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

2023/01/15

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

Listings

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 [14] 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. [10] [11]
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 its 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 [13]

```
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, Dennis Ritchie would develop Bell Lab's programming language, the C Programming Language [9].

The C programming language barely requires introduction, its 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.

Indeed, one would be doing a disservice to history by not mentioning the primary reason C was first developed, a timesharing Operating System, UNIX [15]. The extremely influencial operating system that would eventually come to be used worldwide[3] and bring regular expressions to the mainstream use of today.

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

[2] References/Pointers were introduced in 1968 with ALGOL 68 [16] [7]

[3] Mac-OS and Linux are both UNIX-like OS

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. [12]

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]

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

A context free grammar, or CFG, is a formal grammar such that:

$$A ::= \alpha, A \in NT, \alpha \in NT \cup T \cup \{\epsilon\} \tag{1.1}$$

This concept is what drives the grammar of a parser and a lexer.

1.2.2 Practical component

The Source code for this project is divided into two files, lexer.py and parser.py, using PLY.LEX and PLY.YACC, respectively.

PLY.LEX

Following is an explanation of the role the lexer has in our compiler and the tokens used and the assigned meanings: [4]

Tokens From the source code of the lexer, one may find 14 reserved words and 27 non-reserved tokens, the reserved words are (effectively) special cases of identifiers. As such, the NQC Programming Language accepts strings that are: standard C binary operators; identifiers that contain only alphabetic characters and positive integer numbers.

Role of the Lexer Our lexer has a very minimal overall role on the compiler itself, only serving as a tokenizer and counting lines. Some identifiers however are given special meanings via the reserved words' associative array.

PLY.YACC

Finally, putting an end to the Implementation component of the chapter, we reach, what turns out to the be most important part of this solution: the parser. [5]

Interpreting the translation grammar Starting the interpretation at the **Axiom**, a Program is a set of functions, however, most importantly, this production allows for the generation of the so called **Calling Function** a role that is, in C, performed by the Operating System and whose function is to both call main and exit the program depending on the exit code received.
All functions have a **header** and a **body**.
Each **function header** is defined by a return type, a name, and its arguments, which translates into an update to the identifier table with the following contents:

- The function as a 'function' with data relating to arguments and return type;
- Each argument as variables defined locally in the scope of said function;
- The function as a 'variable' which shall be the equivalent to the `%eax` register in 'x86' assembly.[6]

Having done these operations, the parser will then write a **LABEL** using the name of the function.
The **function body** is a set of **variable declarations** followed by **code logic**.
Variable declarations have a similar structure and follow a similar pattern for all data types and forms of declarations, namely: updating the identifier table and pushing either an integer or a pointer to the stack, the integer will always be zero and the pointer will either be NIL, the location of the first element in the case of arrays and the location of the first element in each row in the case of a matrix.
Code logic is a set of atributions, control flows and function calls in no particular order.
Atributions are a very special production as the compiler cannot know the value of each variable, however, we do know each variable's **relative** address, that is, it's offset to the **base pointer**, and it's that knowledge that guides each atribution by computing an expression and then using the 'store' instruction from the VM ISA.
Expressions and conditional expressions are a concept whose grammar was directly taught in class, thus it's relevance in this document is not primary, even the generation of code is limited to, once again, working around the address of the variable and using the VM ISA to obtain the content of a variable or result of a computation. **Control Flow** is handled by a complementary variable that keeps track of the quantity of non function labels already in the result, this is only used for naming the labels.

[4] The reader may find the lexer in the Appendix, §2.1
[5] The grammar and the parser can both be found in the Appendix, §2.1 and §2.2.
[6] Only if the function does not return 'VOID'

Most importantly, NQC allows for nesting these structures by implementing a code logic in the scope of each conditional structure.
Invoking procedures is no more than the task of comparing the arguments received with the arguments in the identifier table's entry for the function and pushing each argument, from last to first to the stack and, finally calling the subroutine.
These are the main observations that can be understood from the studying the translation grammar for this program, indeed, the most important concept to take of note is the use of **local addresses**, forcing the implementation of pointers in order to manipulate data out of the scope of a subroutine. Indeed, this concept is vitally important to understanding this solution and most if not all the decisions taken for this implementation.

Identifier table The identifier table can be expressed as an associative array of structs of unions. To exemplify, let us look at this C implementation of a content of an identifier table:

```
struct identifier{
    char*class;
    union {
        struct{
            char*address;
            char*type;
            char*size;
            char*scope;
        };
        struct{
            char**args;
            char*return;
        };
    };
};
```

This concept is directly implemented in a Python Dictionary.

§1.3 Analysis

1.3.1 Language Reference Guide

The 'Not-Quite-C' Programming language can be explained very easily as it is a simplistic and not nearly as (although attempting to be) robust as the C programming language. It allows for explicit control of memory, albeit limited to integers and arrays of integers. As such:

Bases One can start a program very simply by invocking the following format of code:

```
INT MAIN()
BEGIN
    /* Program code */
END
```

The **MAIN** subroutine is obligatory and failure to include or the mistype of the procedure will result in a compilation failure. The source file must always end on an empty line.

Of course, one cannot do without variables, as such all declarations are included, by design, at the start of **each procedure**.
These variables, which are always integers shall be initialized as 0, **unless** these are pointer variables, which are always initialized to NIL, a pre-processor define to represent a location in memory that will never by accessed by the program. To initialize with a *different value*, one must atribute one such value to the variable.
Let us then exemplify these concepts:

```
INT MAIN()
BEGIN
    /* Declarations */
    INT variable;
    REF INT pointervar;
    /* Code logic */
    variable := 10;
    pointervar := &variable;
    DEREF pointervar := DEREF pointervar + 10;
    /* Rest of Code */
END
```

Note how we had two assignments using the 'pointervar' identifier, since this identifier represents a **pointer variable**, it holds that its content must be an address[7], thus, we use the '&' operator to obtain the address of 'variable' and then, in order to atribute a new value to 'variable' by reference, we must use the 'DEREF' operator and, to access the value of the variable that 'pointervar' is referencing, one must also use 'DEREF', thus, this operator serves as both a means to store and a means to peek at the current value of the referenced variable.

Pointers As it stands, this instruction is trivial and passing by reference is unecessary, which thus brings up the question, why? Indeed, the NQC Programming Language, much like C, works entirely dependant on the local scope of any 'variable', in other words, how may we access the contents of a variable that is not locally defined?
Exactly in the same manner as the C Programming Language, by passing the variable by reference.

```
INT SWAPF(REF INT px, REF INT py)
BEGIN
  DEREF px := DEREF px * DEREF py;
  DEREF py := DEREF px / DEREF py;
  DEREF px := DEREF px / DEREF py;
END
INT MAIN()
BEGIN
  INT x; INT y;
  x:= 10; y:= 20;
  SWAPF(&x,&y);
  MAIN:=0;
END
```

In this example, we perform the swap algorithm for integers, now what would happen if we passed px and py by value? Indeed we would swap the values of the parameters, however, these parameters are no more than '**copies**' of

[7]The validity of the address is the user's responsability

the desired variables, thus, by knowing their address via pointers, we can alter these from 'anywhere'.
What if perhaps, we desired to perform some conditional programming? The NQC Programming Language is equipped with the following control flow statements: 'IF-ELSE','WHILE-REPEAT','UNTIL-REPEAT','DO-WHILE' and 'DO-UNTIL'.

Data Structures and Control Flow To exemplify these structures let us introduce also the concept of data structures. The NQC Programming Language only contains the most basic data structure, the array. Let us then consider the following implementation of the bubble-sort algorithm, let us also suppose 'SWAPF' from before is defined:

