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

January 15, 2023

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 [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
    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;
   INT FLAG;
```

[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

```
   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;
   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'\>\>'
```

```
########### BOOLEAN ##################
t_GEQ = r'\>\=';t_LEQ = r'\<\='
t_DIF = r'\!\=';t_EQ = r'\='
t_LESSER  = r'\<';t_GREATER = r'\>'
t_CONDAND = r'\&\&';t_CONDOR = r'\|\|'
t_NOT = r'\!'
######### SYNTAX RELATIVE SYMBOLS ##########
t_ATRIB      = r'\:\=';t_COMP     = r'\x3B' # ;
t_ARRCONT    = r'\x2C' # ,
t_ARRINDL    = r'\x5B' # [ Indexing arrays translates to load or store
t_ARRINDR    = r'\x5D' # ] Indexing arrays translates to load or store
t_ADDR       = r'\&'


t_LPAREN     = r'\x28' # (
t_RPAREN     = r'\x29' # )
#t_BLOCK_START = r'BEGIN\n';t_BLOCK_END = r'END\n'
#t_BLOCK_START = r'\{';t_BLOCK_END = r'\}'

def t_STRING(t):
    r'\".*\"';t.type = reserved.get(t.value, 'STRING'); return t
def t_COMMENT(t):
    r'\/\*(.|\n)*?\*\/'; pass
    # Ignores everything between /* */

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value); return t

def t_BLOCK_START(t):
    r'BEGIN';return t
def t_BLOCK_END(t):
    r'END';return t
def t_ID(t):
    r'[A-Za-z]+';t.type = reserved.get(t.value, 'ID'); return t

def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

t_ignore = '\x20\t' # Spaces and Tabs

def t_error(t):
    print(f"Illegal character {t.value[0]}")
    # t.lexer.skip(1)

lexer = lex.lex()

if __name__ == '__main__':
    with open(sys.argv[1], 'r', encoding='UTF-8') as file:
        cont = file.read()

    lexer.input(cont)
    token = lexer.token()
```

```
    while token:
        print(token)
        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',
```

```
                                    'arguments':[], 'return':'INT'}
                parser.namespace['MAIN1'] = {'class':'var',
                                        'address' : '-1',
                                        'type'    : 'INT',
                                        'size'    : '0',
                                        'scope'   : 'MAIN'}
        else:
            if name in parser.namespace:
                print("ERROR: Name already used",
                       file=sys.stderr)
                parser.success = False
            if parser.success:
                try:
                    parser.namespace[name] = {'class':'funct',
                                  'arguments':args.split(','),'return':r_type}
                    for elem in args.split(','):
                        stuff = elem.split(' ')
                        data = ' '.join(stuff[:-1])
                        var_name = stuff[-1]
                        parser.argnum -= 1
                        parser.namespace.update({var_name : {
                            'class'   : 'var',
                            'address' : str(parser.argnum),
                            'type'    : data,
                            'size'     : '0',
                            'scope'   : parser.currentfunc,
                        }})
                except AttributeError:
                    parser.namespace[name] = {'class':'funct',
                                  'arguments':[],'return':r_type}
                if r_type != 'VOID':
                    parser.namespace[name+'1'] = {'class' : 'var',
                                'address' : parser.argnum-1,
                                'type'    : r_type,
                                'size'   : '0',
                                'scope'  : parser.currentfunc
                    }
        if parser.success:
            parser.argnum = 0
            parser.varnum = 0
            p[0] = name + ':\n\tnop\n'

def p_argument_list_head_1(p):
    'argument_list_head : LPAREN RPAREN '
    if parser.success:
        p[0] = []
def p_argument_list_head_2(p):
    'argument_list_head : LPAREN arg_head args_head RPAREN'
    if parser.success:
        p[0] = p[2] + p[3]

def p_arg_head(p):
    'arg_head : data_type ID'
    if parser.success:
```

```
        name = p[2]
        data = p[1]
        if name in parser.namespace:
            if parser.namespace[name]['class'] != 'var':
                parser.success = False
        if parser.success:
            p[0] = data + ' ' + name
def p_args_head_1(p):
    'args_head : '
    if parser.success:
        p[0] = ''
def p_args_head_2(p):
    'args_head : ARRCONT arg_head args_head'
    if parser.success:
        p[0] = p[1] + p[2] + p[3]


