Haskell: The Confusing Parts

echo.rsmw.net/n00bfaq.html

Use this as a cheat sheet, not a textbook.

Misc. Stuff You Should Know Already

Quick review, because if you've heard of Haskell, you should also have heard of most of this. `foo :: Bar baz -> Bar quux` is a type signature for the function `foo`, where `Bar baz` is the type of argument it takes, and `Bar quux` is the type of the return value. Function application looks like `sin x`, not `sin(x)`, because the latter would just look goofy with curried functions (`map foo bar` vs. `map(foo)(bar)`). List syntax (`[1,2,3,4]`) is sugar for "cons" syntax (`1 : 2 : 3 : 4 : []`). An enumerated list looks like `[1..10]`, and a list comprehension looks like `[ x | x <- [1..10] ]`.

There's no `null` and no `void`. Every variable has to have a value, even if it's a placeholder. The standard placeholder value is the empty tuple, `()`, a.k.a. "unit." The type of `()` is also `()`.

Haskell's syntax is indirectly based on ML, so if you stop reading here and go learn OCaml instead, some of this information will still be kinda useful.

I rhyme "Haskell" with "ask Elle," but everyone else rhymes it with "rascal" because otherwise people keep mishearing it as "Pascal," and nobody wants that (except trolls).

Type Names and Constructors

There are two completely separate namespaces in any Haskell source file: The value namespace, and the type namespace. The former is where functions, literals, patterns, etc. reside, and the latter is where you'll find type constructors, typeclasses, and type variables.

Haskell has no need for reflection 90% of the time, so rather than make the type and value namespaces visually distinct with syntax conventions, the designers actually reused many of the same conventions in both namespaces. Thus, in order to know what `Foo bar baz` means, you have to know whether it's in the type namespace or the value namespace. Fortunately, you only have to look for the type namespace in the following places:

1. The "has type" `(::)` operator can appear as an expression in parentheses, or a separate declaration. The stuff on the left of the operator is in the value namespace, and the stuff on the right is in the type namespace, as in `5 :: Int`.
2. The `type` keyword declares a type alias (like `typedef` in C). Everything in that declaration is in the type namespace.
3. In a `class ... where` declaration, everything between `class` and `where` is in the type namespace.
4. Same deal for `instance ... where`.
5. In a `data` (or `newtype`) declaration, any new constructor names to the *right* of the `=` sign are in the value namespace. Stuff between curly braces (when using the record syntax) follows the rules for `(::)`. Everything else is in the type namespace.

The `data` and `newtype` declarations (the latter is a subset of the former) are probably the worst offenders. Just to be clear, the highlighted parts are in the value namespace:

```
data Pizza a b = Pepperoni a
               | TheWorks {
                  sauce :: a,
                  toppings :: [b]
                 }
               deriving (Eq, Ord, Show)
```

(Still writing this section. Leave comments on reddit if you have any particular concerns about it!)

Avoiding Too Many Parentheses

Lisp is known for hating infix notation, but Haskell embraces it. All functions are operators and all operators are functions. There are ten levels of operator precedence (0 through 9), not counting function application (`foo bar`), which binds tighter than any operator. That means you can always write `x + sin y` instead of `x + (sin y)`.

Function infix notation (backticks) will turn any function into an operator at precedence level 4 (although the precedence can be changed for individual functions). Operators always take two arguments. This means that whenever you have `foo (bar 1) (baz 5)`, you can change it to `bar 1 `foo` baz 5`. You can only use backticks around a function name, not an arbitrary expression, so if you want to write something like `x `f g` y`, you have to bind `f g` to some intermediate name (with `let` or `where`).

Any operator (including `` `foo` ``) can be written in prefix notation by surrounding it with parentheses: `(+)`, `(==)`, `(`div`)`. These are called *sections,* and they're regular functions like `map` and `sin`. For example, instead of `2 + 2`, you can write `(+) 2 2`. You can even leave off either the left or right operand to curry an operator. For example, `(`div` 0)` is a function that takes one argument and divides it by zero.

There are two operators in Haskell that don't show up in many other languages: Apply `($)` and compose `(.)`. These have very simple definitions (below) and are used frequently either for currying or just for the sake of leaving out some parentheses.

```
f $ x = f x
(f . g) x = f (g x)
```

```
putStrLn (take 12 (map foo (bar ++ "ack")))
```

```
putStrLn $ take 12 $ map foo $ bar ++ "ack"
```

```
(putStrLn . take 12 . map foo) (bar ++ "ack")
```

```
putStrLn . take 12 . map foo $ bar ++ "ack"
```

In terms of good Haskell style, the last example above is preferable to the others. By the way, note that `($)` has the lowest precedence (zero) of any operator, so you can almost always use arbitrary syntax on either side of it.