```
VOID BSORT(REF INT arr, INT N)
BEGIN
  INT i; INT j;
  i:=N-1;j:=i;
  WHILE (i >0)
  BEGIN
    j:=0;
    WHILE (j < i)
    BEGIN
      IF (arr[j] > arr[j+1])
      BEGIN
        SWAPF(&arr[j],&arr[j+1]);
      END
      j:=j+1;
    END
    i:=i-1;
END
INT MAIN()
BEGIN
  INT arr[3];
  arr[0]:=2;
  arr[1]:=-20;
  arr[2]:=-5;
  BSORT(arr,5);
  MAIN:=0;
END
```

Important observations, BSORT takes a pointer to an integer, yet we only pass an INT, arr, as argument, well, because arr is an array, 'INT name[]' is always interpreted as a 'REF INT name', thus we need not dereference the array. Another aspect that may peek the reader's interest is the nesting of conditional blocks, nesting should however be done with great care as 'breaking' out of a loop is not an allowed instruction.

Matrix A matrix can be declared as such:

```
INT
MAIN()
BEGIN
    INT MAT[10,10]; /* declaring mat of size 100 */
```

```
    INT I;INT J;
    MAT[I,J]:= 4; /* is indexing at I-row and J-col */
    MAIN:=0
END
```

This is a very similar implementation to that of the one dimensional array, thus, it requires little introduction.

Array to Pointer decay Let us look towards this last observation, indeed, we may conclude that undefined behaviour is very likely, as BSORT will accept a **Pointer to an integer** even if it is not an array, thus care is indeed required.

Using the Compiler Having written a program, one can run one of the following UNIX commands:

```
$ parser.py <name_of_file>.nqc
$ parser.py <name_of_file>.nqc -o <new_file>.vm
```

If the first command is used, the result of the parsing is printed to STDOUT, in usual UNIX fashion, otherwise, it is printed directly into the given file. [8]

1.3.2 Expected Results

In order to best analyse our results, let us first prompt ourselves with a few possible procedures that will guide our examplifications, namely, the Swap function, the infamous Ackermann function, an implementation of the Bubble Sort algorithm and an implementation of the Factorial Function.

```
VOID
SWAPF(REF INT PX, REF INT PY)
BEGIN
   DEREF PX := DEREF PX * DEREF PY;
   DEREF PY := DEREF PX / DEREF PY;
   DEREF PX := DEREF PX / DEREF PY;
END

INT
A(INT M, INT N)
BEGIN
   IF (M = 0) BEGIN A := N+1; END
   ELSE BEGIN IF (N = 0) BEGIN A := A((M - 1),1); END
              ELSE BEGIN A := A(M-1,A(M,(N-1))); END
   END
END

VOID
BS(REF INT AR, INT N) /* Bubble Sort */
BEGIN
   INT I;
```

[8] Supposing that parser.py is being ran on a machine using UNIX and that the correct priviledges are given to the parser, otherwise, regular usage is advised

```
   INT FLAG;
   FLAG:=1;
   UNTIL (!FLAG)
   BEGIN
      FLAG:=0;
      I:=0;
      WHILE (I < (N-1))
      BEGIN
         IF (AR[I] > AR[I+1]) BEGIN SWAPF(&AR[I],&AR[I+1]); FLAG:=1; END
         I:=I+1;
      END
   END
END

INT
F(INT N) /* Factorial function */
BEGIN
   INT I;
   F := 1;
   UNTIL ((N-I) <= 0)
   BEGIN
      I:=I+1;
      F:=F*I;
   END
END
```

Having defined these subroutines, let us try to exemplify and predict the behavior the NQC Programming Language would have when computing these procedures. As such let us define the MAIN function of this program.

```
INT
MAIN()
BEGIN
   INT RES;
   INT ARR[2];
   ARR[0]:=10;
   ARR[1]:=-25;
   BS(ARR,2);
   WRITEI(ARR[0]); WRITES("\n");
   WRITEI(ARR[1]); WRITES("\n");
   RES:=A(1,1);
   WRITEI(RES); WRITES("\n");
   RES:=F(2);
   WRITEI(RES); WRITES("\n");
   WRITEI(ATOI(READ()));
   MAIN:=0;
END
```

Trivially computing these values by hand, we have that this program must output:

```
-25
10
3
2
```

1.3.3 Testing the generated code

Having predicted the output, let us run our compiler and analyse the generated assembly pseudo-code, located in the Appendix, in §2.3. Indeed, if this is ran in the Virtual Machine, the output previously predicted will be shown.
Note how these examples are carefully picked for each of them represent a certain concept within computer science that was touched on or mentioned previously, recursion via the Ackermann Function implementation, simple control flow via the imperative factorial implementation, passing variables by reference and handling levels of indirection via the Bubble Sort and Swap implementations. Now there are some features that were not shown in this example however, many more examples will be included in the Appendix, §2.3, all with corresponding generated code.

§1.4 Conclusion

Overall, this project was one that aptly tested both our creativity, practical capabilities and theoretical understanding of the formal languages.
Indeed, this translated into a beautiful, albeit long, program that successfully performs exactly what was prompted and more.
By allowing for at most two levels of indirection we have a, although rugged, precise control of the machine's memory. What results is a beautiful programming language that motivates the usage of **correct programming practices**, such as **Structured Programming**.

1.4.1 Future Work

Of course, due to the amount of features implemented, there are some that were left out, and some behaviors that are not defined, something that can be protected against, or left in. Indeed, much like the C Programming Language, what we have presented in this document is a language that can be evolved into a more robust and powerful programming language via, implementation of compound data and pre-processor capabilites, something that was only 'mimicked' in the implementation of the **NIL** pointer, or into a simpler language by 'hiding' the levels of indirection available. Which in itself is being "held" unto by a lot of hard-coded segments. It would be preferable to, instead, allow to recursively recognize multiple levels of indirection, multiple data types such as floating point variables, char variables, etc.
The NQC Programming Language is by no means a "complete" language, as such, a lot of work is required until these features are satisfied, indeed, it would also be interesting to perform the same tasks in a more "realistic context", in other words, by implementing one's own parser and lexer for a **Real Machine**, allowing for choice between a bottom up or a top down parser, and allowing for better efficiency by not requiring several levels of compiling in order to actually assemble the program.

Chapter 2

Appendix

§2.1 Code

Listing 2.1: NQC Compiler's Lexer

```
"""
    PROJECT 2022/2023
"""
import sys
from ply import lex


reserved = {
        'IF'      : 'IF','ELSE'       : 'ELSE',
        'WHILE'   : 'WHILE','INT'     : 'INT',
        'STR'     : 'STR','REF'       : 'REF',
        'DEREF'   : 'DEREF', 'UNTIL'  : 'UNTIL',
        'DO'      : 'DO', 'VOID'      : 'VOID',
        'WRITES'  : 'WRITES','WRITEI' : 'WRITEI',
        'ATOI'    : 'ATOI','READ'     : 'READ'
}

# List of Tokens
tokens = [
        'NUMBER','SUM','MULT','DIV','MODULO','SUB',
        'ID',#'XOR','AND','OR','SHIFTLEFT','SHIFTRIGHT',
        'NOT','GEQ','LEQ','DIF','EQ','LESSER','GREATER',
        'CONDAND','CONDOR','ATRIB','COMP','ARRCONT',
        'LPAREN','RPAREN','ARRINDL','ARRINDR','BLOCK_START',
        'BLOCK_END','STRING','ADDR'
] + list(reserved.values())

########### INTEGER ARITHMETIC #############
t_SUM    = r'\+';t_MULT   = r'\*'
t_DIV    = r'\/';t_MODULO = r'\%'
t_SUB    = r'\-'
########## BITWISE ##################
#t_XOR = r'\^';t_AND = r'\&'
#t_OR  = r'\|'
#t_SHIFTLEFT = r'\<\<';t_SHIFTRIGHT = r'\>\>'
```

