# Contents

# DPCC: DParo's Own C-Alike Compiler

Davide Paro

December 2020

## 1 Assignment Description

This project is the implementation of an assignment for a course on **Compilers** at the department of Computer Engineering Master Degree Padova (ITA).

The assignment consists in implementing a toy compiler for a toy language. Mostly we are required to design the frontend side (input, lexing, parsing, type checking) and simply emitting a very simple theoretical Intermediate Code (3AC) We are free to design the syntax of this toy language however we like.

The assignment specs out the how the compiler should be composed. We can in fact distinguish these macro components:

- **Input Stage** deals with the input byte stream that composes the source of the program.
- **Lexer/Scanner** has the purpose of grouping characters (lexical analysis) together to compose compunded structures (called tokens). For the project assignment we can use **Flex** to aid in the implementation of the scanner.
- **Parser** for performing the syntax analysis. It is what defines the look & fell (grammar) of the language. For the project assignment we can use **Bison** to aid in the implementation of an LR parser.
- **Intermediate Code Generator**. The ultimate purpose of a compiler is to produce something useful. In this project assignment we are not asked to implement a proper backend. Instead, we need to emit a 3AC representation of our input program. More in this later.

In particular the final Intermediate representation that we need to emit is based on Three Address Code (3AC), that is, each statement can only have 1 operand at the left hand side of the assignment, and 2 operands at the right hand side of the assignment, and an operator driving the operation that should be performed.

You can view more about 3AC at the following wikipedia link.

In practice the emitted 3AC code is on itself a partially valid C program, it's only missing variable declarations at the top for the temporary variables.

For the specification of the Intermdiate Code that is generated please refer to appendix A

So the project requires to produce this kind of 3AC / C hybrid. Control flow is allowed to be implemented trough the usage of C labels and simple if conditional followed by a goto statement. Inside the if conditional there can only be a single element composing the expression.

The assignment requires the following features from the programming langugage that we should develop:

- Variables declaration, initialization and assignment
- Scoping. Variable names are reusable in different scopes. Variable shadowing may or may not warn/fail/pass depending on the design choices.
- Only 2 types of variables: integers, booleans
- Assignment statements, print statements, if statements, and at least 1 loop statement at our liking
- Handling of simple mathematical expressions that we can encounter in common programming languages: addition, subtraction, multiplication, division, modulo, etc ...
- **Function definition, function calls, and custom user definable types are not required**

## 2 DPL and dpcc: A quick peak at the language and at the compiler

After describing the project assignment, from now on the following sections will describe the proposed language and the proposed compiler.

**DPL** and **DPCC** are respectively the **name of the language** and the **name of the implemented compiler**. They are named after their author.

From now on **DPL** and **DPCC** will be used for brevity for refering to the language and to the compiler. We will use **dpcc** (all lowercase) instead, to refer to the actual executable where the compiler lives.

That being said **DPCC** (all UPPERCASE) & **dpcc** (all lowercase) are mostly used interchangeably to refer to the same thing.

### 2.1 DPL: Structure of the language

**DPL** is mostly a C-alike compatible language. It borrows some syntax also from **Rust** & **JS** especially in the variable declaration syntax (usage of the keyword **let**).

**Rust** is a modern system programming language with strong typing guarantees. Among all the interesting features that Rust provides one of them is type deduction. Thanks to Rust's strong typing guarantees and thanks to strong type deduction rules implemented inside the **rustc** compiler, Rust allows one to declare variables with a very low-weight syntax, similar to a syntax provided from a typical dynamic language (for example JS).

**DPL**, like Rust, also have a very simple form of a type deduction system. It is not even remotely close to the Rust type deduction system, but still it allows the user of the language to not always need to specify the type of each variable in a declaration.

Here follows some demonstration **DPL** to show of the concepts and the syntax of the language:

We will see briefly in later sections how this is implemented in the **dpcc** compiler.

In **DPL**:

- Spaces and newlines mostly do not matter, they simply introduce token boundaries.
- Comments start with the double forward slash '//', and C-style multiline comments are instead not supported (at the time of writing).

