SOL: Set Operation Language

Language Reference Manual

Taylor Brown

tbb2107

Contents

1. Introduction

SOL is a computer language based on set theory and functional programming. It is designed to allow easy manipulation of sets, including filtering and set based operations. The targeted user space is those wishing to implement graph based algorithms.

2. Lexical Conventions

SOL utilizes several different kinds of tokens.

2.1. Comments

The characters // denote that the rest of the line is a comment, terminated by a new line. These are ignored.

2.2. Keywords

The following are reserved keywords:

if

else

function

end

2.3. Constants

2.3.1. Integer

An integer consists of a sequence of one or more digits.

2.3.2. String

A string consists of a double quote " followed by zero or more characters, ended by a double quote.

2.4. Whitespace

Whitespace characters, such as spaces, tabs and newlines, are not significant, and only serve to delineate tokens.

2.5. Identifiers

An identifier is a string of letters and digits, of which the first letter is in the set a-Z. Identifiers refer to integers, sets, filters or functions. Functions must be explicitly declared – sets, strings and integers are inferred. Identifiers can be assigned to other identifiers, in which case the value is copied to the new variable.

Identifiers are case sensitive.

3. Operators

Note – SOL doesn't perform type checking on operators – it is up to the user to use proper types and operators.

3.1. Equals

= used to represent assignment

The assignment operator is used for sets strings and integers.

Set variables are assigned with the equals sign followed by open/close brackets.
[identifier] = { [expression] }
Integer variables are assigned with equals and a number
[identifier] = [number]
String variables are assigned with equals and a constant
[identifier] = "string"

3.2. Set Operators

3.2.1. Union

expr+ *expr* used to represent the union of two sets

3.2.2. Intersection

expr& *expr* gives the intersection between two sets

3.2.3. Difference

expr – *expr* gives the difference between the first and second set

3.2.4. Cartesian product

* Cartesian product for sets

The Cartesian product of a set produces unordered pairs, and is thus not a true Cartesian product, but an approximation. E.g.

{1,2}* {3,4} returns {{1,3},{1,4},{2,3},{2,4}}

3.3. Arithmetic

+ between integers, addition

- between integers, subtraction

* between integers, multiplication

/ between integers, division

3.4. Relational Operators

Equality and relational operators compare both sets and integers. They return a true or false for both set and integer operators.

3.4.1. Disjoint Set

!< no common elements between given sets

3.4.2. Equality

== tests equality for sets integers and strings

3.4.3. Subset

< for x<y, tests if x is a subset of y

3.4.4. Less Than, Greater Than

.< less than, for integers

.> Greater than, for integers

3.5. Logical Operators

Logical operators return true or false based on standard operations.

3.5.1. Or

| or operator

3.5.2. And

& and operator

3.5.3. Not

! negates a Boolean value

3.6. Separators

3.6.1. The comma operator ',' is used to separate items delineated in a set, e.g.:

{expr, expr}

3.6.2. Braces are used to demarcate sets. The set is defined as the expressions within braces, e.g.:

{ expr, expr,…}

3.6.3. Colons are used to end function declaration lines, e.g.:

function somefunction somearg :

4. Scope

Scope is defined by nesting. At the top level, all variables are available global. Within a function, variables declared are local and exist only within the function. Use of a global variable name within a function will overwrite the global value, as there is no way to differentiate between a newly assigned variable and the global variable. This applies to all identifiers.

5. Expressions

All statements except function declarations, ends, if, and else are expressions. This implies assignments, function calls and relational expressions. An expression is limited to a single line.

6. Conditionals

Conditionals are defined in the following form:

If (expression) statement **else** statement **end**

The else is required. The expression must be of the form to return true or false, thus using a relational operator.

7. Functions

Functions are created by the use of the function keyword. Functions take a specified number of named arguments. The last line executed in the function is returned. Functions take the form:

function [identifier][n-args]:

    function body

end

The arguments specified by [n-args] are a list of identifiers that will have local scope in the function. They take the form

[identifier] [identifier] …

The function body is made up of other statements – assignments, conditionals, function calls. Nested functions are not allowed. Functions can call themselves. Functions are ended with the end keyword.

Functions may be passed as arguments to a function.

8. Integers

Integers are the most basic data type – they consist of numbers. They may be assigned to variables, members of a set, or returned from functions as values.

9. Strings

Strings are a series of characters. There are no escape sequences. Strings are immutable, and cannot be combined or split apart. They may be returned from functions as values and compared in terms of equality.

10. Sets

Sets are the primary data structure of SOL. They are an unordered collection of members, including other sets, functions, strings and integers. Sets are immutable. They are declared using braces, e.g.

[identifier] = { [members] }

As specified below, there are a number of built in set functions.

Sets are not typed – any number of different members is allowed in a single set.

Every element in a set is unique, as defined by a recursive member comparison. For a set of all integers contains only unique integers. For sets within a set, no two sets may contain the same sub-elements. E.g., A = {1,2,{1,2}, {2,1}} is invalid, as {1,2} is equivalent to {2,1} since sets are not ordered. This applies to nested subsets, so the following would be valid:

{1,2,{1,2,{3}},{1,2,{{3}}}}

The empty set is defined as {}. The empty set is a subset of all sets. The union of the empty set and some set is the set. The intersection of the empty set and some set is the empty set.

11. Built in Functions

11.1. sizeof(set)

The sizeof function takes a set, and returns the size of the set as an integer. E.g., *Sizeof({1,2,3,{1,2}})* returns 4

11.2. pop(set)

The pop function takes a set, and returns an assumed random element from the set. *pop({1,2,3})* could return 2, 1, or 3

11.3. push(set, expr)

The push function takes a set and an expression, and returns a new set with expr added to the set.

Push({1,2},3) returns {1,2,3}

11.4. print(expr)

The print function takes either a set or an integer and prints the value to standard output, returning {}.

print("hello world") returns {} and prints hello world to standard output.

11.5. map(function, set)

The map function takes a function and a set, and applies the function to each element in the set, returning a new set of all the elements.

map(sizeof, {1,2,{3,4,5}}) returns {1, 3}

11.6. filter(function, set)

The filter function takes a function a set, and applies the filter to each item in the set. If the function returns true, the element is added to the return set, otherwise it is dropped.

e.g.:

function greaterThan5 x:

x > 5

end

filter(greaterThan5, {1,2,6,7})

returns {6,7}

12. Sample program

The sample program computes an independent set from the given graph recursively. The set is not maximal.

```
function IndependantSet iSet graph:
  if graph = {}:
    iSet
  else:
    edge = pop graph
    vertext = pop edge
    if iSet & vertex = {}:
      IndependantSet iSet + vertext graph;
    else:
      IndependantSet iSet graph
    end
  end
end
IndependentSet {} {{1,2}{2,3}{3,4}{4,1}}// returns {1,3} or {2,4} or {1} or {2} or {3} or {4} - not maximal
```