Avalon Programming Language Documentation

Release 0.0.1

Ntwali Bashige

Nov 25, 2018

Contents

Quantum computing is slowly but surely taking shape and it promises solutions to some problems we find hard to solve using classical computing. While quantum computers are still not very powerful for practical computing, we can start prototyping.

Just like classical computers, quantum computers need to be instructed about what to do. There are many ways to do this but the most popular one is a quantum computer driven by a classical computer. Right now, there are a few programming languages that target quantum computers but many seem not to be keeping pace with current developments.

That's the niche that the Avalon Programming Language seeks to fill. At the moment, the implementation is an interpreter that runs on your computer. This interpreter also serves as the reference implementation. The next step is to start porting to the Quil instruction set. This instruction set is the most promising in terms of where the classical-quantum computing architecture is headed.

The Avalon Programming Language is a statically typed language based on algebraic data types. It features type inference for variable declaration when the compiler can deduce the complete type. With extended overloading, it gives users the ability to create more powerful functions that are intuitive to users. The language also allows generic programming with the ability to parametrize both function and type declarations. With pattern matching, you are able to inspect inside structures and make decision based on their content.

The Avalon Programming Language is a free and open source project under the MIT license giving you the freedom to evolve the language and contribute back if you so desire.

We hope you will find the language useful and more important, we hope it will make quantum computing accessible to you and everyone interested.

CHAPTER 1

Structure

The programming language is made of two parts:

- *A classical interpreter*: an interpreter for classical computing or in layman's terms, an interpreter for your everyday computer.
- *A quantum interpreter*: an interpreter that offers quantum data types and quantum gates that work on those types. It is built on top of the classical interpreter.

This documentation covers all the different parts but individual interpreters that omits certain parts will be made available. At the moment though, only a single interpreter is available, the quantum interpreter since with it classical and quantum computation can be accomplished.

CHAPTER 2

Topics

2.1 Introduction

In this introduction, we introduce Avalon's classical and quantum features. Using a small example, the user is guided through the use classical and quantum computing features of the language.

Note: Use Phil's algorithm here. Basically, we introduce superposition and entanglement and show how they are created and used in Avalon.

2.2 Installation

Currently, you can only install from source. That is until we produce binaries for different plateforms.

2.2.1 Installation from source

First, you will need to download the Boost libraries since the project depends on it. The header files needed are already in the *deps* folder so you only need to compile dependencies that require separate compilation.

1. **Download Boost**

First, you need to download the `Boost libraries` from boost.org. Extract the archive in a folder of your choice. For the purpose of this manual, we are using ~/Downloads:

```
~/Downloads$ tar --bzip2 -xf /path/to/boost_1_68_0.tar.bz2
```

Note: Make sure to use Boost 1.68 since it is the version against which developement is currently occuring. Also, development is occuring on a Linux system so please select the `Unix` variant of the libraries.

2. **Download Avalon**

Now, download the `Avalon source` from github.com. Extract the archive and copy Avalon in the directory of your choice. Again, for the purpose of this tutorial, we are using `~/Downloads`.

3. **Compile and install Boost into Avalon**

To begin, configure `Boost` to install its compiled libraries in `usr/local/lib` and its header files in `~/Downloads/avaloniq-master/deps/boost` by doing the following:

```
~/Downloads/boost_1_68_0$ ./bootstrap.sh --with-libraries=filesystem --libdir=/usr/
↪local/lib --includedir=~/Downloads/avaloniq-master/deps/boost
```

Next we perform the compilation and library installation:

```
~/Downloads/boost_1_68_0$ sudo ./b2 install
```

4. **Compile Avalon**

Compiling `Avalon` is very simple:

```
~/Downloads/avaloniq-master$ make clean && make
```

5. **Install Avalon**

We perform a system-wide installation of `Avalon` so you can run the interpreter from any directory. To perform the installation, run:

```
~/Downloads/avaloniq-master$ sudo make install
```

6. **Set AVALON_HOME**

The interpreter comes with an SDK that's being updated. This SDK lives at `/usr/lib/avalon-sdk`. You need to set `AVALON_HOME` to point to that directory so the interpreter can find the SDK programs.

You can set `AVALON_HOME` directly for one session with the following:

```
$ export AVALON_HOME=/usr/lib/avalon-sdk
```

You can also edit `~/.bashrc` and add the same line in it and have the SDK always accessible between sessions.

2.2.2 Running your program

The interpreter expects only the file that contains the function `__main__`.

To run your programs, invoke the interpreter from your terminal followed by your file with the main function and optional arguments separated by a space:

```
$ avaloni prog.avl arg_1 arg_2 arg_n
```

2.3 Tutorial

In this tutorial, we continue what the introduction started and introduce the reader to quantum communication. At the end of this tutorial, the reader will have a pretty good idea what quantum computing offers and hopefully decides to study the subject in depth.

2.3.1 Basics

The reader is introduced to the basics concepts that will aid the understanding of this tutorial. We also introduce the objective we seek to accomplish in this tutorial and motivate the reader in reading till the end.

2.3.2 Elementary coding

The reader learns about a simple way of classical information distribution that takes advantage of entanglement.

2.3.3 Super-dense coding

We walk the user through a technique that allows him/her to transmitt two classical bits using one quantum bit.

2.3.4 Teleportation

Finally, we show the user how to send a quantum bit using two classical bits using a technique aptly named teleportation.

2.4 Deutsch's algorithm

Deutsch's algorithm will be our first quantum algorithm to look into. As will be the style throughout the documentation, we will try to keep the mathematics to minimum and focus on the implementation. This doesn't mean that we won't try to understand why and how the algorithm works but we will seek to only use the mathematics necessary to get us to our goal. If it is your wish to study the algorithm in depth, references are given at the end of the section.

Here is how this section is organized: first, we will we will set up the problem with all the information necessary to understand the problem we are trying to solve. Then we will state the problem and comment on it. Afterwards we shall present a classical solution to the problem and comment on the solution. Following the classical solution we will present a quantum solution. Then we shall close with some concluding remarks.

2.4.1 Introduction

Imagine the following: you are given a function $f : \mathbb{B} \to \mathbb{B}$ where $\mathbb{B} = \{0, 1\}$. So this function takes zero or one and returns zero or one.

What are the possible input/output combinations of this function?

- First combination: $f_0(0) \to 1$ and $f_0(1) \to 1$.
- Second combination: $f_1(0) \to 0$ and $f_1(1) \to 1$.
- Third combination: $f_2(0) \to 1$ and $f_2(1) \to 0$.
- Fourth combination: $f_3(0) \to 0$ and $f_3(1) \to 0$.

Notice that for the first and the fourth combinations, the output doesn't change irrespective of the input. The output is constant; hence the function is said to be constant in both combinations. The second and the third combinations though are different: the number of zeros in the output is the same as the number of ones. There is a balance in the output; hence we call it balanced in both combinations.

Balanced and constant functions

A function $f : \mathbb{B} \to \mathbb{B}$ is said to be constant if $f(0) = f(1)$. It is said to be balanced if $f(0) \neq f(1)$.

There are lots of interesting questions we can ask about this function (yes, actually a lot). But here is one we are going to try to solve: suppose we are given one of the four combinations randomly. Without being told which one or being allowed to look inside, we are asked if it is balanced or constant. This is called Deutsch's problem after David Deutsch who first proposed a quantum solution to this problem in 1985 (and mind you, his first solution worked only half the time). When we are given a function which implementation we don't know, we call it a black box or more formally an oracle.

Deutsch's problem

Let $f : \mathbb{B} \to \mathbb{B}$ be an oracle. Is f balanced or constant?

One remark though, however obvious, is that we are going to need to call the oracle and pass it our input(s). Note that nothing is said about reading the output. Read on and find out why.

