SMAP Language Reference Manual

```
 ,-.    .                   .   ,                     .     .
(   `   |       o           |\ /|           o         |     |   o
 `-.    |-  ;-. . ;-. ,-:   | V |   ,-: ;-. . ;-. . . | ,-: |-  . ,-. ;-.
.   )   |   |   | | | | |   |   |   | | | | | | | | | | | | |   | | | | |
 `-'  o `-' '   ' ' ' `-|   '   ' o `-` ' ' ' |-' `-` ' `-` `-' ' `-' ' '
                      `-'                     '

         ,.          .   ;-.            .       .     .   .
        /  \         |   |  )           |       |   o | o |
        |--|   ;-. ,-|   |-'    ;-. ,-. |-. ,-: |-. . | . |-  . .
        |  |   | | | |   |      |   | | | | | | | | | | | |   | |
        '  ' o ' ' `-'   '    o '   `-' `-' `-` `-' ' ' ' `-' `-|
                                                              `-'
```

Introduction

This manual describes the SMAP language that was introduced by the team during the proposal milestone. The manual describes the concrete standards of the language and its syntax that is being used to implement this probabilistic language, along with the other numerous features that can be implemented using the language. The language grammar provides an unambiguous parsing to the SMAP language.

Lexical Conventions

Tokens

There are five classes of tokens: identifiers, keywords, string literals, operators and other separators. Other tokens are ignored because they act as separators, i.e. they separate the tokens. Blanks, horizontal and vertical tabs, newlines, formfeeds and comments are part of the group of tokens that are ignored while parsing the program.

Comments

Single-line comments are denoted by the characters `//` which ignore all text following them until a newline is met. Multi-line comments are denoted by the characters `/*` and `*/` which introduce and terminate the multi-line comment, respectively. Everything between the paired multi-line comment characters is ignored. Comments do not nest and do not occur in literals.

Identifiers

Identifiers are a sequence of letters and digits. SMAP is case-sensitive since lowercase and uppercase letters are distinguished to allow for camel case, and the underscore `_` is included to allow for snake case. Identifiers must begin with an underscore or letter.

```
okayToUseCamelCase //ok
okay_to_use_snake_case //ok
OkayToStartWithCapital //ok
_alsoOkay //ok
9_not_okay_to_start_with_number&not_okay_non_alphanumeric-=@#% // invalid
&not_okay_non_alphanumeric-=@#% // invalid
```

Keywords

The following identifiers are reserved for use as keywords:

```
bool break case char default continue if elif else float for fn int list prob return string switch true false void
while .length
```

Primitive Types and their Literals

Integer Literals

A sequence of digits in decimal format. Cannot start with a 0.

Bool Literals

Either `true` or `false`, implemented as integers 0 or 1.

Float Literals

A sequence of digits in decimal format containing a single `.` for the decimal point.

Character Literals

A text character in single quotes, for example `'a'`. Escape characters `'\n'`, `\\`, `'\''`, `'\"'`, represent new line, regular slash, single quote, and double quote, respectively.

String Literals

A sequence of characters in double quotes, for example, `"hello"`. String literals are implemented as lists of `char`s. See section 7 for details on implementation.

Lists

A list is a collection of a homogenous and mutable type of data, i.e. all of the objects inside the list would be of the same type. A list is also dynamically sized, i.e. the list size is mutable.

There are multiple ways to initialize a list. Lists are defined by the `list` keyword followed by the singular type that the list will contain. Lists can be created with an initial set of data or be empty and can have elements added to them dynamically.

```
// Creating a list
list int myints = [3, 2, 5];
list int myints = [3, 2, 5];
list int emptylist;

//Indexing a list using positional notation:
myints[0] == 3; //evaluates to true
myints[0] = 1;
myints[0] == 1; //evaluates to true
```

Oftentimes, we need to append a value at the end or beginning of the list. This is easily done with the `>` and `<` symbol within list indexing.

```
myints[>] = 42;
myints[<] = 37;
// myints is now: [37, 3, 2, 5, 42]
```

Getting the length of the list is possible by using the `.length` operator which returns the length as an integer.

```
myints.length == 5; //evaluates to true
```

We allow the `++` operator on the `list` type to concatenate two lists of the same type together.

```
list int numsToAdd = [100];
list int concatResult = myints ++ numsToAdd;
// concatResult == [37, 3, 2, 5, 42, 100]
```

