Recursion and Induction: Haskell; Primitive Data Types; Writing Function Definitions

Greg Plaxton
Theory in Programming Practice, Spring 2004
Department of Computer Science
University of Texas at Austin
Functional programming languages are particularly well-suited for recursive programming.

Haskell is a modern functional programming language:
- In this module, we discuss recursive programming in the context of Haskell.
- In the first few lectures we introduce some basic elements of the Haskell programming language.
Primitive Data Types

• We make use of the following built-in Haskell data types:
  – integer (called \texttt{Int})
  – boolean (called \texttt{Bool})
  – character (called \texttt{Char})
  – string (called \texttt{String})

• In the next few slides we discuss each of these types in turn
• Arbitrary precision

• Usual infix arithmetic operators: addition (+), subtraction (−), multiplication (∗), exponentiation (∗∗)

• Other standard binary operators such as maximum (max), minimum (min), modulus (mod), integer division (div), and integer remainder (rem) can be written in one of two ways:
  – The function name can precede the arguments, e.g., the expression “mod 17 4” evaluates to 1
  – The function name can be surrounded by back quotes and used in an infix manner, as in “17 ‘mod‘ 4”

• The arithmetic relations are < <= == /= > >=
  – Note that == is used for equality and /= is used for inequality
  – The result of an expression such as “3 < 5” is the boolean True
The two boolean constants are written True and False

The boolean operators are
  – negation (not)
  – and (&&)
  – or (||)
  – equivalence (==)
  – inequivalence, also known as “exclusive or” (/=)
Here is a short session with hugs, the Haskell interpreter:

Prelude> (3 > 5) || (5 > 3)
True
Prelude> (3 > 3) || (3 > 3)
False
Prelude> (2 `mod` (-3)) == ((-2) `mod` 3)
False
Prelude> even 3 || odd 3
True
Char

- A character is enclosed within single quotes
- Two functions are used to convert between characters and integers
  - Function `ord(c)` returns the value of the character `c` in the internal coding table; it is a number between 0 and 255
  - Function `chr` converts a number between 0 and 255 to the corresponding character
  - `chr(ord(c)) = c`, for all characters `c`
Char

- Here is a hugs session involving `ord` and `chr`:

```
Prelude> ord('a')
97
Prelude> chr(97)
'a'
Prelude> ord(chr(103))
103
Prelude> chr(255)
'\255'
Prelude> (ord '9')- (ord '0')
9
Prelude> (ord 'a')- (ord 'A')
32
```
You can compare two characters using arithmetic relations; the outcome is determined by their integer ord values

Prelude> 'a' < 'b'
True
Prelude> 'A' < 'a'
True

It is illegal to compare a character directly with an integer

Prelude> 'a' < 3
ERROR - Illegal Haskell 98 class constraint in inferred type
*** Expression : 'a' < 3
*** Type : Num Char => Bool
String

- String constants are enclosed in double quotes
  
  Prelude> "a b c"
  "a b c"
  Prelude> "a, b, c"
  "a, b, c"

- A string is implemented as a *list* of characters
  - Later on in the module we discuss lists in detail
  - All the rules governing lists apply to strings
Writing Function Definitions

• Functions cannot be defined at the hugs prompt

• Instead, function definitions are typed into a separate file using a text editor

• The resulting file of function definitions is then loaded into the hugs interpreter
Loading Program Files

```
Prelude> :l 337.hs
Reading file "337.hs":

Hugs session for:
/lusr/share/hugs/lib/Prelude.hs
337.hs
```
Comments

• Any string following -- is considered a comment

• Alternatively, we can begin and end a comment using {– and –}
  – The latter format is particularly useful for multiline comments
Examples of Function Definitions

• Here are three simple function definitions

\[
\begin{align*}
\text{inc } x &= x + 1 \\
\text{imply } p \quad q &= \text{not } p \quad \text{||} \quad q \\
\text{digit } c &= (\text{'0'} \leq c) \quad \text{&&} \quad (c \leq \text{'9'})
\end{align*}
\]

• A function need not have any arguments

\[
\text{offset } = (\text{ord } \text{'a'}) - (\text{ord } \text{'A'})
\]
Using Functions

Main> inc 5
6
Main> imply True False
False
Main> digit '6'
True
Main> digit 'a'
False
Main> digit(chr(inc(ord '8')))
True
Main> digit(chr(inc(ord '9')))
False
Conditional Equations

• It is often convenient to define a function by cases using a conditional equation

• Such a definition consists of a nonempty sequence of *clauses* of the form “\( G = E \)” where \( G \) is a boolean predicate called the *guard* and \( E \) is an expression
  - For a given set of arguments, the function evaluates to the expression associated with the first clause for which the guard evaluates to True
  - If no guard evaluates to True, a run-time error occurs

• Here is a simple example of a function that computes the absolute value of its argument

  \[
  \text{absolute } x \\
  \quad \mid x \geq 0 = x \\
  \quad \mid x < 0 = -x
  \]
Conditional Equations: Otherwise

• The last clause of a conditional equation can have the special guard otherwise

• If such an “otherwise clause” is present, the associated expression determines the value of the function whenever the guards of all other clauses evaluate to False

• Here is an alternative definition of the absolute value function

$$\text{absolute } x$$
$$| x \geq 0 = x$$
$$| \text{otherwise} = -x$$