Note: Throughout the section, we are going to assume that we are given combinations as oracles of the form $f_i(n)$ for $i \in \{0, 1, 2, 3\}$ and $n \in \mathbb{B}$.

2.4.2 Classical solution

So we are given the oracle, now we need to find out if it is balanced or constant. As programmers, part our business is finding out how to solve problems efficiently. Many things may go into the final solution but right now we do know that we would like to reduce the number of times we call the oracle. The more we call it the longer it will take to make our decision of whether we have a balanced or a constant oracle.

We can visualize the classical algorithm running as shown below.

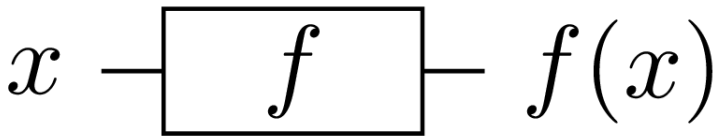

Fig. 2.1: Classical oracle executing the function f.

So how many times exactly do we need to call the oracle here?

If we call the oracle passing it 0, the first combination will return 1. But the third combination will also return 1. Also, the second combination will return 0 when given 0 as input. We cannot hope to know if the oracle is balanced or constant by passing it 0 alone. The same argument applies when 1 is given as input.

But note that if we call the oracle with 0 then with 1, we can make some progress. If $f(0)$ and $f(1)$ are equal then we know the oracle is constant by definition. The same argument can be used by definition of a balanced function.

Therefore we need to call the oracle once for $f(0)$ and once for $f(1)$ to solve the problem. Two calls to the oracle are *necessary* (we can't make a decision with only one call) and *sufficient* (we learn nothing new by a third call).

Here is the classical code that solves Deutsch's problem. You can find it in the `deutsch` folder in the algorithms repository at the Classical and Quantum Algorithms in Avalon.

```
import io
import oracles.classical


def __main__ = (val args : [string]) -> void:
    -- step 1 : call the oracle with 0
    val left = Oracle.f0(0b0)

    -- step 2 : call the oracle again but with 1
    val right = Oracle.f0(0b1)

    -- step 3 : compare both results to determine if the oracle is constant or␣
↪balanced
    if left == right:
        Io.println("The oracle contains a constant function.")
    else:
        Io.println("The oracle contains a balanced function.")

    -- we are done
    return
```

Notice that we are calling the oracle twice, first in step 1 then in step 2. Therefore, any algorithm that allows us to solve the exact same problem in less than two calls (that is in one call) is better than the current classical algorithm. And coming right next up is that solution, first due to David Deutsch.

2.4.3 Quantum solution

Quantum algorithms are a bit harder to figure out and harder to reason about concerning their correctness. But we will do that here at the expense of explaining the oracles.

If you read the code for classical oracles, they are not hard to understand. But it is not immediately obvious how they got translated to quantum oracles. No matter, it is not our objective to construct the oracles, you are not supposed to peek into them anyway. So we are going to focus on the algorithm itself.

Classical oracle to quantum oracle

To get started, we need to transform the way the classical oracle is called into a flow the quantum algorithm can work with. We can't use the flow in Fig. 2.1 because it is not reversible. So we need to build an equivalent flow that has the same effect but runnable on a quantum computer.

To make our oracles reversible, we use the following scheme, dubbing it *XOR encoding of boolean functions*.

XOR encoding of boolean functions

Let $f(x_1, x_2, \ldots, x_n) : \mathbb{B}^n \to \mathbb{B}$ be a boolean function.
Define $U_f(x_1, x_2, \ldots, x_n, y) : \mathbb{B}^{n+1} \to \mathbb{B}$ as $U_f(x_1, x_2, \ldots, x_n, y) = (x_1, x_2, \ldots, x_n, y \oplus f(x_1, x_2, \ldots, x_n))$.
The function $U_f : \mathbb{B}^{n+1} \to \mathbb{B}$ is the XOR encoding of $f : \mathbb{B}^n \to \mathbb{B}$ and is equivalent to it up to the ancilla y.

So we have transformed our classical function into a new function that is equivalent to it but with two important properties:

- The function $U_f : \mathbb{B}^{n+1} \to \mathbb{B}$ is reversible.
- The original function $f(x_1, x_2, \ldots, x_n) : \mathbb{B}^n \to \mathbb{B}$ ouput can be recovered by taking $y \oplus f(x_1, x_2, \ldots, x_n) \oplus y$.

The two properties above of the function $U_f : \mathbb{B}^{n+1} \to \mathbb{B}$ mean that it is executable on a quantum computer and from the answer it provides we are able to recover the original answer the classical function would have given.

As mentioned above, we won't see how to build oracles from $U_f : \mathbb{B}^{n+1} \to \mathbb{B}$ but it is a good exercise if you want to try it. Neither are we going to actually find the output. We are going to do the following though:

- See how to use the oracles from $U_f : \mathbb{B}^{n+1} \to \mathbb{B}$ in the algorithm.
- Understand how the algorithm solves Deutsch's problem.

To begin, we are going to simplify the XOR encoding and limit $f(x_1, x_2, \ldots, x_n) : \mathbb{B}^n \to \mathbb{B}$ to $f(x) : \mathbb{B}^n \to \mathbb{B}$. This means that its encoding is given by $U_f(x, y) : \mathbb{B}^2 \to \mathbb{B}$.

Then we are going to shift to the bracket notation in order to simplify calculations and make $U_f(x, y) : \mathbb{B}^2 \to \mathbb{B}$ accept inputs of the form $|x, y\rangle$. For our satisfaction, let us show that $U_f(|x, y\rangle) : \mathbb{B}^2 \to \mathbb{B}$ is both reversible and $f(x)$ can be recovered from it.

Let us first look at a circuit similar to the one in Fig. 2.1.

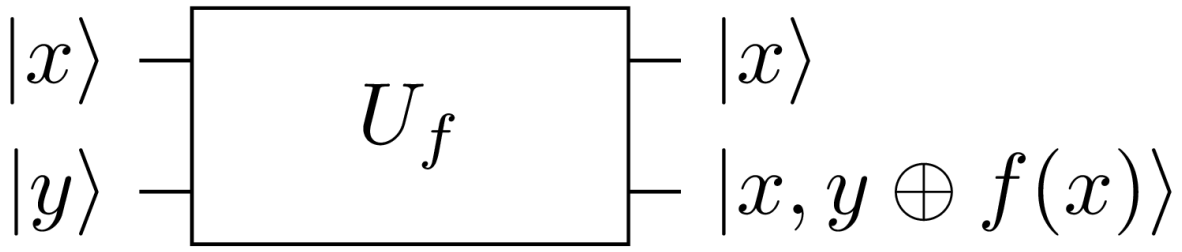

Fig. 2.2: Quantum oracle executing the function f using its encoding $U_f(|x, y\rangle) = |x, y \oplus f(x)\rangle$.

The oracle is given two bits in the form $|x, y\rangle$ and produces output of the form $|x, y \oplus f(x)\rangle$. Looking at Fig. 2.2, we can see how the quantum oracle is truly quantum and at the same time can be used to get back the classical oracle.

- To get back the original oracle from the output, we ignore $|x\rangle$ and XOR $|y \oplus f(x)\rangle$ with $|y\rangle$ resulting in $|f(x)\rangle$ which is the result of the classical oracle.
- To prove that the quantum oracle is truly quantum and therefore must be reversible we only need to show that executing the oracle passing it its own output gives back the original input. To show that, let $z = y \oplus f(x)$. Thus the new input is $|x, z\rangle$. Giving that input to the oracle, the expected output is $|x, z \oplus f(x)\rangle$. This output is equivalent to $|x, (y \oplus f(x)) \oplus f(x)\rangle$. Rearranging, we get $|x, y \oplus (f(x) \oplus f(x))\rangle$. And finally eliminating $f(x)$ due to XOR, we get as final output $|x, y\rangle$. And with that we have the original input! Therefore the quantum oracle is reversible.

