SER 502 - Team 20

DReaM Language

Team members:

Shantanu O.
Maaz A.
Janardhan B.
Ashik S.
Michael M.

April 29, 2022

Introduction
by Shantanu G. Ojha

Introduction

- The design of DReaM language is a mix of database procedural language and imperative programming language.
- The main rationale for choosing this language design is to test out the applicability of described/explicit database language with abstracted imperative language like c++ and java.
- Different stages of the language constructions are
    - Lexer : DReaM uses java to generate tokens
    - Parser : Definite Clause Grammar (DCG)
    - Interpreter : Prolog is used to parse the DCG generated parse tree
    - Execution : Prolog is again used to evaluate and execute the program

Design

- DReaM is case insensitive programming language - lending the property from database procedural language.
- 3 datatypes exist for DReaM
  - **string** : any of the 26 english characters or their combinations.
  - **numbers** : includes positive and negative integers and floating point numbers.
  - **boolean** : takes either true or false value.
- The program can be written in any text editor or DReaM IDE and can be executed from the command line.
- DReaM programs use **.dm* as a file extension to differentiate them from other files.

Language constructs

- **Variable**
  - DReaM is strongly typed language and every variable needs a datatype - lending the property from imperative languages like, c++ and java.
  - Variables also needs to be declared first in order to use them - again lending from imperative languages.
  - Variable name can be any of the 26 english alphabets or their combinations and can be declared anywhere in the program.
  - Unlike imperative languages, DReaM variables have no scope and once they are declared they can be used anywhere in the program.
  - DReam also supports inline declarations and individual declarations.

Language constructs(cont'd)

- Example of variable declaration:

```
% Inline declaration %
x string, y number, z boolean;

% individual declaration %
a string;
b number;
c boolean;
```

Language constructs(cont'd)

- DReaM have two ways of writing a program.
- **Block**
  - Uses a start and stop keyword and between is a body of statements/expressions
  - A block doesn't return a value instead prints the environment variable along with the values at the end of the program.

```
start

    Print "Hello world";

stop
```

Language constructs(cont'd)

- **Function**
  - Start with function keyword and with an empty or list of arguments.
  - Must have a return value.
  - A value can be: a string literal, variable, or a number.
  - Unlike block, functions only print the returned value.

```
function ()
start
        x number, y number, s number;
        x = -1; y = 1;
        s = x + y;
        return s;
stop
```

Language constructs(cont'd)

- **Notes**:
  - Statements in DReaM end with semicolon (;)
  - Present symbols (%...%) are used to signify single or multiple line comments in DReaM code.
  - In variable declarations :
    - string variables initialize to an empty string.
    - number variables initialize to 0.
    - Boolean variables initialized to false.

DEMO

(Block.dm)

(function.dm)

Operators
by Janardhan Reddy B

Operators

- **Mathematical Operators**
  - DReaM uses PEMDAS (Parentheses, Exponents, Multiplication/ Division, Addition/Subtraction) precedence rules for operations.
  - At the current version of DReaM doesn't support exponents.
  - The operations supported by DReaM are
    - Addition (+)
    - Subtraction (-)
    - Multiplication (*)
    - Division (/) - here it’s an integer division

Operators(cont'd)

- Modules (@) - the remainder of two numbers
- Increment operator (++)
- Decrement operator (--)

- Note : Unlike imperative languages, the position of the increment operator does influence the precedence of the assignment.
- DReaM has syntactic sugars for the four arithmetic operations with the same semantics as the imperative languages have.
  - x += y; x*=y; x-=y; x+=y

Operators(cont'd)

- **String Operators**
  - Dream provides a concatenation operator (#) that can be used to join two or more strings together.
  - A string literal, a variable value, or a number can be concatenated together.
  - The concatenation operator assigns the concatenated value to a variable.

DEMO

(arithmetic_operations.dm)

(increment_decrement_modules_operations.dm)

(string_operation.dm)

Operators(cont'd)
by Ashik Elahi S

Operators(cont'd)

- **Relational Operators**
  - Dream provides the following relational operators over numeric values.
    - Equal (==)
    - Greater than (>)
    - Greater than or equal to (>=)
    - Less than (<)
    - Less than or equal to (=<)
    - Not equal to (!=)

Operators(cont'd)

- **Boolean operators**
  - Boolean operators in DReaM are applied to boolean expressions and implicitly return a boolean value.
    - And (&&)
    - Or (||)
    - Not (!)
    - XoR (^)

DEMO

(relational_operations.dm)

(boolean_operations.dm)

Loops
by Maaz Ahmed

Loops

- DReaM language supports 3 kinds of loops which all taken from imperative languages.
- Each loop body block starts with start/stop keyword and is delimited by a semicolon.
- DReaM supports nesting of loops within loops.