Here follows some chunks of **DPL** code to show off the language concepts and syntax:

```
// Print statement with immediate C-style strings. C-style strings can only be used inside print statement
print("Hello world\n");

// Variable declaration and initialization
let a = 10;        // Integer Type deduced
let f = 10.0;      // Float type deduced
let b = false;     // Boolean type deduced

// Explicit types
let i: int = 0xffff & ~0xb00111;
let f: float = 10.0 + 20.0 ** 2;

// Immediate values can be printed
print(10);
print(30 + 4);

// Variables can be printed
print(i);      // Print integer
print(f);      // Print float

// Casting can be used to enforce type conversion
let myInt: int = int(10.00f);
let myFloat = float(0xFF);

// Type deduction
let b = (10 < 20); // Boolean type is deduced
let f = 1 + 2.0f;  // Float type is deduced (the 1 is upcasted to a float)

// Scoping and restricting variable declarations to the current scope
{
    // Simple single dimension arrays declaration
    let buf_i: int[100];         // Known size integer array
    let buf_f: float[100];       // Known size float array
```

```
    // Integer array with deduced size from the RHS initializer list
    let buf = [ 10, 20, 30, 40, 50 ];

    // Arrays can be printed
    print(buf);
}

let buf: int[100];

// Control flow
for (let i = 0; i < 100; i++) {
    buf[i] = i ** 2;

    if (buf[i] == 10) {
        print("buf[i] is 10!!!\n")
    }
    else if (buf[i] == 20) {
        print("buf[i] is 20!!!\n");
    }
    else {
        print("None of above\n");
    }
}
```

The cool thing about **DPL**, is that it is **almost a Javascript subset**. That is one can simply copy the **DPL** code, strip the type information (if they're used anywhere) by manual editing or automatically, and paste the same code into a browser console to evaluate as JS code. With a couple modifications here and there (arrays with no RHS initalizer list must be converted into valid JS array), and the definition of the function 'print', one can test if the compiler **dpcc** is producing the correct output by simply evaluating the same code in a browser console.

This example shows how to convert **DPL** into **JS** by manual editing:

```
let a = 10;                 // This is also valid JS code
let b = [ 10, 20, 30, 40 ]; // This is also valid JS code
let c: int = 10;            // Mostly valid JS code, remove the type info and the colon
let d: int[1024];           // Js arrays grow on demand automatically when touching elements.
                            // No need to specify neither type nor number of elements

print(d);                   // Valid JS code if function print were to be defined
```

Here's the equivalent JS code:

```
const print = console.log;  // Define it once at the top of the script

let a = 10;                 // Same as before
let b = [ 10, 20, 30, 40 ]; // Same as before
let c = 10;                 // Just strip the int type
let d = [];                 // Just initalize with empty array is enough

print(d);                   // Works because print is defined at the top
```

Most other **DPL** syntax and features, like code blocks, conditionals, loops . . . etc are valid JS code thanks on how the grammar for **DPL** was defined.

For now only 5 types are implemented: **bool, int, float, string, bool**[]**, int**[]**, float**[]. Only single dimensions array are for now supported. So arrays do not generalize to any number of dimensions.

Most of these types have full on semantics, meaning that the compiler can deduce a type of an expression given the types of its operand. In some cases it can reject the code if the operands of an expression have invalid types. At the current time of writing this report, **string** types are quirky, meaning that they don't have a full type tracking inside the compiler like other types do. The compiler still knows what a string is, and in fact a string literal is marked with the **string** type, but strings undergo different semantics. They cannot be assigned or operated on like a variable, but instead, they can only be used as a parameter to the print statement.

## 2.2 DPCC: Using the compiler

The **dpcc** is written in the **C** language. Unfortunately at the current time of writing **dpcc** works only Unix like operatins systems. The compiler has been both tested under Ubuntu 20.10, Ubuntu 20.04, and macOS 10.15.

The compiler was developed by his author using an Ubuntu 20.10 machine, while the other distro/OS were tested thanks to Github Actions automated build-check cycles. Windows builds failed due to MSVC rejecting the source code of **dpcc** cause it contains some GCC extensions and some hard coded unix syscalls. In short words **dpcc** can be only compiled with either GCC or CLANG compilers and executed in a Unix/Posix compatible operating system.

The compiler can and should be invoked from the commandline. The **dpcc** executable is self contained and doesn't reach for external assets file and thus can be placed anywhere in the system and invoked from anywhere in the system.

From now on we assume the user has a fired up shell correctly **cd**-ed to the directory holding the **dpcc** executable:

To call the compiler invoke, which will print it's usage help message:

```
./dpcc
```

The `-o` flag is mostly used to override the location of where output files/stream are generated if any.

