People's Democratic Republic of Algeria

Ministry of Higher Education and Scientific Research

ABDELHAMID IBN BADIS UNIVERSITY – UMAB – MOSTAGANEM
FACULTY OF EXACT SCIENCES AND COMPUTER SCIENCE
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

Essentials of Compiler Design Methods

With Exercises and Solutions

Educational handout produced by:

Dr. HOCINE Nadia

Assistant professor (Maître de Conférences - classe A)
Department of Mathematics and Computer Science
University of Mostaganem

Handout intended for third-year Computer Science Bachelor's students
(Licence L3)
Computer Science, Information Systems

Year : 2025 - 2026

PREFACE

Compiler design is among the fundamental areas of computer science that relies on how a high-level programming language can be translated and executed by a machine. The compilation process can be understood through the study of its main phases: lexical analysis, syntax analysis, semantic analysis, generation of an intermediate code, optimization, and code generation. This handout presents the main concepts and explains the formal and practical tools underlying the phases of the compiler. The objective is to equip students with both theoretical knowledge and practical skills to understand how compilers are built. It also aims to enhance problem-solving and algorithmic thinking skills for students interested in understanding compilers' internal functioning or building their own programming language and compiler.

This handout has been created for the subject titled "Compiler Design" for third-year Computer Science Bachelor's students, specializing in Information Systems (SI), at Abdelhamid Ibn Badis University, Faculty of Exact Sciences and Computer Science, Department of Mathematics and Computer Science. It is structured according to the course syllabus as follows:

Chapter I. Introduction to the Internal Structure and Process of a Compiler. This chapter introduces the main compilation modules and explains their internal functioning.

Chapter II. Lexical Analysis. This chapter addresses the first step in compiler analysis: recognizing the lexical units (or tokens) that compose a language. Building on students' prior knowledge of language theory, it presents methods for converting regular expressions (used to define lexical units) into finite state automata, along with techniques for their determinization and minimization. The chapter also introduces tools that support the generation of lexical analyzers, with a focus on Flex, one of the most widely used tools in compiler design.

Chapter III. Syntax Analysis Using Top-Down Methods. This chapter covers the core concepts of syntax analysis in compiler design, with a focus on building syntactic analyzers using top-down methods—particularly the LL(1) method. It also introduces Yacc (Bison), a tool commonly used with Flex to generate the source code for syntax analyzers.

Chapter IV. Syntax Analysis Using Bottom-Up Methods. This chapter explores syntax analysis using more powerful bottom-up methods, including LR(0) and SLR(1), as well as lookahead-based techniques such as LR(1) and LALR(1).

Chapter V. Syntax-Directed Translation to Code Generation. The final chapter explores semantic analysis with an emphasis on syntax-directed translation. This analysis includes actions, such as type control, object binding, and the generation of intermediate code representations. The chapter also addresses code generation, outlining techniques for translating intermediate forms into a target code, as well as strategies for code optimization.

ACRONYMS

AX: Accumulator Register

CFG: Control Flow Graph

DFA: Deterministic finite automaton

ESP: Stack Pointer Register.

NFA: Non-deterministic finite automaton

LL: Left to right; Leftmost derivation

LR: Left to right scanning of the input, Right most derivation in reverse

LALR: Look Ahead LR

LEX: Lexical EXpression,a lexical analyzer generator

SLR: Simple LR

MDP: Member Rights Production

YACC: Yet Another Compiler Compiler

LIST OF FIGURES

LIST OF TABLES

TABLE OF CONTENTS

CHAPTER I. Introduction to Compiler Internal Structure and Process

This chapter introduces the internal structure and processes of a compiler, beginning with a definition of what a compiler is and why it's essential. It breaks down the phases of compilation, from lexical analysis to code generation, highlighting the specific role each phase plays. Finally, it introduces the tools and techniques commonly used to build compilers.

I.1. What is a compiler ?

A **compiler** is a software program that analyzes an input stream written in high-level language (source code) to detect errors and then translates it into a target code (output) written in low-level language (e.g., assembly, object language, or machine code) that a computer's processor can execute. As shown in Figure 1. a compiler can be valuable in assisting the developer by highlighting the main errors in the source code. Once corrected, the compiler follows a sequence of steps to translate it into a target code. Each high level programming language has a dedicated compiler. For instance, the compiler of C language is “gcc” or Gnu C Compiler, and “javac” is the compiler of Java language.

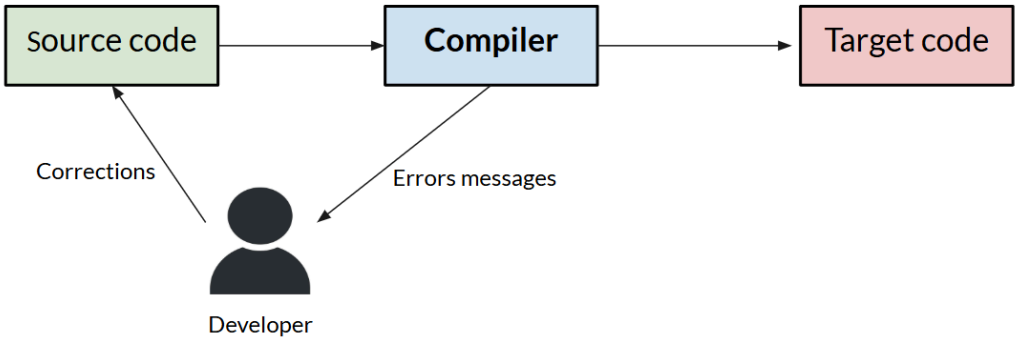


Figure 1. Compiler function

Some languages propose interpreters rather than compilers. Unlike a compiler, an interpreter does not analyze the entire source code (analyzes it instruction by instruction). It is sometimes memoryless, making it fast and tolerant of certain errors. However, after analyzing an instruction, it executes it immediately, which requires exposing the code. Examples of interpreted languages include Caml, Perl, Prolog, Python interpreter, etc.

I.2. General process of a compiler

A compiler is composed of several modules to detect lexical, syntactic, and semantic errors and then generate an intermediate code. Along these steps, a table of symbols is used to facilitate communications between compiler modules. When errors are detected, the compiler generates messages for the developer. Once corrected, the compiler generates the target code. Figure 2 summarizes the main modules and steps of a compiler process.

Among the main compiler steps are:

- **Lexical analysis:** it focuses on the recognition of stings or words in language (also called lexemes or lexical units)
- **Syntax analysis:** it focuses on the order of chaining of words (instructions of the source code)
- **Semantic analysis:** is a higher level of analysis to detect non-syntactic errors

The following subsections introduce each step and module of a compiler as illustrated in Figure 2.

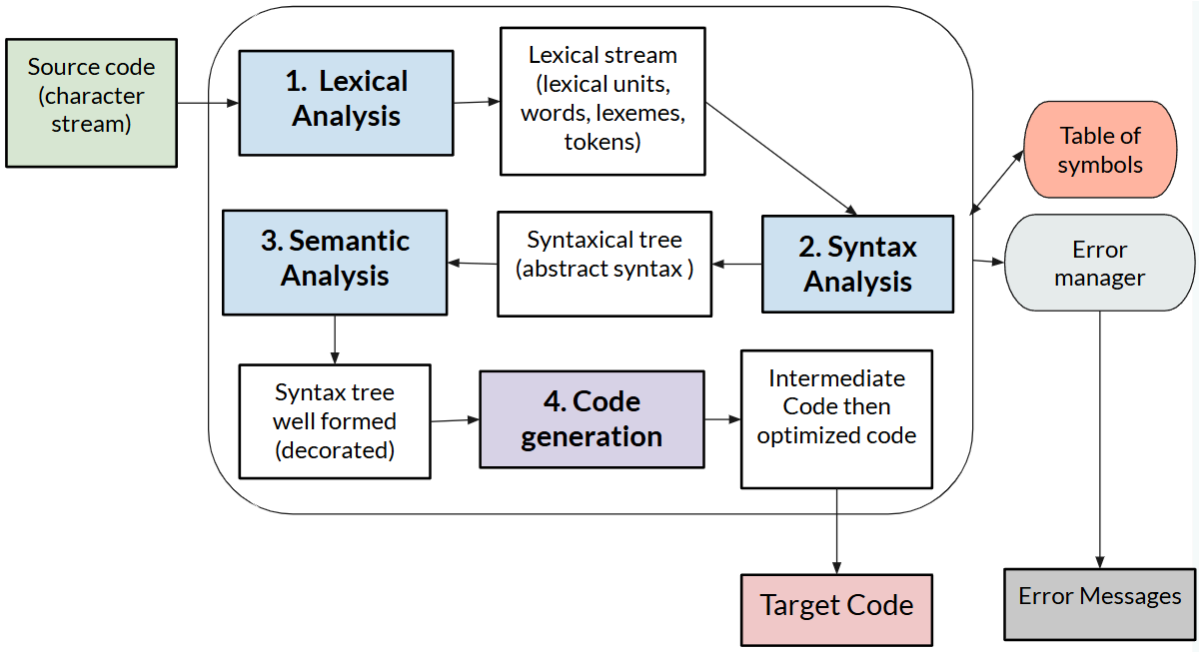


Figure 2. The internal modules and functions of a compiler

I.2.1 Lexical Analysis

During this phase, the stream of characters forming the source program is read from left to right and grouped into "**lexical units**" (also called lexemes, words or tokens), which are sequences of characters that have a collective meaning. Identifiers, keywords, constants, and operators are all examples of lexical units. Comments and whitespace separating the characters that form these lexical units are removed during this phase.

The lexical analyzer read the stream of characters to detect lexical units and return tokens. A token is a positive integer that represents the lexical category of the detected word along with its associated semantic value.

Example:

Let consider the following C instruction: *x = y + 2;*

The *lexical units* are:

- x: variable identifier
- =: assignment symbol
- y: variable identifier
- +: operator of addition
- 2: integer value
- ;: separator indicating the end of an instruction

The lexical analyzer (or parser) stores the identifiers in the **symbol table** with their tokens.

I.2.2 Syntax analysis

This phase consists of grouping the lexical units of the source program into **grammatical structures.** The syntax analyzer checks whether a program is correctly written according to the grammar that specifies the syntactic structure of the language.

In this phase, additional information about lexical units, such as the identifier type, is recorded in the symbol table. The syntax analysis produces an intermediate syntactic representation of the code (often a syntax tree). In case of an error, the process stops and passes it to the error manager.

Example:

Let us consider the following grammar G of arithmetic expressions and analyze the following instruction: x= y + 2;

G: S -> E = T
T -> T + T | T - T | E
E -> id| 0|1|...9

Syntax analysis of this sentence produces the syntax tree of Figure 3:

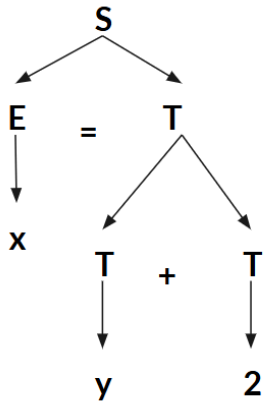


Figure 3. An example of a syntax tree of an instruction

I.2.3 Semantic analysis

In this phase, semantic actions (code snippets) are inserted at specific points in the grammar to address the particularities of the programming language. For example, this phase checks that the operands of each operator conform to the source language specifications. It can also detect for instance incorrect assignment relative to the declaration type of a variable. The semantic analyzer collects information for the production of intermediate code.

For example in the instruction: x= y + 2, the syntax analysis allows recognizing whether the instruction follows the grammar and identifies the types of variables x and y (real). In the semantic analysis, the value 2 will be replaced with 2.0.

I.2.4 Code generation

After the syntax and semantic analysis phases, some compilers explicitly build an **intermediate representation** of the source program. This representation should be easy to generate and easy to translate into the target language.

Different forms of intermediate representation have been proposed and the common representation in the **three-address code form**. In this form, each instruction has at most three operands. For example, for an instruction id_1 + id_2 - id_3 can be translated into three-address code as follows:

$temp_1$:= id_2 - id_3
$temp_2$:= id_1 + $temp_1$

where $temp_1$ and $temp_2$ are temporary variables.

Different intermediate code generation techniques can be used such as abstract machine (or virtual), p-code, byte-code (java), etc.

The intermediate code can be optimized to reduce execution time or memory usage. Among the common optimization techniques include:

- **Invariant Code Motion**: it consists in identifying parts of the code that are invariant and move them out of the body of a loop. This reduces the total number of instructions executed.
- **Dead Code Elimination**: when a part of the code is never used, the optimizer suggests removing it.
- **Common Subexpression Elimination:** when a value of a variable or an expression is calculated in multiple parts of the code, the optimizer can suggest to store and reuse the result to reduce time and memory usage.

The generation of the target code depends on the intermediate code representation such as machine code or other forms. Once the intermediate code is optimized, it can be translated into a sequence of machine instructions that represent the target code.

For example, the intermediate code instruction id_1 := id_2 + 2 can be converted into the following machine instructions:

LOAD id_2
ADD 2
STORE id_1

I.2.5 Manager of the symbol table

The communication between compilation modules (lexical analyzer, syntax analyzer, semantic analyzer) is done through the **symbol table.**

The symbol table is a data structure, often a table of records (name, information) associated with source code identifiers (of variables, constants, functions, etc). The record contains information such as the identifier name, its type, and the memory location. This information is inserted during the analysis process by the various compilation modules. In particular, the information recorded for each identifier recognized in the source code is: the name or the string for identification, primary type or data structure, memory address and size.

The manager of the symbol table is responsible for operations on this table while managing coherence. The operations are typically: checking whether a name of an identifier is in the table (lookup), adding an identifier name, getting information of an identifier, updating information, etc. All information is recorded during lexical and syntax analysis steps. The lexical analysis is responsible for recording the name of identifiers while the syntax analysis adds more information such as the type and memory address.

I.2.6 Error Manager

This manager allows the developer to be notified of errors detected by the various compiler modules. It is intended to return clear and precise messages that will help the developer locate and correct the error by indicating for instance the number of lines in the source code and the type or semantic of the error.

I.3. Compiler design tools

The general process to build and develop a compiler consists in:

1. Define regular expressions that define each lexical unit of the language
2. Convert the regular expressions into a finite state automaton and implement a lexical analyzer that use this automaton to recognize the lexical units of a given source code
3. Implement a syntax analyzer that focuses on a grammar definition to build an abstract syntax tree to parse the source code
4. Improve the previous analyzer to consider semantic errors, produce intermediate code, code and its optimization

These steps can be implemented using different programming languages such as C, Python, etc. We can also use an automatic generator for some parts of the compiler. For example, (F)Lex is a common generator of lexical analyzers using a specification based on regular expressions while Yacc (or Bison) is a common automatic generator of syntactic analyzers using a specification based on a context-free grammar. These tools generate the

essential code source of the analyzers in different programming languages that can be used to develop the compiler.

The next chapter aims to present formal methods for building a lexical analyzer, as well as its implementation using evolutionary programming languages and tools or generators such as Flex.

CHAPTER II. Lexical Analysis

The goal of lexical analysis is to read the input source code character by character and identify lexical units such as the language keywords, identifiers, operators, etc. Once identified it returns tokens that indicate their lexical category and stores them in the symbol table. In the event of an error, the lexical analyzer must locate the error and the error handler provides a meaningful message to the developer.

II.1. Lexical analysis general process

Two methods of lexical analysis operations can be considered:

- **Sequential**: lexical analysis of the entire code then passing the result to the syntax analyzer
- **Parallel**: lexical analysis operates at the request of the syntax analyzer during code analysis. As shown in Figure 4. The lexical units are used during syntax analysis of the source code with the help of the symbol table.

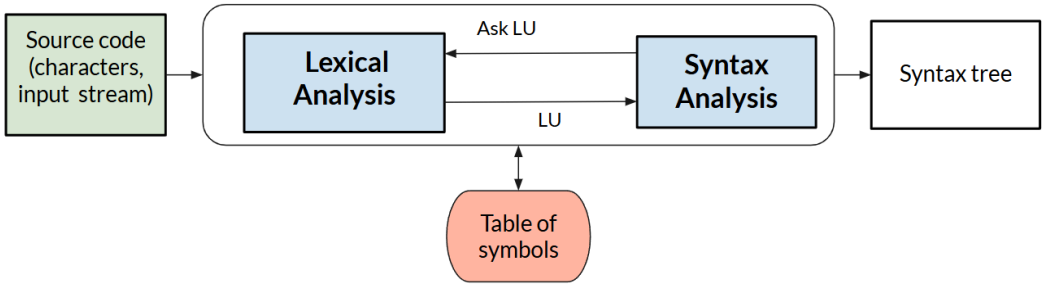


Figure 4. The parallel communication method between lexical and syntax analyzers

II.2. Internal functioning of lexical analysis

The lexical analyzer starts with reading characters by eliminating information that is useless for analysis, such as spaces, tabs, and comments. Then, it concatenates characters to recognize lexical units. It associates each symbol with a lexical unit that corresponds to a lexical category, including:

- Keywords of the language (such as if, int, for, do while, while, etc)
- Identifiers (names of variables, functions, constants, etc.)
- Operators and special characters (+ - * = > < etc)
- Separators (; , \n \t etc)

The analyzer records after that each symbol with its lexical category coded as a token in the table of symbols. Then, it creates a link between the lexical units in a line in the table and numbering the line.

Common errors may be reported during the previous steps, for example:

- A character not accepted by the language

- A string exceeds a certain size
- Failure to identify the lexical category
- Some errors may be also reported, such as the absence of an identifier declaration or a double declaration

II.3. Identification of lexical units

The most common approach to define the lexical units is to use regular expressions. A **regular expression (regex) i**s a sequence of characters (string) constructed using a set of rules that allows us to specify how the language is formed. The analyzer checks whether a set of characters can be recognized as a lexical unit using the different regular expressions defining the language.

To perform lexical analysis (in computational terms), regular expressions must be converted into finite automata. Kleene's theorem proves that a language is described by a regular expression if and only if it is recognized by a finite automaton. In fact, the core tool used in lexical analysis is the deterministic finite automaton (DFA), which is easy to implement and enables efficient recognition of lexical units. However, constructing a DFA directly can be challenging at times. As a result, a non-deterministic finite automaton (NFA) is commonly generated first. NFAs are simpler to create, and there is a straightforward algorithm to convert an NFA into a DFA.

II.4. Designing a lexical analyzer

Consider a regular language L. The main steps to design a lexical analyzer are:

- Step 1: Define lexical units using regular expressions
- Step 2: Convert each regular expression into a non-deterministic finite state automaton (NFA)
- Step 3: Construct the global NFA that recognizes the language by associating the NFA from Step 2 (we can add an initial state with an arc labeled $\in$)
- Step 4: Transform the NFA into a Deterministic Finite State Automaton (DFA)
- Step 5: Minimize the DFA
- Step 6: Implement the minimized DFA

We explain in the next sessions how each step of the lexical analyzer can be performed and/or implemented.

