GILLES GEERAERTS, GUILLERMO A. PEREZ

INTRODUCTION TO LANGUAGE THEORY AND COMPILING

Typeset under LaTeX using the TUFTE-LATEX document class.

The authors would like to thank the following persons for proofreading (part of) those notes and making meaningful suggestions and comments:

- Mourad AKANDOUCH
- Benoît HAAL
- Zakaria JAMIAI
- Franklin LAURELES
- Lucas LEFÈVRE
- Benjamin MONMEGE
- Marie VAN DEN BOGAARD.

Fifth version, September 2023

Contents

List of Figures

1 Introduction

1.1 What is a language?

THE NOTION OF LANGUAGE is obviously most important to humans. It is beyond the scope of these lecture notes to give an exhaustive definition of this notion, but, we would like to highlight several features of *natural languages*, to help us build intuitions that will be useful when introducing the basic definitions of *formal language theory*, the main topic of these notes.

The Concise Oxford Dictionary[1] defines a language as:

> A vocabulary and way of using it prevalent in one or more countries.

[1] H.W. Fowler, J.B. Sykes, and F.G. Fowler. *The Concise Oxford dictionary of current English*. Clarendon Press, 1976

The Merriam-Webster Dictionary[2] is more explicit:

> The system of words or signs that people use to express thoughts and feelings to each other
>
> [. . .]
>
> The words, their pronunciation, and the methods of combining them used and understood by a community.

[2] "Language." Merriam-Webster.com. Accessed August 2, 2014. `http://www.merriam-webster.com/dictionary/language`.

Both definitions explain that a language is built on top of basic building blocks which are *words* (the set of all words forming a vocabulary), that these words must be combined according to a certain *system of rules* (in order to form sentences), and that the purpose of these combinations of words (or sentences) is to carry a certain *meaning*. Hence, two important concepts pertaining to languages are the *form* and the *meaning*.

The *form* can be loosely defined as the set of rules that govern the making of a sentence in a given language. We will rather refer to form as the *syntax* of the language. For a natural language, such as English, it starts with the alphabet, which is the so-called Latin alphabet, on top of which *spelling rules* indicate which words are correct or not. Then, those words can be used to form sentences according to syntactic and grammatical rules.

Syntax is not only relevant to natural languages, but also for formal languages such as *programming languages*. Syntactically correct programs only can be run by a computer, and the first duty of a *compiler* is a syntax check.

[3] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall software series. Prentice Hall, 1988

Example 1.1. Listing 1.1 shows a syntactically correct C program[3]. Deleting the semi-colon from the end of line 5 triggers a compiler syntax error:

Observe that, although we remove the semi-colon from line 5, the error is reported on line 6. Indeed, the compiler 'realises that the semi-colon is missing only when it reads the `return` statement on line 6.

Listing 1.1: A syntactically correct C program.

```
#include <stdio.h>
int i = 5;

int f(int j) {
  int i = j ;
  return i+1 ;
}

int main() {
  printf("Hello␣world␣!") ;
  printf("%d␣%d", i, f(i+1)) ;
  return 0 ;
}
```

```
C-example.c: In function 'f':
C-example.c:6: error: expected ',' or ';' before 'return'
```

On the other hand, when a sentence is syntactically correct, one can try and make sense out of it, that is, to attach a *meaning* to this sentence, which we refer to as its *semantics*.

The difference between syntax and semantics is important. In natural languages, a famous example to illustrate this difference is due to Chomsky[4]:

Example 1.2. Observe that the following sentence is *grammatical* in English:

Colorless green ideas sleep furiously

because it follows the "Subject + Verb + Complement" basic pattern of English, but is clearly nonsensical. In other words, it is syntactically correct, but semantically incoherent.

The contrast between syntax and semantics is perhaps sharper in the setting of programming languages, where the semantics is supposed to be non-ambiguous. Indeed, every programmer knows how easy it is to write a syntactically correct program that, when run, does not perform the task it was intended for... Also, different sentences in different programming languages will produce the same effect:

Example 1.3. The three following statements, respectively in C, Pascal and COBOL, all sum the contents of variables X and Y and store the result in X.

```
X += Y
```

```
X := X+Y
```

```
ADD Y TO X GIVING X
```

[4] N. Chomsky. *Syntactic Structures*. Mouton and Co, The Hague, 1957

In the case of programming languages, associating a semantics to a given piece of code amounts to produce machine-executable code that, when run, has the intended effect of the source code. This is, of course, the purpose of a compiler. To carry out this task, the compiler must first analyse the structure (i.e., the syntax) of the code, in order to identify keywords, variable names, and so forth.

Formalising the syntax and the semantics of languages is an old endeavour. Dictionaries and grammars for the English language have existed since the seventeenth century[5]. In France, the *Académie Française* has been founded in 1635 with a well-defined mission[6]:

> *La principale fonction de l'Académie sera de travailler, avec tout le soin et toute la diligence possibles, à* ***donner des règles certaines à notre langue*** *et à la rendre pure, éloquente et capable de traiter les arts et les sciences.*
>
> The main mission of the Academy will be to labor with all the care and diligence possible, **to give exact rules to our language**, to render it capable of treating the arts and sciences

To this end, the Academy publishes a famous Dictionary[7] and provides advices on the good usage of the French language[8]

Of course, a non-ambiguous, comprehensive and unique formalisation of a natural language's syntax and semantics seems impossible. For instance, these notes try to adhere to the so-called *Oxford Spelling*[9] where *analyse* and *behaviour* are correct spellings for words that would otherwise be spelled *analyze* and *behavior* in the United States. Also, the exact accepted meaning of a single term is subject to local variations.

On the other hand, formal and mathematically precise definitions of the syntax of semantics of programming languages are both desirable, and, hopefully, feasible. Indeed, the syntax of a programming language should:

1. be simple enough that a programmer does not need to refer to a set of rules too often when producing code.

2. offer a clear structure, so that the code is easily readable and maintainable (for instance, the use of functions, blocks, and so forth).

3. be analysable automatically (by means of a program), otherwise no compiler, no syntax highlighting tool,. . . would be possible.

Finally, the semantics of a programming language should be clearly defined, and non-ambiguous. Otherwise, the program might not have the effect that the programmer had in mind when writing the code, or the machine code produced by different compilers from the same source code might have different effects when run on the same machine.

This short discussion clearly shows the need for a theory of *formal languages*, at least for compiler design, the main application we target in these notes.

1.2 Formal languages

In this section, we give the basics of formal language theory.

[5] See http://en.wikipedia.org/wiki/History_of_English_grammars and http://en.wikipedia.org/wiki/Dictionary#English_Dictionaries.

[6] Article 24 of the status of the Academy: http://www.academie-francaise.fr/linstitution/les-missions

[7] The writing of the ninth edition is ongoing. The eight edition is available on-line http://atilf.atilf.fr/academie.htm.

[8] See for instance, the list given on: http://www.academie-francaise.fr/la-langue-francaise/questions-de-langue.

[9] R.M. Ritter. *The Oxford Guide to Style*. Language Reference Series. Oxford University Press, 2002

1.2.1 Basic definitions: alphabet, word, language

Let us start with several basic definitions. We first give the definitions, then comment on them.

Definition 1.4 (Alphabet)**.** An *alphabet* is a *finite* set of symbols. We will usually denote alphabets by Σ.

Intuitively, an alphabet is the set of symbols that we are allowed to use to build words and sentences. Because we expect our formal languages to be processed automatically, it is natural to ask that the alphabet be finite.

An intuitive justification to the *finite alphabet* is that, in real computers, each memory cell can hold only a finite amount of information (8, 32, 64 bits, for instance).

Example 1.5. The set

$$\{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z\}$$

is an alphabet. The set $\{\$,\#,!\}$ is an alphabet too. A classical alphabet in the setting of computer science is $\{0,1\}$. On the other hand, the set of natural number $\mathbb{N}$ for instance, does not qualify as an alphabet because it is not finite.

Definition 1.6 (Word)**.** A *word* on an alphabet Σ is a *finite* (and possibly empty) sequence of symbols from Σ. We use the symbol ε to denote the *empty word*, i.e., the empty sequence (that contains no symbol).

We usually denote words by u, v, or w. We use subscripts to denote the sequence of symbols making up the word, for instance $w = w_1 w_2 \cdots w_n$ indicates that the word w is the sequence of symbols w_1, w_2,..., w_n. In these notes, we will often use the term 'string' instead of 'word', because a sequence of characters is referred to as a 'string' in many programming languages such as C.

Example 1.7. Let Σ_C be the alphabet:

$$\{\mathtt{a},\mathtt{b},\ldots,\mathtt{z},\mathtt{A},\mathtt{B},\ldots,\mathtt{Z},\mathtt{0},\mathtt{1},\ldots,\mathtt{9},\_\}$$

of all characters which are allowed in C variable names. Then, all words on Σ_C that do not begin with a number and are not C keywords are valid variable name in C (assuming no limit is imposed on the length of variable names).

We denote by $|w|$ the *length* of a word w, i.e. the number of characters that it contains. By convention, $|\varepsilon| = 0$. For all i s.t. $1 \le i \le n$, the word $w_1 \cdots w_i$ (made up of the i first characters of w) is called a *prefix of* w and the word $w_i \cdots w_n$ is called a *suffix of* w.

Were we dealing with natural languages, the next definition would probably be that of a *sentence*, and would be something like: 'a finite sequence of words, separated by spaces and punctuation marks, and ending by a dot'. However, such a definition would give a special status to certain symbols (the 'space' and the punctuation symbols), which we have tried to avoid by giving a very general notion of alphabet, where 'space' and punctuation marks can be considered as regular symbols. Actually, for our purpose, such a distinction is useless, so we can directly define the notion of *language*:

Definition 1.8 (Language)**.** A *language* on an alphabet Σ is a (possibly empty or infinite) set of words on Σ.

It is easy to confuse ε, $\{\varepsilon\}$ and $\emptyset$. The first is a *word* that contains no symbol. The second is a *language* which is not empty since it contains the word ε. The last is a language too (empty).

Since a language is a set, we denote the *empty language* by the usual empty set symbol $\emptyset$.

Example 1.9. Here are some examples of languages showing that this definition allows one to capture several interesting problems:

1. The set L_{Cid} of all non-empty words on Σ_C (see example 1.7) that do not begin with a digit, is a language. It contains all valid C identifiers (variable names, function names, etc) and all C keywords (`for`, `while`, etc).

2. The set L_{odd} of all non-empty words on $\{0,1\}$ that end with a 1 is a language. It contains all the binary encodings of odd numbers.

3. Similarly to the previous example, the set $L_{()}$ of all words on $\Sigma = \{(,)\}$ which are well-parenthesised, i.e., s.t. each closing parenthesis matches a previously open and still pending parenthesis, and each open parenthesis is eventually closed. For example $(()()) \in L_{()}$, but neither $)(($ nor $(()$ do.

   This language is also known as the *Dyck language*, named after the German mathematician Walter VON DYCK (1856–† 1934). It is mainly of theoretical interest: we will rely on it several times later to discuss the kind of formalism we need to recognise languages of expressions that contain parenthesis, such as the language L_{alg} defined in the next item:

4. The set L_{alg} of all algebraic expressions that use only the x variable, the $+$ and $*$ operators and parenthesis, and which are well-parenthesised, is a language on the alphabet $\Sigma = \{(,),x,+,*\}$. For instance $((x+x)*x)+x$ belongs to this language, while $)(x+x$ does not, although it is a word on Σ.

5. The set L_C of all syntactically correct C programs is a language.

6. The set L_{Cterm} of syntactically C programs that terminate whatever the input given by the user is a language.

All these examples are more or less related to the field of compiler design, but we will provide examples from other fields of application later.

1.2.2 The membership problem

Considering these examples, it is clear that a very natural problem is to test whether a word w belongs to a given language L:

Problem 1.10 (Membership)**.** Given a language L and a word w, say whether $w \in L$.

Being able to answer such a question in general (i.e., for all languages L) seems to solve meaningful questions. Let us come back to our examples to illustrate this. In the first case, testing whether $w \in L_{Cid}$ allows one to check that w is a valid C identifier. Testing whether a binary number belongs to L_{odd} allows one to check whether it is odd or even. The membership problem for L_{alg} and L_C amount to checking the syntax of expressions and C programs respectively, a task that is important when compiling. Observe that all these criteria are purely syntactical. On the other hand, the last example, L_{Cterm} seems more complex, because the criterion for a word w to belong to L_{Cterm} is that w is string encoding a terminating C program, i.e., a *semantic criterion*, yet the definition of L_{Cterm} makes perfectly sense and is mathematically sound.

Of course, we are particularly interested in solving the membership problem *automatically*. What we mean there is that, given a language L, we want to build a program that, reads on its input *any word* w, and returns '*yes*' iff $w \in L$.

What we will do mainly in these notes is to develop formal tools to provide an answer to that question. Let us already try and build some intuitions by highlighting characteristics of programs that would recognise each of those languages. In all the cases, we assume that the word for which we want to test membership is read character by character, from left to right, i.e., if $w = w_1 w_2 \cdots w_n$, then the program will first read w_1, then w_2, and so forth up to w_n. When w_n is read, the program must output its answer.

1. In the case of L_{Cid}, the program must check that $w_1 \in \{a, b, \ldots, z, A, B, \ldots, Z\}$, then that all subsequent characters are in Σ_C. Observe that this program needs only one bit of memory to operate, as it only needs to remember whether the character it is currently reading is the first one or not.

2. In the case of L_{odd}, the program must only check that all characters it receives are 0's and 1's, and that the last one is 1. This does not even require any memory.

3. The case of $L_{()}$ is a bit more difficult, because reading a single parenthesis is not sufficient to tell whether it is correct or not. Now, the program needs to *remember* an information about the parenthesis it has met along the prefix read so far.

   More concretely, to check whether an expression is well-parenthesised, we can write a program using a counter c that:

   - increments c each time it reads an opening parenthesis.
   - checks that $c > 0$ each time it reads a closing parenthesis, decrements c if it is the case and returns '*no*' otherwise.
   - returns '*yes*' at the end of the word if and only if $c = 0$.

   It is easy to check that, at all times, c contains the number of pending open parenthesis. Observe that, since we have fixed no bound on the *lengths* of the words in $L_{()}$, the values the program needs to store can be arbitrarily large.

Observe that the memory needed to test membership of a word to the languages of cases 1 and 2 is *finite*, while it is *unbounded* in case 3. It seems difficult to make a program to recognise $L_{()}$ that uses a bounded memory only. Indeed, we will give, in Chapter 3 formal arguments showing that it is not possible: $L_{()}$ is a so-called *context-free language*, while L_{Cid} and L_{odd} belong to the 'easier' class of *regular languages*.

4. Checking whether a string is a correct algebraic expression is at least as hard as recognising $L_{()}$. In addition to checking that the expression is well-parenthesised, one must also check each operation $+$ or $*$ has exactly two well-formed operands, which can, in turn, be complex expressions. This suggests that a *recursive approach* might be needed, where the base cases are simple expressions containing only one variable, or one constant. In Chapters 5 and 6, we will develop techniques to generate such recursive programs (called *parsers*) that can answer the membership program for a broad class of languages to which L_{alg} belongs.

5. Checking the syntax of a program written in a high-level language, such as C, is part of what a compiler should do, and techniques to do so will be thoroughly presented in these notes. Let us note however, that the membership problem for $w \in L_C$ is at least as hard as the membership for L_{Cid}, $L_{()}$ and L_{alg} as a C program can obviously contain C identifiers and arithmetic expressions. Also, we must check that curly brackets (`{`, `}`), used to delimit blocks, match; that each `else` corresponds to an `if`, etc.

6. The last language L_{Cterm} models a very hard question: 'can we write a program that tests whether any program written in a given language terminate?' Obviously, such a *termination tester* would be an invaluable tool for all developers, who have already struggled with unexpected infinite loops. Unfortunately, the answer to that question is negative: such a program does not exist. This is proved formally in a 'Computability and Complexity' course.

When we write 'such a program does not exist', we do not mean 'is not known yet', but rather that there is a provable mathematical impossibility to the existence of such a program. This shows that, as surprising as it may seem, there are (natural, meaningful and well-defined) problems that *cannot be solved* by a computer!

In these notes, we will present the mathematical and practical tools that are necessary to attack those questions.

1.3 Application: compiler design

It should now be clear from the previous examples that a first class application to formal language theory is the design of compilers.

A *compiler*[10] is a program that *processes programs* and *translates* a program P_s (the source program, or source code) written in a language L_s (the source language) into an *equivalent* program P_t (the target program) written in a language L_t (the target language). The compiler, being a program itself, might be written in a third language. As an example, `gfortran`[11] is a compiler that translates FORTRAN code into (for instance) Intel i386 machine code, and is written in C.

Remark that here, we are using the term 'language' with the same meaning as in 'programming language', and not in the formal sense of Definition 1.8.

[10] A. Aho, M. Lam, R. Sethi, and Ullman J. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007

[11] See the `gfortran` home page at: `https://gcc.gnu.org/fortran/`.

1.3.1 Anatomy of a compiler

In order to perform its translation, a compiler proceeds in several steps that we detail now. Those steps can be split in two successive parts:

1. First, the *analysis phase* builds an abstract representation of the program structure. This phase consists in first performing a *lexical analysis*, or *scanning*; then a *syntactic analysis*, or *parsing* (more details be-

low). Syntax errors are detected and reported during this phase. Finally, the analysis part usually contain a *semantic analysis*, that performs, a.o., type checking, and reports typing errors.

2. Second, the *synthesis phase* translates the abstract representation of the program into the target language. Several optimisations can occur during this phase.

Let now detail these different steps, by referring to Listing 1.1.

1.3.2 *Scanning*

When teaching a new programming language to someone, one usually starts by introducing the basic blocks from which a program is built: identifiers for variable and function names; (reserved) keywords; special signs such as `{`, `}` or `;`, and so forth. The first task of a compiler is thus to split the input string into a sequence of meaningful sub-strings that will be passed to the next step of analysis. Roughly speaking, this will be the work of the *scanner*. An illustration of the effect of the scanner on the code in Listing 1.1 is shown hereunder[12]. Each gray box delineates one of the sub-strings that the scanner should identify:

The scanner is often called '*lexical analyser*'. In these notes, we will indifferently use both expressions. Some authors, however, consider that *scanning* and *lexical analysis* are different processes: scanning consists only in dividing the input into relevant sub-strings, while lexical analysis also performs the tokenisation (see hereunder for a definition of token).

```
int i = 5 ;


int f ( int j ) {
    int i = j ;
    return i + 1 ;
}


int main ( ) {
    printf ( "Hello␣World␣!" ) ;
    printf ( "%d␣%d" , i , f ( i + 1 ) ) ;
    return 0 ;
}
```

Observe how white spaces are ignored in this example. Here, we use the term 'white space' in a broad sense: it also includes tabulation characters or end-of-line. Those white space symbols are relevant to the compiler only to separate successive sub-strings[13]. Indeed, the two following code excerpts have the same effect:

```
int i = 5 ;
```

```
                      int


            i


    =
```

[12] We have skipped the first line `#include <stdio.h>` because this line is actually never part of the input of the *compiler*. Before being compiled, C code is processed by a pre-processor, that handles so-called *pragma directives*, i.e. those keywords that begin with `#`. In particular, handling `#include` directives amounts to replacing them by the content of the file that is included. See https://gcc.gnu.org/onlinedocs/cpp/Pragmas.html for further reference.

[13] This is the case in most programming languages. A notable exception is the (prank) programming language Whitespace, where '*Any non white space characters are ignored; only spaces, tabs and newlines are considered syntax*'. See http://compsoc.dur.ac.uk/whitespace/.

```
        5
                  ;
```

However, the scanner does not only split the input in a sequence of sub-strings as illustrated above, but also performs a preliminary analysis of those sub-strings and determine their type. For instance, what matters about the `j` sub-string in lines 4 and 5 is not the `j` character, but rather (1) the fact that `j` is identified as a *variable identifier* and (2) the fact that the *same identifier* occurs in lines 4 and 5 but not in other lines where variable identifiers appear (indeed, replacing the two occurrences of `j` in those lines by `LukeIAmYourFather`, or any other legal variable name will yield a compiled code with exactly the same effect). Also, reserved keywords (`while`, `if`,...), operators (=, <=, `!=`,...) and special symbols (`{`, `;`, ...) can be identified as such.

To sum up the role of the lexical analyser is not only to split the input into a sequence of sub-strings, but to relate each of those sub-strings to its *lexical unit*. A *lexical unit* is an abstract family of sub-strings, or, in other words, a *language*, that corresponds to a peculiar feature of the language. The definition of lexical units is a bit arbitrary and depends on the next steps of the compiling process. For instance, for the C language, we could have as lexical units:

- identifiers
- keywords
- ...

or we could refine this list of lexical units and have:

- identifiers,
- the `while` keyword,
- the `for` keyword,
- ...

where each keyword is its own lexical unit. It should be clear that each lexical unit in the lists above corresponds to a set of words, i.e., a language. It is common practice to associate a unique symbolic name to each lexical unit, for instance a natural number, or a name such as 'identifier'. As we are about to see, those values will constitute a part of the scanner's return values.

We are now ready to introduce several definitions that somehow formalise the discussion about scanning we have had so far [14]:

[14] A. Aho, M. Lam, R. Sethi, and Ullman J. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007

Definition 1.11 (Lexeme)**.** A *lexeme* is an element of a lexical unit.

Recall that a lexical unit is a language, this is why it makes sense to speak about 'an element of a lexical unit'. A lexeme is thus one of the sub-strings of the input that has been recognised (or *matched*) by the lexical analyser.

Definition 1.12 (Token)**.** A *token* is a pair (id, att), where *id* is the identifier of a lexical unit, and *att* is an *attribute*, i.e. an additional piece of information about the token.

Tokens are what the scanner actually returns and provides to the next step of the compiling process. The attribute part of the token is optional: it can be used to provide more information about the token, but this is sometimes not needed.

A typical use of the attribute occurs when the matched lexeme is an identifier. In this case, the scanner must check whether this identifier has been matched before, and, if it is the case, to return a piece of information that links all occurrences of the same identifier throughout the code. The scanner achieves this by maintaining, at all times, a so-called *symbol table*.

Roughly speaking the symbol table records, at all times, all the identifiers that the scanner has met so far. Whenever the scanner matches a new lexeme which is an identifier, it looks it up in the symbol table. If the lexeme is not found, the scanner inserts it in the table. Then, the index of the lexeme in the table can be used as a unique symbolic name for this lexeme, which can be put in the attribute part of the token that the scanner returns.

Example 1.13. Let us consider the simple code excerpt:

```
1   int i = 5;
2   int j = 3 ;
3   i = 9 ;
```

Initially, the symbol table is empty. When the lexeme `i` in line 1 is matched, the scanner inserts it into the first entry (index 0) of the symbol table, and returns the token $(\mathsf{identifier}, 0)$. Here, $\mathsf{identifier}$ denotes the symbolic name for identifiers, and we have used the index of the lexeme in the symbol table as the attribute of the token, which is a unique symbolic name for the identifier `i`. When `j` in line 2 is matched, it is inserted into entry number 1 of the symbol table, and the token $(\mathsf{identifier}, 1)$ is now returned. So far, the symbol table has the following content:

index	lexeme
0	`i`
1	`j`

Then, when `i` is matched in line 3, it is found in index 0 of the symbol table, and the scanner returns again $(\mathsf{identifier}, 0)$, thereby indicating to the parser that this identifier is the same as the one that has been matched in line 1. The symbol table is not altered by this step.

Observe that this technique is not completely satisfactory: if we want to apply it to the example of Listing 1.1, we will run into trouble when matching the `i` in line 5, which will be wrongly identified as the identifier of the same variable as the one declared in line 2. The problem with the way to handle symbol table we have described above is that it does not allow to cope with *scoping*. This is, however, hard to achieve at the level of the scanner which has no *global* view on the input code, and cannot tell, for instance, a variable declaration from a variable use. When the symbol table should accommodate scoping, we will defer its creation to the *parser* (see next section). The technique we have described so far, however, works well when no scoping is required, in simple programming languages for instance.

Let us mention another possible use of the symbol table: it can also be exploited to match keywords, and avoid that keywords are used as identifiers. This is achieved by initialising the symbol table with all possible keywords in the first entries of the table. This allows one to treat keywords in a similar fashion to identifiers, which often makes the scanner easier to implement.

Example 1.14. For instance, assume we are building a scanner for a language with three keywords: `while`, `for` and `if`. We initialise the symbol table this way:

index	lexeme
0	`while`
1	`for`
2	`if`

Thus, keywords are present in lines 0–2 of the table, and identifiers will be inserted in the following lines. Assume the scanner matches the lexeme `abc`. It will be compared to all line in the symbol table, and inserted since it is not present:

index	lexeme
0	`while`
1	`for`
2	`if`
3	`abc`

The scanner returns $(\mathsf{identifier}, 3)$, which means that a genuine identifier has been matched, since the index 3 is not among the lines 0–2 that are devoted to keywords. Now, assume the scanner matches `for`, which could be an identifier since the lexeme contains only letters. The scanner will find this lexeme in line 1 of the symbol table and return: $(\mathsf{identifier}, 1)$. Since now the attribute of the token is ≤ 2, the parser can identify this token as the keyword `for`.

<u>The scanner: summing up</u>

The scanner is the first part of the compiling process. Its role is to split the input into a sequence of *lexemes* (Definition 1.11) that are associated to *lexical units* and returns a sequence of *tokens* (Definition 1.12). It can be responsible for inserting identifiers into the *symbol table*, that contains all identifier matched so far, and possibly all keywords. The symbol table is thus used as a communication medium between the different compiling phases.

Now that we have a clear view of *what* the scanner should do, it remains to explain *how* to do it. Namely, we need to answer the following questions:

1. How to *specify* lexical units? So far, we have used vague English descriptions, like: 'all words starting with a letter and followed by an arbitrary number of letters and digits'. This is clearly not satisfactory. In Chapter 2, we will introduce *regular expressions* to this end.

2. How to *build* in a systematic ways, programs that match lexemes against the description of lexical units? In Chapter 2 again, we will introduce the central notion of *finite automaton* to this end.

1.3.3 Parsing

It should now be clear that the duty of the scanner is to perform a *local* analysis of the code: to match a lexeme against a lexical unit, the scanner analyses a sequence of contiguous characters in the code. Such a local analysis is not sufficient to analyse all features of programming language. That for instance the matching of parenthesis in arithmetic expressions, like:

```
( ( x + y ) * 3 )
```

Checking that the first (opening) parenthesis matches the last (closing) one clearly requests a *global view* on the piece of code under analysis.

Building (and, to some extent, analysing) such a global and *abstract* representation of the code is the task of the *parser*. To help us build an intuition of what such an abstract representation could be, let us consider the restricted case of arithmetic expressions that can contain: (1) parenthesis (possibly nested); (2) identifiers (i.e., variable names); (3) natural numbers; (4) the `+`, `-`, `/` and `*` binary operators; (5) the `-` unary operator. A textual description of what 'a syntactically correct expression' is, could be given in an inductive fashion. For instance, a word is a correct expression if and only if it has one of the following forms:

A binary operator is one that has two arguments, such the '$-$' operator in $5-3$, where the two arguments are 5 and 3. A unary operator has only one argument, such as '$-$' in the expression -5.

1. an identifier. For instance, `BeamMeUpScotty` is correct;

2. a natural number. For instance `42` is correct;

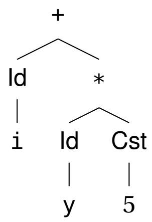
3. '(E)', where E is a correct expression. For instance, `(BeamMeUpScotty)` is correct;

4. '$-E$', where E is a correct expression. For instance `-42` is correct;

5. E_1 op E_2, where E_1 and E_2 are correct expressions, and op is either `+` or `-` or `*` or `/`. For instance, `-42+(BeamMeUpScotty)` is correct.

The abstract syntax tree In addition to checking whether a program is syntactically correct, the parser must also build some kind of formal object that represents the structure of the input. This object will be exploited by the next steps of the compiling process.

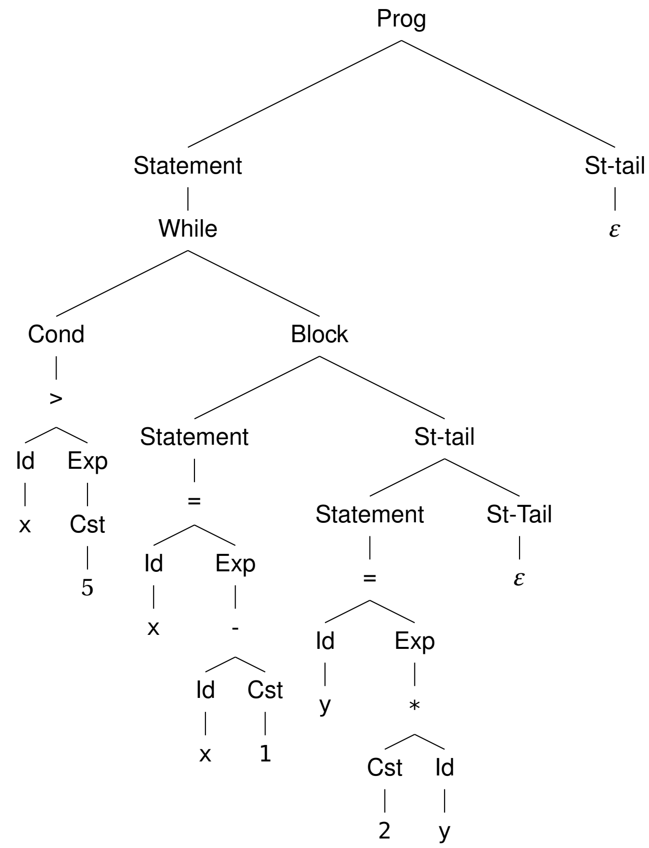
The most usual object that parsers build is the *abstract syntax tree* (AST for short). As the name indicates, this object is a tree that reflects the nesting of the different programming constructs. As an example, the AST of the expression `i+y*5` could be:



Similary, the AST of the following C `while` loop:

```
while(x>5) {
 x = x-1 ;
 y = 2*y ;
}
```

could be:



The parser and the symbol table In the previous section, we have discussed the creation of symbol table entries by the scanner, which is a technique that works fine when the compiler must not handle *scoping*, as in Example 1.13. However, in a realistic example such as 1.1, the scope of variables must clearly be taken into account. Indeed, the `i` variable used in line 5 is *not* the same as the one declared in line 2, because the latter occurs in the global scope, while the former lies in the block containing the code of function `f()`.

To cope with the scoping of identifier names, the compiler can manage *several* symbol tables, one for each scope, that contains all the identifiers from the scope. Since the scanner has no global view on the code and can hardly detect scopes, we ask the parser the populate these symbol tables. All the scanner does is to return an information indicating that it has recognised an identifier, together with the name of the identifier.

All those symbol tables are arranged in a *tree*, in order to reflect the nesting of scopes (see example below). When the parser obtains from the scanner a token corresponding to an identifier whose name is v, it looks up v in several symbol tables: first, the current symbol table, then—if the identifier has not been found—its father, and so forth, up to the root that corresponds to the symbol table of the global scope. This is illustrated in the following example:

Example 1.15. Let us come back once again to the code excerpt of Listing 1.1. Initially, an empty symbol table T_0 is created for the global scope. Then, the parsing goes, a.o. through the following steps (we focus on the handling of identifiers):

1. When the lexeme `i` is matched on line 2, the scanner returns the name `i` to the parser, which looks it up in the current (and only) symbol table T_0. Since T_0 is still empty, `i` is inserted into T_0:

   T_0

   Current $\longrightarrow$

index	lexeme
0	`i`

   In order to identify uniquely each variable[15], the parser can associate it with a pair (T, i), where T is the name of a symbol table, and i is the line in the symbol table. In this case, our variable `i` would be identified by $(T_0, 0)$.

