Assignment 2
Reduction Machine

version 1.3

1. Introduction

This assignment requires you to implement a virtual machine using reduction rules. The virtual machine reads and executes a program written in the prefix format. The program can be written by hand or generated by assignment 1. The program is assumed to be lexically and syntactically right. If there is a semantics error in the program, the virtual machine will print out a corresponding message and stop executing. The machine, otherwise, will execute until it completes the last instruction.

The goals of this assignment are:

- to practise programming in Prolog, a declarative programming language
- to review your knowledge in programming languages such as static type checking, activation record, parameter passing,…,

By completing this assignment, you may find out that some terms such as virtual machine, logic programming, prolog become easy to learn. We hope you feel this assignment interesting.

2. Virtual machine

a. Program structure

The virtual machine executes a program in file '**input.txt**' and prints out the result in file '**output.txt**'. The input program, generated by assignment 1, is a 3-element list followed by a dot and a blank. The first element of the list is the list of global variables, the second one is the list of procedure/function declarations and the last one is the list of instructions. For example,

```
    [[var(a,integer)],[func(foo,[par(a,integer)],integer,[assign(foo,add(a,1))])],
[assign(a,call(foo,[1]))]].
```

Firstly, the machine will check the type of all instructions in the body of all functions/procedures in the second element and all instructions in the last element. If there is an error, a message, described in Section 2.c, will be printed out and the machine will stop immediatedly. Otherwise, the machine executes sequently the instructions in the last element.

An instruction may call a function which must be a built-in one or in the second element. There are 9 built-in procedures: writeInt, writeIntLn, writeReal, writeRealLn, writeBool, writeBoolLn, writeStr, writeStrLn, writeLn and 3 built-in functions: readInt, readReal, readBool. All variable and funtion names in the first and second elements must be distinct and different from the twelve previous names.

b. Instructions:

There are two kinds of instructions: expression and statement. An expression will return a value while the other will not.

Expressions

A unary expression has only one operand while a binary expression has exactly two operands. The first operand of a binary expression is generally evaluated before the second (except for some special cases). Let E_i be an expression.

Numerical expression

sub(E_1)	returns the value of $-E_1$
add(E_1,E_2)	returns E_1+E_2
sub(E_1,E_2)	returns E_1-E_2
times(E_1,E_2)	returns E_1*E_2
rdiv(E_1,E_2)	return E_1/E_2

All Ei must be in **int** or **float** type. The returned value is in **float** type if there is at least one operand in **float** type, otherwise, it is in **int** type.

idiv(E1,E2)	return E_1 div E_2
imod(E1,E2)	return E_1 mod E_2

E_i must be in **int** type and the result is **int** type.

Logical expression

bnot(E_1)	returns the inversion of the Boolean value of E_1
band(E_1,E_2)	returns false if E_1 or E_2 is false, true otherwise.
bor(E_1,E_2)	returns true if E_1 or E_2 is true, false otherwise.

All operands of a logical expression must be in **boolean** type. The result of a logic expression is also in **boolean** type. The **band** and **bor** instructions are short-circuit evaluated. If the first operand of an **band** instruction is evaluated to **false**, the second one is not evaluated. In this case, no error message is issued even though the second operand is not in **boolean** type. For **bor** instructions, the second operand is not evaluated if the first one is **true**.

Relational expression

greater(E_1,E_2)	returns $E_1>E_2$
less(E_1,E_2)	returns $E_1<E_2$
ge(E_1,E_2)	returns $E_1\geq E_2$.
le(E_1,E_2)	returns $E_1\leq E_2$.
ne(E_1,E_2)	returns true if E_1 is not equal to E_2, false otherwise.
eql(E_1,E_2)	returns true if E_1 is equal to E_2, false otherwise.

All realtional expressions return **boolean** values. An **eql** or **ne** instruction requires both its operands in the same type which can be **int** or **boolean**. For greater-, ge-, less- and le-instructions, their operands can be in **int** or **float** type.

Index expression

ele(E1,E2) returns an element of array E_1. The index of the element is given in the list E2. E_1 must be in array type and all elements of the list E_2 must be in **int** type. If E1 is a k-dimensional array, E2 must contain k elements, otherwise, the error message 'Type mismatch' is printed out. The value of an element in E_2 must be greater than or equal to 0 and less than the size of the corresponding dimension of array E_1, otherwise, the machine will print out the error message 'Index out of bound:'.

Function-call expression

call(I,L) where I is the name of the called function and L is the list of argument expressions. The called function must be a built-in one or be declared in the second element of the program as a function,i.e, func(I,…). The machine, otherwise, will print out the error message 'Undeclared function:'. The number of arguments and that of parameters must be the same. Otherwise, the error message 'Wrong number of arguments: ' is printed out. The parameter passing mechanism is call-by-value, that means all arguments are evaluated and their values are copied to the correponding parameters before the called function executes. The machine evaluates the arguments in the left-to-right order. For example,

```
In:     [[var(a,integer)],[func(foo,[par(a,integer),par(b,integer)],
                                    [assign(a,add(a,b)),assign(foo,a)])],
        [assign(a,3),call(writeIntLn,[call(foo,[a,3])]),call(writeIntLn,[a])]].
Out:    6
        3
```

Variable expression

I returns the value assigned to I. The mechanism to determining I is similar to that in C. The determined variable I must be assigned a value before it is used, otherwise, the machine will show the error message 'Undefined variable: '.