II.5. Define a language using regular expressions

Lexical units can be described using regular expressions which are a sequence of characters that match particular defined patterns. A language that is defined by regular expressions can be a **formal language**.

A language is formed using **alphabets or a vocabulary** $\sum$ that are the set of characters or symbols or alphabet that are used to form the language words or strings. A word must have a finite sequence of symbols and be defined by its length. An empty word is denoted ε (or sometimes &) and has length 0. Let the following vocabulary $\sum$ = {a, b,c}, the most common regular expression patterns are shown in Table 1 as follows:

Expression	Description
∈	empty
.	Any character
ab	a followed by b, the word {ab}
a|b	a or b, the words {a,b}
[abc]	Equivalent to a|b|c
[a-z]	Equivalent to a|b|.....|z
a*	Words formed from 0 or more successions of a : {∈, a, aa, aaa, …}
a+	Words formed of 1 or more successions of a : {a, aa, aaa, …}
a?	0 or a, the words {∈, a}
^a (b|c)*	String starts with a, e.g. {ab, ac, abbb, abc, …}
a{3}	Occurrence of a : {aaa}
a{2,4}	Occurrence of a: {aa, aaa, aaaa}
a{3,}	Occurrence of a: {aaa, aaaa, aaaaa, … }
(b|c)a$	String with a as the terminating character {ba, ca}

Table 1. Syntax of regular expressions

Using the above patterns, we can define a different set of words or a language. Examples are provided in Table 2.

Regular expression r_i	Language $L(r_i)$
(a|b)(a|c)	aa, ac, ba, bc
[a-z][2a]	a2, aa, c2, ca, ….
(a|b)c+	ac, bc, acc, bcc, accc, bccc, …
(a|b)*	∈, a, b, aa, bb,ab, aab, aabbb, abb, …
a|c*	∈, a, c, cc, ccc, cccc, …

Table 2. Examples of regular expressions to define languages.

II.6. Recognizing a language through a finite state automaton

A regular language can be defined by regular expressions and recognized by a finite-state automaton (or finite automaton, or simply a state machine). It is a directed graph labeled with a set of language alphabets and has an initial state and acceptance (final) states.

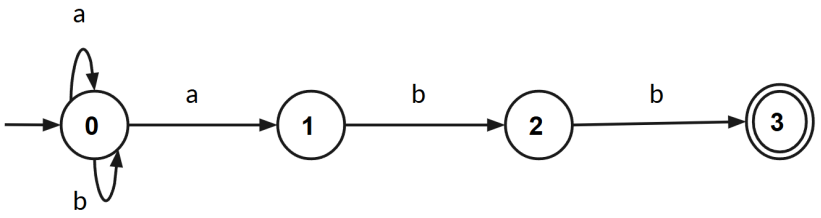
There are two forms of automatons that can be used: non-deterministic finite state automaton and deterministic finite state automaton.

Non-deterministic finite state automaton (NFA) is a mathematical model defined by:

- A set of **states** S
- A set of alphabets or **vocabulary** ∑
- A **starting state** s0
- A set of **final states** or accepting states
- A **transition function** T(state, symbol) which corresponds to each pair (state, symbol), a set of states

Example: **r= (a|b)*abb;**

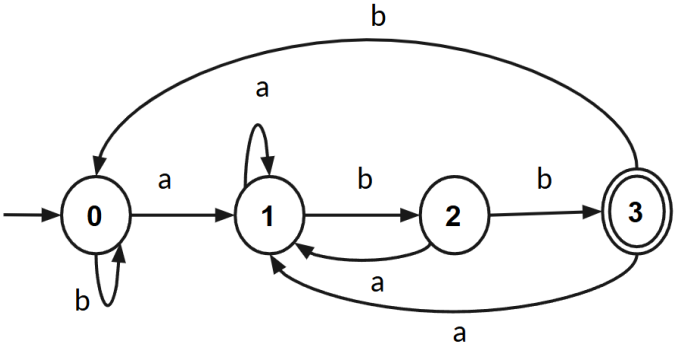
E={0,1,2,3}; ∑={a,b}, e0={0}; F={3}; {0a0, 0b0, 0a1, 1b2, 2b3}



Deterministic finite state automaton (DFA) is a particular case of an NFA that respects the following rules:

- Have only one initial state and can have one or more final states
- No state has a ∈-transition (symbol ∈ on an arc)
- For each state s and each input symbol a, there is at most one arc labeled a that leaves s

Example: for the same previous regular expression **r= (a|b)*abb,** the corresponding DFA is:

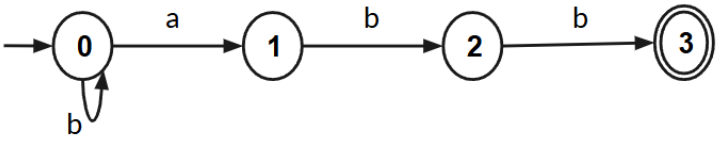


Finite state machines are used to implement a lexical analyzer with the help of a transition table where the lines represent the states and the columns represent the alphabets of the language. For example the previous NFA can be implemented using the following table:

States	**Alphabets**	
	a	b
0	0, 1	0
1	-	2
2	-	3
3	-	-

The process of recognizing a lexeme involves traversing the automaton starting from the initial state and reaching a final state. For each character in the input stream we progress in the table TRANS[currentState][CurrentCharacter]. However, using an NFA, ambiguities can be found. For instance, in state 0 there is a confusion about the next state when the current alphabet is “a”. This is why it is important to convert the NFA to DFA to limit ambiguities when recognizing the lexical units of a given source code.

For example, the following DFA can be implemented using the following table:



States	**Input (alphabet)**	
	a	b
0	1	0
1	-	2
2	-	3
3	-	-

II.7. Convert a regular expression into NFA

There are several methods to transform a regular expression into a non-deterministic finite automaton, including for example the method of Ken Thompson and the method of Victor Glushkov. Below, we will detail Thompson's method which is the simplest to follow and implement:

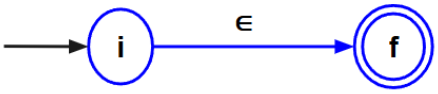
Algorithm: Thompson

Inputs: a Regex r defined on the alphabet ∑

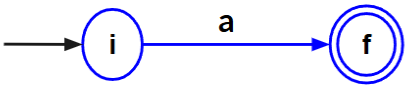
Results: un NFA that recognizes L(r)

Method : We decompose r into subexpressions and then using rules (1) and (2), we construct NFAs for each of the symbols in r. If the symbol appears multiple times, a separate AFN is constructed for each occurrence

Rule (1): For ∈, construct the following NFA, such that i is a new starting state and f a new final state



Rule (2): For each a ∈ ∑, construct the following NFA, such that i is a new starting state and f a new final state



i: new initial state
f: new final state

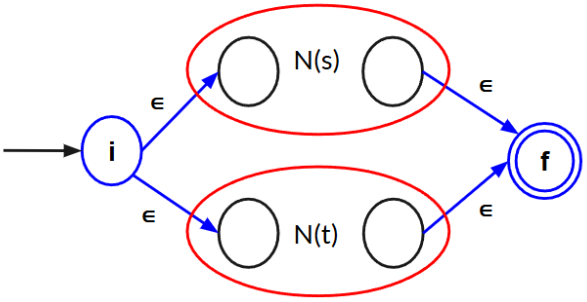
Then, following the syntactic structure of r, we recursively combine these NFAs using rule (3) until we obtain the NFA for the complete expression.

Each intermediate NFA must comply with the following two rules:

1. There is one and only one final state.
2. No arc enters the initial state and no arc leaves the final state.

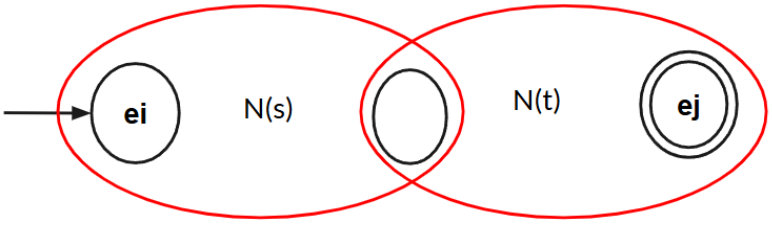
Rule (3): Let N(s) and N(t) the NFAs obtained from the regular expressions s and t respectively.

a) For the regular expression **s|t**, the equivalent NFA N(s|t) is:



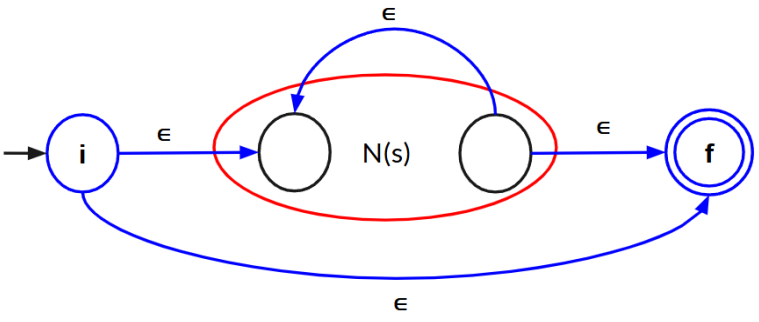
L(s) ∪ L(t)
i: new initial state
f: new final state

b) For the regular expression **st**, the equivalent NFA N(st) is:



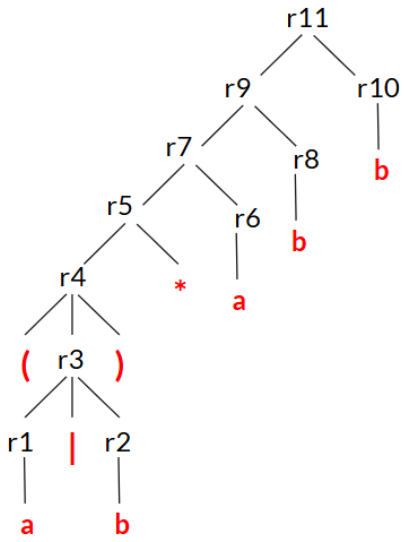
The initial state of N(s) becomes the initial state of N(st) and the final state of N(t) becomes the final state of N(st)

c) For the regular expression **s***, the equivalent NFA N(s*) is:



i: new initial state
f: new final state

Example : r= (a|b)* abb. A syntax tree of r for arithmetic expressions:



The obtained NFA using the Thompson method is shown in Figure 5.

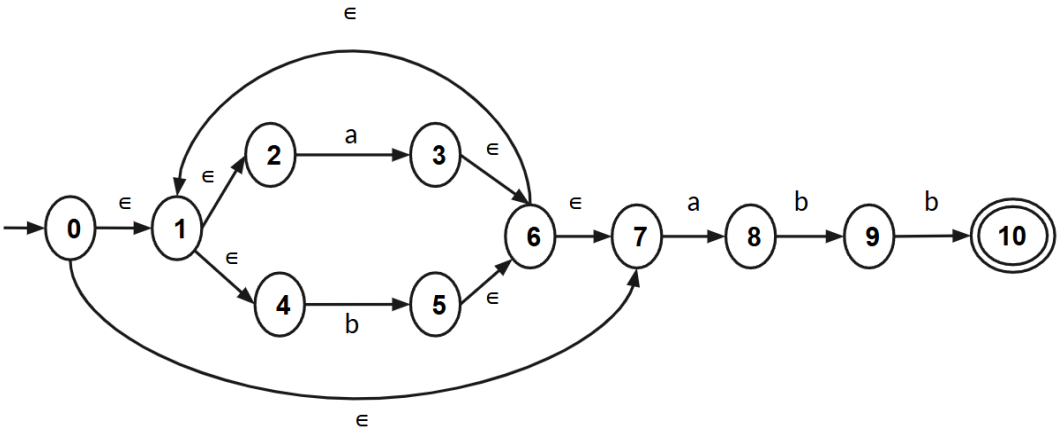


Figure 5. An example of the NFA obtained using Thompson method

II. 8. Transform the NFA into a DFA

Several methods can be suggested to obtain a deterministic finite state automaton from a non-deterministic one. Among the common methods that can be used when the NFA contains ϵ transitions is ϵ-closure method.

Algorithm

Inputs: NFA F recognizing the language L

Result: DFA D that recognizes the same language L

Method: Construct a Dtran transition table for D.

Each state of the D is a set of states of F

1. The initial state A of the D can be obtained through ∈-closure(s), where s is the starting state of the NFA
2. Calculate the ∈-closure of transitions of A for each symbol of the alphabet. If new states (different from A) are obtained, then go to Step 3.
3. Calculate the ∈-closure of transitions of the new states obtained for each symbol of the alphabet. Repeat Step 3 until no new states can be generated.

Functions: Let e be a state of F and T a set of states of the F:

- ∈-closure(e) : a set of states of F accessible from state e of F only by ∈-transitions
- ∈-closure(T): set of states of F accessible from a state e ∈ T of F by ∈-transitions
- Transit(T,a): set of states of F to which there is a transition on a (a ∈ ∑ of L) from e (e ∈ T)

Example:

Let transform the NFA of Figure 5. into a DFA. Next, we will compute is transition matrix Dtran using ∈-closure method.

(1) **∈-closure(0)={0,1,2,4,7}= A** ; ∑= {a,b}

(A)

- ∈-closure(Tansit(A,a))=∈-closure(Transit({0,1,2,4,7},a)) = ∈-closure({3,8})={1,2,3,4,6,7,8}=B
- ∈-closure(Tansiter(A,b))= ∈-closure({5})={1,2,4,5,6,7}=C
- Dtran[A, a]= B; Dtran[A, a]= C

(B)

- ∈-closure(Tansit(B,a))=∈-closure(Transit({1,2,3,4,6,7,8},a))= ∈-closure({3,8})=B
- ∈-closure(Tansit(B,b))= ∈-closure({5,9})={1,2,4,5,6,7,9}=D
- Dtran[B, a]= B, Dtran[B, b]= D

(C)

- ∈-closure(Tansit(C,a))=∈-closure(Transit({1,2,4,5,6,7},a)) = ∈-closure({3,8})=B
- ∈-closure(Tansit(C,b))= ∈-closure({5})=C
- Dtran[C, a]= B, Dtran[C, b]= C

(D)

- ∈-closure(Tansit(D,a))=∈-closure(Transit({1,2,4,5,6,7,9},a)) = ∈-closure({3,8})=B
- ∈-closure(Tansit(D,b))= ∈-closure({5,10})={1,2,4,5,6,7,10}=E
- Dtran[D, a]= B, Dtran[D, b]= E

(E)

- ϵ-closure(Tansit(E,a))=ϵ-closure(Transit({1,2,4,5,6,7,10},a)) = B
- ϵ-closure(Tansit(E,b))= C
- Dtran[E, a]= B, Dtran[E, b]= C

The state A is the initial state of the DFA and E is its only final state

- A= {0,1,2,4,7}
- B={1,2,3,4,6,7,8}
- C={1,2,4,5,6,7}
- D={1,2,4,5,6,7,9}
- E={1,2,4,5,6,7,10}

The transition table of the DFA Dtran is:

	a	**b**
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

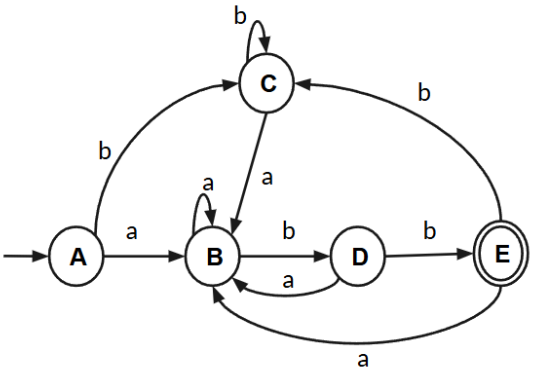


Figure 6. The obtained DFA from the NFA of Figure 5 using ϵ-closure method

II. 9. Minimize the DFA

Constructing a deterministic finite automaton (DFA) intuitively or by transforming a regular expression into a DFA does not always result in a DFA with the minimum number of states. Therefore, it is important to apply a minimization algorithm to reduce the number of states, ensuring that the transition table implemented on the machine is as compact as possible. Various

algorithms have been developed for this purpose, with those by John Hopcroft[1] and Edward Moore[2] being among the most well-known. In the following, we present a straightforward intuitive method for DFA minimization based on a simple class separation algorithm.

Algorithm

Inputs: A finite state automaton (DFA)

Result: A minimized DFA

Method

Step 1 Eliminate any inaccessible states and build the following table :

	State1	State2	…	Staten
Init	C1	C2	…	Cx
Alphabet 1	Ci	…	…	…
Alphabet m	-	Cj	…	…
Result	Ce	Ck	…	…

a. **In init** : create two classes: one that contains the acceptance states and the second that contains the remaining states
b. Complete the table by transitions
c. **Result i:**
   - If we find that the states have the same transitions, we keep them in the same class
   - Otherwise, we must separate them by creating new classes

Step 2: Repeat the remaining classes to check if they can be separated (ignore the previously separated state classes).

Repeat the same procedure as Step 1 (b and c) to calculate Result i+1.

Stop the procedure when Result i= Result i+1.

[1] John Hopcroft, An n log n algorithm for minimizing states in a finite automaton, Theory of machines and computations, 1971, New York: Academic Press, pp. 189–196.
[2] Edward F. Moore, Gedanken-experiments on sequential machines, Automata studies, Annals of mathematics studies, Princeton University Press, 1956, pp. 129-153

	S1	S2	…	Sn
Init	C1	C2	…	Cx
Alphabet 1	Ci	…	…	…
Alphabet m	-	Cj	…	…
Result i	Ce	Ck	…	…
Alphabet 1	Ci'	…	…	…
Alphabet m	-	Cj'	…	…
Result i+1	Ce'	Ck'	…	…

Example: Let consider the DFA of Figure 6 to minimize.

(A, B; C, D): class I and (E) : class II

	A	**B**	**C**	**D**	**E**
Init	I	I	I	I	II
a	I	I	I	I	I
b	I	I	II	I	I
Result 1	I	I	III	I	II

Then we examine whether we can separate the class I (A, B, D)

	A	**B**	**C**	**D**	**E**
Init	I	I	III	I	II
a	I	III	III	III	III
b	I	I	II	I	I
Result 2	I	iV	III	iV	II

Then we examine whether we can separate class IV (B, D)

	A	B	C	D	E
Init	I	IV	III	IV	II
a	IV	III	III	III	III
b	IV	IV	II	IV	IV
Result 2	I	IV	III	IV	II

Result 2= Result 1. The obtained DFA minimized is shown in Figure 7.

