Processamento de Linguagens e Compiladores (3º Ano LCC)

Project 2

Project Report

Bruno Dias da Gião
A96544
a96544@alunos.uminho.pt

Maria Filipa Rodrigues
A97536
a97536@alunos.uminho.pt

December 30, 2022

Resumo

O processo de compilação de uma linguagem de programação é um problema de extrema importância e de elevada complexidade. Através de Lex e Yacc adaptados a Python, o Projeto que este relatório documentará demostrará o processo do reconhecimento de uma linguagem inspirada em ANSI C/ALGOL 60 e da geração de codigo de uma máquina virtual de stack, relativamente mais simples que uma máquina real, a partir dessa linguagem.

Abstract

The process of compiling a programming language is a problem of extreme importance and of increased difficulty. Through the use of Python adapted Lex and Yacc, the Project this report intends to document will demonstrate the process of recognizing an ANSI C/ALGOL 60 inspired language and the generation of code for a stack based Virtual Machine, which is relatively simpler than a real machine, from the created language.

Contents

List of Figures

Chapter 1

Report

§1.1 Introduction

1.1.1 The "Not-Quite-C" language Compiler

Introduction to the Report

The present document introduces a program that compiles text from a simplified version of C, as according to the C99 standard, and ALGOL 60 into a stack based virtual machine that can be accessed via the World Wide Web [6] or in any UNIX-based machine [13] and whose's documentation can be found translated to portuguese by University of Minho's Language Processing and Specification Group [5].

The current chapter, chap:1, was structured with some goals in mind, where each section is representative of such goals:

1. In the Introduction, §1.1, we introduce and provide context to both the report and the project as it is presented.
2. In the Methodology section, §1.2, we present some contextualizing theory, the thought process that guided the elaboration of the Project and the State of the Art itself as it is presented. This is done with the hope of helping the reader best understand how this project was solutioned.
3. With the Analysis section, §1.3, we hope to "prove" very loosely but with inteligently chosen examples that show the correct functioning of the developed compiler.

   Note, we chose to limit our proof of compiler correctness to well chosen examples as the Formally correct method of verifying a compiler is a well known complex and extensive problem, resulting, thus in long proofs by derivation that would steal from the purpose of this report and far exceed the scope of the project. Thus, such a Formal Verification is best left for future work. [9] [10]
4. This chapter finishes with a Conclusion, §1.4, where we deem this report as terminated,as is costumary for any document of this format, with our thoughts on our work and future work, §1.4.1, considerations.

Following the report, this document comes annexed with the source code for the project's solution, chap:2.

As is costumary, a bibliography is also annexed at the very end of the present document.

In order to ease reader comprehension of this report, we have opted by the paradigm of "Literate Programming" in which the code shall always be accompanied by an explanation of the code wherever it is deemed necessary. In practice, what happens is, whenever a code segment is referenced it shall be presented as is in the code and a detailed explanation shall follow, in such a way where understanding of the program comes from reading directly the code and reading our thought process and explanations.

Historical background of the ALGOL and CPL families, B and C

While the first programming language was indeed FORTRAN, however, between FORTRAN and C, the differences are immense, so, in order to best analyse the history of this language we must look to ALGOL-58, Algorithmic Language.
ALGOL-58, a standard developed in 1958, one year after FORTRAN by the Association for Computing Machinery and has had 2 major revisions, ALGOL-60 and ALGOL-68, the latter of which was met with severe criticism [8], mainly due to it being compared to it's predecessor, which is the Language we shall be analysing, ALGOL-60, even though, as a member of the ALGOL family it is not short of elements that greatly inspired the programming world.
ALGOL-60 introduced many of the features we now associate with C and with coding in general. Namely:

- Composition operator, i.e., ';'.
- Code blocks in the form of begin/end.
- Chain assignments.
- Recursion was disallowed by FORTRAN and COBOL, where ALGOL-60, thus, first allowed it.[1]

Let us then look at how the infamous Ackermann function would be implemented in C and in ALGOL-60.

```c
#include <stdio.h>

int ack(m,n)
        int m,n;
{       int ans;
        if (m == 0)
                ans = n+1;
        else    if (n == 0)     ans = ack(m-1,1);
                else            ans = ack(m-1, ack(m,n-1));
        return (ans);
}

int main(argc, argv)
        int argc;
        const char ** argv;
{       int i,j;
        for (i=0; i<6; i++)
                for (j=0;j<6; j++)
                        printf("ackerman (%d,%d) is: %d\n",i,j, ack(i,j));
        return (0);
}
/*      The usage of K&R syntax is done on purpose to compare to the ALGOL
        code */
```

Figure 1.1: K&R (pre-ISO) C implementation of the ackermann function

(Please see next page for the ALGOL code)

[1]Note that, despite LISP's John McCarthy having his language specified in 1960 and natively allowing recursion, LISP's first compiler was only released in 1962 [12]

```
BEGIN
        INTEGER PROCEDURE ackermann(m,n);VALUE m,n;INTEGER m,n;
        ackermann := IF m=0     THEN n+1
                ELSE IF n=0     THEN ackermann(m-1,1)
                                ELSE ackermann(m-1,ackermann(m,n-1));
        INTEGER m,n;
        FOR m:=0 STEP 1 UNTIL 3 DO
        BEGIN
                FOR n:=0 STEP 1 UNTIL 6 DO
                        outinteger(1,ackermann(m,n));
                outstring(1,"0)
        END
END
```