Loops(cont'd)

- **for loop**
  - Uses an already instantiated variable to iterate over a block of statements.
  - Example:

```
start
    x number;
    x = 5;
    for (x; x < 10; x++)
    start
        println x;
    stop;
Stop
```

- Note: `println` prints every result in new line unlike `print`.

Loops(cont'd)

- **While loop**
  - DReaM supports the same kind of while loop adopted by imperative programming languages.
  - If a variable is used as a condition, it must be instantiated to the correct value.
  - Example

```
start
    x number;
    x = 5;
    while (x < 10)
    start
        x = x + 1;
        println x;
    stop;
stop
```

Loops(cont'd)

- **Range loop**
  - It is the mini-version of for loop which loops between ranges of numbers, Inclusive!
  - As with the other loops a variable must be initialized before being used as a counter in range loop.
  - Example:

```
start
     x number;
     x = 5;
     for x in range(5..10)
     start
          println x;
     stop;
Stop
```

- Note: range values should be numbers, not numeric variables

DEMO

(loops.dm)

&

(factorial_numbers_example.dm)

Conditions
by Michael M.

Conditions

- DReaM language supports 3 kinds of conditional statements which again all taken from imperative languages.
- Each conditional body block starts with start/stop keyword and is delimited by a semicolon.
- DReaM supports nesting of conditions within conditions.

Conditions(cont'd)

- If condition
  - The if condition is similar to the imperative languages conditions where it can exist with else statement or by it self.
  - As with the loops, a variable should be instantiated before it can be used in the if condition.
  - Example

```
if (1 == 1)
    start
        println "1 is equal to itself";
    stop;
```

Conditions(cont'd)

- If-else example

```
x number, y number;
if (x == y)
     start
          println "if else works";
     stop;
else
     start
          println "if else is broken";
     stop;
```

Conditions(cont'd)

- Ternary condition
  - It's a the compact representation of if-else statement with a single return condition.
  - DReaM ternary operator returns a value and should be assigned to a of same datatype as the return condition.
  - Example

```
x number, y number;
output string;
output = x == y ? "working" : "broken";
Output = “ternary operator is” # output;
println output;
```

DEMO

(conditional_statements.dm)

&

(prime_numbers_example.dm)

Snapshots

Arithmetic operations

DReaM Development Environment (1_arithmetic_operations.dm)

New | Open... | Save... | Compile | Run | About

```
1  %
2  Arithmetic operations
3  %
4
5  function(x number, y number, z number)
6  start
7       value number;
8       x = 1; y = 2; z = 3;
9
10      % simple addition %
11      value = x + y + z;
12      if (value == 6)
13      start
14           print "sum is";
15           println value;
16      stop;
17
18      % PEMDAS (Parentheses, Exponents, Multiplication/ Division, Addition/Subtraction) %
19      % Testing Parentheses%
20      value = x*(y+z);
21      if (value == 5)
22      start
23           println "Parentheses got precedence";
24      stop;
25      else
26      start
27           println "Incorrect operation";
28      stop;
29
30      % Multiplication/ Division, Addition/Subtraction %
31      value = x / y * z + x + x/y - x;
32      if (value == 0)
33      start
34           println "Correct precedence observed";
35      stop;
36      else
37      start
38           println "Incorrect precedence";
39      stop;
40      return "Arithmetic testing is completed";
41 stop
```

Output

```
sum is 6
parentheses got precedence
correct precedence observed

arithmetic testing is completed

------------------
Process finished.
Running time : 59 ms
```

String operations

DReaM Development Environment (2_string_operation.dm)

New | Open... | Save... | Compile | Run | About

```
1  %
2  String concatenation
3  %
4
5  start
6       x string, y string;
7       x = "hello ";
8       y = "world";
9       x = x # y;
10      println x;
11
12      % multiple concat%
13      x = "repeat";
14      y = "me";
15      x = x # y # x # y # "and I finished repeating";
16      println x;
17 stop
18
```

Output

```
hello world
repeat me repeat me and i finished repeating

[(x,repeat me repeat me and i finished repeating ,string),(y,me ,string)]

------------------
Process finished.
Running time : 59 ms
```

Relational operators

DReaM Development Environment (3_relational_operations.dm)