```
36 ########### BOOLEAN #################
37 t_GEQ = r'\>\=';t_LEQ = r'\<\='
38 t_DIF = r'\!\=';t_EQ = r'\='
39 t_LESSER  = r'\<';t_GREATER = r'\>'
40 t_CONDAND = r'\&\&';t_CONDOR = r'\|\|'
41 t_NOT = r'\!'
42 ######### SYNTAX RELATIVE SYMBOLS ##########
43 t_ATRIB      = r'\:\=';t_COMP     = r'\x3B' # ;
44 t_ARRCONT    = r'\x2C' # ,
45 t_ARRINDL    = r'\x5B' # [ Indexing arrays translates to load or store
46 t_ARRINDR    = r'\x5D' # ] Indexing arrays translates to load or store
47 t_ADDR       = r'\&'
48
49
50 t_LPAREN     = r'\x28' # (
51 t_RPAREN     = r'\x29' # )
52 #t_BLOCK_START = r'BEGIN\n';t_BLOCK_END = r'END\n'
53 #t_BLOCK_START = r'\{';t_BLOCK_END = r'\}'
54
55 def t_STRING(t):
56     r'\".*\"';t.type = reserved.get(t.value, 'STRING'); return t
57 def t_COMMENT(t):
58     r'\/\*(.|\n)*?\*\/'; pass
59     # Ignores everything between /* */
60
61 def t_NUMBER(t):
62     r'\d+'
63     t.value = int(t.value); return t
64
65 def t_BLOCK_START(t):
66     r'BEGIN';return t
67 def t_BLOCK_END(t):
68     r'END';return t
69 def t_ID(t):
70     r'[A-Za-z]+';t.type = reserved.get(t.value, 'ID'); return t
71
72 def t_newline(t):
73     r'\n+'
74     t.lexer.lineno += len(t.value)
75
76 t_ignore = '\x20\t' # Spaces and Tabs
77
78 def t_error(t):
79     print(f"Illegal character {t.value[0]}")
80     # t.lexer.skip(1)
81
82 lexer = lex.lex()
83
84 if __name__ == '__main__':
85     with open(sys.argv[1], 'r', encoding='UTF-8') as file:
86         cont = file.read()
87
88     lexer.input(cont)
89     token = lexer.token()
```

```
90      while token:
91          print(token)
92          token = lexer.token()
```

Listing 2.2: NQC Compiler's Parser

```
#! /bin/python3
"""
    PROJECT
"""
import sys
import re
from ply import yacc
from lexer import tokens

def p_program(p):
    'program : functions'
    if parser.success:
        p[0] = p[1]
        parser.result = 'calling: nop\n\tstart\n\tnop\n\tpushi 0'
        parser.result += '\n\tpusha MAIN\n\tcall\n\tnop\n\tdup 1\n\tnot\n'
        parser.result += '\tjz L0\n\tnop\n\tpop 1\n\tstop\nL0:\n\tpushs "Exited with code "'
        parser.result += '\n\twrites\n\twritei\n\tpushs "\\n"\n\twrites\n\tstop\n'+p[0]
def p_functions_1(p):
    'functions :  '
    if parser.success:
        if 'MAIN' not in parser.namespace.keys():
            print(f"ERROR: Lacking a MAIN function!",
                  file=sys.stderr)
            parser.success = False
        if parser.success:
            p[0] = '\n'

def p_functions_2(p):
    'functions : function functions'
    if parser.success:
        p[0] = p[1] + p[2]

def p_function(p):
    'function : function_header function_code_outline'
    if parser.success:
        p[0] = p[1] + p[2]

def p_function_header(p):
    'function_header : func_type ID argument_list_head'
    parser.currentfunc = p[2]
    if parser.success:
        name = p[2]
        args = p[3]
        r_type = p[1]
        if name == 'MAIN':
            if (r_type != 'INT' or args != []):
                print('ERROR: Incorrect type for MAIN',
                      file=sys.stderr)
                parser.success = False
            if parser.success:
                parser.namespace['MAIN'] = {'class':'funct',
```

```
52                                      'arguments':[], 'return':'INT'}
53                 parser.namespace['MAIN1'] = {'class':'var',
54                                              'address' : '-1',
55                                              'type'    : 'INT',
56                                              'size'    : '0',
57                                              'scope'   : 'MAIN'}
58         else:
59             if name in parser.namespace:
60                 print("ERROR: Name already used",
61                       file=sys.stderr)
62                 parser.success = False
63             if parser.success:
64                 try:
65                     parser.namespace[name] = {'class':'funct',
66                                 'arguments':args.split(','),'return':r_type}
67                     for elem in args.split(','):
68                         stuff = elem.split(' ')
69                         data = ' '.join(stuff[:-1])
70                         var_name = stuff[-1]
71                         parser.argnum -= 1
72                         parser.namespace.update({var_name : {
73                             'class'   : 'var',
74                             'address' : str(parser.argnum),
75                             'type'    : data,
76                             'size'    : '0',
77                             'scope'   : parser.currentfunc,
78                         }})
79                 except AttributeError:
80                     parser.namespace[name] = {'class':'funct',
81                                 'arguments':[],'return':r_type}
82                 if r_type != 'VOID':
83                     parser.namespace[name+'1'] = {'class' : 'var',
84                                 'address' : parser.argnum-1,
85                                 'type'    : r_type,
86                                 'size'    : '0',
87                                 'scope'   : parser.currentfunc
88                     }
89         if parser.success:
90             parser.argnum = 0
91             parser.varnum = 0
92             p[0] = name + ':\n\tnop\n'
93
94 def p_argument_list_head_1(p):
95     'argument_list_head : LPAREN RPAREN '
96     if parser.success:
97         p[0] = []
98 def p_argument_list_head_2(p):
99     'argument_list_head : LPAREN arg_head args_head RPAREN'
100    if parser.success:
101        p[0] = p[2] + p[3]
102
103 def p_arg_head(p):
104    'arg_head : data_type ID'
105    if parser.success:
```

```
106         name = p[2]
107         data = p[1]
108         if name in parser.namespace:
109             if parser.namespace[name]['class'] != 'var':
110                 parser.success = False
111         if parser.success:
112             p[0] = data + ' ' + name
113 def p_args_head_1(p):
114     'args_head :  '
115     if parser.success:
116         p[0] = ''
117 def p_args_head_2(p):
118     'args_head : ARRCONT arg_head args_head'
119     if parser.success:
120         p[0] = p[1] + p[2] + p[3]
121 
122 
123 def p_function_code_outline(p):
124     'function_code_outline : BLOCK_START function_code BLOCK_END'
125     if parser.success:
126         p[0] = p[2]
127 
128 def p_function_code_1(p):
129     'function_code :  '
130     if parser.success:
131         p[0] = ''
132 def p_function_code_2(p):
133     'function_code : declarations code_logic'
134     if parser.success:
135         if parser.varnum:
136             p[0] = p[1] + p[2] + f'\tpop {parser.varnum}\n\treturn\n\tnop\n'
137         else:
138             p[0] = p[1] + p[2] + '\treturn\n\tnop\n'
139 
140 
141 def p_declarations_1(p):
142     'declarations :  '
143     if parser.success:
144         p[0] = ''
145 def p_declarations_2(p):
146     'declarations : declaration declarations'
147     if parser.success:
148         p[0] = p[1] + p[2]
149 
150 
151 def p_declaration_1(p):
152     'declaration : data_type ID COMP'
153     if parser.success:
154         name = p[2]
155         data = p[1]
156         if name in parser.namespace:
157             if parser.namespace[name]['class'] == 'var':
158                 if parser.namespace[name]['scope'] == parser.currentfunc:
159                     print("ERROR: Name already in use!",
```

