Interpreted Scripting Language Specification

Jiří Škrobánek[1]

June 25, 2020, Ostrava

Introduction

This document specifies all the syntax and semantics of this interpreted language. Some trivial examples are provided to illustrate it.
The overall goal of this project was to create a simple scripting language to be used for user components in .NET applications. For example it is possible to implement behavior logic of some intelligent agents in this language. Author of the logic would not need to understand other concepts of .NET or C#.

Contents

[1]Faculty of Mathematics and Physics, Charles University, Prague, jiri@skrobanek.cz

1 Syntax Fundamentals

The program file should be encoded as ASCII without characters above 127. This is not necessary, but the parser is not guaranteed to function in other situations.
Whitespace characters do not modify the semantics of a program. There is an exception with reserved words (typeset in blue in this document), these must be separated by at least one whitespace character from other names, but not operators or parentheses.
Uppercase and lowercase letters are distinct characters, id est the language is case-sensitive.

1.1 Statements

Statements are the elementary parts of code, they specify actions to be executed. A statement is always terminated by the semicolon character (`;`).

1.2 Blocks

Blocks enclose sequences of statements. Blocks are enclosed in parentheses e.g.
`([STATEMENT1] [STATEMENT2])` is a block with two statements.
A block must always contain at least one statement. When no statement is required for the algorithm, depending on the context either return or continue statement can be used.

1.3 Names

Names are strings of ASCII characters of maximum length 8 characters. Names can contain Latin characters and are case sensitive.
Names hold a value within their scope of existence, this value can be used by putting its name in stead of its literal value. The scope of a name is its block and all descendant blocks. If the name is assigned outside any block, it can be accessed from anywhere.
If a name is used but its value is not set, it will default to type none.
Reserved words colored blue in this documentation cannot be used as names.

2 Data Types

Data types are types of objects in this programming language, data types can be assigned to names.

2.1 None

This type means that no value is held by a name or that no value is provided as a result of a function. This type only has one instance. Literally in code it is expressed as the word `none`.
None results in false when converted to logical value.

2.2 Integral Number

This type is 32-bit signed integer. Literally in code is represented in decimal base.
When converted to logical value, integer results in true if and only if it is not equal to 0.

2.3 Character

This type allows the representation of ASCII and high-ASCII characters. Literally in code it is represented as `#[CHARACTER]`, this representation should not be used with high-ASCII characters though.
When converted to logical value, character results in true if and only if it is not equal to ASCII value 0.

2.4 Function

Functions are composed of a block of code which can use names from the scope where defined. Upon being called, it also receives a number of values as arguments which are named by the names provided for the arguments.
Since function body is a block, it must contain at least one statement.
Functions calls may also be used as expressions. If a function contains a return statement, when such statement is executed, value of this expression will be set to this value. If no return is executed during the function execution and the function call is used as an expression, none will be set as value of this expression.
Functions result in true when converted to logical value.

2.5 List

List (vector) contains elements of any type, every element can have different type. Elements are indexed by nonnegative integers. Elements cannot be none.
When converted to logical value, a list results in true if and only if it is not empty.

2.6 Map

Map contains elements of any type, every element can have different type. Elements are indexed by any non-none values. Elements cannot be none.
When converted to logical value, a map results in true if and only if it is not empty.

3 Expressions and Operators

Expressions are parts of program that may be evaluated.

3.1 Simple expression

Simple expression directly represent a value, these are names, integer literals, character literals, none.

3.2 Expression composition

Expressions are composed by operators using the scheme:
`[EXPR1] [OPERATOR] [EXPR2]`.
Or for unary operators only
`[OPERATOR] [EXPR2]`.
Where `[EXPR1]` and `[EXPR2]` are expressions. `[OPERATOR]` is one of the following characters:
`+, -, *, /, &, |, >, <, !, ?`
Function call operation can also be used for expression composition. In that case the syntax is:
`[FN] ([EXPR1], [EXPR2], ...)`.
Where `[FN]` is an expression that evaluates to a function.

3.3 Operators

Operators ordered by priority, operators with the lowest priority are evaluated first. Operators of the same priority cannot be chained (making them unary). Priority can be overridden using parentheses.

Priority	Operators
0	Function call
100	`*, /`
200	`+, -`
300	`?, !, <, >`
400	`&, |`

Multiplication (*) is defined for integers only (as binary operator), should the result overflow, the lowest bytes are returned.
Division (/) is defined for integers only (as binary operator), the result is truncated. Should division by zero occur, none is returned.
Addition (+) is defined for integers only, both unary and binary, should the result overflow, the lowest bytes are returned.
Subtraction (−) is defined for integers only, both unary and binary, should the result overflow, the lowest bytes are returned.
Equality (?) is defined for any two value types (binary only). First the types of arguments is assessed, if the types differ, the values are not equal. Integers and characters are then assessed by value, none is equal to other none. Names representing a function, list, or map are equal if they represent the same object (determined by the call to initializer or function definition). If the two values are determined as equal, the integer 1 is returned, 0 otherwise.
Inequality (!) is always the negation of equality. 0 and 1 are switched.
Lesser than (<) is defined for integers only (as binary operator).
Greater than (>) is defined for integers only (as binary operator).
And (&) is defined for any two value types (binary only). 1 is returned if both values are convertible to true, 0 otherwise.
Or (|) is defined for any two value types (binary only). 1 is returned if at least one of the values is convertible to true, 0 otherwise.
If an operator is not applicable to argument types, none is returned. In particular, build-in functions cannot be arguments for any operator.

