Formal Languages and Translators Laboratory

Course of Studies in Computer Science

AY 2023/2024

Luigi Di Caro, Viviana Patti and Jeremy Sproston
Department of Computer Science — University of Turin

Version dated December 5, 2023

Summary

This document describes the laboratory exercises and exam methods of the Formal Languages and Translators course for the 2023/2024 academic year.

Carrying out and evaluating the laboratory project

It is recommended to take the exam in the first exam session after the course.

Online course support and discussion forum

Two forums are available on the I-learn platform: the first is dedicated to the publication of announcements and news of a general nature, while the second is a discussion forum dedicated to the topics covered during the course. Registration to the announcements forum is carried out automatically, it is possible to unsubscribe but it is advisable to do so only after passing the exam in order to always receive communications from the teacher in a timely manner.

Laboratory project

The laboratory project consists of a series of assisted exercises aimed at the development of a simple translator. The correct carrying out of these exercises presupposes a good knowledge of the Java programming language and the theoretical topics of the Languages course Formals and Translators.

Method of laboratory examination

To take the exam at an appointment it is necessary to book. The laboratory exam is **oral** and **individual,** even if the code was developed in collaboration with other students. During the exam the following are ascertained: the correct carrying out of the laboratory test; understanding of his structure and its functioning; understanding the parts of theory related to the laboratory same.

Important notes

- In order to discuss the laboratory it is necessary to have first passed the written test relating to the theory module. The laboratory exam must be passed in the exam session in which the paper is passed, otherwise the paper must be taken again.

- The presentation of "working" code is not a sufficient condition for passing the laboratory test. In other words, it is possible to be rejected by submitting working code (if the student demonstrates lack of adequate familiarity with the code and related concepts).

- The laboratory project can be carried out individually or in groups of up to 3 students. Although the code was developed in collaboration with other students, the scores obtained by individual students are independent. For example, with the same code presented, it is possible that one student deserves 30, another 25 and yet another is rejected.

- Since during the test you may be asked to make changes to the project code, it is advisable to come to the exam with adequate knowledge of the project and related theoretical topics.

Calculation of the final grade

The marks for the written test and the laboratory test are expressed in thirtieths. The final grade is determined by calculating the weighted average of the grade of the written test and the second laboratory their contribution in CFU (with a possible modification if the student has chosen to take an oral test), that is

$$\text{final grade} = \frac{\text{written grade} \times 2 + \text{laboratory grade}}{3} \pm \text{possible oral outcome}$$

Honors can be awarded to students with a final grade of 30 and who have demonstrated particular brilliance in carrying out laboratory exercises.
Optional oral. After the laboratory discussion, or at the end of the two compulsory tests, it is possible to request, by appointment, a further oral test with the theory teachers, to improve your overall grade.

Validity of this laboratory text

This laboratory text is valid until the February 2025 session.

1 Implementation of a DFA in Java

The aim of this exercise is the implementation of a Java method that is able to discriminate the strings of the language recognized by a given deterministic finite state automaton (DFA).
The first automaton we take into consideration, shown in Figure 1, is defined on the alphabet {0, 1} and recognizes strings in which at least 3 consecutive zeros appear.

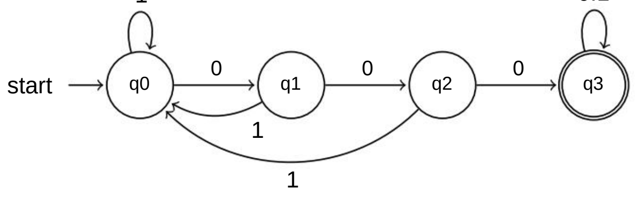


Figure 1: DFA recognizing strings with 3 consecutive zeros.

The Java implementation of the DFA in Figure 1 is shown in Listing 1. The automaton is implemented ted in the scan method that accepts a string if returns a Boolean value indicating whether the string belongs or not to the language recognized by the automaton. The state of the automaton is represented by an integer variable state, while the variable i contains the index of the next character of the string s to be analyzed. The main body of the method is a loop which, analyzing the contents of the string one character at a time, carries out a change in the state of the automaton according to its transition function. Note that the implementation assigns the value ÿ1 to the state variable if a symbol other than 0 and 1 is encountered. This value is not a valid state, but represents an unrecoverable error condition.

Listing 1: Java implementation of the DFA of Figure 1.

```
public class ThreeZeri {

    public static boolean scan(String s) {

        int state = 0; int i = 0;
        while (state >=
        0 && i < s.length()) { final char ch = s.charAt(i++);


            switch (state) { case 0:

                if (ch == '0') state = 1;
                    else if (ch ==
                '1') state = 0; else


                    state = -1; break;


            houses 1:
                if (ch == '0')
                    state = 2;
                else if (ch == '1')
                    state = 0; else

                    state = -1; break;


            houses 2:
                if (ch == '0') state = 3;
                    else if (ch ==
                '1') state = 0; else


                    state = -1; break;


            houses 3:
                if (ch == '0' || ch == '1') state = 3; else


                    state = -1; break;

            }
        }
        return state == 3;
    }
```

```
        public static void main(String[] args)
        {
                System.out.println(scan(args[0]) ? "OK" : "NOPE");
        }
}
```

Exercise 1.1. Copy the code in Listing 1, compile it, and test it on a significant set of strings, e.g. “010101”, “1100011001”, “10214”, etc.