```
                            file=sys.stderr)
                    parser.success = False
            else:
                print("ERROR: Name already in use!",
                        file=sys.stderr)
                parser.success = False
    if parser.success:
        ind = parser.varnum
        parser.varnum += 1
        parser.namespace.update({name: {
                'class'  : 'var',
                'address': str(ind),
                'type'   : data,
                'size'   : '0',
                'scope'  : parser.currentfunc
        }})
        if data == 'REF INT':
            p[0] = '\tpushgp\n\tpushi 99999\n\tpadd\n'
        else: p[0] = '\tpushi 0\n'

def p_declaration_2(p):
    'declaration : data_type ID ARRINDL NUMBER ARRINDR COMP'
    if parser.success:
        name = p[2]
        data = p[1]
        const = p[4]
        if data != 'INT':
            print("Arrays should be INT",
                    file=sys.stderr)
            parser.success = False
        if name in parser.namespace:
            if parser.namespace[name]['class'] == 'var':
                if parser.namespace[name]['scope'] == parser.currentfunc:
                    print("ERROR: Name already in use!",
                            file=sys.stderr)
                    parser.success = False
            else:
                print("ERROR: Name already in use!",
                        file=sys.stderr)
                parser.success = False
    if parser.success:
        ind = parser.varnum
        parser.varnum += 1 + const
        parser.namespace[name] = {
                'class' : 'var',
                'address': str(ind),
                'type'   : 'REF ' + data,
                'size'   : str(const),
                'scope'  : parser.currentfunc
        }
        p[0] = f'\tpushfp\n\tpushi {ind+1}\n\tpadd\n\tpushn {const}\n'
def p_declaration_bin_arr(p):
    'declaration : data_type ID ARRINDL NUMBER ARRCONT NUMBER ARRINDR COMP'
    if parser.success:
```

```
214         row = p[4]
215         col = p[6]
216         total_size = int(row) * int(col)
217         data = p[1]
218         name = p[2]
219         res = ''
220         if data != 'INT':
221             print("ERROR: Array must be of Integers",
222                     file=sys.stderr)
223             parser.success = False
224         else:
225             if name in parser.namespace:
226                 if parser.namespace[name]['class'] == 'var':
227                     if parser.namespace[name]['scope'] == parser.currentfunc:
228                         print("ERROR: Name already in use!",
229                                 file=sys.stderr)
230                         parser.success = False
231                 else:
232                     print("ERROR: Name already in use!",
233                             file=sys.stderr)
234                     parser.success = False
235     if parser.success:
236         ind = parser.varnum
237         parser.varnum += 1+row+total_size
238         parser.namespace[name] = {
239                 'class' : 'var',
240                 'address' : str(ind),
241                 'type' : 'REF REF ' + data,
242                 'size' : str(total_size),
243                 'cols' : str(col),
244                 'rows' : str(row),
245                 'scope' : parser.currentfunc
246                 }
247         arr = list(range(0,int(row)))
248         for i in range(0,int(row)):
249             if i == 0:
250                 arr[i] = ind+int(col)
251             else:
252                 arr[i] = int(col)+arr[i-1]
253         for i in arr:
254             res += f'\tpushfp\n\tpushi {i}\n\tpadd\n'
255         p[0] = res + f'\tpushn {total_size}\n'
256
257
258 def p_code_logic(p):
259     'code_logic :  '
260     if parser.success:
261         p[0] = ''
262 def p_code_logic_atr(p):
263     'code_logic : atributions'
264     if parser.success:
265         p[0] = p[1]
266 def p_code_logic_cond(p):
267     'code_logic : conditionals'
```

```
268     if parser.success:
269         p[0] = p[1]
270 def p_code_logic_func(p):
271     'code_logic : call_functions'
272     if parser.success:
273         p[0] = p[1]
274 
275 
276 def p_atributions(p):
277     'atributions : atribution code_logic'
278     if parser.success:
279         p[0] = p[1] + p[2]
280 
281 def p_atribution_str(p):
282     'atribution : ID ATRIB STRING COMP'
283     if parser.success:
284         name = p[1]
285         string = p[3]
286         if name in parser.namespace:
287             if parser.namespace[name]['class'] == 'var':
288                 if parser.namespace[name]['scope'] != parser.currentfunc:
289                     print("ERROR: Not declared!",
290                             file=sys.stderr)
291                     parser.success = False
292                 elif parser.namespace[name]['type'] != 'STR':
293                     print("ERROR: Not a string",
294                             file=sys.stderr)
295             else:
296                 if name != parser.currentfunc:
297                     print("ERROR: Not a variable!",
298                             file=sys.stderr)
299                     parser.success = False
300                 else:
301                     if parser.namespace[name]['return'] != 'STR':
302                         print("ERROR: Wrong type",
303                                 file=sys.stderr)
304                         parser.success = False
305                     if parser.namespace[name]['return'] == 'VOID':
306                         print("ERROR: Assigning value to void function",
307                                 file=sys.stderr)
308                         parser.success = False
309         else:
310             print("ERROR: Not declared!",
311                     file=sys.stderr)
312             parser.success = False
313         if parser.success:
314             if name == parser.currentfunc:
315                 address = parser.namespace[name+'1']['address']
316             else: address = parser.namespace[name]['address']
317             p[0] = f'\tpushs {p[3]}\n\tstorel {address}\n'
318 def p_atribution_1(p):
319     'atribution : ID ATRIB expression COMP'
320     if parser.success:
321         name = p[1]
```

```
322         if name in parser.namespace:
323             if parser.namespace[name]['class'] == 'var':
324                 if parser.namespace[name]['scope'] != parser.currentfunc:
325                     print("ERROR: Not Declared!",
326                           file=sys.stderr)
327                     parser.success = False
328                 elif parser.namespace[name]['type'] == 'STR':
329                     print("ERROR: A String cannot be an expression",
330                           file=sys.stderr)
331                     parser.success=False
332             else:
333                 if name != parser.currentfunc:
334                     print("ERROR: Not a variable!",
335                           file=sys.stderr)
336                     parser.success = False
337                 else:
338                     if parser.namespace[name]['return'] == 'STR':
339                         print("ERROR: Mismatch type",
340                               file=sys.stderr)
341                         parser.success = False
342                     elif parser.namespace[name]['return'] == 'VOID':
343                         print("ERROR: Assigning value to void function",
344                               file=sys.stderr)
345                         parser.success = False
346         else:
347             print("ERROR: Not declared!",
348                   file=sys.stderr)
349             parser.success = False
350     if parser.success:
351         if name == parser.currentfunc:
352             address = parser.namespace[name+'1']['address']
353         else: address = parser.namespace[name]['address']
354         p[0] = f'{p[3]}\tstorel {address}\n'
355 def p_atribution_deref(p):
356     'atribution : DEREF ID ATRIB expression COMP'
357     if parser.success:
358         name = p[2]
359         if name in parser.namespace:
360             if parser.namespace[name]['class'] == 'var':
361                 if parser.namespace[name]['type'] != 'REF INT':
362                     print("ERROR: Dereferencing value")
363                     parser.success = False
364                 if parser.namespace[name]['scope'] != parser.currentfunc:
365                     print(f"ERROR: {p[1]} Not Declared!")
366                     parser.success = False
367             else:
368                 print(f"ERROR: {p[1]} Not a variable!")
369                 parser.success = False
370         else:
371             parser.success = False
372         if parser.success:
373             address = parser.namespace[name]['address']
374             p[0] = f'\tpushl {address}\n{p[4]}\tstore 0\n'
375
```