New | Open... | Save... | Compile | Run | About

```
%
Relational operators
%

function(x number, y number, z number)
start
     value number;
     result string;
     x = 1; y = 2; z = 3;
     value = x + y;

     % equal operation %
     result = value == z ? "equal operation works" : "equal operation broken";
     println result;

     % greater than operation %
     result = y > x ? "greater than operation works" : "greater than operation broken";
     println result;

     % greater than and equal to operation %
     result = value >= z ? "greater than and equal to operation works" : "greater than and equal to operation broken";
     println result;

     % less than operation %
     result = x < y ? "less than operation works" : "less than operation broken";
     println result;

     % less than or equal to operation %
     result = value =< z ? "less than or equal to operation works" : "less than or equal to operation broken";
     println result;

     % not equal to operation %
     result = y != x ? "not equal to operation works" : "not equal to operation broken";
     println result;

     return "relational operation checking is completed";
stop
```

Output

```
equal operation works
greater than operation works
greater than and equal to operation works
less than operation works
less than or equal to operation works
not equal to operation works

relational operation checking is completed

--------------------
Process finished.
Running time : 58 ms
```

Loops

DReaM Development Environment (4_loops.dm)

New | Open... | Save... | Compile | Run | About

```
1  %
2  Loops
3  %
4
5  function(x number)
6  start
7
8       % for loop %
9       x = 5;
10      println "for loop";
11      for (x; x =< 10; x++)
12      start
13           println x;
14      stop;
15
16      % while loop %
17      x = 5;
18      println "while loop";
19      while (x =< 10)
20      start
21           println x;
22           x += 1;
23      stop;
24
25      % range loop%
26      x = 5;
27      println "range loop";
28      for x in range(5..10)
29      start
30           println x;
31      stop;
32
33      return "loop checking completed";
34 stop
```

Output

```
for loop
5
6
7
8
9
10
while loop
5
6
7
8
9
10
range loop
5
6
7
8
9
10

loop checking completed

--------------------
Process finished.
Running time : 54 ms
```

Conditional statements

DReaM Development Environment (10_conditional_statements.dm)

New | Open... | Save... | Compile | Run | About

```
1  %
2  Conditional statements
3  %
4
5  function (x number, y number, z number)
6  start
7       % instatiating variables%
8       x = 1; y = 2; z = 3;
9
10      % if condition without else %
11      z *= x;
12      if (1 == 1)
13      start
14           println "short addition works";
15      stop;
16
17      % if else condition %
18      x += y;
19      if (x == z)
20      start
21           println "if else works";
22      stop;
23      else
24      start
25           println "if else is broken";
26      stop;
27
28      % Ternary operator %
29      output string;
30      output = z != y ? "working" : "broken";
31      output = "ternary operator" # output;
32      println output;
33
34      return "finished testing conditional statements";
35 stop
```

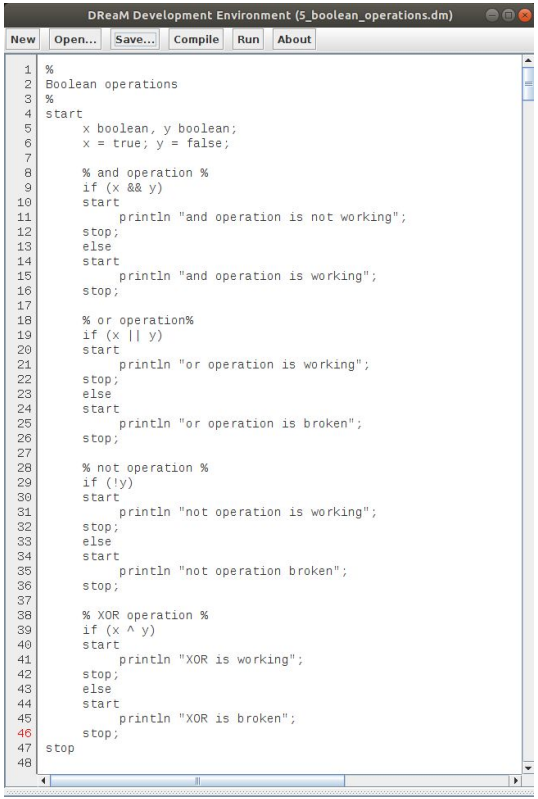
Output

```
short addition works
if else works
ternary operator working

finished testing conditional statements

-------------------
Process finished.
Running time : 58 ms
```