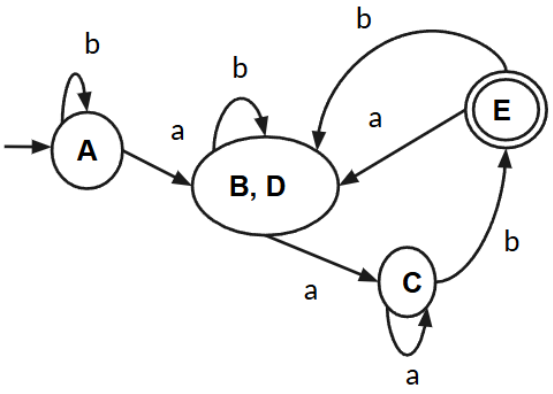
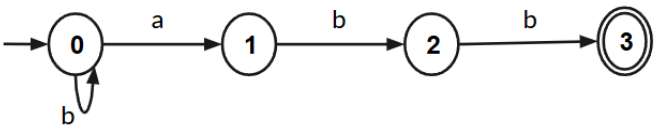


Figure 7. The minimized DFA from the example of Figure 6

II. 10. Implementation of the minimized DFA using an evolutionary programming language

As introduced in Section II.6, the DFA of the lexical analysis can be implemented using its transition matrix Dtran. A transition table can be used to implement the DFA. For example the lines represent the states of the DFA and the columns the alphabets as follows:



States	Input (alphabet)	
	a	**b**
0	1	0
1	-	2
2	-	3
3	-	-

Dtran of DFA

Once the DFA of your language is calculated and minimized, the next step is to implement it and proceed to lexical analysis. The lexical analyzer must be able to recognize **a sequence of lexemes.** Lexeme recognition is performed by navigating the automaton starting with the initial alphabet of the word to arrive at its final alphabet. An example of DFA implementation in C language is:

```
#include <stdio.h>
#define NBSTATES 4

int TRANS[NBSTATES ][256]; /* transition table */
int FINAL[NBSTATES ]; /* final states, final :1 not final : 0 */

/* recognize the next word that ends with $ and return 0 if not
recognized, 1 otherwise */
int accept(){
    int state=0; /*  initial state */
    int c; /* current character */
    while ((c=getchar())!='$') /* it not the end of the word */
        if (TRANS[state][c]!=-1) /* transition exist*/
            state=TRANS[state][c]; /* Advance in the automaton */
        else
            return 0; /* otherwise non recognition of the word */
    return FINAL[state]; /* word recognized if we reach a final state
*/
}

int main(){

  /* build the DFA */
    int i;int j;
    for (i=0;i<NBSTATES;i++){
        for(j=0;j<256;j++)
            TRANS[i][j]=-1;
           /* init transition matrix with empty values*/
            FINAL[i]=0; /* init final states array*/
    }

 /* Transitions of our DFA */
    TRANS[0]['a']=1; TRANS[0]['b']=0;
    TRANS[1]['a']=-1;TRANS[1]['b']=2;
    TRANS[2]['a']=-1;TRANS[2]['b']=3;
    TRANS[3]['a']=-1;TRANS[3]['b']=-1;

  /* final states */
    FINAL[3]=1;

    printf("Enter a word of L defined by:  (a|b)*abb followed by $ ");

    if (accept())  printf("\n word recognized !\n");
    else printf("\n Unrecognized word !\n");

    return 0;}
```

In this example, the program prints a message for each lexeme recognized. To use the result of lexical analysis for other compilation steps, it is important to improve the output of the lexical analyzer. In addition to implementing the transition matrix, you have to define a table to associate **tokens to final states**. In fact, instead of returning a message or boolean when a word is recognized, we return a number that represents the token that provides information on the lexical category of the lexeme.

Some exceptions have to be considered by the lexical analyzer program. For instance, when a word in the language is the **prefix** of another (e.g., for, foreach). In this cas, the **largest possible word rule** can be applied. It is therefore important to read the next character before returning the token to check if a larger word is recognizable. In addition, If you advance in the DFA and you reach a non-terminal state without being able to advance, you must move backward in the automaton to be able to return to the last final state you navigated (e.g. for, foreach, forx).

II. 11. Implementation of the lexical analyzer using Lex

Lex is a common tool for generating a lexical analyzer. It allows to generate a lexical analysis program from definitions of lexeme models through regular expressions and actions to be executed when recognizing these models. We can find different available versions: lex, flex, pclex, …. that can be used to generate lexical analyzers in different languages such as C, C++, python, and ADA, The generated program contains the transition table of the minimal deterministic finite automaton and the lexical analysis function that checks whether inputs belong to the language by navigating the DFA. The generated analyzer can be compiled and executed by the corresponding programming language compiler.

For instance, in the Flex version for C language, we introduce the regular expression defining the language in a file with the ".l" extension. Then by running flex command line, the analyzer in C language is generated (lex.yy.c) as shown in Figure 8. Then we compile the generated file (using C compiler or gcc) to obtain and run our analyzer.

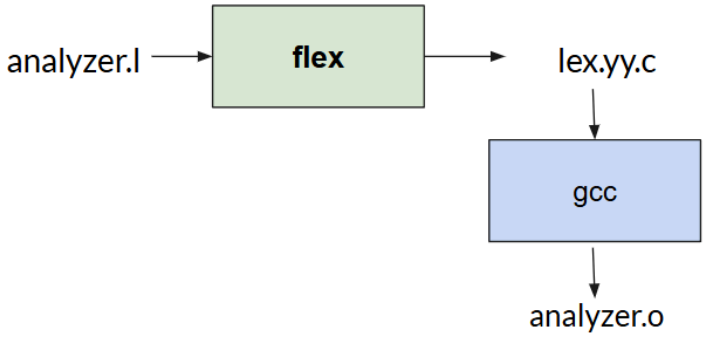


Figure 8. The process to generate a lexical analyzer using Flex.

A flex file structure consists of three parts :

- An optional part of **definitions** that contains inclusion directives and global C definitions (variables, types, . . .) between %{ and %} ;
- A mandatory part that contains lex **rules** delimited by %%. It allows associating regular

expressions that define lexemes patterns to actions of the analyzer, such as through C instructions.

- An optional part of **C functions** defined by the developer of the compiler delimited by %% (including a main function).

The rules can have the following structures:

```
regex ; /* do nothing if regex*/

regex {

 // c or c++ /* if regex do */

}

Regex expC; /* if regex do expC*/
```

Here an example of Flex file that recognizes three patterns of words and return the ASCII code in the case of unrecognized words:

```
/* Zone of definitions (optional) */
/* Zone of rules after %% (mandatory)  */
%%
a {return 300; /*a token is returned */ }
ab+c {return 301;}
bd { /* do nothing */ }
 .|\n { return yytext[0]; /*  the ASCII code for everything else */ }
%%

/* zone of C functions (optional) */
int main(){
 int j;
 char *invite="Enter one or more words following this pattern
                a(b+c)?|bd; then EOF (CTRL-D) : ";
 printf(invite);
 while ((j=yylex())!=0)
      printf("\n Token : %i; of the lexeme %s \n %s", j,
      yytext, invite);
return 0;
}
```

Here are some examples of Lex functions and variables:

- **yylex():** the main Lex function that sequentially analyzes an input file (or stream), use the AFD for analysis, and returns 0 when it encounters the end of file or stops the process in the case of errors.
- **yytext** = variable containing the current recognized character string,
- **yyleng** = length of the string in yytext. The last recognized character is at yytext[yyleng-1].
- **yyless(k)** = a function that removes the last characters from yytext, making its length k, It also moves back the input file reading pointer by (yyleng - k) positions.

- **input()** returns the next input character. The standard input is noted yyin and the standard output is noted yyout.
- **output(c)** outputs the character c.
- **unput(c)** puts the character c back into the input stream.
- **yywrap()** = a function that is called when yylex() encounters the end of file to return 0.
- **yylva**l : a variable used to communicate LEX with syntax analyzer (YACC).

Flex also suggest special action options:

- **ECHO** prints the recognized input to the standard output.
- **BEGIN** changes the analysis state. This directive will be explained later with examples using analysis states.
- **REJECT** causes the next alternative to be tried and is generally used to recognize nested strings.

II. 12. Exercises

Exercise 1.

Give the languages defined by the following regular expressions (regex):

- r1= (a|c) ba
- r2= $(a|b)^*c$
- r3= $(a|b)^+(a|b)^+c?$
- r4= $(a|(b|c)^*c)^*$
- r5= $[a\text{-}z]b^+[0\text{-}9]$
- r6= b(ab|ba|c)b
- r7= (0|1)1(0|1)(0|1)
- r8= ab{2}[0-9]+
- r9= c{2,}ab

Solution

- **r1= (a|c)ba:**
  L(r1)={ab,cba}

- **r2=$(a|b)^*c$**
  L(r2)={c,ac,bc,abc,bac,aac,bbc,...}. strings over {a,b} ending in c

- **r3=$(a|b)^+(a|b)^+c?$**
  L(r3)={ab,abc, ba, bac, aab, abb, bba, abba, abbac, aba, bca,...} strings made of ‘a‘s and ‘b‘s with two non-empty parts, followed optionally by ‘c‘

- **r4=$(a|(b|c)^*c)^*$**
  L(r4)={ϵ,a,bc,bcc,ab,ac,abc,...} strings that are empty or consist of a combination of ‘a‘ and blocks of ‘b‘s and ‘c‘s ending in ‘c‘

- **r5=$[a\text{-}z]b^+[0\text{-}9]$**
  L(r5)={cb8, fb3, fbb3, fbbbb3, bb6,...} a letter followed by b+ followed by a digit

- **r6 = b(ab|ba|c)b**
  L(r6)={babb,bbab,bcb} start with b and then either ab, ba, or c, or start with b and are followed by zero or more cs.

- **r7 = (0|1)1(0|1)(0|1)**
  The language consists of strings that start with either 0 or 1, followed by 1, then any two bits (0 or 1). L(r7)={011,111,010,100}

- **r8= ab{2}[0-9]+** The language consists of strings that start with abb followed by one or more digits. L(r8)={abb1,abb112,abb0,abb1235}

- **r9= c{2,}ab** The language consists of strings that start with two or more c, and end with the substring "ab". L(r9)={ccab,cccab,ccccab,ccccccab,…}

Exercise 2.

Determine all strings of maximum length 4 that belong to the language denoted by each of the following regular expressions

- r1=$(b|ba)^*$
- r2=$a(b^*|c)$
- r3=$(a|b)^*abb$
- r4=a^*cb^+
- r5=$a^*(b|c)d$

Exercise 3.

Provide regular expressions that denote each of the following languages:

- L(r1): the set of strings that start with the letter “a” followed optionally by a sequence of letters and/or digits, e.g. L(r1) = {a, acba, ab, a3, a32, …}

- L(r2): the set of strings that represent passwords consisting of a sequence of 6 letters followed by a digit, e.g. L(r2) = {abcdef2, hgeeaa7, …}

- L(r3): the set of strings composed of lowercase or uppercase letters ending with the letter “c”, e.g. L(r3) = {hfc, AEc, RTEc, dgjcc, AKCCLc, …}

- L(r4): strings over {a, b} that end with ab, e.g. L(r5)={ab, aab, bab, bbab,...}

- L(r5): strings over {a, b, c} of exactly three characters e.g. L(r6)= {abc, aaa, bcb, ccc,…}

- L(r6): lowercase strings where first and last letters are the same without a finite length assumed.

- L(r7): Strings over lowercase letters and digits that start with a lowercase letter, end with a digit. The middle can be letters or digits.

Solution

- r1= a[a-zA-Z0-9]*
- r2=[a-zA-Z]{6}[0-9]
- r3=[a-zA-Z]*c
- r4=(a|b)∗ab
- r5=(a|b|c)(a|b|c)(a|b|c)
- This is not regular in general (requires memory), but if we limit string length, it's expressible. If we accept an approximation for short strings (e.g., length ≤ 4), we can write: r7≈ ∑ x ∈ [a−z] x [a−z] ∗x (i.e., union over all 26 letters, e.g. a[a-z]*a | b[a-z]*b | ...)
- r7=[a−z][a−zA-Z0−9]∗[0−9]

Exercise 4.

I. Write the regular expressions defining the following regular languages and provide their corresponding DFA:

1. Language of identifiers composed of a letter, optionally followed by letters and/or digits and ends with "_"
2. Language of integer numbers (in C)
3. Language of float numbers : a sequence of digits to the left and/or right of the decimal point and an optional integer exponent, a point is also considered as a float (examples: .; 2.; .25; 3.5; 3.2E-10; 2.29e2)
4. Language of keywords: {int, float, if, else}
5. Language of separators: space, \t, \n
6. Language of comments formed by lowercase letters delimited by /* and */

II. Provide the corresponding DFA for each of the previous languages
III. Provide a finite state automaton that recognizes the union of the above languages.

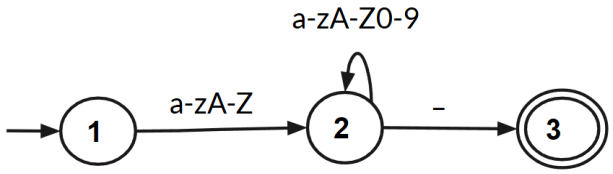
Solution

I/ Regex:

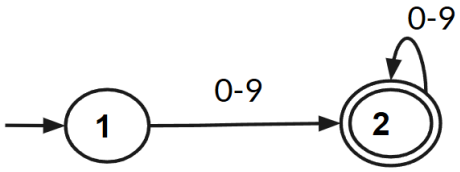
1. r1=[a-zA-Z] ([a-zA-Z0-9])* _
2. r2=[0-9] ([0-9])*
3. r3=[0-9]* "." (([0-9])* | (([0-9])* (E|e) (+|-|ε) ([0-9])+)) OR r3=[0-9]* "." (([0-9])* | (([0-9])* (E|e) (-)? ([0-9])+))
4. r4= int | float | if | else
5. r5= (" " | \t | \n)+
6. r6= "/*" " " ([a-z])* "*/"

II. DFA

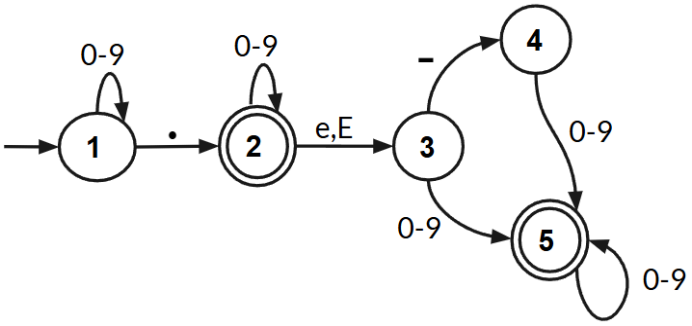
1/


a-zA-Z0-9
a-zA-Z
_
1
2
3

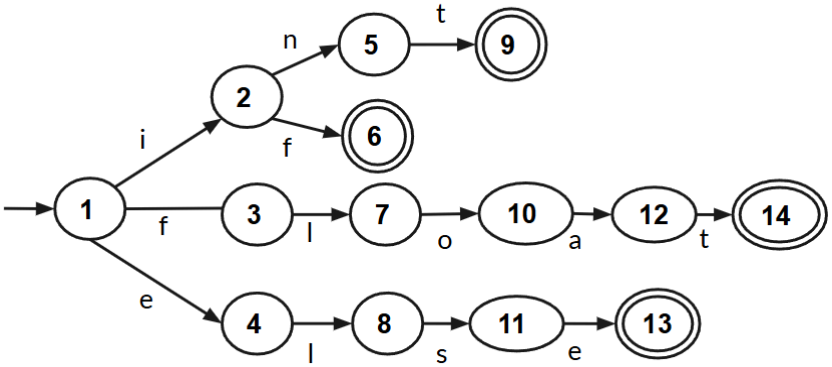
2/


0-9
0-9
1
2

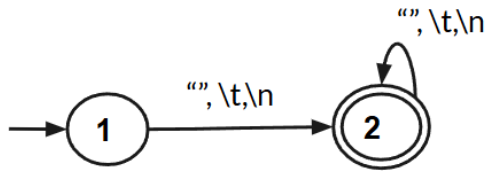
3/


0-9
0-9
.
e,E
_
4
0-9
0-9
5
0-9
1
2
3

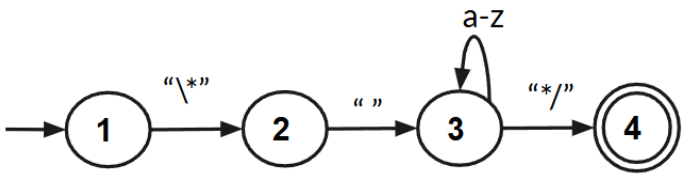
4/


n
t
5
9
2
i
f
6
1
f
3
l
7
o
10
a
12
t
14
e
4
l
8
s
11
e
13

5/



6/



III. General DFA:

The union of all the previous DFA while ensuring that the resulting automaton is deterministic. The intuitive method may be difficult in some cases. Among the solutions is that union can be made through epsilon transitions, then we apply ϵ-closure method to compute the general DFA.

Exercise 5.

1. What is the role of a lexical analyzer?
2. Name the following parts and elements of this example of Flex code:

```
—---------- (Part 1)----------
%{
#include <stdio.h>
#include <stdlib.h>
%}
—---------- (Part 2)----------
%%
a    { printf("word a\n"); }
a|b  { printf("word a or b\n"); }
.    { printf("error\n"); }
%%
—---------- (Part 3)----------
int yywrap() {
   return 1;
}
```

```
int main() {
   printf("Enter a word:\n");
   yylex();
   return EXIT_SUCCESS;
}
```

1. *Part 1:*.....
2. *Part 2:*.....
3. *Part 3:*.....
4. a, a|b, and . are: .....
5. .: means: .....
6. Instead of { printf("word a\n"); } we can return a .....
7. yywrap() is .....
8. yylex() is .....

Exercise 6.