def p_function_code_outline(p):
    'function_code_outline : BLOCK_START function_code BLOCK_END'
    if parser.success:
        p[0] = p[2]

def p_function_code_1(p):
    'function_code : '
    if parser.success:
        p[0] = ''
def p_function_code_2(p):
    'function_code : declarations code_logic'
    if parser.success:
        if parser.varnum:
            p[0] = p[1] + p[2] + f'\tpop {parser.varnum}\n\treturn\n\tnop\n'
        else:
            p[0] = p[1] + p[2] + '\treturn\n\tnop\n'


def p_declarations_1(p):
    'declarations : '
    if parser.success:
        p[0] = ''
def p_declarations_2(p):
    'declarations : declaration declarations'
    if parser.success:
        p[0] = p[1] + p[2]


def p_declaration_1(p):
    'declaration : data_type ID COMP'
    if parser.success:
        name = p[2]
        data = p[1]
        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
        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:
```

```
        row = p[4]
        col = p[6]
        total_size = int(row) * int(col)
        data = p[1]
        name = p[2]
        res = ''
        if data != 'INT':
            print("ERROR: Array must be of Integers",
                  file=sys.stderr)
            parser.success = False
        else:
            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 += row+total_size
        parser.namespace[name] = {
                'class' : 'var',
                'address' : str(ind),
                'type' : 'REF REF ' + data,
                'size' : str(total_size),
                'scope' : parser.currentfunc
                }
        for i in range(0,int(row)):
            res += f'\tpushfp\n\tpushi {ind+i}\n\tpadd\n\tpushi {col}\n\tpushi {i}\n\tadd\n\tpadd\n'
        p[0] = res + f'\tpushn {total_size}\n'


def p_code_logic(p):
    'code_logic : '
    if parser.success:
        p[0] = ''
def p_code_logic_atr(p):
    'code_logic : atributions'
    if parser.success:
        p[0] = p[1]
def p_code_logic_cond(p):
    'code_logic : conditionals'
    if parser.success:
        p[0] = p[1]
def p_code_logic_func(p):
    'code_logic : call_functions'
    if parser.success:
        p[0] = p[1]

```

```
def p_atributions(p):
    'atributions : atribution code_logic'
    if parser.success:
        p[0] = p[1] + p[2]

def p_atribution_str(p):
    'atribution : ID ATRIB STRING COMP'
    if parser.success:
        name = p[1]
        string = p[3]
        if name in parser.namespace:
            if parser.namespace[name]['class'] == 'var':
                if parser.namespace[name]['scope'] != parser.currentfunc:
                    print("ERROR: Not declared!",
                            file=sys.stderr)
                    parser.success = False
                elif parser.namespace[name]['type'] != 'STR':
                    print("ERROR: Not a string",
                            file=sys.stderr)
            else:
                if name != parser.currentfunc:
                    print("ERROR: Not a variable!",
                            file=sys.stderr)
                    parser.success = False
                else:
                    if parser.namespace[name]['return'] != 'STR':
                        print("ERROR: Wrong type",
                                file=sys.stderr)
                        parser.success = False
                    if parser.namespace[name]['return'] == 'VOID':
                        print("ERROR: Assigning value to void function",
                                file=sys.stderr)
                        parser.success = False
        else:
            print("ERROR: Not declared!",
                    file=sys.stderr)
            parser.success = False
        if parser.success:
            if name == parser.currentfunc:
                address = parser.namespace[name+'1']['address']
            else: address = parser.namespace[name]['address']
            p[0] = f'\tpushs {p[3]}\n\tstorel {address}\n'
def p_atribution_1(p):
    'atribution : ID ATRIB expression COMP'
    if parser.success:
        name = p[1]
        if name in parser.namespace:
            if parser.namespace[name]['class'] == 'var':
                if parser.namespace[name]['scope'] != parser.currentfunc:
                    print("ERROR: Not Declared!",
                           file=sys.stderr)
                    parser.success = False
                elif parser.namespace[name]['type'] == 'STR':
```