2. Then, when reaching line 4, the parser detects the declaration of a new function, and thus creates a new scope. Concretely, this amounts to creating a new symbol table T_1, which is inserted in the tree of symbol tables as a son of T_0:

   T_0

index	lexeme
0	`i`

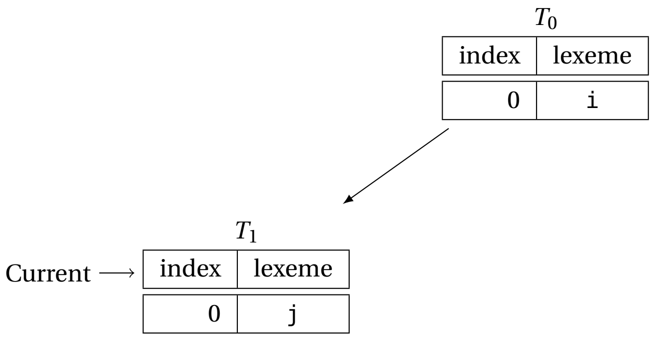
   T_1

   Current $\longrightarrow$

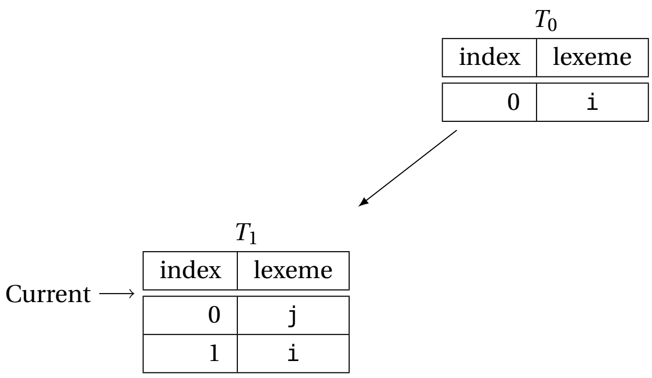
index	lexeme

3. Then, the parser can insert the declaration of `j` (the parameter of `f`) as a new variable in T_1:

[15] This will clearly be necessary when *generating code*.



4. Next, moving to line 5, the parser first inserts a fresh `i` variable, but this time in T_1. As we will see, it will predate variable `i` in T_0 within the scope of function `f`:

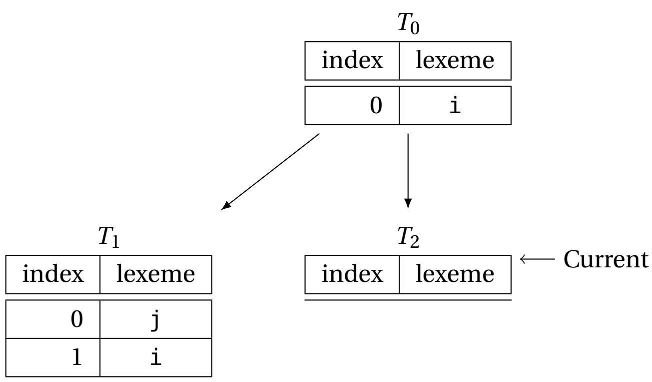


5. On the same line, the parser detects the use of variable `j`. It first looks up `j` in T_1 (which is the current symbol table) and finds it. Thus, this occurrence of `j` will be identified with $(T_1, 0)$.

6. Then, the parser finishes parsing `f`, and detects the use of variable named `i` in line 6. It first looks up in T_1, and finds an occurrence of `i`. Thus, it identifies this `i` with $(T_1, 1)$, i.e., the same `i` as in line 5.

7. When leaving the scope of `f`, the parser changes the current symbol table to T_0. Note however that T_1 is kept in memory for the next steps of compiling.

8. In line 9, the parser detects a new scope and inserts a new symbol table T_2 as a child of T_0, since it is the current table. T_2 becomes the current table:



9. Finally, in line 11, the parser detects the use of a variable called `i`, and looks it up in the tree, starting from the current symbol table T_2 (which contains no entry), then moving up the tree towards the root. Variable `i` is eventually found in the root T_0, so it is correctly identified with $(T_0, 0)$, i.e., the one which is declared in line 2.

The parser: summing up

The parser is the second part of the compiling process. It receives a sequence of *tokens* from the scanner. Its role is to check whether this sequence of tokens respects a given *syntax*, and, when it is the case, to produce an abstract representation (such as the *abstract syntax tree*) of the program's structure, that will be used by the next phases of the compiling process. The parser can also populate the symbol table, in particular when scoping must be taken into account.

As for the scanner, we can now identify several questions that we must solve in order to build parsers:

1. We have said that the parser must check the *syntax* of its input, but how do we *specify* this syntax? We will see, in Chapter 3 that *grammars* can be used for that purpose, just as we have used *regular expressions* in the case of scanners.

2. How can we build a *parser* from a given grammar, and what kind of machine will it look like? In Chapter 4, we will see that *pushdown automata*—an extension of the finite automata from Chapter 2—are abstract machines that can be used to formalise and build parsers. We will review, in Chapter 5 and Chapter 6, several techniques to build parsers that are efficient in practice.

1.3.4 Semantic analysis

Now that the parser has checked that the syntax of the input code is correct, and has built an abstract representation of this code, it is time to start analysing *what the code means*, in order to prepare its translation. This is the aim of *semantic analysis*. Of course, semantic analysis is highly dependent on the input language, this is why we will stay very general when introducing it. Yet, we can identify several essential points to deal with: scoping, typing, and control flow.

Scoping During the semantic analysis phase, the compiler can analyse the links that exist between the declaration(s) of a name (if any) and the uses of this name throughout the code. For instance, the—syntactically correct—code in Figure 1.1 could raise an error during semantic analysis, because variable `i` is used undeclared (although the name `i` is declared later as an integer variable).

Observe that the control of the scoping can already be performed during parsing, thanks to the symbol table (see previous section).

```c
int main() {
  i = 3 ;
  int i ;
}
```

Figure 1.1: A syntactically correct excerpt of C code that raises an error during semantic analysis.

Type checking and type control Each name (variable name, function name, type name,...) in a program is associated with a *data type* (or, simply, a type) that describes uniquely how this name can be manipulated. During semantic analysis, the compiler determines (if possible) the type of each expression, and checks that the operations on those expressions are consistent with their types. Figure 1.2 shows the typical problems that can occur when compiling an assignment in C. The assignment in line 9 is not problematic, because the type of the right-hand side of the assignment is `int`, which is the same as the type of the variable `j`. Indeed, the sum of the `int` variable `i` and the `int` constant 4 is an `int`. The second assignment (line 10) raises a warning (i.e., a non-blocking error): the right-hand side is a pointer, but assigning a pointer to an `int` is allowed in C, and a conversion is implicitly applied by the compiler. The last assignment (line 11) raises an error: the type of the right-hand side is `struct S`, and the compiler does not know how to convert such an object to an `int`.

In order to manage types, the compiler can add information to the AST that has been built during parsing. This operation of adding information to a tree is called 'decoration'[16]. As an example, Figure 1.3 displays the decorated AST of the C statement `x = sum * 1.5`, where `x` and `sum` are integer variables. Since one of the terms of the `sum * 1.5` product is a `float`, the compiler assigns this type to the expression. This allows it to detect that the result will need to be truncated when copying it to `x` (and perhaps to raise a warning, depending on the compiler and its options). Of course, such information will be crucial when generating the target code for this assignment.

```
1  struct S {int i;} ;
2  int main() {
3    int i, j ;
4    struct S s ;
5    struct S * p = &s ;
6  
7    i = 3 ;
8  
9    j = i + 4 ;
10   j = p ;
11   j = s ;
12 }
```

Figure 1.2: Three syntactically correct assignments with different behaviours of the semantic analyser. The first (line 9) is not problematic. The second (line 10) raises a warning because a pointer is cast to an integer. The last (line 11) is not allowed: no conversion is possible.

[16] Which suggests that ASTs are most probably Christmas trees...

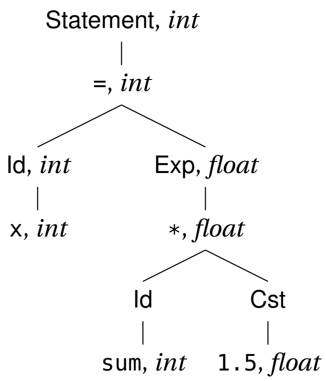


Figure 1.3: A decorated AST with typing information.

Control flow The term 'control flow' refers to the order in which the instructions of a program are executed. Conditionals (`if`), jumps (`goto`), loops (`for`, `while`,...) and function calls can be used to alter the control flow of a program, in an intricate way. As a consequence, it is possible to write syntactically correct programs that contain semantic error related to control flow.

For instance, Figure 1.4 shows a C++ program that does not compile[17] because the `goto` in line 9 jumps inside of the `for`. However, the `i` variable exists only inside the scope of the `for`: it does not exist anymore when reaching the `goto`, so jumping inside the for and executing `std::cout << i` makes no sense: what would be the value of `i`? Other problems can be detected by analysing the control flow of a program, for instance, detecting

```
1  #include <iostream>
2  
3  int main () {
4    for(int i=1;i<10;++i) {
5      infor: std::cout << i ;
6      std::cout << std::endl ;
7    }
8  
9    goto infor ;
10 }
```

Figure 1.4: A C++ program which is syntactically correct, but contains a semantic error: the `goto` bypasses the definition of the `i` variable, which exists only in the scope of the `for`.

[17] On LLVM 6.0.

dead code (code that can never be executed), etc.

To analyse the control flow of a program, the semantic analyser can build an abstract representation thereof, which is the *control flow graph*, a notion introduced by Allen in the early seventies[18]. A control flow graph is a directed graph whose nodes represent the different statements of the program, and whose edges represent the possible jumps in the control flow. So, if a statement s_2 can be executed after a statement s_1 (whether s_2 is on the line next to the line of s_1 or not), there is an edge from s_1 to s_2. It can thus occur that some node has several input edges. A node can also have more than one output edge in the case where this node corresponds to a conditional. In the case of an `if`, for instance, there are two output edges: one for the 'then' block, and one for the 'else'.

[18] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970. ISSN 0362-1340. DOI: 10.1145/390013.808479

Thus, each path in the graph corresponds to one *potential* execution of the program. Of course, some path in the graph will be spurious, and do not correspond to an actual execution of the program. For example, in the case of a `while`, the infinite path that never leaves the `while` will be present in the CFG, but it might be the case (hopefully!) that the loop always terminates[19].

[19] Remember from the Computability and Complexity course that the halting problem is undecidable, hence detecting such spurious infinite loops is not possible in general!

The control flow graph can be built from the AST. Figure 1.5 illustrates this construction for a typical *if* statement of the form **if** B **then** T **else** E, and which is followed by additional statements represented by N. The edges of the AST are shown in gray. Additional edges (dashed) are first inserted in the AST to represent the control flow of the **if**. Similar treatment is applied recursively in the B, E, T and N sub-trees. Then, the edges of the AST are removed, and one obtains the typical diamond shape of the CFG of an **if** statement.

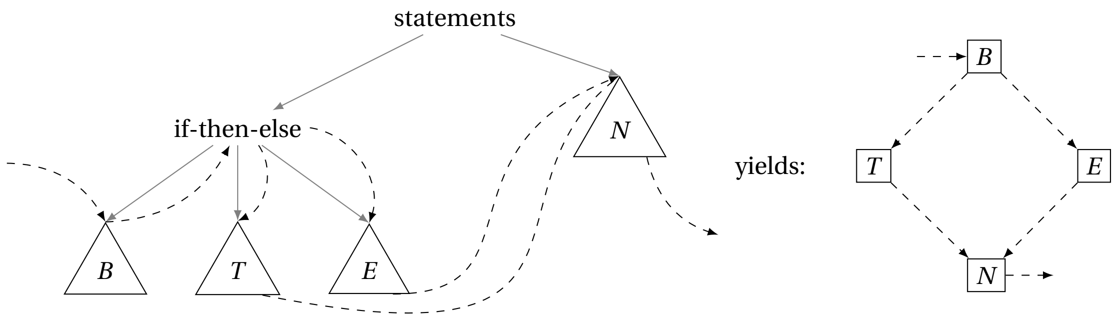


Figure 1.5: The construction of the control flow graph of a typical **if** statement for its AST. B stands for the condition of the **if**; T for the 'then' block; E for the 'else' block; and N for the statements that follow the **if** in the program.

<u>Semantic analysis: summing up</u>

The semantic analysis phase receives the AST built by the parser and performs a (limited) analysis of the semantics of the input code represented by this AST: checking scoping of variables, checking for type consistency and detecting (potentially implicit) type conversions, checking for control flow inconsistency. The output of the semantic analysis phase is a *decorated AST* (and, potentially, a *control flow graph*) that, together with the symbol table built during parsing, contains all the necessary information for the synthesis phase of the compiler.

1.3.5 Synthesis

SYNTHESIS is the phase during which the outcome of the compiler is actually generated. In most cases, this will be an executable program, written in some low-level language such as machine code. However, this is not always the case: these notes, for instance, have been typeset using the $\LaTeX\,2_\varepsilon$ word processing system[20], where one first describes the document using a markup language, and then *compile* this *source* into a PDF file.

[20] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, 1986. ISBN 0-201-15790-X

Again, the actions that should be performed during the synthesis part is highly dependent on the source *and* target languages. So, we will only identify several generalities about synthesis in this section.

The input to the synthesis phase is the decorated AST built during the previous phase of the compiling process. It can also be accompanied by a control flow graph, should it have been built during semantic analysis. They should contain all the necessary information to generate the output code.

Code optimisation Before the output code is actually generated, the compiler might perform several optimisations on the code. Typical optimisations include, but are not limited to:

Control flow optimisation Modifies the control flow graph in order to make the resulting code more efficient. Figure 1.6 shows an example. In the first version of the loop, the conditional `x>2` is tested at each iteration of the loop. However, the loop does not modify `x`, so this condition will be either always true along the loop, or always false. The second version of the loop is therefore more efficient.

Loop optimisation Consists in making loops more efficient, for instance by unravelling them when they are executed a constant number of times.

Constant propagation The compiler can try and detect variables that keep a constant value, and replace the occurrences of these variables by those constant values, thereby avoiding unnecessary and costly memory accesses.

Promotion of parameters to references Function parameters that are passed *by value* and are not modified by the function can be transformed into *references*, to avoid copies of values when calling the function. For instance, in Figure 1.7, the parameter `x` can be promoted to a reference (in C++), and the function's signature becomes `f(struct S &x)`. Then, calling `f` does not request anymore to copy the whole content of the `x` structure.

```
for(int i=0; i<n; ++i) {
  if (x > 2)
    printf("%d", i) ;
  else
    printf("%d", i+1) ;
}
```

Becomes:

```
if (x>2) {
  for(int i=0; i<n; ++i) {
    printf("%d", i) ;
} else {
  for(int i=0; i<n; ++i) {
    printf("%d", i+1) ;
  }
}
```

Figure 1.6: An example of control flow optimisation. The second code excerpt guarantees to test the condition `x > 2` only once.

```
struct S {
  // lots of fields!
} ;

f(struct S x) {
  // no modification of x
}
```

Figure 1.7: An example of a C++ function where the parameter `x` can be safely promoted to a reference to avoid a copy of the whole structure.

Intermediate language Because generating efficient machine code is highly dependent on the target architecture, and might request specialised techniques, it makes sense to introduce an intermediary step in the compiling process: instead of compiling the high-level language (say, C) into machine code (say, x86 machine language), the compiler generates code in an intermediate language, which is close to machine language, but permits the use of some level of abstraction. The intermediate language is afterwards compiled to the target language. This technique allows one to separate cleanly the difficulties that are related to the input language, and those that are linked to the target language.

One of the earliest implementation of this concept is the p-code, introduced by Wirth in 1966[21]. P-code has been used mainly as an intermediary language for Pascal compilers: the compiler would compile the Pascal program into p-code, which could later be compiled to machine code, or even interpreted at the level of the operating system[22].

This principle is still exploited in the LLVM compiler, which is probably the standard compiler today on Unix-like platforms[23]. In LLVM, the input code is first translated to the so-called *intermediary representation*, which is later compiled into machine code, after several optimisation steps. This allows one to rapidly develop efficient compilers for new languages: one simply ought to write the translation to the LLVM intermediary representation (which should be easier than performing a direct translation to machine code thanks to the features of the intermediary language, see below), and benefit from all the optimisations of the LLVM code generator.

[21] Niklaus Wirth and Helmut Weber. EULER: A generalization of ALGOL, and its formal definition: Part II. *Commun. ACM*, 9 (2):89–99, February 1966. ISSN 0001-0782. DOI: 10.1145/365170.365202

[22] This was the case with the UCSD p-system in 1978, just like Java bytecode is today interpreted by a virtual machine!

[23] From the LLVM website (`http://www.llvm.org`): '*The LLVM Project is a collection of modular and reusable compiler and toolchain technologies*'. Note that, on many platforms such as MacOS X, calling `gcc` actually calls LLVM.

Example 1.16. For instance, consider the following C code excerpt:

```
int a, b, c ;
if (a > b)
  c=1 ;
else
  c=2 ;
```

Then, a possible LLVM intermediary representation[24] would be:

```
       %tmp = icmp sgt i32 %a, %b
       br i1 %tmp label %iftru, label %iffls
iftru: %c = 1
       br label %end
iffls: %c = 2
       br label %end
end:
```

[24] This code is not actually LLVM IR since the assignement `%c = 1` is not a valid LLVM IR instruction. Furthermore, we are assigning the register `%c` twice!. We will see how to fix this later.

As can be seen from this example, the LLVM intermediate language is pretty close to a classical machine language, with very low-level instruction such as `icmp sgt i32` to compare two integers on 32 bits, `br i1` for a conditional jump, or `br` for an unconditional jump. But this language allows one to use as many *virtual registers* (whose name begin with `%`) as desired. It is thus easier to generate LLVM intermediate language than machine language for a machine with a fixed (and limited) number of registers.

1.4 Operations on words and languages

Let us close this introduction by giving some technical preliminaries that will be used throughout these notes.

1.4.1 Operations on words

We first describe several operators that can be used to combine different words. They are all variations on the notion of *concatenation*:

Definition 1.17 (Concatenation of two words)**.** Given two words $w = w_1 w_2 \cdots w_n$ and $v = v_1 v_2 \cdots v_\ell$, the *concatenation* of w and v, denoted $w \cdot v$, is the word:

$$w \cdot v \quad = w_1 w_2 \cdots w_n v_1 v_2 \cdots v_\ell$$

By convention, $\varepsilon \cdot w = w \cdot \varepsilon = w$, for all words w. In particular $\varepsilon \cdot \varepsilon = \varepsilon$.

For example, $aa \cdot bb = aabb$ and $aa \cdot \varepsilon = \varepsilon \cdot aa = aa$. Observe that the concatenation is an *operator* on words, which is *non-commutative*, i.e., $w_1 \cdot w_2 \neq w_2 \cdot w_1$ in general; but *associative*, i.e. $w_1 \cdot w_2 \cdot w_3 = (w_1 \cdot w_2) \cdot w_3 = w_1 \cdot (w_2 \cdot w_3)$ for all words w_1, w_2, w_3. The empty word ε is a *neutral* for concatenation.

Concatenation behaves like a non-commutative product operator, such as *matrix multiplication* for instance. This justifies the 'power notation' w^n to denote the concatenation of n copies of w, just like A^n denotes $\underbrace{A \times A \times \cdots \times A}_{n \text{ times}}$ for a matrix A. Observe that, $w^n \cdot w^m = w^{n+m}$, as expected. In particular, for all n, $w^n = w^{n+0} = w^n \cdot w^0$, which explains why $w^0 = \varepsilon$.

Based on this notion, we can introduce another notations. For all natural numbers n, w^n denotes the word obtained by concatenating n copies of w:

$$w^n \quad = \underbrace{w \cdot w \cdot w \cdots w}_{n \text{ times}}$$

By convention, $w^0 = \varepsilon$ for all words w.

1.4.2 Operations on languages

We can lift the concatenation operator to sets of words, i.e., languages. Intuitively, the concatenation of two languages is a new language that contains all the words obtained by concatenating one word from the former language with one word from the latter:

Definition 1.18 (Concatenation of languages)**.** Let L_1 and L_2 be two languages. Then, their concatenation, denotes $L_1 \cdot L_2$ is the language:

$$L_1 \cdot L_2 \quad = \{w_1 \cdot w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$$

By reading this definition carefully, one realises that the empty language $\emptyset$ is *not* a neutral for language concatenation. Indeed, assume $L_1 = \emptyset$, and consider $L_1 \cdot L_2$. For a word w to belong to $L_1 \cdot L_2$, it *must* have a prefix which is a word of L_1. However, there is no word in L_1, so, no word belongs to $L_1 \cdot L_2$; that is, $L_1 \cdot L_2 = \emptyset$. However, $\{\varepsilon\}$ is: $L \cdot \{\varepsilon\} = \{\varepsilon\} \cdot L = L$ for all languages L.

For example, if $L_1 = \{$I␣love␣, I␣hate␣$\}$, and $L_2 = \{$compilers, chocolate$\}$, then $L_1 \cdot L_2 = \{$I␣love␣compilers, I␣love␣chocolate, I␣hate␣compilers, I␣hate␣chocolate$\}$.

On top of language concatenation, we can introduce several other notations:

1. For all languages L, for all natural numbers n, L^n is the language containing all words obtained by taking n words from L an concatenating them:

$$L^n \quad = \{w_1 w_2 \cdots w_n \mid \text{for all } 1 \leq i \leq n : w_i \in L\}$$

For example, if $L = \{a, b\}$, then $L^3 = \{aaa, aab, aba, baa, abb, bab, bba, bbb\}$.

2. For all languages L, the *Kleene closure* of L, denote L^* is the language containing all words made up of an arbitrary number of concatenations of words from L:

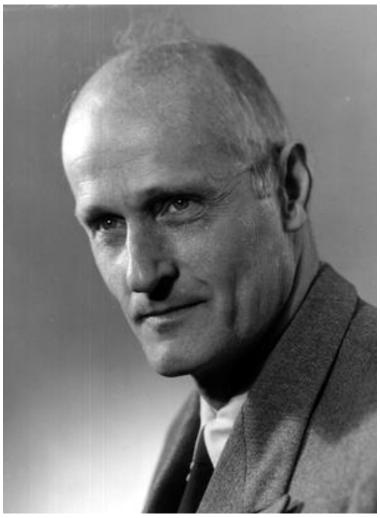
$$L^* = \{w_1 w_2 \cdots w_n \mid n \geq 0 \text{ and for all } 1 \leq i \leq n : w_i \in L\}$$

For example, $\{a\}^* = \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$. Observe that $\varepsilon \in L^*$ for all languages L, and that L^* is necessarily an infinite language, except for the cases where $L = \{\varepsilon\}$ and $L = \emptyset$, since then $L^* = \{\varepsilon\}$.

3. A variation on the Kleene closure is L^+ which is the language containing all words made up of an arbitrary and *strictly positive* number of concatenations of words from L:

$$L^+ = \{w_1 w_2 \cdots w_n \mid n \geq 1 \text{ and for all } 1 \leq i \leq n : w_i \in L\}$$

For example, $\{a\}^+ = \{a, aa, aaa, \ldots\}$. Observe that $\{\varepsilon\}^+ = \{\varepsilon\}$, so it is (tempting but) incorrect to write that $L^+ = L^* \setminus \{\varepsilon\}$...



The Kleene closure is named after Stephen Cole KLEENE, (1909 – † 1994), a prominent American logician, who is one of the giants on the shoulders of whom computer scientists are standing: he was one of the founders of computability theory, along with Kurt GÖDEL, Alonzo CHURCH, Alan TURING, to name a few...
Picture source: `http://math.library.wisc.edu/about.html#kleene`.

The Σ^* notation

Let Σ be an alphabet. Since any alphabet is a set, we can also regard Σ as a *language*, which contains only words of one character. Then, we can write Σ^*, which contains *all the words (including the empty one) that are made up of characters from* Σ. This notation will be used very often in the rest of these notes.

2 All Things Regular: Languages, Expressions…

THE READER SHOULD NOW BE CONVINCED OF THE IMPORTANCE OF LANGUAGE THEORY IN COMPUTER SCIENCE, and in particular for compiler design. Our main objective for now will be to study formal tools to: (1) define, using a finite syntax, languages that are potentially infinite; and (2) manipulate those languages (for instance, combine them using classical set operations such as union or intersection). In particular, we want to be able to answer the *membership problem*, or, in other words, to be able to tell in an automatic way whether a given word belongs to a given language or not.

In this chapter, we will study the class of *regular languages*[1]. Regular languages form one of the most basic classes of languages, yet they contain many useful languages, such as the one we have used to define (most) legal C identifiers and keywords:

[1] In francophone Belgium, regular languages are called « langages réguliers », while in France, they are called « langages rationnels », probably a much better translation.

$$L_{Cid} = \{\text{all non-empty words on } \Sigma_C \text{ that do not begin with a digit}\}$$

where: $\Sigma_C = \{a, b, \ldots, z, A, B, \ldots, Z, 0, 1, \ldots, 9, \_\}$.

Regular languages are equipped with different formal tools that allow us to represent and manipulate them:

- *Regular expressions* can be used to represent those languages. For example, if we let:

$$\begin{aligned} \ell \quad = \quad & \mathtt{a}+\mathtt{b}+\mathtt{c}+\mathtt{d}+\mathtt{e}+\mathtt{f}+\mathtt{g}+\mathtt{h}+\mathtt{i}+\mathtt{j}+\mathtt{k}+\mathtt{l}+\mathtt{m}+\mathtt{n}+\mathtt{o} \\ & +\mathtt{p}+\mathtt{q}+\mathtt{r}+\mathtt{s}+\mathtt{t}+\mathtt{u}+\mathtt{v}+\mathtt{w}+\mathtt{x}+\mathtt{y}+\mathtt{z}+\mathtt{A}+\mathtt{B}+\mathtt{C} \\ & +\mathtt{D}+\mathtt{E}+\mathtt{F}+\mathtt{G}+\mathtt{H}+\mathtt{I}+\mathtt{J}+\mathtt{K}+\mathtt{L}+\mathtt{M}+\mathtt{N}+\mathtt{O}+\mathtt{P} \\ & +\mathtt{Q}+\mathtt{R}+\mathtt{S}+\mathtt{T}+\mathtt{U}+\mathtt{V}+\mathtt{W}+\mathtt{X}+\mathtt{Y}+\mathtt{Z}+\mathtt{\_} \end{aligned}$$

  then, ℓ is a regular expression that defines « any possible letter or the _ symbol » (in regular expressions, the + symbol must be interpreted as an « or »). Similarly, let:

$$d \quad = \quad 1+2+3+4+5+6+7+8+9+0$$

  then, d is a regular expression that defines « any possible digit ». Combining those two expressions into:

$$\ell \cdot (\ell + d)^*$$

  we obtain a new regular expression that denotes exactly L_{Cid}. Here, the $\cdot$ symbol must be interpreted as concatenation, and the $*$ symbol as the Kleene closure (see Section 1.4). In other words, the above regular expression must be interpreted as: « a character matching ℓ (i.e., a non-digit character) followed by any number of characters matching either ℓ or d ».

- *Finite automata* which are abstract machines designed mainly to answer the membership problem. As such, they can also be used to represent languages, and can be used to perform operations on those languages. We will see that there are several types of finite automata, yet they all correspond to the same class of regular languages.

2.1 Regular languages

The formal definition of regular languages is recursive. It starts from the most simple languages, and uses the $+$, $\cdot$ and $*$ operations to build more complex ones:

Definition 2.1 (Regular languages)**.** Let us fix an alphabet Σ. Then, a language L is regular iff:

1. either $L = \emptyset$;
2. or $L = \{\varepsilon\}$;
3. or $L = \{a\}$ for some $a \in \Sigma$;
4. or $L = L_1 \cup L_2$;
5. or $L = L_1 \cdot L_2$;
6. or $L = L_1^*$

where L_1 and L_2 are regular languages on Σ.

Throughout this document, we denote by Reg the set of regular languages. Here are a few examples to illustrate this definition:

Example 2.2.

1. The language $\{\mathtt{abc}, \mathtt{def}\}$ on the alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\}$ is regular. Indeed, $\{\mathtt{a}\}$, $\{\mathtt{b}\}$ and $\{\mathtt{c}\}$ are all regular, by point 3 of Definition 2.1. Then, $\{\mathtt{abc}\}$ is also regular by point 5. Using similar arguments, we can show that $\{\mathtt{def}\}$ is regular. Finally, $\{\mathtt{abc}, \mathtt{def}\} = \{\mathtt{abc}\} \cup \{\mathtt{def}\}$, hence, $\{\mathtt{abc}, \mathtt{def}\}$ is regular by point 4.

   Remark that those arguments can be used to show that *all finite languages are regular*.

2. The language of all binary words (thus on the alphabt $\{0,1\}$) is regular. Indeed, this language can be defined as:

$$\{0,1\}^* = \big(\{0\} \cup \{1\}\big)^*$$

3. The language of all well-parenthesised words over $\Sigma = \{(,)\}$, on the other hand, is *not* regular (this can be proved formally). Intuitively, the definition of regular language does not allow to discriminate between words

on the basis of *unbounded counting arguments*. However, counting is unavoidable in this case: one must, at all times, keep track of the current number of pending open parenthesis to check whether closing parenthesis are legal or not.

This implies also that the set all syntactically correct C programs is *not* a regular language either, as a C program might contain, for instance, algebraic expressions with an arbitrary depth of parenthesis nesting. Hence, all the tools we will develop in this section will not be sufficient to check the full syntax of C programs (or any other « classical » programming language).

Note however, that the language of all well-parenthesised words that have a *bounded length*, over $\Sigma = \{(,)\}$, (for instance, of words containing at most 10 characters) *is* regular, because it is a finite language.

Lets us now introduce several formal tools to deal with regular languages.

2.2 Regular expressions

The first tool we will consider for regular languages are *regular expressions*. Regular expressions are a kind of algebraic characterisation of regular languages. To define regular expressions, we need to define two things: their syntax (i.e., which regular expressions can we write?), and their semantics (i.e., what is the meaning of a given regular expression, in terms of regular language?). These definitions follow closely that of regular languages:

For the more mathematically inclined readers, regular expressions form a so-called Kleene algebra, i.e, an idempotent semi-ring, see:

Dexter Kozen. On Kleene algebras and closed semirings. In *Mathematical foundations of computer science, Proceedings of the 15th Symposium, MFCS '90, Banská Bystrica/Czech. 1990*, volume 452 of *Lecture notes in computer science*, pages 26–47, 1990. URL http://www.cs.cornell.edu/~kozen/Papers/kacs.pdf

Definition 2.3 (Regular expressions)**.** Given a finite alphabet Σ, the following are regular expressions on Σ:

1. The constant $\emptyset$. It denotes the language $L(\emptyset) = \emptyset$.
2. The constant ε. It denotes the language $L(\varepsilon) = \{\varepsilon\}$.
3. All constants $a \in \Sigma$. Each constant $a \in \Sigma$ denotes the language $L(a) = \{a\}$.
4. All expressions of the form $r_1 + r_2$, where r_1 and r_2 are regular expressions on Σ. Each expression $r_1 + r_2$ denotes the language $L(r_1 + r_2) = L(r_1) \cup L(r_2)$.
5. All expressions of the form $r_1 \cdot r_2$, where r_1 and r_2 are regular expressions on Σ. Each expression $r_1 \cdot r_2$ denotes the language $L(r_1 \cdot r_2) = L(r_1) \cdot L(r_2)$.
6. All expressions of the form r^*, where r is a regular expression on Σ. Each expression r^* denotes the language $L(r^*) = \big(L(r)\big)^*$.