We can delete elements from a list using the right shift `>>` and left shift `<<` operators. These operators are followed by an integer indicating how many elements should be deleted. These operations can be considered as 'pushing' operators since the right shift `>>` 'pushes' off a number of elements from the right side and the left shift `<<` 'pushes' off a number of elements from the left side.

```
// concatResult == [37, 3, 2, 5, 42, 100]
concatResult >> 2; // concatResult == [37, 3, 2, 5]
concatResult << 1; // concatResult == [3, 2, 5]
```

Strings

A string is syntactic sugar for a `list` of type `char`. Since strings are of type `list char` under-the-hood, strings can be defined either as a `string` type or as a `list char` as there is no difference.

String literals are defined as any ASCII character between a pair of double quotes (ex. `"this is a string"`). Literals are cast to their respective string representations at compilation time which allows for incredibly easy string manipulation and string building.

String Initialization

Strings can be initialized to string literals or variables of type `list char` or `string`.

```
string stringFromLit = “apple”; //"apple"
string stringFromLit2 = “ banana”; //"banana"
list char listFromString = stringFromLit; //"apple"
list char listFromStringLit = “apple”; //"apple"
```

String Promotion

Any literal of type `int`, `float`, `char`, or `bool` can be promoted into a string when being assigned to a variable of type `string`.

```
string number = 5; //"5"
string floatingNum = 5.0; //"5.0"
string charStr = 'a'; //"a"
string boolStr = true; //"true"
```

String Concatenation

Because lists already have a concatenation operator `++`, concatenating strings is conceptually the same.

```
string concatString = stringFromLit ++ stringFromLit2; //”apple banana”
list char anotherConcat = concatString ++ stringFromLit2 ++ “ “ ++ 5; //"apple banana banana 5"
```

Probability Type

The probability type is polymorphic; any type can be made into a probabilistic type with the `prob` type qualifier. The probability type models a discrete distribution by matching probabilities to values.

Probability types are initialized with a pair of non-empty lists separated by a colon. The left hand side list represents probabilities and must be of type `list float`. The right hand side list represents values and must be of type list `type`, where `type` is the type declared after the `prob` type qualifier in the declaration. The floats within the probability distribution must be positive.

Once initialized, the list of values inside the probability type is immutable.

Here's an example of a good use of a probability type. Consider creating a game that requires enemies to be spawned of some difficulty. The difficulty of the enemy is represented by the integers `1`, `2`, and `3` from easiest to hardest. Let's define a probability type called `randDifficulty` to pick these dificulties according to a programmer-defined distribution.

```
// prob <type> myprob = list float : list <type>
// the two type-identifiers must match!

prob int randDifficulty = [0.5, 0.25, 0.25] : [1, 2, 3];
```

Initialization

If lists of two different lengths are used during probability type initialization, the longer list is truncated to the length of the shorter list by removing elements from end. The length must be greater than or equal to 1 (probability types cannot be initialized with lists of zero length).

The probability distribution does not need to sum to 1 during initialization. SMAP normalizes the probability distribution during initialization and transformations (see *Modifying Probability Types with Transformations* and *Normalization*).

Accessing Fields

`.length`

The 'length' of a probability type returns the length of its lists.

```
prob int aNum = [0.25, 0.25, 0.5 ] : [7, 5]; // truncated and normalized to [0.5, 0.5] : [7, 5] during init
int len = aNum.length; // len == 2
```

Probability Distribution

The `#` operator extracts the list of probabilities. It is placed directly after the probability type.

```
prob int randDifficulty = [0.5, 0.25, 0.25] : [1, 2, 3];
list float vals = randDifficulty#; // vals == [0.5, 0.25, 0.25]
```

Values

Values are selected by a probability range (inclusive) defined within brackets. This would return all the values with a matching probability within the provided range. The return type is a list of the value type defined by the programmer during probability type initialization.

```
prob int randDifficulty = [0.5, 0.25, 0.25] : [1, 2, 3];
// return all values that have a matching probability between 0 and 1 (ie. all values)
list int vals = randDifficulty[0,1]; // vals == [1, 2, 3]

// return all values with probabilities within the range 0.0 - 0.25 (inclusive)
list int unlikelyVals = randDifficulty[0, 0.25]; //vals == [2, 3]
```

Picking Value According To Distribution

The `!` operator extracts the list of probabilities based on the underlying distribution.

```
int num = randDifficulty!; // num is 1, 2, or 3
```

