Swan Documentation

Swan is a dynamically-typed and interpreted programming language that was built using C. This report aims to help the reader understand the scope, implementation and user-interface of the language better.

Contents

0. Installation Guide

1. Firstly, clone the Swan github repository at https://github.com/shadhankkk/swan

   (You can do this by running the command

   git clone https://github.com/shadhankkk/swan

   on your terminal)

2. Enter the terminal, and change your working directory to /swan, then run the command

   (For Mac users) *make*
   (For Windows Users) *gcc src/*.c -o swan*
   *If you don't have gcc you can use any other compiler

3. Now, you can run .swan files via the following usage:

   (For Mac users) *./swan.out <filepath>/<filename>.swan*
   (For Windows Users) *.\swan <filepath>\<filename>.swan*

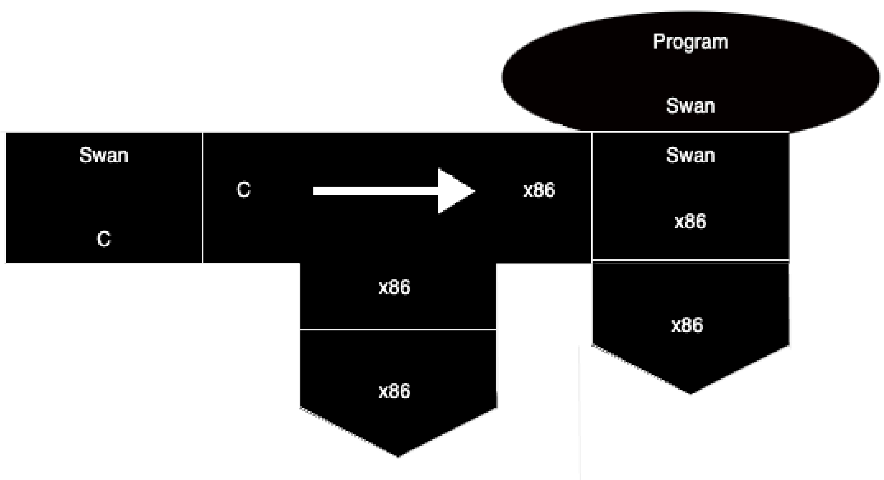
Note: The executable might get deleted by an anti-virus, in this instance it is required to give the file access / trust privileges.

1. The Language

1.1 Language Overview

Firstly, Swan is dynamically typed, meaning that there is no need to specify the types of variables or function parameters / return values.

Secondly, it is interpreted using C, so the interpreter is compiled (since C is a compiled language) and then it is used to execute the .swan file. The T-Diagram for this process is as described below.



1.2 Syntax (Backus-Naur)

program ::= block
block ::= {statement};
statement ::= | let
| assignment
| while-statement
| if-statement

| for-statement
| function-definition
| function-call
| ;
var ::= **var** name = expression
assignment ::= name = expression
return-statement ::= **return** expression
while-statement ::= **while**(expression) block
if-statement ::= **if**(expression) block
for-statement ::= **for**(var name = expression; expression, assignment) block
function-definition ::= **function** name(namelist) block
function-call ::= name(arglist)
expression ::= **false** | **true**
| number
| string
| name
| function-call
| binary-operator expression
| unary-operator expression
binary-operator ::= **+ | - | * | / | == | > | < | && | ||**
unary-operator ::= **- | !**

1.3 Data Types

1.3.1 Strings

A String can be instantiated using double quotation marks; for example, “Hello, World!” is a String literal. Strings are stored as char* values in the interpreting language. Currently, escape sequences are not supported.

1.3.2 Numbers

All Number-Literals are stored as long doubles in the interpreting language, and are instantiated the same way as in any other language. For Example, 132.53 is a number, and so is 23.

1.3.3 Arrays

Arrays are a list of data types, i.e a list of Strings and/or Numbers. Examples of Arrays include:

[1,2,3] or [“a”, “b”, “c”] or [“abc”, 123]

Arrays are stored as an array of Abstract Syntax Tree nodes in the interpreting Language

1.4 Variables

Variables are defined as follows:

```
var name = expression;
```

For example: var x = 3; or var x = “hello”;

Variables are assigned as follows:

```
name = expressions;
```

For example: x = 3; or x = “hello”;

1.5 Statements

1.5.1 For and While Loops

For-Loops are constructed as follows:

```
for(variable definition; expression; assignment)
{
    statements;
};
```

In the for-loop parenthesis, the first statement is a variable definition (e.g var i = 0), and the 2nd statement is a predicate expression (e.g i < 10) and the last statement is an assignment statement (e.g i = i + 1)

While-Loops are constructed as follows:

```
while(expression)
```

```
{
    statements;
};
```

In the while-loop parenthesis, the expression is a predicate expression (e.g true, false, x < 5, x == 0, etc.)