In addition, parenthesis are allowed in regular expressions to group subexpressions (with their usual semantics).

Sometimes, we will also use the r^+ notation as a shorthand for $r \cdot r^*$. That is, $L(r^+) = \big(L(r)\big)^+$.

Example 2.4. It is easy to check that the example $r = \ell \cdot (\ell + d)^*$ given in the introduction of the present chapter follows the definition of regular expression, and that $L(r) = L_{Cid}$.

Given that the definition of regular expressions and of their semantics follows closely the definition of regular languages (Definition 2.1), it is easy to prove that:

Theorem 2.1. *For all regular languages L, there is a regular expression r s.t. $L(r) = L$. For all regular expressions r, $L(r)$ is a regular language.*

Observe that the language $L(r)$ associated to each regular expression r is *unique*, while there can be several regular expressions to denote the same language. For instance a and $a + a$ both denote the language $\{\mathtt{a}\}$, i.e. $L(\mathtt{a}) = L(\mathtt{a} + \mathtt{a}) = \{\mathtt{a}\}$.

Actually, finding a *minimal* regular expression to denote a given regular language L is not an easy problem, since the problem of determining whether two given regular expressions r_1 and r_2 accept the same language (i.e., $L(r_1) = L(r_2)$) is a PSPACE-complete problem, see:

L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 1–9, New York, NY, USA, 1973. ACM. DOI: 10.1145/800125.804029

2.2.1 *Extended regular expressions*

Regular expressions are widely used in practice, in particular by many Unix applications. They can be used, for instance, to look for specific files using the `ls` command. As an example the following command lists all the file names in the current directory (thanks to the command `find .`) and filters them using the `grep` tool, following the pattern `^..g.*\.tex` which is given as an extended regular expression.

```
find . | grep "^..g.*\.tex"
```

The pattern asks to select only the filenames that have a `g` in the third position, and have `.tex` as extension. As can be seen from this example, the syntax of Unix regular expressions (called *extended regular expressions*) departs significantly from Definition 2.3. This is not surprising, since Definition 2.3 has been introduced mainly for theoretical purpose. On the other hand, the syntax of extended regular expressions (see Table 2.1) is probably better fitted for practical purpose. Still, all languages that are definable by extended regular expressions *are* regular, which means these new constructs do not alter the expressiveness.

The difference between the two syntaxes can be confusing: the + denotes the alternative in 'classical' regular expressions, and thus corresponds to | in extended regular expressions. On the other hand + in extended regular expressions is the repetition, i.e., it corresponds to r^+ in 'classical' regular expressions...

<table>
<tr><th>E.R.E.</th><th>Semantics</th></tr>
<tr><td>x</td><td>the character x</td></tr>
<tr><td>.</td><td>any character, except the ‘newline’ special character</td></tr>
<tr><td>"x"</td><td>the character x, even if x is an operator. For instance "." is the character . and not ‘any character’.</td></tr>
<tr><td>\x</td><td>the character x, even if x is an operator (for instance \. is the . character)</td></tr>
<tr><td>[xy]</td><td>either x or y</td></tr>
<tr><td>[a-z]</td><td>any character in the range a, b,...,z. Other ranges can be used, like 1-5 or D-X, for instance</td></tr>
<tr><td>[^x]</td><td>any character but x</td></tr>
<tr><td>^x</td><td>an x at the beginning of a line</td></tr>
<tr><td>x$</td><td>an x at the end of a line</td></tr>
<tr><td>x?</td><td>an optional x</td></tr>
<tr><td>x*</td><td>the concatenation of any number of x’s (Kleene closure)</td></tr>
<tr><td>x+</td><td>the concatenation of any strictly positive number of x’s</td></tr>
<tr><td>x{m,n}</td><td>the concatenation of k numbers of x’s-, where $\mathtt{m} \leq k \leq \mathtt{n}$.</td></tr>
<tr><td>x|y</td><td>either x or y</td></tr>
</table>

Table 2.1: Extended (Unix) regular expressions.

2.3 *Finite automata*

While regular expressions provide us with a compact and (hopefully) readable way of *specifying* regular languages, it is not clear, at the first sight, how one can use regular expressions to manipulate (automatically) regular languages. In particular, we would like to obtain, for all regular languages, some kind of abstract specification of an algorithm that allows us to answer the *membership problem* on that language (i.e., does a given word belong to that language?) Clearly, such algorithms will be important step stones to build compilers.

Such an abstract model of algorithm[2] is given by the notion of *finite automaton*. Finite automata have been introduced in the early fifties (1951) by S. Kleene (see Section 1.4), as a model of biological phenomena, namely, the response of neurons to stimuli[3]. A more systematic study of this model from the computational point of view has been done a few years later, in 1959, by Rabin and Scott[4]. Since then, finite automata have been widely recognised as one of the most fundamental models in computer science.

[2] At least, a model of algorithm which is sufficient for regular language, but might not be sufficient in general. A general model of algorithm is the Turing machine, as postulated by the Church-Turing thesis (see the computability and complexity course).

[3] Stephen C. Kleene. Representation of events in nerve nets and finite automata. Technical Report RM-704, The RAND Corporation, 1951. URL http://minicomplexity.org/pubr.php?t=2&id=2

[4] M.O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2): 114–125, April 1959. ISSN 0018-8646. DOI: 10.1147/rd.32.0114. URL https://www.researchgate.net/publication/230876408_Finite_Automata_and_Their_Decision_Problems

2.3.1 *Intuitions*

A finite automaton is an abstract machine with the following features:

- The machine reads a word, letter by letter, from the first to the last. This can be understood by envisioning the input word written on an input tape, that the machine reads cell by cell thanks to a reading head. Each cell contains one letter of the input. Once the machine has read a letter, the reading head moves to the next cell. The tape cannot be rewound.
- At all times, the machine is in a well-defined *discrete* state. There are only finitely many such states. The reading of each letter triggers a state change.
- The aim of the machine is to discriminate between words that are in a given language, and words that are not. The automaton does so by either accepting or rejecting input words. At all times, the machine produces a binary (yes/no) output, indicating whether the word prefix read so far is *accepted* or not by the machine.

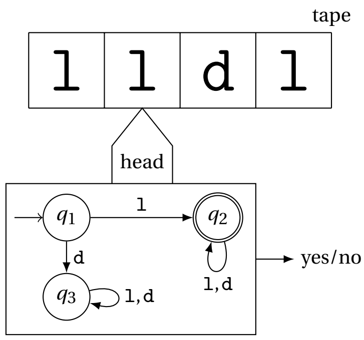


Figure 2.1: An illustration of a finite automaton.

Figure 2.1 is an illustration of those concepts. It displays the input tape (with content `lldl`), the reading head, and the output. The content of the rectangular box represents the different possible states of the automaton, by means of circles (in this case, the states are called q_1, q_2 and q_3) and the possible state changes, by means of labeled arrows between states. In this example, for instance, reading an `l` on the input tape when in state q_1 moves the current state to q_2, and so forth. In addition, we need to indicate:

1. In which state the automaton starts its execution. In our case, it is q_1, as indicated by the edge without source state pointing to q_1.
2. How is the output of the automaton determined at all times? As we have explained, this output depends only on the current state, so states

should be either accepting (in which case the output is 'yes') or rejecting (the output is 'no'). We will display accepting states as nodes with a double border. In this case, q_2 is the only accepting state.

From the intuitions sketched above, it should be clear that the behaviour of the automaton will depend only on its states and on the possible changes between those states, i.e., what is depicted inside the rectangular box in Figure 2.1. So, in the next illustrations of finite automata, we will restrict ourselves to this part, that is, we will display the automaton of Figure 2.1 as in Figure 2.2.

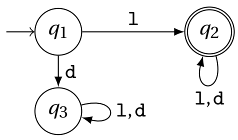


Figure 2.2: We can represent finite automata more compactly by focusing on the 'control', i.e., the states and transitions.

Since an automaton either accepts or rejects any word, it also implicitly *define a language*, which contains all the words the automaton accepts. It is easy to check that the language defined by automaton in Fig 2.2. is exactly the language of the regular expression $\mathtt{l}\cdot(\mathtt{l}+\mathtt{d})^*$ (assuming the input alphabet is $\Sigma=\{\mathtt{l},\mathtt{d}\}$). Indeed:

1. When running with a word starting by an `l` on the input tape, the automaton first moves from q_1 to q_2, which is accepting and where it will stay up to the end of its execution. So all words starting by an `l` will be accepted.
2. When running on a word that does not start by an `l` (i.e., starts by a `d`) on the input tape, the automaton first moves from q_1 to q_3, which is *not* accepting and where it will stay up to the end of its execution. So, all words starting by a `d` will be rejected.

2.3.2 *Syntax*

Let us now formalise these notions:

Definition 2.5 (Finite automaton)**.** A finite automaton is a tuple:

$$A=\langle Q,\Sigma,\delta,q_0,F\rangle$$

where:

1. Q is a finite set of states;
2. Σ is the (finite) input alphabet;
3. $\delta : Q\times(\Sigma\cup\{\varepsilon\})\mapsto 2^Q$ is the transition function;
4. $q_0\in Q$ is the initial state;
5. $F\subseteq Q$ is the set of accepting states.

Let is illustrate this definition with the example of Figure 2.2:

Example 2.6. On the example of Figure 2.2, we have:

1. $Q=\{q_1,q_2,q_3\}$;
2. $\Sigma=\{\mathtt{l},\mathtt{d}\}$;
3. $q_0=q_1$;

4. $F = \{q_2\}$;

5. and, finally, the transition function δ is given by:

$$\begin{array}{lll} \delta(q_1,\mathtt{l}) = \{q_2\} & \delta(q_1,\mathtt{d}) = \{q_3\} & \delta(q_1,\varepsilon) = \emptyset \\ \delta(q_2,\mathtt{l}) = \{q_2\} & \delta(q_2,\mathtt{d}) = \{q_2\} & \delta(q_2,\varepsilon) = \emptyset \\ \delta(q_3,\mathtt{l}) = \{q_3\} & \delta(q_3,\mathtt{d}) = \{q_3\} & \delta(q_3,\varepsilon) = \emptyset \end{array}$$

There are two important features to Definition 2.5 that one should observe. First, the co-domain of the transition function is a *set of states*. Observe that in the example of Figure 2.2, the function always returns either a singleton or the empty set. However, we can also build automata like the automaton of Figure 2.3, where $\delta(q_0, \mathtt{a}) = \{q_1, q_2\}$.

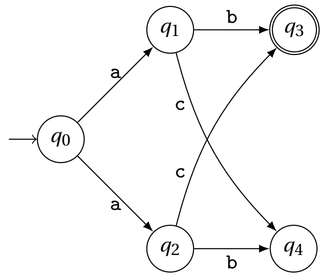


Figure 2.3: A non-deterministic finite automaton.

In this example, there are several possible executions of the automaton on the input word `ab`: the word can be read by an execution visiting q_0, q_1, then q_3, or an execution visiting q_0, q_2 and q_4. This phenomenon is called *non-determinism*, and the automaton of Figure 2.3 is said to be non-deterministic. Non-determinism raises several natural questions:

1. How do we determine the output of the automaton, when there are several possible runs on the same word, that do not all end in an accepting state? This occurs with the word `ab` and the automaton in Figure 2.3. The rule is that, in non-deterministic automata, *there must exist one run that accepts for the word to be accepted.*

2. What is the point of non-determinism anyway? We have introduced finite automata as *abstract machines* that *model algorithms*. Clearly, an algorithm, or, at the very least, a computer program, are *deterministic*, so the existence of non-deterministic automata seems to hurt the intuition. It turns out that non-determinism is a very helpful tool for modeling certain kinds of problems. Also, we will see later that *all non-deterministic automata can be converted into an equivalent deterministic automaton.*

   For example, consider the family of languages L_n (where $n \geq 1$) defined as follows: 'all binary words that contain two `1`'s separated by exactly n characters'. Devising a non-deterministic automaton that recognises this language is quite easy: Figure 2.4 shows a non-deterministic automaton recognising L_2. This example can be generalised to any n, by inserting more states between q_2 and q_3 for instance. Clearly, if the automaton outputs 'yes' on some word w, then the execution of the automaton has gone through all the states, which guarantees that there are two `1`'s separated by two characters (the `1`'s that have been read when moving from q_0 to q_1 and from q_3 to q_4 respectively). Conversely, if a word belongs to L_2, there is clearly at least one run ending in q_4 that reads it, and so the output of the automaton is 'yes' since q_4 is accepting. We will see later (see the paragraph on the size of deterministic automata in Section 2.4.2) that devising a deterministic automaton for this language is a bit more tricky.

A common intuition about non-deterministic automata is that they have the capability to 'guess' something about the future of a word. Assume the automaton in Figure 2.4 is currently in state q_0, has a word from L_2 on its input tape, and reads a `1`. It has thus two 'choices' for its next current state: either stay in q_0 or move to q_1. Clearly, the latter is a good choice to accept the current word only if the `1` that is read is followed, three characters ahead, by another `1`. Since the automaton cannot read ahead of its reading head, nor rewind the tape, it must 'guess' correctly whether each `1` will be followed by another `1` three characters ahead, and any accepting run can be understood as a 'correct guess' from the automaton.

The second important feature of Definition 2.5 is the fact that some transitions can be labeled by the empty word ε. This is called a 'sponta-

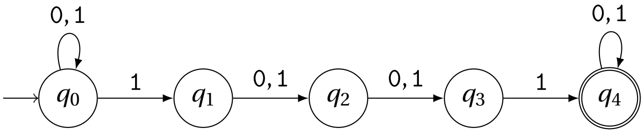
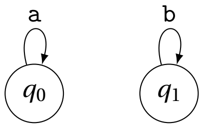
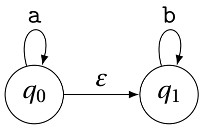


Figure 2.4: An automaton recognising L_2, i.e., the set of all binary words that contain two 1's separated by exactly 2 characters.

neous move' and allows the automaton to change its current state without reading any character on the input (hence, without moving its reading head). Again, spontaneous moves depart radically from our intuition of algorithm, yet they can be useful for modeling purpose. For instance, suppose we want to build an automaton for the language composed of all words that start with a (possibly empty) sequence of a's, followed by a (possibly empty) sequence of b's. One natural way to do it would be to start by building two automata for those two parts of the words in the language:



then, add spontaneous move between those states, to allow the automaton to move from the 'sequence of a's' part to the 'sequence of b 's':



and, finally, add the relevant initial and accepting states:

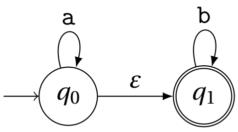


Figure 2.5: A non-deterministic automaton with spontaneous moves that accepts a^*b^*.

Because not all automata may use non-determinism and spontaneous moves, we define the following classes of finite automata:

Definition 2.7 (Classes of finite state automata)**.**

1. A *non-deterministic finite state automata with ε-transitions* (ε-NFA for short) is a finite state automaton, as in Definition 2.5.

2. A *non-deterministic finite state automaton* (NFA for short) is an ε-NFA $A = \langle Q, \Sigma, \delta, q_0, F\rangle$ s.t. for all $q \in Q$: $\delta(q, \varepsilon) = \emptyset$. In this case, we will henceforth assume that the signature of the transition function is $\delta : Q \times \Sigma \mapsto 2^Q$.

3. A *deterministic finite state automaton* (DFA for short) is an NFA $A = \langle Q, \Sigma, \delta, q_0, F\rangle$ s.t. for all $q \in Q$, for all $a \in \Sigma$: $|\delta(q, a)| = 1$. In this case, we will henceforth assume that the signature of the transition function is $\delta : Q \times \Sigma \mapsto Q$, and that the function is *complete*♣.

Observe that, in our definition of DFAs, we request that $|\delta(q, a)| = 1$, for all q and a, i.e., that each state has exactly one successor for each letter. However, when depicting DFAs, and in order to keep the figures readable, we will sometimes omit some transitions that lead to a *sink state*, i.e., a state from which nothing can be accepted (like state q_3 in Figure 2.2). Note also that some authors do not ask for the transition function to be complete and use the weaker constraint $|\delta(q, a)| \leq 1$ in their definition of DFAs.

For instance, the automaton in Figure 2.2 is a DFA, hence also an NFA and an ε-NFA. The automaton in Figure 2.3 is an NFA, hence also an ε-NFA, but is *not* a DFA. The automaton in Figure 2.5 is an ε-NFA, but neither an NFA, nor a DFA.

2.3.3 *Semantics*

Let us now define formally the notions of 'execution', 'accepting a word', etc that we have discussed informally so far.

Definition 2.8 (Configuration of an ε-NFA)**.** A *configuration* of an ε-NFA $A = \langle Q, \Sigma, \delta, q_0, F\rangle$ is a pair $\langle q, w\rangle \in Q \times \Sigma^*$, where q is the *current state*, and w is the *input word suffix* that remains to be read.

The *initial configuration* of A on the input word w is $\langle q_0, w\rangle$.

A configuration $\langle q, w\rangle$ is *accepting* (or final) iff $q \in F$ and $w = \varepsilon$.

Remark that we give these definitions in the most general case of ε-NFAs, but, since NFAs and DFAs are special cases of ε-NFAs, these definitions apply to them too.

Intuitively, a pair $\langle q, w\rangle$ completely characterises the current 'configuration' of the automaton: its current state is q and the word w remains on the input (in other words, the reading head is currently on the first character of w, if $w \neq \varepsilon$; or at the end of the tape, if $w = \varepsilon$). Then, using the transition relation, we can define how an automaton changes its current configuration:

Definition 2.9 (Configuration change)**.** Let $A = \langle Q, \Sigma, \delta, q_0, F\rangle$ be an ε-NFA, and let (q_1, w_1) and (q_2, w_2) be two configurations of A. Then we say that (q_2, w_2) *is a successor of* (q_1, w_1) iff there is a letter $a \in \Sigma \cup \{\varepsilon\}$ such that (i) $w_1 = a \cdot w_2$ and (ii) $q_2 \in \delta(q_1, a)$. We denote this successor relation by:

$$(q_1, w_1) \vdash_{\mathrm{A}} (q_2, w_2)$$

In the rest of these notes, we will often omit the subscript on the operator, when the ε-NFA is clear from the context, and write $(q_1, w_1) \vdash (q_2, w_2)$ instead. Very often, we will consider sequences of configurations (q_1, w_1), $(q_2, w_2), \ldots, (q_n, w_n)$ s.t. $(q_i, w_i) \vdash (q_{i+1}, w_{i+1})$ for all $1 \leq i \leq n-1$. Such sequences are called *runs* of the automaton (on the word w_1, which is the word in the first configuration of the run). We say that a run (q_1, w_1), $(q_2, w_2), \ldots, (q_n, w_n)$ is *accepting* iff its last configuration (q_n, w_n) is accepting; and we say that it is *initialised* iff its first configuration is initial.

Since $\vdash_{\mathrm{A}}$ is a binary relation, we use the classical $\vdash_{\mathrm{A}}^*$ notation to denote its reflexive and transitive closure♣. Then, we can define the accepted language of an automaton:

Definition 2.10 (Accepted language of an ε-NFA)**.** Let $A = \langle Q, \Sigma, \delta, q_0, F\rangle$ be an ε-NFA. Then, its *accepted language* is:

$$L(A) = \left\{w \in \Sigma^* \mid \exists q \in F \text{ s.t. } \langle q_0, w\rangle \vdash_{\mathrm{A}}^* \langle q, \varepsilon\rangle\right\}$$

In other words, a word w is accepted iff the automaton admits an initialised and accepting run on w.

Example 2.11. As an example, let us consider again the ε-NFA in Figure 2.5, and let us check whether it accepts $w = \mathtt{aab}$. The only possible run, starting in q_0, of this automaton on w is:

$$(q_0, \mathtt{aab}), (q_0, \mathtt{ab}), (q_0, \mathtt{b}), (q_1, \mathtt{b}), (q_1, \varepsilon)$$

It is easy to check that the first configuration of the run is initial, that the last is accepting. Hence, $w = \mathtt{aab}$ is accepted.

On the other hand, the maximal run that can be built on the word $w' = \mathtt{ba}$ is:

$$(q_0, \mathtt{ba}), (q_1, \mathtt{ba}), (q_1, \mathtt{a})$$

because there are no $\mathtt{a}$-labeled transitions from q_1. Hence, w' is not accepted since $(q_1, \mathtt{a})$ is not accepting.

Recall that, with non-deterministic automata, several runs are possible on a single input word. In this case, it is sometimes convenient to represent all the possible runs by means of a *tree*, whose nodes are labeled by configurations, and whose edges correspond to the $\vdash$ relation. As an example, the tree of possible runs of the automaton in Figure 2.3 is shown in Figure 2.6.

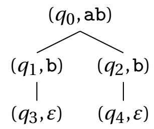


Figure 2.6: The run-tree of the automaton in Figure 2.3 on the word $\mathtt{ab}$.

2.4 Equivalence between automata and regular expressions

So far, we have reviewed two families of models for defining and manipulating languages: regular expressions, on the one hand, and finite automata, on the other hand. We know that regular expressions define exactly the class of *regular languages* (see definition 2.1 and Theorem 2.1), but what about the *expressive power* of the three different classes of automata we have introduced? Obviously, DFAs cannot be more expressive than NFAs, which cannot be more expressive than ε-NFAs, by definition. We have already seen at least one example of automaton that recognises the same language than a given regular expression (see Figure 2.5), but can this be generalised?

The 'expressive power' of a model is a term often used to speak about the class of languages that the model can define. One can thus speak about the *expressive power* of regular expressions (i.e., the regular languages), or the *expressive power* of finite automata and compare them...

It turns out that the expressive power of all three classes of finite automata is exactly the same, and equals that of regular expressions, that is, the regular languages. This result is due to Stephen Kleene[5]:

[5] Stephen C. Kleene. Representation of events in nerve nets and finite automata. Technical Report RM-704, The RAND Corporation, 1951. URL http://minicomplexity.org/pubr.php?t=2&id=2

Theorem 2.2 (Kleene's theorem)**.** *For every regular language L, there is a* DFA *A such that $L(A) = L$. Conversely, for all ε-*NFA*s A, $L(A)$ is regular.*

In other words, all finite automata recognise regular languages and all regular languages are recognised by a finite automaton. To establish this result, we will give constructions that convert finite automata into regular expressions and vice-versa. More precisely, we will give algorithms to:

Our formulation of the theorem might seem restrictive, but one must always bear in mind that DFAs are a special case of NFAs, which are, in turn, a special case of ε-NFAs. Hence, 'For every regular language L, there is a DFA A such that $L(A) = L$' entails that there is also an NFA and an ε-NFA recognising L (actually, the DFA A can serve for that purpose). Conversely, 'for all ε-NFAs A: $L(A)$ is regular' implies that the language of all NFAs and DFAs are also regular!

1. Convert any regular expression into an ε-NFA defining the same language.

2. Convert any ε-NFA into a DFA accepting the same language. This is called 'determinising' the ε-NFA as it somehow turns it into a deterministic version. Observe that this method can be applied, in particular, to any NFA.

3. Convert any DFA into a regular expression defining the same language.

This set of transformations is summarised in Figure 2.7. Together with Theorem 2.1, those transformations allow us to conclude that finite automata recognise exactly regular languages.

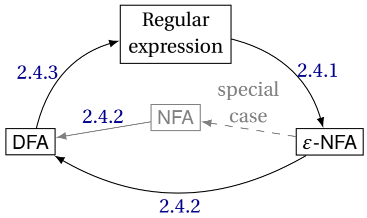


Figure 2.7: The set of transformations used to prove Theorem 2.2, with the section numbers where they are introduced.

2.4.1 *From regular expressions to ε-NFAs*

To turn a regular expression into a finite automaton, we will once again exploit the recursive definition of the syntax of regular expressions. The construction we are about to describe dates back from the sixties. It is widely attributed to Thompson[6], who has based his work on a previous construction by McNaughton and Yamada[7].

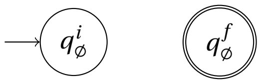
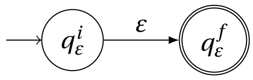
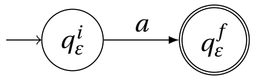
The induction hypothesis of the construction is that it builds, for all regular expressions r, an ε-NFA A_r s.t. (i) $L(A_r) = L(r)$; and (ii) the (necessarily unique) initial state of A_r is called q_r^i, and A_r has exactly one final state that we denote q_r^f. Moreover, no transition enter q_r^i, nor leave q_r^f.

[6] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. ISSN 0001-0782. DOI: 10.1145/363347.363387

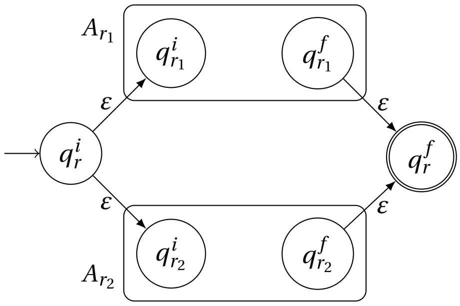
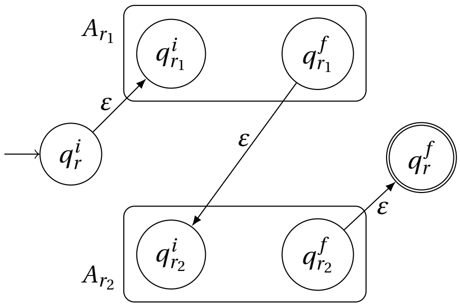
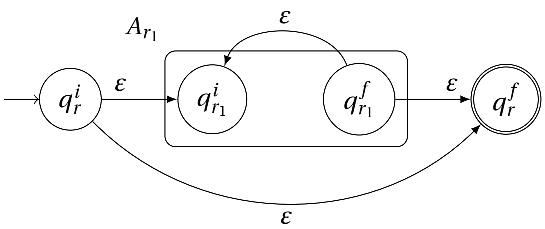
[7] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *Electronic Computers, IRE Transactions on*, EC-9(1):39–47, March 1960. ISSN 0367-9950. DOI: 10.1109/TEC.1960.5221603

Base cases Building ε-NFAs that accept the base cases of regular expressions is easy, as shown in the following table:

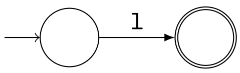
Observe that we could have given simpler constructions. For instance, A_ε could have been made up of only one (initial and accepting) state. However, the construction we present has the benefit to keep initial and final states separated, and is therefore more systematic.

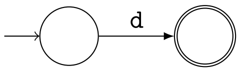
Regular expression r	ε-NFA A_r
$\emptyset$	
ε	
$a \in \Sigma$	

Inductive case For the inductive case, we assume ε-NFAs A_{r_1} and A_{r_2} are already known for two regular expressions r_1 and r_2. We treat the disjunction, concatenation and Kleene closure as follows:

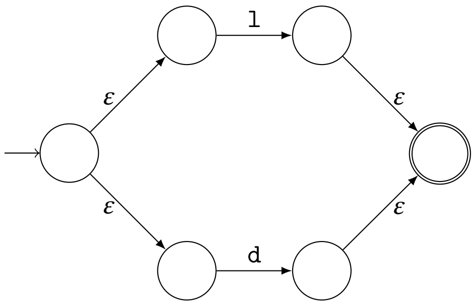
Regular expression r	ε-NFA A_r
$r_1 + r_2$	
$r_1 \cdot r_2$	
r_1^*	

Example 2.12. Let us consider the regular expression $\mathtt{l}\cdot(\mathtt{l}+\mathtt{d})^*$ on the alphabet $\Sigma = \{\mathtt{l},\mathtt{d}\}$. Following the construction, we start with the base cases:

$A_{\mathtt{l}} =$ 

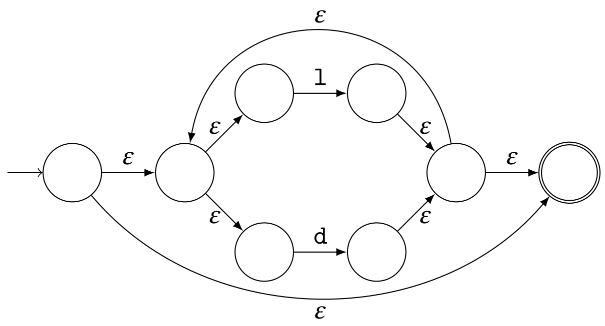
$A_{\mathtt{d}} =$ 

From them, we build the ε-NFA for $\mathtt{l}+\mathtt{d}$:

$A_{\mathtt{l+d}} =$ 

Then, let's apply the Kleene closure:

$A_{(\mathtt{l}+\mathtt{d})^*} =$



Finally, we apply the construction for the concatenation (with automaton $A_\mathtt{l}$ we have computed above) and obtain the ε-NFA $A_{\mathtt{l}\cdot(\mathtt{l}+\mathtt{d})^*}$ displayed in Figure 2.8.

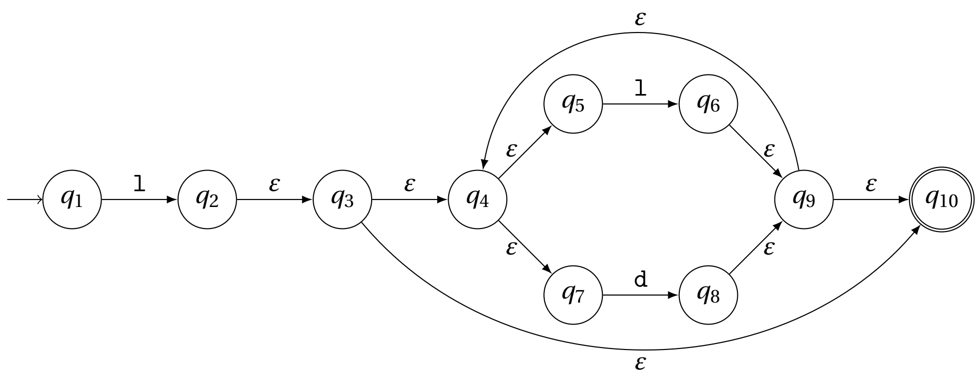


Figure 2.8: The ε-NFA built from the regular expression $\mathtt{l}\cdot(\mathtt{l}+\mathtt{d})^*$, using the systematic method.

2.4.2 *From ε-NFAs to DFAs*

For many practical purposes (like the building of a parser), non-deterministic automata are not acceptable, and a deterministic automaton is necessary. We will now review a technique that converts any ε-NFA into a DFA accepting the same language.

Let us first sketch the intuition behind the construction, by considering the ε-NFA in Figure 2.8. Assume the automaton is currently in state q_3 and the next character on the input is an `l`. The automaton can read this `l`, but must first follow two ε-transitions in order to reach state q_5. Then, after reading the `l` from q_5, the automaton can follow several other ε-transitions and end up in any of the following states: q_4, q_5, q_6, q_7, q_9 or q_{10}. From those states, the automaton might continue its execution, yielding several possible runs on the same word. By definition of ε-NFAs, only one of those runs needs to be accepting for the automaton to accept the word.

Intuitively, the DFA D corresponding to a given ε-NFA A will *simulate* all the possible executions of A on a given input word, by *tracking* the possible states in which A can be at all times. To perform this tracking, the DFA needs some kind of memory, and we will use the states of the DFA to encode this memory. Thus, the states of the DFA D will be *subsets* of A's set of states. Roughly speaking, when the DFA will be in state $S = \{q_1, q_2, \ldots, q_n\}$

(where $q_1, \dots q_n$ are states of the ε-NFA) after reading a prefix w', then, $\{q_1, \dots q_n\}$ is exactly the set of all states that the ε-NFA can reach by reading the same prefix w'.

