# The One-file Programming Language

Dr Jonas Lundberg

Software Technology Group
Department of Computer Science, Linnaeus University
Veides Plats 7, SE 351 95 Växjö, Sweden
Jonas.Lundberg@lnu.se

*Abstract*— **The One-file programming language is a very simple programming language intended as an example language for the course 4DV507/1DT902 – Code transformation and interpretation. This text serves as an introduction and definition of the language.**

```
# iteration.ofp

void main() {
   int res = sumUpTo( 10);
   println(res);  # res = 55

   string s = "All students got grade A!";
   res = countA(s);
   println(res);   # res = 2
}

# Compute 1+2+3+...+n
int sumUpTo(int n) {
   int i = 1;
   int sum = 0;
   while (i < n+1) {
      sum = sum + i;
      i = i + 1;
   }
   return sum;
}

# Count number of A in string str
int countA(string str) {
   int n = 0;
   int i = 0;
   while (i < str.length) {
      char c = str[i];
      if (c == 'A')
         n = n+1;
      i = i+1;
   }
   return n;
}
```

## I. Introduction

The One-file programming language (OFP) is a simple programming language with a limited set of constructs, statements, types, and values. See above for an example. An OFP program always consists of a single file (with postfix `.ofp`, as in One-File Program). An OFP program consists of one or more functions. The execution starts in a function with signature `void main()`. No fields or global variables are allowed.

### A. Functions

An OFP program consists of one or more function declarations. Each function declaration contains a return type (or `void`), a name (identifier), zero or more parameters, and a function body (statements enclosed in curly brackets `{ ... }`). For example:

```
float max(float x, float y) {
   if (x > y)
      return x;
   else
      return y;
}
```

Functions returning `void` do not have any return statements. Non-void functions must have at least one return statement. Apart from the program entry point `main`, return interrupts the execution and return control to the calling function. The value returned by `return` must have a type in agreement with the declared return type.

## II. Statements, Types, and Variables

### A. Control Statements

The OFP language has only two control statements: `while` and `if`. They work exactly as in Java. For example:

```
while (i < 100) {
   n = n + i;
   i++;
}
```

```
if (x > y)
   max = x;
else if (y > x)
   max = y;
else   # x equals y
   max = x;
```

While bodies and if branches can be either a statement block (`{ ... }`) or s single statement. An if statement starts with an `if (...)`, followed by zero or more `else if (...)`, and (optionally) ends with an `else`.

### B. Available Types

OFP only supports the following types: `int, float, bool, char, string, int[], float[], char[]`. They represent integers, decimals, booleans (`true` or `false`), character, strings, and integer-, decimal-, and character-arrays.

### C. Variables

All variables (and function parameters) must be declared (e.g. `int max;`) before they can be used. A variable declaration can also initialize a variable with a value (e.g. `int max = 7;`). Variables can be declared everywhere inside a function. The enclosing statement block defines its scope (visibility). A variable declaration in an inner block hides a variable declaration with the same name in an outer block.

### D. Literals

```
int n = 14;
float pi = 3.14;
bool b = true;
char c = 'p';
string s = "Hello World!";
int[] iArr = {1,2,3,4,5,6,7};
float[] fArr = {1.2, 2.4, 3.6, 4.8, 6.0};
```

Literals (values) are defined in the same way as in Java. See example above.

## III. Strings, Arrays, and Print

### A. String

Strings in OFP behave very much like arrays except that they are immutable. For example:

```
string s = "Hello World!";
int i = 0;
while ( i<s.length) {
   char c = s[i];
   println(c);
}
```

We can always get the number of characters in a string using `s.length` and we can access a singel character using an array access like `char c = s[i]`. However, since strings are immutable, we can **not** update the content in a string (`s[3] = 'j'`). Indices start at 0 and end with `s.length-1` Finally, string concatenation (e.g. `string+string`, `string+int`, etc) is **not** allowed.

### B. Arrays

Arrays in OFP work like arrays in Java. For example:

```
float arr = new float[3];
arr[0] = 2.0;
arr[1] = 3.0;
arr[2] = 3.14;
float sum = 0;
int i = 0;
while ( i<arr.length) {
   float f = arr[i];
   sum = sum + f;
}
```

Hence, we can create an empty array of size 3 using `new float[3]` or create and initialize an array in one statement like

```
int[] iArr = {1,2,3,4,5,6,7};
```

Arrays can only be created for types `int`, `float`, and `char`.

### C. Print

OFP has two built-in print statements: `print(...)` (no line break) and `println(...)` (including a line break). They take a single argument and can both handle the following types: `int, float, bool, char, string`. They can **not** handle arrays.

## IV. Type Mixing and Other Details

### A. Type Mixing

OFP does not allow mixed types in an expression. Not even integer and decimals. Hence, the following expressions should generate a type error in the semantical analysis:

### B. Literals

```
int n = 14.17;  # assign float to an int
float pi = 314; # assign int to a float
float x = 1.22 + 7; # float + int
int[] iArr = {1,2,3,4,5,6,7.5};  # Final value float
float[] fArr = {1.2,2.4,3.6,4.8,6}; # Final value int
```

### C. Other Details

- `#` starts an end-of-line comment
- Identifiers can only consist of the letters a-z and A-Z.
- Strings can only contain the letters a-z and A-Z and the characters !, ., ,, ?, =, :, (, ). Notice that both "" (empty string) and " " (whitespace) are allowed.
- Float literals must contain a decimal dot.
- OFP has the following binary operators: `+,-,*,/,<,>,==` and they follow standard semantics for integers and decimals.
- The compare operator `==` can also be applied on characters (but not strings)
- OFP has a unary operator `-` that only can be applied on numerical expressions.