Haskell is a purely functional programming language for general purposes. Haskell provides high-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic data types, pattern matching, list description, modular system, monadic input-output system and a rich set of primitive data types, including lists, arrays, fixed precision and floating-point number

What did I like most about Haskell?

First and foremost, of course, the purity of the language. Cleanliness in both senses: purity of functions and complete absence of OOP paradigm. It's very cool that you can look at the signature of the function and see if it can produce side effects or not. This allows you to connect small functions with each other, forming larger ones that will work in a predictable manner. The absence of OOP means that there is no way to do scary things like unverified casts from the base class to the subclass.

A function is called deterministic if the value it returns depends only on the arguments. It is said that the function has no side effects, if it is not called, writes to the file, reads from the socket, changes global variables, and so on. The function is called pure if it is deterministic and has no side effects. Haskell warmly welcomes writing clean functions. However much we transfer the same arguments to the function, it will always return one result. In addition, we can be sure that when a clean function is called, no write to a file or transmission of any data on the network occurs. This greatly simplifies the development of programs. But does Haskell do without reading from files, working with the network and user input? After all, these are all side effects. Consequently, the corresponding actions can not be performed in pure functions. The fact is that in Haskell functions are divided into clean and "dirty", that is, nondeterministic and having side effects. Dirty functions are used to enter data, transfer it to clean functions, and output the result. In other words, there is such a small procedural world inside a pure functional language. Or vice versa, this is still how to look. In general, Haskell looks extremely thoughtful language. If the code is written, then it is difficult to interpret it in two ways. For example, you can not confuse the application of a function and a reference to a function, as in Scala, because in Haskell functions are objects of the first class. Or, for example, you can not confuse a function with a type or class, because all functions must begin with a small letter, and types / classes with a large one. In general, Haskell was conceived in such a way that the code on it did not turn out to be ambiguous.

Another important feature of the language is the uniformity of everything. For example, in Java, we have primitives, interfaces, classes, enumerations, arrays, annotations, but in Haskell there is only one way to declare a type - ADT. Haskell's syntax is very pleasant. Thanks to the Hindley-Milner type output system, expression types can be omitted altogether, as a result, some expressions look so short and concise that they can compete in this regard even with dynamic languages. Curly braces, semicolons are not needed. Haskell does not have such an abundance of parentheses that are in many other languages.

This is achieved due to the fact that the function is written not by specifying arguments in parentheses, but by enumerating arguments through a space. As a result, partial application of arguments works simply by enumerating only a part of the arguments.

In Haskell there are strict rules of indentation, which forbid you to write code with some horrible indentations. As a result, the code becomes more readable. The language property, without which the conversation about Haskell would be meaningless is polymorphism. It is hardly an exaggeration if I say that Haskell is the language with the maximum amount of reused code. Any at least some repetitive functionality is put into abstraction. If two functions do similar things, then they abstract the abstract part and end up in a polymorphic function. In a typical Haskell code, a large percentage of functions are higher-order functions, often overloaded. General properties of types are placed in classes: Eq, Show, Num, Read, Bounded, ... Some types require classes with more complex views: (* -¿ *) -¿ Constraint instead of * -¿ Constraint. Among such classes are monads, functors, applicative functors. But they do not stop there. Monads generalize even further - in arrows, etc.

Finally, it is worth mentioning one more feature of Haskell, which distinguishes it from other languages: laziness. In Haskell, expressions are not evaluated until they are needed. As a result, you can do absolutely awesome things like endless lists, recursive types and free streaming I / O. Streaming I / O - this is when you do not read / write all at once, but in parts, as necessary

As a result, a program that reads the contents of a file and writes it to another file looks like it reads everything at once, but does not actually allocate blocks of the size of the entire file. Among other pleasant "trifles" - static compilation, cross-platform, thread safety, a huge number of ready-made libraries.

Perhaps enough about the benefits.

Now let's talk about the shortcomings:

1. First and foremost - a high threshold of entry into the language. Yes, Haskell is beautiful, clean and concise, but it is achieved not for free, but through a long rebuilding of the brain and studying complex abstractions like monads, monadic transformers

2. Secondly, it is the difficulty of writing productive code. Because in Haskell all the code is lazy, then you can easily achieve such a situation, when in memory there are accumulated gigabytes of deferred calculations, but they are not calculated, because they are not yet requested. To prevent this from happening, one must understand the subtleties of the work of lazy calculations.

3. Finally, thirdly, this is a large number of compiler extensions. In Haskell, there are a lot of extensions that you need

to remember. Virtually no libraries are written exclusively on pure Haskell - everywhere at least one or two extensions are used. Some libraries use dozens of extensions, Some libraries use dozens of extensions to remember. Virtually no libraries are written exclusively on pure Haskell - everywhere at least one or two extensions are used. Some libraries use dozens of extensions