

Defining Functions

- So far we have only seen simple function definitions like:

```
cube :: Int -> Int
cube x = x*x*x
```

- When Hugs finds an expression matching the left-hand side of a definition, the evaluation mechanism simply replaces it by the right-hand side

- For example:

```
cube 2 + 1
⇒ (2*2*2) + 1
⇒ 8 + 1
⇒ 9
```

- Haskell provides more than just simple expressions as definitions, e.g.
 - *local definitions* (i.e. *where* clauses)
 - *guards* and alternative *right-hand* sides
 - *patterns* and alternative *left-hand* sides

Local Definitions

- In Haskell we can have function definitions that apply only inside the definition of another function, and that is why they are called *local definitions*

- For example:

```
stdDev :: [Float] -> Float
stdDev observations =
    sqrt (sum (map (^2) deviations))
  where
    deviations =
      map (subtract mean) observations
    mean =
      (sum observations) / fromIntegral count
    count =
      length observations
```

- It is important to remember that the definitions `deviations`, `mean` and `count` only apply inside the definition of `stdDev`

Definitions With Guards

- Guards make it possible to give alternatives in the definitions of functions
- Guards express conditions the function output depends on
- The conditions can be either `True` or `False`, and hence a guard is a Boolean expression

Definitions With Guards

- For example, here is the function to find the minimum of two integers (`min` is already defined so let us give it a different name):

```
myMin :: Int -> Int -> Int
```

```
myMin x y
```

```
  | x <= y = x
```

```
  | x > y  = y
```

- There are two alternative right-hand sides (x and y), and a guard for each of them ($x \leq y$ and $x > y$ respectively)
- Hugs evaluates each guard in turn, *first to last*, until it finds one that equals `True`
- The left-hand side is then replaced by the right-hand side that corresponds to that `True` guard, and this is returned as the result of function application
- For example:

```
myMin 5 0
```

(first guard) $5 \leq 0 \Rightarrow \text{False}$

(second guard) $5 > 0 \Rightarrow \text{True}$

$\Rightarrow 0$

Definitions With Guards

- If *none* of the guards are true, Hugs will report an error
- Therefore it is wise to make sure that one guard will always be true
- Haskell provides a special guard `otherwise` that is always true
- `otherwise` is used as a catch-all guard at the *end* of a sequence of alternatives:

```
myMin :: Int -> Int -> Int
myMin x y
  | x <= y = x
  | otherwise = y
```

Definitions With Guards

- Here is another example (see Lab 2): `isSum` that takes three integer arguments and tests whether one of them is the sum of the other two

- Definition without guards

```
isSum :: Int -> Int -> Int -> Bool
isSum x y z
    = (x + y == z) || (x + z == y) || (y + z == x)
```

- Definition with guards

```
isSum :: Int -> Int -> Int -> Bool
isSum x y z
    | x + y == z = True
    | x + z == y = True
    | y + z == x = True
    | otherwise = False
```

Defining Functions

- Of course, a function definition can use both guards and local definitions, as in the following script:

```
-- Find the roots of a quadratic  $ax^2 + bx + c$ 
-- Check that it has real roots before proceeding
-- If the quadratic has only one real root,
-- the result pair contains that root twice

roots :: Float -> Float -> Float -> (Float, Float)
roots a b c
  | discriminant > 0 = ((-b + (sqrt discriminant))/(2*a),
                       (-b - (sqrt discriminant))/(2*a))
  | otherwise = error "No real roots"
where
  discriminant = b^2 - 4*a*c
```

Function Layout – Recap

- You should adopt the following layout of function definitions as standard:

```
fun p_1 p_2 ... p_n
  | guard_1 = e_1
  | guard_2 = e_2
  ...
  | guard_k = e_k
  where
    local_1 a_1 ... a_m = r_1
    local_2 = r_2
    ...
```

- You do not need to follow this exactly but the main thing is to adopt a *uniform* style

Definitions With Patterns

- Function definitions can consist of a number of separate clauses with patterns on the left-hand sides of the equations
- For example, think of a function that returns either its second or third argument, depending on whether the first argument is `True` or `False`:

```
pick :: Bool -> a -> a -> a
pick True x y = x
pick False x y = y
```

- It is always possible to use guards but patterns can make definitions more readable
- The simplest patterns are just constant values
- For example, in `fact 0 = 1` the pattern is the constant value `0`