```
                    print("ERROR: A String cannot be an expression",
                            file=sys.stderr)
                    parser.success=False
            else:
                if name != parser.currentfunc:
                    print("ERROR: Not a variable!",
                           file=sys.stderr)
                    parser.success = False
                else:
                    if parser.namespace[name]['return'] == 'STR':
                        print("ERROR: Mismatch type",
                                file=sys.stderr)
                        parser.success = False
                    elif parser.namespace[name]['return'] == 'VOID':
                        print("ERROR: Assigning value to void function",
                               file=sys.stderr)
                        parser.success = False
        else:
            print("ERROR: Not declared!",
                   file=sys.stderr)
            parser.success = False
    if parser.success:
        if name == parser.currentfunc:
            address = parser.namespace[name+'1']['address']
        else: address = parser.namespace[name]['address']
        p[0] = f'{p[3]}\tstorel {address}\n'
def p_atribution_deref(p):
    'atribution : DEREF ID ATRIB expression COMP'
    if parser.success:
        name = p[2]
        if name in parser.namespace:
            if parser.namespace[name]['class'] == 'var':
                if parser.namespace[name]['type'] != 'REF INT':
                    print("ERROR: Dereferencing value")
                    parser.success = False
                if parser.namespace[name]['scope'] != parser.currentfunc:
                    print(f"ERROR: {p[1]} Not Declared!")
                    parser.success = False
            else:
                print(f"ERROR: {p[1]} Not a variable!")
                parser.success = False
        else:
            parser.success = False
        if parser.success:
            address = parser.namespace[name]['address']
            p[0] = f'\tpushl {address}\n{p[4]}\tstore 0\n'

def p_atribution_3(p):
    'atribution : ID ARRINDL expression ARRINDR ATRIB expression COMP'
    if parser.success:
        name = p[1]
        ind = p[3]
        atrib_expr = p[6]
        if name not in parser.namespace:
```

```python
            print("ERROR: Atribution without declaration.",
                    file=sys.stderr)
            parser.success = False
    if parser.success:
        if (parser.namespace[name]['class'] != 'var'
                or parser.namespace[name]['type'] != 'REF INT'):
            print("ERROR: Malformed indexing.",
                    file=sys.stderr)
            parser.success = False
        else:
            index = parser.namespace[name]['address']
            p[0] = f'\tpushl {index}\n{ind}{atrib_expr}\tstoren\n'
def p_atribution_4(p):
    'atribution : ID ARRINDL expression ARRCONT expression ARRINDR ATRIB expression COMP'
    if parser.success:
        name = p[1]
        row = p[3]
        col = p[5]
        atrib_expr = p[8]
        if name not in parser.namespace:
            print("ERROR: Atribution without declaration",
                    file=sys.stderr)
            parser.success = False
        if parser.success:
            if (parser.namespace[name]['class'] != 'var'
                    or parser.namespace[name]['type'] != 'REF REF INT'):
                print("ERROR: Malformed indexing.", file=sys.stderr)
                parser.success = False
            else:
                index = parser.namespace[name]['address']
                p[0] = f'\tpushl {index}\n{col}\tpadd\n{row}{atrib_expr}\tstoren\n'
def p_indarr_1(p):
    'indarr : ID ARRINDL expression ARRINDR'
    if parser.success:
        name = p[1]
        const = p[3]
        if name not in parser.namespace:
            print(f"ERROR: Indexing without declaration.",
                    file=sys.stderr)
            parser.success = False
    if parser.success:
        if (parser.namespace[name]['class'] != 'var'
            or parser.namespace[name]['type'] != 'REF INT'):
            print(f"ERROR: Malformed indexing.",
                    file=sys.stderr)
            parser.success = False
        else:
            index = parser.namespace[name]['address']
            p[0] = f'\tpushl {index}\n{const}\tloadn\n'
def p_indmat_2(p):
    'indmat : ID ARRINDL expression ARRCONT expression ARRINDR'
    if parser.success:
        name = p[1]
```

