

|  |  |
| --- | --- |
| **DIPARTIMENTO DI INFORMATICA** | |
|  | |
| *Formal Methods in Computer Science* | |
| *Student:* | *Professor:* |
| Miriana **CALVANO**  **mat.761048** | Giovanni **PANI** |
|  |  |

**Introduction**

This is the documentation of a simple interpreter for the IMP language written using Haskell programming language. The interpreter has been implemented by Calvano Miriana for the final exam of the course “Formal Method in Computer Science”.

An interpreter, by definition, is a computer program that directly executes instructions written in a

programming or scripting language without requiring them previously to have been compiled into a machine language program. The interpreter, indeed, usually transforms the high-level program into an intermediate language before executing it.

The types involved in this language are:

* ***Int:*** represents Integer numbers which can have negative or positive values;
* ***Bool***: represents Boolean values that can assume True or False as values;
* ***Float:*** represents Floating number that can be negative or positive and are represented with the exponential notation;
* ***Array:*** represents the Array, a data structure that contains a group of elements of the same type and in this interpreter the array contains integer values.

Moreover, the imperative language gives us the possibilities to use the basic constructs used in the

most common imperative languages, such as

* **assignment:** assigns a value to a variable identifiEd by a univocal identifier;
* **skip**: just does nothing, only performs a jump;
* **if then else**: if the boolean expression after the if it's true, then executes some commands otherwise (else) executes some other ones;
* **while**: executes and continue executing the commands while the boolean condition
* **Declaration:** it is a construct that specifies properties of an identifier. In this case, declarations specify the data type of the variable we’re declaring;

The IMP interpreter uses an **eager evaluation strategy**. This means that, in case of composition between functions, the inner one is executed first and so on until the most external is reached. In order to perform this, the interpreter uses a **call-by-value**.

Finally, This documentation will describe and explain all the parts realized to implement A-LImp language, from the inner representation of the program to the grammar, passing through the development process.

To carry out the whole project, it’s been necessary to implement different components, which are:

* Grammar
* Parser
* Interpreter

**Grammar**

The grammar has been described in the file Grammar.hs and shows the internal representation of the program that will be accepted by the interpreter. In the this file are defined the types, operations and commands with the relative structure.

The BNF of the grammar of this language is specified in the following schema and goes from an high level of abstraction to a lower level. It shows the structure that the internal Representation tree will assume, after the source code is parsed from the Parser.

**Environment**

The environment (the state of the memory) is defined as a dictionary, where the value and the type associated with each name of variable that is declarated (and eventually updated) in the program are stored.

**Arrays**

**Design**

The interpreter consists of two parts: the first part takes one input file as a String and creates an internal representation of the program (the parser) and the second one which takes the output of the first part and evaluates the program updating the state of the memory (the interpreter).

To keep track of the variables and their values during the program execution, it has been created a Dictionary data structure.

We start from the examination of the Parser represented in the file Parser.hs.

**Parser**

The parser contains a function that takes in input a string and gives in output a Maybe couple (a, String) , where a is a parametrized type and the string is the part of the input string that is not parsed yet.

The type a is the type of value returned in case of correctly parsed input.

We use the Maybe just to have the value of Nothing that will mean to us that the parser failed to

parse.

**Functor, Applicative, Monad and Alternative**

**Interpreter**