Examples of lists are

```
[]                                              ( 0 components )
5 : []                                          ( 1 component )
[] : [] : []                                    ( 2 components )
( 1 : 2 : [] ) : ( 3 : 4 : 5 : [] ) : []        ( 2 components )
( not True ) : ( False || True ) : ( 2*4 < 7 ) : []   ( 3 components )
```

In Haskell, all components of a list must have the same type; however, GH does not check this.

The functions `head` and `tail` are defined for non-empty lists by

```
head ( h : t ) => h
tail ( h : t ) => t
```

For example,

```
head ( 1 : 2 : [] ) => 1
tail ( 1 : 2 : [] ) => 2 : []
head ( head ( tail ( ( 1 : 2 : [] ) : ( 3 : 4 : [] ) : [] ) ) ) => 3
```

The functions `head` and `tail` may not be applied to the empty list.

FUNCTIONS

A **function** is a correspondence from an item (called its **argument**) to an item (called its **result**). Functions are written in the form

```
\ <NAME> -> <EXPRESSION>        ( \ denotes the symbol λ and is pronounced lambda )
```

where *<NAME>* represents an arbitrary argument and *<EXPRESSION>* specifies the result in terms of *<NAME>*, i.e. the item to which it is mapped. For example,

```
\ n -> n * n
```

is a function which expects a number item as argument and returns its square as result.

Function application is denoted simply by writing the function followed by its argument, i.e.

```
( \ <NAME> -> <EXPRESSION> ) <ARGUMENT>
```

and its result is obtained by evaluating *<EXPRESSION>* with each occurrence of *<NAME>* replaced by *<ARGUMENT>* (the parentheses here are required for precedence, and are not an intrinsic part of the function application itself). Thus

```
( \ n -> n * n ) 2  =>  2 * 2  =>  4
```

Every function takes a single item as argument and returns a single item as result. Both the argument and the result may be items of *any* type.

In particular, the result of a function may itself be a function. Such function-generating functions (called *curried functions*) provide a way of representing multi-argument functions. For example, the function

```
\ n1 -> \ n2 -> n1 + n2
```

when applied to an argument a_1 returns the function `\ n2 -> a1 + n2`, which when applied, in turn, to an argument a_2 returns the sum $a_1 + a_2$ of the *two* arguments. Thus

```
( \ n1 -> \ n2 -> n1 + n2 ) 2 3  =>  ( \ n2 -> 2 + n2 ) 3  =>  2 + 3  =>  5 .
```


RELATIONAL OPERATORS

The **relational operators** `==` and `/=` take two items of the same type as operands and generate the boolean item `True` or `False` according as these items are equal or unequal, respectively. For example, each of the following expressions has the value `True`:

```
1 + 1  ==  2
not True  /=  True
1 : 2 : []  ==  8-7 : 1+1 : []
1 : 2 : []  /=  2 : 1 : []
```

Two lists are considered equal iff they have the same number of components and corresponding components are equal.

Testing the equality of functions is a computationally unsolvable problem, so comparing functions using `==` or `/=` is *not* allowed.

The relational operators `<` `<=` `>` `>=` can be used to compare *numbers*.

CONDITIONAL EXPRESSIONS

A **conditional expression** is a way of selecting between the values of two expressions, based on some boolean condition. It is an *expression*, of the form

```
if <CONDITION> then <EXPRESSION-1> else <EXPRESSION-2>
```

and its overall value is the value of either *<EXPRESSION-1>* or *<EXPRESSION-2>*, depending on whether the boolean expression *<CONDITION>* is `True` or `False`, respectively.

For example,

```
\ n -> if n >= 0 then n else -n
    (this function returns the absolute value of its argument)
\ n -> if n > 0 then 1 else if n < 0 then -1 else 0
    (for multi-way branching, use another conditional expression for <EXPRESSION-2>)
```

DEFINITIONS

A **definition** is a means of attaching a *name* to an item. It has the form

```
<NAME> = <EXPRESSION>
```

and subsequently, *<NAME>* denotes the item denoted by *<EXPRESSION>*. For example,

```
hoursperweek = 24 * 7
```

causes `hoursperweek` to denote the number item `168`.

Only a single definition may ever be supplied for any given *<NAME>*.

FILES

Definitions, and nothing else, must be placed inside *files* which must have the extension `.hs`. To read and process the definitions in a given file, issue the top-level command

```
:load <FILENAME>
```

For this command, the extension `.hs` may be omitted from *<FILENAME>*.