Constant expression

integer constant
float constant
true
false
array(L) where L is the comma-separated list of elements
str(E) where E is a string constant which is described in assignment 1.

Input/Output

The machine gets the input value when it executes the expression: call(readInt,[]) or call(readReal,[]) or call(readBool,[]). The valid input value must be the corresponding constants described in the previous section, otherwise, a type mismatch error message is issued. The machine prints out the value of E to file 'output' when the statement call(writeXXX,[E]) or call(writeXXXLn,[E]), or … is executed. The type of E must be in the right type, otherwise, a 'Type mismatch' is throwed out. For example,

```
In:     [[var(a,integer)],[],[assign(a,3),call(writeIntLn,[a])]].
Out:    3
```

Statements

Assignment statement

assign(I,E1) assigns the value of E_1 to I.

The identifier I must be a variable declared as a global or local variable or a parameter or the name of the function containing the assignment statement. Otherwise, the error message 'Undeclared identifier:' is issued.

Compound statement

L where L is the list of statements. The machine will execute sequently the statements in L from the first to the last one (except for some special statements).

If statement

if(E,S1,S2) or if(E,S1) where E must be a **boolean** expression, otherwise, a type mismatch error message will be printed out. If the value of E is true, S_1 will be executed, otherwise, S_2, if any, will be done.

While statement

while(E,S) where E is a loop condition which must be a **boolean** expression, otherwise a type mismatch message is printed out, and S is a loop body. The semantics of this statement is similar to the *while* statement in C.

Do statement

do(L,E) where E is a loop condition which must be a **boolean** expression, otherwise a type mismatch message is printed out, and L is the list of statements. The semantics of this statement is similar to the *do while* statement in C.

Loop statement

loop(E,S) where E is an **integer** expression, and S is the body. The loop statement executes S multiple times that is the value of E in the beginning.

Return statement

There is no return statement in Crazy. When a function terminates, the value of the function name, which is previously assigned by an assignment, is used as the return value. For example,

In: [[],[func(foo,[],integer,[assign(foo,3)])],[call(writeIntLn,[call(foo,[])])]].

Out: 3

Note that for simplicity, YOU MAY ASSUME THAT THERE IS NO LOCAL VARIABLE WHOSE NAME IS THE SAME AS THE NAME OF THE ENCLOSING FUNCTION.

Break statement

break(null) The statement must be enclosed in a loop statement, that is **while, do** or **loop** one. Otherwise, a break-not-in-loop error message will be printed out. When this statement is executed, the machine will terminate the innest loop statement. The execution will continue with the statement that follows the loop one, if any.

Continue statement

continue(null) The statement must be enclosed in a loop statement. Otherwise, a continue-not-in-loop error message will be printed out. When this statement is executed in a **while** or **do** or **loop** statement, the machine will ignore the remaining statements in the body of the loop statement and continue with the condition expression.

Procedure-call statement

call(I,L) where I is the name of the called procedure and L is the list of argument expressions. The called proceduremust be a built-in one or be declared in the second element of the program as a procedure,i.e, proc(I,…). The machine, otherwise, will print out the error message 'Undeclared procedure:'. The requirements in parameters are the same as those in function-call.

c. *Static error messages*

The following errors may occur when the machine does the type checking before it executes the instructions.

Type mismatch

Message: 'Type mismatch: E' where E is an expression or statement in which a type mismatch happens. For example,

```
In:     [[],[],[call(writeInt,[add(10,true)])]].
Out:    Type mismatch: add(10,true)
In:     [[],[],[call(writeBool,[add(10,3)])]].
Out:    Type mismatch: call(writeBool,[add(10,3)])
In:     [[var(a,integer)],[],[assign(a,3),if(a,call(writeInt,[3]))]].
Out:    Type mismatch: if(a,call(writeInt,[3]))
In:     [[],[func(foo,[],integer,[assign(foo,3)])],
                        [call(writeInt,[ele(call(foo,[]),[2])])]].
Out:    Type mismatch: ele(call(foo,[]),[2])
```

Undeclared identifier

Message: 'Undeclared identifier: E' where E is an undeclared variable or constant. For example,

```
In:     [[var(a,integer)],[],[assign(a,add(b,1)),call(writeIntLn,[a])]].
Out:    Undeclared identifier: b
In:     [[var(a,integer)],[proc(foo,[],[var(b,integer),assign(b,1),assign(a,b)])],
                                          [call(writeIntLn,[b])]].
Out:    Undeclared identifier: b
```