```
376 def p_atribution_3(p):
377     'atribution : ID ARRINDL expression ARRINDR ATRIB expression COMP'
378     if parser.success:
379         name = p[1]
380         ind = p[3]
381         atrib_expr = p[6]
382         if name not in parser.namespace:
383             print("ERROR: Atribution without declaration.",
384                     file=sys.stderr)
385             parser.success = False
386     if parser.success:
387         if (parser.namespace[name]['class'] != 'var'
388                 or parser.namespace[name]['type'] != 'REF INT'):
389             print("ERROR: Malformed indexing.",
390                     file=sys.stderr)
391             parser.success = False
392         else:
393             index = parser.namespace[name]['address']
394             p[0] = f'\tpushl {index}\n{ind}{atrib_expr}\tstoren\n'
395 def p_atribution_4(p):
396     'atribution : ID ARRINDL expression ARRCONT expression ARRINDR ATRIB expression
        COMP'
397     if parser.success:
398         name = p[1]
399         row = p[3]
400         col = p[5]
401         atrib_expr = p[8]
402         if name not in parser.namespace:
403             print("ERROR: Atribution without declaration",
404                     file=sys.stderr)
405             parser.success = False
406         if parser.success:
407             if (parser.namespace[name]['class'] != 'var'
408                     or parser.namespace[name]['type'] != 'REF REF INT'):
409                 print("ERROR: Malformed indexing.", file=sys.stderr)
410                 parser.success = False
411             else:
412                 rows = int(parser.namespace[name]['rows'])
413                 cols = int(parser.namespace[name]['cols'])
414                 index = int(parser.namespace[name]['address'])
415                 p[0] = f'\tpushl {index}\n{row}\tpushi {cols}\n\tmul\n\tpadd\n{col}{
                    atrib_expr}\tstoren\n'
416 def p_indarr_1(p):
417     'indarr : ID ARRINDL expression ARRINDR'
418     if parser.success:
419         name = p[1]
420         const = p[3]
421         if name not in parser.namespace:
422             print(f"ERROR: Indexing without declaration.",
423                     file=sys.stderr)
424             parser.success = False
425     if parser.success:
426         if (parser.namespace[name]['class'] != 'var'
427             or parser.namespace[name]['type'] != 'REF INT'):
```

```
428             print(f"ERROR: Malformed indexing.",
429                     file=sys.stderr)
430             parser.success = False
431         else:
432             index = parser.namespace[name]['address']
433             p[0] = f'\tpushl {index}\n{const}\tloadn\n'
434 def p_indmat_2(p):
435     'indmat : ID ARRINDL expression ARRCONT expression ARRINDR'
436     if parser.success:
437         name = p[1]
438         if name not in parser.namespace:
439             print("ERROR: Indexing without declaration.",
440                     file=sys.stderr)
441             parser.success = False
442     if parser.success:
443         if (parser.namespace[name]['class'] != 'var'
444             or parser.namespace[name]['type'] != 'REF REF INT'):
445             print("ERROR: Malformed indexing.")
446             parser.success = False
447         else:
448             rows = parser.namespace[name]['rows']
449             cols = parser.namespace[name]['cols']
450             index = parser.namespace[name]['address']
451             p[0] = f'\tpushl {index}\n{p[3]}\t\npushi {cols}\n\tmull\n\tpadd\n\t{p
                [5]}\tloadn\n'
452
453
454 def p_expression_1(p):
455     'expression : term'
456     if parser.success:
457         p[0] = p[1]
458 def p_expression_2(p):
459     'expression : expression ad_op term'
460     if parser.success:
461         p[0] = p[1] + p[3] + p[2]
462
463 def p_term(p):
464     'term : factor'
465     if parser.success:
466         p[0] = p[1]
467 def p_term_1(p):
468     'term : term mult_op factor'
469     if parser.success:
470         p[0] = p[1] + p[3] + p[2]
471 def p_factor(p):
472     'factor : NUMBER'
473     if parser.success:
474         p[0] = f'\tpushi {p[1]}\n'
475 def p_factor_id(p):
476     'factor : ID'
477     if parser.success:
478         name = p[1]
479         if name in parser.namespace:
480             if parser.namespace[name]['class'] == 'var':
```

```
481                 if parser.namespace[name]['scope'] != parser.currentfunc:
482                     print("ERROR: Not Declared!",
483                           file=sys.stderr)
484                     parser.success = False
485             else:
486                 if (name == parser.currentfunc and
487                     parser.namespace[name]['return'] == 'VOID'):
488                     print("ERROR: Accessing value of void function!",
489                           file=sys.stderr)
490                     parser.success = False
491         else:
492             if name != 'NIL' :
493                 print("ERROR: Not Declared!", file=sys.stderr)
494                 parser.success = False
495     if parser.success:
496         flag = False
497         if name == 'NIL':
498             flag = True
499         if name == parser.currentfunc:
500             address = parser.namespace[name+'1']['address']
501         else:
502             address = parser.namespace[name]['address']
503         if flag:
504             p[0] = '\tpushi 99999\n'
505         else:
506             p[0] = f'\tpushl {address}\n'
507 def p_factor_prio(p):
508     'factor : LPAREN cond_expression RPAREN'
509     if parser.success:
510         p[0] = p[2]
511 def p_factor_not(p):
512     'factor : NOT expression'
513     if parser.success:
514         p[0] = p[2] + '\tnot\n'
515 def p_factor_sym(p):
516     'factor : SUB expression'
517     if parser.success:
518         p[0] = f"\tpushi 0\n{p[2]}\tsub\n"
519 def p_factor_func(p):
520     'factor : call_function'
521     if parser.success:
522         p[0] = p[1]
523 def p_factor_arr(p):
524     'factor : indarr'
525     if parser.success:
526         p[0] = p[1]
527 def p_factor_mat(p):
528     'factor : indmat'
529     if parser.success:
530         p[0] = p[1]
531 def p_factor_address(p):
532     'factor : ADDR ID'
533     if parser.success:
534         name = p[2]
```

```
535         if name in parser.namespace:
536             if parser.namespace[name]['class'] == 'var':
537                 if parser.namespace[name]['scope'] != parser.currentfunc:
538                     print("ERROR: Not Declared!",
539                           file=sys.stderr)
540                     parser.success = False
541             else:
542                 print("ERROR: Not a variable!",
543                       file=sys.stderr)
544                 parser.success = False
545     if parser.success:
546         address = parser.namespace[name]['address']
547         p[0] = f'\tpushfp\n\tpushi {address}\n\tpadd\n'
548 def p_factor_addrarr(p):
549     'factor : ADDR ID ARRINDL expression ARRINDR'
550     if parser.success:
551         name = p[2]
552         const = p[4]
553         if name not in parser.namespace:
554             print(f"ERROR: Indexing without declaration.",
555                   file=sys.stderr)
556             parser.success = False
557     if parser.success:
558         if (parser.namespace[name]['class'] != 'var'
559             or parser.namespace[name]['type'] != 'REF INT'):
560             print(f"ERROR: Malformed indexing.",
561                   file=sys.stderr)
562             parser.success = False
563         else:
564             index = parser.namespace[name]['address']
565             p[0] = f'\tpushl {index}\n{const}\tpadd\n'
566 def p_facto_addrmat(p):
567     'factor : ADDR ID ARRINDL expression ARRCONT expression ARRINDR'
568     if parser.success:
569         name = p[2]
570         if name not in parser.namespace:
571             print("ERROR: Indexing without declaration.",
572                   file=sys.stderr)
573             parser.success = False
574     if parser.success:
575         if (parser.namespace[name]['class'] != 'var'
576             or parser.namespace[name]['type'] != 'REF REF INT'):
577             print("ERROR: Malformed indexing.")
578             parser.success = False
579         else:
580             index = parser.namespace[name]['address']
581             p[0] = f'\tpushl {index}\n{p[4]}\tpadd\n\t{p[6]}\tpadd\n'
582 def p_factor_dereference(p):
583     'factor : DEREF ID'
584     if parser.success:
585         name = p[2]
586         if name in parser.namespace:
587             if parser.namespace[name]['class'] == 'var':
588                 if parser.namespace[name]['type'] != 'REF INT':
```