How the DFA in Figure 1 needs to be modified to recognize complementary language, i.e. the language of strings of 0s and 1s that **do not** contain 3 consecutive zeros? Design and implement the modified DFA, and test its functioning.

Exercise 1.2. Design and implement an identifier language-aware DFA in a Java-style language: an identifier is a non-empty sequence of letters, numbers, and the "underscore" symbol that does not begin _with a number and cannot consist of just the _ symbol. Compile and test its operation on a significant set of examples.

Examples of accepted strings: “x”, “flag1”, “x2y2”, “x 1”, “lft lab”, “ temp”, “x 1 y 2”, “x ”, “ 5”

Examples of unacceptable strings: “5”, “221B”, “123”, “9 to 5”, “ ”

Exercise 1.3. Design and implement a string language-aware DFA contain a serial number followed (immediately) by a surname, where the combination of serial number and surname correspond to students in shift 2 or shift 3 of the Languages laboratory Formals and Translators. Please remember the rules for dividing students into shifts:

- Shift T1: surnames whose initial is between A and K, and the matriculation number is odd;
- Shift T2: surnames whose initial is between A and K, and the matriculation number is even;
- Shift T3: surnames whose initial is between L and Z, and the matriculation number is odd;
- Shift T4: surnames whose initial is between L and Z, and the registration number is even.

For example, “123456Bianchi” and “654321Rossi” are language strings, while “654321Bianchi” and “123456Rossi” no. In the context of this exercise, a registration number does not have a pre-established number of digits (but must consist of at least one digit). A surname corresponds to a sequence of letters, and must consist of at least one letter. Therefore the automaton must accept the strings “2Bianchi” and “122B” but not “654322” and “Rossi”.

Exercise 1.4. Design and implement a DFA that recognizes the language of floating-point numeric constants using scientific notation where the symbol e indicates the exponential function with base 10. The DFA alphabet contains the following elements: the numeric digits 0, 1, . . . ,9, the sign. (dot) preceding any decimal part, the + (plus) and - (minus) signs to indicate positivity or negativity, is the symbol And.

Accepted strings must follow the usual rules for writing numeric constants. In particular, a numeric constant consists of two segments, the second of which is optional: the first segment is a sequence of numeric digits which (1) can be preceded by a + or minus sign -, (2) can be followed by a dot sign which in turn must be followed by a non-empty sequence of numeric digits; the second segment begins with the symbol e, which a in turn and followed by a sequence of numerical digits that satisfies points (1) and (2) written for the first segment. We note that, both in the first segment and in a possible second segment, a dot sign. it does not necessarily have to be preceded by a numeric figure.

Examples of accepted strings: “123”, “123.5”, “.567”, “+7.5”, “-.7”, “67e10”, “1e-2”, “-.7e2”, “1e2.3”

Examples of unacceptable strings: “.”, “e3”, “123.”, “+e6”, “1.2.3”, “4e5e6”, “++3”

Exercise 1.5. Design and implement a DFA with the alphabet {/, *, a} that recognizes the language of "comments" delimited by /* (at the beginning) and */ (at the end): that is, the automaton must accept the strings that contain at least 4 characters starting with /*, ending with */, and containing only one occurrence of the sequence */, the final one (where the asterisk of the sequence */ does not have to be in common with the one in the /* sequence at the beginning).
Examples of accepted strings: “/****/”, “/*a*a*/”, “/*a/**/”, “/**a///a/a**/”, “/**/”, “/*/*/”

Examples of unacceptable strings: “/*/”, “/**/***/”

Exercise 1.6. Modify the automaton from the previous exercise so that it recognizes the language of strings (on the alphabet {/, *, a}) containing “comments” delimited by /* and */, but with the possibility of having before and after strings as specified below. The idea is that it is possible to have comments (even multiple) immersed in a sequence of symbols of the alphabet. Therefore the only constraint is that the automaton must accept the strings in which an occurrence of the sequence /* must be followed (even if not immediately) by an occurrence of the sequence */. Language strings may have **no** occurrences of the sequence /* (case of the sequence of symbols without comments). Implement the automaton following the construction seen in Listing 1.
Examples of accepted strings: “aaa/****/aa”, “aa/*a*a*/”, “aaaa”, “/****/”, “/*aa*/”, “*/a”, “a/**/***a”, “a/**/***/a”, “a/**/aa/***/a”

Examples of unacceptable strings: “aaa/*/aa”, “a/**//***a”, “aa/*aa”

2 Lexical analysis

The exercises in this section concern the implementation of a lexical analyzer for a simple programming language. The purpose of a lexical analyzer is to read a text and obtain a corresponding sequence of tokens, where a token corresponds to a unit lexical, such as a number, an identifier, a relational operator, a keyword, etc.
In the following sections, the lexical analyzer to be implemented will then be used to provide input to syntactic analysis and translation programs.

The language tokens are described in the way illustrated in Table 1. The first column contains the various categories of tokens, the second presents descriptions of the possible lexemes of the tokens, while the third column describes the names of the tokens, expressed as numeric constants.

