Language interpreter written in Haskell

Formal Methods in Computer Science
University of Bari
A.Y. 2021-22

Alessio Pagliarulo
a.pagliarulo13@studenti.uniba.it
Identification Number 765842

Contents

Introduction

As first thing, we want to recognize the command that we defined for the IMP language and possibly add an extensions.

Grammar

program ::= <command> | <command> <program>

command ::= <skipcommand> | <assignmentcommand> | <ifcommand> | <whilecommand> | <dowhilecommand> | <forcommand>

skipcommand ::= skip <semicolon>

assignmentcommand ::= <variable> := (<aexp> | <bexp>) <semicolon> | <letter> := { <array> } <semicolon>

ifcommand ::= if <space> (<bexp>) <space> { <space> (<program> | <program> else <space> <program>) } <semicolon>

whilecommand ::= while <space> (<bexp>) <space> { <space> do <space> <program> <space> } <semicolon>

dowhilecommand ::= do <space> { <space> do <space> <program> <space> } while <space> (<bexp>) <space> <semicolon>

forcommand ::= for <space> <variable> <space> times <space> { <space> <program> <space> }

array ::= <afactor> | <afactor> <colon> <array>

bexp ::= <bterm> | <bterm> <bexpOp> <bexp>

bterm ::= <bfactor> | (<bexp>) | ! <bexp>

bfactor ::= <aexp> | <aexp> <comparisonOp> <aexp> | <variable>

aexp ::= <aterm> | <aterm> <aexpOp1> <aexp>

aterm ::= <afactor> | <afactor> <aexpOp2> <aterm>

afactor ::= <apositivefactor> | <anegativefactor>

anegativefactor ::= **-** | <apositivefactor>

apositivefactor ::= <number> | (<aexp>)

number ::= <positivenumber> | <variable>

positivenumber ::= <digit> | <digit> <positivenumber>

variable ::= <letter> | <letter> <variabile>

semicolon ::= ; | ; <space>

colon ::= , | , <space>

digit ::= 0-9

aexpOp1 ::= + | -

aexpOp2 ::= * | /

bexpOp ::= & | |

comparisonOp ::= < | > | = | <= | >= | !=

letter ::= a-z

space ::= " "

Design

Considering the previous defined grammar, the language interpreter designed is structured in such a way:

- a **parsing component** that performs the syntactic check of the language;
- an **evaluation component** that can evaluate the correct written code.

Thus, the interpreter reads the source code provided in input twice. First for checking the syntax code in input, and then for the evaluation that follows these next steps:

1. **Arithmetic expressions** evaluation;
2. **Boolean expressions** evaluation;
3. Assignment of the **values** to the **variables**;
4. Evaluation of the condition and selecting the statements to execute in case of **if-then-else**;
5. Evaluation of condition and iterating the instructions to be executed in the case of **while**, **do-while** and **for**.

When the interpreter evaluates expressions needs to keep in memory the environment used to bind them.

So, **FCKNG** evaluates expressions and for each assignment of values to variables stores values in the environment in correspondence with the relatives variables. If an interpretation of the variable already exists in the environment, this is replaced. Otherwise, if a variable is within an expression, the interpreter will read its value from the environment and will replace it with the value read. The environment represents the portion of memory used by a data program

Evaluation

In this case, the **Eager Evaluation Strategy** has been adopted, classic of IMP languages. Therefore, **FCKNG** will evaluate the expressions to be associated with the variables as soon as it encounters the related assignment statements.

In the case of composition, in which the evaluation of one expression requires the result of another, the interpreter will use **call-by-value**.

Implementation

For the implementation of the environment it has been adopted a list of pairs containing the name of the variable and its value. This environment has been implemented as a type:

type Env = [(String, String)]