ε-closure To formalise this intuition, we need several ancillary definitions. Let us first introduce the $\varepsilon\mathsf{closure}$ function that takes a state $q \in Q$ and returns *the set of states that automaton A can reach by reading only ε's.* The next definition formalises this. In this definition, we extend slightly the definition of the transition function δ by allowing it to be applied to a *set of states.* That is, for a set of states S, and a letter a, we let:

$$\delta(S, a) = \bigcup_{q \in S} \delta(q, a)$$

In other words, computing $\delta(S, a)$ amounts to computing all the states that the automaton can reach from any state $q \in S$, by reading an a. Then:

Definition 2.13 (ε-closure)**.** Let $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ be an ε-NFA. For all $i \in \mathbb{N}$, let $\varepsilon\mathsf{closure}^i(q)$ be defined as follows:

$$\varepsilon\mathsf{closure}^i(q) = \begin{cases} \{q\} & \text{if } i = 0 \\ \delta(\varepsilon\mathsf{closure}^{i-1}(q), \varepsilon) \cup \varepsilon\mathsf{closure}^{i-1}(q) & \text{otherwise} \end{cases}$$

Then, for all $q \in Q$: $\varepsilon\mathsf{closure}\big(q\big) = \varepsilon\mathsf{closure}^K(q)$, where K is the least value s.t. $\varepsilon\mathsf{closure}^K(q) = \varepsilon\mathsf{closure}^{K+1}(q)$.

The definition might seem hard to read, but the intuition is really easy: $\varepsilon\mathsf{closure}^i(q)$ is the set of states that A can reach from q by following *at most i* transitions labeled by ε.

Example 2.14. Let us consider the ε-NFA in Figure 2.8, and let us compute $\varepsilon\mathsf{closure}\big(q_6\big)$. We compute $\varepsilon\mathsf{closure}^i(q)$ for $i = 0, 1, \dots$ up to stabilisation:

$$\varepsilon\mathsf{closure}^0(q_6) = \{q_6\}$$

$$\begin{aligned}\varepsilon\mathsf{closure}^1(q_6) &= \delta(\varepsilon\mathsf{closure}^0(q_6), \varepsilon) \cup \varepsilon\mathsf{closure}^0(q_6) \\ &= \delta(\{q_6\}, \varepsilon) \cup \{q_6\} \\ &= \{q_9\} \cup \{q_6\} \\ &= \{q_6, q_9\}\end{aligned}$$

$$\begin{aligned}\varepsilon\mathsf{closure}^2(q_6) &= \delta(\{q_6, q_9\}, \varepsilon) \cup \{q_6, q_9\} \\ &= \{q_4, q_9, q_{10}\} \cup \{q_6, q_9\} \\ &= \{q_4, q_6, q_9, q_{10}\}\end{aligned}$$

$$\begin{aligned}\varepsilon\mathsf{closure}^3(q_6) &= \delta(\{q_4, q_6, q_9, q_{10}\}, \varepsilon) \cup \{q_4, q_6, q_9, q_{10}\} \\ &= \{q_4, q_5, q_7, q_9, q_{10}\} \cup \{q_4, q_6, q_9, q_{10}\} \\ &= \{q_4, q_5, q_6, q_7, q_9, q_{10}\}\end{aligned}$$

$$\begin{aligned}\varepsilon\mathsf{closure}^4(q_6) &= \delta(\{q_4, q_5, q_6, q_7, q_9, q_{10}\}, \varepsilon) \cup \{q_4, q_5, q_6, q_7, q_9, q_{10}\} \\ &= \{q_4, q_5, q_6, q_7, q_9, q_{10}\} \cup \{q_4, q_5, q_6, q_7, q_9, q_{10}\} \\ &= \{q_4, q_5, q_6, q_7, q_9, q_{10}\} \\ &= \varepsilon\mathsf{closure}^3(q_6)\end{aligned}$$

So, we let $\varepsilon\mathsf{closure}\big(q_6\big) = \varepsilon\mathsf{closure}^3(q_6) = \{q_4, q_5, q_6, q_7, q_9, q_{10}\}$.

Again, we extend the $\varepsilon\mathsf{closure}$ function to set of states S, as we did for the δ function: $\varepsilon\mathsf{closure}(S) = \cup_{q\in S}\varepsilon\mathsf{closure}(s)$. In particular $\varepsilon\mathsf{closure}(\emptyset) = \emptyset$.

Determinisation We can now give the construction to determinise an ε-NFA.

<u>Determinisation of ε-NFAs</u>

Given an ε-NFA $A = \langle Q^A, \Sigma, \delta^A, q_0^A, F^A\rangle$, we build the DFA $D = \langle Q^D, \Sigma, \delta^D, q_0^D, F^D\rangle$ as follows:

1. $Q^D = 2^{Q^A}$
2. $q_0^D = \varepsilon\mathsf{closure}\left(q_0^A\right)$
3. $F^D = \{S \in Q^D \mid S \cap F^A \neq \emptyset\}$
4. for all $S \in Q^D$, for all $a \in \Sigma$: $\delta^D(S, a) = \varepsilon\mathsf{closure}\left(\delta^A(S, a)\right)$

Recall that, for a finite set S, the notation 2^S denotes the *set of subsets of* S. For example, if $S = \{1,2,3\}$, then $2^S = \{\{1\},\{2\},\{3\},\{1,2\},\{2,3\},\{1,2,3\},\emptyset\}$.

Let us comment briefly on the items of this definition:

1. As expected, the set of states of the DFA is the set of subsets of the ε-NFAs states.

2. The initial state of the DFA is the set of states the NFA can reach from its own initial state q_0^A by reading only ε-labeled transitions. Thus, q_0^D is the set of states in which the ε-NFA can be *before reading any letter*.

3. A state of the DFA is accepting iff it contains at least one accepting state of the ε-NFA. This is coherent with the intuition that at least one execution of the ε-NFA must accept for the word to be accepted.

4. The transition function consists in: first reading a letter, then following as many ε-labeled transitions as possible.

Although we will not present the details here[8], one can show that the DFA obtained from any ε-NFA by the above construction preserves the accepted language of the ε-NFA:

[8] The interested reader can find a proof in: John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321455363

Theorem 2.3 (Determinisation of ε-NFAs)**.** *For all ε-NFA A, the DFA D obtained by determinising A accepts the same language as A: $L(A) = L(D)$.*

Proof. (Sketch) Let us assume $A = \langle Q^A, \Sigma, \delta^A, q_0^A, F^A\rangle$. The proof is based on an extension of the transition function which receives a state q, and a (possibly empty) word w, and returns the set of all possible states that the automaton can reach by reading the word w. Formally, given a transition function δ, its extended version is $\hat{\delta}$ defined as:

$$\hat{\delta}(q, \varepsilon) = \varepsilon\mathsf{closure}\left(q\right)$$
$$\hat{\delta}(q, wa) = \varepsilon\mathsf{closure}\left(\delta(\hat{\delta}(q, w), a)\right)$$

Then, clearly, we can define A's language using $\hat{\delta}^A$ instead of δ^A, since a word is accepted iff reading it from q_0^A allows one to reach at least one accepting state. In other words:

$$L(A) = \left\{ w \in \Sigma^* \mid \hat{\delta}^A(q_0^A, w) \cap F^A \neq \emptyset \right\}$$

Then to prove that $L(D) = L(A)$, it is sufficient to check, that, for all words w the set $\hat{\delta}^A(q_0, w)$ is exactly the state which is reached by D when reading w from its initial state. This can be established by induction on the length of w, which is easy because of the inductive definition of $\hat{\delta}^A$. □

Example 2.15. Let us consider again the ε-NFA in Figure 2.8, and let us build its deterministic counterpart $D = \langle Q^D, \Sigma, \delta^D, q_0^D, F^D \rangle$.

D's initial state $q_0^D = \varepsilon\mathsf{closure}\left(q_1\right) = \{q_1\}$. Let us call this state S_1. From $S_1 = \{q_1\}$, the automaton A reaches $\{q_2\}$ by reading an `1`. From q_2, it can take several ε-labeled transitions. In other words:

$$\begin{aligned}
\delta^D(S_1, \mathtt{1}) &= \varepsilon\mathsf{closure}\left(\delta^A(S_1, \mathtt{1})\right) \\
&= \varepsilon\mathsf{closure}\left(\{q_2\}\right) \\
&= \{q_2, q_3, q_4, q_5, q_7, q_{10}\}
\end{aligned}$$

Let us denote this set S_2. Observe that S_2 is accepting, since $S_2 \cap F^A = \{q_{10}\} \neq \emptyset$.

On the other hand, reading a `d` from S_1 yields the empty set. Hence:

$$\begin{aligned}
\delta^D(S_1, \mathtt{d}) &= \varepsilon\mathsf{closure}\left(\emptyset\right) \\
&= \emptyset
\end{aligned}$$

We continue the construction of the DFA similarly, from S_2:

$$\begin{aligned}
\varepsilon\mathsf{closure}\left(\delta^A(S_2, \mathtt{1})\right) &= \varepsilon\mathsf{closure}\left(\{q_6\}\right) \\
&= \{q_4, q_5, q_6, q_7, q_9, q_{10}\}
\end{aligned}$$

Let us denote this last set by S_3. Observe that S_3 is accepting too.

Reading a `d` from S_2 yields:

$$\begin{aligned}
\varepsilon\mathsf{closure}\left(\delta^A(S_2, \mathtt{d})\right) &= \varepsilon\mathsf{closure}\left(\{q_8\}\right) \\
&= \{q_4, q_5, q_8, q_7, q_9, q_{10}\}
\end{aligned}$$

Let us denote this state S_4.

And from $\emptyset$:

$$\begin{aligned}
\varepsilon\mathsf{closure}\left(\delta^A(\emptyset, \mathtt{d})\right) &= \emptyset \\
\varepsilon\mathsf{closure}\left(\delta^A(\emptyset, \mathtt{1})\right) &= \emptyset
\end{aligned}$$

Now, from S_3:

$$\begin{aligned}
\varepsilon\mathsf{closure}\left(\delta^A(S_3, \mathtt{d})\right) &= \varepsilon\mathsf{closure}\left(\{q_8\}\right) \\
&= S_4 \\
\varepsilon\mathsf{closure}\left(\delta^A(S_3, \mathtt{1})\right) &= \varepsilon\mathsf{closure}\left(\{q_6\}\right) \\
&= S_3
\end{aligned}$$

Finally, from S_4:

$$\begin{aligned}\varepsilon\text{closure}\big(\delta^A(S_4,\mathtt{d})\big) &= \varepsilon\text{closure}\big(\{q_8\}\big)\\ &= S_4\\ \varepsilon\text{closure}\big(\delta^A(S_4,\mathtt{l})\big) &= \varepsilon\text{closure}\big(\{q_6\}\big)\\ &= S_3\end{aligned}$$

The resulting DFA is depicted in Figure 2.9. Actually, this figure shows the part of the DFA which is *reachable* from the initial state (since we have built the states iteratively from the initial state). Indeed, a state like $\{q_1, q_{10}\}$ also exists in the DFA, but is not reachable.

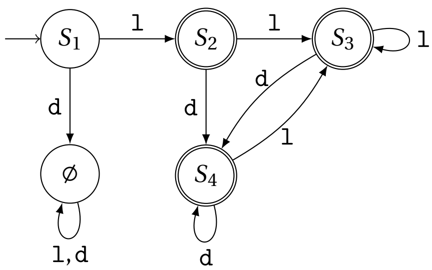


Figure 2.9: The DFA obtained from the ε-NFA $A_{\mathtt{l}\cdot(\mathtt{l}+\mathtt{d})^*}$.

Observe that the result on the determination process does not always yield a minimal automaton (in this case, the states S_2 and S_3 could be 'merged'). We will review, in Section 2.5 a technique for minimising DFAs.

Size of the determinised automaton Since the set of states of the DFA D obtained by the above construction is 2^{Q^A} (where Q^A is the set of states of the original ε-NFA A), D could be, in theory, exponentially larger than A. However, on the previous example, $A_{\mathtt{l}\cdot(\mathtt{l}+\mathtt{r})^*}$ has ten states, while its corresponding DFA (Figure 2.9) has 'only' four states reachable (instead of 1024). So, even if the DFA has many states, most of them are not reachable and their construction can thus easily be avoided.

Is it always going to be the case? The answer, unfortunately, is 'no': we will exhibit an infinite family of languages L_n (for all $n \geq 1$) s.t., (i) for all $n \geq 1$: there is an ε-NFA A_n that recognises L_n, and the size of the A_n's grows linearly with n; and (ii) letting D_n be any deterministic automaton recognising L_n (for all n), the size of the D_n's grow exponentially with n. Observe that the above statement is rather strong: whatever the deterministic automaton D_n we chose to recognise L_n, this automaton is bound to a number of states which is exponential in n. Thus, there is no hope to obtain a determinisation procedure that always produces a DFA that is polynomial in the size of the original ε-NFA (and this holds in particular for the determinisation procedure we have given above).

The languages L_n are those of binary words that contain at least two 1's separated by n characters, i.e.:

$$L_n = \{w_1 w_2 \cdots w_\ell \in \{0,1\}^* \mid \exists 1 \leq i \leq \ell - n - 1 : w_i = w_{i+n+1} = 1\}$$

Building an NFA accepting L_n (for each n) is easy: we have already given in Figure 2.4 an NFA accepting L_2, and Figure 2.10 shows the general construction. It is easy to see that, for all $n \geq 1$, A_n accepts L_n. Indeed, if a run of the automaton reaches the accepting state q_a, it has necessarily traversed the sequence of states q_i, q_0, q_1,... q_n, q_a, which guarantees that the word contains two 1's (read by the transitions from q_i to q_0 and from q_n to q_a respectively), separated by n characters. On the other hand, if a word w is in L_n, then it can be accepted by the automaton: the automaton stays in q_i, up to the point where it reads the first of the two 1's that are separated by n characters, and moves to q_1. Then, the accepting states will be reached for sure.

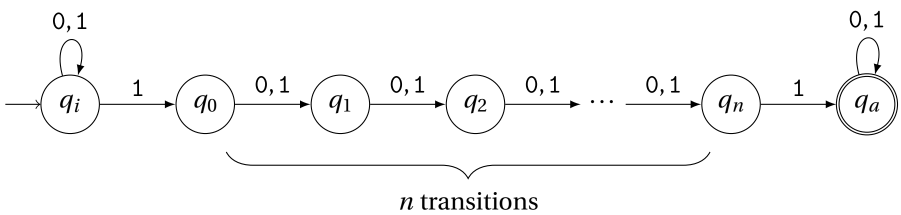


Figure 2.10: The family of ε-NFAs A_n ($n \geq 1$) s.t. for all $n \geq 1$: $L(A_n) = L_n$.

Observe that the only non-deterministic choice of A_n occurs in q_i: when reading a 1, the automaton can either stay in q_i, or move to q_0. If it decides for the latter, it will accept only if this 1 is followed $n+1$ characters later by another 1. In some sense, each time the automaton sees a 1 in state q_i, it must guess whether this 1 will be followed $n+1$ characters later by another 1, in which case it moves to q_0. The purpose of the states q_0, q_1,..., q_n is to check that this guess was correct.

Finally, it is easy to see that, for all $n \geq 1$, A_n has $n+3$ states, so the size of the A_n automata grows indeed *linearly* wrt n.

Now, let us argue that the size of deterministic automata D_n that accept L_n grows *exponentially wrt n*. To support our discussion, we consider the automaton A_1:

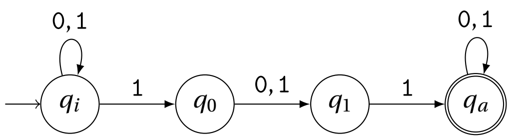


Figure 2.11: The ε-NFA A_1 recognising L_1.

To check that a word contains indeed two 1's separated by 1 character, the automaton must, at all times, *remember the two last read characters*, that we denote b_0 and b_1 (that is, if the automaton is reading character w_i of the input word, then $b_0 = w_{i-2}$ and $b_1 = w_{i-1}$). Then, the automaton proceeds as follows every time it reads a character:

- If the character is 1, then, the automaton must check whether b_0 is a 1. If yes, it accepts. Otherwise, it needs to update its memory, by copying the value of b_1 to b_0, and letting $b_1 = 1$.

- If the character is 0, then, the automaton must only update its memory, by, again, copying the value of b_1 to b_0, and letting $b_1 = 0$.

Thus, the automaton clearly needs those two bits b_0 and b_1 of memory. There are $2^2 = 4$ possible memory values which are encoded in the states of the DFA. Hence, D_1 must have at least 4 states. This reasoning generalises to any n, letting the number of memory bits increase with n: for all $n \geq 1$, the automaton needs $n+1$ bits of memory. So, any DFA D_n recognising L_n must have at least 2^{n+1} states.

As a matter of fact the automaton D_1 obtained by determinising A_1 (using the procedure of Section 2.4.2) is displayed in Figure 2.12. The four states encoding the memory are the four non-accepting states. The gray labels show the values of the two memory bits associated to those states—of course, this intuition is valid only after D_1 has read at least 2 characters. Clearly, this automaton could be made simpler, but only by 'merging' the accepting states: it is not possible to reduce the number of non-accepting states without changing the language of the automaton.

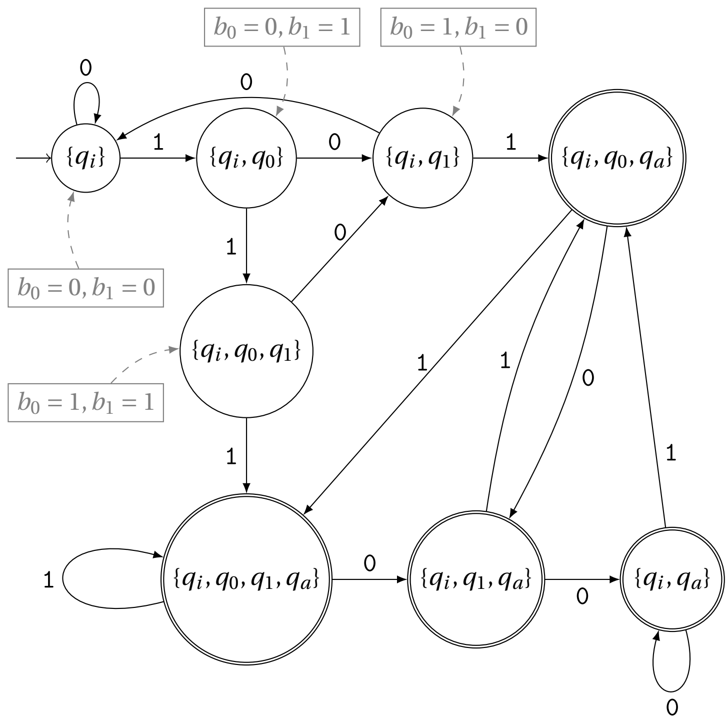


Figure 2.12: The DFA D_1 obtained from the NFA A_1. The gray labels show the values of the memory bits associated to some states.

2.4.3 From ε-NFAs to regular expressions

In order to complete the picture in Figure 2.7, we present now a technique for turning an ε-NFA (thus, in particular, a DFA) A into a regular expression defining the same language as A.

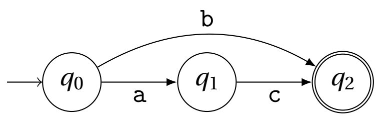
Several techniques exist to do so. The original one can be found in Kleene's seminal paper[9], and has later been rephrased using the (now) standard automata formalism by McNaughton and Yamada[10]. The technique we will now present has been introduced by Brzozowski and McCluskey in 1963[11].

[9] Stephen C. Kleene. Representation of events in nerve nets and finite automata. Technical Report RM-704, The RAND Corporation, 1951. URL http://minicomplexity.org/pubr.php?t=2&id=2

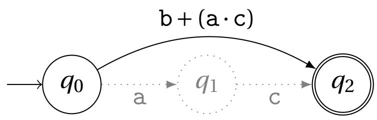
[10] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *Electronic Computers, IRE Transactions on*, EC-9(1):39–47, March 1960. ISSN 0367-9950. DOI: 10.1109/TEC.1960.5221603

[11] J.A. Brzozowski and Jr. McCluskey, E.J. Signal flow graph techniques for sequential circuit state diagrams. *Electronic Computers, IEEE Transactions on*, EC-12(2):67–76, April 1963. ISSN 0367-7508. DOI: 10.1109/PGEC.1963.263416

This technique is often called the *state elimination technique*. Roughly speaking, it consists in eliminating states of the original ε-NFA one by one, and updating the labels of the remaining transitions to make sure that the accepted language does not change. To do this, one has to allow regular expressions (instead of single characters) to label the transitions, as the next simple example demonstrates. Consider the automaton:



Then, eliminating state q_1 can be done if we re-label the transition from q_0 to q_2 by a regular expression:



It is easy to check that the latter automaton (i.e., without state q_1) accepts the same language as the former.

Let us now generalise the idea sketched in this example. Assume we want to remove some state q of an ε-NFA. Let $p_1, p_2, \ldots, p_n$ denote the *predecessors of* q, i.e., all states p_i s.t. $q \in \delta(p_i, a)$, for some $a \in \Sigma \cup \{\varepsilon\}$. Let us further denote by $s_1, s_2, \ldots, s_\ell$ the *successors of* q, i.e. all states s_i s.t. $s_i \in \delta(q, a)$ for some $a \in \Sigma \cup \{\varepsilon\}$. Obviously, the removal of q will affect all transitions from some p_i to q, and all transitions from q to some s_i. But it might also affect some transitions from some p_i to some s_j, as in the above example. In this case, q_0 is a predecessor of $q = q_1$; q_2 is a successor; and we 'report' the information from the two deleted transitions to the direct transition from q_0 to q_2. So, in general, the states and transitions we need to consider when deleting state q are as depicted in Figure 2.13 (left). Observe that we assumed two important things in this figure:

Observe that a state could be at the same time a successor and a predecessor of q, but this is not a problem for our technique.

1. first, all transitions are labeled by *regular expressions* $r_{i,j}$. This will be important because we will iteratively remove states and thus replace some characters labeling transitions by more complex regular expressions. Since each character is also a regular expression, this is not a problem: the initial automaton respects this assumption.

2. second, we have assumed that there is a transition between each pair of states (p_i, s_j) (with $1 \le i \le n$ and $1 \le j \le \ell$). This assumption is important because the information on the transitions we will delete along

with q will be moved to those transitions. If the automaton does not respect this hypothesis, we can always add transitions that are labeled by the regular expression $\emptyset$ without modifying the accepted language of the automaton.

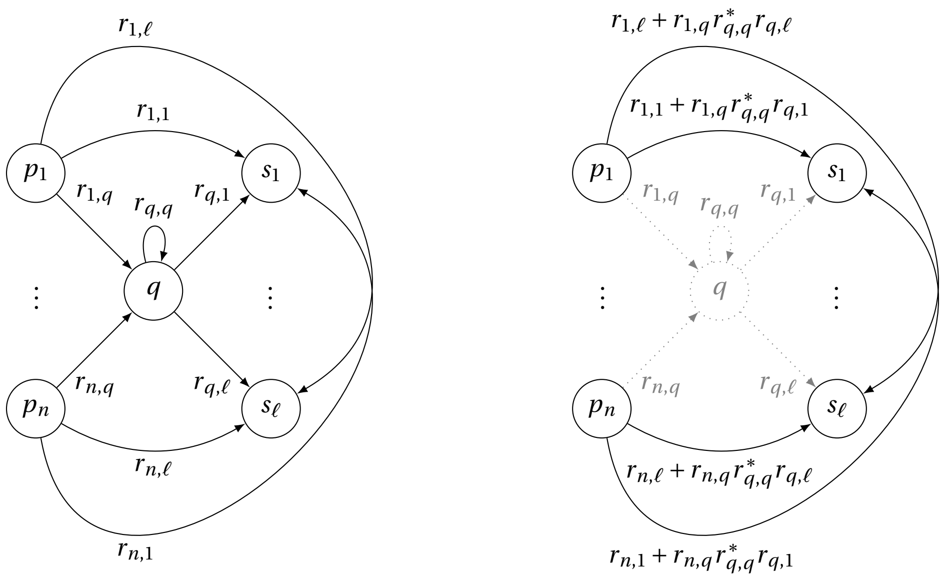


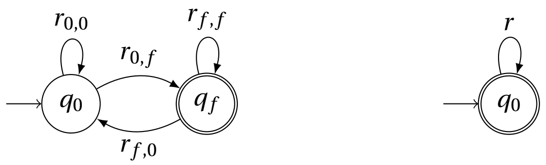
Figure 2.13: The situation before (left) and after (right) deletion of state q

Now, let us observe the right-hand side of Figure 2.13. It shows the automaton one obtains after removing all the part which is now dotted, i.e., q and its incoming and outgoing transitions. To justify this construction, let us consider, for instance, a run fragment from p_1 to s_ℓ. In the original automaton, this can be done at least in two different ways:

- either by following the direct transition from p_1 to s_ℓ, reading a word that matches $r_{1,\ell}$.
- or by following a path that goes from p_1 to q, follows an arbitrary number of times the self-loop on q, then goes from q to s_ℓ. For this path to be taken the automaton thus needs to read a word recognised by $r_{1,q} r^*_{q,q} r_{q,\ell}$.

As we delete state q, the regular expression $r_{1,q} r^*_{q,q} r_{q,\ell}$ corresponding to the latter path must now be reported to the former. Hence, the label from p_1 to s_ℓ now becomes $r_{1,\ell} + r_{1,q} r^*_{q,q} r_{q,\ell}$. The other modified labels are justified in a similar fashion. Since the deletion of q does not affect the rest of the automaton, we conclude that applying this transformation to any state q of any ε-NFA does not modify its accepted language.

Then, the algorithm to convert an ε-NFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ into a regular expression accepting the same language is as follows. For each accepting state $q_f \in F$, we build an equivalent automaton A_{q_f} by *deleting all states except* q_0 *and* q_f from A, using the state elimination procedure described above. Since all states but q_0 and q_f have been removed, A_{q_f} is necessarily of either forms shown in Figure 2.14. Indeed, only states q_0 and q_f are left, and it could be the case that $q_0 = q_f$. In both cases, computing the regular expression that corresponds to those automata is easy—they are displayed under the automata.



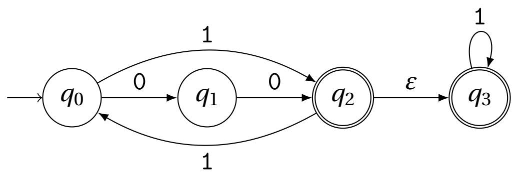
$\left(r_{0,0} + r_{0,f} \cdot r_{f,f}^{*} \cdot r_{f,0}\right)^{*} \cdot r_{0,f} \cdot r_{f,f}^{*}$ $\qquad$ r^{*}

Figure 2.14: The two possible forms for an automaton A_{q_f} obtained by eliminating all states but q_0 and q_f, and their corresponding regular expressions. We obtain the right automaton whenever $q_0 = q_f$.

So, for each accepting state q_f, we can now compute a regular expression r_{q_f} that accepts *all the words A accepts by a run ending in* q_f. However, the language of A is exactly the set of all words that A accepts by a run ending in either of the accepting states. Then, assuming that the set of accepting states of A is $F = \{q_f^1, q_f^2, \ldots q_f^n\}$, we obtain the regular expression corresponding to A as:

$$q_f^1 + q_f^2 + \cdots + q_f^n$$

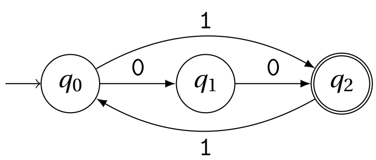
Example 2.16. As an example, consider the following ε-NFA:



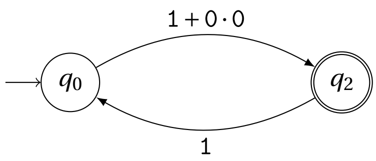
Remember that, when no transition is displayed between a pair of states (q_1, q_2)—potentially with $q_1 = q_2$—we assume that there *is* a transition labeled by the regular expression $\emptyset$. We do not display such transitions to enhance readability.

Then, we apply iteratively the state elimination procedure to obtain A_{q_2} and A_{q_3}:

1. To obtain A_{q_2}, we first observe that q_2 is not reachable from q_3 (in other words, all outgoing transitions from q_3 are labeled by $\emptyset$, except the self-loop). It is thus safe to delete q_3 without further modification of the transitions:



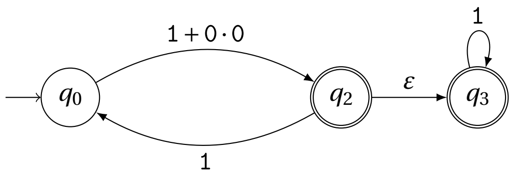
From this automaton, we apply the state elimination procedure to delete q_1 and obtain:



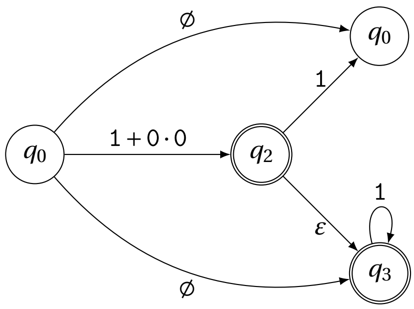
This automaton is A_{q_2}. Its corresponding regular expression is:

$$(1 \cdot 1 + 0 \cdot 0 \cdot 1)^* \cdot (1 + 0 \cdot 0)$$

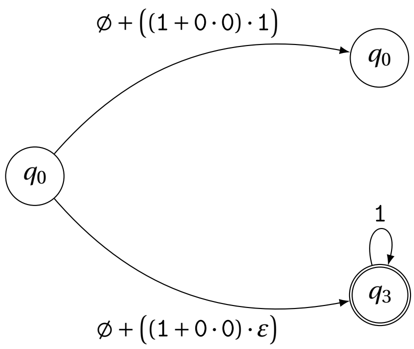
2. To obtain A_{q_3}, we first remove q_1, as in the previous case, and obtain:



Then, we remove q_2, which has one predecessor: q_0; and two successors: q_0 and q_3. Displaying q_2's situation as in Figure 2.13, we obtain:



Observe that we have duplicated q_0 for the sake of clarity, since it is both a predecessor and a successor. Now let us eliminate q_2, we obtain, using still the same representation:



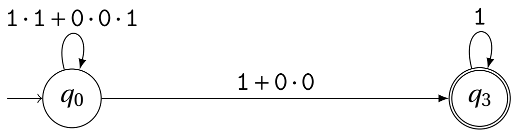
However, the newly introduced regular expressions can be simplified:

$$\begin{aligned} \emptyset + \big((1 + 0 \cdot 0) \cdot 1\big) &= \big((1 + 0 \cdot 0) \cdot 1\big) \\ &= 1 \cdot 1 + 0 \cdot 0 \cdot 1 \end{aligned}$$

and:

$$\begin{aligned} \emptyset + \big((1 + 0 \cdot 0) \cdot \varepsilon\big) &= \big((1 + 0 \cdot 0) \cdot \varepsilon\big) \\ &= 1 + 0 \cdot 0 \end{aligned}$$

Then, putting everything together (and taking into account that the duplicate q_0 is a single state), we obtain the automaton A_{q_3}:



Its corresponding regular expression is:

$$(1 \cdot 1 + 0 \cdot 0 \cdot 1)^* \cdot (1 + 0 \cdot 0) \cdot 1^*$$

So, we conclude that a regular expression accepting the same language as the original ε-NFA A is:

$$\big((1 \cdot 1 + 0 \cdot 0 \cdot 1)^* \cdot (1 + 0 \cdot 0)\big) + \big((1 \cdot 1 + 0 \cdot 0 \cdot 1)^* \cdot (1 + 0 \cdot 0) \cdot 1^*\big)$$

2.5 *Minimisation of* DFA*s*

As we have seen in Section 2.4.2 (see Figure 2.9), there can be several DFAs accepting the same language, and some of them might be larger than the others. It is thus natural to look for *a minimal* DFA *accepting a given regular language*, and to wonder whether *there can be several different minimal* DFA*s accepting the same language.*

Answers to those questions are provided by a central theorem of automata theory, which has been established in 1958 by Myhill and Nerode[12]. To avoid technicalities which are out of the scope of those notes, we will not state the theorem, but rather one of its consequence:

[12] A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):pp. 541–544, 1958. ISSN 00029939. URL http://www.jstor.org/stable/2033204

Corollary 2.4 (Consequence of the Myhill-Nerode theorem)**.** *For all regular languages L, there is a* unique *minimal* DFA *accepting L. This* DFA *can be computed from any* DFA *accepting L.*

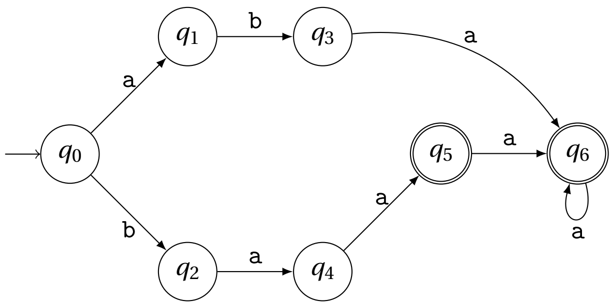


Figure 2.15: A DFA which is not minimal.

