SetC: The SetConcise Language Reference Manual

Julian Kocher (jk3813), Frank Ling (ffl2107), Heather Preslier (hnp2108)

February 22, 2017

Contents

1 Introduction **4**

2 Lexical Conventions **4**
- 2.1 Tokens . . . 4
- 2.2 Comments . . . 4
- 2.3 Identifiers . . . 4
- 2.4 Keywords . . . 5
- 2.5 Literals . . . 5
  - 2.5.1 Integer Literals . . . 5
  - 2.5.2 Floating Literals . . . 5
  - 2.5.3 String Literals . . . 5

3 Conversion **6**

4 Meaning of Identifiers **6**
- 4.1 Basic Types . . . 6
- 4.2 Derived Types . . . 6

5 Expressions **7**
- 5.1 Atoms . . . 7
- 5.2 Primary Functions . . . 7
  - 5.2.1 Indexing . . . 7
  - 5.2.2 Function Calls . . . 7
- 5.3 Set and Array Constructions . . . 7
  - 5.3.1 Simple Bracket Construction . . . 7
  - 5.3.2 Set Theoretic Construction . . . 8
- 5.4 Compound Set Operator . . . 8
- 5.5 Unary Operators . . . 8
- 5.6 Multiplicative Operators . . . 9
- 5.7 Additive Operators . . . 9
- 5.8 Relational Operators . . . 9
- 5.9 Equality Operators . . . 9
- 5.10 Logical AND Operator . . . 9
- 5.11 Logical OR Operator . . . 10
- 5.12 Set Operators . . . 10
  - 5.12.1 Intersection . . . 10

5.12.2 Union . . . 10
5.12.3 Difference . . . 10
5.12.4 Symmetric Difference . . . 10
5.12.5 Cardinality . . . 10
5.12.6 Declaration . . . 11
5.12.7 Element Existence . . . 11
5.12.8 Such-That . . . 11
5.12.9 Set Slicing . . . 11
5.12.10 Iteration . . . 11
5.13 Array/String Operators . . . 11
5.13.1 Concatenation . . . 11
5.13.2 Cardinality . . . 12
5.13.3 Declaration . . . 12
5.13.4 Element Existence . . . 12
5.13.5 Slicing . . . 12
5.13.6 Iteration . . . 12

6 Statements **12**
6.1 Assignment Statements . . . 12
6.2 Expression Statement . . . 13
6.3 Compound Statement . . . 13
6.4 Selection Statement . . . 13
6.5 Iteration Statements . . . 13

7 Scope and Linkage **14**

8 Grammar **14**

1 Introduction

The SetC language is a dynamically typed language that has functionality for basic programming and set construction, manipulation and operation. The most notable attribute of the SetC language is the unique set theoretic notations present in constructs such as the for loop construction and compound set operator. The dynamic typing of the language removes the need for all type declarations except for functions, which are preceded by the string `def`.

2 Lexical Conventions

2.1 Tokens

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Spaces, tabs, and newlines can be used interchangeably to separate tokens. Some whitespace is always required to separate tokens.

2.2 Comments

Similar to the C language, characters `/*` introduce a comment, which terminates with the characters `*/`. Comments do not nest, and they do not occur within string or character literals. Comments are ignored by the syntax; they are not tokens.

2.3 Identifiers

All identifiers must begin with a lowercase alphabetic letter, followed by alphanumeric characters or the underscore _ character. Identifiers refer to memory locations that contain data of a dynamic type. Identifiers can be any length. Letter case is significant.