The identifiers match the regular expression:

and the numbers correspond to the regular expression 0 + (1 + ... + 9)(0 + ... + 9) ÿ

The lexical analyzer will have to ignore all characters recognized as “spaces” (including tabs and carriage returns), but will have to report the presence of illicit characters, such as # or @.

The output of the lexical analyzer should have the form example: ÿtoken0ÿÿtoken1ÿ · · · ÿtokennÿ. To

- for input assign 300 to d; the output will be ÿ259, assignÿ ÿ256, 300ÿ ÿ260,toÿ ÿ257, dÿ ÿ59ÿ ÿÿ1ÿ;
- for the input print(*{dt}) the output will be ÿ267, printÿ ÿ40ÿ ÿ42ÿ ÿ123ÿ ÿ257, dÿ ÿ257,tÿ ÿ125ÿ ÿ41ÿ ÿÿ1ÿ;
- for the input if (> xy) assign 0 to x else print(y) the output will be ÿ261, ifÿ ÿ40ÿ ÿ258, >ÿ ÿ257, xÿ ÿ257, yÿ ÿ41ÿ ÿ259, assignÿ ÿ256, 0ÿ ÿ260,toÿ ÿ257, xÿ ÿ262, elseÿ ÿ267, printÿ ÿ40ÿ ÿ257, yÿ ÿ41ÿ ÿÿ1ÿ;

Tokens	Patterns	First name
Numbers	Numerical constant 256	
Identifier	Letter followed by letters and numbers 257	
Relop	Relational operator (<,>,<=,>=,==,<>) 258	
Assignment	assign 259	
To	to	260
If	if	261
Elsa	else	262
Do	do	263
For	for	264
Begin	begin	265
End	end	266
Print	print	267
Read	read	268
Initialization	:=	269
Disjunction	||	270
Conjunction	&&	271
Denial	!	33
Left round bracket (		40
Right round bracket)		41
Left square bracket [		91
Right square bracket]		93
Left curly bracket {		123
Right curly bracket }		125
Sum	+	43
Subtraction	-	45
Multiplication	*	42
Division	/	47
Semicolon	;	59
Comma	,	44
EOF	End of input	-1

Table 1: Description of language tokens

- for input for (dog:=0; dog<=printread) assign dog+1 to dog the output will be ÿ264, forÿ ÿ40ÿ ÿ257, dogÿ ÿ269, :=ÿ ÿ256, 0ÿ ÿ59ÿ ÿ257, dogÿ ÿ258, <=ÿ ÿ257, printreadÿ ÿ41ÿ ÿ259, assignÿ ÿ257, dogÿ ÿ43ÿ ÿ256, 1ÿ ÿ260,toÿ ÿ257, dogÿ ÿÿ1ÿ.

In general, the tokens in Table 1 have an attribute: for example, the token attribute ÿ256, 300ÿ is the number 300, while the token attribute ÿ259, assignÿ is the string ` assign. Note, however, that some tokens in Table 1 are attributeless: for example, the "for" sign (*) is represented by the token ÿ42ÿ, and the right round bracket ()) is represented by the token ÿ41ÿ.

Note: the lexical analyzer is not responsible for recognizing the command structure of language. Therefore, it will also accept "incorrect" commands such as:

- 5+;)
- (34+26(- (2+15-(27
- else 5 == print < end

Other errors however, such as unexpected symbols or illicit sequences (for example in the case of input 17&5, or the input |||), must be detected.

Support classes. To create the lexical analyzer, the following classes can be used. We define a Tag class in Listing 2, using the integer constants in the Name in column Table 1 to represent token names. For tokens that match a single character (except < and >, which correspond to "Relop", i.e. the relational operators), you can use the ASCII code of the character: for example, the name in Table 1 of the sum sign (+) and 43, the ASCII code of the +.

Listing 2: Tag Class

```
public class Tag {
    public final static int
        EOF = -1, NUM = 256, ID = 257, RELOP = 258,
        ASSIGN = 259, TO = 260, IF = 261, ELSE = 262,
        C = 263, FOR = 264, BEGIN = 265, END = 266,
        PRINT = 267, READ = 268, INIT = 269, OR = 270, AND = 271;
}
```

We define a Token class to represent tokens (a possible implementation of Token class and in Listing 3). We also define the Word class derived from Token, to represent the tokens that correspond to identifiers, keywords, relational operators and syntax elements that consist of multiple characters (for example &&). A possible implementation of the Word class is in Listing 4. Inspired by the Word class, you can extend Listing 5 to define a NumberTok class to represent tokens that correspond to numbers.