```
            if name not in parser.namespace:
                print("ERROR: Indexing without declaration.",
                        file=sys.stderr)
                parser.success = False
        if parser.success:
            if (parser.namespace[name]['class'] != 'var'
                or parser.namespace[name]['type'] != 'REF REF INT'):
                print("ERROR: Malformed indexing.")
                parser.success = False
            else:
                index = parser.namespace[name]['address']
                p[0] = f'\tpushl {index}\n{p[3]}\tpadd\n\t{p[4]}\tloadn\n'


def p_expression_1(p):
    'expression : term'
    if parser.success:
        p[0] = p[1]
def p_expression_2(p):
    'expression : expression ad_op term'
    if parser.success:
        p[0] = p[1] + p[3] + p[2]

def p_term(p):
    'term : factor'
    if parser.success:
        p[0] = p[1]
def p_term_1(p):
    'term : term mult_op factor'
    if parser.success:
        p[0] = p[1] + p[3] + p[2]
def p_factor(p):
    'factor : NUMBER'
    if parser.success:
        p[0] = f'\tpushi {p[1]}\n'
def p_factor_id(p):
    'factor : ID'
    if parser.success:
        name = p[1]
        if name in parser.namespace:
            if parser.namespace[name]['class'] == 'var':
                if parser.namespace[name]['scope'] != parser.currentfunc:
                    print("ERROR: Not Declared!",
                            file=sys.stderr)
                    parser.success = False
            else:
                if (name == parser.currentfunc and
                    parser.namespace[name]['return'] == 'VOID'):
                    print("ERROR: Accessing value of void function!",
                            file=sys.stderr)
                    parser.success = False
        else:
            if name != 'NIL' :
                print("ERROR: Not Declared!", file=sys.stderr)
```

```
                parser.success = False
    if parser.success:
        flag = False
        if name == 'NIL':
            flag = True
        if name == parser.currentfunc:
            address = parser.namespace[name+'1']['address']
        else:
            address = parser.namespace[name]['address']
        if flag:
            p[0] = '\tpushi 99999\n'
        else:
            p[0] = f'\tpushl {address}\n'
def p_factor_prio(p):
    'factor : LPAREN cond_expression RPAREN'
    if parser.success:
        p[0] = p[2]
def p_factor_not(p):
    'factor : NOT expression'
    if parser.success:
        p[0] = p[2] + '\tnot\n'
def p_factor_sym(p):
    'factor : SUB expression'
    if parser.success:
        p[0] = f"\tpushi 0\n{p[2]}\tsub\n"
def p_factor_func(p):
    'factor : call_function'
    if parser.success:
        p[0] = p[1]
def p_factor_arr(p):
    'factor : indarr'
    if parser.success:
        p[0] = p[1]
def p_factor_mat(p):
    'factor : indmat'
    if parser.success:
        p[0] = p[1]
def p_factor_address(p):
    'factor : ADDR ID'
    if parser.success:
        name = p[2]
        if name in parser.namespace:
            if parser.namespace[name]['class'] == 'var':
                if parser.namespace[name]['scope'] != parser.currentfunc:
                    print("ERROR: Not Declared!",
                            file=sys.stderr)
                    parser.success = False
            else:
                print("ERROR: Not a variable!",
                        file=sys.stderr)
                parser.success = False
    if parser.success:
        address = parser.namespace[name]['address']
        p[0] = f'\tpushfp\n\tpushi {address}\n\tpadd\n'
```

```
def p_factor_addrarr(p):
    'factor : ADDR ID ARRINDL expression ARRINDR'
    if parser.success:
        name = p[2]
        const = p[4]
        if name not in parser.namespace:
            print(f"ERROR: Indexing without declaration.",
                  file=sys.stderr)
            parser.success = False
    if parser.success:
        if (parser.namespace[name]['class'] != 'var'
            or parser.namespace[name]['type'] != 'REF INT'):
            print(f"ERROR: Malformed indexing.",
                  file=sys.stderr)
            parser.success = False
        else:
            index = parser.namespace[name]['address']
            p[0] = f'\tpushl {index}\n{const}\tpadd\n'
def p_facto_addrmat(p):
    'factor : ADDR ID ARRINDL expression ARRCONT expression ARRINDR'
    if parser.success:
        name = p[2]
        if name not in parser.namespace:
            print("ERROR: Indexing without declaration.",
                  file=sys.stderr)
            parser.success = False
    if parser.success:
        if (parser.namespace[name]['class'] != 'var'
            or parser.namespace[name]['type'] != 'REF REF INT'):
            print("ERROR: Malformed indexing.")
            parser.success = False
        else:
            index = parser.namespace[name]['address']
            p[0] = f'\tpushl {index}\n{p[4]}\tpadd\n\t{p[6]}\tpadd\n'
def p_factor_dereference(p):
    'factor : DEREF ID'
    if parser.success:
        name = p[2]
        if name in parser.namespace:
            if parser.namespace[name]['class'] == 'var':
                if parser.namespace[name]['type'] != 'REF INT':
                    print("ERROR: Derefencing value!",
                          file=sys.stderr)
                    parser.success = False
                if parser.namespace[name]['scope'] != parser.currentfunc:
                    print("ERROR: Not Declared!",
                          file=sys.stderr)
                    parser.success = False
            else:
                print("ERROR: Not a variable!",
                      file=sys.stderr)
                parser.success = False
    if parser.success:
        address = parser.namespace[name]['address']
```