Deutsch's algorithm

We are ready to tackle the quantum algorithm. We won't discuss how to derive it and will limit ourselves to understanding how it works. Why it works is another matter that is not presented and you are encouraged to read references given at the end of this section.

We present a circuit description of the algorithm from which we shall derive the final program.

Using Fig. 2.3 as reference, we are going to analyze what the algorithm does and how we find out if the oracle is balanced or constant from its output.

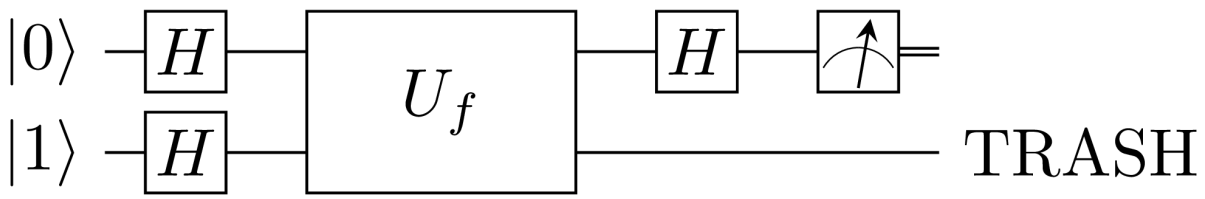

Fig. 2.3: Deutsch's algorithm as the quantum solution to Deutsch's problem.

2.5 Syntax

Avalon's syntax is similar to that of Python with a few differences here and there. It is designed to be consistent and predictable.

All Avalon programs must be stored in a text file ending with the `.avl` extension. Right now, as an interpreted language, programs are run from top to bottom just like a scripting language. The time will come when Avalon programs will be compiled for real hardware but even then nothing is expected to change.

2.5.1 Comments

Like many programming languages, Avalon features single line comments and multiple lines comments.

A single line comments is made of two hyphens.

```
-- this is a single line comment.
```

Multiple lines comments start with `-[` and end with `]-`. They can also be nested.

```
-[
This is a multi-line comment.
-[ It can also contain another multi-line comment like this one ]-

-- And single line comments can also appear in it.
]-
```

2.5.2 Reserved words

As Avalon is still in development, the number of reserved words is expected to increase. But at the moment, here is a list of reserved words:

```
not, bnot, or, bor, and, band, xor, lsh, rsh, in, not in, next in, prev in, is, is␣
↪not,
import, namespace, public, private, ref, dref, type, def, var, val, cast, switch,␣
↪case,
default, if, elif, else, for, empty, while, continue, break, pass, return
```

2.5.3 Identifiers

Type, function and variable declaration naming rules are similar to that of many languages. Unfortunately, for the moment, only ASCII names are supported but in the future Unicode is expected to be supported. Identifiers can start with a letter or underscore followed by more letters, digits and undercores.

Danger: A single underscore cannot be used as a variable name. The single underscore is reserved as wildcard in pattern matching.

Attention: Avoid at all cost to use double underscores around variables since this convention is reserved for builtin functions and variables.

Here are examples of valid identifiers.

```
hello
_sum
c_case
camelCase
PascalCase
```

And the following must be avoid and used under no circumstances.

```
_ -- the compiler will reject the presence of this token outside of a pattern␣
↪matching expression.

-[
while these are valid identifiers, there is always the possibility they will collide␣
↪with an internal identifier now or in the future.
]-
__hello__
__world
bonjour_monde__
```

2.5.4 Newlines

Newlines are very meaningful in Avalon. Newlines are used at the end of an expression statements, they figure after a statement introduction and after type and function declarations.

```
var name = "John Doe" -- a new line must be at the end of an expression statement
Io.println(name)
```

The language allows newlines when declaring multiple variables in one line for better code.

```
var name = "Jane Doe", -- a new line is allowed here for code with better readability
    age  = 32,
    sex  = Female
```

It sometimes happen that one needs a newline where the language wouldn't allow it otherwise. This is facilitated with the use of a backslash \.

```
if  age === 32 and \ -- we use the backslash to let the compiler know that it should␣
↪expect the condition to carry onto the next line
    sex === Female:
    Io.println("Adult female.")
```

2.5.5 Indentation

Indentation is how you form blocks in Avalon. And Avalon is very strict on indentation. Two characters can be used for indentation: whitespace and tabulations. But Avalon imposes two extra rules on what form valid indentation:

- Whitespace and tabs cannot be mixed. It is either one or the other.
- All indentation must be multiples of the very first indentation. This means that if the first indentation is 4 whitespaces long, a 6 whitespaces indentation will not be allowed anywhere else in the entire source file.

Here is an example of valid indentation.

```
type sex = ():
    Male
    | Female
    | Other
```

The same code with invalid indentation.

```
type sex = ():
    Male
  | Female -- This indentation has two spaces while the previous one has four spaces
  | Other
```

Few tokens are allowed to appear at the beginning of a line that's been indented. The following tokens are allowed to do so:

```
logical not(!), bitwise not(~), plus(+), minus(-), end of namespace(/-),
beginning of string("), digits, identifiers, type declarator (type), function␣
↪declarator(def),
variable declarators(var and val)
```

Therefore, if you were to begin a line with say a multiplication sign(*), the compiler will emit an error.

There are 3 places where indentation is ignored by the language: inside parentheses, square brackets and curly braces. This means that you can write function arguments and parameters on multiple lines for better readability.

```
-- indentation inside parentheses is ignored so you can do as you please indentation-
↪wise
val package = (
    name    = "Input/Output",
    author  = "John Doe",
    version = "0.0.1"
)

-- indentation inside square brackets is ignored as well
var physicists = [
    "Isaac Newton",
    "Albert Einstein",
    "Marie Curie",
    "Edward Witten",
    "Donna Strickland"
]

-- indentation is ignored inside curly braces as well
var user = {
    "name": "Jane Doe",
    "street": "4683  South Street",
}
```

2.5.6 Precedence and associativity

While we will go over the meaning of expressions later on and how to use them, you can find in the table below how operators used to form expressions interact. The operator at the top binds tighter than the one at the bottom. Those on the same line have the same precedence but associativity is used to decide which is used before which.

Table 2.1: Operator precedence and associativity

Precedence	Operator	Description	Associativity
1	()	Grouping, function call	Left
2	[]	Subscript	Left
3	.	Member access	Left
4	~	Bitwise not	Right
5	+, -	Unary addition and negation	Right
6	cast	Cast	Left
7	dref	Dereference	Left
8	ref	Reference	Left
9	*, /, %, **	Multiplication, division, modulo, power	Left
10	+, -	Binary addition and substraction	Left
11	<<, >>	Left and right bit shifting	Left
12	&	Bitwise and	Left
13	^	Bitwise xor	Left
14	|	Bitwise or	Left
15	===, =!=, ==, !=, >, >=, <, <=, in, not in, next in, prev in, is, is not	Pattern match, pattern does not match, equal, not equal, greater than, greater or equal to, less than, less or equal to, member of, not member of, next member of, previous member of, reference indentical, reference not indentical	Left
16	! (not)	Logical not	Right
17	&& (and)	Logical and	Left
18	|| (or)	Logical or	Left
19	=	Assignment	Right

2.6 Type system

Avalon is based on algebraic data types. In this section, we introduce how types are created, then we look at built-in types and finally we explore restrictions on types.

2.6.1 Anatomy of a type

A type is a made of possible parameters and a list of value constructors. Instead of showing in one shot the syntax of types, we proceed from the simplest example. Consider the `bool` type. This is a built-in type but how would I one go about creating it? Watch:

The bool type

The bool type is a built-in type but can be constructed in user code. We use it as our first example to show how types are constructed and introduce the terminology.

```
type bool = ():
    True
    | False
```

Let us understand the declaration line by line.

