Hopeful Language Definition

1. Overview

The language is not case sensitive. Non-terminals symbols are represented by angle brackets, e.g. non-terminal *x* is represented as <*x*>. So follows that a terminal symbol is represented without angle brackets. A **bold typeface** is used to represent terminal symbols and reserved words, and so follows that a non-bold typeface is used to group terminal and non-terminal symbols together. Source code should be kept in files with the .hope extension, e.g. *hello_world.hope*

2. Syntax

The reserved words are **int**, **string**, **boolean**, **void**, **main**, **def**, **if**, **else**, **true**, **false**, **while**, **skip**.

The following are tokens in the language: **; , = ( ) “ + - * / % == != < <= > >= & | [ ] { }**

Integers are represented by a sequence of one or more digits, meaning *0* to *9*. Integers may begin with a minus sign, e.g. *-123*. Integers may not start with any leading *0*s, e.g. *01*. Strings are a sequence of letters, special characters, and digits. Strings are delimited by *“ ”*. Booleans are of the values *true* or *false*.

Arrays of all types are delimited by *{ }*, and each value is separated by a comma, e.g. *[1,2,3,4,5]*. Indexing of arrays begins at *0*. Individual values of arrays may be accessed using the index of that element, e.g. given the previous example, *array[2] = 3*.

Identifiers are a sequence of letters, digits, and underscores (“_”). Identifiers may only begin with a letter. They may also not be reserved words. Comments can appear between */** and **/*, they may be nested. They can also appear after *//*, and are delimited by a new line, thus it cannot be nested.

<program> |= <function_declarations> <main>

<main> |= **main ( ) {** <statement_block> **}**

<function_declarations>|= (<function> <function_declarations> | ε)

<function> |= **def** <return_type> <lhs_identifier> **(** <parameter_list> **) {** <statement_block> **return** (<expression> | ε) **; }**

<return_type> |= **int | boolean | string | void**

<parameter_list> |= <parameter> (**,** <parameter>)* | ε

<parameters> |= <type> <lhs_identifier>

```
<statement_block> |= (<statement> <statement_block> | ε)
<statement> |=  <function_call> ; |
                <array_declaration> |
                <print> |
                <assignment> |
                <declaration> |
                skip ;
<assignment> |= <lhs_identifier> = <expression> ;
<declaration> |= <type> <lhs_identifier> ;
<print> |= print ( <expression> ) ;
<if_statement> |= if ( <condition> ) { <statement_block> } else { <statement_block> }
<while_loop> |= while ( <condition> ) { <statement_block> }
<function_call> |= <rhs_identifier> ( <argument_list> )
<argument_list> |= <argument> ( , <argument> )* | ε
<argument> |= <fragment>
<expression> |= <function_call> |
                 <fragment> ( ( <arith_op> | <logic_op> ) <fragment> )*
<condition> |= <fragment> ( <comp_op> <fragment> )*
<fragment> |= <integer> | <string> | <bool> | <rhs_identifier>
<integer> |= number
<string> |= string
<bool> |= boolean
<lhs_identifier> |= identifier
<rhs_identifier> |= identifier
<type> |= int | boolean | string
<arith_op> |= + | - | * | / | %
<logic_op> |= | | &
<comp_op> |= == | != | < | <= | > | >=
```

3. Semantics

Declarations inside a function are local in scope to that function. Function arguments are *passed-by-value*. Variables are statically typed, and cannot be of the *void* type. The *skip* statement does nothing.

The operators in the language are:

Operator	Arity	Description
=	Binary	Assignment
+	Binary	Arithmetic addition
-	Binary	Arithmetic subtraction
*	Binary	Arithmetic multiplication
/	Binary	Arithmetic division

%	Binary	Arithmetic modulus
-	Unary	Arithmetic negation
&	Binary	Logical conjunction (and)
|	Binary	Logical disjunction (or)
==	Binary	Is equal to (arithmetic and logical)
!=	Binary	Is not equal to (arithmetic and logical)
<	Binary	Is less than (arithmetic)
<=	Binary	Is less than or equal to (arithmetic)
>	Binary	Is greater than (arithmetic)
>=	Binary	Is greater than or equal to (arithmetic)

The following table gives the precedence (from highest to lowest) and associativity of these operators:

Operator(s)	Associativity	Notes
-	Right to Left	Arithmetic negation
* / + - %	Left to Right	Arithmetic operators
< <= > >=	Left to Right	Arithmetic comparison operators
== !=	Left to Right	Equality and inequality operators
&	Left to Right	Logical conjunction
|	Left to Right	Logical disjunction
=	Right to Left	Assignment

4. Examples

The simplest non-empty file:
main() { }

A file demonstrating functions, arithmetics operators, and control flow:

```
function string fizzbuzz(int range) {
	string str = "";
	int i = 1;
	while (i < (range + 1)) {
		if (i % 3 == 0 & i % 5 == 0) {
			str = str + "fizzbuzz ";
		}
		elif (i % 3 == 0) {
			str = str + "fizz ";
		}
		elif (i % 5 == 0) {
			str = str + "buzz ";
		}
		else {
			str = str + string(i) + " ";
		}
		i = i + 1;
	}
	return(str);
}

main() {
	string fb = "";
	fb = fizzbuzz(20);
	print(fb);
}
```

A file demonstrating more arithmetic operators:

```
function void kg_to_stone(int kg) {
	float stone = kg * 2.2 / 14;
	print(stone);
	return();
}

main() {
	print(kg_to_stone(75));
}
```

A file demonstrating boolean and logical operators:

```
function boolean test_eq() {
	boolean all_correct = false;

	all_correct = 5 == 5; // true
	all_correct = 6 != 10; // true
```

```
    all_correct = 6 > 5; // true
    all_correct 6 >= 6; // true
    all_correct = 7 < 10; // true
    all_correct = 7 <= 7; // true

    return(all_correct);
}

function boolean test_logic() {
    boolean all_correct = false;

    all_correct = 1 & 1; // true
    all_correct = 0 | 1; // true
    all_correct = ~0; // true

    return(all_correct);
}

main() {
    print(test_eq());
    print(test_logic());
}
```

A file demonstrating arrays:

```
function int sum_list(int[] a) {
    int sum = 0;
    int i = 0;
    while (i < a.length()) {
        sum = sum + a[i];
        i = i + 1;
    }
    return(sum);
}

function int[] mult_list(int[] a, int factor) {
    int i = 0;
    while (i < a.length()) {
        a[i] = a[i] * factor;
        i = i + 1;
    }
    return(a);
}

main() {
    int[] test = [1, 2, -3, 4];
```

```
        print(sum_list);

        test = mult_list(test, 2);
        int i = 0;
        while (i < a.length()) {
                print(a[i]);
                i = i + 1;
        }
}
```