Signal Reference Manual

Isabelle Friedfeld-Gebaide		Serdar Mamadzhanov
if2266@columbia.edu		sm5038@columbia.edu
Manager, Tester	System Architect, Tester	Language Guru, Tester

1. Introduction

Signal is an imperative language designed as an intermediate language for new users. Signal is intended as a second language to bridge the gap between typical introductory programming languages like Python and more complicated languages like Java and C.

2. Lexical Conventions

Signal programs are written in ASCII with each line terminating in `;` .

There are five kinds of tokens: identifiers, keywords, literals, expression operators, and delimiters. In general blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens. At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator-pairs.

2.1. Comments

The characters `/*` introduces a comment, and terminate with `*/` characters. Comments can be multiple lines. Comments are ignored by the syntax.

2.2. Identifiers

Identifiers (also referred to as names) are described as a sequence of letters and digits; the first character must be alphabetic. Within the ASCII range (U+0001..U+007F), the valid characters for identifiers are: the uppercase and lowercase letters A through Z, the underscore _ and, the digits 0 through 9.

Upper and lower case letters are considered different.

```
let letter          = ['a'-'z' 'A'-'Z']
let digit           =  ['0'-'9']
let identifier      =  letter (digit | letter | '_')*
```

2.3. Keywords

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

```
bool                                int
char                                list
do                                  return
else                                string
false                               true
float                               var
for                                 void
func                                while
if
```

2.4. Literals

Literals are notations for constant values of some primitive types.The five literal types are: integer literal, floating point literal, character literal, string literal, and boolean literal.

2.4.1. Integer Literal

Integer literals are described by the following lexical definitions:

```
  let nonzerodigit    =  ['1'-'9']
  let bindigit        =  ('0' | '1')
  let octdigit        =  ['0'-'7']
  let hexdigit        =  digit | ['a'-'f'] | ['A'-'F']

  let decinteger      =  digit*|nonzerodigit(['_']?digit)*
  let bininteger      =  ['0']('b' | 'B')(['_']bindigit)+
  let octinteger      =  ['0']('o' | 'O')(['_']octdigit)+
  let hexinteger      =  ['0']('x' | 'X')(['_']hexdigit)+
  let integer         =  decinteger | bininteger | octinteger
| hexinteger
```

Underscores are ignored for determining the numeric value of the literal. They can be used to group digits for enhanced readability. One underscore can occur between digits, and after base specifiers like 0x. Note that leading zeros in a non-zero decimal number are not allowed.

An integer literal is always of type int.

2.4.2. Floating Point Literal

Floating point literals are described by the following lexical definitions:

```
  let digitpart       =  digit*|digit(['_']?digit)*
  let exponent        =  ('e' | 'E')('+'|'-')?digitpart
  let pointfloat      =  digitpart['.'](digitpart)?|
(digitpart)?['.']digitpart
  let exponentfloat =  (digitpart|pointfloat)exponent
  let float           =  pointfloat | exponentfloat
```

As in integer literals, underscores are supported for digit grouping. A floating point literal is always of type float.

2.4.3. Character Literal

A character literal is expressed as a character, enclosed in ASCII single quotes(The single-quote, or apostrophe, character is \u0027) .

```
  let character       = ['a'-'z' 'A'-'Z' '0'-'9' ' ' '\t' '\r'
'\n' '!' '#' '$' '%' '&' '(' ')' '*' '+' ',' '-' '.' '/' ':'
';' '<' '=' '>' '?' '@' '[' '\\' ']' '^' '_' '`' '{' '|' '}'
'~']
  let char = '\'' character '\''
```

A character literal is always of type char.

2.4.4. String Literal

A string literal consists of zero or more characters enclosed in ASCII double quotes (The single-quote, or apostrophe, character is \U+0022) .

```
let string = '"' (character)* '"'
```

A string literal is always of type string.

2.4.5. Boolean Literal

The boolean type has two values, represented by the boolean literals true and false, formed from ASCII letters. Described by the following lexical definition.

```
| "true"   { BLIT(true)  }
| "false"  { BLIT(false) }
```

A boolean literal is always of type bool.

2.5. Expression Operators

Operators are specific lexical elements reserved for use by the language and may not be used otherwise. Refer to the Statements and Expressions section for functionality and use cases. The following tokens, separated whitespace, are operators:

```
+ - * ** / // %& || ^ ~ < > <= >= == != && || !
```

2.6. Delimiters

Delimiters are specific lexical elements reserved for use by the language and may not be used otherwise. They are responsible for denoting the separation between tokens. White space is considered a separator. The following tokens, separated whitespace, are delimiters in Signal.

```
( ) { } [ ] | ; : , .
```

3. Types

3.1. Primitive data types

3.1.1. int

The integer type variables contain whole, non-fractional values in 4 bytes.

```
var int x;
x = 5;
var int y;
y = 2;
```

3.1.2. float

The float type variables contain floating point numbers in 4 bytes.

```
var float y;
y = 3.7;
var float x;
x = .2;
y = 4.;
```

3.1.3. bool

The bool type variables can have either true or false values.

```
var bool x;
x = true;
var bool y;
y = false;
```

3.1.4. void

The void type cannot be used as a variable type but it can be used as the type for functions which do not return any value.

```
func void print_hello() {
   print("Hello World!");
}

func int add_five(var int x) {
   var int y;
   y = 5;
   return y + x;
}
```

3.1.5. char

The char type variable contains a single ASCII character surrounded by a single quotation mark or by its decimal representation according to the ASCII table. The value is stored in 1 byte memory.

```
var char a;
a = 'z';

var char b;
b = 'b';
```

3.1.6. str

The str type contains a sequence of ASCII characters as a literal constant surrounded by double quotation marks and the last character is always a null character.

```
var string name;
name = "Lee Bollinger";
```

3.2. Non-primitive Data Types

3.2.1. Lists

The lists type variables represent a fixed size collection of elements, each selected by a single index that can be computed at run time during the execution of the program. lists start indexing at 0.

```
var list int[5] numbers;
numbers = [6, 7, 8, 9, 0];
var list string[2] fullName;
fullName = ["Lee", "Bollinger"];
```

4. Operators

4.1. Arithmetic

The usual binary arithmetic operators are:

```
+ - * / %
```

The types of expressions must match in order for the arithmetic binary operations to be evaluated.