Modifying Probability Types With Transformations

We use transformations to safely modify a probability type’s underlying distribution. Transformations use the arithmetic operators `+`, `-`, `*`, `/` as part of a “zipwith” function, and then normalize the resulting list of floats to represent a new distribution. Transformation expressions can be generalized into four forms, with a probability type required on the left side of the operation.

```
probType OP anotherProbType
probType OP list float
probType OP list int
probType OP float
probType OP int
```

Let’s take a look at these transformations in action. Consider the first case of `probType OP anotherProbType`.

```
prob int num = [0.25, 0.15, 0.15, 0.45] : [6, 87, 4, 302];
prob char letter = [0.25, 0.75 ] : [‘a’,’b’];
letter = letter * num; // letter is [0.36,0.64] : [‘a’,’b’]
```

How did we get the result `[0.36,0.64] : [‘a’,’b’]`? Remember that the value lists of `num` and `letter` are immutable. Since `letter` is on the left hand side of the transformation, its values are chosen to appear in the resulting probability type. **Note that this means a transformation between two probability types is associative but not commutative!** Next, `letter#` and `num#` are zipped with the `*` operator. Transformations use the probability type’s length (the minimum of its two list lengths) when performing `zipwith`, so the expression can be broken down as follows.

```
letter * num
zipwith letter# num# *
zipwith [0.25, 0.75]  [0.25, 0.15, 0.15, 0.45] *
[0.25*0.25, 0.75*0.15]
[0.0625, 0.1125]
```

Then the transformation normalizes the probabilities,

```
N([0.0625, 0.1125]) = [0.0625/0.157, 0.1125/0.157] = [0.36,0.64]
```

and final result is

```
[0.36,0.64] : [‘a’,’b’]
```

Another example, `probType OP list float`:

```
prob int changedNum = Num * [0.2, 0.3];
// [0.53, 0.047]:Num[0,1]
// zipwith Num# letter# *
// zipwith [0.25, 0.15, 0.15, 0.45] [0.2, 0.3] *
// [0.25*0.2, 0.15*0.3]
// [0.05, 0.045]
// Normalize([0.05, 0.045]) = [0.05/0.095, 0.045/0.095] = [0.53, 0.047]
// final result is [0.53, 0.047] : [6, 87, 4, 302]
```

Since `int` s are implicitly converted to `float` s in `float op int` or `int op float` expressions, `probType OP list int` expressions are implicitly converted expressions of `probType OP list float`.

Another example, `probType OP float`:

Expressions of `probType OP float` are implicitly converted to expressions of `probType OP list float`, where the `float` transforming the probability is promoted to a list of equal length with a single repeated value.

```
letter = letter + 3;
// [0.48, 0.52]:[‘a’, ‘b’]
// since letter.length = 2, 3 gets promoted to [3, 3]
// now we can do zip [0.36, 0.64] [3,3] + = [3.36, 3.64]
// Normalize([3.36, 3.64]) = [3.36/7, 3.64/7,] = [0.48, 0.52]
// final result is [0.48, 0.52]:[‘a’, ‘b’]
```

Since integers are implicitly converted to `float`s in `float op int` or `int op float` expressions, `probType OP int` expressions are implicitly converted expressions of `probType OP float`.

Normalization

When a probability type is first initialized, or whenever a probability type is transformed, its internal probability list is normalized to ensure it represents a valid distribution. There are three steps in the normalization process.

1. Check both the probability list and the value list are non-empty. If this check fails, throw an error.
2. Check the probability list contains only non-negative elements, and that the sum of the elements > 0. If this check fails, throw an error.
3. Divide each element by the sum

Expressions

Function Calls

A function call is an expression that is used to call a defined function, also called a function designator. The function name is followed by parentheses and then the type and name of the required parameters which are separated by comma, if there are multiple parameters. The parameters should match the type of the data which is defined during the function declaration. The type of the function call entire expression is the type of the function's return type specified during declaration, or `void` if the function doesn't have one.

Example of valid function call expressions:

```
f_id ()
g_id (type arg1_id, type arg2_id)
```

Here f_id, g_id, arg1_id and arg2_id are all identifiers.

Unary Operators

The unary operators are `~` and `!`.

Unary expressions group left to right and are of two possible forms:

```
~ expression   // bitwise not; flip all the bits
! expression   // logical not; non zero values become 0, zeroes become 1
```

Binary Operators

There are four basic categories of binary operators, as well as concatenation.