**dpcc** can work in 6 different modes: **lex, parse, 3ac, c, gcc, run**:

- `./dpcc lex <input> [-o <out>]`: Lex the input and show list of tokens in either stdout or in the given file
- `./dpcc parse <input> [-o <out>]`: Parse the program and produce a text representation of the AST in either stdout or in the given file.
- `./dpcc 3ac <input> [-o <out>]`: Parse the program and perform additional type validations and type checking. If the program is still valid emit 3AC in either stdout or in the given file
- `./dpcc c <input> [-o <out>]`: Same as 3AC but also emit preamble and postamble required to produce a semantically valid C program that can be compiled. The output is emitted in either stdout or in the given file
- `./dpcc gcc <input> [-o <out>]`: Same as `'dpcc c'` but the generated C program is piped into GCC standard input and the final executable is either compiled in `a.out` or in the given filepath. This requires GCC to be in the path.
- `./dpcc run <input>`: Parse, typecheck, emit the C code, compile it and run it in one single command. The executable produced by GCC is outputted in a temp file (`/tmp`), the temp executable is executed right away and the removed. The `-o` flag is ignored. This requires GCC to be in the path.

`Lex` and `parse` mode are mostly used for debugging the produced tokens and the AST. The `run` mode is the most convenient mode since it takes care of everything. If the input program is valid and you call `dpcc run` you will see the output generated from you **DPL** program.

# 3 The proposed implementation

In the following sections it is described how the proposed

- **Input Stage**. The compiler only supports file. The input stream is implemented in the following way. The file is loaded and entirely copied into a memory buffer. This is more than adequate for what it's necessary to do for the project assignment.
- Un **Lexer/Scanner** per la tokenizzazione del sorgente da caratteri a tipi di dati strutturati. La scelta ricade su **Flex** per la gestione dell'analisi lessicale
- Un **Parser** per implementare la sintassi del linguaggio e gestire l'analisi sintattica. La scelta ricade su **Bison** per la gestione di questa componente
- Un semplice **generator di codice intermedio**. Il progetto provede di generare un ibrido Assembly/C/3AC come semplice esempio di gestione di generazione

# 4 Implementation Details

## 4.1 Custom log is implemented to override default bison behaviour

## 4.2 Common types used withing the compiler

## 4.3 Most notable functions used in the compiler

## 4.4 How the AST is traversed

## 4.5 How type checking and type deduction is implemented

## 4.6 A brief introduction at the code implementing the compiler

- The order of the childs in each node **is very important** and it drives the correctness of the final output. In this way we can use simple printf to generate the code each time we find something while traversing the tree

## 4.7 Typescript used for code generation of some part of the compiler

## 4.8 Encountered problems

## 4.9 Testing framework

# 5 Problem analysis

...

# 6 Program design

...

# 7 Evaluation of the program

...

# 8 Process description

...

# 9 Conclusions

...

# 10 Appendix A: Structure of the Intermediate Code

# 11 Appendix: program text

```
let a: int[] = 10;
let s = "Hello world";
{

}
```

```
int main() {

}
char **argv;
```

# 12 Appendix B: Example Program Iterative Merge Sort

```
let array = [
    15, 59, 61, 75, 12, 71, 5, 35, 44,
    6, 98, 17, 81, 56, 53, 31, 20, 11,
    45, 80, 8, 34, 71, 83, 64, 28, 3,
    88, 50, 48, 80, 5
];


for (let curr_size = 1; curr_size < len; curr_size = 2 * curr_size) {
    for (let left_start = 0; left_start < len - 1; left_start = left_start + 2 * curr_size) {
        let mid = len - 1;

        if ((left_start + curr_size - 1) < len - 1) {
            mid = left_start + curr_size - 1;
        }

        let right_end = len - 1;

        if ((left_start + 2 * curr_size - 1) < len - 1) {
            right_end = left_start + 2 * curr_size - 1;
        }

        {
            let l = left_start;
            let m = mid;
            let r = right_end;
            let n1 = m - l + 1;
            let n2 = r - m;

            let L: int[1024];
            let R: int[1024];

            for (let i = 0; i < n1; i++) {
                L[i] = array[l + i];
            }

            for (let i = 0; i < n2; i++) {
                R[i] = array[m + 1 + i];
            }


            let i = 0;
            let j = 0;
            let k = l;

            while (i < n1 && j < n2) {
                if (L[i] <= R[j]) {
                    array[k++] = L[i++];
                } else {
                    array[k++] = R[j++];
                }
            }

            while (i < n1) {
                array[k++] = L[i++];
            }
            while (j < n2) {
                array[k++] = R[j++];
            }
        }
    }
}

print("Sorted array\n");
print(array);
```