Algonquin College Logo

# SCHOOL OF ADVANCED TECHNOLOGY

### ICT - Applications & Programming

### Computer Engineering Technology – Computing Science



A11

Language Specification

Team:

Isaac Proulx - Id: 041007853

Language Name: Arrow

|  |  |
| --- | --- |
| **Part**  **1** | **Language User Reference** |

**EXPLANATION**

This language is inspired by JavaScript and will have a somewhat similar syntax, for example curly braces “{“ and “}” to denote blocks, and I also want to include the arrow function syntax (probably implemented as a compiler macro). I also took the concatenation operator “..” from LUA.

* 1. **User Manual**

**Element 1: Name / Extension**

Language Name: Arrow

File Extension: .aro

The syntax of this language is mostly inspired by JavaScript.

**Element 2 – Comments**

Single line comments will treat anything that comes after them on the same line as a comment.  
Single line comments are created by typing “**//”** anywhere in the program (except inside of strings).

|  |  |  |
| --- | --- | --- |
|  | //This is a single line comment. |  |

Multiline comments are opened by typing “***/\*”*** and are closed by typing “***\*/”.*** Both the opening and closing tags can be typed anywhere in the program (except inside of strings), however, closing tags must come after an opening tag; A closing tag with no respective opening tag is a syntax error (Unexpected Token), An unclosed comment (i.e. an opening tag with no respective closing tag) is a syntax error (EOF in Comment). Multiline comments are also non-greedy meaning *they will close at the first closing tag encountered* (i.e. /\*This is a comment\*/ is valid but /\*This is a comment\*/\*/ is not).

|  |  |  |
| --- | --- | --- |
|  | /\* this is  a multiline comment \*/ |  |

**Element 3 – Keywords**

|  |  |
| --- | --- |
| let | Declares a variable that can be reassigned |
| const | Declares a variable that CANNOT be reassigned |
| int | Integer data type |
| float | Floating point data type |
| string | String data type |
| bool | Boolean data type |
| list | Array List data type |
| if | Used to declare an if statement |
| else | Used to declare an else statement |
| for | Used to declare a for loop |
| while | Used to declare a while loop |
| break | Used to exit a loop |
| continue | Used to skip to the next iteration of a loop |
| function | Used to declare a function |
| return | Used to return a value from a function |
| try | Used to declare a try block |
| catch | Used to declare a catch block |
| finally | Used to declare a finally block |

**Element 4 – Datatypes**

|  |  |
| --- | --- |
| Type | Implementation |
| int | C long (signed 64 bit -263 to 263-1) |
| float | C double (64 bit IEEE 754 standard) |
| string | Most likely implemented using a C union containing a C char pointer and a C unsigned int, meaning the string can hold up to 232 characters (since the length is stored, there is no need for null byte). |
| bool | Boolean (true or false) implemented using C char (1 byte). |
| list | (Array List) Most likely implemented in C using an array of structs that would be some kind of wrapper for all data types in this language. The array would also need to be resized as necessary. |

**Element 5 – Variables**

Variables are declared with the **let** or **const** keywords followed by the data type and then the name.

|  |  |  |
| --- | --- | --- |
|  | let list myList;  let int myInt = 1;  let float myFloat = 1.23;  let string myString = “Hello World”;  const int myConstantInt = 1;  myList = [myInt, myFloat, myString, myConstantInt]; |  |

**Element 6 - Commands**

Values can be assigned to variables by using an equals sign “=” followed by the value to assign.

If the programmer tries to assign an expression, the expression will be evaluated and the variable will be assigned the result of the expression.

Parenthesis “(“ and “)” can be used to contain an expression, they aren’t required to make an expression, but they can allow you to include a subexpression inside an expression.

|  |  |  |
| --- | --- | --- |
|  | let int myVar = (4 + 2); // becomes: let int myVar = 6;  4 + 2 \* 5; // evaluates to 14  (4 + 2) \* 5; // evaluates to 30 |  |

String concatenation uses concatenation operator “..” :

|  |  |  |
| --- | --- | --- |
|  | let string myString = “First part ” .. “Second part”;  print(myString); // expected output: “First part Second part” |  |

Math Operators (order of precedence: BEDMAS):

|  |  |  |
| --- | --- | --- |
| Operator | Description | Precedence  (lower number = higher precedence) |
| \*\* | Exponentiation | 1 |
| / | Division | 2 |
| \* | Multiplication | 2 |
| % | Modulo | 2 |
| + | Addition | 3 |
| - | Subtraction | 3 |
| ++ | Increment | 1 |
| -- | Decrement | 1 |

Conditions are expressions that evaluate to true or false.

