Amsterdam University College

Advanced Programming

Assignment 4

The goal of this assignment is to guide you into writing an interpreter for a fragment of a (conceptually improved) version of C.

1 General Description

The language will have the following main characteristics:

- **Paradigm:** The language will be imperative with a clear distinction between expressions and commands, i.e., no command will return a value and no expression will have side effects.
- **Expressions:** The language will provide ways of dealing with basic arithmetical and boolean expressions (including a conditional ternary operator).
- **Commands:** Our language will be block-based, i.e., a program consists of possibly nested blocks of commands and procedures. The main data structure of our language will be variables (no pure names will be allowed). The language will support assignment, conditional commands, while loops, and procedures.
- **Scoping Rule and Parameter Passing:** The scoping rule for procedures should be *static*, and parameter passing should be *by value*.
- **Pointers:** Our language will also support simple dynamical memory management via pointers, see, e.g., this link if you did not program with pointers before. The language will then offer commands to allocate and de-allocate memory, see below for a more precise description of this.

2 Syntax:

In this assignment we will exclusively work with the *abstract syntax* and will avoid dealing with parsing of concrete programs.

The abstract syntax of our language is the following:

$$
\begin{aligned}
Ide &:= String \\
Num &:= Integer \\
Bl &:= True | False \\
Exp &:= Int(Num) \mid Plus(Exp, Exp) \mid Mult(Exp, Exp) \mid Minus(Exp) \\
&\quad Bool(Bl) \mid And(Exp, Exp) \mid Or(Exp, Exp) \mid Not(Exp) \mid Equal(Exp, Exp) \mid \\
&\quad If(Exp, Exp, Exp) \mid Deref(Ide) \mid Val(Ide) \\
Com &:= Assign(Ide, Exp) \mid While(Exp, Com) \mid CIf(Exp, Com, Com) \\
&\quad \mid Procedure(Ide, Listof\ Ide, Listof\ Com) \mid Call(Ide, ListofExp) \\
&\quad \mid NewPointer(Ide, Exp) \mid DestroyPointer(Ide) \\
&\quad \mid UpdatePointerVal(Ide, Exp) \mid Block(Listof\ Decl, Listof\ Com) \\
&\quad \mid Show(Exp) \\
Decl &:= Decl(Ide, Exp)
\end{aligned}
$$

3 Expressions:

Concerning expressions we would like to interpret any operation but conditional as strict operations (i.e., eager policy). As usual conditional should be interpreted as a non-strict operation (i.e., only the relevant expressions are evaluated). The expression $Val(Ide)$ should return the content of the variable with the given identifier (note that, unless val is used as an actual parameter, Ide is not allowed to be the identifier of a pointer, see below pointers are not expressible).

4 Procedures:

Our language offers two syntactic constructs that allow to define and call procedures:

$$Procedure(name, formal\ parameters, body),$$

which allows to define a new procedure with the given name, formal parameters, and body. Then a procedure can be called using:

$$Call(name, actual\ parameters),$$

which allows to call the procedure with the given name and values for the actual parameters.

As mentioned before, the scoping rule for procedures should be static (i.e., a procedure call should be evaluated in the environment and store where the procedure was defined). Parameter passing should be by value (i.e., formal parameters are variables and the values of the actual parameters are *assigned* to the formal parameters). As for expressions, the evaluation policy for procedures should be eager (i.e., all the value of the actual parameters should be computed before the procedure call is evaluated). Finally note that given that our formal parameters are variables, the actual parameter can and must evaluate to storable values.

5 Printing:

In our language we have a special command to print expressible values. The syntax of this command is:

$$Show(Exp),$$

where Exp should be an expression. This command can be used to print the content of variables which are not pointers as follows:

$$Show(Val(\text{"}x\text{"})),$$

where "x" is the name of the variable whose content we want to print. If given $Int(n)$ Show should print n and if given $Bool(b)$ should print b.

6 Pointers:

Our language should implement pointers. Note that pointers in our language will not be the same as locations. Indeed, locations refer to store cells and are therefore allocated and de-allocated automatically, pointers will be addresses of cells stored on a different structure called the *heap*. The heap is where we store user-managed memory. The user is allowed to store in the heap only two types of values (which we call *heapable* values), Integers and Booleans. Our language offers the following two commands to create and destroy pointers:

- $NewPointer(Ide, Exp)$: Create a new variable whose content is going to be a pointer which points to the value of the expression,
- $DestroyPointer(Ide)$: Frees the memory associated to the pointer contained in the variable whose name is the given identifier.

Note that heap-addresses (i.e., pointers) will be stored in variables and cannot be directly accessed in any way.

Your heap should have a fixed size and we would like to use our heap memory efficiently, e.g., if space is freed it should be reusable. [Hint: You may want to keep a list of free cells in your heap.]

Finally to access the value stored in an heap-cell, to access the address of an heap-cell, and to modify the value of an heap-cell we can use the following constructs:

- $Deref(Ide)$: Assumes that Ide is the name of a variable containing a pointer. Returns the value in the cell pointed by the pointer.
- $Val(Ide)$: If Ide is the name of a variable containing a pointer returns the address of the cell pointed by the pointer otherwise returns the value of the variable. Note again that while Val is used both to inspect the content of variables and to get the address of a value in the heap (a pointer), it should not be possible for an expression to return a pointer (i.e., pointers are not expressible nor heapable nor denotable they should only be storable). This means that if x is a variable which contains a pointer (e.g., a variable made using $NewPointer(\text{“}x\text{”}, exp)$) then $Val(\text{“}x\text{”})$ can only appear on the right side of assignments or as actual parameter of a procedure and in no other place.
- $UpdatePointerVal(Ide, Exp)$: Assumes that Ide is the name of a variable containing a pointer. Updates the cell pointed by the pointer with the value of Exp.

7 Testing and Submitting Your Code

As usual you should create your Git repository on Forgejo and work there. Your submission should consist of a file called Interpreter.py which contains the code of your interpreter.

In addition to submitting to Forgejo you should submit your Interpreter.py to CodeGrade for automatic grading. To pass this test, your main interpretation function should have the following signature:

$$interpret(c : Com),$$

where Com is the class of commands. This function should interpret the command c starting with empty environment, empty store, and empty heap.

Finally your program should raise exceptions if the given program does not follow our specifics, e.g., if we try to print a pointer.

[Hint: If you want you can use the code that we presented in class as your starting code, but note that that code is not guaranteed to be efficient nor in the right style (e.g., the code is essentially comment free)].

The following is an example of a program written in our abstract syntax:

```
Block([Decl("y", Int(2))],
      [NewPointer("x", Int(10)),
       NewPointer("w", Int(0)),

       Procedure("p", ["z"],
       [While(Not(Equal(Deref("z"), Deref("x"))),
        Block([Decl("y", Int(1))],
              [UpdatePointerVal("z", Plus(Deref("z"), Val("y"))),
               Show(Deref("z"))]))]),

       Call("p", [Val("w")]),
       Show(Deref("w"))])
```

The output that you should obtain when running it using your interpreter is the following:

```
1
2
3
4
5
6
7
8
9
10
10
```