The first line is the type header.

```
type bool = ():
```

The `type` keyword is used to let the compiler know that a type declaration is coming up. After the `type` declarator, the type name follows. In this case, the type constructor name is `bool`. After the type name and the equal sign, follows are type parameters to appear inside the parentheses. In our case, there are not type parameters. The type header always ends with a colon `:`.

The second line is a value constructor. A value constructor is responsible for creating the atomic data (values) that your users will be interacting with.

```
True
```

In this line, we create a single value contructor called `True` that constructs a single value also called `True`.

On the third line we have the second value constructor called `False`. The vertical bar acts as a separator between value constructors.

Attention: Value constructors must always appears on different lines. If you come from Haskell, you could have placed both `True` and `False` constructors on the same line but in Avalon, this is not allowed.

Now that we have seen how to create our first type, let us clarify a few concepts that were introduced.

- A type refers to the name of the type that comes after the type declarator.
- A type constructor creates a type instance. In our case above, the `bool` type constructor creates the `bool` type instance. With the next example, it will become quite clear why the distinction is made.
- A value constructor creates values. So the `True` value constructor creates the value `True`.

Note: As a matter of convention, type names are always in lower_case. Value constructor names are always in PascalCase.

The maybe type

The maybe type is also built-in. We are going to use it to show how types can be parametrized. This will also highlight why we make a difference between a type constructor and a type instance.

```
type maybe = (a):
    None
    | Just(a)
```

The `maybe` type admits a parameter called *a*. So what is that parameter and what makes a valid parameter?

A type parameter always us to constructs type instances that depend on other type instances. In the case of the `maybe` type, we can have the following as valid type instances: `maybe(bool)`, `maybe(int)` and so on.

> **Danger:** A type parameter in the type header must not share the same name with an existing type. Hence, as a matter of convention, type parameters are single letters while it is discouraged to create types with a single letter as type name. The compiler doesn't enforce this though.

Let us further expand on type instances. The `maybe(bool)` is a type instance while `maybe(a*)` is a type constructor. The star in `a*` indicates that `a` is to be replaced with a proper type.

Each value created by a value constructor has a type instance that comes from the type constructor. In other words, a value constructor creates values with a type instance created by value constructor type constructor.

The `maybe(bool)` type instance has as possible values `None`, `Just(True)` and `Just(False)`.

Another important concept to remember is that `None` and `Just(a)` are called default constructors. This is to distinguish them with record constructors and they will be introduced next.

The point type

We create a new type that demonstrates the different types of value constructors that we mentioned above.

```
type point = (a):
    Point(a, a)
    | Point(x : a, y : a)
```

The `point` type above demonstrates two types of constructors: a default constructor and a record constructor. The difference is simple: a record constructor has its fields named while a default constructor doesn't. As one can see from the snippet above, the record constructor `Point(x : a, y : a)` conveys more information than the default constructor `Point(a, a)`.

In a record constructor, we have a comma separate list of fields with each field having the syntax: **field_name : field_type_instance**.

The compiler comes with the built-in types `int` and `float` so we can create dicrete points of type instance `point(int)` and continuous points with type instance `point(float)`.

At the moment, that is all there is to know about user defined types. Some restrictions are in place but they are going to be introduced at the right time and place.

2.6.2 Built-in types

In this section, we introduce built-in types, their special features and restrictions that apply to them.

The void type

The `void` type creates a type instance without any values. It can be used as any other types but the compiler will prevent its use in certain places due to other restrictions. For instance, one can declare a variable of type instance `void` but since all variables must be initialized and `void` has no element, that variable declaration will be rejected by the compiler.

The unit type

The unit type is recognized by the compiler as `()` and it has one element also called `()`. As a type, when one is writing purely functional programs, it is used where `void` is used to indicate the lack a *meaningful* value. This convention is not followed by Avalon though.

The bool type

The `bool` type has two value constructors called `True` and `False`. It has been elaborated on above and there is nothing else interesting to say about it.

The following operations are currently supported on `bool` values: logical conjuction, logical disjunction and logical negation. The cast operator is enable allowing casting of `bool` values to `string`. Equality and lack of equality is supported as well. Pattern matching is enabled for booleans as well.

```
-- logical conjuction
True and False
True && False

-- logical disjuction
False or False
False || False

-- logical negation
not True
! True

-- Cast to string
cast(True) -> string
string(True)

-- Comparison
True == False
False != False

-- Pattern matching
True === True
False =!= True
```

The int type

The `int` type is the type of integers. Internally it corresponds to the biggest interger value that the machine the program is running on can support. Integer literals look the same as in other languages. But Avalon also allows placing single quotes in them for better readability.

```
23
0
1233
76'456
```

The following operations are currently supported on `int` values: uninary addition, negation, addition, substraction, multiplication, division, modulus and exponentiation. The cast operator is enabled for `string` and `float` allowing casting an integer to a string and a floating point number respectively. The following comparators are enabled on integers: equal, not equal, greater than, greater or equal to, less than and less than or equal to. Pattern matching is available on integers.

```
-- Operations
-- unary positive
+2
-- unary negative
-2
-- addition
1 + 2
-- substraction
1 - 3
-- multiplication
1 * 3
-- division
3 / 2
-- modulus
5 % 2
-- exponentiation
3 ** 2

-- Casting
-- cast to string
cast(12) -> string
string(12)
-- cast to float
cast(12) -> float
float(12)

-- Comparison
-- equal
1 == 1
-- not equal
3 != 2
-- greater than
34 > 12
-- greater or equal to
34 >= 34
-- less than
45 < 12
-- less or equal to
23 <= 90

-- Pattern matching
12 === 34
12 =!= 34
```

The float type

The `float` type is the type of floating point numbers. Internally it correponds to the highest precision that the machine the program is running on can support. Floating point numbers as currently supported are written with a integral part and a decimal part. Scientific notation is not yet supported.

```
0.0
1.234
12'097.34'912
```

The following operations are supported on floating point numbers: unary positive, unary negative, addition, substraction, multiplication and division. The cast operator is enabled for `string`.

```
-- Operations
-- unary positive
+2.0
-- unary negative
-2.0
-- addition
1.0 + 2.5
-- substraction
1.4 - 3.6
-- multiplication
1.5 * 3.23
-- division
3.3 / 2.3

-- Casting
-- cast to string
cast(12.5) -> string
string(12.3)
```

The string type

The `string` type is the type of character sequences. All string literals appear enclosed inside double quotes. At the moment, character escaping is not support and neither is Unicode but both are coming before release 1.0.0.

```
"Hello"
"Salut"
"Jambo"
"Hisashiburi" -- you can look forward to writing this in Unicode in the future
```

The following operations are enabled on strings: concatenation and reversal. Pattern matching is enabled on strings. Since `string` implements the `__hash__` function, its values can be used dictionary keys.

```
-- concatenation
"Hello " + "world!"

-- reversal
-"madam"

-- pattern matching
"madam" === "madam"
```

The `string` type has the following restriction:

- A variable of `string` type instance must be immutable.

The bit types

There are 4 bit types: `bit`, `bit2`, `bit4` and `bit8`. They correponds to bitset of size 1, 2, 4 and 8. They are created by writing `0b` followed by a series of zeros and ones. The number of zeros and ones must correspond to the type instance. Hence there cannot be a bitstring with 6 zeros and ones.

```
0b1         -- type instance <bit>
0b10        -- type instance <bit2>
0b1001      -- type instance <bit4>
```

(continues on next page)

(continued from previous page)

```
0b1001'0011 -- type instance <bit8>
            -- note we placed a single quote to help with readability
```

The following operations are currently available on bitstrings: bitwise not, bitwise and, bitwise or and bitwise xor.

```
-- bitwise not
~ 0b0
bnot 0b0

-- bitwise and
0b0 & 0b1
0b0 band 0b1

-- bitwise or
0b0 | 0b0
0b0 bor 0b0

-- bitwise xor
0b1 ^ 0b0
0b0 xor ob0
```

The qubit types

At the moment, only one qubit type is fully supported and is called `qubit`. While `qubit2`, `qubit4` and `qubit8` are recognized, no operations can be performed on them.

```
0q1         -- type instance <qubit>
```

There are multiple restrictions on qubits that are listed here but will be reiterated later on again.

