C Programming – Project #3: Pseudo-Code Interpreter

Introduction

In computer science, an *interpreter* is a computer program that directly executes, i.e. *performs*, instructions written in a programming or scripting language[1]. The goal of the project is to develop an interpreter for the pseudo-code defined in the first chapter of the course. A brief summary of the specification of the pseudo-code is provided in the appendix.

The project must be prepared in pairs, defended on Friday March 17th and submitted on Campus the same day by 23:55.

Please read this paper thoroughly before you begin the project. If you have any question, please post it in the Q&A forum, in the "Project #3" thread. We will not answer direct emails.

Contents

1 Requirements

The interpreter takes as input a program written in pseudo-code. The program is usually stored in file whose name is provided to the interpreter as a command line argument. If no argument is provided, the

[1] wikipedia

interpreter reads the program from standard input. (This will ease testing.). The execution of the program is directed to standard output.

You will be given sample programs ranging in complexity in order to test your interpreter.

To help you with the development of the interpreter, we suggest the following steps.

1.1 Expressions

The interpreter must evaluate expressions of all types. Below are some examples:

Integer expressions:

```
1
1 - 2
1 + 2 * (3 + 5)
```

String expressions:

```
"hello"
"hello" + " " + "world"
```

Boolean expressions:

```
true
1 = 2
(3 + 4) >= 5
```

At this development stage, the input of the interpreter only consists of expressions, entered one per line. The interpreter prints the result of the evaluation to standard output. Later on (see the Input/Ouput section), the interpreter will only perform output as the result of a `write` statement.

1.2 Variables

The interpreter must handle variables of all types. Variable management includes:

Variable declaration:

The variables of the program are declared in the `var` section of the program. The instructions of the program are enclosed between `begin` and `end`. Here are some examples of variable declarations:

```
var
   integer x, y
   string message, color
   boolean isBlue, isOdd
begin
   …
end
```

Variable substitution:

A variable can be used as an operand in any expression of the same type. Here are some valid expressions using the variables declared above:

```
x
x + 2 * y
"error: " + message + "!"
isBlue and (x = 2)
```

To ease programing, the interpreter automatically converts integer and boolean variables to a string. Thus, the following string expressions are valid:

```
"result: " + x
"the wall is blue: " + isBlue
```

As an extension, the interpreter automatically converts any *expression* of type integer or boolean to a string when needed. (For example, this occurs when assigning a integer expression to a string variable or a string parameter.)

Variable assignment:

A variable can be assigned any expression of its type. Here are some examples, based on the variables declared above:

```
x <- 1
y <- 1 + 2 * y
message <- "hello"
message <- message + " world"
isBlue <- color = "blue"
isOdd <- x % 0 = 1
```

1.3 Input/Output

The interpreter must handle the `write` and `read` statements.

`write` takes a single parameter of type string. The argument passed will be converted to string if needed, as per the above rule. Here are some examples:

```
write(“hello world!")
write("result: " + x)
write("the wall is blue: " + isBlue)
```

`read` takes a single parameter of any type. The value input by the user must match that of the parameter. Here are some examples:

`read(message)`

valid input: any string, including the empty string.

`read(x)`

valid input: any integer constant, like 0, 123, etc.

`read(isOdd)`

valid input: "true" or "false"

1.4 Execution Control

The interpreter must handle the following execution control statements: if, while, do while, and for.

As a first step, you may assume that the body of the statement only contains a sequence of simple instructions, as in the examples bellow:

```
if isBlue then
      isOdd <- false
else
      write(OK)
endif

while x < 10 do
      write(x)
      x = x + 1
endwhile
```

Next, you will consider the general case, where the body of the statement contains nested execution control statements, as in:

```
while x < 10 do
      write(x)
      if isBlue then
            x = x + 1
      else
            x = x + 2
      endif
endwhile
```

2 Bonus Features

Below is a non-exhaustive list of additional features you can implement.

2.1 Error Handling

Your interpreter will be tested against correct programs. However, when presented with a program that contains one or more syntax or semantic errors, your interpreter could display an informative *error message* such as:

```
line 15: keyword “then” expected
```

2.2 Array Type

In addition to primitive types (integer, string, boolean), your interpreter could handle the complex type *array*: array of primitive types, but also multi-dimensional arrays.

2.3 Functions and Procedures

Your interpreter could allow the definition of *procedures* and *functions*, with parameters.

3 Deliverable

The deliverable consists of a single archive named LASTNAME1.FirstName1.LASTNAME2.FirstName2.rar or .zip to identify the project members. The archive contains:

- the report, named report.pdf
- the CodeBlocks project, ***including source files***

The report is between 5 to 10 pages long. It must describe the data structures of the interpreter, the main algorithms, the architecture of the interpreter (i.e the modules it contains and their purpose), your achievements with respect to the requirements and the problems you faced. In appendix, the report must give the screenshot of the execution of each of the test programs you will be given.

The code must be properly formatted and commented. It must comply with the standard naming rules, especially for variables and functions. A special attention will be paid to the architecture of your code: function prototypes, modules, etc.

4 Schedule

The schedule of the project is as follows:

Date	**Step**
February 10th	Release of the project’s paper
February 24th	Follow-up session
March 17th	Defense of the project
March 17th 23:55	Submission of the project on Campus

5 Marking Scheme

The (tentative) marking scheme is as follows:

Item	Marks
Code	10
expressions	*2*
variables	*2*
input/output	*1*
execution control	*3*
quality	*2*
Defense	5
Report	5
Total	20

6 Project Defense

Each project team has 20 minutes to defend its work, as follows:

- During the first 10 minutes, you will briefly introduce the project, describe the organization of the team, expose your achievements with respect to the requirements, and the problems you faced. *Both team members must speak in turn*. Note that you must *prepare* and *rehearse* your speech.

- For the next 10 minutes, you will answer the questions of the examiner. *Each team member must be able to answer any question, on any part of the project.*

Please note that, depending on the quality of their performance, project members may get different marks.

Appendix

Types. The language defines the following types: integer, string and boolean. The standard operators apply:

Integers: + - * / %

Strings: +

Booleans: and or not

Keywords. The keywords of the language are:

Types: `boolean string integer`

Boolean Operators: `and not or`

Program Structure: `begin end var`

Execution Control: `do else if endfor endif endwhile for then to while`

Input / Output: `read write`

Keywords and variable identifiers are case-sensitive.

Execution control statements. The language supports the following execution control constructs:

```
if condition then
   …
[else
   …  ]
endif

while condition do
   …
endwhile

do
   …
while condition

for  variable <- expression to expression [expression] do
   …
endfor
```