Strlang language Reference Manual

strlang is a simple imperative programming language, designed specifically to make manipulating text strings easy. The language features a minimalistic syntax and includes support for a variety of string-oriented operations including matching and substituting regular expressions. While taking syntactic cues from Perl and other more traditional text processing languages, strlang is a compiled language and offers native performance and full static semantic checking.

3.1 Lexical conventions

strlang has five types of tokens: names, number constants, string constants, operators, and punctuation. The language does not have any keywords.

Names are any sequence of one or more alphabetic (upper and lower-case), numeric or _ characters. The first character must be alphabetic. Names are case sensitive. A small number of names are reserved for use as built-in functions and may not be used for other purposes: **`open`**, **`read`**, **`end_input`**, **`write`**, **`to_num`**, **`to_string`** and **`exit`**.

A number constant is any sequence of one or more integer characters. As a practical matter, integer constants are limited to 2^31.

A string constant is a sequence of characters surrounded by double quotes. C-style escape characters are used.

Operators and punctuation include: **`( ) { } [ ] + - * / % | & < > <= >= == != ! ^ ~ @@ @% <- -> $ # %`** and **`;`**.

Whitespace, meaning spaces, tabs, carriage-returns and newlines, is ignored.

Comments are begun with the characters // and run to the end of the line. Comments are ignored.

3.2 Types

strlang has two fundamental types of variables: strings and numbers. Strings contain text, whereas numbers contain integer quantities. In addition to the basic types, the language includes the map type for associating key-value pairs. strlang is a strongly typed language, and all conversions between different types must be made explicitly.

Strings **`$`**

Text strings are the bread and butter of strlang. Strings are sequences of ASCII character values. Regular expressions are no different than other strings, and are interpreted only in the context of searching and replacement. They use the extended Perl syntax. String operators include concatenation, substring creation, searching, matching and replacing.

Numbers `#`

Number variables are included to allow for integer and boolean arithmetic. Numbers are signed 32-bit integer quantities. They support the five standard integer arithmetic operators, as well as comparison, and boolean connectors. In a boolean context, the value zero is treated as false, and all other values are treated as true.

Maps `%[t1 ; t2]`

Maps are the only aggregate type in strlang. Maps are sets of key-value pairs. Only a single value may be associated with each key. When a map variable is defined, the types of its keys and values must also be defined. Keys and values may be strings or numbers, but must be homogeneous. Accordingly, there are four types of maps: number-to-number, number-to-string, string-to-number and string-to-string.

Map operations include insertion of a key-value pair, lookup of a value by key, deletion of a key-value pair by key and two special operators used to extract all the values or keys in a map to be used as values in a new map with number keys. Note that maps with differing key or value types are considered to be of different types.

None `^`

Certain expressions have no value associated with them. Their type is consequently considered to be 'none'. Variables of this type cannot be explicitly created.

Type	Name	Notes
`$`	String	Text strings.
`#`	Number	Signed integer quantities.
`%[k;v]`	Map (k-to-v)	Set of key value pairs. k and v can be either `$` or `#`. The map has keys of type k and values of type v.
`^`	None	Used to indicate functions with no parameters, or no return value.

3.3 Expressions

Operator Precedence and Associativity

Precedence and associativity of the various operators in strlang is given below, ordered from lowest to highest precedence.

Operator	Associativity	Notes
`<-`	Right to Left	Assignment. Requires identical type operands (no implicit conversion).
`|`	Left to Right	Logical or `|`. No short-circuit evaluation.

`&`	Left to Right	Logical and `&`. No short-circuit evaluation.
`== !=`	Left to Right	Structural equality `==` and inequality `!=`.
`< > <= >=`	Left to Right	Numeric comparison for numbers, lexicographic comparison for strings.
`+ -`	Left to Right	Addition `+` and subtraction `-` for numbers, concatenation `+` and substring `-` for strings, deletion `-` for maps.
`* / %`	Left to Right	Multiplication `*`, division `/` and modulus `%` for numbers, match `/` and index `%` for strings.
`~~`	Left to Right	Replacement for strings (ternary operator).
`- ! ^`	Left to Right	Arithmetic `-` and logical negation `!` for numbers, length `^` for strings and maps.
`[]`	Left to Right	Accessor for maps.
`@% @@`	Right to Left	Keys `@%` or values `@@` for maps.

