LAZY EVALUATION

An important and powerful aspect of CH is its use of *lazy evaluation*: operations are not performed unless/until their results are actually needed. This has several significant consequences.

Call-by-Need: When a function is applied to an argument, the argument remains unevaluated until its value is needed within the function. In particular, if its value is never needed, it is never evaluated; thus, `( \ n -> 2 + 3 ) ( div 1 0 )` evaluates to 5, rather than giving an error.

Short-Circuit Boolean Operators: Unless its value is needed, the second operand of `&&` or `||` is not evaluated; thus

- if *P* evaluates to `False` then *P* `&&` *Q* is known to be `False`
- if *P* evaluates to `True` then *P* `||` *Q* is known to be `True`

and in each case *Q* need not, and will not, be evaluated.

Lazy Definitions: When a definition is processed, its *<EXPRESSION>* is not actually *evaluated* and thus can contain occurrences of *<NAME>*s which have yet to be defined. This allows:

Recursive Definitions:

```
factorial = \ n -> if n == 0 then 1 else n * factorial ( n - 1 )
```

Forward Definitions:

```
a = b + 1
b = 4
```

Mutually Recursive Definitions:

```
iseven = \ n -> n == 0 || isodd  ( n - 1 )
isodd  = \ n -> n /= 0 && iseven ( n - 1 )
```

Infinite Lists: The `:` operator is lazy, which allows infinite lists to be specified and manipulated with ease in CH. For example,

```
ones = 1 : ones
```

defines the infinite list `1 : 1 : 1 : 1 : ...`, while

```
from = \ n -> n : from ( n + 1 )
```

results in `from 1` being the infinite list `1 : 2 : 3 : 4 : 5 : ...`.

Entire infinite lists are never actually *constructed*; instead, their components are produced only upon demand, with the list being expanded just as far as is strictly necessary. For example,

```
head ( tail ( from 1 ) )
=> head ( tail ( 1 : from ( 1 + 1 ) ) )
=> head ( from ( 1 + 1 ) )
=> head ( ( 1 + 1 ) : from ( ( 1 + 1 ) + 1 ) )
=> 1 + 1
=> 2
```


FURTHER DETAILS

Operator Precedence: The operators of CH, in decreasing order of precedence, are as follows:

```
Function Application (note that div, mod, not, head, tail are functions)
*
+  -
:
==  /=  <  <=  >  >=
&&
||
```

In evaluating a multi-operator expression, operators of higher precedence are applied before those of lower precedence. Operators of equal precedence are applied from left-to-right, apart from `:` which is applied from right-to-left. However, parentheses may be used to override precedence and explicitly control the order of application of operators; this is the *only* use of parentheses in CH.

Syntax of a *<NAME>*: A *<NAME>* must start with a *lower*-case letter, and can continue with any sequence of lower-case letters, upper-case letters, digits, or the characters `'` or `_`. The words `if`, `then`, `else` are reserved and cannot be used as *<NAME>*s.

Scope: The scope of the *<NAME>* in a *function* is the associated *<EXPRESSION>*, apart from any nested functions in which that *<NAME>* is re-used; the scope of the *<NAME>* in a *definition* is the entire program, apart from any functions in which that *<NAME>* is re-used. For example, with the definition

```
n = 1
```

the following top-level expression has the value 12 (= 1 + 3 + 2 * 2 + 3 - 1):

```
n + ( \ n -> ( n + ( \ n -> n * n ) 2 - n ) ) 3 + n
```

Expressions Evaluated Once: A *<NAME>* becomes bound to an *<EXPRESSION>* during either a function application or a definition. Under lazy evaluation, when that *<NAME>* is first used, the *<EXPRESSION>* is evaluated; at that point, *all* occurrences of the *<NAME>* are re-bound to the *value* of the *<EXPRESSION>*. Thus, for example, in the function application

```
( \ n -> n * n ) ( 2 + 3 )
```

the expression `2 + 3` is evaluated only *once*; likewise, with the definition

```
hoursperweek = 24 * 7
```

the expression `24 * 7` will be evaluated only the *first* time that `hoursperweek` is used.

Format of Output: Output is produced when a top-level expression is evaluated.

- A **number** item is written out in standard decimal form.
- A **boolean** item is written out as `True` or `False`.
- A **list** item is written out by writing out each component, with sublists in parentheses, separated by `:` symbols and terminated by `[]`.
- A **function** item is simply written out as `<FUNCTION>`.

Comments: The symbol `--` introduces a *comment*, and all further text on that line is ignored.

Code Layout: Blanks, tabs, and newlines are called *whitespace* characters. When writing CH, whitespace may be used freely to separate tokens. CH does not strictly enforce the 'offside rule' of Haskell; however, in multi-line definitions, lines after the first one must be indented.