```
        p[0] = f'\tpushl {address}\n\tload 0\n'

def p_ad_op_sum(p):
    'ad_op : SUM'
    if parser.success:
        p[0] = '\tadd\n'
def p_ad_op_sub(p):
    'ad_op : SUB'
    if parser.success:
        p[0] = '\tsub\n'

def p_mult_op_1(p):
    'mult_op : MULT'
    if parser.success:
        p[0] = '\tmul\n'
def p_mult_op_2(p):
    'mult_op : DIV'
    if parser.success:
        p[0] = '\tdiv\n'
def p_mult_op_3(p):
    'mult_op : MODULO'
    if parser.success:
        p[0] = '\tmod\n'

def p_conditionals(p):
    'conditionals : conditional code_logic'
    if parser.success:
        p[0] = p[1] + p[2]

def p_conditional_while(p):
    'conditional : WHILE cond_expression cond_code'
    if parser.success:
        loop_label = 'L' + str(parser.labelcounter)
        parser.labelcounter += 1
        end_label = 'L' + str(parser.labelcounter)
        parser.labelcounter += 1
        p[0] = f'{loop_label}:\n{p[2]}\tjz {end_label}\n{p[3]}\tjump {loop_label}\n{end_label}:\n'

def p_conditional_do_while(p):
    'conditional : DO cond_code WHILE cond_expression'
    if parser.success:
        loop_label = 'L' + str(parser.labelcounter)
        parser.labelcounter += 1
        p[0] = f'{loop_label}:\n{p[2]}\t{p[4]}\tjz {loop_label}\n'

def p_conditional_until(p):
    'conditional : UNTIL cond_expression cond_code'
    if parser.success:
        loop_label = 'L' + str(parser.labelcounter)
        parser.labelcounter += 1
        end_label = 'L' + str(parser.labelcounter)
        parser.labelcounter += 1
        p[0] = f'{loop_label}:\n{p[2]}\tnot\n\tjz {end_label}\n{p[3]}\tjump {
```

```
        loop_label}\n{end_label}:\n'

def p_conditional_do_until(p):
    'conditional : DO cond_code UNTIL cond_expression'
    if parser.success:
        loop_label = 'L' + str(parser.labelcounter)
        parser.labelcounter += 1
        p[0] = f'{loop_label}:\n{p[2]}\t{p[4]}\tnot\n\tjz {loop_label}\n'

def p_conditional_if(p):
    'conditional : IF cond_expression cond_code'
    if parser.success:
        cond_label = 'L' + str(parser.labelcounter)
        parser.labelcounter += 1
        p[0] = f'{p[2]}\tjz {cond_label}\n{p[3]}{cond_label}:\n'

def p_conditional_if_else(p):
    'conditional : IF cond_expression cond_code ELSE cond_code'
    if parser.success:
        else_label = 'L' + str(parser.labelcounter)
        parser.labelcounter += 1
        end_label = 'L' + str(parser.labelcounter)
        parser.labelcounter += 1
        p[0] = f'{p[2]}\tjz {else_label}\n{p[3]}\tjump {end_label}\n'
        p[0]+= f'{else_label}:\n{p[5]}{end_label}:\n'
def p_cond_expr(p):
    'cond_expression : expression'
    if parser.success:
        p[0] = p[1]
def p_cond_expr_1(p):
    'cond_expression : cond_expression bool_op expression'
    if parser.success:
        p[0] = p[1] + p[3] + p[2]
def p_bool_op_eq(p):
    'bool_op : EQ'
    if parser.success:
        p[0] = '\tequal\n'
def p_bool_op_dif(p):
    'bool_op : DIF'
    if parser.success:
        p[0] = '\tequal\n\tnot\n'
def p_bool_op_leq(p):
    'bool_op : LEQ'
    if parser.success:
        p[0] = '\tinfeq\n'
def p_bool_op_geq(p):
    'bool_op : GEQ'
    if parser.success:
        p[0] = '\tsupeq\n'
def p_bool_op_les(p):
    'bool_op : LESSER'
    if parser.success:
        p[0] = '\tinf\n'
def p_bool_op_gre(p):
```