3.3.1 String Expressions

String Constants `"[^\"]*"`

Returns the string containing the text between the double-quote symbols.

```
a <- "str const"; // string variable 'a' now contains 'str const'
```

Concatenation `$ '+' $`

Returns the concatenation of the two string operands.

```
a <- "hello " + "world"; // 'a' now contains 'hello world'
```

Substring `$ '-' #`

Returns a substring of the first operand. If the second operand is non-negative, these are the characters of the first operand starting at position n (n being the second operand). Otherwise, these are the characters of the first operand except for the last |n|.

```
a <- "hello world" - 6; // 'a' now contains 'world'
a <- "hello world" - -6; // 'a' now contains 'hello'
```

Match `$ '/' $`

Return the first substring in the first operand that matches the second operand. If there is no match, an empty string is returned. The second operand will be interpreted as a regular expression, so special symbols will need to be escaped if they are to be interpreted as literals.

```
a <- "hello world." / "world"; // 'a' now contains 'world'
a <- "hello world." / "h[e-l]*o"; // 'a' now contains 'hello'
a <- "hello world." / "."; // 'a' now contains 'h'
```

```
a <- "hello world." / "\\."; // 'a' now contains '.'
```

Index `$ '%' $`

Return the position in the first operand of the first match for the second operand. If there is no match, the value -1 is returned. As with match, the second operand will be interpreted as a regular expression.

```
i <- "hello world" % "world"; // 'i' now contains '6'
i <- "hello world" % "h[e-l]*o"; // 'i' now contains '0'
i <- "hello world" % "world."; // 'i' now contains '-1'
```

Replace `$ '~' $ '~' $`

Return a string consisting of the first operand, with all occurrences of the second operand replaced by the third operand. The second operand will be interpreted as a regular expression.

```
a <- "hello world" ~ "[eo]" ~ "x"; // 'a' now contains 'hxllx wxrld'
a <- "hello world" ~ "world" ~ ""; // 'a' now contains 'hello '
```

Length `'^' $`

Return the length of the operand - the number of ASCII characters in the string.

```
i <- ^ "hello world"; // 'i' now contains '11'
```

Comparison `$ '<' $` or `$ '>' $` or `$ '<=' $` or `$ '>' $`

Return 1 if the first operand is lexicographically less than, greater than, less than/equal or greater than/equal to the second operand. Otherwise return 0.

```
i <- "hello" < "helloo"; // 'i' now contains '1'
i <- "hello" > "helloo"; // 'i' now contains '0'
i <- "hello" >= "hello"; // 'i' now contains '1'
```

3.3.2 Number Expressions

Number Constants `[0-9]+`

Returns a number containing the integer value in the expression.

```
i <- 7; // 'i' now contains '7'
```

Addition `# '+' #`

Returns the sum of the two operands.

```
i <- 7 + 4; // 'i' now contains '10'
```

Subtraction `# '-' #`

Returns the difference of the two numbers.

```
i <- 7 - 3; // 'i' now contains '4'
```

Multiplication `# '*' #`

Returns the product of the two numbers.

```
i <- 7 * 3; // 'i' now contains '21'
```

Division `# '/' #`

Returns the whole-number quotient of the two numbers. If the value of the second operand is 0, a runtime error will occur.

```
i <- 7 / 3; // 'i' now contains '2'
i <- 7 / 0; // runtime error
```

Modulus `# '%' #`

Returns the remainder of the two numbers. If the value of the second operand is 0, a runtime error will occur.

```
i <- 7 % 3; // 'i' now contains '1'
i <- 7 % 0; // runtime error
```

Boolean Connectors `# '|' #` or `# '&' #`

Returns the logical disjunction or conjunction of the two operands. Evaluation is not short-circuited, meaning both operands are always evaluated.

```
i <- 0 | 1; // 'i' now contains '1'
i <- 1 & 0; // 'i' now contains '0'
i <- 0 & (0 / 0); // runtime error, even though 0 is first operand
```

