

# Output C Code

## Computer Language Processing '16 Final Report

Cédric Viaccoz    Bastian Nanchen

EPFL

{cedric.viaccoz}@epfl.ch

### 1. Introduction

The first part of the lab was to implement a compiler for Toy Object-Oriented Language (Tool), which is a small object-oriented programming language.

This has led us to conceive a Lexer, a  $LL(1)$  grammar, a Parser, a Type Checking system and to generate binary code for the Java Virtual Machine.

The aim of this project is to replace code generation for the Java Virtual Machine with a pretty printer, which outputs C code.

The difficulty is to make object-oriented code into C code which does not have built in data types to support class, as it is an imperative programming language .

### 2. Examples

The example below presents how a class and the inheritance could be depicted in C programming language.

```
1 class Animal{
2     var isAPet: Bool;
3     var speed: Int;
4 }
5
6 class Dog extends Animal{
7     def bark(): String = {
8         return "Woof!";
9     }
10 }
```

This Tool code will be pretty-printed in C like this:

```
1 struct Animal{
2     int isAPet;
3     int speed;
4 };
5
6 struct Dog{
7     int isAPet;
8     int speed;
9     char * (*bark)(void *);
10 };
```

```
11
12 char * Dog_bark (void * this) {
13     // some code that will be depicted in the
14     implementation part
15 }
```

The class are represented in C by a structure. The variables of the class are members of the structure. They are declared in the same fashion.

The variables of the class “Animal”, they also find themselves in the “Dog” structure because the class “Dog” inherits from the class “Animal”.

Contrariwise the method of the class is declared outside of the structure. In C, a structure can’t have a function declaration as a member. The function shall be declared outside. But the structure has a pointer to the function. The function is declared with the name of the class as prefix followed by an underscore. Thus the belonging to the “Dog” class is suggested.

Therefore this is how the C code mimics the behavior of an object-oriented class.

There exists other dilemma like method chaining, dynamic dispatch,..., which be discussed in the implementation section.

### 3. Implementation

This is a very important section, you explain to us how you made it work.

#### 3.1 Theoretical Background

The only background needed for this extension is knowing the tool’s AST and the C language. However since we are trying to emulate an object-oriented language into one that is not, the book “Object-oriented programming in ANSCI-C”<sup>1</sup> by Axel Schreiner has

<sup>1</sup><https://www.cs.rit.edu/~ats/books/ooc.pdf>

been useful for some concepts and it will be referenced throughout this report.

## 3.2 Implementation Details

Two files were created to implement this project.

### 3.2.1 Data Type

The first one is “CDataType.scala”. It contains a class that depicts the C’s structure.

---

```
1 class StructDef(val name: String,
2   val membersList: ListBuffer[StructMember])
```

---

A trait “StructMember” is extended by two classes to implement the two kinds of structure’s members, which are a variable or a pointer to a function (as explained in section 2. Examples). The pointer to a function holds two fields, one for an internal representation of a C function pointer (the class “FunctionPtr”), and one for the AST version of the method declaration that should be linked to the pointer.

---

```
1 class StructVar(val name: String,
2   val tpe: CType) extends StructMember {
3   // some code
4 }
5
6 class StructFunctionPtr(val ptr: FunctionPtr,
7   var mtDcl: MethodDecl) extends StructMember {
8   // some code
9 }
10
11 class FunctionPtr(val name: String,
12   val retType: CType, val args: List[CType]) {
13   // some code
14 }
```

---

### 3.2.2 Code Generation

The second file is “COutputGeneration.scala”. Its skeleton and behavior is based on the “CodeGeneration.scala” completed to output binary code for the Java Virtual Machine. The program uses the same architecture of methods like: “cGenMethod”, “cGenStats”, “cGenExpr”. Each of this method returns a `StringBuilder`. The final is a .h and .c, which encloses all the `StringBuilders` concatenated.

**Overview** The main method of the object `COutputGeneration` is `def run(ctx: Context)(prog: Program)`, which takes in parameter the AST of the code.

The method starts by creating a `StringBuilders` for the preprocessor directives. It includes the headers `stdio.h`, `string.h` and `stdlib.h`. It contains `#define INT_MAX_LENGTH 12`, which is useful in order to have correct `Int` to string of characters concatenation. It also encloses the definition of default constructor for every classes of the program.

---

```
1 #define nAnimal 0
2 #define nDog 1
```

---

**Listing 1.** In the case of the example used in section 2. Examples

Finally, a `StringBuilder` contains the inclusion of the file’s header, which is directly created.

Afterward a method takes care of generating the structures corresponding to the Tool’s classes:

`def genStructDef(ct: ClassDecl): StructDef.` The method takes as parameters a `ClassDecl` and returns a corresponding `StructDef`, which was presented in section 3.2.1 Data Type. Then the `StructDef`’s method `toStringRepr` to return the structures as `String`. Thereafter the method manages the program’s methods.

---

```
1 def genMethods(ct: ClassDecl): StringBuilder =
2   (for(mt <- ct.methods)yield(cGenMethod(ct,
3     mt))).foldLeft(new
4     StringBuilder())((a,b) => a append b)
```

---

The method `cGenMethod` traverses the AST node `MethodDecl` to translate the statements and the expressions in C programming language, using the methods `cGenStats` and `cGenExpr` like in the Code Generation lab. The method returns a `StringBuilder` representing a function, that corresponds to a Tool’s class method.