Definitions With Patterns

- A useful class of patterns are the $n + k$ patterns:
 $n+1, n+2, n+3, \dots$
- $n + k$ matches any integer greater than or equal to k
- Most often, the pattern is just $n+1$, as in the following script:

```
fact :: Int -> Int
fact 0 = 1
fact (n+1) = (n+1)*(fact n)
```

- An equivalent definition using guards is:

```
fact' :: Int -> Int
fact' n
  | n == 0 = 1
  | otherwise = n*(fact (n-1))
```

Definitions With Patterns

- Definitions using $n + k$ patterns are just like *inductive* definitions:
 - some clause(s) define the *basic/basis* case(s)
 - some clause(s) define the *inductive* cases
- For `fact` the first clause is the *basis* case:
`fact 0 = 1`
- The second clause is the *inductive* case, this is `fact (n+1)` is defined in terms of `fact n`, eventually getting down to the basis case:
`fact (n+1) = (n+1)*fact(n)`

Recursive Functions

- A function like the factorial function (either with guards or patterns) is called *recursive* because `fact` is used to describe `fact` itself
- Put this way it may sound strange: how can we describe something in terms of itself?
- But the definition is perfectly sensible since it gives:
 - a starting point: the value of `fact` at 0, and
 - a way of going from the value of `fact` at a particular point, `fact (n)`, to the value of `fact` at the next point, namely `fact (n+1)`

Recursive Functions

- Let us go back to the example of the factorial function:

```
fact :: Int -> Int
```

```
fact 0 = 1
```

```
fact (n+1) = (n+1)*fact(n)
```

- The recursive reasoning goes as follows for expression `fact 4`:

```
fact 4
```

```
⇒ 4 * fact 3
```

```
⇒ 4 * (3 * fact 2)
```

```
⇒ 4 * (3 * (2 * fact 1))
```

```
⇒ 4 * (3 * (2 * (1 * fact 0)))
```

```
⇒ 4 * (3 * (2 * (1 * 1)))
```

```
⇒ 4 * (3 * (2 * 1))
```

```
⇒ 4 * (3 * 2)
```

```
⇒ 4 * 6
```

```
⇒ 24
```

Recursive Functions

- `fact 4`
 - $\Rightarrow 4 * \text{fact } 3$
 - $\Rightarrow 4 * (3 * \text{fact } 2)$
 - $\Rightarrow 4 * (3 * (2 * \text{fact } 1))$
 - $\Rightarrow 4 * (3 * (2 * (1 * \text{fact } 0)))$
 - $\Rightarrow 4 * (3 * (2 * (1 * 1)))$
 - $\Rightarrow 4 * (3 * (2 * 1))$
 - $\Rightarrow 4 * (3 * 2)$
 - $\Rightarrow 4 * 6$
 - $\Rightarrow 24$
- We start with the goal to be evaluated (`fact 4`) and simplify the equation until the basis case (`fact 0`) is reached before calculating and returning the result (24)
- We work from the goal back down to the basis case using the recursion
- Recursion takes us from a more complicated case to a simpler one, and we have given a value for the simplest case, which is eventually reached

Recursive Functions

- Recursion makes it possible to implement iteration in a functional language; it is actually the only way to implement iteration
- Iterative constructs like `while` and `for` loops no longer make sense since we cannot change the variables that might appear in the conditions that control the iteration
- Here is another way of implementing a function that calculates factorials (using conditional expressions `if` and `then`):
`fact n = if n == 0 then 1 else n*fact(n-1)`
- This compares with the following imperative definition:

```
int fact(int n)
{
    int x = 1
    while (n > 0) { x = x * n; n--; }
    return x;
}
```

Recursive Functions

- The definitions of `fact` we saw are not completely satisfactory: if we apply `fact` to a negative integer, the computation never terminates
- Rather than proceeding indefinitely, we might want to terminate the computation with a suitable error message
- One way of achieving this is to rewrite the definition with guards as:

```
fact' :: Int -> Int
fact' n
  | n < 0 = error "negative argument to fact"
  | n == 0 = 1
  | otherwise = n*(fact (n-1))
```

- The predefined function `error` takes a string as argument; when evaluated it causes immediate termination and displays the given error message:

```
fact -1
Program error:  negative argument to fact
```