4 Statement patterns

All statements must adhere to one of the following patterns.

4.1 Assignment

```
[NAME] = [EXPRESSION];
```

The `[EXPRESSION]` expression will be evaluated and its result will be assigned to the name `[NAME]`.

4.2 Function Definition

```
[NAME] @ ([ARG1], [ARG2], ...) [BLOCK];
```

A function will be created and assigned to the name `[NAME]`. All arguments must be names, these names will be assigned values passed as arguments when called. `[BLOCK]` will be the body of this function.

4.3 Function Call

```
[NAME]([ARG1], [ARG2], ...);
```

Function with the name `[NAME]` will be called with passed arguments, the count of the arguments need not match the count in definition. Any unfilled name will be none.

4.4 Conditional branching

```
if [EXPRESSION] then [BLOCK1] else [BLOCK2];
```

The `[EXPRESSION]` expression will be evaluated and if the result converts to true, `[BLOCK1]` will be executed, otherwise `[BLOCK2]` will be executed.
The second clause may be omitted:

```
if [EXPRESSION] then [BLOCK1];
```

4.5 Loop

```
while [EXPRESSION] do [BLOCK];
```

The [BLOCK] will continually be executed while the [EXPRESSION] expression is evaluated to a value convertible true.

4.6 Break

```
break;
```

This statement will skip the execution of the remaining statements in the current block. If this statement is used in loop, this loop will not be repeated anymore.

4.7 Continue

```
continue;
```

This statement will skip the execution of the remaining statements in the current block. If this statement is used in loop, this loop will be repeated if the condition still evaluates to a truth equivalent.

4.8 Return

```
return [EXPRESSION];
```

This statement will skip the execution of the remaining statements in the current function. The result of this function will be set to the [EXPRESSION].
The following can also be used in that case the return value is set to none.

```
return;
```

4.9 Print

```
print [EXPRESSION];
```

The [EXPRESSION] expression will be evaluated and printed to the output designated by current environment.

5 Undefined behavior

Should the code not comply with this specification, its execution will either produce error upon parsing, execution, or the interpreter will behave in an unpredictable way. For majority of cases, there is an exception produced by the parser.

6 Examples

The following function returns the maximum of two integers:

```
max @ (first, second)
(
    if first > second then
    (
        return first;
    )
    else
    (
        return second;
    );
);
```

The following function provides the optimal move for the basic Nim game (1 to 3 matches are removed in one move).

```
nim @ (matches)
(
   whole = (matches / 4) * 4;
   remove = matches - whole;
   if remove ! 0 then (return remove;) else (return 1;);
);
```

It is assumed that the programmers could create games (exampli gratia *The Epic Robot Wrestling 2019*) and let others submit strategies in form of functions in this language as this language is easy to master and easy to control. The following function conducts a binary search in a sorted list of integers and returns the index of searched item or none if not present:

```
binfind @ (data, count, item)
(
   if count ? 0 then (return none;);
   left = 0;
   right = count - 1;
   while left < right do
   (
      middle = (left + right) / 2 + 1;
      if data(middle) ? item then
      (
         return middle;
      );
      if data(middle) > item then
      (
         right = middle - 1;
      )
      else
      (
         left = middle + 1;
      );
   );
   if data(left) ? item then (return left;) else (return none;);
);
```

The following function received a natural number and returns the Fibonacci number with that index. The suboptimal recursive algorithm is used to demonstrate recursion capability.

```
fib @ (amount)
(
   if amount ? 0 then (return 0;);
   if amount ? 1 then (return 1;);
   return fib(amount - 1) + fib(amount - 2);
);
```

Let us now assume that we want to supply the global scope with additional methods. We can extend `DefaultNamespace` class and provide C# method implementations for these methods. See below.

```
private class ValueIteratingEnvironment : DefaultEnvironment
{
    [BuiltinVariable("goal")]
    public IntegralValue goal = 2047;

    public int current = 1;
```

```
    [Interpreter.Environment.BuiltinFunction("move_next")]
    public void MoveNext(IList<IValue> args, out IValue result)
    {
        current *= 4; current += 3;
        result = (IntegralValue) current;
    }

    public int result = 0;

    [Interpreter.Environment.BuiltinFunction("set_result")]
    public void SetResult(IList<IValue> args, out IValue result)
    {
        if (args.Count != 1) throw new ArgumentException();
        if (args[0] is IntegralValue i) this.result = i;
        else throw new ArgumentException();
        result = new None();
    }
}
```

Reflection is used to extract builtin methods and variables from environment. Here we see how this could be used.

```
i = 0;
while move_next() ! goal do
(
    i = i + 1;
);
set_result(i);
```

This script calculates the number of values obtained by calling `move_next` before `goal`.
One important thing to note is, that variables are bound only once and their value will not be synchronized. Use getters or setters if synchronization is intended.