Then the Tool’s main method is translated into C programming language.

---

```
1 def genMainMethod(main: MainObject):
2   StringBuilder = {
3   val mainMethod = new StringBuilder("int
4     main(void){\n")
5   main.stats.foldLeft(mainMethod)((sB, stmt) =>
6     sB append(cGenStat(stmt)(1, None)))
7   return mainMethod.append("\treturn 0;\n")
8 }
```

---

The `genMainMethod` method receives as parameter the AST node `MainObject`. It is converted as the main function in C programming language.

```

1  int main(void){
2      return 0;
3  }

```

MainObject contains a list of StatTree. Each StatTree is evaluated by the method cGenStat and the returned StringBuilder(s) are the body of the C main function.

In the run method, there exists an object defaultConstructor. It represents the construction of the default constructor. The function is called new. The function takes as parameter an integer. It is one of the defined integer for each Tool's class, that we have seen before. Subsequently a method addStructConstructor takes care of adding a `case` element to the new function for a correct initialization of a Tool's object in C programming language.

```

1  void * new(int type){
2      void * object;
3      switch(type){
4          case nAnimal:
5              object = malloc(sizeof(struct Animal));
6              break;
7
8          case nDog:
9              object = malloc(sizeof(struct Dog));
10             ((struct Dog*) object)->bark = Dog_bark;
11             break;
12         default:
13             return NULL;
14     }
15     return object;
16 }

```

**Listing 2.** In the case of the example used in the section 2. Examples

b

At the end, the functions

`void helper_reverse_plus(char str[], int len)`, `char* itoa(int num)` and `int * arrayAlloc(int size)` are constructed. Their usefulness will be explained in the subsections to come.

Finally all these snippet of C code are concatenated and written in a .c file.

## Inheritance

**Dynamic dispatch** Dynamic dispatch is accomplished during the default construction of a new Instance of a class. In Tool, there is only on type of constructor, the default one. One idea we borrowed from our reference

book<sup>2</sup> was to have a default method be outputted called “new” which would take care of creating a new Instance of a struct and returning it as a generic pointer “void \*”. *insert example code of new() here*

**Concatenation** The AST node “Plus” asked for some attention. The Tool language allows to apply the “+” operator on Int and String operands.

The situation, where the two operands are Int, is straightforward to emulate in C programming language. It's also an addition.

The event where the two operands are String is a little bit more tricky. It is necessary to allocate memory before concatenate the two string of characters.

```

1  strcpy(malloc(strlen("+ lhsString +") + strlen("
    + rhsString +") + 1), " +
2      lhsString +"), " + rhsString +")"

```

Finally when the two operands are different, we were obliged to create two C functions:

`void helper_reverse_plus(char str[], int len)` and `char* itoa(int num)`. These functions transform an integer into a string of characters and return it.

These two functions are written in every .c file generated by the compiler.

<sup>2</sup>[www.cs.rit.edu/~ats/books/ooc.pdf](http://www.cs.rit.edu/~ats/books/ooc.pdf) at page 11, si jamais faudra mettre ca dans la bibliography la fin scuse

**Method chaining** This problem was one, that we had not seen coming.

In Tool, there is the possibility to write multiple call to a method on one line. It works because each method returns an object. We can't directly translate it in C programming language with the model we choosed to make object oriented possible. Our "methods" need to have as a first argument a pointer to the struct calling, so to make a method call on a struct we would need to write : `void * a = new(nA); ((struct A *)a)->foo(a)`. Our variable a needs to be referenced two times, first to get the good function pointer and then as the first argument if "foo" need to access fields defined in A. Therefore chained method calls need intermediate variables to work correctly. To tackle this problem, we created a simple object: "tmpVarGen".

---

```
1 object tmpVarGen{
2   private var counter = 0
3   def getFreshVar: String = {
4     counter += 1
5     return "tmp"+counter
6   }
7   def getLastVar: String = "tmp"+counter
8 }
```

---

It is used to have a unique variable name. It has two methods. "getLastVar" returns the last created variable name. "getFreshVar" return a new variable name.

Now at each evaluation of an AST node, which is an expression, it is important first to evaluate the expressions and assign to intermediate variables using the object "tmpVarGen". Next the expression is written using the variables.

---

```
1 case Equals(lhs: ExprTree, rhs: ExprTree) =>
2   val lhsString = cGenExpr(lhs)
3   val lhsLastVar = tmpVarGen.getLastVar
4   val rhsString = cGenExpr(rhs)
5   val rhsLastVar = tmpVarGen.getLastVar
6   val andExprResultVar = genTabulation(indentLvl)+
7     CInt.toString()+" "+tmpVarGen.getFreshVar+" =
8     lhsLastVar+" == "+rhsLastVar+";\n"
9   return
    lhsString.append(rhsString).append(andExprResultVar)
```

---

Here it is an example with the evaluation of an expression AST node: Equals. The left-hand side and right-hand side expressions are first evaluated using the "cGenExpr()" method and are stored in intermediate variable using the "tmpVarGen" object. Afterward the

equal expression as known in C: "lhs == rhs" is written using the intermediate variable. At the end we append all element together. Thus one Tool line to write an equality is three lines in C programming language.

### Tabulation

## 4. Possible Extensions

### References