Listing 3: Token Class

```
public class Token {
    public final int tag;
    public Token(int t) { tag = t; }
    public String toString() {return "<" + tag + ">";}
    public static final Token
        not = new Token('!'),
        lpt = new Token('('),
        rpt = new Token(')'),
        lpq = new Token('['),
        rpq = new Token(']'),
        lpg = new Token('{'),
        rpg = new Token('}'),
        plus = new Token('+'),
        minus = new Token('-'),
        mult = new Token('*'),
        div = new Token('/'),
        semicolon = new Token(';'),
        comma = new Token(',');
}
```

Listing 4: Word Class

```
public class Word extends Token {
    public String lexeme = "";
    public Word(int tag, String s) { super(tag); lexeme=s; }
    public String toString() { return "<" + tag + ", " + lexeme + ">"; }
    public static final Word
        assign = new Word(Tag.ASSIGN, "assign"),
        to = new Word(Tag.TO, "to"),
        iftok = new Word(Tag.IF, "if"),
        elsetok = new Word(Tag.ELSE, "else"),
        dotok = new Word(Tag.DO, "do"),
        fortok = new Word(Tag.FOR, "for"),
        begin = new Word(Tag.BEGIN, "begin"),
        end = new Word(Tag.END, "end"), print = new
        Word(Tag.PRINT, "print"), read = new Word(Tag.READ,
        "read"), init = new Word(Tag.INIT , ":="), or = new
        Word(Tag.OR, "||"), and = new Word(Tag.AND, "&&"),
        lt = new Word(Tag.RELOP, "<"), gt = new
        Word(Tag.RELOP, ">"), eq = new Word(Tag.RELOP,
        "=="), le = new Word(Tag.RELOP, "<="), ne =
        new Word(Tag .RELOP, "<>"), ge = new
        Word(Tag.RELOP, ">=");




}
```

Listing 5: NumberTok class

```
public class NumberTok extends Token { // ... complete ...

}
```

A possible structure of the lexical analyzer (inspired by the text [1, Appendix A.3]) is described in the Lexer class in Listing 6.

Listing 6: Lexical analyzer of simple commands

```
import java.io.*; import
java.util.*;

public class Lexer {

        public static int line = 1; private char peek
        = ' ';

        private void readch(BufferedReader br) {
                try
                    { peek = (char) br.read(); } catch
                 (IOException exc) {
                        peek = (char) -1; // ERROR
                 }
        }

        public Token lexical_scan(BufferedReader br) { while (peek == ' ' || peek == '\t'
                || peek == '\n' || peek == '\r') { if (peek == '\ n') line++; readch(br);


                 }

                switch (peek) {
                        homes '!':
                                peek = ' '; return
                                Token.not;

                // ... handle cases of ( )                    [ ]     {    } + - * / ;            , ... //

                        homes '&':
                                readch(br); if
                                (peek == '&') { peek = ' ';
                                        return Word.and; }
                                        else {
```

```
                        System.err.println("Erroneous character"
                                  +   " after & : + peek );  "
                        return null;
                    }

        // ... handle || cases < > <= >= == <>                          ... //

                case (char)-1:
                    return new Token(Tag.EOF);

                default:
                    if (Character.isLetter(peek)) {

        // ... handle the case of identifiers and keywords //

                    } else if (Character.isDigit(peek)) {

        // ... handle the case of numbers ... //

                    } else {
                            System.err.println("Erroneous character: "
                                      + peek );
                            return null;
                    }
            }
    }

    public static void main(String[] args) {
        Lexer lex = new Lexer();
        String path = "...path..."; // the path to the file to read
        try {
            BufferedReader br = new BufferedReader(new FileReader(path));
            Tok Token;
            do {
                tok = lex.lexical_scan(br);
                System.out.println("Scan: "                  + tok);
            } while (tok.tag != Tag.EOF);
            br.close();
        } catch (IOException e) {e.printStackTrace();}
    }

}
```

Exercise 2.1. Write a lexical analyzer in Java that reads input from a file and prints it matching token sequence. For this exercise, you can use them without modification Tag, Token and Word classes. Instead the NumberTok and Lexer classes need to be completed.

Exercise 2.2. Consider the following new definition of identifiers: an identifier is a non-empty sequence of letters, numbers, and the “underscore” symbol _ ; the sequence does not begins with a number and cannot be composed only of the symbol. More precisely, the identifiers correspond to the regular expression:

_ _ _

(where a + ... + Z abbreviates the regular expression to + ... + z + A + ... + Z). Extend the method lexical_scan to handle identifiers that match the new definition.

Exercise 2.3. Extend the lexical_scan method so that it can handle the presence of comments in the input file. Comments can be written in two ways:

• comments delimited with /* and */;

• comments starting with // and ending with a newline or EOF.

Comments should be ignored by the lexical analysis program; in other words, for the parts of the input that contain comments, no token should be generated. For example, Let's consider the following input.

```
/* calculate the speed */
assign 300 to d; // distance
assign 10 to t; // time
print(* dt)
```

The output of the lexical analysis program will be ÿ259, assignÿ ÿ256, 300ÿ ÿ260,toÿ ÿ257, dÿ ÿ59ÿ ÿ259, assignÿ ÿ256, 10ÿ ÿ260,toÿ ÿ257,tÿ ÿ59ÿ ÿ267, printÿ ÿ40ÿ ÿ42ÿ ÿ257, dÿ ÿ257,tÿ ÿ41ÿ ÿÿ1 ÿ.

In addition to the symbol pairs /*, */, and //, a comment can contain symbols that are not part of any token's pattern (for example, /*@#?*/ or /*calculate speed*/).
If a comment of the form /* ... */ is opened but not closed before the end of the file (see for example the case of input assign 300 to d /*distance) an error must be reported. Yes note that there can be multiple consecutive comments not separated by any token, for example:

```
assign 300 to d /*distance*//*from Turin to Lyon*/
```

Furthermore, the symbol pair */, if written outside a comment, must be treated by lexer as the multiplication sign followed by the division sign (for example, for input x*/y the output will be ÿ257, xÿ ÿ42ÿ ÿ47ÿ ÿ257, yÿ ÿÿ1ÿ). In other words, the idea is that in this case the sequence of symbols */ will not be interpreted as the closing of a comment but as a sequence of the two tokens mentioned.