Boolean expressions




```
%
Boolean operations
%
start
     x boolean, y boolean;
     x = true; y = false;

     % and operation %
     if (x && y)
     start
          println "and operation is not working";
     stop;
     else
     start
          println "and operation is working";
     stop;

     % or operation%
     if (x || y)
     start
          println "or operation is working";
     stop;
     else
     start
          println "or operation is broken";
     stop;

     % not operation %
     if (!y)
     start
          println "not operation is working";
     stop;
     else
     start
          println "not operation broken";
     stop;

     % XOR operation %
     if (x ^ y)
     start
          println "XOR is working";
     stop;
     else
     start
          println "XOR is broken";
     stop;
stop
```

Output

```
and operation is working
or operation is working
not operation is working
xor is working

[(x,true,boolean),(y,false,boolean)]

--------------------
Process finished.
Running time : 57 ms
```

Increment, decrement, and modules

DReaM Development Environment (6_increment_decrement_modules_operations.dm)

New | Open... | Save... | Compile | Run | About

```
1  %
2  Modules, incrementer, decrementer operators
3  %
4  function(x number, y number, z number)
5  start
6       x = 2; y = 3;
7
8       % Modules operation%
9       z = x@y;
10      print "The remainder of 2 divided by 3 is ";
11      println z;
12
13      % Incrementer %
14      x++;
15      ++x;
16      print "The value of 2 after two increments is";
17      println x;
18
19      % Decrementer %
20      y--;
21      --y;
22      print "The value of 3 after two decrements is";
23      println y;
24
25      return "checking Modules Incrementer Decrementer operators is completed";
26 stop
```

Output

```
the remainder of 2 divided by 3 is 2
the value of 2 after two increments is 4
the value of 3 after two decrements is 1

checking modules incrementer decrementer operators is completed

------------------
Process finished.
Running time : 55 ms
```

Factorial numbers

DReaM Development Environment (7_factorial_numbers_example.dm)

New | Open... | Save... | Compile | Run | About

```
1  %
2  Factorial
3  Calculates factorial of a number
4  %
5
6  function(factorial number, counter number)
7  start
8       result string;
9       factorial = 1;
10      for counter in range(1 .. 5)
11      start
12           factorial *= counter;
13      stop;
14      result = "the factorial of 5 is " # factorial;
15      return result;
16 stop
17
```

Output

```
the factorial of 5 is 120

------------------
Process finished.
Running time : 60 ms
```

Fibonacci sequence

DReaM Development Environment (8_fibonacci_numbers_example.dm)

New | Open... | Save... | Compile | Run | About

```
1  %
2  Fibonacci number
3  Lists the fibonacci numbers starting from 1
4  %
5
6  start
7       last number;
8       previous number;
9       upperbound number;
10
11      % initialize variables %
12      last = 0;
13      previous = 1;
14      upperbound = 100;
15
16      output string;
17      output = "Prime numbers upto " # upperbound;
18      println output;
19
20      % loop until we hit 100 %
21      while (last < upperbound)
22      start
23           sum number;
24           sum = previous + last;
25           println sum;
26           last = previous;
27           previous = sum;
28      stop;
29 stop
```

Output

```
prime numbers upto 100
1
2
3
5
8
13
21
34
55
89
144
233

[(last,144,number),(previous,233,number),(upperbound,100,number),(output,prime numbers upto 100,string),(sum,233,number)]

------------------
Process finished.
Running time : 61 ms
```

Prime numbers

DReaM Development Environment (9_prime_numbers_example.dm)

New | Open... | Save... | Compile | Run | About

```
%
Prime numbers
Lists prime numbers between 1 to 100
NOTE: takes a while to compile (~ 10943 ms)
%

start
     counter number, outerIndex number, innerIndex number, maxnumber number;
     output string;

     maxnumber = 100;
     output = "Prime numbers until" # maxnumber # "are";
     println output;

     % loop until 100 %
     outerIndex = 1;
     for (outerIndex; outerIndex =< 100; outerIndex++)
     start
          %reset counter to zero%
          counter = 0;
          innerIndex = outerIndex;
          for (innerIndex; innerIndex > 0; innerIndex--)
          start
               if (outerIndex @ innerIndex == 0)
               start
                    counter += 1;
               stop;
          stop;
          %  print prime number %
          if (counter == 2)
          start
               println outerIndex;
          stop;
     stop;
stop
```

Output

```
prime numbers until 100are
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97

[(counter,9,number),(outerindex,101,number),(innerindex,0,number),(maxnumber,100,number),(output,prime numbers until 100are ,string)]

------------------
Process finished.
Running time : 1153 ms
```