Wrong number of arguments

Message: 'Wrong number of arguments: E' where E is a function call expression whose number of arguments is not equal to that of parameters. For example,

```
In:     [[],[proc(foo,[],[])],[call(foo,[1])].
Out:    Wrong number of arguments: call(foo,[1])
In:     [[],[proc(foo,[par(a,integer)],[])],[call(foo,[])]].
```

Out: Wrong number of arguments: call(foo,[])

Redeclared identifier

Message: 'Redeclared identifier: E' where E is the redeclared variable/constant. A global variable/constant is redeclared if it is one of built-in function/procedure names, or there is another previously declared global variable/constant whose name is the same. A parameter is redeclared if its name has been used for another parameter in the same function/procedure. A local variable/constant is redeclared if, in the same block, there is another local variable/constant whose name is the same. For example,

```
In:     [[var(a,integer),var(b,real),var(a,boolean)],[],[]].
Out:    Redeclared identifer: var(a,boolean)
In:     [[],[proc(foo,[par(a,integer),par(b,integer),par(a,real)],[],[])],[]].
Out:    Redeclared identifier: par(a,real)
In:     [[],[proc(foo,[],[const(a,7),var(a,real)])],[]].
Out:    Redeclared identifier: var(a,real)
In:     [[],[proc(foo,[],[var(a,real),[var(a,integer)]])],[call(writeIntLn,[3])]].
Out:    3 //Ok because the first a and the second a are in different scopes
```

Redeclared function/procedure

Message: 'Redeclared function/procedure: E' where E is the redeclared function/procedure name, respectively. A function/procedure is redeclared if its name has been declared as the name of another previously declared function/procedure or a global variable/constant. For example,

```
In:     [[],[func(readInt,[],real,[])],[]].
Out:    Redeclared function: readInt
In:     [[],[func(foo,[],integer,[]),proc(foo,[a],[])],[]].
Out:    Redeclared procedure: foo
In:     [[var(foo,integer)],[proc(foo,[a],[])],[]].
Out:    Redeclared procedure: foo
```

Undeclared function

Message: 'Undeclared function: E' where E is a function call expression whose callee is not built-in function and is not found in the second element of the program as a function. For example,

```
In:     [[var(a,integer)],[],[assign(a,call(foo,[]))]].
Out:    Undeclared function: call(foo,[])
```

Undeclared procedure

Message: 'Undeclared procedure: S' where S is a procedure call statement whose callee has been declared. For example,

```
In:     [[],[],[call(foo,[])]].
Out:    Undeclared procedure: call(foo,[])
```

Break not in a loop

Message: 'Break not in a loop: break(null)'. This message is printed out when a break statement is executed but it is not enclosed in a loop statement. For example,

```
In:     [[],[],[break(null)]].
```

Out: Break not in a loop: break(null)

In: [[],[func(foo,[],[],[break(null)])],[do(call(foo,[]),true)]].

Out: Break not in a loop: break(null)

Continue not in a loop

Message: 'Continue not in a loop: continue(null)'. The condition for this error message is similar to that of the above error message except that it is applied to the **continue** statement instead of the **break** one.

Cannot assign to a constant

This message will be printed out when the left handside of an assignment is a constant identifier. For example

In: [[const(a,7),[],[assign(a,8)]].

Out: Cannot assign to a constant: assign(a,8)

d. Runtime error messages

The following errors may happen during the execution of the instructions.

Index out of bound

Message: 'Index out of bound: E' where E is an index expression. For example,

In:

[[var(a,arr([2],integer))],[],[assign(a,array[1,2]),call(writeIntLn,[ele(a,[-1])])]].

Out: Index out of bound: ele(a,[-1])

In:

[[var(a,arr([2],integer))],[],[assign(a,array[1,2]),call(writeIntLn,[ele(a,[2])])]].

Out: Index out of bound: ele(a,[2])

Invalid expression

Message: 'Invalid expression: E' where E is an expression whose value is undef. For example,

In: [[var(a,integer)],[func(foo,[],integer,[])],[assign(a,call(foo,[]))]].

Out: Invalid expression: call(foo,[])

In: [[var(a,integer),[],[call(writeInt,[a])]].

Out: Invalid expression: a

3. Initialization

To initialize this assignment, you

- Download initial code of assignment 2 and upzip it.
- Download SWI-Prolog and make sure that **swipl** is in your path
- Install swiplserver by typing “pip install swiplserver”
- Move to folder assignment2-initial/src and typing “python run.py test VMSuite”. The result should be ok (pass 3 test-cases and fail 1).

Then, you are free to modify file assignment2-initial/src/main/crazy/vm/ass2.pl.

You MAY assume that there is ONLY ONE error in an input program, if any.

4. Submission, Late penalty and Plagiarism

You are required to submit just file **ass2.pl** and **VMSuite.py** containing 100 test-cases. You must make sure that file **ass2.pl** contains the entry rule **go**.

You must complete this assignment **by yourself**, otherwise you will get **0 for this subject.**