Conditional operators always have a left hand side (LHS) and a right hand side (RHS), a conditional operator will evaluate the LHS and the RHS and then compare the results, it will then return the result of the comparison which is either true or false.

Conditional Operators:

|  |  |
| --- | --- |
| Operator | Description |
| == | Equal to |
| != | Not equal to |
| < | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |
| ? | Start ternary statement |
| : | Defines second case for ternary statement |

Boolean Operators:

|  |  |
| --- | --- |
| Operator | Description |
| ! | Not  true->false  false->true |
| && | And  false && false -> false  true && false -> false  false && true -> false  false && false -> false |
| || | Or |

Bit Wise Operators:

|  |  |  |
| --- | --- | --- |
| Operator | Description | Precedence  (lower number = higher precedence) |
| ~ | Bit Wise NOT | 1 |
| << | Bit Shift Left | 2 |
| >> | Bit Shift Right | 2 |
| & | Bit Wise AND | 3 |
| ^ | Bit Wise XOR | 3 |
| | | Bit Wise OR | 3 |

Assignment Operators:

|  |  |
| --- | --- |
| Operator | Description |
| = | Assignment |
| \*\*= | Exponentiation Assignment |
| /= | Division Assignment |
| \*= | Multiplication Assignment |
| %= | Modulo Assignment |
| += | Addition Assignment |
| -= | Subtraction Assignment |
| <<= | Bit Shift Left Assignment |
| >>= | Bit Shift Right Assignment |
| &= | Bit Wise AND Assignment |
| ^= | Bit Wise XOR Assignment |
| |= | Bit Wise OR Assignment |
| => | Used to define an arrow function |

Try, Catch, Finally:

|  |  |  |
| --- | --- | --- |
|  | try{  print(“This wont work”/2);  }catch(list error){  //error[0] contains the error message (string)  //error[1] contains the stack trace (string)  print(“Error: “..error[0]);  }finally{  print(“This will always run”);  } |  |

For loop:

|  |  |  |
| --- | --- | --- |
|  | for(let int loopVar = 0; loopVar<10; loopVar++){  print(loopVar);  } |  |

While loop:

|  |  |  |
| --- | --- | --- |
|  | let string s = “”;  while(s != “a”){  s = input(“enter char: “);  }  print(s); |  |

Input:

Subject to change, may add something like scanf in which you specify what type you’re reading

|  |  |  |
| --- | --- | --- |
|  | //let string userInput = input(PROMPT);  let string firstName = input(“Enter first name: “);  print(“Hello ” .. firstName); |  |

Output:

Subject to change, may add something like printf in which you specify what type you’re outputting

|  |  |  |
| --- | --- | --- |
|  | //print(STRING);  print(“Hello World”); |  |

Functions:

|  |  |  |
| --- | --- | --- |
|  | /\*  function TYPE NAME (PARAMETERS) {  CODE  }  \*/  function int addOne(int a){  return a+1;  } |  |

**Element 7 – Proper elements**

* Passing functions as parameters:

I’d like to allow functions to be passed to other functions.

|  |  |  |
| --- | --- | --- |
|  | function int callFunc(int callback()){  callback();  return 0;  }  function int doThing(){  print(“Hello World”);  return 0;  }  callFunc(doThing); // expected output: “Hello World” |  |

* Arrow Functions:

|  |  |  |
| --- | --- | --- |
|  | //let TYPE NAME = (PARAMETERS) => BODY  //const TYPE NAME = (PARAMETERS) => BODY  //const and let can both be used for arrow functions  let int arrowFunction = (int a) => {  return a+1;  }  arrowFunction(5); // expected return value: 6  //implicit return (only valid with arrow functions)  let int arrowFunctionWithoutBrackets = (int a) => a+1;  arrowFunctionWithoutBrackets(7); // expected return value: 8  //implicitly evaluates and returns the result of the expression  //(only valid with arrow functions)  const int arrowFunctionWithExpression = (int a) => (a+1)/2;  arrowFunctionWithExpression(4); // expected return value: 2 |  |