- A variable with qubits cannot be mutable.
- A variable with qubits cannot be copied into another variable either by direct assigment or by passing it to a function.
- A reference to qubits cannot be dereferenced.
- Qubit type instances cannot be used as type instances parameters not as value constructors fields parameters.

The tuple type

Avalon comes with two types of tuples: named tupes and unnamed tuples. Tuples are enclosed in parentheses.

1. **Named tuples**

A named tuple is of the following form:

```
-- a named tuple of type instance <(string, int)>
(name = "John Doe", age = 32)
```

Named tuples have the following operations enabled on them: member access.

```
-- accessing the name of the named tuple in the previous example
tuple.name
```

Named tuples have two restrictions:

- They cannot be used to initialize local variables, only global variables.
- They cannot be passed as function arguments.

These restrictions will be lifted when/if refinement types are introduced.

2. **Unnamed tuples**

An unamed tuple is of the following form:

```
-- an unnamed tuple of type instance (string, maybe(int))
("Jane Doe", Just(32))
```

Unnamed tuples have the following operations enabled on them: indexing.

```
-- accessing the first element of an unnamed tuple
tuple[0]
```

Tuples have the following restriction:

- A variable containing a tuple cannot be mutable.

The list type

Lists are arrays of elements of the same type. Lists are enclosed inside square brackets.

```
-- a list of type instance <[int]>
[1, 2, 3, 5, 7, 11]
```

The following operations are available on lists: indexing.

```
-- accessing the first element of a list
list[0]
```

Lists have the following restrictions:

- A variable containing a list cannot be mutable.

The map type

Maps are dictionaries with keys of same type instance and values of same type instance as well. Maps are enclosed inside curly braces.

```
-- a map of type instance <{string:int}>
{
    "age": 32,
    "year": 1986
}
```

The following operations are available on maps: indexing.

```
-- get the value associated with the year key
map["year"]
```

Maps have the following restrictions:

- A variable containing a map cannot be mutable.

2.6.3 Reference type instances

References are aliases to external resources. The values they alias can be obtained by dereferencing the reference. References are created with the `ref` keyword both for type instances and for values. Observe:

```
-- create a reference to a variable of type string
var name = "John Doe"
var alias = ref name    -- alias has type instance <ref string>

-- we get the original name by perform a dereference with type instance <string>
var original_name = dref alias
```

The following operations are available on references: identity comparison.

```
-- variables to reference
val q1 = 0q0, q2 = 0q1
val ref_q1 = ref q1, ref_q2 = ref q2

-- check if two references are identical - meaning they reference the same variable
if ref q1 is ref q2:
    Io.println("Both references alias to the same variable.")

-- check if two references are not identical - meaning they don't reference the same␣
↪variable
if ref_q1 is not ref_q2:
    Io.println("Both reference do not alias the same variable.")
```

References have the following restrictions:

- A variable containing a reference is immutable. It means that a reference variable cannot reasigned once set.
- References cannot be returned from functions. This is to avoid dead references.
- Reference to references are not allowed.

2.7 Expressions

Expressions arise from the user of literals, operators and function calls. Due to the type system, the need may arise to supplement type information on expressions in order to write better code. Hence, on select expressions under specific circumstances, type instances can be set on expressions.

This functionality becomes handy when declaring variables where one doens't want to set the type instance on the variable itself, maybe due to style concerns. Or more importantly when passing an expression with an incomplete type to a function as argument.

2.7.1 Literal expressions

Though totally not needed, the ability to set type instances on literals is allowed.

```
-- type information on an integer literal, just in case
12:int

-- string
"Hello World":string
```

2.7.2 Values from value constructors

The ability to set type instances on expressions become important. If one wants to pass, say `None` as a function argument, one has to supply it with complete type information. This because expressions passed to functions must always have a complete type and `None` doesn't.

```
-- calling a function that expects <maybe(int)> and passing it None
-- the following will fail since ``None`` has incomplete type information
f(None)
```

To fix the code above, one supplies a type instance to `None` as follows:

```
-- the following will work as expected
f(None:maybe(int))
```

The ability to attach a type instance to (select) expressions avoids the need to always declare a variable and attach the type instance on the variable.

Here is a more complicated example that illustrates the point.

```
-- a gender type
type gender = ():
    Male
    | Female
    | Queer

-- a user type
type user = (a):
    User(
        name    : string,
        age     : int,
        gender  : maybe(a)
    )

-- we create a user with an incomplete type
var u = User(
    name    = "Alexis Doe",
    age     = 21
    gender  = None
):user(gender) -- since gender is <None>, we need to give a complete type and here we␣
↪attached it on the expression itself

-- the above is equally similar to the following
-- we attach the type instance on the variable name itself
var u:user(gender) = User(
    name    = "Alexis Doe",
    age     = 21
    gender  = None
)
```

If one needed to pass a plain user value without declaring a variable, this syntax becomes the only way to provide values with complete type information to functions.

2.7.3 Tuple expressions

As with values constructed from value constructors, tuples can also have type instances attached to them for the same reasons elaborated on above.

```
-- a tuple with an incomplete type made complete by attaching a type instance
var tuple = ("Margaret Doe", None):(string, maybe(int))
```

Warning: If you have a tuple of only one element, *always* add a comma after that one element else you will be created a grouped expression instead of a tuple. This works as follows: `(True,)`. Note the comma after the only element that makes the tuple.

2.7.4 List expressions

Type instances can be attached to lists as well. This is particularly important for empty lists since every expressions must have a type instance in Avalon.

```
-- things can get hairy quite fast
var list = [Just(None), Just(Just(0))]:[maybe(maybe(int))]

-- empty lists must have a type instance
var list = []:[int]
```

2.7.5 Map expressions

Type instances can be attached to maps as well.

```
-- we declare an empty map with strings as key and integers as values
var others = {}:{string:int}
```

2.7.6 Conditional expressions

Avalon offers conditional expressions in order to avoid the use of an if statement for simple expressions. The syntax is `primary_expression if condition else alternative_expression`.

The `primary_expression` will be returned if the `condition` is True and the `alternative_expression` will be returned otherwise.

```
val age = 90
var maturity = "Major" if age > 18 else "Minor"
```

There are restriction on conditional expressions to be aware of when using them.

- Conditional expressions cannot be nested. This means that one cannot use a conditional expression as a `primary_expression` nor as a `condition` nor as an `alternative_expression`.
- The `primary_expression` must be have the same type instance as the `alternative_expression`.
- If the `primary_expression` is a string, list or map, it must have the same length as the `alternative_expression`.

2.7.7 Restrictions on type instance attachment

A type instance cannot be attached to an expression used a key of a map expression.

2.8 Variables

Variable declarations can either be mutable or immutable. Variables must always be initialized. If a type instance is provided on a variable, it must have a complete type instance. Expressions that initialize variables must have a complete type instance or the expression incomplete type instance must match the type instance set on the variable.

2.8.1 Anatomy of a variable

A variable declaration is made declarator that also serves as mutability specifier, a name and optional type instance followed by an equal sign with an initializer expression following the equal sign.

A declarator can be either `val` for immutable variables and `var` for mutable variables.

```
-- a mutable variable initialized to integer 32
var age:int = 32
```

Variable declarations also allow serial initialization. Observe:

```
-- we declare and initialize multiple variables in one go
var a = b = c = d = 10
```

Let's note as well that if two or more variables share the same declarator, they can declared on the same line or indented to reflect the sharing.

```
-- we declare two variables containing qubits on the same line
val source = 0q1, destination = 0q0

-- the above can also be written as follow for clarity where needed
val source = 0q1,
    destination = 0q0
```

The indentation is not necessary in the second example but it helps with readability.

