MiniLang: assignment instructions

Grao en Intelixencia Artificia, Lógica

February 27, 2024

Introduction

In this assignment, we will build a Prolog program that accepts an imperative program and executes it. To that aim, we will define the following predicate,

```
run(PrevState, Program, PosState)
```

where `PosState` is the state of the program after running `Program` starting from a previous state `PrevState`. The state of a program is a list of pairs `(Variable, Value)`, recording the values of the variables used in the program. For instance, the following state,

```
[(x, 5), (y, 7)]
```

means that we have two variables `x` and `y`, which have values of 5 and 7 respectively. The goal of the `run/3` predicate is to update the variables in `PrevState` according to the sentences in `Program`, producing a new state `PosState`. During this *execution*, variable values may change and new variables may be created.

The rest of this document defines the syntax of *MiniLang*, the programming language we will interpret, and provides examples of running programs.

Programming Language

The accepted programs will be written in the *MiniLang* language, a simplified programming language that allows **variable assignments**, **arithmetic expressions** (including its evaluation) and a **printing operation** that outputs messages through default output, as well as flow control via "**if then else**" and "**while do**" constructs.

A *MiniLang* program is a **sequence of sentences of different types**, separated by '`;`'. Next, we define the different types of sentences.

Variable assignment

A **variable assignment** is an expression of the form

```
<variable> = <arithmetic-expression>
```

where `<variable>` is a term starting with a lowercase letter that acts as the identifier of a variable, whereas `<arithmetic-expression>` is a (Prolog-like) arithmetic expression that may include **variables or constants**, as those defined in the class. This is a variable, a constant, or any subsequent application of operators, constants and variables. Below we provide some examples of valid variable assignments:

```
x = 3;
y = x + 2;
varz = ((x + 2)*2 // y) ** 4;
```

Note that the value being assigned to the variable in an assignment is the result of evaluating the expression and not the expression itself.

After calling `run/3` for executing an assignment `var = exp`, the value of the variable `var` is updated in `PosState` according to `PrevState` and expression `exp`. If the `var` does not exist in `PrevState`, then it is created in `PosState`. As examples, take the results after running the following queries:

```
?- run([], x=3, S).                                          →   S = [(x, 3)]
?- run([(x,2), (y,3)], x=3; y=y-1, S).                       →   S = [(x, 3), (y, 2)]
?- run([(varz,6)], x=3; y=x+2; varz=((x+2)*2 // y) ** 4, S).  →   S = [(y, 5), (x, 3), (varz, 16)]
```

Note that interpreting any expression including a variable that was not mentioned before in the program should lead to an execution error.

`print` operation

The language admits a `print` operation for printing *strings* or the result of evaluating *expressions* through default output. Syntactically, it will be used by writing the reserved keyword '`print`' followed by a single argument within parentheses as below

```
print([<string>|<arithmetic-expression>])
```

where `<string>` represents a text string enclosed in double quotation marks (as in Prolog), and `<arithmetic-expression>` represents any arithmetic expression including operators, variables and constants as discussed in the previous section.

The execution of a `print` operation does not update the state of the program but writes values in the terminal. Consider the following examples:

```
?- run([(x,2),(y,3)], print(x), S).
2
S = [(x, 2), (y, 3)]

?- run([(x,2),(y,3)], print("The result of (x+y)**2 is: "); print((x+y)**2), S).
The result of (x+y)**2 is: 25
S = [(x, 2), (y, 3)]
```

Flow control: "`if then else`" and "`while do`"

The syntax of *MiniLang* includes the use of "`if then else`" and "`while do`" constructs to control the flow of the program depending on the evaluation of *boolean comparison expressions*. A boolean comparison expression involves the use of a (Prolog-accepted) comparison operator such as

```
comparison-op := > | < | >= | =< | =:= | =\=
```

and it is an expression of the form:

```
condition := <arithmetic-expression> <comparison-op> <arithmetic-expression>
```

where `<comparison-op>` is one of the operators defined above, and `<arithmetic-expression>` follows its correspondent definition given in the previous sections

The syntax of an "`if then else`" construct is as follows:

```
if <condition> then (<block>) [else (<block>)]
```

where `<block>` is a sequence of *MiniLang* sentences separated by '`;`' enclosed between parentheses and both the word `else` and its corresponding block are optional. If the condition is evaluated as `true` the flow of the program must follow the `then` block. In contrast, the flow must follow the `else` block if it is provided and the condition is evaluated as `false`. As an illustration, consider the following examples:

```
?- run([(e, 19)],
    if e>=18 then (print("Es mayor de edad. Edad = ");
    print(e) ), S).
Es mayor de edad. Edad = 19
S = [(e, 19)]

?- run([(n, 33)],
    if n mod 2 = 0 then (print("n es par "))
    else (print("n es impar")),
    S).
n es impar
S = [(n, 33)]

?- run([(d, 0), (t, 65)],
    if t > 50 then (d = 0.2; print("Descuento del 20%\n"));
    print("Precio final: ");
    print(t - (t*d)),
    S).
Descuento del 20%
Precio final: 52.0
S = [(d, 0.2), (t, 65)]
```

Similarly, the syntax of a "`while do`" construct is as follows:

```
while <condition> do (<block>)
```

where the both the syntax of `<condition>` and `<block>` is it was previously explained. In the case of "`while do`" constructs, the block of code must be executed in a loop while the condition is evaluated as true. Take the following queries as examples:

```
?- run([],(x=1; while x<11 do (print(x);print(" ");x=x+1) ),S).
1 2 3 4 5 6 7 8 9 10
S = [(x, 11)]

?- run([(n, 5)],
    result = 1;
    count = n;
    while count > 1 do
        (result = result*count; count = count-1);
    print("factorial(5)=");
    print(result),
    S).
factorial(5)=120
S = [(count, 1), (result, 120), (n, 5)]
```

Executing programs from files

For convenience, we will provide you with a predicate for reading programs from files. The programs must be written as a Prolog `term`, this is they must follow the syntax described above and end with a dot '.'. We will use the following predicate and its correspondent definition below:

```
run_from_file(InitialState, FileName, FinalState) :-
    open(FileName, read, Stream),
    read_term(Stream, Prog, []),
    close(Stream),
    run(InitialState, Prog, FinalState).
```

For instance, assuming that we have the following contents in a file named *factorial.minilang*:

Listing 1: *factorial.minilang*

```
result = 1;
count = n;
while count > 1 do (
    result = result*count;
    count = count-1
);
print("factorial("); print(n); print(")= ");
print(result).
```

Then this is the result for the following query:

```
?- run_from_file([(n,5)], 'factorial.minilang', S).
factorial(5)= 120
S = [(count, 1), (result, 120), (n, 5)]
```

Important Notes

- Arithmetic and comparison operators are defined by Prolog by default, we do not need to define new operators for this. The ';' operator is defined in Prolog as well, **please do not provide any new definition for this operator** as it could lead to a priority clash error.
- When including more than one sentence in a `then`, `else` or `do` block, we must enclose them parentheses. Otherwise, it will cause a priority clash error.
- If an undefined variable (is not present in the current state) is used in an expression, the `run/3` predicate must say `false`.