Comparison `# '<' #` or `# '>' #` or `# '<=' #` or `# '>=' #`

3.3.3 Map Expressions

Accessor `% '[' $ ']'` or `% '[' # ']'`

Return the value for the key given as the second operand, in the map given as the first operand. The type of the second operand must match the type of the map's keys. If the

key is not found, zero or the empty string are returned, depending upon whether the map's values are numbers or strings.

```
s <- ms["hi"]; // 's' now contains the value associated with 'hi' in 'ms'
i <- mn[3]; // 'i' now contains the value associated with '3' in 'mn'
```

Deletion `% '-' $` or `% '-' #`

Delete the key-value pair associated with the key given as the second operand and the map given as the first operand. Return that value.

```
s <- ms - "hi";
// 's' has same value as above example, but 'ms' no longer contains
// the given pair
```

Emptying `% '<-' '0'`

Empty the map given as operand of all key-value pairs, and return the same empty map.

```
empty <- ms <- 0; // 'ms' and 'empty' are now both maps with no contents
```

Length `'^' %`

Return the number of key-value pairs in the map.

```
i <- ^m; // 'i' contains the number of keys in m
```

Keys `'@%' %`

Return a new map containing the keys of the map given as the operand. The new map's values are the keys of the old map. The new map's keys are numbers, zero through the size of the map.

```
keys <- @% m; // 'keys' contains the keys of 'm' as its values
```

Values `'@@' %`

Return a new map containing the values of the map given as the operand. The new map's values are the values of the old map. The new map's keys are numbers, zero through the size of the map.

```
vals <- %% m; // 'vals' contains the values of 'm' as its values
```

3.3.4 General Expressions

Lvalues `name`

Lvalues are simply variables. They can both be read from and assigned to. They return the value stored in the given variable. As there are no implicit type conversions, the type of the return value is simply the same as the type of the variable.

Equality/Inequality `expr '==' expr` or `expr '!=' expr`

Return the number 1 if the two expressions are structurally equal, and 0 otherwise. Strings, numbers and maps may be compared, but both operands must be of the same type.

```
i <- 1 == 1; // 'i' contains 1
i <- "s" == "S"; // 'i' contains 0
```

Assignment `lvalue '<-' expr`

Returns the value of the second operand, and also stores that value into the first operand. Accordingly, the types of the first and second operands must be the same.

```
j <- i <- 1 == 1; // 'i' and 'j' contains 1, if both i and j are # vars
```

Function Calls `name '(' expr1 ';' expr2 ';' ... ')'`

Function calls evaluated as follows: first all of the arguments (the expressions within parenthesis) are evaluated, left-to-right. Then the function code is called and executed. The return value is precisely the value returned by the function. In the case that the function's return type is none, no value is returned.

Unlike variables, functions need not necessarily be declared prior to use, but the name must have a matching declaration somewhere in the program. Moreover, the number and types of the expressions given as arguments to the function must match with the number and types of the parameters given in that function's declaration.

Parameters are passed by reference.

3.4 Statements

Statements comprise the bulk of a strlang program. Statements are executed in sequence. In addition to simple expressions, statements include loop and conditional control structures, logical code blocks, and return statements. Each type of statement is described in detail below.

Variable Declarations `type name ';'`

All variables must be declared prior to use. The declaration associates a type to a given name. The type may be any of the types given above, save for `^` (none). Unlike other types of statements, variable declarations may only appear at the beginning of a block, or

at the beginning of a program before any other code. Declarations not within any block are considered global. Declarations within a block are considered local to that block.

All variables are initialized automatically. Number variables are set to zero by default, strings to the empty string, and maps to a map containing no keys.

Expressions `expr ';'`

Expression statements are evaluated according to the rules described in the previous section.

Return `'->' expr`$_{opt}$ `';'`

Return statements are used to stop execution of the current function and return control to the calling function, at the place where the current function was called. The associated expression is evaluated, and its value is given back to the caller. The type of the expression must match the return type of the function. Functions with return type of 'none' should omit the expression portion of the statement.

Blocks `'{' decl_list`$_{opt}$ `stmt_list`$_{opt}$ `'}'`

Blocks are logical units of code. Blocks consist of grouping markers (curly braces) around an optional list of variable declarations followed by an optional list of statements. Variables declared within a block are not valid outside that block. Two variables of the same name may not be declared in the same scope. Within a block, references to a name are bound to the variable declared in the nearest enclosing scope.

