Function Notation (I)

Consider the definition and application/use of a function in mathematics:

$f(x) \;\widehat{=}\; x+1$ $\qquad\qquad$ $f(42)$

In a C-like language we might write:

```
int f (int x) { return (x+1) }                f(42)
```

In Haskell we could write:

```
f1(x) = x+1                                   f1(42)
```

Usually, however, in Haskell, we write:

```
f2 x = x+1                                    f2 42
```

Function Notation (II)

Lets add a few more arguments:

$g(x,y,z) \;\widehat{=}\; x+y+z$ $\qquad\qquad$ $g(42,57,99)$

In a C-like language we might write:

```
int g (int x,y,z) { return (x+y+z) }      g(42,57,99)
```

In Haskell we could write:

```
g1(x,y,z) = x+y+z                              g1(42,57,99)
```

Usually, however, in Haskell, we write:

```
g2 x y z = x+y+z                               g2 42 57 99
```

Function Notation (III)

Why does Haskell have this strange function notation?

Reason 1
: Because, defining and using functions is so common that the notation should be as lightweight as possible.

Reason 2
: With more than one argument, the Haskell notation proves to be surprisingly flexible (and powerful!)

We'll learn about this flexibility and power later.

Function Notation (IV)

As far as Haskell is concerned, `f1 x` and `f2(x)` are the same.

However, `g1(x,y,z)` and `g2 x y z` are not:

- Their types are different:

  ```
  g1 :: Num a => (a,a,a) -> a
  g2 :: Num a => a -> a -> a -> a
  ```

- The implementation of `g2` is faster and uses less memory than that of `g1`.

Haskell: Syntactical Details

- Now time for a proper introduction to the *language* of Haskell.
- Official Reference: "Haskell 2010 Language Report"
  - Online: `http://www.haskell.org/onlinereport/haskell2010/`
- In this course we refer to sections of that report thus:
  - [*H2010* 3.4]
  - Haskell 2010 Language Report, Section 3.4

Haskell is Case-Sensitive [*H2010* 2.4]

For example, the following names are all *different*:

```
ab aB Ab AB
```

Program Structure [*H2010* 1.1]

A Haskell script can be viewed as having four levels:

1. A Haskell program is a set of *modules*, that control namespaces and software re-use in large programs.
2. A module consists of a collection of *declarations*, defining ordinary values, datatypes, type classes, and fixity information.
3. Next are *expressions*, that denote values and have static types.
4. At the bottom level is the *lexical structure*, capturing the concrete representation of programs in text files.

(We focus on the bottom three for now).

Notational Conventions [*H2010* 2.1]

- The report uses the following notation for syntax:

$[syn]$	optional occurence of syn
$\{syn\}$	zero or more repetitions of syn
(syn)	grouping
$syn_1 \mid syn_2$	choice between alternatives
$syn_{\langle syn' \rangle}$	difference—elements generated by syn, except those generated by syn'
`fibonacci`	terminal syntax in `typewriter font`

- It uses BNF-like syntax, with productions of the form:

$$nonterm \quad \rightarrow \quad alt_1 \mid alt_2 \mid \ldots \mid alt_n$$

"*nonterm* is either an alt_1 or alt_2 or . . ."

- The trick is distinguishing $\mid$ (alternative separator) from `|` , the vertical bar character (and similarly for characters `{}[]()`).

Comments [*H2010* 2.3]

A Haskell script has two kinds of comments:

1. End-of-line comments, starting with `--`.
2. Nested Comments, started with `{-` and ending with `-}`

Example, where comments are in red.

```
myfun x -- end-of-line, but -} won't end it
 = let
     y = 2 {- nested, but -- ignored here -} ; z = 3
     {-
     a = 4 {- was 42 but I changed my mind -}
     b = 5
     -}
   in y + z * x
```

Namespaces [*H2010* 1.4]

- Six kinds of names in Haskell:
  1. *Variables*, denoting values;
  2. *(Data-)Constructors*, denoting values;
  3. *Type-variables*, denoting types;
  4. *Type-constructors*, denoting 'type-builders';
  5. *Type-classes*, denoting groups of 'similar' types;
  6. *Module-names*, denoting program modules.
- Two constraints (only) on naming:
  - *Variables* (1) and *Type-variables* (3) begin with lowercase letters or underscore,
    Other names (2,4,5,6) begin with uppercase letters.
  - An identifier cannot denote both a *Type-constructor* (4) and *Type-class* (5) in the same scope.
- So the name `Thing` (*e.g.*) can denote a module, data-constructor, and either a class or type-constructor in a single scope.

