Defining Functions

Yan Huang
yh33@indiana.edu
The Very Basic Syntax

double :: Int → Int

double x = x + x
Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions.

\[
\text{abs} :: \text{Int} \to \text{Int}
\]

\[
\text{abs } x = \begin{cases} 
    x & \text{if } x \geq 0 \\
    -x & \text{else}
\end{cases}
\]

abs takes an integer \( n \) and returns \( n \) if it is non-negative and \(-n\) otherwise.
Conditional Expressions

As in most programming languages, functions can be defined using **conditional expressions**.

\[
\text{abs} :: \text{Int} \rightarrow \text{Int} \\
\text{abs } n = \text{if } n \geq 0 \text{ then } n \text{ else } -n
\]

abs takes an integer \( n \) and returns \( n \) if it is non-negative and \(-n\) otherwise.
Conditional expressions can be nested:

Define the function signum, which returns -1 when given a negative integer; returns 1 when given a positive integer; and 0 if given 0.

\[
\text{signum} :: \text{Int} \rightarrow \text{Int}
\]

\[
\text{signum } n = \begin{cases} 
\text{if } n > 0 \text{ then } 1 \\
\text{else if } n < 0 \text{ then } -1 \\
\text{else } 0
\end{cases}
\]
Conditional expressions can be nested:

```haskell
signum :: Int -> Int
signum n = if n < 0 then -1 else
           if n == 0 then 0 else 1
```

• In Haskell, conditional expressions must always have an else branch, which avoids any possible ambiguity problems with nested conditionals.
Guarded Equations

As an alternative to conditionals, functions can also be defined using *guarded equations*.

\[
\begin{align*}
\text{abs} & : \mathbb{N}^+ \rightarrow \mathbb{N}^+ \\
\text{abs} \quad n & \mid n \geq 0 = n \\
\mid n < 0 & = -n \\
\mid \text{otherwise} & = -n
\end{align*}
\]
Guarded Equations

As an alternative to conditionals, functions can also be defined using *guarded equations*.

\[
\text{abs } n \mid n \geq 0 \quad = n \\
\mid \text{otherwise} \quad = -n
\]

As previously, but using guarded equations. The catch all condition `otherwise` is defined in the “Prelude” by otherwise = True.
Guarded equations can be used to make definitions involving multiple conditions easier to read. E.g., try define signum using guarded equations.

\[
\text{Signum } n \begin{cases} 
  n > 0 & = 1 \\
  n = 0 & = 0 \\
  \text{otherwise} & = -1
\end{cases}
\]
Guarded equations can be used to make definitions involving multiple conditions easier to read:

\[
\text{signum } n \mid n < 0 \quad = -1 \\
\mid n == 0 \quad = 0 \\
\mid \text{otherwise} \quad = 1
\]
Pattern Matching

Many functions have a particularly clear definition using *pattern matching* on their arguments.

\[
\text{not} :: \text{Bool} \rightarrow \text{Bool} \\
\text{not False} = \text{True} \\
\text{not True} = \text{False}
\]

not maps False to True, and True to False.
Functions can often be defined in many different ways using pattern matching. For example

```plaintext
(&&) :: Bool → Bool → Bool

True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

can be defined more compactly by

```plaintext
True && True = True
_    && _    = False
```

The underscore symbol `_` is a wildcard pattern that matches any argument value.
However, the following definition is more efficient, because it avoids evaluating the second argument if the first argument is False:

```
True  && b = b
False && _ = False
```
Patterns are matched in order. For example, the following definition always returns False:

```
_ && _ = False
True && True = True
```

You may not repeat variables in the same pattern. For example, the following definition gives an error:

```
b && b = b
_ && _ = False
```
List Patterns

Internally, every non-empty list is constructed by repeated use of an operator (:) called "cons" that adds an element to the start of a list.

\[ [1,2,3,4] \]

Means \( 1:(2:(3:(4:[]))). \)
Functions on lists can be defined using \texttt{x:xs} pattern.

\begin{align*}
\text{head} & : [a] \rightarrow a \\
\text{head} (x:_\hspace{1em}) & = x \\
\text{tail} & : [a] \rightarrow [a] \\
\text{tail} (_:xs) & = xs
\end{align*}

head and tail map any non-empty list to its first and remaining elements.
z
x:xs	patterns	must	be	parenthesised,	because
application	has
priority	over	(:).		For
texample,
the
following
definition
gives	an	error:

