Code Quality
Week 5 Friday

COMP1100/1130

Style

Until now, you've just been writing code that works.

Now, you should also focus on writing code with good *style*.

Code style is important. Using good style helps others to understand your code.

There will be marks for style in Assignment 2 and the final exam.

- More importantly, it's essential for when you start programming in your career.

Style

Why is it important to make your code easy to read?

Who's going to read your code?

- colleagues and your team members for projects
- code reviewers
- future programmers who may be using/extending your code
- you in the future!

Style

Do you need to explain everything?

- no need to assume that you have to teach the reader everything

- you can assume that the readers are competent programmers

- assume that they know how to look up library documentation, etc.

Comments

Comments are brief descriptions that you put into the code, to help people to understand what it does.

The Haskell syntax for comments is:
- - dashes at the start of each line

Another way of inserting comments:
{- some text -}

The idea is to explain what the code does, but not how it does it.

Comments

Too many comments are not good. Too little is not good either!

Just explain the functionality clearly.

```
-- Finds the length of a given list.
myLength :: [a] -> Int
myLength list = case list of
  []    -> 0
  _:xs -> 1 + myLength xs
```

Descriptions in Other Parts of the Code

Comments are not the only things that help readers to understand the code.

Remember how helpful type declarations are:

```
myLength :: [a] -> Int
```

- helps to explain what the function does
- also explains how the function can be used with other functions

Names

The names of variables and functions make a big difference.

Any name will *work*, but you need to choose names that help the reader.

Don't use really short names.

Use the standard naming conventions, like *camelCase*.

e.g. `myLength`

Coding Style

There are certain ways of programming that better than others:

- Use case expressions appropriately:

Don't do this:

```haskell
myAbs :: Int -> Int
myAbs x =
   case x >= 0 of
    True   ->  x
    False  -> -x
```

Do this:

```haskell
myAbs :: Int -> Int
myAbs x
   | x >= 0 = x
   | otherwise = -x
```

Coding Style

There are certain ways of programming that better than others:

- Use case expressions appropriately:

Don't do this:

```
isPositive :: Int -> Bool
isPositive x
  | x > 0       = True
  | otherwise = False
```

Do this:

```
isPositive :: Int -> Bool
isPositive x = x > 0
```

or do this:

```
isPositive :: Int -> Bool
isPositive = (> 0)
```

Coding Style

Don’t leave any unresolved warnings.

- e.g. use _ for unused input

Don’t do this:

```
addFour :: (Int, Int) -> Int
addFour (x,y) = x + 4
```

Do this:

```
addFour :: (Int, Int) -> Int
addFour (x,_) = x + 4
```

Coding Style

- Don’t leave any dead code – code the main program never uses.
- Minimise repeated code – e.g. use a separate function.
- Create structure for large definitions, e.g. by using helper functions.
- Avoid unnecessary code, e.g. using the Prelude functions.

Coding Style

- Always use consistent indentation – use spaces not tabs.
- Avoid extra long lines – no more than 80 characters wide.

- Break up long lines:

```
getAllEmployees (Node leftChild (Employee name _) rightChild)
    = (getAllEmployees rightChild)
      ++ (getAllEmployees leftChild) ++ [name]
```

Coding Style

- For more tips, see the Style Guide on our website (under Resources).

- Note that this is a beginner course, so we don't always show you the way a professional would write the code.

- You still need to write the code in the most appropriate way.

Testing and Verification

How do you know whether your code is correct?

- ❑ Looking at your code.
- ❑ Testing your code.
- ❑ Formal verification (mathematical proofs).

Testing can only show the presence of bugs, not the absence (Dijskstra).

Black Box Testing vs. White Box Testing

Black box testing

- Based on the specification.
- Doesn't use the code (can even be designed before you write the code).

White box testing

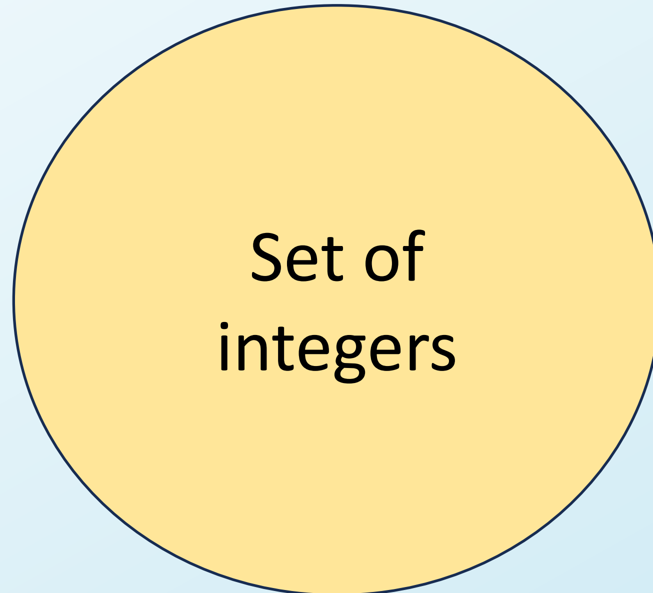
- Based on the code.
- Find ways to try and break the code, e.g. border cases.

Black Box Testing

Black box testing

Black Box Testing

What should we test?



Zero

Positive value

Negative value

Max/min values of Int

Black Box Testing

Identify groups

- No need to check several items in the same group.
- Inputs in the same group should behave in a similar way.
- Inputs in different groups should behave differently.
- Groups should collectively cover all possibilities.
- Pay attention to special cases, e.g. boundaries, zero.

Black Box Testing

- Strings: test a string with one char, many chars, empty string.
- Lists: test a list with one element, many elements, empty list.
- {1..10}: test 1, test 10, test a number in the middle.

Black Box Testing

maxThree :: Int -> Int -> Int -> Int

- The group where the first number is the greatest.
- The group where the second number is the greatest.
- The group where the third number is the greatest.
- Boundary cases: some inputs are equal.

White Box Testing

White box testing

- Based on the code.
- Identify points where the code makes a choice, e.g. cases, guards, base case vs. step case in recursions.
- Watch out for otherwise and _
- Focus on inputs at boundaries, overlapping situations

White Box Testing

```
maxThree :: Int -> Int -> Int -> Int
maxThree x y z
   | x > y && x > z = x
   | y > x && y > z = y
   | otherwise      = z
```

- ❑ Based on the boundary of the 1st two cases, test e.g. 2 2 1