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 implemented 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

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 delcarations (usage of the keyword **let**) 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.

```
// 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;

// 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");
    }
}
```

So as the reader can see from above there's not much fancy stuff about the syntax, it is mostly C-alike. The cool thing though about this implementation is that we can mostly declare variables with a simple 'let' keyword like in Javascript, but unlike Javascript the compiler has a way to deduce types whenever possible and still provides strong type semantics to each indentifier like in C (or any other strongly typed language).

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

This section shows an high level of an user calling a built compiler to run and or test

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);
```