The aim of this section is to discuss the *minimisation procedure* for DFAs. Let us start with the simple example shown in Figure 2.15. This DFA is clearly *not minimal.* Consider for instance the two accepting states q_5 and q_6: it is easy to check that 'merging' them (preserving the a-labeled loop on the merged state) retains the language of the automaton, because, *both states accept the same language* $\mathtt{a}^*$, i.e., any word suffix read from q_5 will eventually be accepted iff it is accepted from q_6. This characterisation of states that 'can be merged' is the central notion that we need to minimise DFAs:

Definition 2.17 (Language acccepted from a state)**.** Given an ε-NFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$, and a state $q \in Q$, we let $L(A,q)$ be the *language accepted from* q defined as $L(A,q) = L(A')$ where $A' = \langle Q, \Sigma, \delta, q, F \rangle$ is the ε-NFA obtained by replacing the initial state of A by q.

In other words, $L(A,q)$ is the language A would accept if its initial state were q instead of q_0. Then, we can characterise states that we will be able to merge. Those states are said to be *equivalent*:

Definition 2.18 (Equivalence between states)**.** Let $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ be an ε-NFA, and let $q_1 \in Q$ and $q_2 \in Q$ be two states of A. Then, q_1 and q_2 are *equivalent* (denoted $q_1 \equiv q_2$) iff $L(A,q_1) = L(A,q_2)$.

It is easy to check that $\equiv$ is indeed an equivalence relation♣. For all states q, we denote by $[q]$ its equivalence class, i.e., the set of all states that accept the same language as q. Thanks to these definitions we can present the minimisation procedure for DFAs. As expected, it consists in 'merging' equivalent states and updating the transitions accordingly. Concretely, this amounts to using the set of equivalence classes of $\equiv$ as the states of the minimal automaton:

We will give, on page 57, arguments explaining why this definition of the transition relation is well-founded and makes sense.

Minimisation of DFAs

Given a DFA $A = \langle Q^A, \Sigma, \delta^A, q_0^A, F^A \rangle$, the minimal DFA accepting $L(A)$ is $B = \langle Q^B, \Sigma, \delta^B, q_0^B, F^B \rangle$ where:

1. $Q^B = \{[q] \mid q \in Q^A\}$
2. For all $[q] \in Q^B$, for all $a \in \Sigma$: $\delta^B([q], a) = \left[\delta^A(q,a)\right]$
3. $q_0^B = \left[q_0^A\right]$
4. $F^B = \{[q] \mid q \in F^A\}$.

Example 2.19. Let us consider the example in Figure 2.15. Here are the languages accepted by the different states (denoted as regular expressions):

State q	Accepted language $L(A,q)$
q_0	$\mathtt{a} \cdot \mathtt{b} \cdot \mathtt{a} \cdot \mathtt{a}^* + \mathtt{b} \cdot \mathtt{a} \cdot \mathtt{a} \cdot \mathtt{a}^*$
q_1	$\mathtt{b} \cdot \mathtt{a} \cdot \mathtt{a}^*$
q_2	$\mathtt{a} \cdot \mathtt{a} \cdot \mathtt{a}^*$
q_3	$\mathtt{a} \cdot \mathtt{a}^*$
q_4	$\mathtt{a} \cdot \mathtt{a}^*$
q_5	$\mathtt{a}^*$
q_6	$\mathtt{a}^*$

Clearly, $q_5 \equiv q_6$, $q_3 \equiv q_4$, but no other pair of states are equivalent. Thus, the equivalence classes (and also the states of the minimal DFA) are:

$$[q_0] = \{q_0\}$$
$$[q_1] = \{q_1\}$$
$$[q_2] = \{q_2\}$$
$$[q_3] = [q_4] = \{q_3, q_4\}$$
$$[q_5] = [q_6] = \{q_5, q_6\}$$

The minimal DFA is shown in Figure 2.16. ❦

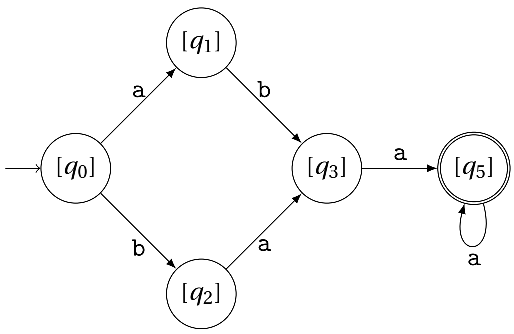


Figure 2.16: A minimal DFA.

As is, this technique is not really practical, since it requests to compute, for each state, its accepted language. A more efficient way of minimising DFAs is to compute directly the equivalence relations by a process called *partition refinement*. This algorithm is based on the two following observations:

1. It is not possible that an accepting state q_1 be equivalent to a non-accepting state q_2. Indeed, $\varepsilon \in L(A, q_1)$ (since q_1 is accepting and we are considering a DFA, hence an automaton without ε-transitions), but $\varepsilon \not\in L(A, q_2)$ (since q_2 is not accepting). Hence, $L(A, q_1)$ is necessarily different from $L(A, q_2)$.

2. If two states q_1 and q_2 are equivalent, then it must be the case that, for all letters a: $\delta(q_1, a) \equiv \delta(q_2, a)$. That is, reading the same letter from two equivalent states yields necessarily equivalent states.

   This can be shown by contradiction. Assume $q_1 \equiv q_2$ but $\delta(q_1, a) \not\equiv \delta(q_2, a)$ for some letter a. Since $\delta(q_1, a) \not\equiv \delta(q_2, a)$, the language accepted from $\delta(q_1, a)$ must be different from the language accepted from $\delta(q_2, a)$, by definition of the equivalence relation (Definition 2.17). Hence, there is at least one word w that differentiates these two languages. Without loss of generality, let us assume that w can be accepted from $\delta(q_1, a)$ but not from $\delta(q_2, a)$. Since we consider DFAs, we conclude that $a \cdot w \in L(A, q_1)$, but that $a \cdot w \not\in L(A, q_2)$. Hence, it is not possible that $q_1 \equiv q_2$.

Then, the partition refinement procedure consists in refining, iteratively, a symmetrical and reflexive relation♣ $\sim$ on the states, s.t. two states q_i and q_j are kept in relation ($q_i \sim q_j$) as long as they are believed to be equivalent (or, in other words, as long as they have not been proved to be non-equivalent). Initially, all final states are in relation with each others, and all non-final states are too. However, no final state is in relation with a non-final one, since we know for sure that final and non-final states cannot be equivalent.

Observe that, once the algorithm has declared that $q_i \not\sim q_j$, then, we are sure that $q_i \not\equiv q_j$. However, $q_i \sim q_j$ does not imply that $q_i \equiv q_j$. The fact that $q_i \sim q_j$ only represents the current belief of the algorithm, but it could be revised later.

The current state of the relation is stored in a matrix P indexed by the states (in both dimensions). We let $P[q_i, q_j] = 1$ iff $q_i \sim q_j$. Since the relation is symmetrical and reflexive, there are only 1's on the diagonal and the matrix is symmetrical , and so we keep only the (strictly) upper triangular part of the matrix. For instance:

$$P: \begin{array}{cc|c} q_2 & q_3 & \\ \hline 0 & 1 & q_1 \\ & 0 & q_2 \end{array}$$

indicates that $q_1 \sim q_3$, but that $q_1 \not\sim q_2$ and that $q_2 \not\sim q_3$.

Then, the *refinement step* consists in finding two states q_i and q_j s.t.:

- $q_i \neq q_j$;
- q_i is currently believed to be equivalent to q_j, i.e., $P[q_i, q_j] = 1$; but
- there is a letter a s.t. $P[\delta(q_i, a), \delta(q_j, a)] = 0$.

Because $P[\delta(q_i, a), \delta(q_j, a)] = 0$, we *know for sure* that $\delta(q_i, a) \not\equiv \delta(q_j, a)$. Hence, as discussed above, it is not possible that $q_i \equiv q_j$, and so we put a 0 in $P[q_i, q_j]$. We go on like that as long as we can update some cells of the matrix. Algorithm 1 presents this algorithm.

Obviously, the algorithm terminates after having updated all cells of the matrix in the worst case. It is easy to check that it runs in polynomial time[13]. One can prove[14] that, upon termination, this algorithm computes exactly the relation $\equiv$ that we are looking for:

Proposition 2.5. *The refinement algorithm always terminates. Upon termination, $q_i \equiv q_j$ iff $P[q_i, q_j] = 1$, for all pairs of states (q_i, q_j).*

Example 2.20. Let us apply Algorithm 1 to the example in Figure 2.15. Remember that, following our convention, we have not shown, in the figure, a sink state q_s to which the automaton goes every time a transition is not represented explicitly (for instance, $\delta(q_1, \mathtt{a}) = q_s$). In the algorithm, however, we must take this state explicitly into account. So, we start with:

q_1	q_2	q_3	q_4	q_5	q_6	q_s	
1	1	1	1	0	0	1	q_0
	1	1	1	0	0	1	q_1
		1	1	0	0	1	q_2
			1	0	0	1	q_3
				0	0	1	q_4
					1	0	q_5
						0	q_6

because q_5 and q_6 are the only accepting states.

Then, the algorithm first treats the q_0 line and discovers that:

[13] Actually in $\mathcal{O}(n^5)$, which is not very good. A more clever implementation allows one to achieve $\mathcal{O}(n \log(n))$.

[14] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321455363

Input: A DFA $A = \langle Q = \{q_1, \ldots, q_n\}, \Sigma, \delta, q_0, F \rangle$.
Output: A strictly upper diagonal Boolean matrix P s.t. $P[q_i, q_j] = 1$ iff $q_i \equiv q_j$.

```
P ← strictly upper diagonal matrix of Boolean ;

foreach 1 ≤ i ≤ n do
    foreach i < j ≤ n do
        if q_i ∈ F ↔ q_j ∈ F then
            P[q_i, q_j] ← 1 ;
        else
            P[q_i, q_j] ← 0 ;

Boolean finished ← 0 ;

while ¬finished do
    finished ← 1 ;
    foreach 1 ≤ i ≤ n do
        foreach i < j ≤ n do
            if P[q_i, q_j] = 1 then
                foreach a ∈ Σ do
                    if P[δ(q_i, a), δ(q_j, a)] = 0 then
                        P[q_i, q_j] ← 0 ;
                        finished ← 0 ;

return P ;
```

Algorithm 1: The algorithm to compute the matrix encoding the equivalence classes of $\equiv$.

- q_0 is not equivalent to q_3 because we have that $\delta(q_0, \mathtt{a}) = q_1$ and $\delta(q_3, \mathtt{a}) = q_6$; but $P[q_1, q_6] = 0$;
- q_0 is not equivalent to q_4 because we have that $\delta(q_0, \mathtt{a}) = q_1$ and $\delta(q_4, \mathtt{a}) = q_5$; but $P[q_1, q_5] = 0$;

and updates the q_0 line accordingly. Then, the algorithm processes the q_1 line, and discovers that:

- q_1 is not equivalent to q_3 because we have that $\delta(q_1, \mathtt{a}) = q_s$ and $\delta(q_3, \mathtt{a}) = q_6$; but $P[q_s, q_6] = 0$;
- q_1 is not equivalent to q_4 because we have that $\delta(q_1, \mathtt{a}) = q_s$ and $\delta(q_4, \mathtt{a}) = q_5$; but $P[q_s, q_5] = 0$;

and updates the q_1 line too. Then, for similar reasons, it discovers that q_2 is not equivalent neither to q_3 nor to q_4. It also finds that neither q_3 nor q_4 can be equivalent to q_s. At the end of the first iteration of the **while** loop, the matrix P is thus as follows:

q_1	q_2	q_3	q_4	q_5	q_6	q_s	
1	1	**0**	**0**	0	0	1	q_0
	1	**0**	**0**	0	0	1	q_1
		0	**0**	0	0	1	q_2
			1	0	0	**0**	q_3
				0	0	**0**	q_4
					1	0	q_5
						0	q_6

At the next step the algorithm discovers that:

- q_0 is not equivalent to q_1, because: $\delta(q_0, \mathtt{b}) = q_2$ and $\delta(q_1, \mathtt{b}) = q_3$, but $P[q_2, q_3] = 0$;
- q_0 is not equivalent to q_2, because: $\delta(q_0, \mathtt{a}) = q_1$ and $\delta(q_2, \mathtt{a}) = q_4$, but $P[q_1, q_4] = 0$;
- q_1 is not equivalent to q_2, because: $\delta(q_1, \mathtt{b}) = q_3$ and $\delta(q_2, \mathtt{b}) = q_s$, but $P[q_3, q_s] = 0$;
- q_1 is not equivalent to q_s, because: $\delta(q_1, \mathtt{b}) = q_3$ and $\delta(q_s, \mathtt{b}) = q_s$, but $P[q_3, q_s] = 0$;
- q_2 is not equivalent to q_s, because: $\delta(q_2, \mathtt{a}) = q_4$ and $\delta(q_s, \mathtt{a}) = q_s$, but $P[q_4, q_s] = 0$.

At the end of the second iteration of the **while** loop, the matrix P is thus as follows:

q_1	q_2	q_3	q_4	q_5	q_6	q_s	
0	**0**	0	0	0	0	1	q_0
	0	0	0	0	0	**0**	q_1
		0	0	0	0	**0**	q_2
			1	0	0	0	q_3
				0	0	0	q_4
					1	0	q_5
						0	q_6

Finally, during the third and last iteration of the **while** loop, the algorithm discovers that q_0 is not equivalent to q_s because $\delta(q_0, \mathtt{a}) = q_1$, $\delta(q_s, \mathtt{a}) = q_s$, but $P[q_1, q_s] = 0$. Hence, the final matrix is:

q_1	q_2	q_3	q_4	q_5	q_6	q_s	
0	0	0	0	0	0	**0**	q_0
	0	0	0	0	0	0	q_1
		0	0	0	0	0	q_2
			1	0	0	0	q_3
				0	0	0	q_4
					1	0	q_5
						0	q_6

which indeed corresponds to the equivalence classes used to build the automaton in Figure 2.16.

2.6 Operations on regular languages

In this section, we consider different operations on sets that also apply to and are particularly relevant for languages, i.e., the union, the complement and the intersection. We also consider the problems of testing emptiness, inclusion and equality of regular languages. Of course, we want to realise all those operations and tests in an algorithmic way. Since regular languages are potentially infinite, we need to fix a finite representation for them. Unsurprisingly, we will rely on finite automata.

Union Given two ε-NFAs $A_1 = \langle Q^1, \Sigma, \delta^1, q_0^1, F^1\rangle$ and $A_2 = \langle Q^2, \Sigma, \delta^2, q_0^2, F^2\rangle$, building an ε-NFA that accepts $L(A_1) \cup L(A_2)$ is easy: it amounts to adding a fresh initial state that can, by means of an ε-transition, jump to either q_0^1 or q_0^2. That is, if we let:

Remember that the $\uplus$ symbol denotes the 'disjoint union', i.e.: $A \uplus B$ is $A \cup B$ assuming $A \cap B = \emptyset$. We use it to formalise the facts that the set of states of both automata should be disjoint, and that the initial state q_0 is a 'newly created' state.

$$A = \langle Q^1 \uplus Q^2 \uplus \{q_0\}, \Sigma, \delta', q_0, F^1 \uplus F^2\rangle$$

where, for all $q \in Q^1 \uplus Q^2 \uplus \{q_0\}$, all $a \in \Sigma \cup \{\varepsilon\}$:

$$\delta(q,a) = \begin{cases} \{q_0^1, q_0^2\} & \text{if } q = q_0 \text{ and } a = \varepsilon \\ \delta^1(q,a) & \text{if } q \in Q^1 \\ \delta^2(q,a) & \text{if } q \in Q^2 \\ \emptyset & \text{otherwise} \end{cases}$$

then, $L(A) = L(A^1) \cup L(A^2)$.

Observe that the resulting automaton is necessarily an ε-NFA, even if A^1 and A^2 are DFAs. It can of course be determinised, like every ε-NFA, using the procedure discussed in Section 2.4.2.

Complement Given an ε-NFA A, we want to compute an ε-NFA $\overline{A}$ s.t. $L(\overline{A}) = \overline{L(A)} = \Sigma^* \setminus L(A)$. Probably the first idea that comes to one's mind when looking for a technique to complement automata is to 'swap accepting and non-accepting states'. This idea, unfortunately, does not work in general as shown in Figure 2.17: the automaton (on alphabet $\Sigma = \{\mathtt{a}\}$) in the figure accepts $\mathtt{a}^*$, so, the complement of its language is $\Sigma^* \setminus \mathtt{a}^* = \emptyset$.

However, the automaton obtained from the one in Figure 2.17 by having q_2 only as accepting state accepts $\mathtt{a}^+ \neq \emptyset$.

From this example, it is clear that the problem comes from the non-determinism. A DFA, however, has exactly one execution on each word w that ends in an accepting state iff w is accepted. So, swapping accepting and non-accepting states of a DFA A, and keeping the rest of the automaton identical yields an automaton $\overline{A}$ accepting the complement of A's language. On each word w, the sequence of states traversed by $\overline{A}$ will be the same as in A. Only the final state will be accepting in $\overline{A}$ iff it is rejecting in A.

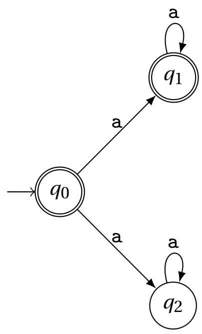


Figure 2.17: Swapping accepting and non-accepting states does not complement non-deterministic automata.

To sum up, for a DFA $A = \langle Q, \Sigma, \delta, q_0, F\rangle$, we let:

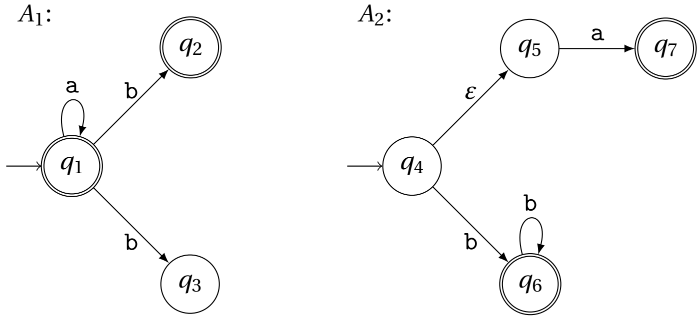
$$\overline{A} = \langle Q, \Sigma, \delta, q_0, Q \setminus F\rangle$$

If the given automaton is not deterministic, we first determinise it using the procedure of section 2.4.2.

Intersection To compute the intersection of two finite automata languages $L(A_1)$ and $L(A_2)$, we build a new FA A that simulates, at the same time, the executions of A_1 and A_2 on the same word. To do this, A needs to remember the current states of A_1 and A_2. So A's states are pairs of states (q_1, q_2), where q_1 is a state of A_1 and q_2 is a state of A_2. Moreover, the transition function of A must reflect the possible moves of both automata when reading the same letter a.

As an example, consider the automata A_1 and A_2 given hereunder. They accept $\mathtt{a}^* \cdot (\mathtt{b} + \varepsilon)$ and $\mathtt{a} + \mathtt{b}^+$ respectively:

A_1: A_2:



Obviously, the initial state of A will be (q_1, q_4), since they are the respective initial states of A_1 and A_2. From this state, we can consider several options, for the transition function:

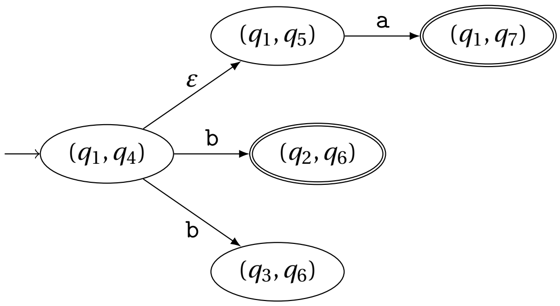
1. Either the input word begins with an a. A_1 can read this a, but A_2 cannot since there is not a-labeled transition from q_4, A_2's current state. Thus, there is no a-labeled transition from (q_1, q_4) in A.

2. Or, the input word begins with a b. Both automata can read this letter: A_1 will move either to q_2 or to q_3, and A_2 will move to q_6. Hence, in A, (q_1, q_4) has two b-labeled successors: (q_2, q_6) and (q_3, q_6). Only (q_2, q_6) is accepting, since q_2 and q_6 are *both* accepting.

3. Or, one of the automata (in this case, A_2) makes a spontaneous move, while the other (A_1) is left unchanged. This is possible because there

is an ε-labeled transition from q_4 to q_5 in A_2. Hence, there is, in A, an ε-labeled transition from (q_1, q_4) to (q_1, q_5).

Continuing this construction, we obtain the automaton hereunder:



It is easy to check that this automaton accepts a+b which is indeed $L(A_1) \cap L(A_2)$.

Formally, assume $A_1 = \langle Q^1, \Sigma, \delta^1, q_0^1, F^1 \rangle$ and $A_2 = \langle Q^2, \Sigma, \delta^2, q_0^2, F^2 \rangle$ are two ε-NFAs Then, we let $A_1 \cap A_2$ be the ε-NFA $\langle Q, \Sigma, \delta, q_0, F \rangle$ where:

Remember that the *product* $A \times B$ of two sets A and B is the set of all pairs (a, b) s.t. the former element a belongs to A and the latter b, to B. For instance: $\{1,2\} \times \{3,4\} = \{(1,3),(1,4),(2,3),(2,4)\}$.

1. $Q = Q^1 \times Q^2$;
2. For all $(q_1, q_2) \in Q$, for all $a \in \Sigma \cup \{\varepsilon\}$, $\delta\big((q_1, q_2), a\big)$ contains (q_1', q_2') iff one of the following holds:
   - $q_1' \in \delta(q_1, a)$ and $q_2' \in \delta(q_2, a)$; or
   - $a = \varepsilon$, $q_1' \in \delta(q_1, \varepsilon)$ and $q_2' = q_2$; or
   - $a = \varepsilon$, $q_1' = q_1$ and $q_2' \in \delta(q_2, \varepsilon)$.
3. $q_0 = (q_0^1, q_0^2)$;
4. $F = F^1 \times F^2$.

The following can easily be established by induction on the length of the input words:

Theorem 2.6. *Let A_1 and A_2 be two ε-NFAs. Then, $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$.*

Finally, observe that the construction for intersection can easily be modified to obtain an alternative algorithm to compute the union of two ε-NFAs, simply by letting the set of final states be $\{(q^1, q^2) \mid q^1 \in F^1 \text{ or } q^2 \in F^2\}$. The advantage of this construction is that it produce a DFA when applied to two DFAs A_1 and A_2, unlike the previous one that needs ε-transitions.

Emptiness Clearly, an ε-NFA accepts a word iff there exists a path in the automaton from the initial state to any accepting state, whatever the word recognised along the path. Thus, testing for emptiness of ε-NFAs boils down to a graph problem, which can be solved by classical algorithms such as breadth- or depth-first search. Algorithm 2 shows a variation of the breadth-first search to check for emptiness of ε-NFAs. At all times, it maintains a set Passed of states that it has already visited and a set Frontier containing all the states that have been visited for the first time at the previous iteration of the **While** loop. Each iteration of this loop consists in computing a new Frontier set: it contains all direct successors of nodes from Frontier that are not in Passed. The loop terminates either when all states have been explored (Frontier $= \emptyset$) or when an accepting state has been reached (Passed $\cap F \neq \emptyset$).

Input: An ε-NFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$
Output: True iff $L(A) = \emptyset$
Passed $\leftarrow \emptyset$;
Frontier $\leftarrow \{q_0\}$;
while Frontier $\neq \emptyset$ *and* Passed $\cap F = \emptyset$ **do**
    Passed $\leftarrow$ Passed $\cup$ Frontier ;
    NewFrontier $\leftarrow \emptyset$;
    foreach $q \in$ Frontier **do**
        foreach $a \in \Sigma \cup \{\varepsilon\}$ **do**
            NewFrontier $\leftarrow$ NewFrontier $\cup \big(\delta(q, a) \setminus \text{Passed}\big)$;
    Frontier $\leftarrow$ NewFrontier ;
return Passed $\cap F = \emptyset$;

Algorithm 2: Checking for emptiness of ε-NFAs.

Language inclusion Given two ε-NFAs A_1 and A_2, we would like to check whether $L(A_1) \subseteq L(A_2)$, i.e., whether all words accepted by A_1 are also accepted by A_2. To do so, we can rely on the machinery we have developed before. Indeed, it is easy to check that:

$$L(A_1) \subseteq L(A_2)$$
$$\text{iff}$$
$$L(A_1) \cap \overline{L(A_2)} = \emptyset$$

Indeed, if $L(A_1) \subseteq L(A_2)$, then all words $w \in L(A_1)$ do *not* belong to $\overline{L(A_2)}$ (otherwise, they would be rejected by A_2). Hence, there is certainly no intersection between $L(A_1)$ and $\overline{L(A_2)}$. On the other hand, if $L(A_1) \cap \overline{L(A_2)}$ is empty, this means that there is no word w which is (i) accepted by A_1 and (ii) rejected by A_2. Thus, all words that are accepted by A_1 are also accepted by A_2, hence $L(A_1) \subseteq L(A_2)$.

Checking whether $L(A_1) \cap \overline{L(A_2)} = \emptyset$ can be done by using the techniques we have described above: by first building the automaton $A = A_1 \cap \overline{A_2}$, then checking whether $L(A) = \emptyset$ using Algorithm 2.

In practice, however, the operations (determinising and complementing A_2, computing the intersection and checking whether it is empty) can be carried on-the-fly: this allows to stop the algorithm (and potentially avoid a costly determinisation) as soon as a word accepted by A_1 and rejected by A_2 is found. This on-the-fly algorithm will not be detailed here. It allows to prove that the language inclusion problem belongs to PSPACE.

Equality testing To test whether $L(A_1) = L(A_2)$, for two ε-NFAs A_1 and A_2, one can simply check whether $L(A_1) \subseteq L(A_2)$ and $L(A_2) \subseteq L(A_1)$. Again, an efficient, on-the-fly, version of this algorithm better be implemented in practice.

2.7 Exercises

2.7.1 Definition of regular languages

Exercise 2.1. Consider the alphabet $\Sigma = \{0, 1\}$. Using the inductive definition of regular languages, prove that the following languages are regular:

The definition of regular languages is Definition 2.1.

1. The set of words made of an arbitrary number of ones, followed by 01, followed by an arbitrary number of zeroes.

2. The set of odd binary numbers.

Exercise 2.2. Prove that any finite language is regular. Is the language $L = \{0^n 1^n \mid n \in \mathbb{N}\}$ regular? Give an intuition of why or why not.

Problem 2.3. Prove that, for all languages L and M: $(L^* M^*)^* = (L \cup M)^*$. Problem taken from Niwińsky and Rytter[15].

[15] Damian Niwińsky and Wojciech Rytter. *200 Problems in Formal Languages and Automata Theory*. University of Warsaw, 2017

2.7.2 Finite automata

Exercise 2.4. For each of the following languages (defined on the alphabet $\Sigma = \{0, 1\}$), design a nondeterministic finite automaton (NFA) that accepts it:

We have defined the classes of finite automata in Section 2.3.

1. the set of strings ending with 00;

2. the set of strings whose 3rd symbol, counted from the end of the string, is a 1;

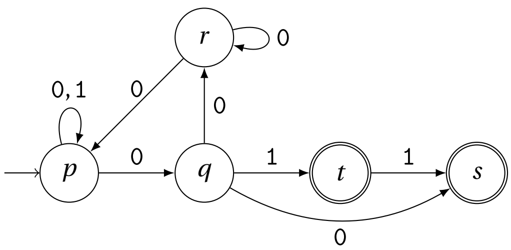
3. the set of strings where each pair of zeroes is directly followed by a pair of ones;

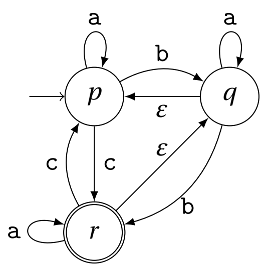
4. the set of strings not containing 101;

5. the set of binary numbers divisible by 4.

Exercise 2.5. Transform the following ε-NFA into DFA:

See the procedure for determinising automata in Section 2.4.2.





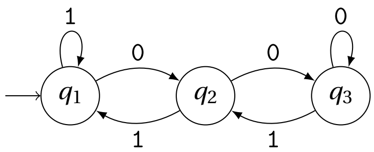
2.7.3 *Regular expressions*

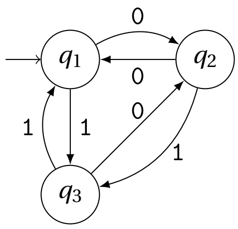
Exercise 2.6. Consider again all the languages from Exercise 2.4, and give a regular expression that define each of them.

The inductive definition of regular expressions is Definition 2.3.

Exercise 2.7. For each of the following DFA, give a regular expression accepting the same language:

A procedure to turn RE into ε-NFA has been given in Section 2.4.3.





Exercise 2.8. Convert the following RE into ε-NFA:

A procedure to turn RE into ε-NFA has been given in Section 2.4.1.

1. $\mathtt{01}^*$

2. $(\mathtt{0}+\mathtt{1})\mathtt{01}$

3. $\mathtt{00}(\mathtt{0}+\mathtt{1})^*$

2.7.4 *Extended regular expressions*

For the next exercises, you are asked to provide regular expressions using the 'extended regular expression' format (see Section 2.2.1), that is used in practice. You can test your answers using the regular expression library[16] `re` in Python, with its `re.search(pattern, string, flags=0)` method. The method receives an extended regular expression as the `pattern`, and returns a `Match object` indicating the first substring of `string` (if any) that matches the pattern. For example:

```
>>> import re
>>> re.search("(a|b|c)+","abcbcab")
<re.Match object; span=(0, 7), match='abcbcab'>
>>> re.search("(a|b|c)+","abcdef")
<re.Match object; span=(0, 3), match='abc'>
>>> re.search("(a|b|c)+","decbaf")
<re.Match object; span=(2, 5), match='cba'>
>>> re.search("(a|b|c)+","def")
```

Observe that the last call returns nothing because no match was possible.

[16] Python Software Foundation. re – Regular expression operations. https://docs.python.org/3/library/re.html. Online: accessed on April 12th, 2023

Exercise 2.9. Give an extended regular expression (ERE) that accepts any sequence of 5 characters, including the newline character `\n`.