```
589                     print("ERROR: Derefencing value!",
590                             file=sys.stderr)
591                     parser.success = False
592                 if parser.namespace[name]['scope'] != parser.currentfunc:
593                     print("ERROR: Not Declared!",
594                             file=sys.stderr)
595                     parser.success = False
596             else:
597                 print("ERROR: Not a variable!",
598                         file=sys.stderr)
599                 parser.success = False
600     if parser.success:
601         address = parser.namespace[name]['address']
602         p[0] = f'\tpushl {address}\n\tload 0\n'
603 
604 def p_ad_op_sum(p):
605     'ad_op : SUM'
606     if parser.success:
607         p[0] = '\tadd\n'
608 def p_ad_op_sub(p):
609     'ad_op : SUB'
610     if parser.success:
611         p[0] = '\tsub\n'
612 
613 def p_mult_op_1(p):
614     'mult_op : MULT'
615     if parser.success:
616         p[0] = '\tmul\n'
617 def p_mult_op_2(p):
618     'mult_op : DIV'
619     if parser.success:
620         p[0] = '\tdiv\n'
621 def p_mult_op_3(p):
622     'mult_op : MODULO'
623     if parser.success:
624         p[0] = '\tmod\n'
625 
626 def p_conditionals(p):
627     'conditionals : conditional code_logic'
628     if parser.success:
629         p[0] = p[1] + p[2]
630 
631 def p_conditional_while(p):
632     'conditional : WHILE cond_expression cond_code'
633     if parser.success:
634         loop_label = 'L' + str(parser.labelcounter)
635         parser.labelcounter += 1
636         end_label = 'L' + str(parser.labelcounter)
637         parser.labelcounter += 1
638         p[0] = f'{loop_label}:\n{p[2]}\tjz {end_label}\n{p[3]}\tjump {loop_label}\n{
                end_label}:\n'
639 
640 def p_conditional_do_while(p):
641     'conditional : DO cond_code WHILE cond_expression'
```

```
642     if parser.success:
643         loop_label = 'L' + str(parser.labelcounter)
644         parser.labelcounter += 1
645         p[0] = f'{loop_label}:\n{p[2]}\t{p[4]}\tjz {loop_label}\n'
646 
647 def p_conditional_until(p):
648     'conditional : UNTIL cond_expression cond_code'
649     if parser.success:
650         loop_label = 'L' + str(parser.labelcounter)
651         parser.labelcounter += 1
652         end_label = 'L' + str(parser.labelcounter)
653         parser.labelcounter += 1
654         p[0] = f'{loop_label}:\n{p[2]}\tnot\n\tjz {end_label}\n{p[3]}\tjump {
            loop_label}\n{end_label}:\n'
655 
656 def p_conditional_do_until(p):
657     'conditional : DO cond_code UNTIL cond_expression'
658     if parser.success:
659         loop_label = 'L' + str(parser.labelcounter)
660         parser.labelcounter += 1
661         p[0] = f'{loop_label}:\n{p[2]}\t{p[4]}\tnot\n\tjz {loop_label}\n'
662 
663 def p_conditional_if(p):
664     'conditional : IF cond_expression cond_code'
665     if parser.success:
666         cond_label = 'L' + str(parser.labelcounter)
667         parser.labelcounter += 1
668         p[0] = f'{p[2]}\tjz {cond_label}\n{p[3]}{cond_label}:\n'
669 
670 def p_conditional_if_else(p):
671     'conditional : IF cond_expression cond_code ELSE cond_code'
672     if parser.success:
673         else_label = 'L' + str(parser.labelcounter)
674         parser.labelcounter += 1
675         end_label = 'L' + str(parser.labelcounter)
676         parser.labelcounter += 1
677         p[0] = f'{p[2]}\tjz {else_label}\n{p[3]}\tjump {end_label}\n'
678         p[0]+= f'{else_label}:\n{p[5]}{end_label}:\n'
679 def p_cond_expr(p):
680     'cond_expression : expression'
681     if parser.success:
682         p[0] = p[1]
683 def p_cond_expr_1(p):
684     'cond_expression : cond_expression bool_op expression'
685     if parser.success:
686         p[0] = p[1] + p[3] + p[2]
687 def p_bool_op_eq(p):
688     'bool_op : EQ'
689     if parser.success:
690         p[0] = '\tequal\n'
691 def p_bool_op_dif(p):
692     'bool_op : DIF'
693     if parser.success:
694         p[0] = '\tequal\n\tnot\n'
```

```
695 def p_bool_op_leq(p):
696     'bool_op : LEQ'
697     if parser.success:
698         p[0] = '\tinfeq\n'
699 def p_bool_op_geq(p):
700     'bool_op : GEQ'
701     if parser.success:
702         p[0] = '\tsupeq\n'
703 def p_bool_op_les(p):
704     'bool_op : LESSER'
705     if parser.success:
706         p[0] = '\tinf\n'
707 def p_bool_op_gre(p):
708     'bool_op : GREATER'
709     if parser.success:
710         p[0] = '\tsup\n'
711 def p_bool_op_and(p):
712     'bool_op : CONDAND'
713     if parser.success:
714         p[0] = '\tand\n'
715 def p_bool_op_or(p):
716     'bool_op : CONDOR'
717     if parser.success:
718         p[0] = '\tor\n'
719 def p_cond_code(p):
720     'cond_code : BLOCK_START code_logic BLOCK_END'
721     if parser.success:
722         p[0] = p[2]
723 def p_call_functions(p):
724     'call_functions : call_function COMP code_logic'
725     if parser.success:
726         p[0] = p[1] + p[3]
727 def p_call_function(p):
728     'call_function : ID args_lst'
729     if parser.success:
730         name = p[1]
731         args = p[2]
732         if name not in parser.namespace:
733             print("ERROR: Function not declared before use",
734                   file=sys.stderr)
735             parser.success = False
736         if parser.success:
737             if parser.namespace[name]['class'] != 'funct':
738                 print("ERROR: not a function",
739                       file=sys.stderr)
740                 parser.success = False
741             else:
742                 if len(parser.namespace[name]['arguments']) != len(args):
743                     print("ERROR: incorrect length of arguments",
744                           file=sys.stderr)
745                     parser.success = False
746     if parser.success:
747         if parser.namespace[name]['return'] == 'VOID':
748             res = ''
```

```
749             for arg in args[::-1]:
750                 res += f'{arg}'
751         else:
752             res = '\tpushi 0\n'
753             for arg in args[::-1]:
754                 res += f'{arg}'
755         p[0] = res + f'\tpusha {name}\n\tcall\n\tpop {len(args)}\n'
756 def p_call_read(p):
757     'call_function : READ LPAREN RPAREN'
758     if parser.success:
759         p[0] = '\tread\n'
760 def p_call_writes(p):
761     'call_function : WRITES LPAREN STRING RPAREN'
762     if parser.success:
763         p[0] = f'\tpushs {p[3]}\n\twrites\n'
764 def p_call_writesid(p):
765     'call_function : WRITES LPAREN ID RPAREN'
766     if parser.success:
767         name = p[3]
768         if name in parser.namespace:
769             if parser.namespace[name]['class'] == 'var':
770                 if parser.namespace[name]['scope'] != parser.currentfunc:
771                     print("ERROR: Not Declared!",
772                           file=sys.stderr)
773                     parser.success = False
774                 elif parser.namespace[name]['type'] != 'STR':
775                     print("ERROR: Not a string variable",
776                           file=sys.stderr)
777                     parser.success = False
778             else:
779                 print("ERROR: Not a valid variable!",
780                       file=sys.stderr)
781                 parser.success = False
782         else:
783             print("ERROR: Not declared!",
784                   file=sys.stderr)
785             parser.success = False
786     if parser.success:
787         address = parser.namespace[name]['address']
788         p[0] = f'\tpushl {address}\n\twrites\n'
789 def p_call_writeread(p):
790     'call_function : WRITES LPAREN READ LPAREN RPAREN RPAREN'
791     if parser.success:
792         p[0] = '\tread\n\twrites\n'
793 def p_call_writeint(p):
794     'call_function : WRITEI LPAREN expression RPAREN'
795     if parser.success:
796         p[0] = f'{p[3]}\twritei\n'
797 def p_call_atoi(p):
798     'call_function : ATOI LPAREN STRING RPAREN'
799     if parser.success:
800         p[0] = f'\tpushs {p[3]}\n\tatoi\n'
801 def p_call_atoi_1(p):
802     'call_function : ATOI LPAREN ID RPAREN'
```