3 Syntactic analysis

Exercise 3.1. Write a recursive descent parser that parses expressions very simple arithmetic, written in infix notation, and composed only of non-negative numbers (i.e. sequences of decimal digits), addition and subtraction operators + and -, operators of multiplication and division * and /, bracket symbols (and). In particular, the analyzer must recognize the expressions generated by the grammar

where P is the following set of productions:

Note that we use ::= instead of´ ÿ to indicate a production, for example ÿstartÿ::=ÿexpr ÿEOF and a production with head ÿstartÿ and body ÿexpr ÿEOF.

The program must make use of the previously developed lexical analyzer. Note that the token set corresponding to the grammar of this section is a subset of the token set corresponding to the lexical rules of Section 2. In cases where the input matches the grammar, the output must consist of the input token list followed by a message indicating that the input matches the grammar. Instead, in cases where the input does not correspond to the grammar, the output of the program must consist of an error message (as illustrated in the classroom lessons) indicating the procedure being executed when the error was detected.

A possible structure of the program follows (inspired by the text [1, Appendix A.8]).

Listing 7: Simple expression parser **import** java.io.*;

```
public class Parser { private Lexer
    lex; private BufferedReader
    pbr; private Token look;


    public Parser(Lexer l, BufferedReader br) {
        lex = l; pbr =
        br; move();

    }

    void move() {
        look = lex.lexical_scan(pbr);
        System.out.println("token =                    "   + look);
    }

    void error(String s) { throw new
        Error("near line "                    + lex.line + ": "              + s);
    }

    void match(int t) { if (look.tag
        == t) { if (look.tag != Tag.EOF)
            move();
        } else error("syntax error");
    }

    public void start() { // ... complete ...
        expr(); match(Tag.EOF); // ...
        complete ...


    }

    private void expr() { // ... complete ...

    }

    private void exprp() {
        switch (look.tag) { case '+':

        // ... complete ... }

    }
    private void term() { // ...
        complete ...
    }

    private void termp() {
        // ... complete ...
    }

    private void fact() { // ...
        complete ...
    }

    public static void main(String[] args) { Lexer lex = new Lexer();
        String path = "...path..."; // the path of
        the file to read try { BufferedReader br = new BufferedReader(new FileReader(path)); Parser
        parser
            = new Parser(lex, br); parser.start(); System.out.println("Input OK"); br.close(); } catch
            (IOException e) {e.printStackTrace();}




    }
}
```

Exercise 3.2. Following are the productions of a grammar for a simple programming language. As in Exercise 3.1, variables are denoted with angle brackets (for example, ÿprogÿ, ÿstatlistÿ, ÿstatlistpÿ, etc.). The terminals of the grammar correspond to the tokens described in Section 2 (in Table 1).

Note that RELOP matches an element of the set {==, <>, <=, >=, <, >}, NUM matches
to a numeric constant and ID corresponds to an identifier. Also, note that the expressions
arithmetic are written in prefix or Polish notation, unlike what happened in the previous exercise where they
were written according to infix (standard) notation. Similarly the
Boolean expressions are written in prefix notation, following the convention of placing the relational operator
to the left of the expressions. Modify the grammar to obtain an equivalent LL(1) grammar, and write a
recursive dropdown parser for the grammar
obtained.

4 Direct translation from syntax

Exercise 4.1 (Evaluator of simple expressions). Modify the parser of exercise 3.1 to evaluate simple
arithmetic expressions, referring to the scheme

of translation directed by the following syntax:

Note that the NUM terminal has the value attribute, which is the numeric value of the terminal, provided by the lexical analyzer.

A possible structure of the program is the following.

Note: As indicated, it is strongly recommended that you create a new class (called Evaluator in Listing 8).

Listing 8: Evaluation of simple expressions

```
import java.io.*;

public class Evaluator { private Lexer lex;
    private BufferedReader pbr;
    private Token look;


    public Evaluator(Lexer l, BufferedReader br) { lex = l;

        pbr = br;
        move();
    }

    void move() {
        // as in Exercise 3.1
    }

    void error(String s) { // as in Exercise
        3.1
    }

    void match(int t) { // as in
        Exercise 3.1
    }

    public void start() { int expr_val;


        // ... complete ...

        expr_val = expr();
        match(Tag.EOF);
```