2.8.2 Meaning of the declarator on reference variables

Once a reference is set on a variable, it cannot be changed later. Which gives declarators a different meaning: a variable holding a reference declared with `var` indicates that through it, the variable it points to can be modified (if it was declared mutable).

```
-- A immutable variable
val name = "John Doe"
-- The following will fail since the variable <name> is immutable
var alias = ref name
```

Danger: The restriction above is not yet implemented for function parameters and will be implemented soon. So for the time being, you are on your own in passing correct references to functions.

2.8.3 Restrictions on variable declarations

Certain type instances do not allow their values to be replaced. The `qubit`, `string`, tuple - `()`, list - `[]` and map - `{}` type instances do not allow their values to replaced. Hence, the following code will fail:

```
-- a variable containing a string cannot mutable
var name = "John Doe"
```

2.8.4 Type instance inference

When the expression that initializes a variable has a complete type instance, it is not necessary to supply a type instance either on the expression or on the variable.

```
-- since the expression has a complete type instance, we don't need to specify the␣
↪type instance
var a = Just(10)

-- with this variable though, we need to specify the type instance
-- we specify one on the variable but it can also attached on the expression itself
var b:maybe(int) = None
```

The type instance set the on the variable must be complete and it must match the type instance deduced for the initializer expression.

2.8.5 Restrictions on variables

A variable cannot share the same name with a namespace, a type or function if they appear in the same namespace.

2.9 Functions

Functions are the basic units of code reuse in modern programming languages. Avalon is no exceptions and gives functions with one exciting feature: extended overloading.

The language is introduced as having algebraic data types but unfortunately while it has sum and product types, it lack power types, aka functions. What does this mean? While we can create functions and call them, we can hold them in variables, pass them to value constructors or even to other functions as arguments. At the moment, functions are not first-class objects in the language.

But worry not, functions as they are in the language are already powerful enough to allow you to program in the wild.

2.9.1 Anatomy of a function

A function begins with the function declaration `def` followed by the function name. If the function is to be generic, after the function name, one or multiple type parameters can be provided. Afterwards, the function signature is provided followed by a colon.

Here a skeleton of a function that searches a list of integers for a given needle and returns the index in the list where the needle was found.

```
def search = (val list : [int], val needle : int) -> maybe(int):
    return Just(0)
```

Please note that we provide the mutability specifier (`var` and `val`) to each parameter though it is not mandatory. If a mutability specifier is not provided, the parameter is assumed by default to be immutable. So the declaration above is the exactly the same as below.

```
def search = (list : [int], needle : int) -> maybe(int):
    return Just(0)
```

Now, let's elaborate a little bit on the function's signature. Everything in the parentheses are the function's parameters. A function can also admit an empty list of parameters.

```
def rand = () -> int:
    return 4
```

Above is the best random number generation function that returns an integer but accepts not parameters.

Note: At the risk of being pedantic, type instances are not inferred for function parameters and must always be provided. Therefore a signature of `def search = (list, needle) -> maybe(int)` is not allowed. Obviously, the return type must also be provided for each and every function.

After the function signature and the colon that follows it, **an indentation is expected**. The indentation introduces the function body.

Now, imagive our search algorithm can work with any data that's thrown at it, we can parametrize it with a type constraint so that the compiler can perform replacements of the constratint with complete types later. Observe a generic function:

```
def search : a = (list : [a], needle : a) -> maybe(int):
    return Just(0)
```

Whether we pass a list of integers, strings and so on, our search function is guaranteed to work.

Attention: Note that the type constraint on the function is a single, lower case letter. This is in keeping with the same convention for type declarations where we use single letters for type parameters. Again, the compiler doesn't enforce this convention.

2.9.2 Termination analysis

Unless your function returns `void`, you must always make sure that it returns. The compiler assists mildy in this by perform a reachability and termination analysis but since such analysis cannot be done fully, it is conservative. This means that there will be cases where your function doesn't terminate and the compiler won't breathe a word of it.

2.9.3 Calling functions

Functions are called by writing the function name followed by a comma separated list of arguments enclosed in parentheses, if applicable. If we are to call our search function above, we could so as shown below:

```
var index = search([1, 2, 3, 4, 5], 5)
```

Avalon provides another handy syntax if you have long functions so as to help future maintainers or anyone reading your code. You can prepend arguments with parameter names as shown below:

```
var index = search(
    list    = [1, 2, 3, 4, 5]
    needle  = 5
)
```

Both function calls are equivalent except if you have a long list of parameters, the second syntax is far more readable.

2.9.4 Extended overloading

Sometimes one needs two functions with the same name and same parameters but to return values of different types. This static version of multiple dynamic dispatch is what we call *extended overloading* since it acts not only on the function's parameters but also on the function return type.

This feature is used for the cast operator for instance. Imagine the following for instance: we wish to cast a `int` to both `string` and `float` (this comes out of the box for `int` but the same strategy is applicable for your own types).

Most Avalon operators can be overloaded in the case of the `cast` operator, the corresponding magic function is called `__cast__`. Here is how the signature of the cast function would look:

```
-- function to cast integers to floating point numbers
def __cast__ = (i : int) -> float:
    return 0.0

-- function to cast integers to strings
def __cast__ = (i : int) -> string:
    return ""
```

Our simplistic example is already very useful because without extended overloading it would be impossible to have user defined cast operators. This is not a problem in dynamically typed languages but a problem in statically type programming languages.

To call a function that been overload in this manner, the return type instance must be provided. Observe:

```
-- cast an integer to a string
var str = cast(12) -> string -- we must provide the return type instance else the␣
↪compiler won't know which of the many functions to choose from
```

2.9.5 Magic functions

Most operators can be overloaded, meaning you can use the same operators on your own types. The table below show the list of operators, expected function names, arity and where applicable the expected signature.

Table 2.2: Magic functions

Operator	Operator name	Function name	Arity
+	Unary positive	__pos__	1
-	Negation	__neg__	1
~	Bitwise not	__bnot__	1
+	Plus	__add__	2
-	Minus	__sub__	2
*	Times	__mul__	2
/	Divide	__div__	2
%	Modulus	__mod__	2
**	Power	__pow__	2
<<	Left shift	__lshift__	2
>>	Right shift	__rshift__	2
&	Bitwise and	__band__	2
|	Bitwise or	__bor__	2
^	Bitwise xor	__xor__	2
==	Equal	__eq__	2
!=	Not equal	__ne__	2
>	Greater than	__gt__	2
>=	Greater or equal to	__ge__	2
<	Less than	__lt__	2
<=	Less or equal to	__le__	2
cast	Cast	__cast__	1

In order to enable the use of values from a type to be used as keys in maps, the `__has__` magic method must be implemented. It takes the type of interest as its only parameter and returns an integer.

```
-- implement the <__has__> magic method in order to enable your type's values to be␣
↪used as key in map expressions
def __hash__ = (v : your_type_instance) -> int:
    return 0
```

Note: There are four other magic methods available, `__setitem__`, `__getitem__`, `__setattr__` and `__getattr__` that are meant respectively to set an item using indexing, get an item using indexing, set an attribute using member access and get an attribute using member access. Their current implementation is misguided and is being re-engineered.

Danger: If you look into the source code, you will notice that the compiler has magic functions for logical and, or and not. Please do not rely on them as they planned to be removed before version 1.0.0 considering that two of them are short-circuit operators.

Note: The `__main__` magic function serves the special purpose of being the entry point of the entire application. It is associated with any operators and as best practive, it best never to name your own functions after it.

2.9.6 Restrictions on functions

No function can share the same name, in the same namespace, with a variable or a namespace. A function and a type can shared the same name.