Write the Flex program to recognize to recognize numbers (integers), parentheses, and identifiers (variable names) along with the arithmetic operators (+, -, *, /) .
Here an example of the program running:

```
Enter an arithmetic expression
(a + b) * 3 - (x / y) + 2
Identifier: a
Addition Operator: +
Identifier: b
Parentheses: (a + b)

Multiplication Operator: *
Number: 3
Subtraction Operator: -
Parentheses: (x / y)

Division Operator: /
Identifier: x
Identifier: y
Addition Operator: +
Number: 2
```

Solution

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%%
"+"            { printf("Addition Operator: +\n"); }
```

```
"-"            { printf("Subtraction Operator: -\n"); }
"*"            { printf("Multiplication Operator: *\n"); }
"/"            { printf("Division Operator: /\n"); }

[0-9]+         { printf("Number: %s\n", yytext); }
                /* Matches integers */
"("            { printf("Left Parenthesis: (\n"); }
")"            { printf("Right Parenthesis: )\n"); }
[a-zA-Z_][a-zA-Z0-9_]*  { printf("Identifier: %s\n", yytext); }
[ \t\n]+       { /* Ignore whitespace */ }
%%
int main(void) {
    printf("Enter an arithmetic expression:\n");
    yylex();  /* Start lexical analysis */
    return 0;
}
```

Exercise 7.

Provide the Flex program to recognize the following lexical units:

1. Arithmetic operators (+, -, *, /) and display "operator"
2. Words consisting of a sequence of a: a, aa, aaa, aaaa, … and display "succession of a"
3. Words consisting of a sequence of cbcb:, cbcbccb, cbcbcbcb, … and display "succession of cb"
4. Words of length 4: ab32, 4rTE, r3Te, AAEs, … and display "word of length 4"
5. Display an error message if none of the preceding words are detected.

Exercise 8.

Give the Flex program to define the regular expression that recognizes the following words: b Ac ART c23 r47c2 aMT_plr_9; but does not recognize the following words: zbT_ _pl ht__h.. Use CHAR and DIGIT definitions to describe your regular expression.

Exercise 9.

I/ Provide the Flex program to define the regular expression that recognizes variable identifiers, comments, and strings in a particular language.

- An integer and a real number and display the type of number detected
- An identifier that begins with a letter (or an underscore) followed by letters, numbers, or underscores.
- Comments that can be single-line comments (with //) or multi-line comments (between /* and */).
- Strings that are delimited by double quotes (") and can contain escaped characters.
- Displays "unrecognized word" if it detects other words.

Test your program with a C code containing the previous lexical units.

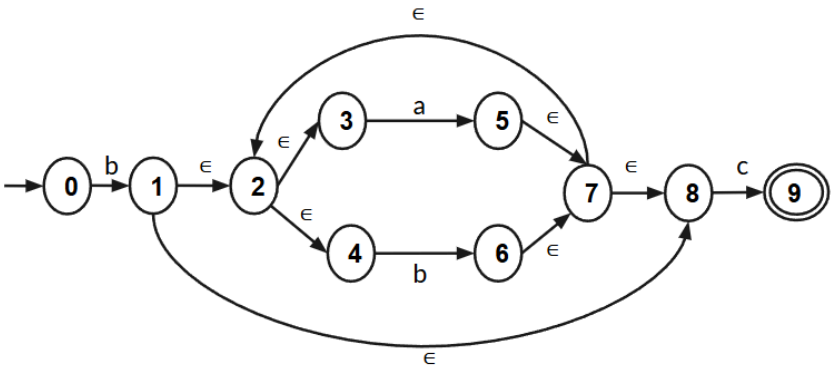
Exercise 10.

Give a finite state automaton equivalent to each of the following regular expressions using Thompson's algorithm:

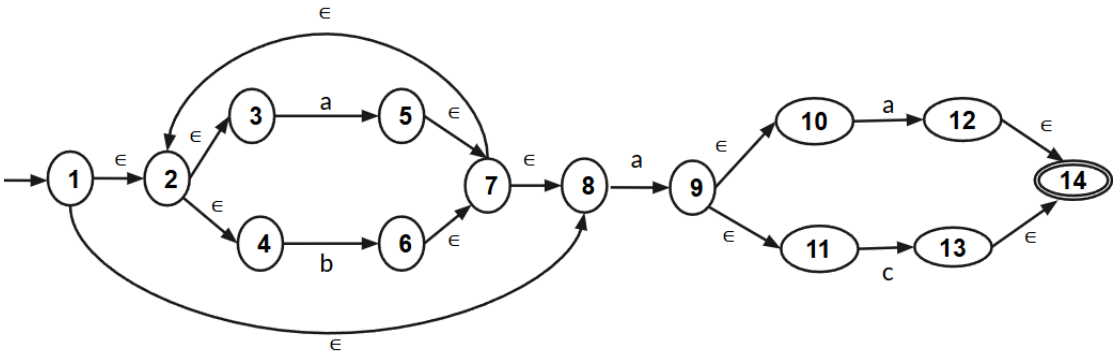
1. $b(a|b)^*c$
2. $(a|b)^*a(a|c)$
3. $(a^*ba(aa|bb))^*$

Solution

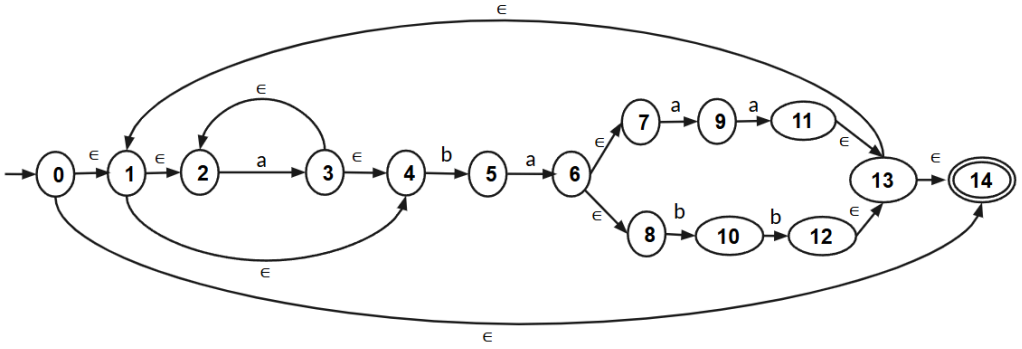
1. $b(a|b)^*c$



2. $(a|b)^*a(a|c)$



3. $(a^*ba(aa|bb))^*$



Exercise 11.

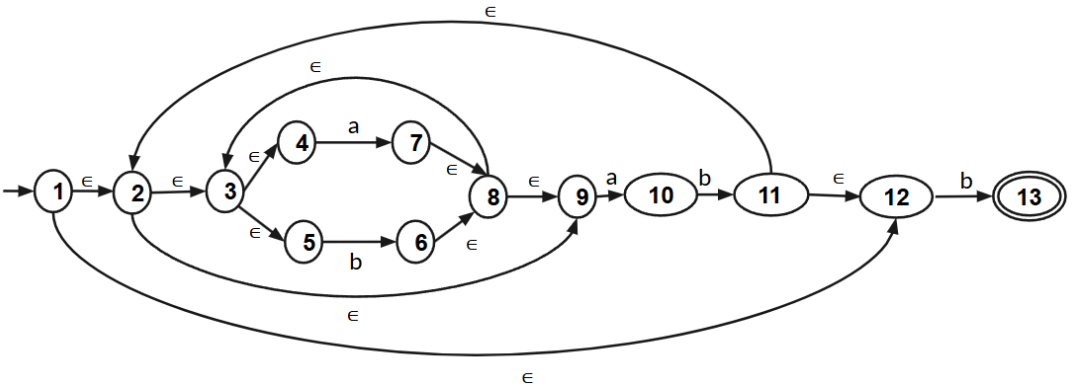
Given the following regular expression: r = ((b|a)*ab)*b

1. Construct a finite state automaton equivalent to **r** using Thompson's algorithm.
2. Is this automaton deterministic? Why or why not? If it is not deterministic, determinize it using ϵ-closure method.

Solution

1. Construct a finite state automaton equivalent to r using Thompson's algorithm.

   r = ((b|a)*ab)*b



2. Is this automaton deterministic? Why or why not? If it is not deterministic, determinize it using ϵ-closure method.

Σ= {a,b}
ε-closure(1)={1, 2, 3, 4, 5, 9, 12}= A

State A:
ε-closure(T(A,a))= ε-closure ({7, 10})= {7, 8, 9, 3, 4, 5, 10} =B
ε-closure(T(A,b))= ε-closure ({6, 13})= {6, 8, 9, 3, 4, 5, 13} =C

State B:
ε-closure(T(B,a))= ε-closure({7, 10})=B
ε-closure(T(B,b))= ε-closure({11, 6})= {6, 8, 9, 3, 4, 5, 11, 2, 12} =D

State C:
ε-closure(T(C,a))= ε-closure({7, 10})=B
ε-closure(T(C,b))= ε-closure({6})= {6, 8, 9, 3, 4, 5} =E

State D:
ε-closure(T(D,a))= ε-closure({7, 10})=B
ε-closure(T(D,b))= ε-closure({6, 13})= C

State E:
ε-closure(T(D,a))= ε-closure({7, 10})=B

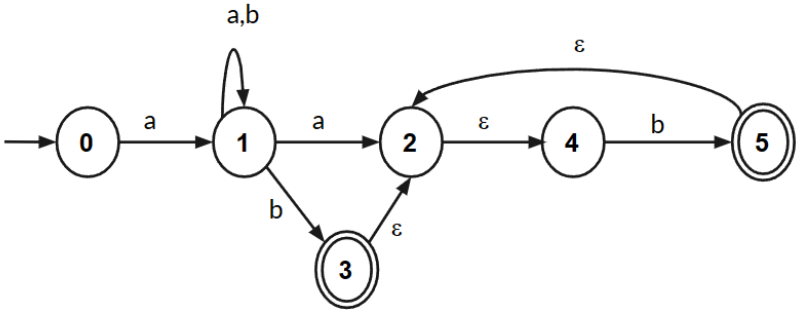
ε-closure(T(D,b))= E
Dtran: (transition matrix or our DFA)

	a	b
A	B	C
B	B	D
C	B	E
D	B	C
E	B	E

init =A; final= {C}

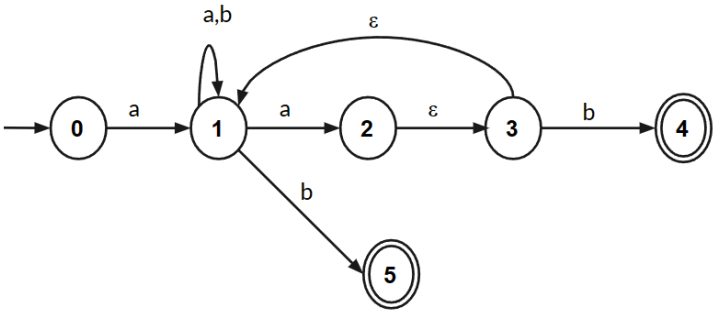
Exercise 12.

Given the following finite state automaton B; Provide the equivalent DFA for B using ϵ-closure method.



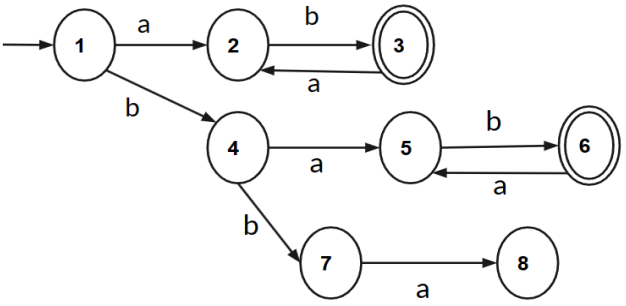
Exercise 13.

Given the following finite state automaton. Provide its equivalent DFA using ϵ-closure method.



Exercise 14.

Minimize the following finite state automaton



Exercise 15.

Minimize the DFA of the Exercise 11.

Solution

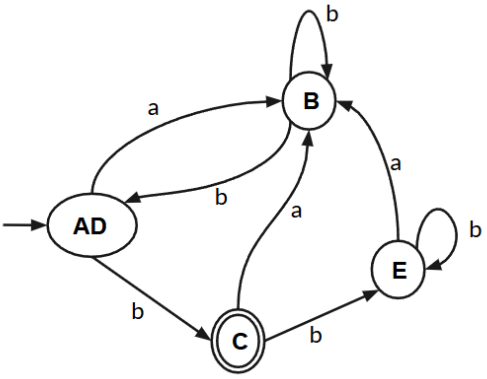
1) Remove inaccessibles states -> none

2/ Form classes: II: (C) I: (A, B, D, E)

	A	B	C	D	E
init	**I**	**I**	**II**	**I**	**I**
a	**I**	**I**	I	**I**	**I**
b	**II**	**I**	I	**II**	**I**
result 1	**I**	**III**	**II**	**I**	**III**
a	**III**	**III**		**III**	**III**
b	**II**	**I**		**II**	**III**
result 2	**I**	**III**	**II**	**I**	**IV**
a	**III**			**III**	
b	**II**			**II**	
result 3	**I**			**I**	

Result 1: (A,D) (B,E) (C)
Result 2: (A,D) (B) (E) (C)

result 2= result 3 -> end of algorithm
initial state: (AD); Other states : (B) (E)
Final state C



CHAPTER III. Syntax Analysis Using Top-down Methods

A syntax analyzer uses the grammar definition of a language to check whether the grouping of lexemes is ordered. It produces as a result an intermediate syntactic representation of the code (often using a tree data structure). The communication between lexical and syntax analyzers can be sequential or parallel (see Figure 4. Chapter II). In the sequential method, the compiler begins with the lexical analysis of the entire code then it performs the syntax analysis. However, this method can be time-consuming, especially when lexical errors occur early in the source code and the code spans multiple lines. The parallel method can therefore optimize the compilation process as lexical analysis operates only at the request of the syntax analyzer during code analysis.

III. 1. Defining a grammar for syntax analysis

In order to construct the syntax analyzer, we need first to define the grammar that generates the language. Starting from the definition of lexemes in lexical analysis using regular expressions, the next step is to define the language (regular) through its grammar.
Let the **formal grammar $G=\{V_T, V_N, S, R\}$** used to define a language

- $\mathbf{V_T}$: a finite set of **terminal symbols** (language alphabets)
- $\mathbf{V_N}$: a finite set of **non-terminal symbols**
- **S**: a **start symbol** that belongs to V_T
- **R**: a set of **production rules** that are pairs of a non-terminal that produces a set of terminal symbols and/or non-terminal symbols

Example: G= ($\mathbf{V_T}$={a,b}, $\mathbf{V_N}$ ={S}, S, R)
R: $S \rightarrow aSb$
$S \rightarrow \in$
G generates the language that consists of words following this pattern: a^nb^n

III.1.1 Types of grammars (chomsky hierarchy)

Grammars can have different types according to the restrictions of defining rules and words generated. Following Chomsky hierarchy, we can find generally four types of grammars:

- **Type 0:** Unrestricted Grammars. No restrictions are imposed on the rules.The generated languages are recursively enumerable and recognizable by the Turing machine.
- **Type 1:** Context-Sensitive Grammars. The production rules $r \in R$ must have the following form: $r = \alpha X\beta \rightarrow \alpha m\beta$, $\alpha, \beta \in V*$; $X \in V_N$; $m \in V+$. m cannot be ε, and α and β generally represent the context.
- **Type 2:** Context-Free Grammars. Each production rule $r \in R$ has the form: $r = X \rightarrow \alpha$, $\alpha \in V*$; $X \in V_N$. These grammars are generally recognized by Pushdown automaton
- **Type 3:** Regular grammar . Productions are of the form : $A \rightarrow aB$ or $A \rightarrow a$, where A and $B \in V_N$ and $a \in V_T$. These grammars are recognized by a Finite automaton.

III.1.2 Derivation tree and ambiguous grammar

The syntax analyzer verifies that the string can be generated by the language's grammar G. However, **G must be unambiguous,** i.e. one and only one production tree for a given string of the language.

G is **ambiguous** if and only if there are at least two distinct derivation trees for a string m. The language generated by a grammar G can be denoted by L(G) which is the set of derivations using the production rules.

There are three ways to perform a **derivation**:

- **Simple derivation:** from the starting symbol of the grammar, we try to replace the terminal and non-terminal symbols to reach the leaves of the tree in order to find a string that belongs to the language.
- **Left-most derivation**: This involves always replacing the leftmost non-terminal symbol.
- **Right-most derivation**: This involves always replacing the rightmost non-terminal symbol.

Example: Let the following grammar G = ({a, +, (,) }, {E, F}, E, R) where R is defined as follows:

E → E + F
E → F
F → (F) | a

Does the string w= a+a belong to the L(G) language?

- With Left-most derivation: E→E+F→F+F→a+F→a+a
- With Right-most derivation: E→E+F→E+a→F+a→a+a

G is therefore ambiguous because we found two distinct derivation trees for w.

III.1.3 Eliminating ambiguity from a grammar

We can resolve the ambiguity on the basis of the following two properties:

- **Operator priority:** The level at which the production rule is present in the tree denotes the priority of the operator used. In the analyzer tree, nodes at levels higher than or close to the root node will contain operators of lower priority.
- **Associativity:** If the same priority operators are in production, then associativity must be considered. If associativity is left-to-right, then we must induce left recursion in the production. The analyzer tree will also remain recursive and grow on the left side. +, -, *, / are left associative operators.

Example 1.

Let the following rules of a grammar G that generates the language {id, id-id, id-id-id, …} :

E → E - E | id, it

For w= id-id-id and 3-3-3, we have the same operator so we apply the associativity rule **by adding a new non-terminal.** The new production rules for the grammar are :

E → E – P | P
P → id

Example 2.

Let G be the grammar of arithmetic expressions with the following rules:

E→ E + E | E ∗ E | (E) | id

This grammar is ambiguous as we have: id+id*id We can eliminate ambiguity with the same principal as follows :

E → E + T | T
T → T ∗ F | F
F → (E) | id

III.1.4 Eliminating left recursion from a grammar

In some syntax analysis methods we also need to eliminate the left recursion of a grammar. A grammar G is **left recursive** if it contains a non-terminal A such that there exists a derivation **A** → **A** α, and α is any string. We can also have an indirect left recursion which appears after several derivations. Example: A → B C and B → A E therefore A⇒BC⇒AEC

We apply the following rule to eliminate left recursion:

If **A** → **A** α **|** β => A → β A'

A' → α A' | ε

Example: Let G defined by the following rules:

E → E + T | T
T → T ∗ F | F
F → (E) | id

G becomes:

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

III.1.5 Left factorization of a grammar

In some syntax analysis methods we also need to factorize the grammar. A grammar G is **left-factored** if the right-hand parts of two rules with the same left-hand part **do not have a common prefix**: A → **α** β_1 | **α** β_2 and **α=ε**

Factorization of a grammar rule consists of finding, for each non-terminal symbol A, the longest prefix α ≠ ε that is common to two or more rules with A as their left-hand side.

$A \rightarrow \boldsymbol{\alpha}\ \beta_1 \mid \boldsymbol{\alpha}\ \beta_2 \mid \ldots \mid \boldsymbol{\alpha}\beta_n \mid \gamma$ Such that γ represents all right parts $\not\asymp \alpha$,

The factorisation consists in replacing it by:

$A \rightarrow \alpha\ A' \mid \gamma$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Example. Let G defined by the following rules:

S → **nEtS** | **nEtS**eS | a

E → b

Left-factored G rules are :

S → nEtS**S'** | a

S' → eS | ε

E → b

III.2 Syntax analysis methods

There are three main categories of methods for parsing:

- **Universal methods:** These methods can theoretically parse any grammar, but are difficult to apply for industrial compilers.
- **Top-down methods:** These methods construct syntax trees from root to leaf. The grammar must be **unambiguous** and **non-left-recursive**. Examples of methods include Recursive top-down analysis, LL(1) and LL(k).
- **Bottom-up methods**: These methods construct trees from the leaves to the root. The grammar must be **unambiguous**. Examples of bottom-up analysis include SLR(1), LR(1), and LALR methods.

III.3 Implementing a syntax analyzer using YACC

Yacc ("Yet Another Compiler Compiler") is a parsing tool and language for syntax analysis using a Bottom-up method, especially LALR(1). The developer has only to define the grammar rules and actions can be associated with each. These actions are instructions from a programming language (C or C++) as well as specific actions from yacc.
There are many versions of yacc, including bison (Gnu).

The source code of a yacc file consists of 3 parts:

- C and yacc declarations between %{ and %}
- Grammar rules and associated actions delimited by %% at the beginning
- C Functions C delimited by %% at the beginning

Rule: is defined by a non-terminal symbol, the character ":", a sequence of symbols (terminal (tokens) or non-terminal) and action blocks {...}, terminated by a ";". Spaces, tabs, and newlines are only considered as separators. The rule must start at the beginning of the line and end with a ";".

Example : Let G be the grammar of Boolean expressions with the following rules:
E→ (E) | E I | I E | E & E | ! E | 0 | 1. The corresponding yacc file is:

```
%{
#include   <stdio.h>

int yylex(void);

void yyerror(char *s);
%}

%%
%%

expr  :     '(' expr ')' {}
|     expr '|' expr {}
|     expr '&' expr{}
|     '!' exp{}
| '0'{}
| '1'{};

%% /* C functions */

int yylex(void) {     /* lexical analyzer filtering out blanks */
      int c;
      while(((c=getchar())==' ') || (c=='\t'));
      return (c);
}

void yyerror(char *s) {
      /*  called  by  yyparse  when  a  syntax  error  occurs  */
      fprintf(stderr,"%s\n",s);
}

int main(void){
      if (!yyparse())
            /* call the analyzer generated by yacc */
            printf("\n Recognized Expression \n");
      else
            printf("\n Unrecognized expression \n");
      return 0;
}
```

Yacc is generally used with flex to perform both lexical and syntax analysis. The Yacc is responsible for calling the lexical analyzer (yylex() function) through the function yyparse() of Yacc (see Figure 9.).

Here an example of flex and yacc (bison) files:

//ex.l

```
%{
#include "ex.tab.h"
%}
%option noyywrap
%%
"*" return STAR;
a return A;
b return B;
[ \t\n] /* ignore */
. printf("Lexical error \n");
%%
```

//ex.y

```
%{
#include <stdio.h>
#include "ex.c"
void yyerror(char* s);
 %}

 %token A B STAR
 %start S

 %%
 S : STAR S | T ;
T : A
| B {printf("correct syntax \n");};
%%

void yyerror(char* s){
 printf("Syntax error \n");
}

void main() {
   printf("Enter your code:\n");
    yyparse();
         printf("end of parsing \n");
}
```

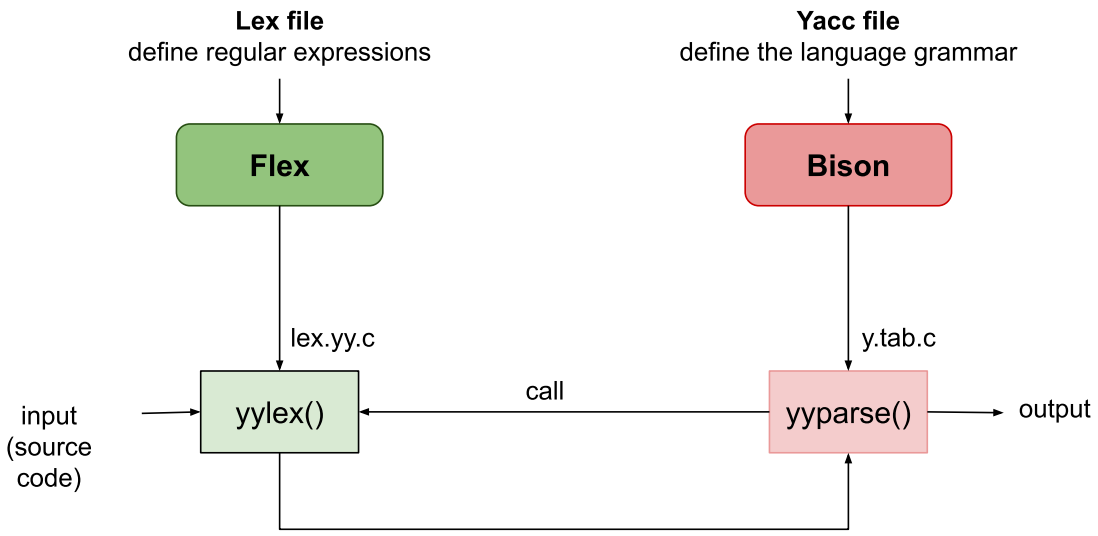


Figure 9. Flex and Bison internal functioning and communication

The Flex file does not contain a main() function since the result of the lexical analysis will be used directly by Bison (for syntax analysis). %option noyywrap is used to inform the compiler that the function yywrap() has not been defined.

Once the flex file is generated (through flex command line) it can be used by Yacc (bison line command). #include "ex0.tab.h" is a generated file that must be included in order to link the lexical and syntax analyzers. When using the syntax analyzer, it is the latter that calls Flex to detect the lexemes (tokens).

The two analyzers (files) are connected through the tokens (STAR, A, B) and yyerror is used to handle error messages. %start specifies the start symbol (one or more characters) while yyparse() launches the syntax analysis using the LALR method.

III.4 Top-down syntax analysis methods

Top-down syntax analysis methods are parsing techniques that begin analyzing the input (source code that contains lexeme sequence) from the start symbol of the grammar and attempt to derive the input string. These methods use grammar rules by deriving from the top (start symbol) to the bottom (terminal symbols). Two top-down methods can be used: Recursive top-down analysis method based on defining recursive functions for each non terminal symbol and LL(1) which is a predictive method (Left-to-right scanning of input, Leftmost derivation) that is based on pushdown automaton. Finally, top-down methods can be used only when the grammar is **unambiguous** and **not left-recursive.**

III.4.1 Recursive top-down analysis

This analysis method consists in writing a program where each **function** (often recursive) is used to define a **non-terminal symbol** of the grammar. These functions are called following the recognition of certain **tokens** in the input stream corresponding to the beginning of the right-hand parts of the production rules. The tokens therefore make it possible to **predict** the

production rule to be chosen.

Here an example of implementation of recursive to-down syntax analyzer for the grammar G defined by the following rules in C language:

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

```
#include<stdio.h>
#include<stdlib.h>
#include <ctype.h>

#define ADVANCE {
      token=getchar();
      Numcar++;
 }

#define ADVANCE_TEST(expected) {
      if (token==(expected))  ADVANCE /*without  ; for macros */
      else SYNTAX_ERROR
 }
#define SYNTAX_ERROR {
      printf("\unrecognized string ; syntax error at character
            number%d \n",numcar);
      exit(1);
  }


int token;         /* current character of input stream */

int  numcar=0;    /* the number of current character (token) */

void E(void){       /* rule : E->TE'  we rename  E' R */
      T(); R();
}

void R(void){

      if (token=='+') {
      /* rule : E'>+TE' */
      ADVANCE
            T(); R();
      }
      else ;/* rule: E'->epsilon */
}
```

```
void T(void){
     F(); S();  /* rule: T->FT' we rename  T' S */
}

void S(void){
     if (token=='*') {    /* rule: T'->*FT' */
          ADVANCE
       F(); S();
     }
     else ;                   /* rule: T'->epsilon */
}

void F(void){        /* rule: F->(E) */
     if (token=='(') {
          ADVANCE
          E();
          AVANCE_TEST(')')
          }
     else if (isdigit(token)) ADVANCE/* rule: F->0|1|...|9 */
         else  SYNTAX_ERROR
}


int main(){

     ADVANCE/* init the token of the first character */

     E();  /* start non terminal*/
     if (token==EOF)  /* recognized expression */
          printf("\n recognized expression\ n");
     else SYNTAX_ERROR
    /*unrecognized expression but with remaining characters */

 return 0;

}
```

III.4.2 Top-down analysis based on pushdown automaton

Pushdown automata are used in both top-down and bottom-up parsing, with differences in the types of actions and the types of symbols used in the stack. A pushdown automaton in top-down parsing uses a stack to iteratively read terminal symbols (tokens) from the input stream and execute actions based on an analysis table (see Figure 10.). The analysis table is used to support the parsing process and indicate when the parser has to generate a syntax error.

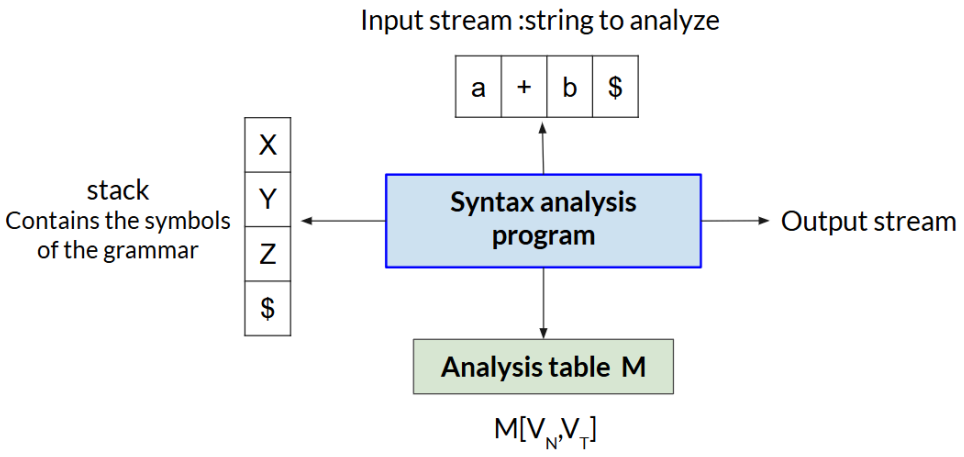


Figure 10. Top-down analysis using pushdown automaton

Example

Let the previous grammar G defined by the following rules

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

Let the input stream (string to analyze) w= id+id. The corresponding analysis table and initial stack values are the following :

a	+	b	$

E	
$	

Stack

Non terminal	Input Symbol					
	id	+	*	(	)	$
E	E→ TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→ FT'			T→ FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→id			F→(E)		

The table M [V_N , $V_T \cup \{\$\}$] contains a production rule or the ERROR action in each of these cells. Parsing the input stream consists of reading the production rule corresponding to the symbol at the top of the stack and the input symbol (token). Then, depending on the case, the automaton either:

- stops by generating a syntax error,
- advances through the stream and pops a token,
- pushes the right part of the rule upside down,
- or terminates by indicating that the parsing was successful.

Considering **X** as the symbol at the top of the stack and **a** as the current input symbol:

- If **X=a=$,** the parser stops and announces that the parsing was successful.
- If **X=a≠$**, the parser pops X off the stack and advances the pointer to the next input symbol (+).
- **If X is a non-terminal**, the parser consults the analysis table entry M[X, a] and replaces X with its x-production. Otherwise, it reports an error.

In the next sections, we will explain how to construct the analysis table and how to analyze an input stream using this table.

III.4.3 Construction of the analysis table M using FIRST and FOLLOW sets

To construct the analysis table we need first to calculate the sets FIRST and FOLLOW sets of the grammar.

Calculation of FIRST sets

If X is a symbol in the grammar, **FIRST(X)** is the set of terminals that begin strings derived from X. If X *=>ε, ε is in FIRST(X).

To calculate the FIRST sets, we apply the following algorithm:

For any symbol in grammar X, apply the following rules until no terminal or ε can be added to the FIRST sets:

- **Rule 1:** If **X is a terminal or ε,** FIRST(X)={x}
- **Rule 2**: If **X is a non terminal** and **X** $\rightarrow \mathbf{y_1\ y_2\ \ldots y_k}$ is a production and **ε** ∈ FIRST(Y_1)...FIRST(Y_{i-1}) (Y_1…. ou Y_{i-1}) $\rightarrow^*$ ε
    - If **ε** ∈ FIRST(Y_j) ∀j= 1…K: add ε to FIRST(X)
    - If y_1 does not derive to ε add nothing, otherwise add FIRST(y_2)....

Example:

Considering the previous grammar G defined by the following rules:

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

The FIRST sets of G are:

- FIRST(E)= FIRST(T)=FIRST(F)= {(, id}
- FIRST(E')= {+, ε}
- FIRST(T')= {*, ε}

Calculation of FOLLOW sets

For each non-terminal A, **FOLLOW(A)** defines the set of a-terminals that can appear immediately to the right part of A in a protophase i.e. the set of terminals a such that there exists a derivation of the form **S** →* α **A** β. If A is the rightmost symbol of a protophase, then $ is in FOLLOW(A).

To calculate the FOLLOW sets, we apply the following algorithm:

Apply the following rules until no terminals can be added to the FOLLOW sets:

- **Rule 1**: Put **$** in FOLLOW (S) if S is the axiom.
- **Rule 2**: If there is a production **A** → α **B** β, the contents of FIRST(β), except ε, are added to FOLLOW (B).
- **Rule 3**: If there exists a production **A** → α **B** or **A** → α **B** β such that FIRST(β) contains ε, the elements of FOLLOW(A) are added to FOLLOW(B).

Example:

Considering the grammar G, FOLLOW sets are :

- FOLLOW(E)= FOLLOW(E')= {), $}
- FOLLOW(T)=FOLLOW(T')= {+,), $}
- FOLLOW(F)= {*,+,), $}

III.4.4 Construction of the top-down analysis table (LL(1) table)

To construct the table, we need to calculate the FIRST and FOLLOW sets as described in the previous sub-sections, then we apply the following method to fill the table M [V_N , $V_T \cup \{\$\}$].

1. For each production **A** → α proceed to steps 2 and 3
2. For each terminal **a** in **FIRST(α)** Add A → α to M[A,a]
3. If **ε is in FIRST(α)**
   a. Add A→α to M[A, b] for each terminal b in **FOLLOW(A)**
   b. If ε is in FIRST(α) and $ in FOLLOW(A) add A → α (A → ε) to M[A,$]
4. Make every undefined entry of M an error

Example: after calculating FIRST and FOLLOW sets for the grammar G, the obtained analysis table is:

Non terminal	Input symbol					
	id	**+**	*	**(**	**)**	**$**
E	E→ TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→ FT'			T→ FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→id			F→(E)		

All empty cells are equivalent to error action. As the table does not contain multiple entries for each cell, the grammar is therefore LL(1) or Left-to-right scanning of input, Leftmost derivation.

III.4.5 LL(1) and LL (k) grammars

LL(1) is a top-down method that is simple to implement and efficient for many languages. The acronym refers to: **L**: means that the parser scans the input **Left to right; L**: it produces a **Leftmost derivation. 1**: refers to the use of only one **token of lookahead** to make parsing decisions.

In general, a grammar G is **LL(1)** if and only if:

1. whatever X → α|β there do not exist two derivations of α and β having a common terminal prefix.
   a. Only a right part α or β can be deleted.
   b. If α can be deleted, then β does not derive into a string having a

common terminal prefix with FOLLOW(X).

2. A grammar whose **analysis table** can be calculated and all entries of which have **a unique production or error.**

Finally, an **ambiguous** or **left-recursive grammar** is NOT LL(1).

Consider the grammar G = ({a, b, c}, {S}, S, {S → ab | ac}). G' is not LL(1) because FIRST(ab)= {a} and FIRST(ac) = { a } → conflict on lookahead a in the LL(1) table. This grammar can be LL(2) i.e. by considering two-token lookahead: On lookahead (a,b): S → ab and on lookahead (a, c): S → ac, No multiply-defined entries, therefore the grammar G' is LL(2).

Therefore we can increase the value of the lookahead parameter to eliminate ambiguity as the parser peek further ahead to disambiguate productions that share longer common prefixes. However, increasing k can imply complexity in analysis as decision logic grow quickly with larger k. Practically, compilers keep k small, usually 1 or 2.

III.4.6 Top-down analysis algorithm following LL(1) method

Using the analysis table, we can proceed to the analysis of inputs using the following algorithm:

Algorithm

Input: A string w and an analysis table M for the grammar G

Result: if w is in L(G) return Success, otherwise Error

Method:

Initialization:

-Stack: $S with S, the axiom of G, at the top

-Input buffer: w$

-Set the source pointer ps to the first symbol of w$

Repeat

Let X be the top symbol of the stack and a be the current input symbol

If X is a terminal or $ **then**

If X=a **then** Pop X and advance Ps

Else error()

else /* X non terminal */

If M [X,a] = x-> y1y2…yk **then**

Pop X

Put yk, yk-1, ….y1 onto the stack with y1 on top

Output X-> y1y2….yk

else error()

Until X=$ /*empty stack*/

End

Example : w=id+id*id

STACK	Input	Action
$E	id+id*id$	E→ TE’
$E’T	id+id*id$	F→ FT’
$E’T’F	id+id*id$	F→id
$E’T’id	id+id*id$	
$E’T’	+id*id$	T’→ ε
$E’	+id*id$	E’→+TE’
$E’T+	+id*id$	
$E’T	id*id$	T→ FT’
$E’T’F	id*id$	F→ id

STACK	Input	Action
$E’T’id	id*id$	
$E’T’	*id$	T’→ *FT’
$E’T’F*	*id$	
$E’T’F	id$	F→id
$E’T’id	id$	
$E’T’	$	T’→ ε
$E’	$	E’→ ε
$	$	ACCEPT

III. 5. Exercises

Exercise 1.

Eliminate ambiguity and left recursion from the following grammar G that has the following rules R: E→ Eab + E | Eac - E | d.

Solution

G unambiguous:

E→ Eab + T | T
T → T ac - F | F
F → d

G non-left recursive:

E→ T E’
E’→ ab+T E’| ε
T → F T’
T’ → ac - F T’| ε
F → d

Exercise 2.

Let the grammar G= ({-, *, id :{a, b,...z,0}}, {S}, S, {S → S-S| SS | S*; S → (S) |id})

1. Is G ambiguous and/or left-recursive? If so, eliminate the ambiguity and left recursion.
2. Determine the First and Follow sets of G.
3. Construct the LL(1) table (top-down parsing table) for G.
4. Is G an LL(1) grammar? Justify your answer.
5. Give the result of the parsing for the input string *w = id id-* id*.

Solution

1. Yes, G is ambiguous since there exist multiple derivations for words due to both left and right associativity.

The unambiguous grammar equivalent to G

S → S-E|E
E → ET| T
T → T*|F
F → (S)| id

Elimination of left recursion :

S → ES'
S'→-ES'|ε
E → TE'
E'→ TE'|ε
T → FT'
T' → *T'|ε
F → (S)| id

we rename S' : X ; E' : R ; et T' : Y

S → EX
X → -EX|ε
E → TR
R → TR|ε
T → FY
Y → *Y|ε
F → (S)| id

2. First and Follow sets of G.

	First	**Follow**
S	(, id	), $
X	-, ε	), $
E	(, id	-,), $
R	(, id, ε	-,), $
T	(, id	(, id, -,), $
Y	*, ε	(, id, -,), $
F	(, id	*, (, id, -,), $

3. LL(1) table (top-down parsing table) for G.

	id	(	)	-	*	$
S	S → EX	S → EX				
X			X → ε	X → -EX		X → ε
E	E → TR	E → TR				
R	R → TR	R → TR	R → ε	R → ε		R → ε
T	T → FY	T → FY				
Y	Y → ε	Y → ε	Y → ε	Y → ε	Y → *Y	Y → ε
F	F → id	F → (S)				

4. No multiply-defined entries, therefore the grammar G is LL(1)

5. Parsing result for the input string w = id id- id*.

Stack	Input	Action		Stack	Input	Action
$S	id id*- id$	S → EX		$XRY	- id$	Y → ε
$XE	id id*- id$	E → TR		$XR	- id$	R → ε
$XRT	id id*- id$	T → FY		$X	- id$	X → -EX
$XRYF	id id*- id$	F → id		$XE**-**	- id$	Move forward
$XRY**id**	**id** id*- id$	Move forward		$XE	id$	E → TR
$XRY	id*- id$	Y → ε		$XRT	id$	T → FY
$XR	id*- id$	R → TR		$XRYF	id$	F → id
$XRT	id*- id$	T → FY		$XRY**id**	**id**$	Move forward
$XRYF	id*- id$	F → id		$XRY	$	Y → ε
$XRY**id**	**id***- id$	Move forward		$XR	$	R → ε
$XRY	*- id$	Y → *Y		$X	$	X → ε
$XRY*	*- id$	Move forward		$	$	**Accept**

Exercise 3.

Let the grammar G=({a,b,+}, {S, E, T, F}, S, {S → aE; E → TE|ε; T → SF; F → +|b}

1. Determine the First and Follow sets of G.
2. Construct the predictive parsing table (top-down parsing table) for G.
3. Is G an LL(1) grammar? Justify your answer.
4. Show the result of predictive parsing for the input string w1=aab and w2=aa++b.

Exercise 4.

Consider the grammar G= ({a, b, c,d}, {S, A, B}, S, {S → aA | bB; A → c; B → d}) and the grammar G' = ({a, b, c}, {S}, S, {S → ab | ac}).

1. Construct the LL(1) parsing table for G. Is it LL(1)? Justify.
2. Show why G' is not LL(1) but LL(2)..

Exercise 5.

Let the grammar G=({;,=,_,(,), a}, {S,E,T}, S, R), R:

S-> ES|E
E-> T;
T-> (T=T)
T-> (T_T)
T->a

1. Write the Flex file ex5.l to perform lexical analysis for grammar G.
2. Write the Bison file ex5.y to perform syntax analysis for grammar G. Display the message "Correct syntax" for each rule of T.
3. Test your analyzers using, for example, the following program: (a=a);(a_(a=a));a;

Solution

ex5.l

