The Meltean Programming Language

A domain specific programming language for processing & solving set theory problems

*Authors:*

Bogdan **Buduroiu** (bb4g15)

Mircea **Mihalea** (tbc)

# Introduction

The Meltean Programming Language is a weakly typed, dynamic, domain specific programming language. It’s specialises on processing set operation for handling computation on Languages. Such set operations include: *intersection, union, set subtraction, adding element to set.*

Meltean supports *string concatenation,* for the aid of operations on various languages*.*

# Syntax

## Data Types

Meltean supports declarations and operation on *Integers*, *Strings*, *Booleans* and *Sets*.

The language is dynamic and weakly typed, to favour programmer efficiency and writing less code to achieve the programmer’s goal.

Meltean treats all languages as regular sets. However, an important thing to note is that in Meltean, all the elements in the set are treated as strings.

### 2.1.1 Declaration & Initialization

As the language is weakly typed, all the variables are easily declared using the var keyword.

For declaration, the following syntax must be used:

**|** var <VARIABLE\_NAME> = <INT>|<STRING>|<BOOL>|<SET>;

**var** i = 0; // Initializes i as an integer

**var** j = **“Meltean”**; // Initializes j as a string

**var** m = **true**; // Initializes m as a Boolean with value true.

**var** n = {}; // Initializes n as an empty set

**var** p = **{“a”,”b”}**; // Initializes p as {a,b}

**! The Meltean language does not support declaration without initialization (this will result in a ParseError being thrown)**

As the language is also dynamic, any variable can change its type during the execution of the program. For example:

**var** i = 0; // Initializes i as an integer

**print** i; // Prints 0

i = **“String”**; // i to becomes a string

**print** i; // Prints String

### 2.1.2 Operations on Data Types

Operations supported:

* Addition (+), subtraction (-), multiplication (\*), division (/) and modulo (%) (**on integers**)
* Compound operations (^=) (**on strings**)
* Comparison operations (<, >, <=, >=) (**on integers**)
* Equality operations (==,!=) (**on integers, strings and bools**)
* Negation (!) (**on bools**)
* Concatenation (^) (**on strings**)
* Union (union), Intersection (intersect), Set Difference (diff), Append to set (add) (**on sets**)

Operations must be in form:

<operation> ::= <data\_type><operator><data\_type>

**where**

<data\_type> ::= <variable> | <INT> | <STRING> | <BOOL> | <SET>

**var** i = 0; **var** j = 1;

i = i + j; // i = 0 + 1

i = i + 2 \* j; // \* takes precedence over +

**print** i; // Prints 3

**var** str1 = **“Mel”**; **var** str2 = **“tean”**;

**print** str1 != str2; // Prints true

**var** str3 = str1 ^ str2; // str3 becomes concat of str1 & str2

str1 ^= **“Gibson”**;

**print** str3; // Prints Meltean

**print** str1; // Prints MelGibson

**var** a = **{“a”, “d”}**; **var** b = **{“a”, “b”}**;

**print** a **union** b; // Prints {a,b,d} in lexicographic order

**print** a **intersect** b; // Prints {a}

a = a **add** “c”; // a is now {a,b,c,d}

**print** a **union** b; // Prints {a, b, c, d}

**var** bool1 = **true**; **var** bool2 = **false**;

**print** bool1 == bool2; // Prints false

**print** i <= j; // Prints false

**print !**bool1; // Prints false

**print** 1 < 2; // Prints true

**The interpreter will throw an Illegal\_Operation exception should an invalid operation be performed, e.g. performing integer addition between two sets**

## Control Structures

### Conditionals

Meltean has support for IF..THEN statements and also IF..THEN..ELSE statements. The structure is

if (< bool\_condition >) { <statements> };

or if (< bool\_condition >) { <statements> } else { <statements> };

**if** (1 < 2) {

print **“Yes”**;

} **else** {

print **“No”**;

}; // Prints Yes

### Loops

Meltean has support for both FOREACH type loops as well as the FOR(<init>;<condition>;<step>) type loops. The structure for the FOREACH loop is:

for <var> in <set> { <statements> };

The structure for the traditional FOR loop is:

for <var> to <integer> { <statements> };

**At the moment, the language only supports an increment of 1.**

**In the traditional FOR loop, the variable used when looping (<var>) must be declared before the loop, similarly to the C language.**

**var a = {“a”, “b”};**

**for** i **in** a {

**print** i;

}; // Prints a \n b \n

**var** x = 0;

**for** x **to** 5 {

**print** x;

}; // Prints 0 \n 1 \n 2 \n 3 \n 4 \n

## Program Structure

Every program must begin with a begin and end with an end.

**begin**

**var** set = **{}**; **var** a = **“a”**; **var** i = 0;

**for** i **to** K {

a = a ^ **“a”**;

set = set **add** a;

};

**print** set;

**end** // Prints out K iterations of Kleene Star of {a}

# Interpreter

To execute a program, the program file must be passed to the interpreter as a command line argument, and should an input file be required, the input file must be passed to the interpreter through the standard input device handle.

$> ./mysplinterpreter prog1.ol < prog1.in

The interpreter treats each line in the input file as a set, and stores it as a variable INPUTn (where **n** is the line number in the file). The interpreter continues to read the file until it reaches an integer, which it stores as variable K.

**Should the integer at the end of the file be missing, the interpreter will read until EOF and then output a “Variable not declared” error.**