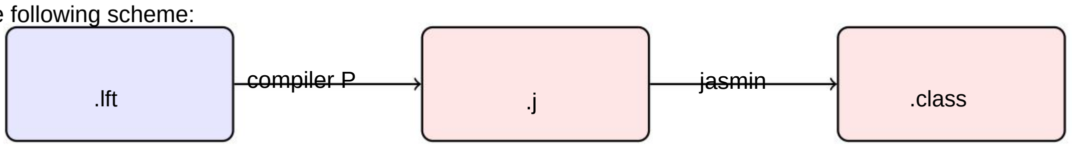
```
            System.out.println(expr_val);

            // ... complete ...
      }

      private int expr() { int term_val,
            exprp_val;

            // ... complete ...

            term_val = term();
            exprp_val = exprp(term_val);

            // ... complete ... return exprp_val;

      }

      private int exprp(int exprp_i) { int term_val, exprp_val;
            switch (look.tag) { case '+':


                        match('+');
                        term_val = term(); exprp_val
                        = exprp(exprp_i + term_val); break;


            // ... complete ... }

      }

      private int term() { // ... complete ...

      }

      private int termp(int termp_i) { // ... complete ...

      }

      private int fact() { // ... complete ...

      }

      public static void main(String[] args) { Lexer lex = new Lexer();
            String path = "...path..."; // the path of
            the file to read try { BufferedReader br = new BufferedReader(new FileReader(path)); Evaluator evaluator
            = new
                  Evaluator(lex, br); evaluator.start(); br.close(); } catch (IOException e) {e.printStackTrace();}




      }
}
```

5 Bytecode generation

The objective of this last part of the laboratory is to create a translator for programs written in the simple programming language, which we will call P, seen in exercise 3.2. P language program files have a .lft extension, as suggested in the lessons
of theory. The translator must generate bytecode [4] executable by the Java Virtual Machine (JVM).

Generating executable bytecode directly from the JVM is not a simple operation due to the complexity of the .class file format (which, among other things, is a binary format) [3]. The bytecode will then be generated using a mnemonic language that refers to the JVM instructions (assembler language [2]) and which will subsequently be translated into the .class format by the assembler program. The mnemonic language used refers
to the set of JVM instructions [5] and the assembler performs a 1-1 translation of the mnemonic instructions into the corresponding instruction (opcode) of the JVM. The assembler program
that we will use is called Jasmin (distribution and manual are available at http:
//jasmin.sourceforge.net/).

The construction of the .class file starting from the source written in the P language occurs according to the following scheme:



The source file is translated by the compiler (object of implementation) into the assembly language for the JVM. This file (which must have the .j extension) is then transformed into a file by the Jasmin .class
assembler program. In the code presented in this section, the file generated by the
compiler is called Output.j, and the command java -jar jasmin.jar Output.je uses `
zated to transform it into the Output.class file, which can be executed with the java command
Outputs.

Meaning of P language instructions.

assign [ÿexpr ÿ1 to ÿidlistÿ1] · · · **[ÿexpr ÿn to** ÿidlistÿn]

For every i ÿ {1, · · · , n}, assign the value of the expression ÿexpr ÿi to all identifiers included in the identifier list ÿidlistÿi . For example, education

\texttt {assign [3 to x,y] [+(x,y) to z]}

corresponds to assigning the value 3 to both identifiers x and y, followed by assigning the sum of the value of x and the value of y to the identifier z.

print (ÿexprlistÿ)

Prints the value of all expressions included in the expression listÿexprlistÿ on the terminal.
For example, print(+(2,3),4]) prints the value 5 followed by the value 4 on the terminal.

read (ÿidlistÿ)

The command read(ID1, ..., IDn) allows the insertion of n non-negative integers from keyboard; the i-th number entered from the keyboard and assigned to the identifier IDi . For example, the command read(a,b) specifies that the user of the program written in P `
must enter two non-negative integers with the keyboard, the first of which is assigned to the identifier a and the second of which is assigned to the identifier b.

for (ÿbexpr ÿ) do ÿstatÿ

Allows cyclic execution of ÿstatÿ. The condition for the loop execution is ÿbexpr ÿ.
Note that ÿstatÿ can be a single statement or a sequence of statements enclosed in curly braces. For example, education

\texttt {for (> x 0) \{ assign [- x 1 to x]; print(x) \} }

decrements and prints the value of the identifier x on the terminal, until x reaches the value 0.

for (ID := ÿexpr ÿ ; ÿbexpr ÿ) **do** ÿstatÿ

This command is similar to the previous one, except for the identifier ID and assigned the value of the expression ÿexpr ÿ at the beginning of the command execution, that is before the first check of the condition ÿbexprÿ. For example, education

\texttt {for (y := 10; > y 0) \{ assign [- y 1 to y]; print(y) \} }

assigns the value 10 to the identifier y, then decrements and prints the value of on the terminal y, until y reaches the value 0.

if (ÿbexpr ÿ) ÿstatÿ **end**

It is a conditional command that resembles the Java if command (without else): if the condition ÿbexprÿ is evaluated as true then ÿstatÿ is executed, then we proceed to the instruction following the if...end; instead if the condition ÿbexprÿ is evaluated as false we proceed directly to the instruction following the if...end.

if (ÿbexpr ÿ) ÿstat1 ÿ **else** ÿstat2 ÿ **end**

It is a conditional command that resembles the Java if...else... command: if the condition ÿbexprÿ is evaluated as true then ÿstat1ÿ is executed, then we proceed to the statement following the if ...else...end; instead, if the condition ÿbexprÿis evaluated as false then ÿstat2ÿ is executed, then we proceed to the instruction following the if...else... end.

{ ÿstatlistÿ }

Allows you to group a sequence of instructions. For example, a block of statements which reads a number as input and subsequently prints the number incremented by 1 on the terminal can be written as follows: {read(x); print(+(x,1))}.