```
803     if parser.success:
804         name = p[3]
805         if name in parser.namespace:
806             if parser.namespace[name]['class'] == 'var':
807                 if parser.namespace[name]['scope'] != parser.currentfunc:
808                     print("ERROR: Not Declared!",
809                           file=sys.stderr)
810                     parser.success = False
811                 elif parser.namespace[name]['type'] != 'STR':
812                     print("ERROR: Not a string variable",
813                           file=sys.stderr)
814                     parser.success = False
815             else:
816                 print("ERROR: Not a valid variable!",
817                       file=sys.stderr)
818                 parser.success = False
819         else:
820             print("ERROR: Not declared!",
821                   file=sys.stderr)
822             parser.success = False
823     if parser.success:
824         address = parser.namespace[name]['address']
825         p[0] = f'\tpushl {address}\n\twrites\n'
826 def p_call_atoi_2(p):
827     'call_function : ATOI LPAREN READ LPAREN RPAREN RPAREN'
828     if parser.success:
829         p[0] = '\tread\n\tatoi\n'
830
831 def p_args_lst(p):
832     'args_lst : LPAREN RPAREN'
833     if parser.success:
834         p[0] = []
835 def p_args_lst_1(p):
836     'args_lst : LPAREN expression args RPAREN'
837     if parser.success:
838         p[0] = [p[2]] + p[3]
839 def p_args(p):
840     'args : '
841     if parser.success:
842         p[0] = []
843 def p_args_1(p):
844     'args : ARRCONT expression args'
845     if parser.success:
846         p[0] = [p[2]] + p[3]
847
848 def p_func_type_1(p):
849     'func_type : VOID'
850     if parser.success:
851         p[0] = p[1]
852
853 def p_func_type_2(p):
854     'func_type : data_type'
855     if parser.success:
856         p[0] = p[1]
```

```
857
858 def p_data_type(p):
859     'data_type : STR'
860     if parser.success:
861         p[0] = p[1]
862 def p_data_type_1(p):
863     'data_type : INT'
864     if parser.success:
865         p[0] = p[1]
866 def p_data_type_2(p):
867     'data_type : pointer data_type'
868     if parser.success:
869         p[0] = p[1] + ' ' + p[2]
870
871 def p_pointer_1(p):
872     'pointer : REF'
873     if parser.success:
874         p[0] = p[1]
875 def p_pointer_2(p):
876     'pointer : REF REF'
877     if parser.success:
878         p[0] = p[1] + ' ' + p[2]
879
880
881 def p_error(p):
882     parser.success = False
883     print(f'ERROR: Could not parse this file.\n{p.lineno}\n{p}',
884             file=sys.stderr)
885 def main():
886     parser.namespace = {
887         'READ' : {
888             'class': 'funct',
889             'arguments':[],
890             'return':'STR'
891             },
892         'WRITEI':{
893             'class':'funct',
894             'arguments':['INT i'],
895             'return':'VOID'
896             },
897         'WRITES':{
898             'class':'funct',
899             'arguments':['STR str'],
900             'return':'VOID'
901             },
902         'ATOI':{
903             'class':'funct',
904             'arguments':['STR str'],
905             'return':'INT'
906             },
907         'INT':{'class':'data'},
908         'STR':{'class':'data'},
909         'IF':{'class':'reserved'},
910         'ELSE':{'class':'reserved'},
```

```
911         'WHILE':{'class':'reserved'},
912         'RETURN':{'class':'reserved'},
913         'UNTIL':{'class':'reserved'},
914         'DO':{'class':'reserved'}
915     }
916     parser.labelcounter = 1
917     parser.currentfunc  = ''
918     parser.varnum       = 0
919     parser.argnum       = 0
920     parser.result       = ''
921     parser.success      = True
922     flag_err            = False
923     argc                = len(sys.argv)
924     flag_name = False
925     if argc >= 2:
926         name = re.search(r'([A-Za-z\_0-9]+)\.nqc', sys.argv[1])
927         if not name:
928             print("ERROR: not a nqc file",
929                   file=sys.stderr)
930             flag_err = True
931     else:
932         print("ERROR: Not enough arguments",
933               file=sys.stderr)
934         flag_err = True
935 
936     if not flag_err and argc > 3:
937         if sys.argv[2] == '-o':
938             if argc >= 4:
939                 new_name = re.match(r'(.*\.vm)',sys.argv[3])
940                 new_name = new_name.group(1)
941                 flag_name = True
942             else:
943                 print("ERROR: Missing new name",
944                       file=sys.stderr)
945                 flag_err = True
946 
947     if not flag_err:
948         with open(sys.argv[1],'r',encoding='UTF-8') as f:
949             cont = f.read()
950         parser.parse(cont)
951         res = str(parser.result)
952         if parser.success:
953             if flag_name:
954                 with open(new_name, 'w+', encoding='UTF-8') as nf:
955                     nf.write(res)
956             else:
957                 print(res)
958             print("Code Generated",file=sys.stderr)
959         else:
960             print("Error generating code",file=sys.stderr)
961     return flag_err
962 
963 parser = yacc.yacc(debug=0)
964 sys.exit(main())
```

§2.2 Grammar

Listing 2.3: NQC Language's Formal Grammar

```
<program> ::= <functions>
<functions> ::=
                | <function> <function>
<function> ::= <function_header> <function_code_outline>
<function_header> ::= <func_type> ID <argument_list_head>
<argument_list_head> ::= LPAREN RPAREN
                         | LPAREN <arg_head> <args_head> RPAREN
<arg_head> ::= <data_type> ID
<args_head> ::=
                | ARRCONT <arg_head> <args_head>
<function_code_outline> ::= BLOCK_START <function_code> BLOCK_END

<function_code> ::=
                    | <declarations> <code_logic>
<declarations> ::=
                   | <declaration> <declarations>
<declaration> ::= <data_type> ID COMP
                  | <data_type> ID ARRINDL NUMBER ARRINDR COMP
                  | <data_type> ID ARRINDL NUMBER ARRCONT NUMBER ARRINDR COMP
<code_logic> ::=
                  | <atributions>
                  | <conditionals>
                  | <function_calls>
<atributions> ::= <atribution> <code_logic>

<atribution> ::= ID ATRIB STRING COMP
                 | ID ATRIB <expression> COMP
                 | DEREF ID ATRIB <expression> COMP
                 | ID ARRINDL <expression> ARRINDR ATRIB <expression> COMP
                 | ID ARRINDL <expression> ARRCONT <expression> ARRINDR ATRIB <
                   expression> COMP
<indarr> ::= ID ARRINDL <expression> ARRINDR
<indmat> ::= ID ARRINDL <expression> ARRCONT <expression> ARRINDR
<expression> ::= <term>
                 | <expression> <ad_op> <term>
<term> ::= <factor>
           | <term> <mult_op> <factor>
<factor> ::= NUMBER
             | ID
             | LPAREN <cond_expression> RPAREN
             | NOT <expression>
             | SUB <expression>
             | <call_function>
             | <indarr>
             | <indmat>
             | ADDR ID
             | ADDR ID ARRINDL expression ARRCONT expression ARRINDR
             | DEREF ID
<ad_op> ::= SUM
            | SUB
```

```
<mult_op> ::= MULT
            | DIV
            | MODULO
<conditionals> ::= <conditional> <code_logic>
<conditional> ::= WHILE <cond_expression> <cond_code>
                | DO <cond_code> WHILE <cond_expression>
                | UNTIL <cond_expression> <cond_code>
                | DO <cond_code> UNTIL <cond_expression>
                | IF <cond_expression> <cond_code>
                | IF <cond_expression> <cond_code> ELSE <cond_code>
<cond_expression> ::= <expression>
                    | <cond_expression> <bool_op> <expression>
<bool_op> ::= EQ | DIF | LEQ | GEQ | LESSER | GREATER
            | CONDAND | CONDOR
<cond_code> ::= BLOCK_START code_logic BLOCK_END