2.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
array       if
break       in
case        int
continue    none
def         return
default     set
elif        switch
else        while
float
```

2.5 Literals

Literals are constant values of some built-in type. There are three type of literals: ints, floats, and strings.

2.5.1 Integer Literals

Integer constants are a sequence of digits that are represented as a decimal.

2.5.2 Floating Literals

A floating constant consists of an integer part, a decimal point, and a fractional part. The integer part may be absent if the decimal point and fractional part are present. The fractional part may be absent if the integer part and decimal point are present.

2.5.3 String Literals

A string literal consists of a collection of characters enclosed in double quotes, such as "...". It is possible to index through a statically declared string, but it is not possible to manipulate the data contained in the string. Adjacent string literals are concatenated into a single string.

3 Conversion

Conversion of types in SetC is mostly through built-in functions, such as `arr_to_str()`, `str_to_arr()`, `arr_to_set()`, `set_to_arr()` and `flt_to_int()`. Automatic promotion of types only occurs for the following case between floating point numbers and integers: if either argument is a floating point number, the other is converted to floating point.

4 Meaning of Identifiers

Identifiers can be function names, variable names, or other such declared expressions. The basic types of identifiers include int, float, string, and none. The `none` type is equivalent to the concept of `void` in C or `None` in Python. The derived types of identifiers include sets and arrays. All identifiers must be initialized when declared.

4.1 Basic Types

The fundamental types present in SetC are integers, floats, none and strings. All basic types are immutable data types.

4.2 Derived Types

The derived types are collections constructed from the fundamental types in any number of ways. Both derived types are mutable. While both a set and array can contain multiple data types, there is a key difference between them. A set is a possibly empty unordered sequence of unique fundamental types enclosed in curly braces. An array is a possibly empty ordered sequence of fundamental types enclosed in square brackets. When a comma-separated list of expressions is supplied to arrays, its elements are evaluated from left to right and placed into the array object in that order. When set theoretic notation is supplied, the set and array types are constructed from the elements resulting from the notation.

5 Expressions

5.1 Atoms

Atoms are the most basic elements of expressions. Identifiers, constants, strings, or expressions in parentheses are atoms.

5.2 Primary Functions

Primary functions are the operations that bind most tightly to atoms. Primary functions group left to right.

5.2.1 Indexing

A primary function followed by an expression in square brackets is a subscripted array reference. The first expression must have array, set, or string type and the expression enclosed in brackets must evaluate to an integer type.

5.2.2 Function Calls

A function call is a primary function followed by parentheses containing a possibly empty comma-separated list of arguments.

5.3 Set and Array Constructions

Sets and arrays can be constructed similarly. This section details the construction of sets.

5.3.1 Simple Bracket Construction

Sets can be declared by listing the set contents explicitly within square brackets (and curly braces for arrays).

```
e.x set1 = [1, 2, 3];
```

5.3.2 Set Theoretic Construction

This construction, modeled after set notation, has the format of an expression, any number of constraints, the 'such that' operator (|), followed by an expression (i.e. expression, constraints | expression). If undefined identifiers are used in the first expression, constraints must specify the range of values that they can take. The first identifier must have well defined parameters (i.e. it must not be contingent on any other identifier in the set-expression). Any following identifier can be contingent on identifiers present in the expression. This expression returns a set constructed from the set-expression for each value of the constraints.

```
E.x.  set2= [2*i-1, 0 < i < 5 | i <= 5]; /* [1, 3, 5] */
E.x.  set3 = [[x,y], 0 <= x <= 2, y = x + 1];
                    /* [[0,1],[1,2],[2,3]] */
```

5.4 Compound Set Operator

The compound set operator is used to compound a result from set or array elements of the same type. It applies the operator to the result of the expression after each iteration. It returns a single literal type corresponding to the types of the elements of the set. The set operator will return an error if the collection has elements of different types. The syntax is as follows:

