq \texttt{1 + } 2^{\texttt{height lt}} - 1 \texttt{ + } 2^{\texttt{height rt}} - 1 \\
&\quad \text{(by inductive assumption for both } \texttt{lt} \text{ and } \texttt{rt}\text{)} \\
&\leq 2^{\texttt{height lt}} \texttt{ + } 2^{\texttt{height rt}} - 1 \\
&\quad \text{(arithmetic)} \\
&\leq 2^h + 2^h - 1 \\
&\quad \text{(with } h = \texttt{max (height lt) (height rt)}\text{)} \\
&= 2^{1+h} - 1 \\
&\quad \text{(arithmetic)} \\
&= 2^{\texttt{height (Node x lt rt)}} - 1 \\
&\quad \text{(by def. of } h \text{ and } \texttt{height}\text{, as desired)}
\end{aligned}
$$