Note the use of prefix notation for arithmetic expressions. Also, operations addition and multiplication operations can have n number of arguments with n ÿ 1. Instead, subtraction and division operations have exactly two arguments. For example, the expression *(2,3,4) has value 24, the expression -*(2,4)3 has value 5, the expression +(2,- 7 3) has value 6, the expression +(/ 10 2, 3) has the value 8, and the expression +(5, - 7 3, 10) has the value 19.

Support classes. To create the compiler we use the following classes.

The OpCode class is a simple enumeration of mnemonic names (see [5] for a list of names, but we only use those useful for translating the P language). The Instruction class will be used to represent individual instructions of the mnemonic language. The toJasmin method returns the statement in the appropriate format for the Jasmin assembler.

Listing 9: A possible implementation of the OpCode class

```
public enum OpCode {
    ldc, imul, ineg, idiv, iadd,
    isub, istore, ior, iand, iload,
    if_icmpeq, if_icmple, if_icmplt, if_icmpne, if_icmpge,
    if_icmpgt, ifne, GOto, invokestatic, dup, pop, label }
```

Listing 10: A possible implementation of the Instruction class

```
public class Instruction {
    OpCode opCode;
    int operand;
```

```
        public Instruction(OpCode opCode) {
                this.opCode = opCode;
        }

        public Instruction(OpCode opCode, int operand) {
                this.opCode = opCode;
                this.operand = operand;
        }

        public String toJasmin() {
                String temp="";
                switch (opCode) {
                        case ldc : temp = case          " ldc "         + operand + "\n"; break;
                        invokestatic :
                              if( operand == 1)
                                    temp =      " invokestatic "              + "Output/print(I)V" + "\n";
                              else
                                    temp =      " invokestatic "              + "Output/read()I" + "\n"; break;
                        case iadd : temp = + "\n"; break;"iadd"
                        case imul : temp = + "\n"; break;"imul"
                        case idiv : temp = + "\n"; break;" idiv "
                        case isub : temp = + "\n"; break;"isub"
                        case ineg : temp = " ineg " + "\n"; break;
                        case istore : temp = " istore " + "\n"; break;          + operand + "\n"; break;
                        case ior : temp = " ior " case iand : temp
                        = case iload : temp = case      "iand"          + "\n"; break;
                        if_icmpeq : temp = case           " iload " "         + operand + "\n"; break;
                        if_icmple : temp = case if_icmplt :       if_icmpeq L" + operand + "\n"; break;
                        temp = case if_icmpne : temp = case       " if_icmple L" + operand + "\n"; break;
                        if_icmpge : temp = case if_icmpgt :       " if_icmplt L" + operand + "\n"; break;
                        temp = case ifne : temp = goto L" +       " if_icmpne L" + operand + "\n"; break;
                        operand + "\n" ; break;                   " if_icmpge L" + operand + "\n"; break;
                                                                  " if_icmpgt L" + operand + "\n"; break;
                                                      " ifne L" + operand + "\n"; break;
                                                      "
                        case GOto : temp = " dup " +
                        case dup : temp = " pop" +     "\n" ; break;
                        case pop : temp = case      "\n" ; break;
                        label : temp = "L" + operand + ":\n"; break;
                }
        return temp;
        }
}
```

The CodeGenerator class has the purpose of storing the list of instructions (such as objects of type Instruction) generated during parsification. The emit methods they are used to add jump instructions or labels in your code. The header and footer constants define the preamble and epilogue of the code generated by the translator to return, through the toJasmin method, a file whose structure meets the requirements of the Jasmin assembler.

Listing 11: A possible implementation of the CodeGenerator class

```
import java.util.LinkedList;
import java.io.*;

public class CodeGenerator {

        LinkedList <Instruction> instructions = new LinkedList <Instruction>();

        int label=0;
```

```
    public void emit(OpCode opCode) {
            instructions.add(new Instruction(opCode));
    }

    public void emit(OpCode opCode ,                        int operand) {
            instructions.add(new Instruction(opCode, operand));
    }

    public void emitLabel(int operand) {
            emit(OpCode.label, operand);
    }

    public int newLabel() {
            return label++;
    }

    public void toJasmin() throws IOException{
            PrintWriter out = new PrintWriter(new FileWriter("Output.j"));
            String temp = "";
            temp = temp + header;
            while(instructions.size() > 0){
                    Instruction tmp = instructions.remove();
                    temp = temp + tmp.toJasmin();
            }
            temp = temp + footer;
            out.println(temp);
            out.flush();
            out.close();
    }

    private static final String header = ".class public Output \n"
            + ".super java/lang/Object\n"
            + "\n"
            + ".method public <init>()V\n"
            +   " aload_0\n"
            +   " invokenonvirtual java/lang/Object/<init>()V\n"
            +   " return\n"
            + ".end method\n"
            + "\n"
            + ".method public static print(I)V\n"
            +   " .limit stack 2\n"
            +   " getstatic java/lang/System/out Ljava/io/PrintStream;\n"
            +   " iload_0 \n"
            +   " invokestatic java/lang/Integer/toString(I)Ljava/lang/String;\n"
            +   " invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V\n"
            +   " return\n"
            + ".end method\n"
            + "\n"
            + ".method public static read()I\n"
            +   " .limit stack 3\n"
            +   "
                  new java/util/Scanner\n"
            +   "dup\n"
            +   " getstatic java/lang/System/in Ljava/io/InputStream;\n"
            +   " invokespecial java/util/Scanner/<init>(Ljava/io/InputStream;)V\n"
            +   " invokevirtual java/util/Scanner/next()Ljava/lang/String;\n"
            +   " invokestatic java/lang/Integer.parseInt(Ljava/lang/String;)I\n"
            +   "ireturn\n"
            + ".end method\n"
```

```
               + "\n" +
               ".method public static run()V\n"
               +   " .limit stack 1024\n" " .limit locals
               +   256\n";

        private static final String footer = + ".end method\n" +       " return\n"
               "\n" + ".method public static

               main([Ljava/lang/String;)V\n"
               +   " invokestatic Output/run()V\n" " return\n" + ".end
               +   method\n";

}
```