```
    'bool_op : GREATER'
    if parser.success:
        p[0] = '\tsup\n'
def p_bool_op_and(p):
    'bool_op : CONDAND'
    if parser.success:
        p[0] = '\tand\n'
def p_bool_op_or(p):
    'bool_op : CONDOR'
    if parser.success:
        p[0] = '\tor\n'
def p_cond_code(p):
    'cond_code : BLOCK_START code_logic BLOCK_END'
    if parser.success:
        p[0] = p[2]
def p_call_functions(p):
    'call_functions : call_function COMP code_logic'
    if parser.success:
        p[0] = p[1] + p[3]
def p_call_function(p):
    'call_function : ID args_lst'
    if parser.success:
        name = p[1]
        args = p[2]
        if name not in parser.namespace:
            print("ERROR: Function not declared before use",
                    file=sys.stderr)
            parser.success = False
        if parser.success:
            if parser.namespace[name]['class'] != 'funct':
                print("ERROR: not a function",
                        file=sys.stderr)
                parser.success = False
            else:
                if len(parser.namespace[name]['arguments']) != len(args):
                    print("ERROR: incorrect length of arguments",
                            file=sys.stderr)
                    parser.success = False
    if parser.success:
        if parser.namespace[name]['return'] == 'VOID':
            res = ''
            for arg in args[::-1]:
                res += f'{arg}'
        else:
            res = '\tpushi 0\n'
            for arg in args[::-1]:
                res += f'{arg}'
        p[0] = res + f'\tpusha {name}\n\tcall\n\tpop {len(args)}\n'
def p_call_read(p):
    'call_function : READ LPAREN RPAREN'
    if parser.success:
        p[0] = '\tread\n'
def p_call_writes(p):
    'call_function : WRITES LPAREN STRING RPAREN'
```

```
    if parser.success:
        p[0] = f'\tpushs {p[3]}\n\twrites\n'
def p_call_writesid(p):
    'call_function : WRITES LPAREN ID RPAREN'
    if parser.success:
        name = p[3]
        if name in parser.namespace:
            if parser.namespace[name]['class'] == 'var':
                if parser.namespace[name]['scope'] != parser.currentfunc:
                    print("ERROR: Not Declared!",
                          file=sys.stderr)
                    parser.success = False
                elif parser.namespace[name]['type'] != 'STR':
                    print("ERROR: Not a string variable",
                           file=sys.stderr)
                    parser.success = False
            else:
                print("ERROR: Not a valid variable!",
                      file=sys.stderr)
                parser.success = False
        else:
            print("ERROR: Not declared!",
                  file=sys.stderr)
            parser.success = False
    if parser.success:
        address = parser.namespace[name]['address']
        p[0] = f'\tpushl {address}\n\twrites\n'
def p_call_writeread(p):
    'call_function : WRITES LPAREN READ LPAREN RPAREN RPAREN'
    if parser.success:
        p[0] = '\tread\n\twrites\n'
def p_call_writeint(p):
    'call_function : WRITEI LPAREN expression RPAREN'
    if parser.success:
        p[0] = f'{p[3]}\twritei\n'
def p_call_atoi(p):
    'call_function : ATOI LPAREN STRING RPAREN'
    if parser.success:
        p[0] = f'\tpushs {p[3]}\n\tatoi\n'
def p_call_atoi_1(p):
    'call_function : ATOI LPAREN ID RPAREN'
    if parser.success:
        name = p[3]
        if name in parser.namespace:
            if parser.namespace[name]['class'] == 'var':
                if parser.namespace[name]['scope'] != parser.currentfunc:
                    print("ERROR: Not Declared!",
                          file=sys.stderr)
                    parser.success = False
                elif parser.namespace[name]['type'] != 'STR':
                    print("ERROR: Not a string variable",
                           file=sys.stderr)
                    parser.success = False
            else:
```