Figure 1.2: ALGOL 60 implementation of the ackermann function

We might notice how procedures in ALGOL 60 are not terminated by return keywords or GOTOs or any of the like, such as BASIC, FORTRAN and COBOL, instead, we say that "the procedure ackermann is assigned the value that is computed in this conditional expression", ALGOL procedures only return to the calling procedures if and only if there are no more statements to execute, in other words, ALGOL is, by design, a structured, procedural, imperative language, following the principles of what would be called "Single Entry, Single Exit". [4]

It is clear to see why, when the University of Cambridge need a language in order to expand on to bring wider industrial applications, they were inspired by ALGOL 60 [1]. This language, however, was not very popular and had severe issues, namely relying on symbols that are not widespread in many systems, such as the section symbol (§), and, was thus superceeded by a much simpler language for compiler systems programming, BCPL which would in turn influence Bell Labs' Ken Thompson's first language, the B Programming Language.

The B Programming Language is a typeless language where variables were always words, but, depending on context, could be an integer or a memory address[2]. With the need for user specified and varying internal types, the very same people in Bell Labs would develop their flagship programming language, the C Programming Language.

The C programming language barely requires introduction, it's impact on the computing world has been tremendous, from the development of UNIX to the development of most languages used today, C has been on the front of all. C is a structured, procedural, imperative language, just like ALGOL 60. However, it does allow for Multi-Paradigm programming.

Historical background of Lex, Yacc and PLY

Yacc (Yet Another Compiler-Compiler) is a program for UNIX operating Systems, that generates LALR parsers based on a formal grammar in the BNF format. It was developed using B and later adapted to C. Lex is a lexical analyzer generator for UNIX operating Systems as well, thus complementing YACC by saving easing the exhausting process of writing a lexer in C, that is much more simplified by a tool such as Lex. [11]

The tool we use for this project, however, is PLY (Python Lex & Yacc) that simplifies the process of writing Lex and Yacc code even further by allowing for it to be done in Python with some other quality of life improvements. [3] [2]

[2]References/Pointers were introduced in 1968 with ALGOL 68 [14] [7]

Importance of this project

The specification of a language is an extremely important topic as it is a both complex and enriching subject, testing one's capabilities to understand input recognition, text filters and the generation of the appropriate code the machine may need to compute what was passed as input.

Background of the Project

The project this report documents is a class project for the third year curricular unit, Compilers and Language Processing, where we were prompted to develop a compiler for a language we would create that featured typeless variables that could be declared, attributed values to, control flow statements, loop statements, one dimensional and two dimensional arrays and procedures that received no arguments and returns a single Integer type expression or variable.

Expansions of the Project

In hopes of boosting the quality and utility of the project, our group decided to incorporate some extra features such as instead of procedures that receive no arguments, one can have arguments in their subroutines, these arguments can be passed by value or by reference; in order to incorporate arguments by reference, pointers are also implemented; because pointers are specified, one might find the need for a pointer that points to "nothing" for with the NIL pointer was incorporated, however to maintain modularity, this was done via pre-processor definitions, instructions that replace given words in the code with a value or expression, i.e. C's define; of course, having implemented pre-processor capabilities one might also implement user custom types, in the form of compound data, i.e. records and unions; we also decided it would be an interesting idea to implement floating-point support and bitwise operations, however the latter is difficulted by machine limitations.

§1.2 Methodology

1.2.1 Theoretical Background

CFG

Compiler

Lexer

Parser

1.2.2 Practical component

PLY.LEX

PLY.YACC

§1.3 Analysis

1.3.1 Expected Results

1.3.2 Testing the generated code

§1.4 Conclusion

1.4.1 Future Work

Chapter 2

Appendix

§2.1 Code

§2.2 Grammar

Bibliography

[1] D. W. Barron, J. N. Buxton, D. F. Hartley, E. Nixon, and C. Strachey. The Main Features of CPL. *The Computer Journal*, 6(2):134–143, 1963.

[2] David Beazley. Documentation for PLY. `http://www.dabeaz.com/ply/ply.html`.

[3] David Beazley. WebPage for PLY. `http://www.dabeaz.com/ply`.

[4] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press Ltd., GBR, 1972.

[5] EPLDIUM. Portuguese Documentation for the Virtual Machine VM. `https://eplmediawiki.di.uminho.pt/uploads/Vmdocpt.pdf`.

[6] EPLDIUM. Web version of the Virtual Machine VM. `https://ewvm.epl.di.uminho.pt`.

[7] CAR Hoare. A note on indirect addressing. *ALGOL Bulletin*, 21:75–77, 1965.

[8] CAR Hoare. Critique of ALGOL 68. *ALGOL Bulletin*, 29:27–29, 1968.

[9] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of The ACM*, 52(7):107–115, 2009.

[10] Xavier Leroy. Formally verifying a compiler: what does it mean, exactly? Talk at ICALP, 2016.

[11] Tony Mason and Doug Brown. *Lex & Yacc*. O'Reilly & Associates, Inc., USA, 1990.

[12] John McCarthy. *History of LISP*, page 173–185. Association for Computing Machinery, New York, NY, USA, 1978.

[13] Christine Paulin-Mohring. Source for the Virtual Machine VM. `https://www.lri.fr/~paulin/COMPIL/introduction.html`.

[14] A. van Wijngaarcien, B. J. Mailloux, J. E. L. Peck, C. H. A. Kostcr, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language algol 68. *SIGPLAN Not.*, 12(5):1–70, 1977.