```
[operator:  constraints | boolean expression] expression
Ex.  [+:  x in 1,3,4,6,10 | x % 2 == 0] /* returns 20 */
```

5.5 Unary Operators

There are three unary operators: the #, !, and – characters. The # operator is a cardinality operator used with sets, arrays or strings and returns the length. The ! operator is used for logical negation of boolean expressions. This returns an int of 1 or 0, which are true and false, respectively. The – operator is used for arithmetic expressions or literals and the result is the negative of its operand.

5.6 Multiplicative Operators

The multiplicative operators *, / and % group left to right. The operands of * and / must be of arithmetic type. The operands of % must of integral type. The binary operator * denotes multiplication. The binary operator / denotes division. The binary operator % denotes modulo.

5.7 Additive Operators

The additive operators + and − groups from left to right. If the operands of + and − are of arithmetic type, then the result of the operation is the standard arithmetic result. The + operator denotes addition and the operation results in the sum of the operands. The − operator denotes subtraction and the operation results in the difference of the operands.

5.8 Relational Operators

The relational operators < (less), > (more), <= (less or equal) and >= (more or equal) group from left to right. The operands of relational operators must be of the arithmetic type. An relational operation evaluates to 1 or 0, which is true or false, respectively.

5.9 Equality Operators

The equality operators == (equal to) and != (not equal to) groups from left to right. The equality operators have a lower precedence than the relational operators. An equality expression evaluates to a 1 or 0, which is true or false, respectively.

5.10 Logical AND Operator

The logical AND operator && groups from left to right. It returns 1 if both operands evaluate unequal to zero, and returns 0 otherwise. The operands of the AND operation must evaluate to an arithmetic value. It returns a value of int type.

5.11 Logical OR Operator

The logical OR operator || evaluates from left to right. It returns 1 if either of the operands compare unequal to zero and returns 0 otherwise. If the left hand operand compares unequal to zero, the right hand operand is not evaluated.

5.12 Set Operators

Evaluation of set operations groups from left to right. Operands must be of set type.

5.12.1 Intersection

The intersection operator + results in a new set that contains only the elements that are present in both operand sets.

5.12.2 Union

The union operator * results in a new set that contains all the elements in both operands without repeating elements.

5.12.3 Difference

The difference operator − evaluates to a set that contains all elements of the left operand that are not present in the right operand.

5.12.4 Symmetric Difference

The symmetric difference operator ^ for sets evaluates to a set that contains the elements that are either present in the left or right operand but not in both.

5.12.5 Cardinality

The unary cardinality operator # returns the number of elements in a set where the right operand denotes the set.

5.12.6 Declaration

Sets are declared with { and } respectively, where the open brace denotes the beginning of a set and the closed brace denotes the end.

5.12.7 Element Existence

The operator `?` is used to check the existence of an element in a set. Specifically, the left operand is the element being searched for, while the right operand is the set the check is being performed on.

5.12.8 Such-That

The such that operator `|` is used to delineate portions of set theoretic construction, where the left operand can be an expression followed by constraints and the right operand is a boolean expression.

5.12.9 Set Slicing

Set slicing is a method of creating subsets from sets using the `:` operator. A slicing selects a range of items in a object. Slicing returns a new set that contains the sliced portion. The `:` should be preceded and followed by integer expressions corresponding to the start and stop indices.

5.12.10 Iteration

The iterative operator `in` is used to iterate through a set of elements. This is very similar to the `for value in range` statement in Python, but without the `for` token.

5.13 Array/String Operators

5.13.1 Concatenation

The concatenation operator + concatenates two strings or arrays. The operands of the operator must both be of type string or array.

5.13.2 Cardinality

The cardinality operator `#` returns the number of elements in an string or array. Similar to the set cardinality operation.

5.13.3 Declaration

Arrays are declared with `[` and `]` respectively where the open bracket denotes the beginning of an array and the close bracket denotes the end. A string is declared with double quotation marks.

5.13.4 Element Existence

The operator `?` is used to check the existence of an element. Specifically, the left operand is the element being searched for, while the right operand is the array or string the check is being performed on. This operation is similar to set element existence.

5.13.5 Slicing

Slicing is a method of creating smaller subsets of arrays or strings using the `:` operator. A slicing selects a range of items in a object. Slicing returns an array that contains the sliced portion. Similar to the set slicing operation.

5.13.6 Iteration

The iterative operator `in` is used to iterate through a set of elements. This is very similar to the `for value in range` statement in Python, but without the `for` token. Similar to the set iteration operation.

6 Statements

A statement describes an action to be performed (usually sequentially). Statements are completed using the semicolon `;` operator.

6.1 Assignment Statements

Assignment statements are used to rebind names to values and to modify attributes or items. An assignment statement therefore requires that the