* Closures/Anonymous Functions:

|  |  |  |
| --- | --- | --- |
|  | function int callFunc(int callback()){  callback();  return 0;  }  callFunc(int () => {  print(“Hello World”);  return 0;  }); // expected output: “Hello World” |  |

|  |  |
| --- | --- |
| **Part**  **2** | **Examples** |

**Hello World**

|  |  |  |
| --- | --- | --- |
|  | print(“Hello World”); |  |

**Sphere Volume Expression (or any other example)**

|  |  |  |
| --- | --- | --- |
|  | const int volumeOfSphere = (int radius) => {  return 4.0/3.0 \* 3.14 \* radius\*\*3;  }  function int volumeOfSphere(int radius){  return 4.0/3.0 \* 3.14 \* radius\*\*3;  } |  |

**Miscellaneous examples**

|  |  |  |
| --- | --- | --- |
|  | //This is a single line comment  /\*This is  a multiline comment  \*/  //string concatenation  let string myString = “First part” .. “Second part”;  print(myString);  //if statement  let int x = 1;  let int y = 2;  if(x==y){  print(“equal);  }else{  print(“not equal”);  }  //functions  function add(int a, int b){  return a + b;  }  add(4,5);  //arrow functions  let sub(int a, int b) => {  return a – b;  }  sub(9,4); |  |

|  |  |
| --- | --- |
| **Part**  **3** | **Architectural Aspects** |

**Advantages**

My goal is to create a language that I would personally like to use. I want it to be simple but still complex enough to not be too restrictive.

**Strategy: C Implementation**

* Data Types
  + For data types, refer to element 4.
* Parsing:

For parsing the language, first the input (the source code) should be streamed to the lexer as characters. The lexer will most likely act as a state machine which will perform actions like creating, modifying, and emitting tokens based on each character encountered in the current state. The tokens will then be streamed to the parser which will construct an abstract syntax tree based on them. The parser would probably also be some kind of state machine which would determine meaning based on each token encountered in the current state, for example if a variable declaration token were encountered, it should switch to a variable declaration state, then it should expect a data type token, and then a variable name token (or something like that), and then if it encounters something like an end of statement token it would know that the declaration is complete, but it could also encounter an assignment token, in which case it would enter the assignment state.

* Blocks
  + Curly braces “{“ and “}” define a block

|  |  |  |
| --- | --- | --- |
|  | for(let int i=0; i<10; i++){ // start of block to be looped  print(i); // 0, 1, 2, 3, 4, 5, 6, 7, 8, 9  } // end of block to be looped  print(i); // 10 |  |

* Scoping
  + I’ll probably use block scoping

|  |  |  |
| --- | --- | --- |
|  | const float PI = 3.14; // globally accessible constant  let string writableValue = “”; // globally accessible variable  function float myFunc(int radius) { // start of block 1  let string parentValue = “in parent scope”;  function int internalFunction(){ // start of block 2  let string childValue = “in child scope”;  // has access to parent scope  print(parentValue); // expected output: “in parent scope”  return 0;  } // end of block 2  internalFunction();  // does not have access to child scope  print(childValue); // Reference Error: childValue is not defined  // has access to parent scope  writableValue = “New Value”;  return 3.0 \* PI \* radius \*\* 2;  } // end of block 1 |  |

**Basic ideas about C implementation**

Structures and datatypes I’ll probably use in my implementation are: long, double, char, unsigned int, structs, unions, as well as pointers and arrays of most (or all) of the previously mentioned types.

I think the most difficult part of parsing this language will be the planning involved in setting up all the necessary data structures, for example:

* creating the lexer
  + how tokens are generated and returned
  + keeping track of state
  + how to represent all different kinds of tokens
* how to stream tokens to the parser
* creating the parser
  + handling the logic of what each token means in the given context
  + representing the abstract syntax tree

**Problems when using C implementation**

Some of the main problems I will probably face:

* Memory Management
  + Dynamic sizing for the list data type
  + Making sure to free memory when it’s no longer in use (garbage collector)
* Representing and managing things like functions and variables (not the data types but the containers themselves)

**References**

* [https://en.wikipedia.org/wiki/Double-precision\_floating-point\_format#IEEE\_754\_double-precision\_binary\_floating-point\_format:\_binary64](https://en.wikipedia.org/wiki/Double-precision_floating-point_format" \l "IEEE_754_double-precision_binary_floating-point_format:_binary64)
* <https://docs.julialang.org/en/v1/>

Algonquin College

Winter, 2023