

Haskell offers two primary ways of working with code via GHCi :

i) typing directly into GHCi or REPL

ii) writing source files and loading them into the GHCi.

↓

Using the REPL (Read, evaluate, print, loop)

↓

originated with LISP, but are now common to modern programming langs. including Haskell.

→ The normal method of working with REPL and loading source files in REPL and playing with them and also unloading them. (:module)

→ Understanding expressions : everything in Haskell is an expression or declaration. Expressions may be values, combination of values, and/or functions applied to values.

↓

Expressions evaluate to a result. In the case of a literal value, the evaluation is trivial as it only evaluates to itself. In the case of an arithmetic eq., the evaluation process is the process of evaluating / computing the operator and its arguments.

↓

That is, all of Haskell's expression work to evaluate

a result in a predictable, transparent manner.



Expressions are the building blocks. Programs → big expressions made of smaller expressions.



Declarations are top level bindings which allows us to name expressions. These names can be used to refer to them multiple times without copying.

→ As mentioned before expressions are the most basic unit of a Haskell program, and functions are a special type of expression.



functions in Haskell are related to functions in mathematics, meaning they map inputs to outputs.



As in lambda calculus, all functions in Haskell take one argument and return one result.



The way to think of this is that, in Haskell, when it seems we are passing multiple arguments to a function, we are actually applying a series of nested functions, each to one argument. This is called currying.



functions can appear in expressions that form the bodies

of other functions to be used as arguments to functions ,
just as any other value can be.



→ lowercase letter init

function definition : name of the fn , followed

← by parameters of the fn , , separated only
by white space. Then an equal sign ,

which expresses equality of terms . finally

there is an expression that is the body of
the function and can be evaluated to
return a value .

corresponding to
the head of the
lambda .



CamelCase
Convention



even for variables



Haskell uses a nonstrict evaluation strategy
which defers evaluation of terms until they're
forced by other terms referring to them .

Note : canonical ≡ normal form



Haskell doesn't evaluate everything to canonical
or normal form by default . Instead , it only
evaluates to a weak head normal form (WHNF)
by default . This means that not everything
will get reduced to its irreducible form immediately .
i.e. $(\text{f} \rightarrow (1, 2 + \text{f})) 2 \Rightarrow (1, \underline{\underline{2}} + 2)$

not evaluated to 4 until
the last moment.

These are prefix functions. There are functions that operate
in an infix manner as well, like the arithmetic operators.



prefix functions can be used as infix functions, with
a small change in syntax. Using back ticks.



infix functions can be used in prefix fashion by
using them in brackets.



If the fn.name is alphanumeric, it is a prefix fn.
by default, and if the fn.name is a symbol, then they
are infix by default.

→ Associativity and precedence : info can be used to get
information about a given fn.

: info(+) \Rightarrow infixl 6 + precedence (on a scale
of 0-9)

↓
infix operator, left associative ↑
higher
precedence is
applied first

→ Order of declaration matters in RERL, but doesn't matter in
source files.

→ Declaring files in source files. first we declare the name of our module , so it can be imported by name in a project.

module Learn where

x = 10 * 5 + y

camelCase ← myResult = x * 5
y = 10

capitalized
Indentation is imp.
in Haskell.

use spaces

→ Whitespace is significant in Haskell in general.

→ Make certain when we break up lines of code

] a common
point of
error in
Haskell.

that the second line begins at least one space
from the beginning of that line.

→ Another cause of error is to not start a declaration at
a start of the line

→ '--' used to comment out lines in source code.

Operator	Name	Purpose/application
+	plus	addition
-	minus	subtraction
*	asterisk	multiplication
/	slash	fractional division
div	divide	integral division, round down
mod	modulo	like 'rem', but after modular division
quot	quotient	integral division, round towards zero
rem	remainder	remainder after division

-) usually want to use 'div' for integral division , due to the way div and quot round .
-) 'rem' and 'mod' have different use cases .

→ laws for quotients and remainders : in Haskell , if one or both arguments are negative , the results of mod will have the same sign as the divisor , while the result of the rem will have the same sign as the dividend :

$$\Rightarrow -5 \text{ 'mod' } 2$$

1

$$\Rightarrow 5 \text{ 'mod' } (-2)$$

1

$$\Rightarrow (-5) \text{ rem } 2$$

1

⇒ Negative numbers : due to interaction of parentheses, varying and infix syntax, negative numbers get special treatment

got to do more
with precedence
order, than
anything else.
+ (parathesis around
them) ↓

→ \$ operator: does almost nothing. Defn. of (\$):

$$f \circ a = f a$$

↓

infix operator with
least precedence.

used for convenience for when we want to express something with less pair of

paranthesis,



'\$' allows everything to the right to be executed first and can be used to delay fn. appln.

→ need to wrap infix operators with parenthesis. Also (+!) is the addn. fn. but with one argument applied, making it return the next argument it's applied to plus one. This is called sectioning and allows us to pass around partially applied fns.



using section with a fn. where order matters, should be taken into careful consideration.



Subtraction is a special case. Since (-) has already been used for negation.

→ let and where : used to introduce components of expressions.



constraint is that let introduces an expression, so it can be used wherever you can have an expression, but where is a declaration and is bound to a surrounding syntactic construct.

(*) scope is the area of source code where a binding of a variable applies.

(*) As we build larger projects, we will use a project manager called stack.

```
-- FunctionWithWhere.hs
module FunctionWithWhere where

printInc n = print plusTwo
where plusTwo = n + 2
```

```
-- FunctionWithLet.hs
module FunctionWithLet where

printInc2 n = let plusTwo = n + 2
              in print plusTwo
```