Character Types (I) [*H2010* 2.2]

The characters can be grouped as follows:

- *special* : `( ) , ; [ ] ‘ { }`
- *whitechar* → *newline*|*vertab*|*space*|*tab*
- *small* → `a`|`b`|...|`z`|`_`
- *large* → `A`|`B`|...|`Z`
- *digit* → `0`|`1`|...|`9`
- *symbol* : `! #  % & * + . / < = > ? @ \ ^ | - ~`
- the following characters are not explicitly grouped- `:  " '`

(There is also stuff regarding Unicode characters (beyond ASCII) that we shall ignore—so the above is not exactly as shown in [*H2010* 2.2]).

Lexemes (I) [*H2010* 2.4]

The term "lexeme" refers to a single basic "word" in the language.

- *Variable Identifiers* (*varid*) start with lowercase and continue with letters, numbers, underscore and single-quote.
  `x x' a123 myGUI  _HASH  very_long_Ident_indeed''`
- *Constructor Identifiers* (*conid*) start with uppercase letters and continue with letters, numbers, underscore and single-quote.
  `T  Tree  Tree'  My_New_Datatype  Variant123`
- *Variable Operators* (*varsym*) start with any symbol, and continue with symbols and the colon.
  `<+>  |:|  ++  +  -  ==>  ==  &&  #!#`
- *Constructor Operators* (*consym*) start with a colon and continue with symbols and the colon.
  `:+:   :~  :===   :$%&`

Identifiers (*varid*, *conid*) are usually prefix, whilst operators (*varsym*,*consym*) are usually infix.

Lexemes (II) [*H2010* 2.4]

- *Reserved Identifiers* (*reservedid*):

  ```
  case class data default deriving do else foreign if
  import in infix infixl infixr instance let module
  newtype of then type where _
  ```

- *Reserved Operators* (*reservedop*):

  ```
    ..  :  ::  =  \  |  <-  ->  @  ~  =>
  ```

Literals [*H2010* 2.5,2.6]

We give a simplified introduction to literals (actual basic values)

- *Integers* (*integer*) are sequences of digits
  Examples: `0 123`
- *Floating-Point* (*float*) has the same syntax as found in mainstream programming languages. `0.0 1.2e3  1.4e-45`
- *Characters* (*char*) are enclosed in single quotes and can be escaped using backslash in standard ways.
  `'a' '$' '\'' '"' '\64' '\n'`
- *Strings* (*string*) are enclosed in double quotes and can also be escaped using backslash in standard ways.

  ```
  "Hello World"  "I ‘like’ you"
  "\" is a dbl-quote"  "line1\nline2"
  ```

Function Notation (V)

We can define and use functions whose names are either *Variable Identifiers* (*varid*) or *Variable Operators* (*varsym*)

For *varid* names, the function definition uses “prefix” notation, where the function name appears before the arguments:

```
myfun x y  =  x+y+y                        myfun 57 42
```

For *varsym* names, the function definition uses “infix” notation, where the function has exactly two arguments and the name appears inbetween the arguments:

```
x +++ y  =   x+y+y                         57 +++ 42
```

Function Notation (VI)

For *varid* names, with functions having two[1] arguments, we can define and use them “infix-style” by surrounding them with backticks:

```
x `anof` y  =  x+y+y                        57 `anof` 42
```

For *varsym* names, we can define and use them “prefix-style” by enclosing them in parentheses:

```
(++++) x y  =   x+y+y                       (++++) 57 42
```

We can define one way and use the other—all these are valid:

```
57 `myfun` 42            (+++) 57 42
anof 57 42               57 ++++ 42
```

[1] or more ?!?

Getting GHC

- Can't wait for the 1st exercise in order to get going?
- Strongly recommended:
  install `stack` (see
  https://docs.haskellstack.org/en/stable/README/)
- Follow the Quickstart guide
  for Unix/OS X the default behaviour is usually fine
  for WIndows read the Windows stuff carefully
  (use the installers, rather than manual download).