Exercise 2.10. Give an ERE that accepts any string starting with an arbitrary number of \ followed by any number of $*$.

Exercise 2.11. UNIX-like shells (such as `bash`) allow the user to write *batch* files in which comments can be added. A line is defined to be a comment if it starts with a # sign. What ERE accepts such comments?

Exercise 2.12. Design an ERE that accepts numbers in scientific notation. Such a number must contain at least one digit and has two optional parts:

- a *decimal* part: a dot followed by a sequence of digits; and
- an *exponent* part: an `E` followed by an integer that may be prefixed by `+` or `-`.

For example, the following strings are valid numbers in scientific notation: `42`, `66.4E-5`, `8E17`

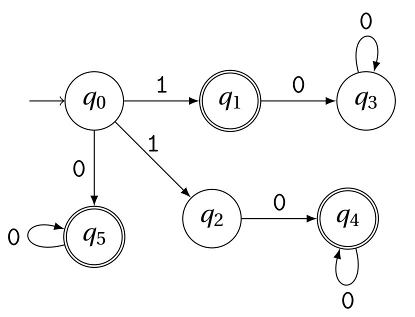
Exercise 2.13. Design an ERE that accepts 'correct' sentences that fulfill the following criteria: (i) no prepending/appending spaces; (ii) the first word must start with a capital letter; (iii) the sentence must end with a dot `.`; (iv) the phrase must be made of one or more words (made of the characters `a...z` and `A...Z`) separated by a single space; (v) there must be one sentence per line; and (vi) punctuation signs other than the dot are not allowed.

Exercise 2.14. Give an ERE that accepts old school DOS-style filenames respecting the following criteria. First, each filename starts by 8 characters (among `a...z`, `A...Z` and `_`), and the first five characters must be `abcde`. Next, each filename has an extension which is `.ext`. Finally, the ERE must accept accept the filename *only* (i.e., without the extension)!

For example, on `abcdeLOL.ext`, the ERE must accept `abcdeLOL`.

2.7.5 Minimisation of automata and other operations

Exercise 2.15. Here is a finite automaton:



We want to compute a *minimal* DFA that accepts the same language as this automaton:

1. First, determinise this automaton using the subset construction technique.
2. Is the resulting automaton *minimal* (in terms of number of states)?
3. For each state q of the resulting DFA, give, as a regular expression, the language L_q that the automaton would accept if q were the initial state.

The subset construction technique is the method of Section 2.4.2.

4. Based on the information computed at the previous point, propose a smaller DFA accepting the same language as the original automaton.

The method to minimise DFA is in Section 2.5, Algorithm 1.

5. Finally, apply the systematic method to minimise DFA and compare the results.

Exercise 2.16. Consider again the finite automaton at the beginning of Exercise 2.15. Let us denote this automaton by $A = \langle Q, \Sigma, \delta, q_0, F \rangle$. Then:

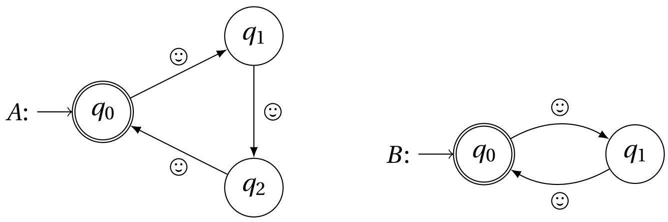
1. draw the automaton $A' = \langle Q, \Sigma, \delta, q_0, Q \setminus F \rangle$. How can you describe the relationship between A and A' in plain english?

2. what is the relationship between $L(A)$ and $L(A')$? Do they have a non-empty intersection? Is it the case that $L(A)$ is the complement of $L(A')$?

The method to complement a finite automaton is found in Section 2.6

3. . Apply the systematic method to compute the complement of a finite automaton, and check that the result indeed accepts the complement of $L(A)$.

Exercise 2.17. Here are two finite automata A and B on the singleton alphabet {☺}:



1. Describe, in plain english, the respective languages of these automata. Then, describe, again in plain english, the intersection of these two languages.

The algorithm to compute the intersection is in Section 2.6 again.

2. Use the method to compute the intersection of two finite automata on A and B, and check whether the result matches your intuition.

2.7.6 *The scanner generator JFlex*

For this part of the exercises, we will rely on the scanner generator JFlex. A *scanner* is a program that reads text on the standard input and prints it on standard output after applying operations. For example, a filter that replaces all `a` with `b` and that receives `abracadabra` on input would output `bbrbcbdbbrb`. Then, JFlex is a tool that generates such a scanner based on a set of regular expressions that specify which part of the input should be matched and modified. To recognise these regular expressions, JFlex is based on the theory of finite automata that we have studied. The generated scanned is, in fact, a Java function.

JFlex can be dowloaded from `http://jflex.de` and a user manual is at `http://jflex.de/manual.html`.

We start by a very short explanation of the tool.

Specification format A JFlex specification is made of three parts separated by lines with `%%`:

1. the first part is the user code. It can contain any Java code, that will be added at the beginning of the generated scanner.

2. the second part contains options and declarations.

   The options include:

   - `%class Name` to tell JFlex to produce a scanner inside the classe called `Name`;
   - `%unicode` to enable unicode input;
   - `%line` and `%column` to enable line and column counting respectively;
   - `%standalone` to generate a scanner that is not called by a parser.

   Then, some extra Java code included between `%{` and `%}` can be generated. It will be copied verbatim *inside* the generated Java class (contrary to the code of the first part which appears outside of the class).

   Finally, some ERE can be defined. They will be used as macros in part 3 of the file to enhance readability. For example:

```
Comment    = "/*" [^*] ~"*/" | "/*" "*"+ "/"
```

   defines the macro `Comment` and associates it to the given ERE.

3. the third part contains the core of the scanner. It is a series of rules that associate actions (in terms of Java code) to the regular expressions. Each rule is of the form:

```
Regex    {Action}
```

   where:

   - `Regex` is an *extended regular expression* (ERE), that can use some of the regular expressions defined in part 2 as macros (using curly braces around their names, for example: `{Comment}`);
   - `Action` is a *Java code snippet* that will be executed each time a *token* matching `Regex` is found.

   For example, the rule:

```
"=="        { return symbol(sym.EQEQ); }
```

   instructs the scanner to return `sym.EQEQ` every time == is found on the input.

The reader is advised to have a look at the JFlex documentation[17] for a comprehensive example

[17] Gerwin Klein, Steve Rowe, and Régis Décamps. Jflex user's manual. https://jflex.de/manual.html, March 2023. Version 1.9.1. Online: accessed on April, 12th, 2023

Variables and special actions When writing *actions*, some special variables and macros can be accessed:

- `yylength()` contains the *length* of the recognized token
- `yytext()` is a the actual string that was matched by the regular expression.

- `yyline` is the line counter (requires the option %line).
- `yycolumn` is the column counter (requires the option %column).
- `EOF` is the End-of-file marker .

On Mac, files which do not end with an empty line can make the lexer "forget about" the last line. When you test your lexer, make sure that your test files end with an empty line.

Meta states In order to track the progress of the scanner, *states* can be used. Each action can change or query the current state of the scanner. This amounts to having a finite automaton running in parallel to the scanner. This way, when one particular token is recognised, the scanner can change the current state, which allows to 'store' some information that can be checked during a further call to the scanner.

There are actually two kind of states:

- inclusive states, declared by `%state`, which acts as Booleans (the scanner can be in several of those states at a time);
- exclusive states, declared by `%xstate` which are mutually exclusive (like regular automata states).

Then, the rules can be associated to states, and are active only in these states. A state can be 'activated' using the function `yybegin(S)` in the code (where `S` is the name of the state to activate). Here is an example:

```
%%
xstate YYINITIAL, PRINT;
%%
<YYINITIAL> {
   "print" {yybegin(PRINT);}
}
<PRINT> {
   ";" {yybegin(YYINITIAL);}
   .   {System.out.println(yytext());}
}
```

Executable To obtain the scanner executable, follow these steps[18]:

1. Generate the scanner code with:

```
java -jar jflex-1.9.1.jar myspec.flex+
```

   which creates the file `Lexer.java` containing the `Lexer` class (the `%class` option can be used to change this);

2. compile the code into a class file: `javac Lexer.java` which creates `Lexer.class`;
3. run it with `java Lexer inputfile`.

[18] Assuming you are using version 1.9.1, which is the last version at the time of writing.

Here are now some exercises to get you familiar with JFlex:

Exercise 2.18. Write a scanner that outputs its input file with line numbers in front of every line.

Exercise 2.19. Write a scanner that outputs the number of alphanumeric characters, alphanumeric words and alphanumeric lines in the input file.

Exercise 2.20. Write a scanner that only shows the content of comments in the input file. Such comments are enclosed within curly braces `{ }`. You can assume that the input file does not contain curly braces inside comments.

Exercise 2.21. Write a scanner that transforms the input text as follows. It should replace the word `compiler` by `nope` if the line starts with an `a`; by `???` if it starts with a `b` and by `!!!` if it starts with a `c`.

Exercise 2.22. Write a *lexical analysis function* that recognises the following *tokens*:

- decimal numbers in scientific notation (e.g. `-0.4E-1`);
- C99 variable identifiers: they start with an an alphabetical symbol, followed by an arbitrary number of alphanumeric symbols or underscores;
- relational operators (<, >, ==, `!=`, >=, <=, `!`)
- The keywords `if`, `then` and `else`.

Each call to the function must seek the next token on the input. Every time a token is found, your function must output a message of the form `TOKEN NAME: token` (for example: `C99VAR: myvariable`) and return an object `Symbol` containing the token type, its value and its position (line and column). Templates for the `Symbol` and `LexicalUnit` classes are provided on the *Université Virtuelle*.

3 Grammars

GRAMMARS ARE THE TOOL WE WILL USE TO SPECIFY THE FULL SYNTAX OF PROGRAMMING LANGUAGES. They are also the basic bulding block of the systematic construction technique of parsers that we will discuss in Chapter 5 and Chapter 6. Before giving the formal syntax and semantics of grammars, we start with a discussion on the limits of regular languages, to motivate the need for other, more expressive, formalisms.

3.1 The limits of regular languages and some intuitions

In the previous chapter, we have seen several examples of applications of finite automata, and thus, also, several examples of languages that are regular. We have also sketched[1] the intuitions explaining that the language $L_{()}$ of well-parenthesised expressions is *not* regular. Recall that the intuition was the following. To check that an expression is well-parenthesised, one scans the expression from the left to the right, and maintains, at all times, a counter that tracks *the number of pending open parenthesis.* Then, whenever an opening parenthesis is met, the counter is incremented. Whenever a closing parenthesis is found, the counter must be strictly positive (otherwise the word is rejected) and is decremented. At the end, the counter must be equal to 0 (all opened parenthesis have been closed, and no pending open parenthesis remain) for the word to be accepted. Observe that one cannot bound the value of the counter, because the length of the words in $L_{()}$ is not bounded. Intuitively, it seems that this *unbounded* counter is necessary to recognise words from $L_{()}$, and that such an *unbounded* counter cannot be coded in the *finite* structure of finite automata.

[1] see Example 1.9, and the discussion on page 14.

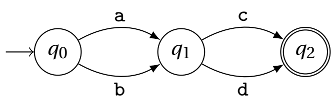


Figure 3.1: An automaton that 'forgets' whether the first letter was an a or a b.

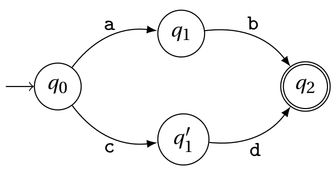


Figure 3.2: An automaton that 'remembers' whether the first letter was an a or a b.

Indeed, the only 'memory' that finite automata have is their set of states. To illustrate how states can be used as a 'memory' consider the language $\{\mathtt{ab}, \mathtt{cd}\}$. This language can be loosely characterised as: 'if the former letter is an a, then the latter should be a b; if the former is a c, the latter should be a d. So, to decide whether we should accept after reading the latter letter, we should *remember* the former one.

A first (and very naive!) attempt at building an automaton recognising $\{\mathtt{ab}, \mathtt{cd}\}$ could be the DFA in Figure 3.1. Clearly, this attempt fails because the automaton always ends up in state q_1 after reading the first letter, regardless of the letter. As a consequence the automaton accepts $\{\mathtt{ab}, \mathtt{cd}, \mathtt{ad}, \mathtt{bc}\}$. Of course, the automaton in Figure 3.2 now accepts the right language, because states q_1 and q_1' act as a memory: when the automaton reaches q_1, it has recorded that the first letter was an a; and when it reaches q'_1, the first letter was surely a c.

Now that we have a good intuition of what 'memory' means for a finite automaton, let us prove formally that $L_{()}$ is indeed not regular. Our proof strategy will be by contradiction: we will assume that $L_{()}$ *is* regular, and hence, that there exists an ε-NFA $A_{()}$ that recognises it (by Theorem 2.2). Then, we will exploit the fact that this hypothetical $A_{()}$ has *finitely many states* to derive a contradiction. Assuming $A_{()}$ has n states, we will select a word from $L_{()}$ which is 'very long': this means that the word will be long enough to guarantee that, when the automaton accepts it, an accepting run necessarily visits twice the same state. Coming back to our intuition that each state of a finite automaton represents a possible memory content, this means that, after reading two different prefixes w_1 and w_2—that contain a different number of pending open parenthesis—the automaton has memorised the exact same knowledge about those prefixes[2]. However, since w_1 and w_2 contain a different number of pending open parenthesis, the behaviour of the automaton should be different after reading these two prefixes. Unfortunately, since the automaton is in the same state in both cases, there will be at least one common execution after w_1 and w_2, i.e., by lack of memory, the automaton gets mixed up, and accepts words that are not part of $L_{()}$. Let us now formalise this intuition.

[2] Just as in the example of Figure 3.1, where the automaton has the same knowledge when it reads either a or c as the first letter.

Theorem 3.1. *$L_{()}$ is not regular*

Proof. The proof is by contradiction. Assume $L_{()}$ is regular, and let $A_{()}$ be an ε-NFA s.t. $L(A_{()}) = L_{()}$. Such an ε-NFA exists by Theorem 2.2. Let n be the number of states of $A_{()}$, and let us consider the word:

$$w = \underbrace{((\cdots(}_{n}\underbrace{))\cdots))}_{n}$$

Clearly, $w \in L_{()}$, hence w is accepted by $A_{()}$. Thus, there exists an accepting run of $A_{()}$ on w. Let us assume this run visits the sequence of states $q_0, q_1, \ldots q_{2n}$, which we represent as:

$$q_0 \xrightarrow{(} q_1 \xrightarrow{(} \cdots \xrightarrow{(} q_n \xrightarrow{)} q_{n+1} \cdots \xrightarrow{)} q_{2n}$$

where $q_i \xrightarrow{(} q_{i+1}$ means that the automaton moves from state q_i to state q_{i+1} by reading a ((and similarly for the) character).

Then, since $A_{()}$ has n states, the run portion that accepts the prefix $\underbrace{(\cdots(}_{n}$ must necessarily visit twice the same state[3]. Formally, we let k and ℓ be two positions s.t. $0 \le k < \ell \le n$ and $q_k = q_\ell$. In other words, the run portion between q_k and q_ℓ is actually a *loop* in the automaton, that we can repeat as many times as we want, obtaining a word which is not in $L_{()}$. More precisely, the run is of the form:

[3] One can invoke here the (in)famous 'pigeonhole principle' stating that if m pigeons occupy n holes, and there are strictly more pigeons than holes ($m > n$), then there is necessarily a hole that contains at least two birdies.

$$q_0 \xrightarrow{(} \cdots \xrightarrow{(} q_k \underbrace{\xrightarrow{(} \cdots \xrightarrow{(}}_{\text{loop}} q_\ell \xrightarrow{(} \cdots \xrightarrow{(} q_n \xrightarrow{)} q_{n+1} \cdots \xrightarrow{)} q_{2n}$$

Hence, the run obtained by repeating the loop twice is also an accepting run:

$$q_0 \xrightarrow{(} \cdots \xrightarrow{(} q_k \underbrace{\xrightarrow{(} \cdots \xrightarrow{(}}_{\text{loop}} \underset{=q_\ell}{q_k} \underbrace{\xrightarrow{(} \cdots \xrightarrow{(}}_{\text{loop}} q_\ell \xrightarrow{(} \cdots \xrightarrow{(} q_n \xrightarrow{)} q_{n+1} \cdots \xrightarrow{)} q_{2n}$$

However, since $\ell - k > 0$, this run accepts a word of the form:

$$w' = \underbrace{(\cdots\cdots(}_{m \text{ times}})\underbrace{\cdots\cdots)}_{n \text{ times}}$$

with $m > n$, which means that $w' \not\in L_{()}$, as not all opened parenthesis have been properly closed. Thus, we have just shown that $A_{()}$ accepts a word which is not in $L_{()}$. This contradicts our assumption that $L(A_{()}) = L_{()}$. Hence, the hypothetical automaton $A_{()}$ does not exist. Since there is no ε-NFA that accepts $L_{()}$, we conclude, by Theorem 2.2 that $L_{()}$ is not regular. □

This Theorem clearly shows that there are interesting and (arguably) simple languages that are not regular (hence, they cannot be specified by means of a regular expression, nor recognised by a finite automaton). This motivates the introduction of *grammars*, which are a much more powerful formalism for *specifying* languages (in the next chapter, we will study extensions of automata to handle more languages than finite automata can do).

Intuitive example In order to introduce grammars, we start with an intuitive example that somehow 'generalises' the language $L_{()}$. Let us consider a definition of 'expressions', which is inductive:

1. The sum of two expressions is an expression;
2. The product of two expressions is an expression;
3. An expression between matching parenthesis is an expression;
4. An identifier Id is an expression;
5. A constant Cst is an expression.

As an example, the string $(\mathsf{Cst} + \mathsf{Id}) * \mathsf{Id}$ is an expression but $)(\mathsf{CstId})$ is not, as can be checked with the definition.

This definition is fine, and can easily be applied to check whether a given string is a (syntactically correct) expression. However, we would like to have a more 'generative' way of defining expressions. To achieve this, we will rely on the idea of *rewrite system*. Roughly speaking, a rewrite system is a set of rules that allow one to modify a given string of symbols to obtain a new string. More precisely, each rewrite rule is of the form $\alpha \to \beta$, where α and β are strings of symbols. Such a rewrite rule means, intuitively, that the string of symbols α can be replaced (or 'rewritten') by β. For instance, given the rule $Ab \to Bc$, the string $aAbBc$ can be rewritten as $aBcBc$, by substituting Bc for Ab in the string.

Using this intuition, we can now set up a set of rewriting rules to generate any syntactically correct grammar (and only those grammars). To achieve this, we need to introduce certain *intermediary symbols* that we call *variables*. In our case, we need only one variable Exp that represents an expression. Then, an expression Exp can be rewritten following one of the five items used in the inductive definition above (i.e., as a sum of expressions, or a product of expressions, or...). This yields the following rules:

(1)	Exp	$\rightarrow$	Exp + Exp
(2)	Exp	$\rightarrow$	Exp ∗ Exp
(3)	Exp	$\rightarrow$	(Exp)
(4)	Exp	$\rightarrow$	Id
(5)	Exp	$\rightarrow$	Cst

Since all right-hand sides of the rules share the same variable, we often omit to repeat it, and rather present the rules as:

(1)	Exp	$\rightarrow$	Exp + Exp
(2)		$\rightarrow$	Exp ∗ Exp
(3)		$\rightarrow$	(Exp)
(4)		$\rightarrow$	Id
(5)		$\rightarrow$	Cst

It is easy to check that this set of rules indeed allows to *generate*, by successive rewriting the string $(\mathsf{Cst} + \mathsf{Id}) * \mathsf{Id}$, starting from the sequence Exp. Indeed, Exp can be rewritten as $\mathsf{Exp} * \mathsf{Exp}$, by rule 2. In this new sequence, the latter occurrence of Exp, can be rewritten as Id, by rule 5, yielding $\mathsf{Exp} * \mathsf{Id}$, and so forth. The whole sequence of rewriting is given hereunder:

$$\mathsf{Exp} \overset{2}{\Rightarrow} \mathsf{Exp} * \mathsf{Exp} \overset{4}{\Rightarrow} \mathsf{Exp} * \mathsf{Id} \overset{3}{\Rightarrow} (\mathsf{Exp}) * \mathsf{Id} \overset{1}{\Rightarrow} (\mathsf{Exp} + \mathsf{Exp}) * \mathsf{Id} \overset{4}{\Rightarrow} (\mathsf{Exp} + \mathsf{Id}) * \mathsf{Id} \overset{5}{\Rightarrow} (\mathsf{Cst} + \mathsf{Id}) * \mathsf{Id}$$

Note that the initial sequence of symbols Exp contains only one variable; that the last sequence $(\mathsf{Cst} + \mathsf{Id}) * \mathsf{Id}$ is actually a word (it contains only symbols from the language's alphabet); and that all the intermediary sequences contain symbols from the alphabet, *and* variables (that are eventually eliminated by rewriting). Let us now formalise these notions.

3.2 *Syntax and semantics*

Syntax The formal definition of grammar follows the intuitions we have sketched above:

Definition 3.1 (Grammar)**.** A *grammar* is a quadruplet $G = \langle V, T, P, S\rangle$ where:

- V is a finite set of *variables*;
- T is a finite set of *terminals*;
- P is a finite set of *production rules* of the form $\alpha \rightarrow \beta$ with:
  - $\alpha \in (V \cup T)^* V (V \cup T)^*$ and
  - $\beta \in (V \cup T)^*$
- $S \in V$ is a variable called the *start symbol*.

Example 3.2. Formally, the grammar that defines expressions is the tuple:

$$G_{\mathsf{Exp}} = \langle \{\mathsf{Exp}\}, \{\mathsf{Cst}, \mathsf{Id}, (,), +, *\}, P, \mathsf{Exp}\rangle$$

with:

$$P = \left\{ \begin{array}{c} \mathsf{Exp} \rightarrow \mathsf{Exp+Exp} \\ \mathsf{Exp} \rightarrow \mathsf{Exp*Exp} \\ \mathsf{Exp} \rightarrow \mathsf{(Exp)} \\ \mathsf{Exp} \rightarrow \mathsf{Id} \\ \mathsf{Exp} \rightarrow \mathsf{Cst} \end{array} \right\}$$

Observe that our definition of the syntax of grammars is very general. Rules are of the form $\alpha \rightarrow \beta$ with: $\alpha \in (V \cup T)^* V (V \cup T)^*$ and $\beta \in (V \cup T)^*$, which means that both left- and right-hand sides of each rules can contain variables and terminals. The only requirement is that the left-hand side contains at least one variable. The next example shows the kind of languages that one can define when left-hand sides of rules are not restricted to a single variable:

Example 3.3. Consider the grammar:

$$G_{ABC} = \langle \{A, B, C, S, S'\}, \{a, b, c\}, P, S \rangle$$

where P contains the following set of rules:

(1)	S	$\rightarrow$	ε
(2)		$\rightarrow$	S'
(3)	S'	$\rightarrow$	$ABCS'$
(4)		$\rightarrow$	ABC
(5)	AB	$\rightarrow$	BA
(6)	BA	$\rightarrow$	AB
(7)	AC	$\rightarrow$	CA
(8)	CA	$\rightarrow$	AC
(9)	BC	$\rightarrow$	CB
(10)	CB	$\rightarrow$	BC
(11)	A	$\rightarrow$	a
(12)	B	$\rightarrow$	b
(13)	C	$\rightarrow$	c

We claim that this grammar allows one to generate all words on the alphabet $\{a, b, c\}$ that contains the same number of a's, b's and c's. For instance, consider the word $aabcbc$. Since this word is not empty, we first apply rule 2. Then, we apply rules 3 and rule 4 to generate 2 A's, 2 B's and 2 C's:

$$S \stackrel{2}{\Rightarrow} S' \stackrel{3}{\Rightarrow} ABCS' \stackrel{4}{\Rightarrow} ABCABC$$

Then, we use rules 6 and 8 to move the latter A two positions to the left:

$$ABCABC \stackrel{8}{\Rightarrow} ABACBC \stackrel{6}{\Rightarrow} AABCBC$$

Finally, we use rules 11–13 to obtain the desired word:

$$AABCBC \stackrel{11}{\Longrightarrow} aABCB \stackrel{11}{\Longrightarrow} aaBCBC \cdots \stackrel{12}{\Longrightarrow} aabcbc$$

It is easy to generalise this example to any word on $\{a, b, c\}$ with the same number of a's, b's and c's, and to check that only those words can be generated by the grammar.

Unfortunately, this generality is very powerful and makes many problems on grammars undecidable. In Section 3.3, we will introduce restricted syntactic classes of grammars that are still sufficient in practice, but for which meaningful problems are decidable.

Actually, it is possible to encode a Turing machine into a grammar. The rough idea is to use the derivable strings of the grammar to describe the reachable configurations of the machine. A configuration of a Turing machine can be encoded as a word of the form $w_1 q w_2$, where w_1 is the content of the tape to the left of the head, and w_2 is the content of the tape (excluding blank characters) under and to the right of the head. Such a configuration can be encoded by the string $w_1 Q w_2$ where w_1 and w_2 contain only terminals, and Q is a variable. Then, if the machine can move from q to q', reading an a, replacing it by a b, and moving the head to the right, the grammar would contain the rule $Qa \to bQ'$, and so forth...

Semantics Let us now formally define the *semantics* of grammars, i.e., the set of words that a grammar can generate.

Definition 3.4 (Derivation). Let $G = \langle V, T, P, S\rangle$ be a grammar, and let γ and δ be s.t. $\gamma \in (V \cup T)^* V (V \cup T)^*$, and $\delta \in (V \cup T)^*$. Then, we say that δ *can be derived from* γ *(under the rules of* G*)*, written:

$$\gamma \Rightarrow_G \delta$$

iff there are $\gamma_1, \gamma_2 \in (V \cup T)^*$ and a rule $\alpha \to \beta \in P$ s.t.: $\gamma = \gamma_1 \cdot \alpha \cdot \gamma_2$ and $\delta = \gamma_1 \cdot \beta \cdot \gamma_2$.

On top of this definition, we introduce several notations and shorthands. When the grammar G is clear from the context, we will often omit it from $\Rightarrow_G$. That is, we write $\gamma \Rightarrow \delta$ instead of $\gamma \Rightarrow_G \delta$. Clearly, for all grammar G, $\Rightarrow_G$ is a *relation* on words from $(V \cup T)^*$. Thus, we denote by $\Rightarrow^*$ the reflexive and transitive closure of $\Rightarrow$, similarly to what we did for finite automata in Chapter 2. We also write $\gamma \Rightarrow^i \delta$ for $i \in \mathbb{N}$ iff δ can be derived from γ in i steps, i.e., there are $\gamma_1, \gamma_2, \ldots, \gamma_{i-1}$ s.t. $\gamma \Rightarrow \gamma_1 \Rightarrow \gamma_2 \cdots \gamma_{i-1} \Rightarrow \delta$.

Do not confuse $\gamma \Rightarrow^i \delta$, which means: '$\delta$ can be derived from γ in at most i steps' and $\gamma \overset{i}{\Rightarrow} \delta$, which means: '$\delta$ can be derived from γ by applying rule number i once'.

Finally, let us introduce the following important vocabulary:

Definition 3.5 (Sentential form). Let $G = \langle V, T, P, S\rangle$ be a grammar. A *sentential form* is a word from $(V \cup T)^*$ that can be derived from the start symbol. Formally: $\gamma \in (V \cup T)^*$ is a sentential form (of G) iff $S \Rightarrow^*_G \gamma$.

Of course, we will be interested in certain sentential forms: those that contain terminals only, because they form the *language of the grammar*:

Definition 3.6 (Language of a grammar). Let $G = \langle V, T, P, S\rangle$ be a grammar. The *language of* G is:

$$L(G) = \{w \in T^* \mid S \Rightarrow^*_G w\}$$

3.3 The Chomsky hierarchy

As sketched before, the definition of grammar we have given (Definition 3.1) is so general that many problems on grammars[4] are undecidable. Yet, as the example of grammar G_{Exp} clearly shows, grammars are excellent tools to describe the syntax of programming languages (among other potential applications). The aim of the *Chomsky hierarchy* we are about to introduce is to identify *syntactic classes* of grammars that are useful for practical and/or theoretical purposes.

[4] including the language membership problem, i.e., does a given word w belong to the language $L(G)$ of a given grammar G?

Definition 3.7 (Chomsky hierarchy). The *Chomsky hierarchy* is made up of four classes of grammars, defined according to syntactic criteria:

Class 0: Unrestricted grammars All grammars are in this class.

Class 1: Context-sensitive grammars A grammar $G = \langle V, T, P, S\rangle$ is *context sensitive* iff each rule $\alpha \rightarrow \beta \in P$ is s.t.: 1. either $\alpha = S$ and $\beta = \varepsilon$; 2. or $|\alpha| \leq |\beta|$ and S does not appear in β.

Class 2: Context-free grammars A grammar $G = \langle V, T, P, S\rangle$ is *context-free* iff each rule $\alpha \rightarrow \beta \in P$ is s.t.: $\alpha \in V$, i.e., the left-hand side is only one variable.

Class 3: Regular grammars A grammar $G = \langle V, T, P, S\rangle$ is *regular* iff it is *either* left-regular *or* right-regular:

Left-regular grammars G is left-regular iff each rule $\alpha \rightarrow \beta \in P$ is s.t. $\alpha \in V$ and either $\beta \in T^*$, or $\beta \in V \cdot T^*$.

Right-regular grammars G is left-regular iff each rule $\alpha \rightarrow \beta \in P$ is s.t. $\alpha \in V$ and either $\beta \in T^*$, or $\beta \in T^* \cdot V$.

Observe that a grammar that contains rules of the form $A \rightarrow wB$ and of the form $A \rightarrow Bw$ at the same time is *not* regular.

This definition calls for several comments. First, let us give some examples of grammars:

Example 3.8.

1. The grammar $G_{a^*} = \langle\{S\}, \{a\}, P, S\rangle$ where P contains the rules:

(1)	S	$\rightarrow$	Sa
(2)		$\rightarrow$	ε

   is left-regular and $L(G_{a^*}) = \{a\}^*$. Observe that replacing the first rule by $S \rightarrow aS$ yields a right-regular grammar that accepts the same language.

2. To the contrary, the grammar $G = \langle\{S\}, \{a\}, P, S\rangle$ where P contains the rules:

(1)	S	$\rightarrow$	Sa
(2)		$\rightarrow$	aS
(3)		$\rightarrow$	ε

   is *not* regular because it is neither left-regular nor right-regular. In other words, mixing left-recursive and right-recursive rules is not permitted in a regular grammar. Yet, the language accepted by G *is* still $\{a\}^*$ and thus regular.

3. The grammar G_{Exp} of Example 3.2 is context-free but not regular, because, for instance, of rule $\mathsf{Exp} \rightarrow \mathsf{Exp} + \mathsf{Exp}$.

4. The grammar G_{ABC} of Example 3.3 is context-sensitive but not context-free, because of rule $AB \rightarrow BA$ for instance (two variables on the left-hand side).

Then, let us discuss the name 'hierarchy'. This term seems to imply that the classes of grammars are contained into each other, in other words that each class 3 grammar is a class 2 grammar, each class 2 grammar is a class 1 grammar, and each class 1 grammar is a class 0 grammar. Unfortunately this is not the case, as shown by the next example:

Example 3.9. Consider the grammar with the two following rules:

(1)	S	$\rightarrow$	S'
(2)	S'	$\rightarrow$	ε

Obviously, this grammar is regular (class 3) because the first rule is of the form $S \rightarrow wS'$, with $w = \varepsilon \in T^*$; and the latter is of the form $S' \rightarrow w$ with $w = \varepsilon \in T^*$ again. It is also context-free (class 2) because the left-hand sides of all its rules are made up of only one variable. Of course, it is also unrestricted (class 0). But it is clearly not context-sensitive (class 1) because of the rule $S' \rightarrow \varepsilon$ where $S' \neq S$.

