Introduction to Functional Programming

Pattern matching, guards, and recursion

Some slides are based on Graham Hutton’s public slides

GÖTEBORGS UNIVERSITET
CHALMERS

Recap previous lecture

- Introduction to:
  - Course
  - Programming languages and computers
  - Tools (terminal, editor, …)
  - Writing code
- Functions
  - Learn how to define simple functions
  - Application on arguments
  - Composing functions
  - Variables

Today

- Groups!

- Building an executable
- Pattern matching
- Guarded equations (cases)
- Recursion
- Testing

Guarded equations

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

- Guarded equations can be used to make definitions involving multiple conditions easier to read

- The catch all condition `otherwise` is defined in the `Prelude` by:

```
otherwise = True
```

As previously, but using guarded equations

```
abs n | n >= 0     = n
      | otherwise = -n


signum n | n < 0       = -1
         | n == 0      = 0
         | otherwise = 1
```

Pattern matching

- Many functions have a particularly clear definition using *pattern matching* on their arguments
- A variable matches *everything*
- Patterns are matched *in order*, that is top-down
- Patterns may not repeat variables. For example, the following definition gives an error:

```
equal x x = True
```

`not` maps `False` to `True`, and `True` to `False`

```
not False = True
not True = False
```

LIVE CODING!

RECURSION

Recursion

- As we have seen, many functions can naturally be defined in terms of other functions.
- Expressions are evaluated by a stepwise process of applying functions to their arguments.

```
fac n = product [1..n]
```

`fac` maps any integer `n` to the product of the integers between 1 and `n`.

```
fac 4
  =>
product [1..4]
  =>
product [1,2,3,4]
  =>
1*2*3*4
  =>
24
```

Recursive functions

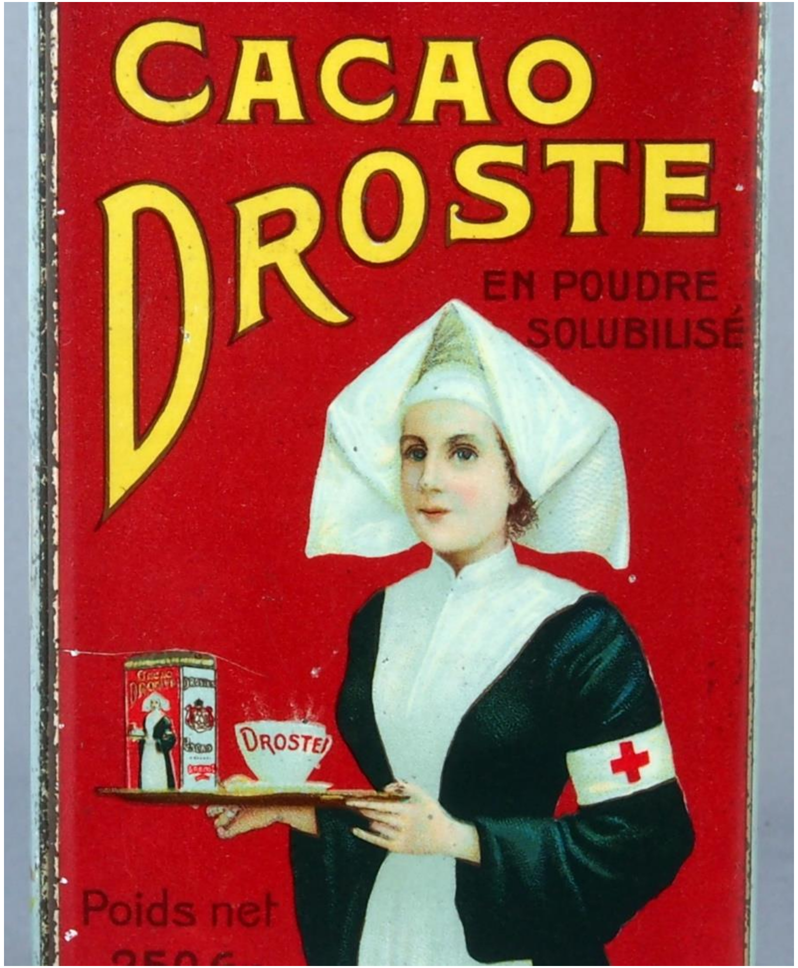
- In Haskell, functions can also be defined in terms of themselves. Such functions are called *recursive*.
- `fac 0 = 1` is appropriate because 1 is the identity for multiplication: `1*x = x = x*1`.
- The recursive definition *diverges* on integers < 0 because the base case is never reached:

```
ghci> fac (-1)
*** Exception: stack overflow
```

```
fac 0 = 1
fac n = n * fac (n-1)


fac 2
  =>
2 * fac 1
  =>
2 * (1 * fac 0)
  =>
2 * (1 * 1)
  =>
2 * 1
  =>
2
```

`fac` maps 0 to 1, and any other integer to the product of itself and the factorial of its predecessor.



Why is recursion useful?

- Some functions, such as factorial, are *simpler* to define in terms of other functions.
- As we shall see, however, many functions can *naturally* be defined in terms of themselves.
- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of *induction*.

GÖTEBORGS UNIVERSITET

CHALMERS