The binary operators `* / %` have the same precedence which is higher than the precedence of `+ -` . These operators are associated from left to right.

```
var int x;
x = (10 + 5) / 3;
```

4.2. Assignment

The equal binary operator `=` assigns an expression to a variable and it is right associative.

```
var bool b;
b = false;
```

4.3. Equivalence

The relation operators are:

```
< > <= >= == !=
```

The `==` and `!=` have the same precedence, which is lower than `< > <= >=`

All relational operators have lower precedence than arithmetic operators.

```
var bool y = (1 <= 3);    /* true */
```

4.4. Logical

The Signal language has three logical operators. They are: `&&, ||, !` which are used as `and`, `or`, and `not` respectively.

```
var bool bad;
bad = false;
var bool clean;
clean = true;
if (!bad && clean) do {
      statement;
}
```

5. Statements and Expressions

5.1. Variables

Variables are declared with the keyword var. Variables are usable since declaration until their reference count drops to 0 and is garbage collected. Signal accepts the following literal expressions: integer, floating point, character, string, and boolean.

List expressions are also allowed such as:

```
var list int[5] arr;
arr = [1,2,3,4,5];
```

5.2. Functions

Functions are a sequence of operations enclosed in a specific namespace. Functions require type definition of the return value. Parameters are optional and type. To declare a function use the keyword func.

```
func void print_hello() {
   prints("Hello World!");
}

func int addFour(var int x) {
   var int addFour;
   addFour = x + 4;
   return addFour;
}
```

6. Control Flow

The statements inside source files are generally executed from top to bottom, in the order that they appear. Control flow statements, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code. This section describes the decision-making statements (if-else), and the looping statements (for, while) supported by Signal.

6.1. For Loops

The for statement allows the programmer the iterate over a range of values

```
var int i
for (i = 1; i <= 10; i = i + 1) do {
     printi(i);
}
```

6.2. While Loops

The while statement executes a user-defined block of statements as long as a particular conditional expression evaluates to true

```
while ((x < 8) == true) do {
     x = x + y;
}
```

6.3. Branching Statements

```
if (bool == true) do {
     printb(bool);
```

```
} else do {
      x = 7;
}
```

7. Scope Rules

Scope refers to which variables and functions are available at any given time in the program. All functions are available to all other functions regardless of their relative position in a program or library.

7.1. Lexical Scope

The lexical scope of names declared in external definitions extends from their definition through the end of the file in which they appear. Variable scope falls into two categories: global variables, which are defined outside of any function, and local variables, which are defined within a function.

The lexical scope of local variables declared at the head of functions (either as formal parameters or in the declarations heading the statements constituting the function itself) is the body of the function. The lexical scope of global variables are any piece of code following the declaration.

It is an error to redeclare identifiers already declared in the current context, unless the new declaration specifies the same type and storage class as already possessed by the identifiers.

8. Built-In functions

Signal will provide the following out-of-the-box functions: print.

8.1. print functions

There are 5 different print functions that can be used: `printi` , `printb` , `printc` , `prints` , and `printf` . Each print function is for a literal type `printi` for integers, `printb` for booleans, `printc` for char, `prints` for strings , and `printf` for floats.

```
var int x;
x = 5;
printi(x); /* will print “5” */
var string y;
y = “Signal is cool”;
prints(y); /* will print “Signal is cool” */
```

9. Language Features Not implemented

9.1. Len

Len was originally going to be a function obtaining the length of a list, but due to limited time we were unable implement it

9.2. Structs

Our group ultimately decided against implementing structs because we found it unnecessary with the ability for a user to gain the same functionality through the usage of functions.

9.3. Type

While our group had greatly hoped to be able to implement a getType function, due to the nature of how OCaml compiles and our usage of C functions in the `irgen.ml` it was decided against due to how it did not seem possible. Both C and OCaml do not offer this functionality out of the box and if you wanted to find the type of a C variable you would likely get the size and then compare it to known sizes of primitives, the issue with this approach is variables in our language can have the same size of each other so it is not a solution in Signal.

9.4. Garbage collection

Garbage collection was decided against in Signal due to the fact of us not giving the user the ability to dynamically allocate variables. Since that is a major feature of having an automatic garbage collection ability in a language we saw it as unnecessary to implement garbage collection in Signal.

10. References

10.1. The Java Language Specification:
https://docs.oracle.com/javase/specs/jls/se7/html/index.html

10.2. Rusty Language Reference Manual:
http://www.cs.columbia.edu/~sedwards/classes/2016/4115-fall/lrms/rusty.pdf

10.3. C Manual:
https://www.bell-labs.com/usr/dmr/www/cman.pdf

10.4. Crust Language
https://github.com/tianqizhao-louis/Crust

10.5. The Python Language Reference:
https://docs.python.org/3/reference/

10.6. Futhark Language Reference:
https://futhark.readthedocs.io/en/latest/language-reference.html

10.7. Dice Project Report:
http://www.cs.columbia.edu/~sedwards/classes/2015/4115-fall/reports/Dice.pdf