```haskell
> head []
ERROR
head x:_ = x
```

Employers are looking for specific business and technology skills. According to sources like PayScale and Monster.com, these 13 are among the most likely to make you stand out.

Most valuable **COMPUTER SCIENCE** skills

**SKILL #1**

**Scala**

- 22% average pay boost
- Apply functional programming paradigms to write elegant Scala code.

**Functional Programming in Scala**

[Join Now]
Scala

• Full support for functional programming with a very strong static type system, heavily influenced by Haskell
  - Currying
  - Type inference
  - Immutability
  - Lazy evaluation
  - Pattern matching
  - Algebraic data types

• Compiles to JVM bytecode
Lambda Expressions

Functions can be constructed without naming the functions by using **lambda expressions**.

\[ \lambda x \to x + x \]

The nameless function that takes a number \( x \) and returns the result \( x + x \).
The symbol \( \lambda \) is the Greek letter \texttt{lambda}, and is typed at the keyboard as a backslash “\".

In mathematics, nameless functions are usually denoted using the \( \mapsto \) symbol, as in \( x \mapsto x + x \).

In Haskell, the use of the \( \lambda \) symbol for nameless functions comes from the \texttt{lambda calculus}, the theory of functions on which Haskell is based.
Why Are Lambda's Useful?

Lambda expressions can be used to give a formal meaning to functions defined using currying.

For example:

\[
\text{add } x \ y = x + y
\]

means

\[
\text{add} = \lambda x \rightarrow (\lambda y \rightarrow x + y)
\]
Lambda expressions are also useful when defining functions that return functions as results.

For example:

\[
\text{const} :: a \to b \to a
\]
\[
\text{const } x \_ = x
\]

is more naturally defined by

\[
\text{const} :: a \to (b \to a)
\]
\[
\text{const } x = \_ \to x
\]
Lambda expressions can be used to avoid naming functions that are only referenced once. For example:

\[
\text{odds } n = \text{map } f \left[ 0..n-1 \right]
\]

where

\[
f x = x^2 + 1
\]

can be simplified to

\[
\text{odds } n = \text{map } \left( \lambda x \rightarrow x^2 + 1 \right) \left[ 0..n-1 \right]
\]
An operator written between its two arguments can be converted into a curried function written before its two arguments by using parentheses.

For example:

```plaintext
> 1+2
3
> (+) 1 2
3
```
This convention also allows one of the arguments of the operator to be included in the parentheses.

For example:

\[
\begin{array}{l}
> (1+) 2 \\
\geq 6 \\
> (+2) 1 \\
\geq 6
\end{array}
\]

In general, if \(\oplus\) is an operator then functions of the form \((\oplus)\), \((x\oplus)\) and \((\oplus y)\) are called sections.
Why Are Sections Useful?

Useful functions can sometimes be constructed in a simple way using sections. For example:

- \((1+)\) - successor function
- \((1/)\) - reciprocation function
- \((*2)\) - doubling function
- \((/2)\) - halving function
Exercises

(1) Consider a function `safetail` that behaves in the same way as `tail`, except that `safetail` maps the empty list to the empty list, whereas `tail` gives an error in this case. Define `safetail` using:

(a) a conditional expression;
(b) guarded equations;
(c) pattern matching.

Hint: the library function `null :: [a] → Bool` can be used to test if a list is empty.
(2) Give three possible definitions for the logical or operator (||) using pattern matching.

(3) Redefine the following version of (&&) using conditionals rather than patterns:

\[
\begin{align*}
\text{True} \land \text{True} &= \text{True} \\
_ \land _ &= \text{False}
\end{align*}
\]

(4) Do the same for the following version:

\[
\begin{align*}
\text{True} \land b &= b \\
\text{False} \land _ &= \text{False}
\end{align*}
\]