```
                    print("ERROR: Not a valid variable!",
                            file=sys.stderr)
                    parser.success = False
            else:
                print("ERROR: Not declared!",
                        file=sys.stderr)
                parser.success = False
    if parser.success:
        address = parser.namespace[name]['address']
        p[0] = f'\tpushl {address}\n\twrites\n'
def p_call_atoi_2(p):
    'call_function : ATOI LPAREN READ LPAREN RPAREN RPAREN'
    if parser.success:
        p[0] = '\tread\n\tatoi\n'

def p_args_lst(p):
    'args_lst : LPAREN RPAREN'
    if parser.success:
        p[0] = []
def p_args_lst_1(p):
    'args_lst : LPAREN expression args RPAREN'
    if parser.success:
        p[0] = [p[2]] + p[3]
def p_args(p):
    'args : '
    if parser.success:
        p[0] = []
def p_args_1(p):
    'args : ARRCONT expression args'
    if parser.success:
        p[0] = [p[2]] + p[3]

def p_func_type_1(p):
    'func_type : VOID'
    if parser.success:
        p[0] = p[1]

def p_func_type_2(p):
    'func_type : data_type'
    if parser.success:
        p[0] = p[1]

def p_data_type(p):
    'data_type : STR'
    if parser.success:
        p[0] = p[1]
def p_data_type_1(p):
    'data_type : INT'
    if parser.success:
        p[0] = p[1]
def p_data_type_2(p):
    'data_type : pointer data_type'
    if parser.success:
        p[0] = p[1] + ' ' + p[2]
```

```
def p_pointer_1(p):
    'pointer : REF'
    if parser.success:
        p[0] = p[1]
def p_pointer_2(p):
    'pointer : REF REF'
    if parser.success:
        p[0] = p[1] + ' ' + p[2]


def p_error(p):
    parser.success = False
    print(f'ERROR: Could not parse this file.\n{p.lineno}\n{p}',
            file=sys.stderr)
def main():
    parser.namespace = {
        'READ' : {
            'class': 'funct',
            'arguments':[],
            'return':'STR'
            },
        'WRITEI':{
            'class':'funct',
            'arguments':['INT i'],
            'return':'VOID'
            },
        'WRITES':{
            'class':'funct',
            'arguments':['STR str'],
            'return':'VOID'
            },
        'ATOI':{
            'class':'funct',
            'arguments':['STR str'],
            'return':'INT'
            },
        'INT':{'class':'data'},
        'STR':{'class':'data'},
        'IF':{'class':'reserved'},
        'ELSE':{'class':'reserved'},
        'WHILE':{'class':'reserved'},
        'RETURN':{'class':'reserved'},
        'UNTIL':{'class':'reserved'},
        'DO':{'class':'reserved'}
    }
    parser.labelcounter = 1
    parser.currentfunc  = ''
    parser.varnum       = 0
    parser.argnum       = 0
    parser.result       = ''
    parser.success      = True
    flag_err            = False
    argc                = len(sys.argv)
```

```
    flag_name = False
    if argc >= 2:
        name = re.search(r'([A-Za-z\_0-9]+)\.nqc', sys.argv[1])
        if not name:
            print("ERROR: not a nqc file",
                    file=sys.stderr)
            flag_err = True
    else:
        print("ERROR: Not enough arguments",
                file=sys.stderr)
        flag_err = True

    if not flag_err and argc > 3:
        if sys.argv[2] == '-o':
            if argc >= 4:
                new_name = re.match(r'(.*\.vm)',sys.argv[3])
                new_name = new_name.group(1)
                flag_name = True
            else:
                print("ERROR: Missing new name",
                        file=sys.stderr)
                flag_err = True

    if not flag_err:
        with open(sys.argv[1],'r',encoding='UTF-8') as f:
            cont = f.read()
        parser.parse(cont)
        res = str(parser.result)
        if parser.success:
            if flag_name:
                with open(new_name, 'w+', encoding='UTF-8') as nf:
                    nf.write(res)
            else:
                print(res)
            print("Code Generated",file=sys.stderr)
        else:
            print("Error generating code",file=sys.stderr)
    return flag_err

parser = yacc.yacc(debug=0)
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
```