Writing Functions

It's called *functional* programming for a reason. Haskell actually permits more ways of turning functions into other functions than any other language I've ever seen (even J). The scariest-looking of these is easily the lambda

notation:

```
f x = (x, foo)
f = \x -> (x, foo)
```

A lambda expression, which defines an anonymous function, starts with a backslash (which is special syntax, not an operator) and extends till the next semicolon/newline/closing (brace|bracket|paren). The arrow (always `->`, never `=>`) separates the argument list from the body of the function. You can use pattern matching in the argument list (eg. `\3 -> 4`), but it's generally a bad idea, because you can only write one branch, and if that fails it'll throw a runtime exception. If you need to do pattern matching, you should do it in a separate function, and pass that function in place of the lambda.

It's pretty rare to see explicit lambda expressions in Haskell, either way, because there are much nicer ways of writing anonymous functions. For example, anywhere you might write `\x -> foo x [1..10]`, you should instead write `flip foo [1..10]`, where `flip` is a standard Prelude function that *flips* the first two arguments of whatever function you give it. The `curry` and `uncurry` functions stand in for `\f x y -> f (x, y)` and `\f (x, y) -> f x y`, respectively.

There's a whole bunch of other stuff I can't even think of right now. To see some of it, go into the IRC channel and play around with lambdabot's `@pl` command.

Statements vs. Expressions

There's no difference between statements and expressions in Haskell. If something is not an expression, then it's a declaration, even if the language definition calls it a statement.

This is more important than you think. A lot of newbies wander into the IRC channel wondering why the following code doesn't work:

```
main = do
  foo <- getLine
  putStrLn "Pull my finger"
  if length foo < 9000
     then return ()
     else sunglasses <- getLine
          putStrLn "YEEEEAAHHH"
```

The problem is that the special `do` syntax fools people into thinking that they've somehow escaped into an imperative language, when in reality, it's syntactic sugar for the `>>=` and `>>` operators and the `\foo -> ...` lambda notation. Yes, this has to do with monads, and no, I'm not going to explain them. Go read the appropriate chapter of *Real World Haskell,* or check out "You could have invented monads." I *will* point out that `return` is, in fact, not a return statement. It's a function, and an inappropriately named function, at that. Writing `return ()` in your `do` block will **not** cause the function to return.

The Indentation Thing

The keywords `let`, `where`, `do`, and `of` (as in `case foo of`) begin *layout blocks*. Under the hood, a block is enclosed in braces with the lines separated by semicolons. You can write your code this way by hand, if you like, and whitespace between the braces will be as insignificant as it gets in Haskell:

```
module Main where{main=do{putStrLn "Look at me";putStrLn
```

```
"I'm writing all my code on four lines";dice};dice=do{input<-getLine;
let{val::Int;val=read input};putStrLn$ "What a "++if val<5 then
"small number"else "not-so-small number"};}
```

Obviously, this is not the preferred way to format your code. Haskell provides the whitespace-sensitive “layout” syntax as sugar for the above. After a layout keyword, the compiler will look for the next non-whitespace character (skipping both comments and newlines). If that character is a left brace, it parses the non-layout syntax directly. Otherwise, it compares all the lines after that to the column where that first character appeared.

I prefer to explain the rest of the process using a visual aid:

```
module Main where


main = do foo 1
          foo 2
          if pizza

             then foo 5
             else bar 8
          bar 3
baz = quux
```

The red background is the left “margin” of the layout block. The compiler inserts a virtual left curly brace before the first “foo.” Every other line that lines up **exactly** with column N gets a virtual semicolon inserted before it. (The `if/then/else` expression is considered one “line” in the block.) When the compiler reaches a line indented less than N columns, it inserts a virtual right curly brace, ending the layout block. Thus, the above code desugars to the following:

```
module Main where

{main = do {foo 1
           ;foo 2
           ;if pizza
               then foo 5
               else bar 8
           ;bar 3
};baz = quux
}
```

Note that an outer block (which I have also desugared) encloses all the declarations in the file, and that the default module declaration (`module Main where`) is inserted by the compiler at the top of any file that does not explicitly declare a module. This is why you can’t split a “normal” function definition across two lines without indenting the second line.

If you like using tab characters for indentation, note that Haskell (like most languages) defines a tab as eight columns. There’s no GHC extension to change this behavior, but you can compensate by adjusting your style. A common practice is to start a new line after a layout keyword, and indent by one “tab” width. (Keep in mind that the first non-whitespace character in the block, rather than the keyword itself, is what determines the block’s indentation level.)

```
example = do
```

```
  foo
  bar
```

There's one other fairly common source of grief. A lot of people, when writing pure functions, get into the habit of lining up the titular keywords in an `if/then/else` branch. And then they get to monadic code, and the following doesn't work:

```
main = do
  if foo
  then bar
  else baz
```

After layout desugaring, this parses as `if foo; then bar; else baz`. The issue is simply that semicolons are not allowed in an `if/then/else`.