```
%{
#include "ex5.tab.h"
%}
%option noyywrap
%%
";" return POINTV;
"=" return EGAL;
"_" return UNDER;
"(" return PO;
")" return PF;
"a" return A;
[ \t\n] /* ignore*/
. printf("lexical error\n");
%%
```

ex5.y

```
%{
#include <stdio.h>
#include "ex5.c"
void yyerror(char* s);
%}
%token POINTV EGAL UNDER PO PF A
```

```
%start S
%%
S : E S | E ;
E : T POINTV ;
T : PO T EGAL T PF
| PO T UNDER T PF
| A {printf("Correct Syntax \n");};
%%
void yyerror(char* s) {
  printf("syntax Error \n");
}
void main() {
      printf("Enter your program:\n");
      yyparse(); // cal syntax analyzer
      printf("end of syntax analysis \n");
}
```

Exercise 6.

Let the grammar G=({;,+,*, id}, {S,E,T}, S, R), R:

S-> ES|E
E-> T;
T-> T+T
T-> T*T
T-> id

1. Write the Flex file ex6.l to perform lexical analysis for grammar G, assuming that id is an unsigned integer.
2. Write the Bison file ex6.y to perform syntax analysis for grammar G.
3. Test your analyzers using the following example program:
   9+8;5+4*3;2;
4. Test your parser again. What do you observe?
5. Modify your grammar by applying associativity and precedence rules. Assume that the + operator has higher precedence than the * operator.

Exercise 7.

Let the grammar G = ({if, (,), n, >}, {S, expr}, S, {S → if (expr) expr; expr → n | expr > expr}).

1. Provide the lexical analyzer in Flex for grammar G where n is an unsigned integer.
2. Provide the corresponding Bison parser

Exercise 8.

Let the following program example:

```
Start
ADD 1, 4, -3, 8
ADD -4, 10, 20
ADD 8
End
```

1. Define a grammar G that recognizes the language described in the previous program.
2. Write the Flex code ex8.l to perform lexical analysis based on grammar G.
3. Write the Bison code ex8.y to perform syntax analysis based on grammar G. Using semantic actions, the code should also display the result of the operations. For example:

   ```
   ADD = 10
   ADD = 26
   ADD = 8
   End of semantic analysis
   End of syntax analysis
   ```

4. Test your analyzers with the above example.

Exercise 9.

Let the grammar G= ({ *, +, a, b,c}, {A , C, B, E, D}, A, {A → BC; C → *BC| ε; B → DE; E→ +DE|c; D → bAc|a}).

1. Determine the First and Follow sets of G.
2. Construct the predictive parsing table (top-down parsing table) for G.
3. Is G an LL(1) grammar? Justify your answer.
4. Show the result of predictive parsing for the input string w=baccc.

CHAPTER IV. Syntax Analysis Using Bottom-up Methods

Unlike Top-down methods, bottom-up analysis can handle a wider class of grammars, especially for programming languages. Bottom-up parser starts from the input symbols of the source code and works upward to reach the start symbol of the grammar. Bottom-up analysis is also called the shift-reduce method. In fact, the analyzer shifts input symbols onto the stack when reading the input stream. When a sequence of stack symbols matches the right-hand side of a grammar production, it is reduced to the left-hand side non-terminal. To use this parser the grammar must be **unambiguous,** i.e. we have only one possible action for a given input symbol. We can find different bottom-up analysis methods, including : **LR** (Left-to-right, Rightmost derivation) methods, **SLR** (Simple LR), and **LALR** (Look-Ahead LR).

IV.1 Principles of LR parsers

LR is a bottom-up analysis method where **L** refers to how scanning of the input is performed (Left to right scanning of the input and **R** to how derivation is constructed (Right most derivation in reverse). LR can describe more languages than LL grammars and is generally used for context-free grammars. LR is also a predictive method, LR(0) refers to methods without a lookahead parameter and LR(k) to methods that process the k-symbols of the input line.

LR parsers focus on a stack and an analysis table like top-down parsers (see Figure 11). The parsing table contains actions that guide the parser on whether to shift or move the pointer on the input, reduce or replace the set of symbols of the stack by the left non terminal symbol of the production rule, accept or end the analysis. In the case of no actions or remaining strings that can not be analyzed the parser reports an error.

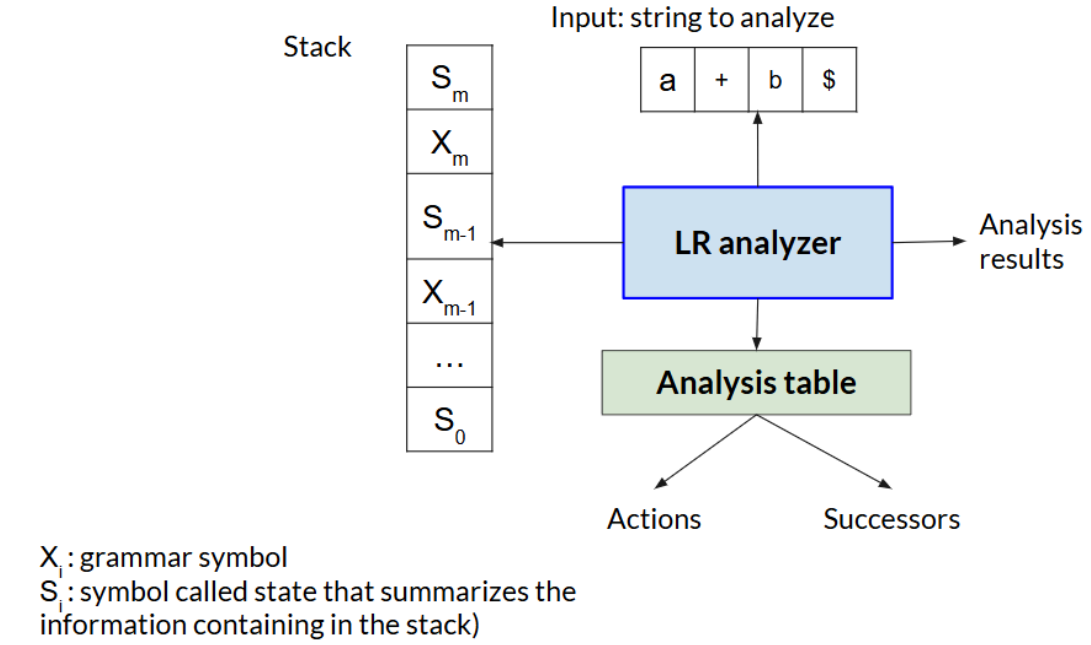


Figure 11. LR parser internal functioning

The construction of the analysis table depends on the method:

- **SLR** (Simple LR): which is easy to implement but the least powerful method.
- **Canonical LR** (LR(1)): it is more powerful than the other methods but the most expensive.
- **LALR** (Look Ahead LR): which is intermediate in cost and power and generally used by parser generation tools such as Yacc.

In the rest of this chapter we introduce how the analysis table can be constructed following these methods as well as LR parsing algorithm.

IV.2 Construction of the SLR(1) analysis table

It is based on the construction of a set of items of a deterministic finite state automaton to construct the analysis table (also called LR(0) item set). The Actions and Successors functions of the analysis table of SLR(1) represent the transition function of the finite state automaton.

IV.2.1 Construction of the list of LR(0) items

An LR(0) item of a grammar G is a production of G with a point "●" representing a position on its right side.

Examples:

- A→ ●XYZ: we hope to see a string derivable from XYZ
- A→ X●YZ: we have a string that derives from X and we hope to see as input a string derived from YZ
- A→ XY●Z: we have a string that derives from XY and we hope to see as input a string derived from Z
- A→ XYZ●: we reached the leave and we have the string derived from XYZ
- A→ ● is the same writing as A→ ε

Augmented grammar

To calculate the list of items we have to use an augmented grammar in order to define later our initial state of the automaton.

If G is a grammar with a starting symbol S, G' is its **augmented grammar** if we add the following production rule to G rules: S' → S, such as S' is the starting symbol of G'

Example

Let the grammar G defined by the following rules:

E→ E + T | T
T→ T ∗ F | F
F→ (E) | id

The augmented grammar G' is therefore defined by the following rules:

E'→ E
E→ E + T | T
T→ T ∗ F | F
F→ (E) | id

such that **E'** is the starting symbol of G'.

To construct the set of items of G' we use two functions **Closure** and **transition**.

Closure of an item

If I is a set of items for a grammar G, **Closure(I)** is the set of items constructed from I by the following two rules:

1) Initially, place each item of I in Closure(I)
2) If A → α ● B β is in Closure(I) and B →ɣ is a production then add B →●ɣ to Closure(I)

We apply this rule until no more items can be added.

Examples:

G': E'→E
E→E + T |T
T→T ∗ F|F
F→(E)|id

- I={ [E' →● E] } : closure(I)= { [E'→●E], [E→●E+T| ●T], [T→●T ∗ F |●F], [F→●(E)|●id] }
- I={ [T→T ∗ ●F] , [E→T●]} : closure(I)= {[T→T∗●F] , [E→T●], [F→●(E)| ●id]}

Transition of an item

Transition(I, X): such that I is a set of items and X a symbol of the grammar is the closure of the set of all items [A → α X ● β] such that [A → α ● X β] ∈ I

Exemple:

I={ [E' → E●] , [E → E●+T]}

Transition (I, +)= {[E → E + ●T], [T→●T ∗ F| ●F], [F→●(E)| ●id]}

IV.2.2 Algorithm of LR(0) list of items

Algorithm : Construct C, the canonical collection of the LR(0) item set for G'

C:= {Closure({S' → ●S})};

Repeat

For each set of items I of C and for each symbol X of G such that Transition(I, X) is non-empty and ∉ C

Add **Transition (I, X)** to **C**

Until no more itemsets can be added to C

Example:

Let the grammar G=({id}, {S,E}, S, R), R:

S → SE | E
E → id

Give the automaton of LR(0) items for G.

Augmented grammar:

S' → S
S → SE | E
E → id

- FIRST(S')=FIRST(S)=FIRST(E)={id}
- FOLLOW(S')={$}
- FOLLOW(S)=FOLLOW(E)={id, $}

LR(0) items

- I0=Closure([S' → ● S])=

  {[S' → ● S], S → ● SE | ● E], [E → ●id]}

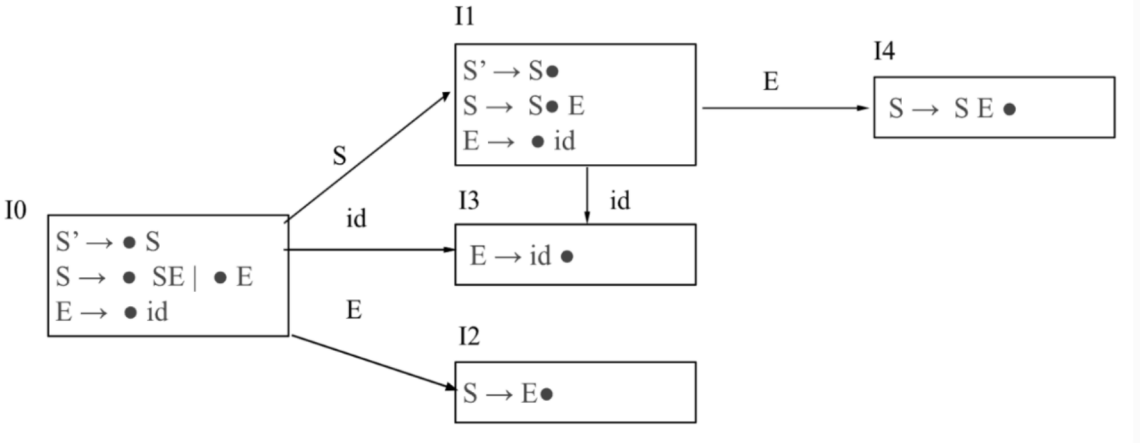
I0 :

- I1= Transit(I0, S)= {[S' → S●], [S → S●E], [E → ●id]}
- I2= Transit(I0, E)= { [S → E●]}
- I3= Transit(I0, id)= {[E → id●]}

I1:

- I4= Transit(I1, E)= {[S → SE●]}
- Transit(I1, id)= I3

The automaton of LR(0) items for G



IV.2.3. Construction of the analysis table SLR(1)

Once the set of items and/ or the automaton of LR(0) is constructed, the next step is to build the analysis table.

SLR(1) table lines correspond to the states of the automaton (or items) and the columns are the actions of the analyzers and the successors that represent the transition in the case of a non terminal symbol of the input.

SLR(1) table can be constructed following this algorithm:

1) Construct C={I_0, …I_n}: LR(0) for G'

2) The state i is constructed from I_i

The actions are :

- If [**A** → α ● a β] ∈ I_i and Transit(I_i, a) =I_j

  If a is a terminal Fill **Action [i, a]** with **shift j**

  If a is a non terminal Fill **Successor [i, a]** with **j**

- If [**A** → α ●] ∈ I_i and A ≠ S

  Fill **Action [i, a]** with **reduce** with **A**→α for all a of FOLLOW(A)

  If [**S'** → **S●**] ∈ I_i Fill **Action [i, $]** with **ACCEPT**

Example

Let the augmented grammar G=({id}, {S,E}, S, R}, R is defined by:

S → SE | E
E → id

G' the corresponding augmented grammar is defined by the following production rules:

(0) S' → S
(1) S → SE
(2) S → E
(3) E → id

The corresponding LR(0) table is:

Etats	Actions		Successeurs	
	id	**$**	**S**	**E**
0	d3		1	2
1	d3	ACCEPT		4
2	r2	r2		
3	r3	r3		
4	r1	r1		

Where:

d_i: shift and stack state i (also named **s_i**)
r_j: reduce with the rule j
accept: accept the input
Empty entry: error

G is SLR(1) since there is no conflict in the table (its table is deterministic, without multiple actions).

IV.3. LR syntax analysis algorithm

Using an LR table (SLR(1), LR(k), LALR(1), we can analyze an input string on the basis of the following algorithm:

Algorithm

Initialize the pointer Ps to the first symbol of w$

Begin

Repeat

Let S be the state at the top of the stack and a be the current symbol pointed to

by Ps

If **Action [S, a]= shift S'** then

Stack a then S

Advance Ps to the next entry symbol

If **Action [S, a]= reduce with A→β** then

Pop 2 x |β| symbol from the stack

Let S' the new state at the top of the stack

Stack A the Successor[S', A]

Output the production rule A→β

Else if **Action [S, a]= ACCEPT** then Return;

Else error();

End

Example. Considering the previous grammar and the following string to parse w= id id

G':
(0) S' → S
(1) S → SE
(2) S → E
(3) E → id

Stack	Input	Action
0	id id$	d3
0 id 3	id$	r (E → id)
0 E 2	id$	r (S → E)
0 S 1	id$	d3
0 S 1 id 3	$	r (E → id)
0 S 1 E 4	$	r (S → SE)
0 S 1	$	ACCEPT

w is therefore an accepted string by G.

IV.4 LR(1) Analysis method

LR(k) is an efficient bottom-up parsing technique that can be used for many context-free grammars. L: represents the left-to-right path, R represents the rightmost branch in reverse, and k represents the number of feedforward input symbols. Among the most used methods is LR(1).

LR (1) analysis uses a large set of items called LR(1) items. The difference between LR(0) and LR(1) is that in LR(1) items, it is possible to carry more information in a state, which will exclude unnecessary reduction states. This additional information is incorporated into the state by the look-ahead symbol.

The general syntax of an item becomes [A → ∝● B, a] Where A→∝●B is the production rule and a is a terminal or right end marker $. Next, we describe how the set of LR(1) items are calculated and used to build the LR(1) table.

IV.4.1 Construction of set of items LR(1)

The items in LR(1) are calculated by the same functions of LR(0), i.e. Closure and Transition functions, but with the consideration of the look-ahead parameter.

Closure(I) is the set of items constructed using the following rules:

- Initially, we place all the elements of I in closure(I)
- For each item of closure (I) of the form **[A → α ● B ɣ, a]** and for each rule of the form B→ **σ**, we add the items [B→ ●**σ**, b] in closure (I) with: α, ɣ, **σ** string containing terminal and/or non-terminal symbols, B a non-terminal symbol, **b ∈ FIRST(ɣa)**

Transition(I, X) with X a symbol of the grammar is calculated as follows:

- We search in I for all items of the form **[A→α ●X β, a]**
- We compute the Closure set ([A → α X ● β, a])
- We add to Transition (I, X) all the sets constructed by the two previous rules until no items can be added

Example.

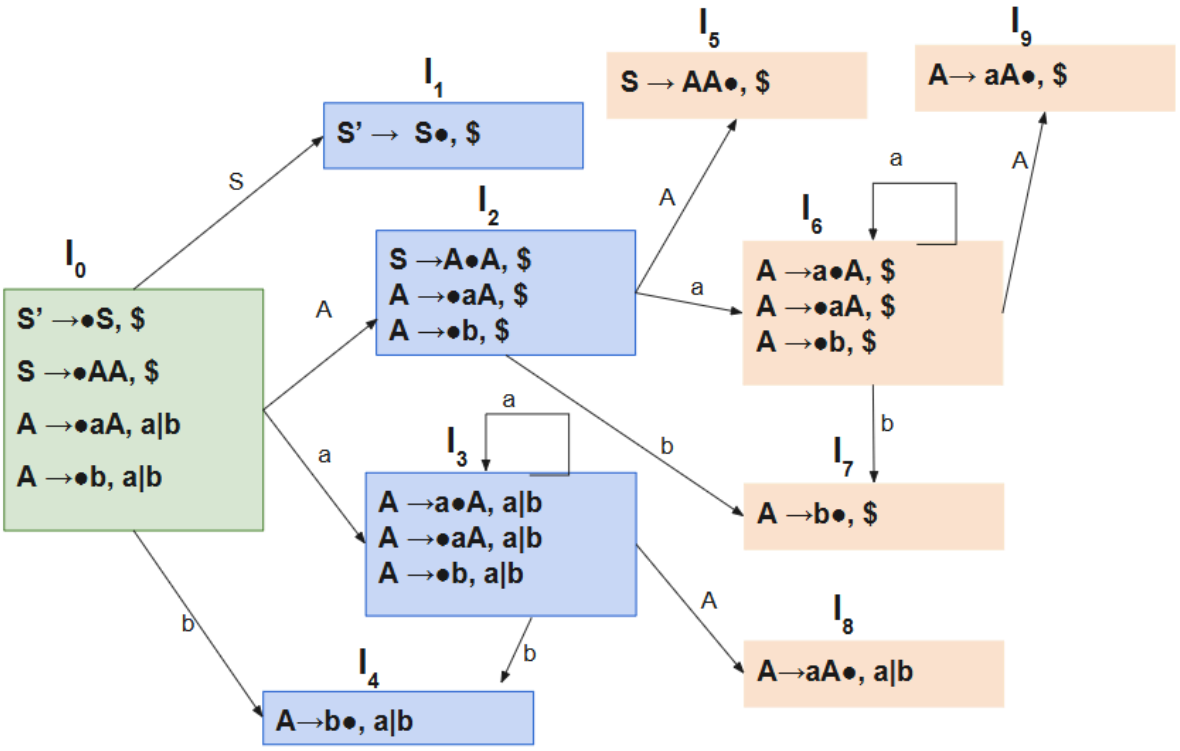
Let the grammar G=({a,b}, {S,A}, S, R}, R is defined by:

S→AA
A→aA|b

The augmented grammar G' rules are:

S'→S
S→AA
A→aA|b

The corresponding LR(1) automaton is:



IV.4.2 Construction of the analysis table LR(1)

The construction of LR(1) table is similar to SLR(1). The only difference is in considering the look-ahead parameters in some actions. The algorithm of table construction is:

1) Build C={I_0, …I_n}: LR(1) for the augmented grammar G'
2) The state i is constructed using $I_i \in$ **C.** The actions are:
   a) If [**A** → α ● a β, b] ∈ I_i and Transit(I_i, a) =I_j

      If a is a terminal, Fill Action [i, a] with **shift j**

      If a is not a terminal, Fill **Successor[i, a]= j**

   b) If [**A** → α ●, b] ∈ I_i and A ≠ S' Fill Action [i, b] with **reduce with A** →α
   c) If [**S'** → **S●**] ∈ I_i Fill Action [i, $] with **ACCEPT**

Example

G':

(0) S'→ S
(1) S→ AA
(2) A→ aA
(3) A→ b

State	Actions			Successors	
	a	**b**	**$**	**A**	**S**
0	s3	s4		2	1
1			acc		
2	s6	s7		5	
3	s3	s4		8	
4	r3	r3			
5			r1		
6	s6	s7		9	
7			r3		
8	r2	r2			
9			r2		

No multiple entries, therefore G is LR(1)

IV.5 LALR Analyzer

LR analyzers are the most powerful methods as they can analyze a large set of grammars. However, LR methods are time-consuming because of the large number of states. LALR is a method that combines the benefit of SLR(1) method by analyzing all grammars SLR(1), but also a number of grammars LR(1) especially for programming languages. The relationship between the class of LR grammars is presented in Figure 12.

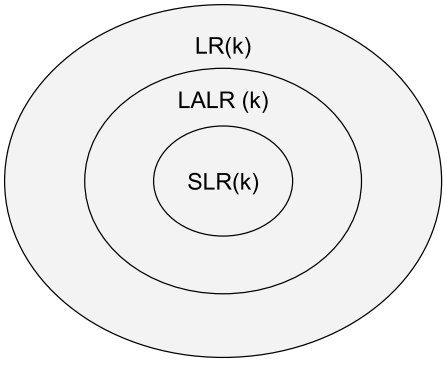


Figure 12. Class of LR grammars

LALR method is based on the optimization of the LR(1) analyzer by reducing the number of states. It consists in merging similar states that differ only by their look-ahead parameter. Similarly, this consists in reducing the lines of the LR(1) table. The grammar is LALR(1) only

if merging all similar items that have different look-ahead parameters allows one to obtain an analysis table without multiple entries. In the previous example of LR(1 table,: I3 and I6 are similar except their look ahead parameter. It is the same for (I4, I7) and (I8, I9). We can reduce the previous table as follows:

State	Actions			Successor	
	a	b	$	A	S
0	s36	s47		2	1
1			acc		
2	s36	s47		5	
36	s36	s47		9	
47	r3	r3			
5			r1		
36	s36	s47			
47			r3		
89	r2	r2			
89			r2		

The obtained reduced table is:

State	Actions			Successors	
	a	b	$	A	S
0	s36	s47		2	1
1			acc		
2	s36	s47		5	
36	s36	s47		89	
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

No multiple entries, therefore G is also LALR(1).

IV.6 Exercises

Exercise 1

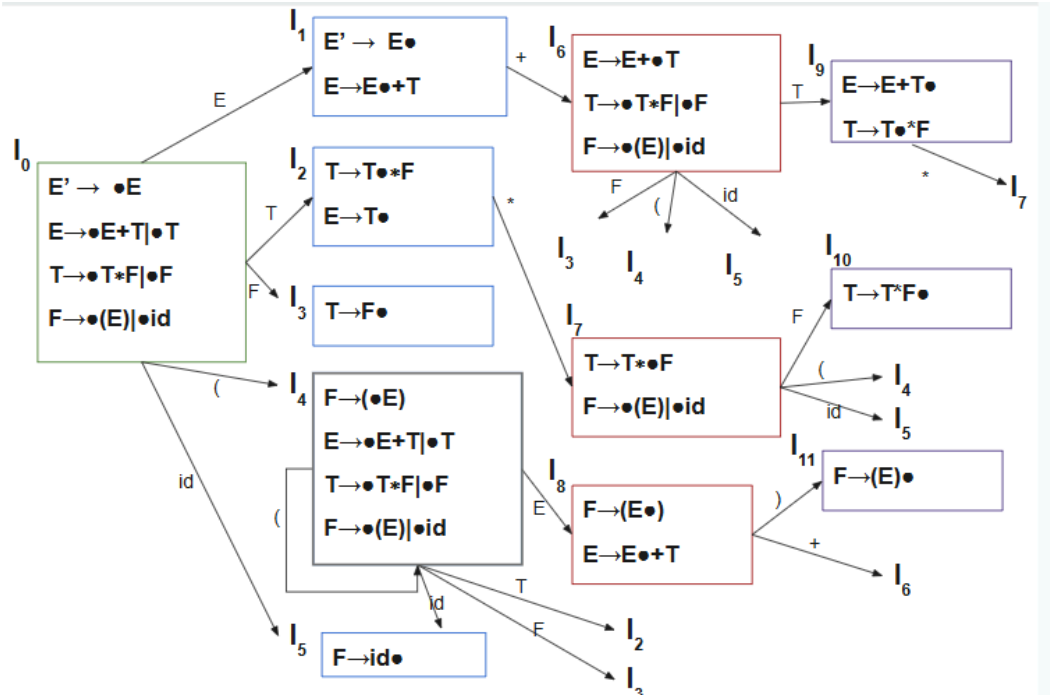
Let the grammar G=({+, *, (,), id}, {E, T, F}, E, R). R:

E → E + T [T
T → T * F | F
F → (E) | id

1. Determine the First and Follow sets of the grammar G.
2. Construct the canonical LR(0) items automaton for G.
3. Build the LR(0) parsing table for G.
4. Is G an SLR(1) grammar? Justify your answer.
5. Parse the input string w = id * id + id using this parsing table.

Solution

	First	Follow
E	(, id	$,), +
T	(, id	$,), +, *
F	(, id	$,), +, *



State	Actions						Successor		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				**acc**			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

The parsing result for w=id*id+id:

Stack	**input**	**Action**
1) 0	Id *id + id$	shift
2) 0 id 5	*id + id$	r F->id
3) 0 F 3	*id + id$	r T-> F
4) 0 T 2	*id + id$	shift
5) 0 T 2 * 7	id + id$	shift
6) 0 T 2 * 7 id 5	+ id$	r F -> id
7) 0 T 2 * 7 F 10	+ id$	r T-> T*F
8) 0 T 2	+ id$	r E->T

9) 0 E 1	+ id$	shift
10) 0 E 1 + 6	id$	shift
11) 0 E 1 + 6 id 5	Id$	r F -> id
12) 0 E 1 + 6 F 3	$	r T -> F
13) 0 E 1 + 6 T 9	$	r E -> E+T
14) 0 E 1	$	Accept

Exercise 2

Let the grammar G =({a}, {S,E}, S, {S → E: E → Ea|a})

1. Is G an LL(1) grammar? Justify your answer.
2. Construct the canonical LR(0) items automaton for G.
3. Build the LR(0) parsing table for G.
4. Is the grammar G an SLR(1) grammar? If so, analyze the input string w = aaaa using the corresponding parsing table.

Exercise 3

Let the grammar G =({a,b,c}, {S,T}, S, {S → T | cb; T → aTb | c})

1. Construct the canonical LR(0) items automaton for G.
2. By constructing the parsing table, show that the grammar G is not SLR(1).
3. Construct the canonical LR(1) items automaton for G.

Solution

1. LR(0) item collection

Augmented grammar:

(0) S' → S
(1) S → T
(2) S → cb
(3) T → aTb
(4) T → c

LR(0) items

I0=Closure([S' → ●S])= {[S' → ●S], [S → ●T| ●cb], [T → ●aTb| ●c] }

I0 :

I1= Transit(I0, S)= {[S' → S●]}
I2=Transit(I0, T)={ [S → T●]}
I3=Transit(I0, c)= { [S → c●b], [T → c●] }
I4=Transit(I0, a)= { [T → a●Tb] [T → ●aTb| ●c]}

I3 :

I5= Transit(I3, b)= { [S → cb●] }

I4:

I6= Transit(I4, T)={[T → aT●b]}
Transit(I4, a)= I4
I7= Transit(I4, c)= {[T → c●]}

I6:
I8= Transit(I6, b)={[T → aTb●]

2. G is not SLR(1):

- First(T)= First(S)= First(S')={a, c}
- Follow(S') = Follow(S)= {$}
- Follow(T)={b, $}

States	Actions				Successors	
	a	b	c	$	S	T
0	s4		s3		1	2
1				ACCEPT		
2						
3		**s5 r4**		r4		
4	s4	r4		r4		6
5				r2		
6		s8				
7		r4		r4		
8		r3		r3		

G is not SLR(1) because of the conflict shift/reduction in the item I3.

5. Parsing result for w= aacbb

Stack	Input	Action
0	aacbb$	s4
0a4	acbb$	s7
0a4a7	cbb$	s8
0a4a7c8	bb$	r (T → c)
0a4a7T10	bb$	s11
0a4a7T10b11	b$	r (T → aTb)
0a4T6	b$	s9
0T2	$	r (T → aTb)
0S1	$	r (S → T)
0S1	$	ACCEPT

Parsing for w= aacb

Stack	Input	Action
0	aacb$	s4
0a4	acb$	s7
0a4a7	cb$	d8
0a4a7c8	b$	r (T → c)
0a4a7T10	b$	s11
0a4a7T10b11	$	Error

Exercise 4

Let the following grammar:

G= ({if, =, >,), then, (, id}, {S, E, A, B}, S, {S→E; E→if A then E|id=A; A→ A>B|B; B→ (A)|id})

1. Construct the canonical LR(0) items automaton for G.
2. Build the LR(0) parsing table for G.
3. Is the grammar G an SLR(1) grammar? If so, analyze the input string w =if id>id then id=id
4. Without building new tables, Is G LL(1) ? is G LR(1)? Justify your answer.

Exercise 5.

Let the grammar G= ({x,y,z}, {A, B, C, D}, A, {A→ B; B→ CD; C→ xCz|y; D→ xD|y|ε})

1. Determine the First and Follow sets of G.
2. Build the top-down analysis table.
3. Is G LL(1) ? Justify.
4. Construct the canonical LR(0) items set (or the automaton) for G.
5. Build the LR(0) parsing table for G.
6. Is G an SLR(1) grammar? Justify your answer. If so, parse the input string w = yx using the SLR(1) table.

Exercise 6

Let the grammar G= ({a,b}, {S, A, B, C}, S, {S→ A; A→ B|BC; B→ aAb|b, C→ a})

1. Construct the LR(1) items list of G.
2. Build the LR(1) parsing table for G.
3. Is the grammar G an LR(1) grammar? If so, analyze the input string w=abba.
4. Is G LALR(1)? Justify your answer.

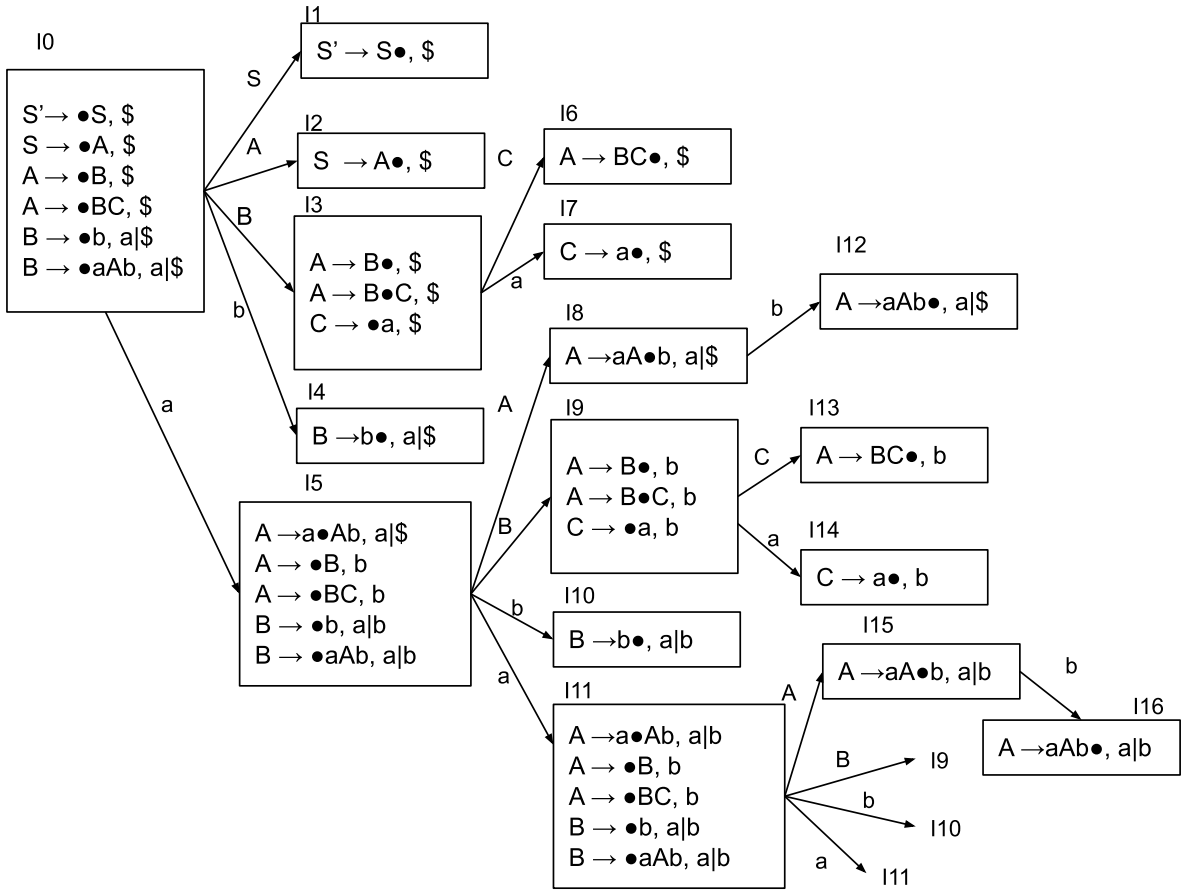
Solution

Augmented grammar

S' → S (1)
S → A (2)
A → B (3)
A → BC (4)
B → aAb (5)
B → b (6)
C → a (7)

First and follow sets

- First(S)={b, a}; First(A)={b, a}; First(B)={b, a}, First(C)={a}
- Follow(S)= {$}; Follow(A)= {$, b}; Follow(B)= {$, b, a}, Follow(C)= {$, b}



LR(1) table

	a	b	$	S	A	B	C
0	s5	s4		1	2	3	
1			ACC				
2			r2				
3	s7						6
4	r6		r6				
5	s11	s10			8	9	
6			r4				
7			r7				
8		s12					
9	s14	r3					13
10	r6	r6					

11	s11	s10			15	9	
12	r5		r5				
13		r4					
14		r7					
15		s16					
16	r5	r5					

No conflict in the table, G is therefore LR(1).

Analysis of w:

Stack	Input	Action
0 0a5 0a5b10 0a5B9 0a5A8 0a5A8b12 0B3 0B3a7 0B3C6 0A2 0S1	abba$ bba$ ba$ ba$ ba$ a$ a$ $ $ $ $	s5 s10 r(B → b) r(A → B) s12 r(B → aAb) s7 r(C → a) r(A → BC) r(S → A) ACCEPT

G is LALR as the following reduced table does not contain multiple entries.

	a	**b**	**$**	**S**	**A**	**B**	**C**
0	s511	s410		1	2	39	
1			ACC				
2			r2				
39	s714	r3					613
410	r6	r6	r6				
511	s511	s410			815	39	
613		r4	r4				
714		r7	r7				
815		s1216					
1216	r5	r5	r5				

Exercise 7

Let the grammar G= ({a,b,*}, {E, T, F}, E, {E→ *TF; T→ F|ϵ; F→ a|b|ϵ})

1. Construct the LR(1) items list of G.
2. Build the LR(1) parsing table for G.
3. Is the grammar G an LR(1) grammar? If so, analyze the input string w=*aa.
4. Without constructing a new table, Is G LALR(1)? Justify your answer.

CHAPTER V. Syntax-Directed Translation and Code Generation

Syntax-driven translation refers to processing operations made during syntax analysis. These operations, known as semantic actions or routines, include type control, object binding, and the generation of an intermediate form.

Type control of variables and functions is among the basic operations in semantic analysis. For example, if a variable, x, has been defined as an integer type, then it has been assigned an array, an error is associated because x is not an array. The analyzer will also perform object building after identifying and checking the type of all variables and functions outputs. The syntax-directed translation result is an intermediate form of the target code. This intermediate form, which is an abstraction of the source program, can be an abstract tree (Abstract Syntax Tree), a three-way code (such as quadruplets, triplets), etc. Perl is an example of a programming language that generates an abstract syntax tree.

Intermediate code aims to facilitate the transformation of source code to machine language. The intermediate form is independent from a particular machine or operating system. Some programming languages do not produce machine code, but an intermediate form such as bytecode in Java language and a Common Intermediate Language in .NET. These intermediate codes have to be transformed into native code. This operation is performed by JIT (Just-In-Time compilers) or on-the-fly compilers.

V.1 Syntax-directed translation

The goal of parsing is to verify the conformity of the input to be analyzed with a given syntax. The syntax tree underlying parsing is not explicitly constructed but used to detect syntax errors. The objective of the compiler is to produce an output that can be, for example, intermediate code, native code, or the evaluation of an expression. As the syntax tree is not explicitly constructed, syntax-directed translation is therefore used to generate an output that is closely linked to the parsing. To achieve this, the symbols in the grammar will have attributes that can be integer values or any other type necessary for the desired translation. In addition, suitable data structures can be used to meet the specific needs of the translation. The translation is carried out by inserting code fragments commonly called semantic routines (actions or rules) into the parser. This grammar is named “attributed grammar”.

V.1.1 Attributed grammar

An **attributed grammar** associates a set of attributes with each symbol, terminal and non-terminal, in the grammar. An attribute stores typed information (integers, char strings, etc.) and the attribute **X.val** is the value attribute associated with a symbol X. A symbol can have multiple X attributes {val_1, val_2,, …, val_k}.

Each grammar symbol in the stack will "point" to its attributes. These attributes can be classified into two categories: inherited attributes and synthesized attributes.

- **Synthesized attribute**: the value of a symbol is calculated from the descendants of this symbol in the syntax tree.
- **Inherited attribute:** the value of a symbol is calculated from the ancestors and siblings of this symbol in the syntax tree.

Each production rule of the grammar corresponds to one or more semantic rules indicating the calculation method for certain attributes. The calculation of some attributes may depend on others. When the rule is recursive (same symbol in the left and right parts of the production), the right-hand occurrences are indicated to distinguish them from the left-hand occurrence.

Example 1:

Let G be the grammar used for arithmetic expressions. We use only synthesized attributes to evaluate an expression. X.val denotes the attribute associated with a symbol X in the grammar.

Rule	Semantic rules or values
$E \rightarrow T$	E.val=T.val
$E \rightarrow E_1 + T$	E.val=E_1.val+T.val
$T \rightarrow F$	T.val=F.val
$T \rightarrow T_1 * F$	T.val=T_1.val*F.val
$F \rightarrow (E)$	F.val=E.val
$F \rightarrow 0$	F.val=0
$F \rightarrow 1$	F.val=1
$F \rightarrow 9$	F.val=9

We can also limit the semantic actions to type evaluation and attribution.

Example 2:

Let G be the grammar based on real and integer symbols. We use in this case the attribute type for semantic actions.

Rule	**Semantic rules or values**
$S \rightarrow EF$	F.Typehl <- E.type
E → ‘integer”	E.type <-integer
E → ‘real’	E.type <-real
$F \rightarrow F_1$, id	F_1.Typehl <- F.typeh
$F \rightarrow$ id	add id to symbol table

V.1.2 Implementation of semantic analysis using YACC

With YACC, each symbol is associated with a single semantic value (of type YYSTYPE, which can be a union of different types). Each symbol's attribute can be a pointer to a C structure containing multiple pieces of information.

The notation of the attribute associated with a symbol X in a production X → α is **$$**.

The notation of the attribute associated with an occurrence of the symbol X in a production Y → d1d2d3Xd5d6 is **$4**, that is, its index on the right-hand side.

Example.

Let the following example of Bison file used for arithmetic expressions:

```
%{
#include <stdio.h>
#include <ctype.h>
#define YYSTYPE int /*explicit definition of YYSTYPE as int */

int yylex(void);

void yyerror(char *s);
%}

%token
DIGIT

%%
list:              { /* empty string at the end of file Ctrl-D */}
| liste line;
l line: '\n'        {/* empty line : empty expression */}
| error '\n'         {yyerrok; /* following the end of line */}
| expr '\n'          {printf("Result : %d\n",$1);};
expr: terme       {$$ = $1; /* default value */}
| expr '+' terme  {$$ = $1 + $3;};
term : fact         {$$ = $1;}
| term '*' fact  {$$ = $1 * $3;};
fact: DIGIT {$$ = $1;}
| '(' expr ')'  {$$ = $2;};
%%

int yylex(void){
int c=getchar();
while(c==' '||c=='\t')c=getchar(); /* filtering*/
if (isdigit(c)){
yylval=c-'0';
```

```
return  DIGIT;
}
else
return c;
}


void yyerror(char *s) {
fprintf(stderr,"%s\n",s);
}

int main(void){
yydebug=0;
return yyparse();
}
```

V.1.3 Translation schemes and generation methodology

A syntax-directed translator is equivalent to a syntax analyzer where the grammar symbols have attributes and semantic procedures that will be added to the source code of this analyzer.

A translation schema is a syntax-driven abstraction of the translator. It is defined by the syntactic grammar in which attributes are associated with the grammar symbols and semantic actions (C or other code) are inserted into the MDPs (Member Rights Productions). During parsing, the semantic routines defined in the MDPs will be executed after processing the grammar symbols preceding them in these MDPs.

In order to design a translation schema we have to ensure that the value of an attribute is available when an action refers to it and insert the semantic routines into the right attributes, depending on the problem to be solved. To design a translation scheme generating an intermediate form, the designer must first define the form to be obtained. For example, the translation of an if-then-else statement into 3-address code can result in two intermediate forms.

Example of source code:

```
if (a>b)
then a++
else b--
```

The corresponding intermediate form can be:

```
1. CMP a,b
2. Jump Greater 5
3. b := b-1
4. Jump 6
5. a :=a+1
```

The form can be freely defined by the designer and has to produce the fewest instructions and whose translation is adapted to the reading direction of the source program. The translation scheme has also to consider the complexity of the common structures used in programming languages.

For the translation that generates three-address code, we will use in the semantic routines that generate quadruplet instructions (opcode, source l, source 2, destination) that inserts the 4-tuple given as an argument into the data structure generated. The procedure call depends on the syntactic analysis method: top-down or bottom-up.

V.2 Running environments

It is important to understand how to proceed program execution before discussing how the target code or native code is generated by the compiler. Programming languages provide mechanisms for procedure calls and dynamic memory allocations. The execution environment must specify the implementation details of these mechanisms.

V.2.1 Procedures call and runtime activation blocks

An executable file stored in secondary memory contains machine instructions, initialized static data, and metadata about uninitialized static data (bss section). Together, the data and bss sections are often collectively referred to as the data segment. The sizes of these segments are determined at the end of the compilation process. When the executable is run, the operating system allocates virtual memory for the program, typically ranging from address 0 to a maximum limit (e.g., 512 GB in Linux). This **memory** is organized with the text segment at lower addresses, followed by the data and bss segments. Additional memory is dynamically allocated during execution in two areas: the heap and the stack. The heap, used for general dynamic allocations, grows upward from the end of the bss segment, while the stack, used for managing function calls, grows downward from the upper end of the memory space. The operating system manages the mapping of virtual to physical memory, often using paging mechanisms, though such low-level details are outside the scope of this discussion.

A **procedure** is defined by a name (identifier) and a body containing a sequence of instructions. While the procedure body is fixed in the code, it can be called multiple times during execution, including recursively. The implementation of procedure calls varies by programming language, but generally follows a common structure: a prologue, the procedure body, an epilogue, and a return instruction. When a procedure is called, the code includes instructions for argument passing, pre-call setup, the JSUB (jump to subroutine) instruction, and post-call handling. To manage these calls, especially when multiple or nested calls occur, the system creates an **activation block** on the stack for each call.

An **activation block or record** is a memory structure created on the stack during a procedure call. It contains all the necessary information for managing the procedure's execution and returning control to the caller. While the exact structure may vary depending on the language or system, a typical activation block can include one or more of the following components:

- **Return address**: where execution should resume after the procedure ends.
- **Effective parameters**: the actual arguments passed from the calling procedure.

- **Control link**: a reference to the activation block of the calling (or previous) procedure.
- **Access link**: used to access non-local variables from enclosing scopes.
- **Saved machine state**: includes register values or other state information saved before the call.
- **Local data**: variables declared within the procedure.
- **Temporary data**: used for intermediate computations during the procedure.

Example:

When a procedure A calls procedure B, which then calls procedure C. The stack evolves as follows:

1. After calling A → stack contains A's activation block.
2. A calls B → B's activation block is added above A's.
3. B calls C → C's block is added on top.
4. When C finishes, its block is removed, and control returns to B.
5. After B finishes, its block is removed, returning control to A.

This illustrates how the stack grows and shrinks dynamically with nested procedure calls, managing each call's context through activation blocks.

V.2.2 Memory allocation

Memory in a program is allocated in different ways depending on how and when data is used. The basic method is the **static allocation** which is used for variables with known size at compile time. In this case, the memory is reserved in the data or bss section.

Example: In C language a static memory allocation is performed for variable declaration:

```
int x; // 4 bytes allocated statically
```

When the program uses local variables and procedure calls, **stack allocation** is used. Each call creates an activation block pushed to the stack, containing: return address, parameters, local and temporary data, and saved machine state. The memory is managed using a stack pointer. On return, the block is removed, and the previous state is restored. The size of variable data on the stack depends on their type. For data like arrays whose size is only known at runtime. They are still allocated on the stack, but space is **dynamically** calculated.

Example:

```
void func(int n) {
    int arr[n]; // size decided at runtime
}
```

This system ensures efficient memory use during function execution and allows support for recursion, nested calls, and **dynamic local storage**.

V.3 Code generation and optimization

The code generation phase, corresponding to a source program, is the final phase of compilation. It takes as its starting point the intermediate code generated by the previous phases. The generated code is machine-dependent, meaning it is specific to a particular processor type and its associated instruction set.

Machine code, also called **native code**, will have a form similar to the following, expressed in hexadecimal:

```
Adresse   Code
0060      B80400
0063      890424
0066      B80500
0069      890C24
006C      8B0424
006F      83C014
0072      890424
```

To make this code more readable, it can be expressed using mnemonics, as in the following program:

```
MOV AX, 0004H       Load the value 4 into AX
MOV [ESP], AX       Store AX at the address pointed to by ESP (the stack)
MOV AX, 0005H       Load the value 5 into AX
MOV [ESP+4], AX     Store AX at the address ESP + 4
MOV AX, [ESP]       Load into AX the value at the address ESP
ADD AX, 14H         Add 20 (14H) to AX
MOV [ESP], AX       Save the result at the address ESP
```

Where AX is the Accumulator Register and ESP is the Stack Pointer Register.

V.3.1 Control Flow Graphs (CFG)

Before generating final machine code, the compiler repeatedly analyzes the intermediate code (usually in three-address or quadruple format). One of the key steps in this process is building a **Control Flow Graph (CFG).** A **CFG** is a directed graph where nodes are individual basic blocks and edges are possible paths of execution flow, such as jumps and conditionals. The CFG aims to optimize the code generated by minimizing memory accesses, using registers effectively, and reducing unnecessary jumps or operations.

A **basic block** is a straight-line code sequence with one entry point (the first instruction) and one exit point (the last instruction). It does not contain branches and labels (jumps). Once execution enters a basic block, it continues to the end without interruption. In order to identify basic blocks, we need to identify "leaders" or instructions that start a basic block, especially : the first instruction in the program, the target of any jump (branch instruction) and the instruction immediately after a jump. To build blocks we have to start by leaders that begin a new block,

and to include all following instructions until the next leader or end of the program.

Example: Le the following C code

```
max = a[0];
i = 1;
while (i < 10) {
    if (a[i] > max) {
        max = a[i];
    }
    i++;
}
```

The equivalent Three-Address Code is:

```
1.  max := a[0]
2.  i := 1
3.  t1 := i < 10
4.  if False t1 goto 12
5.  t2 := 4 * i
6.  t3 := a[t2]
7.  t4 := max
8.  t5 := t3 > t4
9.  if False t5 goto 10
10. max := t3
11. i := i + 1
12. goto 3
13. // end
```

The Basic Blocks from this Code are:

Block B1:

```
1.  max := a[0]
2.  i := 1
3.  t1 := i < 10
4.  if False t1 goto 13
```

Block B2:

```
5.  t2 := 4 * i
6.  t3 := a[t2]
7.  t4 := max
8.  t5 := t3 > t4
9.  ifFalse t5 goto 11
```

```
10. max := t3
```

Block B3:

```
11. i := i + 1
12. goto 3
```

Block B4:

```
13. // end of loop
```

Control Flow Graph (CFG):

- B1 → B2 → B3 → B1 (loop back)
- B1 → B4 (exit when i >= 10)

Here, B1 initializes max and i and checks the loop condition. B2 compares the current element with max and updates if needed. B3 increments i and loops back and B4 marks the exit point when the loop ends. This structure enables optimizations like minimizing repeated memory accesses and improving branch prediction.

V.3.2 Transformations on Basic Blocks

The goal of transforming a basic block is to optimize execution efficiency to produce optimized machine code. Transformations can be based for instance on **structure-preserving transformations** and **algebraic transformations.**

Structure-Preserving Transformations

These optimizations maintain the original control flow and logic of the program but improve performance or reduce resource usage. This consists for instance in eliminating: repetitive instructions such as those based on the same computing result, dead code or expressions that are never executed, unused local variables.

Example:

The following code:

```
t1 = a + b
t2 = a + b
t3 = t1 * c
```

can be transformed to:

```
t1 = a + b
```

```
t2 = t1
t3 = t1 * c
```

These transformations are key to optimizing compiler output without changing the logical behavior of the code.

Algebraic Transformations

Algebraic transformations involve modifying the structure of expressions within a basic block to produce simpler code. These optimizations focus on mathematical simplification and the replacement of costly operations.

For example, expressions that involve constants known at compile time can be evaluated during compilation to reduce runtime calculations. The following expressions:

```
x = 10 + 0;
y = 3 * 1;
```

Can be transformed for example to

```
x = 10;
y = 3;
```

Furthermore, constant expressions can be precomputed like in these instructions:

```
z = 7 * 8;
w = b * (100 + 200);
```

that can be transformed as follows:

```
z = 56;
w = b * 300;
```

Algebraic transformations also consists in using **less expensive operators** by using for instance multiplication or bit shifting instead of exponentiation or division provided the outcome For example, let the following expressions:

```
x = y * y; // equivalent to y squared
z = a / 2; // division is slower
```

They can be transformed to:

```
x = y << 1;    // uses bit shift for multiplication by 2 (if appropriate)
z = a >> 1; // uses bit shift for division by 2 (only with integers)
```

V.3.3 A Simple Code Generator

Among the common methods of generation of machine code from intermediate representations (specifically quadruples) for each basic block of a control flow graph is the simple code generation method. The main objective of code generation is to produce efficient, low-cost

machine code while prioritizing register use over memory where possible and optimizing register allocation.
This method relies on tracking variable usage within a basic block in order to avoid storing values that won't be reused and reuse registers effectively. To collect this information, the compiler performs a backward analysis of the basic block:

1. Traverse the basic block in reverse order (from last instruction to first).
2. For each quadruple of the form A := B op C:
    - Attach to the quadruple the current usage info of A, B, and C from the symbol table.
    - Update the symbol table:
        - Mark A as not active with no next use (it is being overwritten).
        - Mark B and C as active, with their next use at this current instruction.
          Certainly! Here's a **reformulated version** of the content, using clearer language and replacing the example with an **equivalent but different one**, all presented in a unified paragraph without sub-sections:

During the code generation phase, two key data structures help manage the allocation and use of registers and memory locations: register descriptors and address descriptors.

The register descriptor tracks what each register currently holds. Initially, all registers are considered empty. As machine code is generated, this descriptor is updated to reflect which variables are currently stored in which registers. Similarly, the address descriptor tracks where the current value of each variable in a basic block can be found in a register, in memory, or both. These descriptors are often stored or referenced via the symbol table to inform decisions during code generation.

For example, consider the expression x = (p + q) * (r - q). The corresponding intermediate code (in quadruple form) can be:

```
t1 := p + q
t2 := r - q
t3 := t1 * t2
x := t3
```

As the code generator processes each instruction:

For t1 := p + q, it loads p into register R1, adds q, and stores the result in R1. The register descriptor is updated to reflect that R1 contains t1, and the address descriptor notes that t1 is currently only in R1.

For t2 := r - q, r is loaded into R2, q is subtracted, and R2 now contains t2. The descriptors are updated accordingly.

For t3 := t1 * t2, the generator multiplies the values in R1 and R2 (which contain t1 and t2), placing the result in, say, R3. Now, R3 holds t3.

Finally, x := t3 moves the value in R3 to the memory location for x, and both descriptors record that x is in R3 and in memory.

This system ensures that variables are not needlessly reloaded or stored multiple times, optimizing the use of registers and memory. The final machine code is generated by scanning the basic block from start to end and translating each quadruple using the current state of the descriptors to guide instruction selection and register allocation.

REFERENCES

- Alfred Aho, Ravi Sethi et Jeffrey Ullman, Compilers, Principles techniques and tools, Addison Wesley 1986, ISBN: 0201100886 (available in UMAB library)
- Sami Ait-Aoudia. Compilation: cours et exercices corrigés. Office des publications universitaires, Algérie 2012.
- John Levine, Flex and Bison, O'Reillly, 2009, ISBN 0596155972.
- Keith D. Cooper, Linda Torczon, Engineering a Compiler Morgan Kaufmann publisher, 3rd Edition, 2022, ISBN: 9780128154120.
- Nora Sandler, Writing a C Compiler: Build a Real Programming Language from Scratch, No Starch Press, online book, 2024, ISBN. 1718500424