```
        jz L3
        pushl -2
        pushi 1
        add
        storel -3
        jump L4
L3:
        pushl -2
        pushi 0
        equal
        jz L1
        pushi 0
        pushi 1
        pushl -1
        pushi 1
        sub
        pusha A
        call
        pop 2
        storel -3
        jump L2
L1:
        pushi 0
        pushi 0
        pushl -2
        pushi 1
        sub
        pushl -1
        pusha A
        call
        pop 2
        pushl -1
        pushi 1
        sub
        pusha A
        call
        pop 2
        storel -3
L2:
L4:
        return
        nop
BS:
        nop
        pushi 0
        pushi 0
        pushi 1
        storel 1
L8:
        pushl 1
        not
        not
        jz L9
        pushi 0
```

```
        storel 1
L6:
        pushl 0
        pushl -2
        pushi 1
        sub
        inf
        jz L7
        pushl -1
        pushl 0
        loadn
        pushl -1
        pushl 0
        pushi 1
        add
        loadn
        sup
        jz L5
        pushl -1
        pushl 0
        pushi 1
        add
        padd
        pushl -1
        pushl 0
        padd
        pusha SWAPF
        call
        pop 2
        pushi 1
        storel 1
L5:
        pushl 0
        pushi 1
        add
        storel 0
        jump L6
L7:
        jump L8
L9:
        pop 2
        return
        nop
F:
        nop
        pushi 0
        pushi 1
        storel -2
L10:
        pushl -1
        pushl 0
        sub
        pushi 0
        sup
```

```
        jz L11
        pushl 0
        pushi 1
        add
        storel 0
        pushl -2
        pushl 0
        mul
        storel -2
        jump L10
L11:
        pop 1
        return
        nop
MAIN:
        nop
        pushi 0
        pushfp
        pushi 2
        padd
        pushn 2
        pushl 1
        pushi 0
        pushi 10
        storen
        pushl 1
        pushi 1
        pushi 0
        pushi 25
        sub
        storen
        pushi 2
        pushl 1
        pusha BS
        call
        pop 2
        pushl 1
        pushi 0
        loadn
        writei
        pushs "\n"
        writes
        pushl 1
        pushi 1
        loadn
        writei
        pushs "\n"
        writes
        pushi 0
        pushi 1
        pushi 1
        pusha A
        call
        pop 2
```

```
storel 0
pushl 0
writei
pushs "\n"
writes
pushi 0
pushi 2
pusha F
call
pop 1
storel 0
pushl 0
writei
pushs "\n"
writes
pushi 0
storel -1
pop 4
return
nop
```

Listing 2.5: Matrix example

```
INT
MAIN()
BEGIN
    INT ARR[5,5];
    INT I; INT J;
    WHILE (I<=5)
    BEGIN
        WHILE (J<=5)
        BEGIN
            ARR[I,J] := I-J;
            J := J+1;
        END
        I := I+1;
    END
    MAIN := 0;
END
```

Listing 2.6: 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 0
        padd
        pushi 5
        pushi 0
        add
        padd
        pushfp
        pushi 1
        padd
        pushi 5
        pushi 1
        add
        padd
        pushfp
        pushi 2
        padd
        pushi 5
        pushi 2
        add
        padd
        pushfp
        pushi 3
        padd
        pushi 5
        pushi 3
        add
        padd
        pushfp
        pushi 4
        padd
```

```
        pushi 5
        pushi 4
        add
        padd
        pushn 25
        pushi 0
        pushi 0
L3:
        pushl 30
        pushi 5
        infeq
        jz L4
L1:
        pushl 31
        pushi 5
        infeq
        jz L2
        pushl 0
        pushl 31
        padd
        pushl 30
        pushl 30
        pushl 31
        sub
        storen
        pushl 31
        pushi 1
        add
        storel 31
        jump L1
L2:
        pushl 30
        pushi 1
        add
        storel 30
        jump L3
L4:
        pushi 0
        storel -1
        pop 32
        return
        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.