However, observe that the class 1 grammar

$$\langle \{S\}, \emptyset, \{S \rightarrow \varepsilon\}, S \rangle$$

accepts the same language as G.

So, while the Chomsky hierarchy does not form a hierarchy of *grammars*, it defines a *hierarchy of languages*. To establish this, let us begin with a few more definitions:

Definition 3.10 (Context-free language)**.** A language L is *context-free* iff there exists a context-free grammar G s.t. $L(G) = L$.

And, similarly:

Definition 3.11 (Context-sensitive language)**.** A language L is *context-sensitive* iff there is a context-sensitive grammar G s.t. $L(G) = L$.

Definition 3.12 (Recursively enumerable language)**.** A language L is *recursively enumerable* (or *recognisable*) iff there is a grammar G s.t. $L(G) = L$.

We denote by CFL and CSL and RE the sets of context-free, context-sensitive, and recursively enumerable languages respectively[5] Then, the next theorem justifies the name 'Chomsky *hierarchy*':

Recall that we have used the notation Reg to denote the set of regular languages.

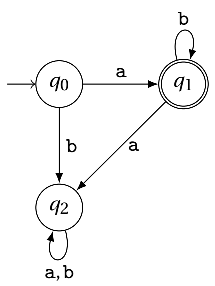
[5] Note that, by abuse of notation, we often write 'a CFL' or 'a CSL' to mean 'a language belonging to CFL' or 'a language belonging to CSL', etc.

Theorem 3.2. *For $i = 0, 1, 2, 3$ let us denote by $\mathscr{L}^i$ the class of languages recognised by type i grammars in the Chomsky hierarchy. Then:*

$$\mathscr{L}^3 = \mathsf{Reg} \subsetneq \mathscr{L}^2 = \mathsf{CFL} \subsetneq \mathscr{L}^1 = \mathsf{CSL} \subsetneq \mathscr{L}^0 = \mathsf{RE}$$

Proof. (Sketch) We will not prove the theorem with great detail, but we will highlight the main arguments behind each of those relations.

1. $\mathscr{L}^3 = \mathsf{Reg}$. We first show that each DFA A can be converted into a grammar G_A s.t. $L(A) = L(G_A)$. The intuition of the construction is as follows. The set of variables of G_A is the set of states of A. We will build the set of rules in such a way that all sentential forms are of the form wq, where $w \in \Sigma^*$ is the prefix read so far by the automaton, and q is the current state. Then, for each transition from some q_1 to some q_2 labeled by a, the grammar contains a rule $q_1 \rightarrow aq_2$. Finally, when an accepting state is reached, we must be able to get rid of the variable from the sentential form, so we have rules of the form $q \rightarrow \varepsilon$ for all $q \in F$. Fig. 3.3 shows an example of this transformation.



(1)	q_0	$\rightarrow$	$\mathtt{a}q_1$
(2)		$\rightarrow$	$\mathtt{b}q_2$
(3)	q_1	$\rightarrow$	$\mathtt{b}q_1$
(4)		$\rightarrow$	$\mathtt{a}q_2$
(5)		$\rightarrow$	ε
(6)	q_2	$\rightarrow$	$\mathtt{a}q_2$
(7)		$\rightarrow$	$\mathtt{b}q_2$

Figure 3.3: An example of a DFA and its corresponding right-regular grammar.

Formally, assuming $A = \langle Q, \Sigma, \delta, q_0, F \rangle$, we build the grammar $G_A = \langle Q, \Sigma, P, q_0 \rangle$, where:

$$P = \{q \to aq' \mid \delta(q,a) = q'\} \cup \{q \to \varepsilon \mid q \in F\}$$

Observe that the resulting grammar is right-regular, so all regular languages can be accepted by right-regular grammars. This shows that $\mathscr{L}^3 \supseteq$ Reg. To show Reg $\subseteq \mathscr{L}^3$, we must show that both right-regular and left-regular grammars define regular languages only. For right-regular grammars, we can adapt the arguments of the proof above and transform all right-regular grammars G into an ε-NFA A_G s.t. $L(G) = L(A_G)$. We turn each variable of the grammar into a state of the automaton. For all rules of the form $A \to wB$, we add transitions from A to B reading word w, adding intermediary states if appropriate. For all rules of the form $A \to w$, we let the automaton read the word w from state A and reach an accepting state. We do not give the details of the construction, but Fig. 3.4 gives an example.

(1)	S	$\to$	$\mathtt{aa}B$
(2)		$\to$	$\mathtt{a}$
(3)	B	$\to$	$\mathtt{b}B$
(4)		$\to$	ε

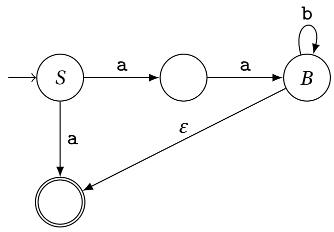


Figure 3.4: An example of a right-regular grammar and its corresponding ε-NFA.

Finally, it is possible to adapt this construction to build, for all *left-regular grammars* G, an ε-NFA A_G s.t. $L(A_G)$ is the mirror image of $L(G)$, i.e., $L(A_G)$ contains all words from G but reversed. Then, it is easy to modify A_G to let it accept $L(G)$, by swapping final and initial states[6] and reversing the directions of the transitions. We omit the details here.

[6] Adding a fresh initial state if necessary.

2. $\mathscr{L}^2 =$ CFL. This equality holds by definition (see Definition 3.10).

3. Reg $\subsetneq \mathscr{L}^2$. To show that Reg $\subseteq \mathscr{L}^2$, it suffices to observe that all regular grammars are also context-free (check Definition 3.7). Hence, $\mathscr{L}^3 \subseteq \mathscr{L}^2$, but since Reg $= \mathscr{L}^3$, we have Reg $\subseteq \mathscr{L}^2$. To show the strict inclusion, it is sufficient to exhibit a language which is a CFL but not regular. This is the case of $L_{()}$ (see beginning of the Chapter). We have shown, in Theorem 3.1 that $L_{()}$ is not regular. We can show that it is in CFL (hence a in $\mathscr{L}^2$) by providing a context-free grammar that defines it:

(1)	S	$\to$	SS
(2)		$\to$	(S)
(3)		$\to$	ε

It is easy to check that this grammar is indeed context-free and defines $L_{()}$, so $L_{()}$ is a CFL.

4. $\mathscr{L}^1 =$ CSL. Again, this holds by definition (Definition 3.11).

5. CFL $\subsetneq \mathscr{L}^1$. To prove this inclusion, we must first show that all context free languages (which can be defined by a context free grammar, since $\mathscr{L}^2 =$ CFL) are also context-sensitive languages. Unfortunately, as shown by Example 3.9 above, not all context-free grammars are context-sensitive, so we cannot use a simple and direct syntactic argument as we did when proving that all regular languages are also context-free. Observe that, in a context-free grammar, a rule of the form $\alpha \to \beta$ that violates the property $|\alpha| \leq |\beta|$ is necessarily a rule of the form $A \to \varepsilon$. Indeed, in a context-free grammar, all left-hand sides of rules contain only one variable. Thus $|\alpha| = 1$ in all rule $\alpha \to \beta$ and so, $|\alpha| > |\beta|$ means $|\beta| = 0$, hence $\beta = \varepsilon$.

Based on this observation, we propose a technique that turns any context-free grammar into an *equivalent* context-free grammar without rules of the form $A \to \varepsilon$, except when A is the start symbol, because this is the only case where an ε in the right-hand side is allowed in a context-sensitive grammar. The procedure works as follows. If a context-free grammar $G = \langle V, T, P, S\rangle$ contains a rule of the form $A \to \varepsilon$, with $A \neq S$, then:

(a) Remove from P the rule $A \to \varepsilon$:

$$P' = P \setminus \{A \to \varepsilon\}$$

(b) Find in P' all the rules of the form $B \to \beta$ where β contains A, and, for each of those rules, add to P' all the rule $B \to \beta'$, where β' has been obtained by removing all the A symbols[7] from β:

[7] Or, in other words, replacing all A's by ε.

$$P'' = P' \cup \left\{B \to \beta_1\beta_2\cdots\beta_n \mid B \to \beta_1 A\beta_2 A\cdots A\beta_n \in P' \text{ with } \beta_i \in (T \cup V \setminus \{A\})^* \text{ for all } i\right\}$$

This yields a new grammar $G' = \left\langle V, T, P'', S\right\rangle$. Observe that this transformation preserves the language of the grammar: $L(G') = L(G)$. We can iterate this process up to the point where no rule of the form $A \to \varepsilon$, with $A \neq S$ remain. Then, clearly, the resulting grammar $\overline{G}$ has the same language as the original one, and contains no rule of the form $A \to \varepsilon$, with $A \neq S$. Figure 3.5 illustrates this transformation.

Original grammar:

(1)	S	$\to$	Aa
(2)		$\to$	A
(3)		$\to$	Bb
(4)	A	$\to$	a
(5)		$\to$	ε
(6)	B	$\to$	ε

After treating $A \to \varepsilon$:

(1)	S	$\to$	Aa
(2)		$\to$	a
(3)		$\to$	A
(4)		$\to$	ε
(5)		$\to$	Bb
(6)	A	$\to$	a
(7)	B	$\to$	ε

After treating $B \to \varepsilon$:

(1)	S	$\to$	Aa
(2)		$\to$	a
(3)		$\to$	A
(4)		$\to$	ε
(5)		$\to$	Bb
(6)		$\to$	b
(7)	A	$\to$	a

Figure 3.5: Removing rule of the form $V \to \varepsilon$ in two steps. Observe that in the resulting grammar, the variable B does not produce any terminal, so we could also remove the rule $S \to Bb$, but what matters is that the language of the resulting grammar is the same as the original one.

However, we are not done yet, because the definition of context-sensitive grammar also asks that S does never occur in a right-hand side of rule. Again, we propose a transformation that eliminates such rules while preserving the language of the grammar. It works as follows:

(a) Add a new variable S' to the grammar:

$$V' = V \uplus \{S'\}$$

(b) For each rule of the form $S \to \beta$ with $\beta \neq \varepsilon$, add to the grammar the rule $S' \to \beta$:

$$P' = P \cup \{S' \to \beta \mid S \to \beta \in P \text{ with } \beta \neq \varepsilon\}$$

(c) *If the rule $S \to \varepsilon$ exists in the grammar*, make a copy of all $A \to \beta$ where β contains S, replacing all S by ε:

$$P'' = P' \cup \left\{ A \to \beta_1 S'\beta_2 S'\cdots S'\beta_n \;\middle|\; \begin{array}{c} A \to \beta_1\beta_2\cdots\beta_n \in P' \\ \text{with} \\ \beta_i \in (T \cup V \setminus \{S\})^* \end{array} \right\}$$

(d) Finally, replace all occurrences of S by S' in right-hand sides of rules:

$$P''' = \left\{ A \to \beta_1 S'\beta_2 S'\cdots S'\beta_n \;\middle|\; \begin{array}{c} A \to \beta_1 S\beta_2 S\cdots S\beta_n \in P'' \\ \text{with} \\ \beta_i \in (T \cup V \setminus \{S\})^* \end{array} \right\}$$

Figure 3.6 illustrates the construction. It is easy to check that the resulting grammar has the same language as the original one, and that it now respects the syntax of context-sensitive grammars.

This shows that $\mathsf{CFL} \subseteq \mathscr{L}^1$. Now, to prove that the inclusion is *strict*, we need to exhibit a language which is context-sensitive but not context-free. It is the case of the language $L(G_{ABC})$ generated by the grammar given in Example 3.3. Clearly, this language is context-sensitive since it is generated by a context-sensitive grammar. We will not prove here that this language is *not context free*. Suffice it to say that this can be proved by techniques similar to those we have used to show that $L_{()}$ is not regular (proof of Theorem 3.1). The interested reader should refer to the so-called '*pumping lemmata*' for regular and context-free languages, which are general techniques allowing one to prove that a language is not regular and not context-free respectively[8].

6. $\mathscr{L}^0 = \mathsf{RE}$. This equality holds by Definition 3.12.

7. $\mathsf{CSL} \subsetneq \mathscr{L}^0$. The fact that $\mathsf{CSL} \subseteq \mathscr{L}^0$ holds by definition: all grammars belong to class 0, so all CSL, that can be defined by a context-sensitive grammar (by definition) can be defined by a class 0 grammar. Showing that the inclusion is strict requires techniques that are beyond the scope of this course, so we will not prove it here.

□

Original grammar:

(1)	S	$\rightarrow$	ε
(2)		$\rightarrow$	aS
(3)		$\rightarrow$	A
(4)	A	$\rightarrow$	aS

After step (b):

(1)	S	$\rightarrow$	ε
(2)		$\rightarrow$	aS
(3)		$\rightarrow$	A
(4)	S'	$\rightarrow$	aS
(5)		$\rightarrow$	A
(6)	A	$\rightarrow$	aS

After step (c):

(1)	S	$\rightarrow$	ε
(2)		$\rightarrow$	aS
(3)		$\rightarrow$	a
(4)		$\rightarrow$	A
(5)	S'	$\rightarrow$	aS
(6)		$\rightarrow$	a
(7)		$\rightarrow$	A
(8)	A	$\rightarrow$	aS
(9)		$\rightarrow$	a

Final grammar:

(1)	S	$\rightarrow$	ε
(2)		$\rightarrow$	aS'
(3)		$\rightarrow$	a
(4)		$\rightarrow$	A
(5)	S'	$\rightarrow$	aS'
(6)		$\rightarrow$	a
(7)		$\rightarrow$	A
(8)	A	$\rightarrow$	aS'
(9)		$\rightarrow$	a

Figure 3.6: Removing all occurrences of the start symbol S from right-hand sides of rules, while preserving the language of the original grammar.

[8] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321455363

It is not easy to exhibit a 'natural' language that strictly separates CSL and RE. The arguments can be found in the Computablity and Complexity course: the class CSL corresponds exactly to the class of languages that can be decided by a non-deterministic Turing machine running in linear space. The *space hierarchy theorem* tells us that there are languages (which are decidable, and thus recognisable or, in other words, recursively enumerable) that require strictly more than linear space. For instance, the language of any *decider* that runs in exponential space is a recursively enumerable language that is not context-sensitive.

3.4 Exercises

3.4.1 Context-free languages

Exercise 3.1. Is the language $L = \{1^n \mid \exists m \in \mathbb{N} : n = m^2\}$ regular? Prove your answer using the techniques we have used at the beginning of the chapter.

3.4.2 Grammars

Exercise 3.2. Informally describe the languages generated by the following grammars and specify the classes of the Chomsky hierarchy they belong to:

(1)	S	$\rightarrow$	$\mathtt{0}$
(2)		$\rightarrow$	$\mathtt{1}$
(3)		$\rightarrow$	$\mathtt{1}S$

(1)	S	$\rightarrow$	$\mathtt{0}$
(2)		$\rightarrow$	$*SS$
(3)		$\rightarrow$	$+SS$

(1)	S	$\rightarrow$	$\mathtt{abc}A$
(2)		$\rightarrow$	$A\mathtt{abc}$
(3)	A	$\rightarrow$	ε
(4)	$A\mathtt{a}$	$\rightarrow$	$S\mathtt{a}$
(5)	$\mathtt{c}A$	$\rightarrow$	$\mathtt{c}S$

Then, give a derivation of the word $\mathtt{1110}$ according to the first grammar; a derivation of the word $* + \mathtt{a} + \mathtt{aa} * \mathtt{aa}$ according to the second grammar G_2 and a derivation of the word $abcabc$ produced by grammar G_3.

Exercise 3.3. Consider the following grammar:

(1)	S	$\rightarrow$	AB
(2)	A	$\rightarrow$	$A\mathtt{a}$
(3)	A	$\rightarrow$	$\mathtt{b}B$
(4)	B	$\rightarrow$	$\mathtt{a}$
(5)	A	$\rightarrow$	$S\mathtt{b}$

1. To which classes of the Chomsky hierarchy does this grammar belong?
2. Give the derivation trees of the three following sentential forms:
   (a) $\mathtt{ba}S\mathtt{b}$;
   (b) $\mathtt{b}BAB\mathtt{b}$;
   (c) $\mathtt{baabaab}$.
3. Give the *leftmost* and *rightmost* derivations for $\mathtt{baabaab}$.

Exercise 3.4. Give a *context-free* grammar that generates all strings of $\mathtt{a}$ and $\mathtt{b}$ (in any order) such that there are strictly more $\mathtt{a}$ than $\mathtt{b}$. Test your grammar on the input $\mathtt{baaba}$ by giving a derivation on this word.

Exercise 3.5. Give a *context-free* grammar that generates the language:

$$\{\mathtt{a}^n\mathtt{b}^m\mathtt{c}^\ell \mid n + m = \ell\}$$

Exercise 3.6. Give a context-sensitive grammar for the language

$$\{\mathtt{a}^m\mathtt{b}^n\mathtt{c}^m\mathtt{d}^n \mid m \geq 1, n \geq 1\}.$$

Do you think such a language can be generated by a context-free grammar? Explain why.

Exercise 3.7. Give a context-free grammar that generates all the arithmetic expressions on the alphabet $\{(,),+,.,0,1\}$ that evaluate to 2. Problem taken from Niwiński and Rytter[9].

Hint: start by generating all expressions that evaluate to 0, then to 1, then to 2.

[9] Damian Niwińsky and Wojciech Rytter. *200 Problems in Formal Languages and Automata Theory*. University of Warsaw, 2017

4 All things context free…

CONTEXT-FREE LANGUAGES ARE THE SECOND IMPORTANT CLASS OF LANGUAGES THAT WE WILL CONSIDER IN THIS COURSE, after regular languages. This chapter will be, in some sense, the 'context-free' analogous to Chapter 2, where we introduced and studied regular languages.

Let us first summarise quickly what we have learned so far about CFLs and their relationship to regular languages. First, recall, from the Chomsky hierarchy (Definition 3.7) that regular languages are all CFLs and that the containment is strict: for instance the Dyck language[1] $L_{()}$ is a CFL which is not regular (see Theorem 3.1). Moreover, we already know several formal tools to deal with regular languages and CFLs, as summarised in the table below:

[1] Recall that this languages contains all well-parenthesised words on $\{(,)\}$, i.e., a parenthesis is closed only if it has been opened before, and, at the end of the word, all opened parenthesis are eventually closed.

	Reg	CFL
Specification	Regular expressions, regular grammars	Context-free grammars
Automaton	DFA, NFA, ε-NFA	??

As can be seen, one cell is still empty in this table: which automaton model allows us to characterise the class of CFLs, just as finite automata correspond to regular languages? We will answer this question in Section 4.2, but let us already try and build some intuitions.

As explained in Chapter 2, finite automata (whether they are deterministic or not) can be regarded as a model of programs that have access to a *finite amount of memory*. This allows them to recognise simple languages such as $(01)^*$, for instance, because the only piece of information the program must 'remember' (in this example) is the last bit that has been read, in order to check that the next one is indeed different. So, one bit of memory is sufficient for $(01)^*$, which explains why the automaton accepting it has two states (see Figure 4.1).

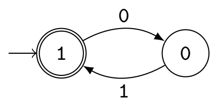


Figure 4.1: A finite automaton accepting $(01)^*$. The labels of the nodes represent the automaton's memory: it remembers the last bit read if any.

Now, let us consider a typical CFL, which is the language of all *palindromes* on $\{0,1\}$ (where the two parts of the palindromes are separated by $\#$). Formally, we consider the language:

$$L_{pal\#} = \left\{ w \cdot \# \cdot w^R \mid w \in \{0,1\}^* \right\}$$

We will prove later that $L_{pal\#}$ is indeed a CFL (and is not regular). Let us admit this fact for now, and let us understand why this language cannot be recognised by a finite automaton. Continuing the intuition we have sketched above, a program recognising $L_{pal\#}$ must, when reading a word of the form $w\#w'$:

1. store the whole prefix w up to the occurrence of #;

2. skip the symbol #;

3. read the suffix w', letter by letter, checking that w' is indeed w^R, the mirror image of w.

It should be clear that, since the length of the prefix w is *not bounded* (in the definition of $L_{pal\#}$), such a program needs an *unbounded* amount of memory, to store this prefix w. Since we are considering automata that read the input word from the left to the right, in one pass, and cannot modify the input, one can hardly imagine that a program (or an automaton) could recognise $L_{pal\#}$ using only a finite amount of memory.

So, to recognise CFLs, we need to extend finite automata with some form of unbounded memory. In the case of $L_{pal\#}$, this memory can be restricted to be a *stack*. Indeed, our program can be rewritten as:

Recall that a stack is a data structure where elements are stored as a sequence, where only the last inserted symbol can be accessed (it is called the *top* of the stack), and that can be modified only by appending symbols to the end of the sequence (a *push* to the stack); or by deleting the last symbol if it exists (a *pop* from the stack). Therefore, a stack is often referred to as a LIFO (an acronym of 'Last In First Out'), because the first elements that will be read from the stack is the last one that has been written.

1. read the prefix w up to the occurrence of #, letter by letter, *pushing* each letter on the stack;

2. skip the symbol #;

3. read the suffix w', letter by letter. Compare each letter from the input to the top of the stack. If they differ, or if the stack is empty, reject the word, otherwise *pop* the letter.

4. If the whole suffix has been read and the stack is empty, accept the word, otherwise reject.

As an example, Figure 4.2 illustrates the execution of this program on the word `001#100`, which is recognised as a palindrome since the stack is empty after the suffix is read. Each arrow between the stacks is labelled by a letter which is read by the program.

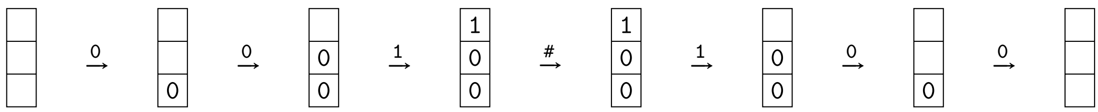


Figure 4.2: Recognising a palindrome using a stack.

So, in Section 4.2, we will extend finite automata by means of a stack, allowing the automaton to perform one operation on the stack at each transition (in addition to reading an input letter). We will formally study this new model, called *pushdown automata*[2] (PDA for short), and show that they recognise exactly the class of CFLs, just as finite automata recognise regular languages.

[2] *Pushdown* is a synonymous of stack.

In order to prove this last result, we will present a formal connection between PDAs and CFLs. Again, let us sketch some intuitions, by considering the grammar:

(1)	S	$\to$	$0S0$
(2)		$\to$	$1S1$
(3)		$\to$	#

that generates exactly $L_{pal\#}$. We can check against Definition 3.7 that this grammar is indeed context-free. It is easy to turn such a grammar into a recursive program that recognises $L_{pal\#}$, by regarding each variable in the right-hand side of the rule as a recursive call. In a python-like syntax, such a program could be:

```
def S():
  n = read_next_character()

  if n == '#':
    return True

  if n == '0':
    r = S()
    if (!r): return False
    n = read_next_character()
    if (n == '0'): return True
    else: return False

  if n == '1':
    r = S()
    if (!r): return False
    n = read_next_character()
    if (n == '1'): return True
    else: return False
```

where, as expected `read_next_character()` reads the next character on the input and returns it. This code matches the semantics of the grammar because, roughly speaking, a rule such as:

$$S \rightarrow 0S0$$

can be interpreted as:

> *Read a* `0`*; then check that it is followed by a palindrome, i.e. a word that can be generated by S; and finally read a matching* `0`.

Observe that this intuition holds because the grammar *is context-free*, i.e., all the rules are of the form $A \rightarrow \alpha$, where A is a single variable, that can be assimilated to a function name. The mapping of grammar rule to recursive functions is harder to figure out (if any) when the rules are allowed to be context sensitive, such as $aSb \rightarrow cAd$ for instance. Observe also that the recursive functions obtained from CFGs by this construction do not need unbounded memory: they only need to store a finite, bounded portion of the input word (which is a finite word on a finite alphabet). In the case of $L_{pal\#}$, each function call needs to store *locally* the *first character read* (either `0` or `1`) to compare it to the first character read after the recursive call, so one bit of *local* memory is sufficient. The unboundedness of the memory needed to recognise words from $L_{pal\#}$ stems from the (unbounded) depth of the recursive calls.

Now we can bridge the gap between context free grammars and pushdown automata easily: a context-free grammar is, roughly speaking, a recursive program where each function call needs only bounded memory. Thus, the behaviour of each function call can be captured by a finite automaton, but the number of recursive calls cannot be bounded. This can be captured by a PDA, as sketched in Figure 4.3. Each time the function performs a recursive call, the PDA pushes information about its current state (that encodes the value of the local variables) to the stack and moves to its initial state. At each return, the PDA pops the top of the stack, to recover the value of the local variables and uses it to move to the state that models the state of the function after the recursive call.

There are many different syntax's for PDAs. The one we are using in this example aims at illustrating easily the intuitions of this introduction. Note that the formal syntax we will use in the rest of the chapter will differ slightly.

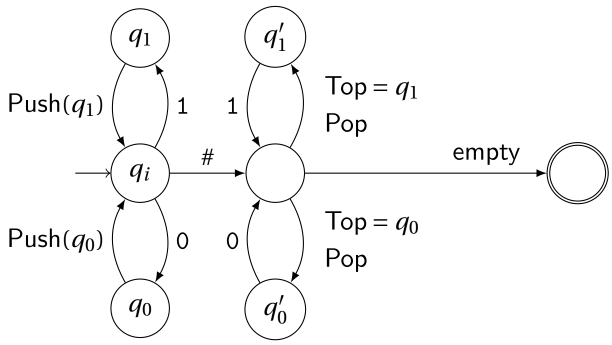


Figure 4.3: An intuition of a pushdown automaton that recognises $L_{pal\#}$. The keywords Push, Pop and Top have their usual meaning. The edge labelled by empty can be taken only when the stack is empty. Note that, instead of pushing q_0 or q_1, one could simply store 0 and 1's on the stack.

Transitions between q_i, q_0 and q_1 simulate the recursive calls by pushing information on the stack. Once a # is read, the automaton starts popping the content of the stack to simulate the returns of the recursive calls. The accepting state is reached only when the stack is empty, i.e., all pending recursive calls have completed.

This short example shows that, somehow, CFGs can be translated into PDAs that accept the same language. Using the same ideas, the reverse translation (from PDAs to CFGs) can be achieved. This is the outline of the proof we will present in Section 4.2.3. The close connection between CFGs, PDAs and recursive functions we have just sketched will also be central in Chapter 5, where we will discuss the automated construction of *parsers* from CFGs, in a compiler design perspective. Before that, we first take a fresh look at grammars, by specialising the theory we have developed in Chapter 3 to the particular case of CFGs.

4.1 *Context-free grammars*

Let us start by discussing some formal tools that are useful when dealing with *context-free* grammars (hence, also regular grammars). Some of them (such as the notion of derivation) have already been introduced in Chapter 3, but will be specialised for CFGs.

4.1.1 *Derivation*

(1)	Exp	→	Exp + Exp
(2)		→	Exp ∗ Exp
(3)		→	(Exp)
(4)		→	Id
(5)		→	Cst

Figure 4.4: The grammar G_{Exp} to generate expressions.

We have already discussed the notion of derivation in the previous chapter, see Definition 3.4 *sqq.* and the intuition given before. Let us consider again the grammar G_{Exp}, given in Figure 4.4, and let us consider the word $\mathsf{Id}+\mathsf{Id}*\mathsf{Id}$ which can be generated by this grammar. Indeed, the following is a possible derivation of G_{Exp} for this word (where we have underlined the variable which is rewritten at each rule application):

$$\underline{\mathsf{Exp}} \overset{2}{\Rightarrow} \underline{\mathsf{Exp}}*\mathsf{Exp} \overset{1}{\Rightarrow} \mathsf{Exp}+\underline{\mathsf{Exp}}*\mathsf{Exp} \overset{4}{\Rightarrow} \mathsf{Exp}+\mathsf{Id}*\underline{\mathsf{Exp}} \overset{4}{\Rightarrow} \underline{\mathsf{Exp}}+\mathsf{Id}*\mathsf{Id} \overset{4}{\Rightarrow} \mathsf{Id}+\mathsf{Id}*\mathsf{Id} \tag{4.1}$$

We can already observe that, since G_{Exp} is *context-free*, all the derivations it generates have a particular shape: they consist, at each step, in *replacing one variable by a word over* $(\Sigma \cup V)^*$. This peculiarity of CFGs allows us to define new notions: the *leftmost* and *rightmost derivations*, the *derivation trees* and the notion of *ambiguity*.

Leftmost and rightmost derivations Although the sequence (4.1) of derivations above is sufficient to prove that $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$ is accepted by G_{Exp}, other sequences could be exhibited. Indeed, recall that there is some amount of *non-determinism* in the definition of the language of a grammar (Definition 3.6): there should exist *at least one* derivation producing the word to be accepted. Other sequences of derivations producing $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$ are:

$$\underline{\mathsf{Exp}} \overset{2}{\Rightarrow} \mathsf{Exp} * \underline{\mathsf{Exp}} \overset{4}{\Rightarrow} \underline{\mathsf{Exp}} * \mathsf{Id} \overset{1}{\Rightarrow} \mathsf{Exp} + \underline{\mathsf{Exp}} * \mathsf{Id} \overset{4}{\Rightarrow} \underline{\mathsf{Exp}} + \mathsf{Id} * \mathsf{Id} \overset{4}{\Rightarrow} \mathsf{Id} + \mathsf{Id} * \mathsf{Id} \tag{4.2}$$

$$\underline{\mathsf{Exp}} \overset{2}{\Rightarrow} \underline{\mathsf{Exp}} * \mathsf{Exp} \overset{1}{\Rightarrow} \underline{\mathsf{Exp}} + \mathsf{Exp} * \mathsf{Exp} \overset{4}{\Rightarrow} \mathsf{Id} + \underline{\mathsf{Exp}} * \mathsf{Exp} \overset{4}{\Rightarrow} \mathsf{Id} + \mathsf{Id} * \underline{\mathsf{Exp}} \overset{4}{\Rightarrow} \mathsf{Id} + \mathsf{Id} * \mathsf{Id} \tag{4.3}$$

Such sequences are called *rightmost* and *leftmost* respectively, because we have obtained them by always deriving the rightmost and leftmost variable in the all sentential forms. On the other hand, (4.1) is neither leftmost nor rightmost: at the second step, we have derived the leftmost variable; at the third step, we have derived a variable Exp which was neither leftmost nor rightmost; and at the fourth step we have derived the rightmost variable.

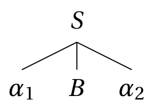
Here is the formal definition capturing these intuitions:

Definition 4.1 (Left- and right-most derivation)**.** Let $G = \langle V, T, P, S\rangle$ be a context-free grammar. A derivation $w\,S\,w' \Rightarrow w\,\alpha\,w'$ of G which is obtained by applying $S \to \alpha$ is *leftmost* iff: $w \in T^*$. It is *rightmost* iff: $w' \in T^*$.

Leftmost and rightmost derivations are important because they are the ones which will be generated by the parsers we will define in Chapter 5 and Chapter 6. Selecting the left- or right-most derivation allows somehow to get rid of a part of the grammar's non-determinism, which is exactly what we need when we build a program (which must be deterministic).

That is, in a leftmost derivation, one can replace variable S by α in $w\,S\,w'$, yielding derivation $w\,S\,w' \Rightarrow w\,\alpha\,w'$ if and only if w contains only terminals (i.e., no variable, otherwise, the derivation would not be leftmost). Symmetrically for a rightmost derivation, where w' must contain only terminals.