Arithmetic: `+`, `-`, `*`, `/`

Bitwise: `&`, `|`, `^`, `<<`, `>>`

Comparison: `<`, `>`, `<=`, `>=`, `==`, and `!=`

Logical: `||`, `&&`

These expressions group left to right and follow the general form

```
expression binop expression
// where binop is a binary operator and
// the expressions on either side of the operation are of the same type
```

The only exceptions to the same type rule requirement are arithmetic operations between floats and ints (in which ints are implicitly cast to floats), concatenation operator between a string and float, string and int, or string and char (where floats, ints, and chars are implicitly cast to strings), and right and left shifts on lists for deleting elements, which require an argument of type int on the right hand side.

Type	Binary Operators Supported
`int`	`~`, `+`, `-`, `*`, `/`, `&`, `|`, `^`, `>>`, `<<`, `<`, `>`, `>=`, `<=`, `==`, `!=`
`float`	`+`, `-`, `*`, `/`, `<`, `>`, `>=`, `<=`, `==`, `!=`
`bool`	`==`, `!=`, `&&`, `||`
`char`	`==`, `!=`
`list <type>`	`==`, `!=`, `<<`, `>>`, `++`
`prob <type>`	`==`, `!=`, `+`, `-`, `*`, `/`

Note that `==` and `!=` on lists and probability types enacts a "deep comparison". Two probability types are equal if their probability lists and value lists are equal. Lists are equal if they contain the same values in the same order.

Assignment Operator And Variations

The assignment operator takes the value on its right hand side and binds it to the identifier on the left.

```
identifier = expression
x = “hey” //the string literal “hey” gets bound to the name x
```

Arithmetic expressions can be combined with the assignment operator to create four variations: `+=`, `-=`, `*=`, and `/=`.

```
id OP= expr  // syntactic sugar for  id = id OP expr
x += 5       // syntactic sugar for x = x + 5
x -= 5       // syntactic sugar for x = x - 5
x *= 5       // syntactic sugar for x = x * 5
x /= 5       // syntactic sugar for x = x / 5
```

Speciality Operators For Strings and Lists

Left And Right Element Shifts

Two ternary operators are supported specifically for string manipulation, `<< :` and `>> :`, left and right element shifts respectively.

These ternary expressions group left to right and follow the form:

```
expr1 << expr2 : expr3
expr1 >> expr2 : expr3
// where expr1 is an expression of type string
//       expr2 is an expression of type int
//       expr3 is an expression of type char
```

In the left byte shift, `expr1` is shifted to the left by `expr2` bytes. The newly created space is filled with characters of value `expr3`. Similarly, in the right shift, `expr1` is shifted right by `expr2` number of bytes, and the newly created space is filled in with copies of `char` `expr3`.

Since strings are lists, there exists a more general form of this operator on any list:

```
expr1 << expr2 : expr3
expr1 >> expr2 : expr3
// where expr1 is an expression of type list <type>
//       expr2 is an expression of type int
//       expr3 is an expression of type <type>
```

End Deletion Operator

`<<` and `>>` are binary operators that allow deletions from the beginning or end of a list (see *Lists* for usage and examples).

Overlap Operator

The overlap operator, `^^ : ,...,` is designed specifically for string manipulation and takes in three or more expressions, with expressions after the third separated by commas. It groups left to right and follows the form:

```
expr1 ^^ expr2 : expr3 //minimum number of expressions needed
expr1 ^^ expr2 : expr3, expr4,...,exprn
// where expr1 is an expression of type string
//       expr2 is an expression of type string
//       expr3 is an expression of type char
//       any following expressions are optional and of type char
```

The overlap operator treats strings as “layers” placed on top of each other, and uses the third argument and onward to determine which parts of the layers are treated as transparent. Based on its provided definition of “transparent” characters, overlap overlays `expr1` on top of `expr2`, resulting in a new string containing characters from both `expr1` and `expr2`. An example for clarity:

```
“---O_O---” ^^ “.........” : “-” // yields “...O_O...”
```

`expr1` and `expr2` do not need to be of equal length. Overlap will stop overlapping characters when either or both of the strings end.