<call_functions> ::= <call_function> COMP <code_logic>
<call_function> ::= ID <args_lst>
                  | READ LPAREN RPAREN
                  | WRITES LPAREN STRING RPAREN | WRITES LPAREN ID RPAREN
                  | WRITES LPAREN READ LPAREN RPAREN RPAREN
                  | WRITEI LPAREN <expression> RPAREN
                  | ATOI LPAREN STRING RPAREN
                  | ATOI LPAREN ID RPAREN
                  | ATOI LPAREN READ LPAREN RPAREN RPAREN
<args_lst> ::= LPAREN RPAREN
             | LPAREN <expression> <args> RPAREN
<args>     ::=
             | ARRCONT <expression> <args>

<func_type> ::= VOID
              | <data_type>
<data_type> ::= STR
              | INT
              | <pointer> <data_type>
<pointer> ::= REF
            | REF REF
```

§2.3 Examples

Listing 2.4: Output of test used in the Analysis section

```
calling: nop
         start
         nop
         pushi 0
         pusha MAIN
         call
         nop
         dup 1
         not
         jz L0
         nop
         pop 1
         stop
L0:
         pushs "Exited with code "
         writes
         writei
         pushs "\n"
         writes
         stop
SWAPF:
         nop
         pushl -1
         pushl -1
         load 0
         pushl -2
         load 0
         mul
         store 0
         pushl -2
         pushl -1
         load 0
         pushl -2
         load 0
         div
         store 0
         pushl -1
         pushl -1
         load 0
         pushl -2
         load 0
         div
         store 0
         return
         nop
A:
         nop
         pushl -1
         pushi 0
         equal
```

```
51          jz L3
52          pushl -2
53          pushi 1
54          add
55          storel -3
56          jump L4
57  L3:
58          pushl -2
59          pushi 0
60          equal
61          jz L1
62          pushi 0
63          pushi 1
64          pushl -1
65          pushi 1
66          sub
67          pusha A
68          call
69          pop 2
70          storel -3
71          jump L2
72  L1:
73          pushi 0
74          pushi 0
75          pushl -2
76          pushi 1
77          sub
78          pushl -1
79          pusha A
80          call
81          pop 2
82          pushl -1
83          pushi 1
84          sub
85          pusha A
86          call
87          pop 2
88          storel -3
89  L2:
90  L4:
91          return
92          nop
93  BS:
94          nop
95          pushi 0
96          pushi 0
97          pushi 1
98          storel 1
99  L8:
100         pushl 1
101         not
102         not
103         jz L9
104         pushi 0
```

```
105         storel 1
106 L6:
107         pushl 0
108         pushl -2
109         pushi 1
110         sub
111         inf
112         jz L7
113         pushl -1
114         pushl 0
115         loadn
116         pushl -1
117         pushl 0
118         pushi 1
119         add
120         loadn
121         sup
122         jz L5
123         pushl -1
124         pushl 0
125         pushi 1
126         add
127         padd
128         pushl -1
129         pushl 0
130         padd
131         pusha SWAPF
132         call
133         pop 2
134         pushi 1
135         storel 1
136 L5:
137         pushl 0
138         pushi 1
139         add
140         storel 0
141         jump L6
142 L7:
143         jump L8
144 L9:
145         pop 2
146         return
147         nop
148 F:
149         nop
150         pushi 0
151         pushi 1
152         storel -2
153 L10:
154         pushl -1
155         pushl 0
156         sub
157         pushi 0
158         sup
```

```
159         jz L11
160         pushl 0
161         pushi 1
162         add
163         storel 0
164         pushl -2
165         pushl 0
166         mul
167         storel -2
168         jump L10
169 L11:
170         pop 1
171         return
172         nop
173 MAIN:
174         nop
175         pushi 0
176         pushfp
177         pushi 2
178         padd
179         pushn 2
180         pushl 1
181         pushi 0
182         pushi 10
183         storen
184         pushl 1
185         pushi 1
186         pushi 0
187         pushi 25
188         sub
189         storen
190         pushi 2
191         pushl 1
192         pusha BS
193         call
194         pop 2
195         pushl 1
196         pushi 0
197         loadn
198         writei
199         pushs "\n"
200         writes
201         pushl 1
202         pushi 1
203         loadn
204         writei
205         pushs "\n"
206         writes
207         pushi 0
208         pushi 1
209         pushi 1
210         pusha A
211         call
212         pop 2
```

```
213         storel 0
214         pushl 0
215         writei
216         pushs "\n"
217         writes
218         pushi 0
219         pushi 2
220         pusha F
221         call
222         pop 1
223         storel 0
224         pushl 0
225         writei
226         pushs "\n"
227         writes
228         pushi 0
229         storel -1
230         pop 4
231         return
232         nop
```

Listing 2.5: General example

```
VOID
SWAPF(REF INT PX, REF INT PY)
BEGIN
    DEREF PX := DEREF PX * DEREF PY;
    DEREF PY := DEREF PX / DEREF PY;
    DEREF PX := DEREF PX / DEREF PY;
END

INT
A(INT M, INT N)
BEGIN
    IF (M = 0)
    BEGIN A := N+1; END
    ELSE
    BEGIN
        IF (N = 0)
        BEGIN
            A := A((M-1),1);
        END
        ELSE
        BEGIN
            A := A((M-1), A(M,N-1));
        END
    END
END

VOID
BS(REF INT AR, INT N) /* BUBBLE SORT */
BEGIN
    INT I;
    INT FLAG;
    FLAG:=1;
    UNTIL (!FLAG)
    BEGIN
        FLAG:=0;
        WHILE (I < (N-1))
        BEGIN
            IF (AR[I] > AR[I+1]) BEGIN SWAPF(&AR[I],&AR[I+1]); FLAG:=1; END
            I:=I+1;
        END
    END
END

INT
F(INT N) /* FACTORIAL FUNCTION */
BEGIN
    INT I;
    F := 1;
    WHILE ((N-I) > 0)
    BEGIN
        I:=I+1;
        F:=F*I;
    END
```

```
54 END
55
56 INT
57 MAIN()
58 BEGIN
59     INT RES;
60     INT ARR[2];
61     ARR[0]:=10;
62     ARR[1]:=-25;
63     BS(ARR,2);
64     WRITEI(ARR[0]); WRITES("\n");
65     WRITEI(ARR[1]); WRITES("\n");
66     RES:=A(1,1);
67     WRITEI(RES); WRITES("\n");
68     RES:=F(2);
69     WRITEI(RES); WRITES("\n");
70     MAIN:=0;
71 END
```

Listing 2.6: Matrix example

```
1  INT
2  MAIN()
3  BEGIN
4      INT ARR[3,3];
5      INT I; INT J;
6      WHILE (I<3)
7      BEGIN
8          J:=0;
9          WHILE (J<3)
10         BEGIN
11             ARR[I,J]:=I-J;
12             J:=J+1;
13         END
14         I:=I+1;
15     END
16     MAIN:=0;
17 END
```

Listing 2.7: VM code for matrix example

```
calling: nop
         start
         nop
         pushi 0
         pusha MAIN
         call
         nop
         dup 1
         not
         jz L0
         nop
         pop 1
         stop
L0:
         pushs "Exited with code "
         writes
         writei
         pushs "\n"
         writes
         stop
MAIN:
         nop
         pushfp
         pushi 3
         padd
         pushfp
         pushi 6
         padd
         pushfp
         pushi 9
         padd
         pushn 9
         pushi 0
         pushi 0
L3:
         pushl 13
         pushi 3
         inf
         jz L4
         pushi 0
         storel 14
L1:
         pushl 14
         pushi 3
         inf
         jz L2
         pushl 0
         pushl 13
         pushi 3
         mul
         padd
         pushl 14
         pushl 13
```

```
54          pushl 14
55          sub
56          storen
57          pushl 14
58          pushi 1
59          add
60          storel 14
61          jump L1
62  L2:
63          pushl 13
64          pushi 1
65          add
66          storel 13
67          jump L3
68  L4:
69          pushi 0
70          storel -1
71          pop 15
72          return
73          nop
```

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] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

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

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

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

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

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

[15] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, jul 1974.

[16] 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.