Assignment
Reduction Machine
version 1.1

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 is assumed to be lexically and syntactically right. If there is a semantics error in the program, the virtual machine will raise a corresponding exception and stop executing. The machine, otherwise, will execute until it completes the last instruction.

The goals of this assignment are:

- to practise logic programming.

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 is a 3-element list followed by a dot. The first element of the list is the list of global variable/constant declarations, the second one is the list of procedure/function declarations and the last one is the list of statements. For example,

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

The machine executes sequently the instructions in the last element using the declarations in the first and second elements. During the execution, if there is an error, an exception, described in Section 2.e, will be raised and the machine will stop immediatedly.

A statement may call a procedure 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/constant and function/procedure names in the first and second elements must be distinct and different from the twelve previous names. This requirement must be checked before executing the last element.

b. Declarations:

There are five kinds of declarations: variable, constant, parameter, function and procedure declaration.

Variable declaration

`var(I,type)` where I is a variable, and type is one of integer, float, boolean or string

Constant declaration

const(I,E) where I is a constant and E is a constant which is described in constant expression of Section 2.c

Parameter declaration

par(I,type) where I is a parameter and type is one of integer, float, boolean or string.

Function declaration

func(I,L,type,IL) where I is the function name; L is the list of parameter declaration; type is one of integer, float, boolean or string and IL is the list of statements.

Procedure declaration

Proc(I,L,IL) where I is the procedure name; L is the list of parameter declaration and IL is the list of statements.

c. *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.

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 raise exception 'undeclare_function'. The number of arguments and that of parameters must be the same. Otherwise, the exception 'wrong_number_of_argument' is thrown. 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 which I is declared is similar to that in C. The determined variable I must be assigned a value before it is used, otherwise, the machine will throw exception 'undefined_variable'.

Constant expression

integer constant
float constant
true
false
string constant which is a sequence of character enclosed by double quotes.

Input Expression

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' exception is thrown.

d. Statements:

Let S, E, and L represent a statement, an expression, and a list, respectively.

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 exception 'undeclare_identifier' is issued.

Compound statement

block(L_1,L_2) where L_1 is the list of variable/constant declaration and L_2 is the list of statements. The machine will check if there is any re-declaration in L_1 and execute sequently the statements in L_2 from the first to the last one (except for some special statements). The declarations in L_1 create a local environment for statements in L_2. The block-structure rules are applied for this language.

If statement

if(E,S_1,S_2) or if(E,S_1) where E must be a **boolean** expression, otherwise, a 'type_mismatch' exception will be thrown. 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' exception is raised, and S is a loop body which is a statement. 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' exception is raised, 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 which is a statement. The loop statement executes S multiple times that is the value of E in the beginning.

Return statement

There is no return statement. 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' exception will be raised. 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 exception will be raised. 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 procedure must 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 raise exception 'undeclare_procedure'. If I is one of 9 built-in procedure, the machine prints out the value of the element in the argument list to file 'output'. The requirements in parameters are the same as those in function-call.

For example,

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

Out: 3

e. *Error exceptions:*

The following errors exception may occur when the machine execute the program.

Redeclared identifier

Exception: 'redeclare_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

Exception 'redeclare_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

Type mismatch

Exception '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,[],float,[assign(foo,3.0)])],
[call(writeInt,[call(foo,[])])]].
Out: Type mismatch: call(writeInt,[call(foo,[])])

Undeclared identifier

Exception 'undeclare_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

Exception '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,[])

Invalid expression

Exception '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

Undeclared function

Exception 'undeclare_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

Exception 'undeclare_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

Exception 'break_not_in_loop(E)' where E is a break statement which 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

Exception 'continue_not_in_loop(E)'. The condition for this exception is similar to that of the above error except that it is applied to the **continue** statement instead of the **break** one.

Cannot assign to a constant

Exception 'cannot_assign(E)' where E is an assignment whose left handside is a constant identifier. For example

In: [[const(a,7),[],[assign(a,8)]].
Out: Cannot assign to a constant: assign(a,8)

3. Initialization

To initialize this assignment, you

- Download initial code of the assignment 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 assignment-initial/src and typing "python run.py test VMSuite". The result should be ok.

Then, you are free to modify file assignment-initial/src/main/crazy/vm/vm.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 only file **vm.pl**. You must make sure that your submission file name is exactly **vm.pl** containing the entry rule **go**.

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

5. Change Log:

- From 1.0:
    i. Fix the last example of type mismatch