2.10 Control flow

At the moment, avalon provides two means of controlling the flow of a program, namely if the `if` conditional and the `while` loop. In the future, the `switch` conditional and the `for` loop will be added.

2.10.1 Conditional statements

Conditional statements control the flow of the program allowing the program to choose which path the execution is to take depending on the whether a specific condition is fulfilled.

At the moment, only the `if` conditional is implemented. It allows execution branching based on comparison between values and based on whether values match through pattern matching.

If conditional

The `if` conditional is made of the main branch introduced by `if`, optional multiple `elif` branches and an optional `else` branch.

Let us first demonstrate how an `if` statement can be used to perform branching using comparison.

```
import io

-- we create a user global variable
var user = (
    name = "John Doe",
    age  = 32
)

-- the program entry point
def __main__ = (val args : [string]) -> void:
    -- we perform branching depending on the age of the user
    if user.age < 18:
        Io.println(user.name + " is a minor.")
    elif user.age >= 18 and user.age < 65:
        Io.println(user.name + " is an adult in education, employment or training.")
    elif user.age >= 65 and user.age < 120:
        Io.println(user.name + " is a senior retiring or retired.")
    else:
        Io.println(user.name + ", may you live another 120 years!")
```

As mentionned before, the `if` statement can also be used to do pattern matching. We are going to adapt the previous example to one that work with pattern matching using a user defined type.

```
import io

-- our user type
type user = ():
    User(
```

(continues on next page)

(continued from previous page)

```
        name : string,
        age  : int,
        alive: bool
    )

-- the program entry point
def __main__ = (val args : [string]) -> void:
    var u = User(
        name    = "John Doe",
        age     = 32,
        alive   = True
    )

    -- we begin by matching against the user so we get the user details
    if u === User(
        name    = n:string, -- the type instance must occur when capturing values
        age     = a:int,
        alive   = _         -- we use the underscore to let the compiler know that we
↪are not interested in the <alive> field
    ):
        -- now we can use the capture values
        if a < 18:
            Io.println(n + " is a minor.")
        elif a >= 18 and a < 65:
            Io.println(n + " is an adult in education, employment or training.")
        elif a >= 65 and a < 120:
            Io.println(n + " is a senior retiring or retired.")
        else:
            Io.println(n + ", may you live another 120 years!")

    else:
        -- this branch will never execute because the type only has one value
↪constructor and we are matching against it
        Io.println("We didn't get a valid user!")
```

That is pretty much all there is to the `if` conditional statement.

2.10.2 Loop statements

Loop statements allow us to execute the same code multiple times until we decide to stop loop using either a `break` or `return` statement if the condition is not met already.

Only the `while` loop is currently implemented but the `for` loop is in the works as well to allow range based looping.

While loop

A `while` loop allows the looping to continue until the condition is no longer met or the loop is stopped using a `break` or a `return` statement. Pattern matching expressions can also figure as condition to loops and this will be demonstrated with a search example at the end of this section.

For the moment, let us see how to implement FizzBuzz.

```
import io

def __main__ = (val args : [string]) -> void:
```

(continues on next page)

(continued from previous page)

```
    -- the buzz counter
    var buzzer = 1

    -- we keep looping so long as the buzzer is less than 101
    while buzzer < 101:
        -- We print "Fizz" or "Buzz" or "FizzBuzz" or the number depending on our␣
↪divisor
        if buzzer % 15 == 0:
            Io.println("FizzBuzz")
        elif buzzer % 3 == 0:
            Io.println("Fizz")
        elif buzzer % 5 == 0:
            Io.println("Buzz")
        else:
            Io.println(string(buzzer))

        -- we don't forget to increment the buzzer else we end up with infinite loop
        buzzer = buzzer + 1

    -- we end execution
    return
```

2.10.3 Example that combines conditional statements and loops

We are going to implement a generic linear search that uses comparison based conditional and pattern matching looping. The function itself is not complicated but combines different elements of what features in the documentation so if you are having trouble understanding the code, look in the reference.

```
import io

-[
search
    Performs a linear search of the needle inside the given list.

:params
- list      : [a*]
    A generic list of elements to search.
- needle    : a*
    A generic element to search.

:returns
- index     : maybe(a*)
    `Just(i)` where `i` is the index where the needle was found,
    `None` if no element was found.
]-
def search : a = (val list : [a], val needle : a) -> maybe(int):
    -- the current index and the element at that index
    var index   = 0,
        current = list[index]

    -- perform the search
    -- notice how we are using pattern matching in the while loop itself
    while current === Just(value:a):
        if needle == value:
            return Just(index)
```

(continues on next page)

(continued from previous page)

```
        else:
            index = index + 1
            current = list[index]

    -- if we reach here, the needle wasn't found
    return None:maybe(int)


-[
main
    The main entry point.

:params
- args      : [string]
    A list of strings that were passed to the program as commandline arguments.

:returns
- nothing   : void
]-
def __main__ = (val args : [string]) -> void:
    -- search data
    val list   = [1, 2, 3, 4],
        needle = 2

    -- we perform the search
    var result = search(list, needle)

    -- we use pattern matching to see if we found the value and print the index where␣
↪is was found
    if result === Just(index:int):
        Io.println("Found element <" + string(needle) + "> at index <" +␣
↪string(index) + ">.")
    else:
        Io.println("Element <" + string(needle) + "> not found.")

    -- we are done
    return
```

2.11 Modularity

Modularity is the essence of programming and the first line of offense against programs that threaten to become unwieldy. Avalon provides two simple tools to assist you in creating modular programs.

2.11.1 The import statement

The first thing to do is to spread a program into separate files so as to avoid having one huge monolithic program. This is done by creating `packages` which are just files containing Avalon programs and ending with the extension `.avl` like any other Avalon program.

Let us consider that one has created a package localed at `./stdlib/algorithms.avl` then this package can be imported by using the following:

```
-- every public declaration inside the package can be used after a successful import
import stdlib.algorithms
```

2.11.2 Namespaces

Even after we have separated programs into small manageable packages, there is still the threat of name collision. This is resolved by using namespaces. Namespaces allow the same name to be used in two packages without colliding so long as that name happens to be in two different packages. Of course, if the same name is reused twice in the same namespace in two different packages that are imported simultaneously, a collision wil occur. It is thus the responsibility of the programmer to manage their namespace to avoid name collisions.

A namespace is created by the following syntax.

```
-- import statements must figure outisde of a namespace
import io

-- your namespace declarations go here
namespace Ns -/
    -- type declarations come here

    -- global variables come here

    -- function declarations come here
/-

-- you can have two namespaces in the same package
namespace Wh -/

/-
```

Attention: Import statements must occur outside of namespace declarations.

Note: You can have two namespaces in the same package.

Tip: You can choose to indent the declarations inside a namespace (or not).

One might ask what happens when we don't declare a namespace. If no namespace is provided, then all your declarations go into the global namespace which is the namespace without a name.

```
-- the global namespace
-- this is the same as not declaring a namespace at all
namespace -/
    -- your declarations go here
/-
```

Note: The `__main__` function must always occur in the global namespace.

Attention: As a convection, namespace names are in PascalCase. This is not enforced by the compiler but makes for very readable code.

Using declarations inside a namespace

A declaration inside a namespace is used by preprending it with the namespace name. Imagine the `Io` namespace that contain the `println` function. To use this function, one does the following:

```
import io
import math

-[ <namespace>.<declaration> is how a namespaced declaration is used ]-

-- using a namespaced function
Io.println("Hello World!")

-- using a namespaced variable
Math.PI
```

2.12 Encapsulation

Not everything in a package is meant for outside consumption. Avalon provides the declaration access modifiers `public` and `private` to indicate that a declaration inside a package shall not be accessible outside the package.

This comes with a few rules especially for type declarations and function declarations.