Conditionals `'[' expr ']' block` or `'[' expr ']' block '![]' block`

In a conditional statement, the expression is first evaluated and if the result is non-zero, the block immediately after is executed. The expression must be of type number. If the result of evaluating the expression is zero and the conditional uses the second form, the second block is executed.

Loops `'<' expr '>' block`

In a loop statement, the expression is first evaluated. If the result is non-zero, the following block is executed and then control moves back to the top of the loop, where the process repeats. The expression for a loop must be of type number.

3.5 Functions

Functions `name '(' decl_list ')' '->' type block` or
`name '(' '^' ')' '->' type block`

Functions in strlang consist of a signature and a block. The signature is the name, declaration list, and type. The declaration list indicates the names and types of the input

parameters for the function (or `^` for none). The type indicates the kind of value that will be returned to the caller. Finally, the block contains the actual code of the function.

Function names must be unique within a program. Variables also may not use the same names as functions. The scope of a function is the entire program.

Variables declared in the input parameter section have the same scope as variables declared at the top of the function's code block.

A functions with a return type other than `^` and lacking an explicit return statement will automatically return zero, the empty string, or an empty map, in accordance with the return type given in its signature.

3.6 Program Structure

Program `decl_list`$_{opt}$ `func_list`

A strlang program consists of an optional list of variable declarations, followed by a list functions. Variables declared at the program level have program-wide scope.

Functions need not be declared in any particular order in a strlang program. However, every strlang program is required to contain a function by the name of main with the following signature:

```
main(^) -> ^
```

Execution of a strlang program begins with the first statement in the code block of this main function. The program is then executed statement-by-statement, as described above.

3.7 Built-in Functions

strlang provides a few built-in functions to facilitate input and output and conversion between different types.

Signature	Description
`open($ io_type;` `  $ filename) -> ^`	open() opens an input or output stream for subsequent reads or writes. If io_type matches "in", it will attempt to open filename for reading and use that file for subsequent read() calls. If filename is "stdin", it will use standard input. If io_type match "out", it will attempt to open filename for writing and use it for subsequent write() calls. If filename is "stdout", it will use standard output.
`read(^) -> $`	read() gets the next line of input from the current input

	stream, and returns it as a string.
`end_input(^) -> #`	end_input returns 1 if the input stream has reached the end of input, and 0 otherwise.
`write($ outstr) -> ^`	write() prints the string outstr to the current output stream.
`to_num($ str) -> #`	to_num() converts a string to a number, which it returns.
`to_string(# num) -> $`	to_string() converts a number to string, which it returns.
`exit(^) -> #`	exit() terminates the current program, returning the given number value to the calling environment.

3.8 Syntax Summary

```
program:
       decl_list func_list func

decl_list:
       /* empty */
       decl_list decl ;

func_list:
       /* empty */
       func_list func

func:
       name ( formals_list_opt ) -> ret_type block



decl:
       type name

type:
       $             /* string */
       #             /* number */
       %[$; $]       /* map from string to string */
       %[$; #]       /* map from string to number */
       %[#; $]       /* map from number to string */
       %[#; #]       /* map from number to number */

formals_list_opt:
       ^             /* void - empty argument list */
       formals_list

formals_list:
       decl
       decl ; formals_list

ret_type:
       ^             /* void - no return value */
       type

block:
       { decl_list stmt_list }

stmt_list:
       /* empty */
       stmt stmt_list

stmt:
```

```
        block                       /* code block */
        expr ;                      /* single expression */
        < expr > block              /* while(expr) block */
        [ expr ] block              /* if(expr) block
        [ expr ] block ![] block    /* if(expr) block else block */
        -> ;                        /* return; */
        -> expr;                    /* return expr; */

expr:
        expr binop expr             /* binary operation */
        unop expr                   /* unary operation */
        expr ~ expr ~ expr          /* search/replace */
        lvalue <- expr              /* assignment */
        ( expr )                    /* grouping */
        name ( actuals_list_opt )   /* function call */
        lvalue                      /* variable */
        number_literal              /* number literal: [0-9]+ */
        string_literal              /* string literal: "[^\"]*" */

lvalue:
        name                        /* variable */
        name [ expr ]               /* map accessor variable */

binop:        /* any of the following */
        + - * / % == != < > <= >= | &

unop:         /* any of the following */
        - ! ^ @@ @%
```