1.5.2 If-Else statements

If-Else statements are constructed as follows:

```
if(expression)
{
    statements;
}
else if(expression)
{
    statements;
}
else {
    statements;
}
```

Where the “else if” and “else” are optional, and all expressions are predicate expressions. Further-more, an indefinite number of “else if” statements can be used.

1.5.3 Function Definitions

Functions are defined as follows:

function name(namelist) block

Example:

```
function sum(x, y, z)
{
    return x + y + z;
};
```

1.6 Expressions

1.6.1 Arithmetic Operators

Expressions can just consist of basic data types, but they can also consist of arithmetic expressions, for example, (3 * 5 + 2 * 8) is an expression.

The **Binary** arithmetic operators supported are as follows:

operator: arguments => return value : syntax
+ : (a,b) => a + b : a + b
- : (a,b) => a - b : a - b
* : (a,b) => a x b : a * b
/ : (a,b) => a ÷ b : a / b

There is also one **Unary** arithmetic operator:

- : (a) => -a : -a

Furthermore, for the + operator, it supports the addition of strings with strings and numbers with strings. For example, “s” + 3 is evaluated as “s3”, or “s” + “d” is evaluated as “sd”, hence strings are concatenated and numbers are converted to strings when added with a string.

Additionally, the + and * operators are also overloaded for matrices (i.e 2 Dimensional arrays), 2 matrices can be added or multiplied as per normal matrix addition and multiplication rules.

1.6.2 Logical Operators

&&: (a,b) => a ^ b: a && b
||: (a,b) => a v b: a || b
! : (a) => ~a : !a

1.6.3 Relational Operators

> : (a,b) => a > b: a > b
< : (a,b) => a < b: a < b
== : (a,b) => a = b : a == b

1.6.4 Precedence of Operators

The precedence of Operators follows BODMAS closely, with the precedence ranking from highest to lowest being:

1. *, /
2. + , -
3. ==, >, <
4. &&

5. ||

1.6.5 Associativity of Operators

Currently, all operators are left-associative. Meaning that for some expression where operators of equal precedence, for example: 3 * 2 * 1, the first operator is called - meaning 3 * 2 * 1= ((3*2) * 1), hence all operators are left associative.

2. Built-In Functions

print(expr**) :** prints the evaluated expression argument onto the terminal

push(array, expr**) :** pushes the resulting value from evaluating expr into the array

append(array1, array2) **:** returns an array that is the result of appending the elements of array2 to the end of array1

length(array**) :** returns the length of the array

3. Test Cases

Here are 10 test cases that you may use to test the language:

Test Case 1:
print("Hello, World!");

Expected Result: “Hello, World!” printed on terminal

Test Case 2:

```
var s1 = "Hello";
var s2 = ", ";
var s3 = "World!";
print(s1 + s2 + s3);
```

Expected Result: “Hello, World!” printed on terminal

Test Case 3:

```
var y = 3;
var x = [1, y * y + y * y - y];

print(x[1]);
```

Expected Result: 15

Test Case 4:

```
function foo(x)
{
  print(x);
};
foo("Hello, World!");
```

Expected Result: “Hello, World!” printed on terminal

Test Case 5:

```
function foo_repeat(x, n)
{
  for(var i =0; i < n; i = i + 1)
  {
    print(x);
  };
};
foo_repeat("Hello, World!", 10);
```

Expected Result: “Hello, World!” printed on terminal 10 times

Test Case 6:

```
function fact(x)
{
  if(x == 1)
  {
    return 1;
  };

  return x * fact(x-1);
```

```
};

print(fact(5));
```

Expected Result: 120 printed on terminal

Test Case 7:

```
var arr = [];
for(var i = 0; i < 1000000; i = i + 1)
{
  push(arr, i);
};
print(length(arr));
print(arr[200000]);
```

Expected result: 1000000 and 200000 printed on terminal

Test Case 8:

```
var x = 0;

if(x < 0)
{
  print("negative");
}
else if(x == 0)
```

```
{
  print("zero");
}
else
{
  print("positive");
};
```

Expected result: “zero” printed on terminal

Test Case 9:

```
var a1 = [1,2,3];
var a2 = [6,7,8];
print(append(append(a1,a2), a1));
```

Expected result: [1,2,3,6,7,8,1,2,3] printed on terminal

Test Case 10:

```
  var x
=
[
  [5, 9, 10, 129, 99],
  [46, 23, 17, 66, 28],
  [35, 39, 88, 82, 76]
];
```

```
var y
=
[
  [3,12,56],
  [90, 72, 44],
  [53, 78, 0],
  [9, 2, 61],
  [420, 12, 2]
];

var z = x * y * x;

print(z);

function foo(a)
{
  a[1] = 5;
};

var a = [3,2];
foo(a);
print(a);
```

Expected Result:
[

[
661449 805323 1260222 6598954 5112124
]

[
529857 530343 897624 2887799 2225813
]

[
1027463 969667 1376266 6742021 5036547
]

]

[
3 2
]