To keep track of identifiers and their addresses, a symbol table must be set up. The address associated with an identifier can be used as an argument to the iload or istore commands. The following is a possible implementation of the SymbolTable class:

Listing 12: A possible implementation of the SymbolTable class

```
public class SymbolTable {

        Map <String, Integer> OffsetMap = new HashMap <String,Integer>();

        public void insert( String s, int address )                              {
                       if( !OffsetMap.containsValue(address) )
                              OffsetMap.put(s,address);
                       else
                              throw new IllegalArgumentException("Reference to a
                                    memory location already occupied by another variable");
        }

        public int lookupAddress ( String s )                          {
                       if( OffsetMap.containsKey(s) ) return
                              OffsetMap.get(s); else

                              return -1;
        }
}
```

Bytecode generation examples. Below are some simple P programs accompanied by two corresponding examples of JVM bytecode. Note that there may be different ways of correctly translating the same P program.

Listing 13: A.lft Program

```
read(a);
print(+(a,1))
```

Listing 14: A.lft invokestatic bytecode example Output/

```
 read()I istore 0

 goto L1
L1:
 iload 0
 ldc 1
 iadd
 invokestatic Output/print(I)V goto L2

L2:
 goto L0
L0:
```

Listing 15: B.lft Program

```
assign [10 to x] [20 to y]
           [30 to z];

print(+(x,*(y,z)))
```

Listing 16: Example of B.lft bytecode

```
 ldc 10
 istore 0
 ldc 20
 istore 1
 ldc 30
 istore 2
 goto L1
L1:
 iload 0
 iload 1
 iload 2
 imul
 iadd
 invokestatic Output/print(I)V goto L2

L2:
 goto L0
L0:
```

Exercise 5.1. Write a translator for programs written in the P language (using one of the lexers developed for the exercises in Section 2). Remember that the grammar of the language P is defined in exercise 3.2.

A code snippet of a possible implementation can be found in Listing 17 (note that code is an object of the CodeGenerator class).

Listing 17: A code fragment of a possible implementation of the program in Exercise 5.1

```
import java.io.*;

public class Translator { private Lexer
     lex; private BufferedReader
     pbr; private Token look;


     SymbolTable st = new SymbolTable();
     CodeGenerator code = new CodeGenerator();
     int count=0;
```

```
public Translator(Lexer l, BufferedReader br) { lex = l; pbr = br; move();



 }

void move() {
        // as in Exercise 3.1
 }

void error(String s) { // as in Exercise
        3.1
 }

void match(int t) { // as in
        Exercise 3.1
 }

public void prog() { // ...
        complete ... int lnext_prog =
        code.newLabel(); statlist(lnext_prog);
        code.emitLabel(lnext_prog);
        match(Tag.EOF); try {


                        code.toJasmin();
        }
        catch(java.io.IOException e) { System.out.println("IO
                        error\n");

        }; // ... complete ...
 }

public void stat( /* complete */ ) switch(look.tag) { // ...              {
        complete ... case Tag.READ:
        match(Tag.READ); match('('); idlist(/
                * complete */);
                        match(')'); // ... complete ... }






   }

private void idlist(/* complete */) {
        switch(look.tag) { case Tag.ID:

                        int id_addr = st.lookupAddress(((Word)look).lexeme); if (id_addr==-1) { id_addr = count;
                        st.insert(((Word)look).lexeme,count+
                                +);

                        }
                        match(Tag.ID); // ...
        complete ... }

 }
```

```
    private void expr( /* complete */ )                              {
        switch(look.tag) { // ...
        complete ...
            homes '-':
                match('-'); expr();
                expr();

                code.emit(OpCode.isub); break;

        // ... complete ... }

    }

// ... complete ... }
```

Bibliographical references

[1] Aho, Alfred V., Lam, Monica S., Sethi, Ravi, and Ullman, Jeffrey D. Compilers: Principles, techniques and tools. Pearson, 2019.

[2] Assembly language. http://en.wikipedia.org/wiki/Assembly_language Wikipe- day, 2023.

[3] Java class file. http://en.wikipedia.org/wiki/Java_class_file Wikipedia, 2023.

[4] Java bytecode. http://en.wikipedia.org/wiki/Java_bytecode Wikipedia, 2023.

[5] Java bytecode instruction listings. http://en.wikipedia.org/wiki/Java_ bytecode_instruction_listings Wikipedia, 2023.