Since strings are lists, there exists a more general form of this operator on any list:

```
expr1 ^^ expr2 : expr3 //minimum number of expressions needed
expr1 ^^ expr2 : expr3, expr4,...,exprn
// where expr1 is an expression of type list <type>
//       expr2 is an expression of type list <type>
//       expr3 is an expression of type <type>
//       any following expressions are optional and of type <type>
```

Declaration And Initialization

Type Declaration

Five type specifiers: `int`, `bool`, `float`, `string`, and `char`

Two type qualifiers: `prob` and `list`

Types are declared using the following format:

```
<qualifier>* <type-specifier> identifier; // where * means 0 or more
```

Note that the type of these declarations are distinct:

```
prob list int specialList; // one of a group of possible lists

list prob int listOfSpecialNums; /* a list where each elt is one of a group of possible integers */

list prob list list int num;   /* a list of where each elt is one of a group of possible lists of int lists */

prob list list int num;    // one of a group of possible lists of int lists
```

Type Initialization

Types are initialized after declaration using the assignment operator. Initialization can occur immediately after declaration,

```
<qualifier>* <type-specifier> identifier = initializer;
```

or some time later in a separate statement:

```
<qualifier>* <type-specifier> identifier; //declared here
...
identifier = initializer;                 //initialized here
```

Initializers for `bool`, `int`, `float`, and `char` are expressions that evaluate to `bool`, `int`, `float`, and `char` literals respectively.

Function Declarators And Definitions

The function declarator consists of the `fn` keyword followed by the function signature. The `fn` keyword signifies a function declaration or definition. The function signature lists the return type (`void` keyword is used if no return type) , name of the function, and a list of parameters inside parentheses (if the function has 0 arguments, an empty pair of parentheses are used).

The function definition consists of the `fn` keyword, function signature, and function body. The latter is a block of statements denoted with curly braces.

A Note On Blocks: *Curly braces surrounding a group of statements denotes a "block" of statements. Entering a block signals entrance into a new, more local scope. Contents like bindings (identifiers and their values) declared inside a block are not visible outside the block. Statements inside a block are executed in sequence.*

To declare a function without immediately specifying its definition, write out the fn keyword, then the type signature followed by a semicolon.

```
fn return-type name (type-specifier arg1_id, ... type-specifier argn_id)
```

To declare and define a function or to define a function declared earlier, write the type signature followed by the body of the function enclosed in curly braces.

```
fn return-type name (type-specifier arg1_id, ... type-specifier argn_id){
    statement;
    …
    statement;
    return identifier; //where identifier is of type return-type
}
```

Statements

A statement can be broken into three types:

1. `expression-statement`
2. `selection-statement`
3. `iteration statement`

The special control flow statements are `return`, `continue`, and `break`.

`return` exits the function scope and returns a value if supplied.

`continue` jumps to the top of an iteration statement.

`break` exits an iteration statement.

Expression Statements

The most common kind of statements are expression statements. Expression statements consist of an expression followed by a semicolon.

```
expr;      //general format
int x = 3; //declaration and initialization statement (expr here is int x)
```

Selection Statements

Selection statements choose one of multiple execution paths based on a condition

`if`-`else` statement format:

```
if (expr) {
     // this block only executes if expr evaluates to true
}
else{
    // this block only executes if expr evaluates to false
}
```

`if`-`elif`-`else` statement format:

```
if (expr) {
     // this block only executes if expr evaluates to true
}
elif(expr2){ //expr2 is only evaluated if expr evaluates to false
     // this block only executes if expr2 evaluates to true
}
else{
    // this block only executes if none of the other blocks executed
}
```

Any number of `elif` blocks are allowed for each `if`-`else` pair. Lone `if` statements (without an attached `else` or `elif` block) are also permitted.

Iteration Statements

Iteration statements consist of a block of statements executed in a loop until a condition is met.

`while` loop format:

```
while(condition){
  //statements
}
```

`for` loop format:

```
for(scoped variable assignment; condition; statement){
//statements
}
```

`switch` loop format:

```
switch(val){
  case val1:
    // statements here only execute if
    // val == val1
    break; //exit switch
  case val2:
    // statements here only execute if
    // val == val2
    break;
  ...
  case valn:
    // statements here only execute if
    // val == valn
    break;
  default:
    // statements here only execute if
    // default case is checked
```

```
        break;
}
```

`switch` must contain at least the `default` case. Without the `break` statement, execution flow continues down to the next case statement.

Grammar

We define the grammar using productions and rules in the `parser.mly`. The parser can be found here: https://github.com/magidandrew/smap/blob/iteration2/parser.mly.