Attention: All declarations are public by default. Unlike the commonly accepted convention of making everything private, I have found so far this restriction to be getting in the way while providing little benefit unless the user truly wants a declaration to be private.

2.12.1 Type declarations

A type declaration is declared public or private by prepending the type declarator with the `public` or `private` access modifiers.

```
-- example of a private type
private type trio = ():
    Uno
    | Secundo
    | Tertio
```

The type `trio` in the code above cannot be accessed outside the package within which it is declared. It is not necessary to declare a type public since all declarations, type declarations included, are public by default.

Warning: A type declared private cannot be used with public global variables or in public functions signature.

2.12.2 Variable declarations

A variable declaration is declared public or private by prepending the variable declarator with the `public` or `private` access modifiers.

```
-- example of a private variable
private val PI = 3.14
```

The variable `PI` above cannot be used outside of the package within which it is declared. As with all other declarations, variable declarations are public by default so there is no need to mark them public.

2.12.3 Function declarations

A function declaration is declared public or private by prepending the function declarator with the `public` or `private` access modifiers.

```
-- example of a private function
private def test = (a : trio) -> int:
    return 0
```

Same with other declarations, a function declaration is public by default so there is rarely any need to declare it as public. And to reitarate, a public function cannot use in its signature a type instance from a private type.

2.13 Quantum gates

Avalon currently only supports 1-Qubit gates. Multi-qubits gates can be created from those. Control gates are parametrized by 1-Qubit gates.

2.13.1 1-Qubit gates

Any 1-Qubit unitary gate is of the form:

$$Gate\left(\theta,\phi,\lambda\right) = \begin{pmatrix} e^{-i(\phi+\lambda)}\cos\left(\frac{\theta}{2}\right) & -e^{-i(\phi-\lambda)}\sin\left(\frac{\theta}{2}\right) \\ e^{i(\phi-\lambda)}\sin\left(\frac{\theta}{2}\right) & e^{i(\phi+\lambda)}\cos\left(\frac{\theta}{2}\right) \end{pmatrix}$$

This is the form that Avalon implements directly as $Gate\left(\theta,\phi,\lambda\right)$.

Indeed, `Gate(float, float, float)` is a value constructor that constructs values of type instance `gate` which can be applied to qubits.

As an example, let us show how one creates the Hadamard gate:

```
-- import math since it contains pi
import math

-- and here we have the Hadamard gate
val had_gate:gate = Gate(Math.PI / 2.0, 0.0, Math.PI)
```

And all other 1-Qubits gates are created the same way.

Controlled gates

Controlled gates are parametrized by 1-Qubit gates. All controlled gates are of the form:

$$CGate\,(gate) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & Gate_{00} & Gate_{01} \\ 0 & 0 & Gate_{10} & Gate_{11} \end{pmatrix}$$

Each $Gate_{ij}$ is an element from the `Gate` constructor. The form above is the value constructor `CGate(gate)` and constructs values of type instance `cgate`.

For instance, to construct a controlled Hadamard gate, only simple does the following:

```
-- import math since it contains pi
import math

-- and here we have the Hadamard gate
val had_gate:gate = Gate(Math.PI / 2.0, 0.0, Math.PI)

-- we create a controlled hadamard gate
val had_cgate:cgate = CGate(had_gate)
```

Applying gates to single qubits

Applying gates to qubits is extremely simple. One simply calls the `apply` function, passing it the gate and the qubit(s) to apply the gate to. For 1-Qubit gates, `apply` has the signature `apply(g : gate, q : ref qubit) -> void` and for controlled gates it has the signature `apply(cg : cgate, control : ref qubit, target : ref qubit) -> void`.

Let us demonstrate with an example, reusing our previous code:

```
-- import math since it contains pi
import math

-- and here we have the Hadamard gate
val had_gate = Gate(Math.PI / 2.0, 0.0, Math.PI)

-- we create a controlled hadamard gate
val had_cgate = CGate(had_gate)

-- we create two qubits and we shall apply gates to them
val q1 = 0q0, q2 = 0q1

-- create the |+> state using the hadamard gate
apply(had_gate, ref q1)

-- apply the controlled hadamard gate using q2 as control and q1 as target
apply(had_cgate, ref q2, ref q1)
```

Note: The SDK that comes with the compiler has a few builtin gates that you can find in the table below. So you do not need to create them. Please see the table at the end of this section to see the list of those gates.

Warning: Controlled gates and the swap gate require that their two qubit references arguments not be identical and the SDK will enforce it. Unfortunately, at the moment, the gate will simply not be applied if for instance references to control qubit and the target qubit are identical. No error message will be emitted. This will be corrected in future versions of the SDK.

Measuring single qubits

Once you have applied unitary transformations on your qubit(s), it often desirable to measure them. This also is very easy, just use the `measure` function. On single qubit variables, the measure function returns a value of type instance `bit`.

```
-- initialize q to |0>
val q = 0q0

-- measure it
val b = measure(ref q)
```

Note: You can use the `cast` operator to perform measurement as this is implemented internally for you. It is done as follows: `val b = cast(ref q) -> bit`.

List of standard 1-Qubit gates

Please find below a table of gates that come with the SDK, their names, signatures and example usage. All standard gates live in the `quant` package and are bound to the `Quant` namespaces

Table 2.3: Standard gates

Gate name	Signature	Example
Indentity	id(q : ref qubit) -> void	Quant.id(ref q)
Controlled identity	cid(control : ref qubit, target : ref qubit) -> void	Quant.cid(ref q1, ref q2)
Pauli X	px(q : ref qubit) -> void	Quant.px(ref q)
Controlled X	cx(control : ref qubit, target : ref qubit) -> void	Quant.cx(ref q1, ref q2)
Pauli Y	py(q : ref qubit) -> void	Quant.py(ref q)
Controlled Y	cy(control : ref qubit, target : ref qubit) -> void	Quant.cy(ref q1, ref q2)
Pauli Z	pz(q : ref qubit) -> void	Quant.pz(ref q)
Controlled Z	cz(control : ref qubit, target : ref qubit) -> void	Quant.cz(ref q1, ref q2)
Rotation about X	rx(q : ref qubit, theta : float) -> void	Quant.rx(ref q, Math.PI)
Controlled rotation about X	crx(control : ref qubit, target : ref qubit, val theta : float) -> void	Quant.crx(ref q1, ref q2, 0.0)
Rotation about Y	ry(q : ref qubit, theta : float) -> void	Quant.ry(ref q, Math.PI / 2.0)
Controlled rotation about Y	cry(control : ref qubit, target : ref qubit, val theta : float) -> void	Quant.cry(ref q1, ref q2, Math.PI / 2.0)
Rotation about Z	rz(q : ref qubit, phi : float) -> void	Quant.rz(ref q, 0.0)
Controlled rotation about Z	crz(control : ref qubit, target : ref qubit, val phi : float) -> void	Quant.crz(ref q1, ref q2, Math.PI)
Phase	phase(q : ref qubit, lambda : float) -> void	Quant.phase(ref q, Math.PI / 8.0)
Controlled phase	cphase(control : ref qubit, target : ref qubit, val lambda : float) -> void	Quant.cphase(ref q1, ref q2, Math.PI / 8.0)
S	s(q : ref qubit) -> void	Quant.s(ref q)
Controlled S	cs(control : ref qubit, target : ref qubit) -> void	Quant.cs(ref q1, ref q2)
T	t(q : ref qubit) -> void	Quant.t(ref q)
Controlled T	ct(control : ref qubit, target : ref qubit) -> void	Quant.ct(ref q1, ref q2)
Hadamard	had(q : ref qubit) -> void	Quant.had(ref q)
Controlled hadamard	chad(control : ref qubit, target : ref qubit) -> void	Quant.chad(ref q1, ref q2)
Swap	swap(control : ref qubit, target : ref qubit) -> void	Quant.swap(ref q1, ref q2)