lvalue (left hand value) is one that is mutable or modifiable. It also requires that both operands evaluates to the same type. When the = operator is used, the lvalue expression will be replaced by the right hand expression and the type of the lvalue will take the type of the right hand expression. The assignment operators `=, *=, /=, %=, +=` and −= group from right to left. Use of the assignment operators other than = is as follows:

expr1 op= expr2 which is equivalent to

expr1 = expr1 op expr2.

The operands of this statement must be compatible with the operator used.

6.2 Expression Statement

Expression statements can be used to compute or write values or for function calls. Statements can be simple or compound; simple statements only enclose themselves.

6.3 Compound Statement

A compound statement consists of simple statements, or other compound statements. Compound statement declarations retain scope within their encompassing curly braces. If identifiers of the same name are declared earlier in scope, only the most recent declaration identifier will be visible.

6.4 Selection Statement

Selection statements are made up of `if, elif, else` and `switch` statements. They are used to evaluate an expression, and depending on this evaluation, execute a specific action. These selection statements are applied similarly in SetC as they are in C.

6.5 Iteration Statements

Iteration statements have two varieties: statements declared with `while` and statements using set theoretic notation. `while` loops are contingent on the structure of the conditional statement inside the parentheses that follow. For set theoretic iteration, constraints must be inside the parentheses. The variables in these constraints are the loop variables. Encompassing this variable with < or <= operators will increment the variable, and > or >= operators

will decrement the variable. The bounds on the variable must produce a valid range. For instance, `0 < x <= 5` will increment the variable `x` from 1 to 5, but `0 < x > 5` is invalid. For more information about set theoretic notation, see section 5.3.2.

7 Scope and Linkage

The scope of the identifiers are defined within the region of code in which they are declared, denoted using curly braces { and }. SetC consists of local and globally declared identifiers. The lifetime of the identifier is determined by its scope. For linkage, identifiers can be shared across multiple files given that the identifiers are unique and non-repeated.

8 Grammar

statement:
- labeled-statement
- assignment-statement
- expression-statement
- compound-statement
- selection-statement
- iteration-statement
- jump-statement

labeled-statement:
- `case` constant-expression : statement
- `default` : statement

assignment-statement:
- identifier assignment-operator atom
- unary-expression assignment-operator assignment-statement

assignment-operator: one of
- `= *= /= %= += -=`

expression-statement:

expression$_{opt}$;

compound-statement:
{ statement-list$_{opt}$ }

statement-list:
statement
statement-list statement

selection-statement:
`if` (expression) statement
`if` (expression) statement `else` statement
`if` (expression) statement `elif` statement `else` statement
`switch` (expression) statement

iteration-statement:
`while` (expression) statement
(constraints — expression$_{opt}$) statement
identifier `in` collection

jump-statement:
`continue` ;
`break` ;
`return` expression$_{opt}$;

expression:
compound-set-expression

compound-set-expression:
conditional-expression
compound-set-expression conditional-expression
`[` operator: constraints `|` expression `]` conditional-expression

operator: one of
`+ - * / %`

conditional-expression:
logical-OR-expression

logical-OR-expression:
    logical-AND-expression
    logical-OR-expression `||` logical-AND-expression

logical-AND-expression:
    equality-expression
    logical-AND-expression `&&` equality-expression

equality-expression:
    relational-expression
    equality-expression == relational-expression
    equality-expression `!=` relational-expression

relational-expression:
    additive-expression
    relational-expression < additive-expression
    relational-expression > additive-expression
    relational-expression <= additive-expression
    relational-expression <= additive-expression

additive-expressions:
    multiplicative-expression
    additive-expression + multiplicative-expression
    additive-expression − multiplicative-expression

multiplicative-expression:
    set-expression
    multiplicative-expression `*` set-expression
    multiplicative-expression / set-expression
    multiplicative-expression `%` set-expression

set-expression:
    unary-expression
    set-expression `?` unary-expression
    set-expression ^ unary-expression

unary-expression:

primary-function
unary-operator primary-function

unary-operator: one of
! # -

primary-function:
atom
primary-function [expression]
primary-function (argument-expression-list)
slicing

atom:
identifier
literal
collection

collection:
set
array
(expression)

argument-expression-list:
assignment-expression
argument-expression-list, assignment-expression

slicing:
primary-function [slice-list]

slice-list:
slice-item
slice-list, slice-item

slice-item:
expression | proper-slice

proper-slice:
expression : expression : expression

literal:
    integer-literal
    float-literal
    string-literal

set:
    [expression-list]
    [construction]

array:
    { expression-list }
    { construction }

expression-list:
    expression
    expression-list, expression

construction:
    expression , constraints | boolean-expression
    constraints | boolean-expression
    expression , constraints

constraint-list:
    constraint
    constraint-list , constraint

constraint:
    integer-literal < identifier < integer-literal
    integer-literal > identifier > integer-literal
    integer-literal <= identifier <= integer-literal
    integer-literal >= identifier >= integer-literal
    integer-literal < identifier <= integer-literal
    integer-literal > identifier >= integer-literal
    integer-literal <= identifier < integer-literal
    integer-literal >= identifier > integer-literal