Derivation tree Another way to prove that a given word belongs to the language of a grammar is to exhibit a *derivation tree* for this word. The idea behind the derivation tree is similar to the intuition we have given in the introduction that a rule like $S \to \alpha_1\,B\,\alpha_2$ can be interpreted as 'match α_1; then a string that can be generated by B; then α_2', which suggests a recursive definition of the acceptance of a word. Such a recursive view can easily be expressed by means of a tree:



This tree can then be completed up to the point where the leaves contain only terminals.

Example 4.2. Let us illustrate this idea by a more concrete example. Consider again the grammar G_{Exp} in Figure 4.4, and the word $\mathsf{Id}+\mathsf{Id}*\mathsf{Id}$. Then, a derivation tree of this word is given in Figure 4.5.

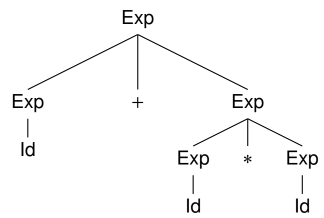


Figure 4.5: A derivation tree for the word $\mathsf{Id}+\mathsf{Id}*\mathsf{Id}$.

From this example, it is easy to determine the characteristics of a derivation tree for a word w. Its root must be labelled by the start symbol S of the grammar; the children of each node must correspond to the right-hand side of a rule whose left-hand side is the label of the node; and the sequence of leaves from left to right must be the word w. Here is a more formal definition:

Definition 4.3 (x-tree)**.** Let $G = \langle V, T, P, S\rangle$ be a CFG, and let $x \in V \cup T$ be either a variable or a terminal of G. Then, an *ordered*[3], *labelled* tree $\mathcal{T}$ is an x-tree iff:

1. either $x \in T$ and $\mathcal{T}$ is a leaf labelled by x; or
2. $x \in V$ is the label of $\mathcal{T}$'s root; and there is a rule $x \to \alpha_1\alpha_2\cdots\alpha_k$ in P s.t. the sub-trees of $\mathcal{T}$ are $\mathcal{T}_1$, $\mathcal{T}_2$,..., $\mathcal{T}_k$ where $\mathcal{T}_i$ is an α_i-tree for all $1 \le i \le k$.

[3] An ordered tree is a tree in which we have fixed a total order on the children of all nodes. So, one can speak of the first, second,..., last child of a node. A particular case of ordered trees are the classical binary trees, where the first and second children are called left child and right children respectively.

Then, we can define what is a derivation tree for a given word w:

Definition 4.4 (Derivation tree)**.** Given a CFG $G = \langle V, T, P, S\rangle$, a tree $\mathcal{T}$ is a *derivation tree* of w iff $\mathcal{T}$ is an S-tree and w is obtained by traversing $\mathcal{T}$'s leaves from the left to the right.

As we will see in the next chapter, derivation trees are a most important tool in compiler construction. Very often, the output of the parser[4] will be a derivation tree (possibly with some extra information as we will see later). Indeed, the structure of the derivation tree provides us with more information on the structure of the input word than a derivation does. For example, the derivation tree in Figure 4.5 reveals a possible structure for the expression $\mathsf{Id}+\mathsf{Id}*\mathsf{Id}$, and suggests that it should be understood as the sum of Id and $\mathsf{Id}*\mathsf{Id}$. In other word, the structure of the tree suggests that the semantics of the expression corresponds to that of $\mathsf{Id}+(\mathsf{Id}*\mathsf{Id})$ which indeed matches the priority of the arithmetic operators. Such information will clearly be important for the *synthesis phase* of compiling, where the executable code corresponding to the expression will be created.

[4] See Section 1.3.1 for the different phases of the compiling process and their relative connections.

Ambiguities We have seen before that a given word might be derived using several different derivations (which has prompted us to introduce the notions of leftmost and rightmost derivations). One natural question is thus whether there can be different *derivation trees* for the same word? The answer is yes, as can be seen in Figure 4.6.

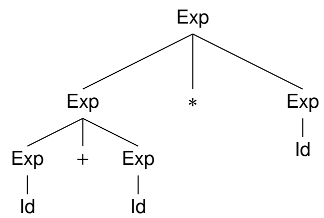


Figure 4.6: Another derivation tree for the word $\mathsf{Id}+\mathsf{Id}*\mathsf{Id}$.

Contrary to the tree in Figure 4.5, this tree suggests that the expression $\mathsf{Id}+\mathsf{Id}*\mathsf{Id}$ should rather be understood as $(\mathsf{Id}+\mathsf{Id})*\mathsf{Id}$, instead of $\mathsf{Id}+(\mathsf{Id}*\mathsf{Id})$. This is rather unfortunate: were such a tree returned by the parser, the code generated by the synthesis phase would not correspond to the actual priority of the operators. The question is then: 'how can we decide, based only on the grammar G_{Exp} (Figure 4.4) which derivation tree should be generated for $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$?' Unfortunately, the answer is 'we can't!', because the grammar does not contain any information about the priority of the operators. In other words, the grammar is intrinsically *ambiguous*:

Definition 4.5 (Ambiguous grammar)**.** A CFG is *ambiguous* iff it generates at least one word which admits two different derivation trees.

Example 4.6. As witnessed by Figure 4.5 and Figure 4.6, grammar G_{Exp} (Figure 4.4) is ambiguous.

For the reason explained above, ambiguous grammars will be a big issue when generating parsers. We will see, in Section 4.4, techniques to turn an ambiguous grammar into a non-ambiguous one, that accepts the same language, by taking into consideration extra information such as the priority and associativity of the operators.

Relationship between derivations and derivation trees So far, we have seen two mathematical tools that allow to show that a given word w belongs to the language of a grammar G: either by exhibiting a derivation of G that generates w, or by giving a derivation tree of w for G. A natural question is thus to understand the relationship between derivations and derivation trees. Roughly speaking, this relationship is a one-to-many one: to each derivation tree of w correspond potentially several derivations producing w, while each derivation of w corresponds to one and only one derivation tree.

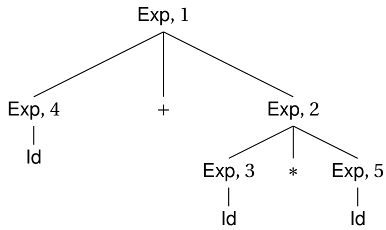


Figure 4.7: The derivation tree for $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$ of Figure 4.5 with a top-down traversal indicated by the position of each node in the sequence

To make this intuition more formal, consider again the derivation tree in Figure 4.5. We call a *top-down traversal* of a tree any sequence of the tree's internal nodes s.t. each time a node occurs in the sequence, all its ancestors have occurred before. As an example, consider the tree in Figure 4.7, which is the same derivation tree as in Figure 4.5, with all internal nodes labelled by their index in a top-down traversal: first the root, then the right son of the root, then the left son of this last node, etc.

It is easy to check that this top-down traversal corresponds to the following derivation:

$$\begin{aligned}&\mathsf{Exp} \Rightarrow \mathsf{Exp} + \mathsf{Exp} \Rightarrow \mathsf{Exp} + \mathsf{Exp} * \mathsf{Exp}\\ &\Rightarrow \mathsf{Exp} + \mathsf{Id} * \mathsf{Exp} \Rightarrow \mathsf{Id} + \mathsf{Id} * \mathsf{Exp} \Rightarrow \mathsf{Id} + \mathsf{Id} * \mathsf{Id}\end{aligned}$$

This derivation has been obtained by following the sequence of internal nodes corresponding to the top-down traversal, and applying, each time, the derivation which has been used in the tree to generate the sons of this node.

Then, it is clear that all derivations corresponding to a given derivation tree are those that can be obtained by a top-down traversal of this tree. In particular, the leftmost-derivation is obtained by the classical infix traversal, and the rightmost-derivation is obtained by the infix traversal where the left and right sons have been swapped (the right son is visited before the left one).

4.1.2 The membership problem

We close this section on CFGs by discussing the *membership problem*, when specialised to such grammars. It is defined as follows:

Problem 4.7. Given a CFG G on the alphabet Σ and a word $w \in \Sigma^*$, determine whether $w \in L(G)$.

Clearly, this problem is central to the construction of compilers: if, as explained in the Introduction, we specify the syntax of a programming language by means of a CFG, then, checking whether the syntax of a given program is correct boils down to check whether this program (actually, the sequence of tokens that compose it) belongs to the language of the grammar.

We will now show that checking whether a word w is in $L(G)$ can always be done in time $\mathcal{O}(n^3)$ and memory $\mathcal{O}(n^2)$ where $n = |w|$, and when G is a CFG. To achieve this result, we first need a very convenient transformation of CFGs, which is called the *Chomsky normal form*.

This complexity sounds like good news, as it suggests that *parsing* can actually be carried out efficiently. After all, we have all learned that polynomial-time algorithms are *efficient*, haven't we? However, assume that we want to check the syntax of a program with $10{,}000 = 10^4$ tokens (probably a few thousands of lines of code). Then, the syntax check would need 10^{12} steps. Assume each of these steps can be carried out in 10^{-6} sec, i.e. 1μsec. Then, checking the syntax of this input would already take more than 11 days! Whereas a quadratic algorithm would run in less than two minutes, and a linear time algorithm would take a few milliseconds. As a matter of fact, we will see how to craft efficient (linear-time) parsers in Chapter 5 and Chapter 6, for certain kinds of CFGs.

The Chomsky normal form The *Chomsky normal form* is a simple syntactic restriction that can be applied to all CFGs, in the sense that all CFGs can be converted into an equivalent CFG (one that accepts the same language) respecting the normal form. As expected, this special form was introduced[5] by... Noam CHOMSKY (in 1959). Here is the definition:

Definition 4.8 (Chomsky normal form for CFG)**.** A CFG $G = \langle V, T, P, S\rangle$ is in *Chomsky normal form* (CNF for short) iff every rule is of one of the following forms

$$\begin{aligned} A &\to BC \\ A &\to a \\ S &\to \varepsilon \end{aligned}$$

where:

- A is any variable (including S): $A \in V$;
- B and C are any variable different from the start symbol: $\{B, C\} \subseteq V \setminus \{S\}$; and
- a is any terminal (i.e., different from ε): $a \in T$.

Roughly speaking, all the rules in the grammar either replace one variable A by a sequence of exactly two variables BC (that are different from the start symbol); or replace a variable A by a single non-empty terminal a. The only exception to this rule is that the start symbol S can generate ε: this is necessary to ensure that the empty word can be accepted by a grammar in CNF (and this also explains why we do not allow S to occur in any right-hand part).

As announced above, we can build, from all CFGs an equivalent one which is in CNF:

[5] N. Chomsky. On certain formal properties of grammars. *Information and Computation (formerly known as Information and Control)*, 2(2):137 – 167, 1959. DOI: 10.1016/S0019-9958(59)90362-6

Theorem 4.1. *For all* CFG*s* G*, we can build, in polynomial time, a* CFG G' *in* CNF *s.t.* $L(G) = L(G')$.

We will not prove this theorem. A formal proof can be found in CHOMSKY's original paper[6], but we recommend the construction that can be found in M. SIPSER's book[7] on complexity. Let us highlight the main point of the transformation through an example.

[6] N. Chomsky. On certain formal properties of grammars. *Information and Computation (formerly known as Information and Control)*, 2(2):137 – 167, 1959. DOI: 10.1016/S0019-9958(59)90362-6

[7] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996. ISBN 053494728X

Example 4.9. Consider the grammar given in Figure 4.8. It is clearly not in CNF. To turn it into an equivalent CFG in CNF, we first take care of the so-called *unit rules*, i.e. rules like $A \to B$ where the right-hand side contains just one variable. We can remove rule $A \to B$ provided we add the rule:

$$S \to \mathtt{a}BB,$$

since $S \to \mathtt{a}AB$ is the only rule where A occurs in the right-hand side. Then, we get rid of $A \to \varepsilon$ by deleting A from all right-hand side where it still occurs (since ε is now the only way to derive A). Thus, our grammar has now become:

(1)	S	$\to$	$\mathtt{a}B$
(2)	S	$\to$	$\mathtt{a}BB$
(3)	B	$\to$	$\mathtt{a}B\mathtt{c}$
(4)	B	$\to$	$\mathtt{d}$

(1)	S	$\to$	$\mathtt{a}AB$
(2)	A	$\to$	B
(3)	A	$\to$	ε
(4)	B	$\to$	$\mathtt{a}B\mathtt{c}$
(5)	B	$\to$	$\mathtt{d}$

Figure 4.8: A CFG which is not in CNF.

This is not yet satisfactory! As a matter of fact, only the last rule complies with the definition of CNF. We can transform the other rules by introducing a limited amount of variables. Then, $S \to \mathtt{a}B$ becomes:

$$S \to V_1 B$$
$$V_1 \to \mathtt{a}$$

where V_1 is a fresh variable. Similarly, $S \to \mathtt{a}BB$ becomes:

$$S \to V_1 V_2$$
$$V_2 \to BB$$

and $B \to \mathtt{a}B\mathtt{c}$ becomes:

$$B \to V_1 V_3$$
$$V_3 \to BV_4$$
$$V_4 \to \mathtt{c}.$$

The final grammar, which is now in CNF is given in Figure 4.9.

(1)	S	$\to$	$V_1 B$
(2)	S	$\to$	$V_1 V_2$
(3)	V_2	$\to$	BB
(4)	B	$\to$	$V_1 V_3$
(5)	V_3	$\to$	BV_4
(6)	V_1	$\to$	$\mathtt{a}$
(7)	V_4	$\to$	$\mathtt{c}$
(8)	B	$\to$	$\mathtt{d}$

Figure 4.9: A CFG in CNF that corresponds to the CFG in Figure 4.8.

Checking language membership of CFG*s* Equipped with the notion of CNF, we can now describe a polynomial-time algorithm to check whether a given word w belongs to the language of a given CFG $G = \langle V, T, P, S \rangle$, *that we assume to be in* CNF (recall that, if the given CFG is not in CNF, we can obtain an equivalent CFG which is in CNF, in polynomial time—see Theorem 4.1).

Let us first sketch some intuitions. We observe that, thanks to the special syntax of production rules in CNF grammars, checking membership is particularly easy for words of length 0 or 1. Indeed:

- The only word of length 0 is $w = \varepsilon$. Since $S \to \varepsilon$ is the only rule that can appear with an ε on the right-hand side, we conclude that $\varepsilon \in L(G)$ iff $S \to \varepsilon$ appears in P.
- Similarly, a word w of length 1 is a single terminal $\mathtt{a}$. Again, $w = \mathtt{a}$ is accepted iff the grammar has a rule $S \to \mathtt{a}$. Indeed, if $S \to \mathtt{a}$ is a rule of G, it is clear that G accepts $\mathtt{a} = w$. On the other hand, if $S \to \mathtt{a}$ is not a rule of G, then there is no way G can accept $\mathtt{a} = w$, since all other rules with S as the left-hand side are either of the form $S \to \mathtt{b}$, with $\mathtt{b} \neq \mathtt{a}$; or of the form $S \to AB$ and will thus generate words of length at most 2.

The second item of the above reasoning can be generalised: we can check whether any variable $A \in V$ can generate a word $w = \mathtt{a}$ of length 1 simply by checking whether the grammar has a rule $A \to \mathtt{a}$ or not.

Now, let us turn our attention to the following more general problem: checking whether some given variable A can generate a given word $w = w_1 w_2 \cdots w_n$ for $n \geq 2$? Clearly, if we can answer that question, then we will be able to answer the membership problem, simply by letting $A = S$. We will base our reasoning on an intuition that we have already given: that a rule $A \to \alpha$ of a CFG can be regarded as a recursive function A which performs a series of calls as given by α. In the setting of CNF grammars, all 'recursive' rules are of the form $A \to BC$, hence they correspond to two successive 'recursive' calls. In other words, a variable A can generate w iff we can find a rule $A \to BC$ and we can split w into two non-empty subwords u and v (i.e., $w = uv$) s.t.: (i) B generates u; and (ii) C generates v. Observe that this recursive definition is sound since u and v are both non-empty, and thus have necessarily a length which is strictly smaller than n. So eventually, this definition will amount to check whether all characters of the word w can be generated by some variables, which can be done easily as we have observed above.

Clearly, this discussion suggests a recursive procedure for checking membership of a word w to $L(G)$. However, such a procedure could run in exponential time. In order to avoid this, we will rely on the very general idea of *dynamic programming*. In our case, dynamic programming consists in storing in a (quadratic) table the result of the procedure when called on all the possible sub-strings of w. By filling this table following a smart order, we will manage to keep the computing time polynomial. Basically, we will first fill in the table for the shortest substrings of w (i.e., the individual letters), then use this information to deduce whether we can accept longer and longer substrings… up to the whole word w itself.

Dynamic programming is a general algorithmic technique, defined by Wikipedia as '*a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions - ideally, using a memory-based data structure*'. It has been introduced by Richard BELLMAN in the forties. This technique occurs in many classical algorithms such as the BELLMAN-FORD and FLOYD-WARSHALL algorithms to compute shortest paths in a graph.

More precisely, we will build a table Tab of dimension $n \times n$ s.t. *each cell* $\mathsf{Tab}(i, j)$ *will contain the list of all variables that can generate the subword* $w_i \cdots w_j$. Formally, $A \in \mathsf{Tab}(i, j)$ iff $A \Rightarrow^* w_i \cdots w_j$. When this table is complete, checking whether $w \in L(G)$ amounts to checking whether the start symbol can generate $w_1 \cdots w_n$, i.e. whether $S \in \mathsf{Tab}(1, n)$.

As explained above, we start by filling the cells that correspond to the individual letters making up $w = w_1 w_2 \cdots w_n$. For all $1 \leq i \leq n$, we put variable A in $\mathsf{Tab}(i, i)$ iff the rule $A \to w_i$ occurs in the grammar.

Then, we fill the cells corresponding to subwords of length ℓ for increasing values of $\ell = 2, 3, \ldots$. Assuming all the cells for subwords of length $< \ell$

Input: A CFG $G = \langle V, T, P, S\rangle$ (in CNF), a word
$w = w_1 w_2 \cdots w_n \in T^*$.
Output: *True* iff $w \in L(G)$.

```
if w = ε then
    if S → ε ∈ P then return True ;
    else return False ;
foreach 1 ≤ i ≤ n do
    Tab(i,i) ← {A | A → w_i ∈ P} ;
foreach 1 ≤ ℓ ≤ n do
    foreach 1 ≤ i ≤ n − ℓ + 1 do
        j ← i + ℓ − 1 ;
        foreach i ≤ k ≤ j − 1 do
            foreach rule A → BC ∈ P do
                if B ∈ Tab(i,k) and C ∈ Tab(k+1, j) then
                    Add A to Tab(i, j) ;
if S ∈ Tab(1,n) then return True ;
else return False ;
```

Algorithm 3: An $\mathcal{O}(n^3)$ algorithm to check whether $w \in L(G)$ for a CFG grammar in CNF.

have been filled, we fill the cells corresponding to some subword $w_i \cdots w_j$ of length ℓ as follows. We put variable A in $\mathsf{Tab}(i, j)$ iff there is a rule $A \to BC$ in the grammar, and a split position $i \le k < j$ s.t. $B \in \mathsf{Tab}(i,k)$ and $C \in \mathsf{Tab}(k+1, j)$ (i.e., iff B can generate the suffix $w_i \dots w_k$ and C can generate the suffix $w_{k+1} \cdots w_j$, which we can test by querying the corresponding cells of the table). The algorithm that implements this procedure is given in Algorithm 3, following the presentation of Sipser[8]. The paternity of this algorithm has been attributed to several researchers by whom is has been independently re-discovered... It is often referred to as the Cocke-Younger-Kasami algorithm (CYK algorithm for short), although it seems to have been introduced first[9] by SAKAI in 1961.

[8] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996. ISBN 053494728X

[9] Itiiro Sakai. Syntax in universal translation. In *International Conference on Machine Translation of Languages and Applied Language Analysis*, pages 593–608. London: Her Majesty's Stationery Office, 1961

Example 4.10. Let us consider the grammar in Figure 4.10, which is in CNF. One can check that this grammar accepts the language $\mathtt{a}^+\mathtt{b}$. Observe in particular that the word $w = \mathtt{aaab}$ can be accepted by two different derivations:

$$S \Rightarrow XB \Rightarrow XAB \Rightarrow XAAB \Rightarrow \mathtt{a}AAB \Rightarrow \mathtt{aa}AB \Rightarrow \mathtt{aaa}B \Rightarrow \mathtt{aaab},$$

and

$$S \Rightarrow X'B \Rightarrow AYB \Rightarrow AAAB \Rightarrow \mathtt{a}AAB \Rightarrow \mathtt{aa}AB \Rightarrow \mathtt{aaa}B \Rightarrow \mathtt{aaab}.$$

So, in particular, the subword $w_1 w_2 w_3 = \mathtt{aaa}$ can be generated either by X or by X'.

(1)	S	$\to$	XB
(2)		$\to$	$X'B$
(3)	X	$\to$	XA
(4)		$\to$	$\mathtt{a}$
(5)	X'	$\to$	AY
(6)	Y	$\to$	AA
(7)	A	$\to$	$\mathtt{a}$
(8)	B	$\to$	$\mathtt{b}$

Figure 4.10: An example CNF grammar generating $\mathtt{a}^+\mathtt{b}$.

Let us now apply the above algorithm on word $w = \mathtt{aaab}$, and fill a 4×4 table Tab (since our word w is of length 4). We will actually only fill the cells $\mathsf{Tab}[i, j]$ s.t. $i \le j$ because there is no subword starting in i and ending in j with $i > j$. Observe that we are interested in checking whether $S \in$ `Tab`$[1,4]$, which is the top right cell of the table. We proceed by increasing length of subwords:

- For the subwords of length 1, we consider the subwords $w_1 = \mathtt{a}$, $w_2 = \mathtt{a}$, $w_3 = \mathtt{a}$ and $w_4 = \mathtt{b}$. Clearly, a can be generated by X and A only; and b can be generated by B only. Indeed, all other variables have derivations of the form $F \rightarrow GH$ (where F, G, H are variables), hence, they will all generate words of length at least 2, since no rule of the form $F \rightarrow \varepsilon$ is allowed for a variable $F \neq S$ (by the Chomsky normal form). So, we have the table:

1	2	3	4	
A,X				1
	A,X			2
		A,X		3
			B	4

- For the subwords of length 2, we consider the subwords $w_1 w_2 = \mathtt{aa}$, $w_2 w_3 = \mathtt{aa}$ and $w_3 w_4 = \mathtt{ab}$. The only way to split these subwords is 'in the middle', i.e. $w_1 w_2$ is split into w_1 and w_2 for example. So, we will fill in `Tab`$[1,2]$ using the information of `Tab`$[1,1] = A, X$ and `Tab`$[2,2] = A, X$. To exploit this information, we need to find a variable V_1 in `Tab`$[1,1]$, and a variable V_2 in `Tab`$[2,2]$ s.t. the grammar has a rule of the form $V \rightarrow V_1 V_2$. In this case, we can add V to `Tab`$[1,2]$. There are two such choices for V_1 and V_2. Either we let $V_1 = X$, $V_2 = A$ and consider the rule $X \rightarrow XA$; or we let $V_1 = A$, $V_2 = A$ and consider the rule $Y \rightarrow AA$. Observe that indeed, X and Y can both generate aa. Continuing so for the other cells corresponding to subwords of length 2, we have:

Recall the intuition about the table: V_1 can generate w_1 and V_2 can generate w_2.

1	2	3	4	
A, X	X, Y			1
	A, X	X, Y		2
		A, X	S	3
			B	4

- For the words of length 3, we consider $w_1 w_2 w_3 = \mathtt{aaa}$ and $w_2 w_3 w_4 = \mathtt{aab}$. We can recognise $w_1 w_2 w_3$ by splitting it into w_1 and $w_2 w_3$; or into $w_1 w_2$ and w_3. Let us first consider the case where we split into $w_1 = \mathtt{a}$ and $w_2 w_3 = \mathtt{aa}$. By `Tab`$[1,1]$, we know that w_1 can be generated either by A or by X. By `Tab`$[2,3]$, $w_2 w_3$ can be generated either by X or by Y. The only rule that allows us to fill `Tab`$[1,3]$ in this case is $X' \rightarrow AY$, so $w_1 w_2 w_3 = \mathtt{aaa}$ can be generated by X'. In the latter case (where we split into $w_1 w_2$ and w_3), we use rule $X \rightarrow XA$ and discover that X can generate $w_1 w_2 w_3 = \mathtt{aaa}$ as well. We proceed similarly for $w_2 w_3 w_4 = \mathtt{aab}$, and obtain:

1	2	3	4	
A, X	X, Y	X, X'		1
	A, X	X, Y	S	2
		A, X	S	3
			B	4

- Finally, for the single subword of length 4, which is w itself, we need to consider three possible splits: either into w_1 and $w_2w_3w_4$; or into w_1w_2 and w_3w_4; or into $w_1w_2w_3$ and w_4. Only the last split will yield a new piece of information into Tab[1,4], since $w_1w_2w_3$ can be generated either by X or by X' and w_4 can be generated by B. Then, both rules $S \rightarrow XB$ and $S \rightarrow X'B$ allow us to conclude that S can generate w:

1	2	3	4	
A,X	X,Y	X,X'	**S**	1
	A,X	X,Y	S	2
		A,X	S	3
			B	4

4.2 Pushdown automata

Let us now define formally the notion of *pushdown automaton* (PDA for short) that we have described informally in the introduction of this chapter. Remember that, in essence, a PDA is a finite state automaton augmented with a stack that serves as a memory: at each transition, the automaton can test the value on the top of the stack, and modify (push, pop) this top of stack.

4.2.1 Syntax and semantics

Syntax The formal definition of the syntax of PDA clearly shows that they are an extension of finite automata:

Definition 4.11 (Pushdown automaton)**.** A *Pushdown automaton* (PDA for short) is a tuple $\langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$ s.t.:

1. Q is a finite set of states;
2. Σ is a finite input alphabet;
3. Γ is a finite stack alphabet;
4. $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow 2^{(Q \times \Gamma^*)}$ is the transition function;
5. q_0 is the initial state;
6. $Z_0 \in \Gamma$ is the initial symbol on the stack;
7. $F \subseteq Q$ is the set of accepting states.

Clearly, the elements Q, Σ, δ and F were already present in the definition of finite automaton. Γ is the stack alphabet: it contains all the symbols that can be pushed on the stack (in practice, we can thus store on the stack symbols that are not taken from the input, i.e., not in Σ). Z_0 is a symbol that we assume will always be present on the stack initially. This will be important to have a clean definition of operations that test whether the stack is empty or not. Finally, observe that δ has a different signature:

it takes, as input, the current state and the next symbol on the input (as in a finite automaton), but also a stack symbol, which is meant to be the symbol on the top of the stack. It outputs a set of pairs of the form (q, w), where q is a destination state (as in a non-deterministic finite automaton) and w is a word from Γ^* that will replace the symbol on the top of the stack after the transition has been fired. So, intuitively:

$$\delta(q, a, b) = \{(q_1, \gamma_1), \dots, (q_n, \gamma_n)\}$$

means:

> When in state q, reading a from the input, and having b on the top of the stack, chose non-deterministically a pair (q_i, γ_i), move to q_i, and replace b on the top of the stack by γ_i (where the leftmost letter of w_i goes on the top).

Before making this definition of the transition function more formal, let us give an example of PDA following this syntax. Such an example can be found in Figure 4.11. In order to depict the translation relation, we draw an arrow between q and q', labelled by $a, b/w$ iff $(q', w) \in \delta(q, a, b)$, i.e. we can go from q to q' while reading an a on the input, seeing a b on the top of the stack, and replacing this b by w. In other words, in this example, we have:

This syntax might look hard to read and is indeed way less intuitive than the classical 'pop' and 'push'. It allows, however, a very clean definition of the semantics of PDAs, as we will see later.

1. $Q = \{q_0, q_1, q_2\}$;

2. $\Sigma = \{0, 1, \#\}$;

3. $\Gamma = \{0, 1, Z_0\}$;

4. $F = \{q_2\}$;

5. and the transition function is as follows:

$\delta(q_0, i, t) =$

i/t	0	1	Z_0
0	$\{(q_0, 00)\}$	$\{(q_0, 01)\}$	$\{(q_0, 0Z_0)\}$
1	$\{(q_0, 10)\}$	$\{(q_0, 11)\}$	$\{(q_0, 1Z_0)\}$
#	$\{(q_1, 0)\}$	$\{(q_1, 1)\}$	$\{(q_1, Z_0)\}$
ε	$\emptyset$	$\emptyset$	$\emptyset$

$\delta(q_1, i, t) =$

i/t	0	1	Z_0
0	$\{(q_1, \varepsilon)\}$	$\emptyset$	$\emptyset$
1	$\emptyset$	$\{(q_1, \varepsilon)\}$	$\emptyset$
#	$\emptyset$	$\emptyset$	$\emptyset$
ε	$\emptyset$	$\emptyset$	$\{(q_2, Z_0)\}$

and $\delta(q_2, t, i) = \emptyset$ for all $t \in \Gamma$, $i \in \Sigma$.

With the intuitive definition of the semantics we have sketched, one can understand that the self-loop on state q_0 consists in pushing all 0 and 1 read from the input. Indeed, whenever a 0 is read, the automaton systematically tests for all possible characters x (that can be either 0 or 1 or Z_0) on the top of the stack, and replaces it by $0x$, which amounts to pushing a 0 (and symmetrically when a 1 is read). The PDA moves from state q_0 to q_1 only when a # is read on the input, and does not modify the stack in this case. Then, the self-loop on q_1 consists, when reading a 0, in checking that the top of the stack is indeed a 0 too, and replacing it by ε, i.e., popping the

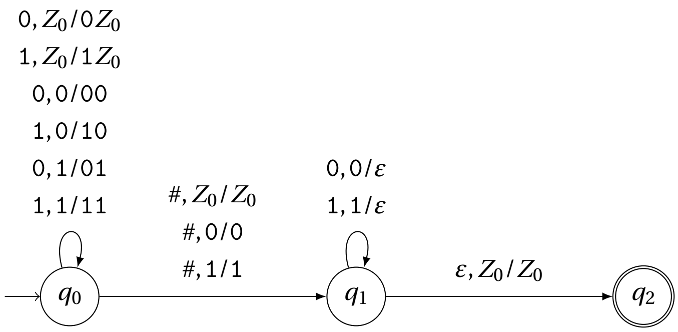


Figure 4.11: An example PDA recognising $L_{pal\#}$ (by accepting state).

0 (and, again, symmetrically when a 1 is read). So the self-loop on q_1 pops all the stack content while checking that the characters that are popped correspond to the symbol read on the input. Because of the LIFO properties of the stack, this amounts to checking that the suffix of the input word that occurs *after* the # is the mirror image of the prefix that occurs *before* the #. Finally, the PDA moves to the accepting state q_2 only when the stack is empty, which guarantees that the mirror image of the whole prefix had been found in the suffix. Hence, the automaton indeed recognises $L_{pal\#}$. Let us make these notions more precise by formally defining the semantics of PDAs.

We have silently assumed that, to accept a word, a PDA must reach an accepting state, as in the case of finite automata. While this is sufficient for the intuitive examples we are discussing for the moment, bear in mind that we will later refine this notion by defining two kinds of acceptance conditions for PDAs.

Configuration of a PDA As for finite automata, a *configuration* of a PDA describes the situation in which the different components of the automaton are while it is busy reading a word. In the case of a PDA, this configuration contains the following information:

1. the current state (an element from Q);
2. the remaining input word $w \in \Sigma^*$; and
3. the current content of the stack (an element from Γ^*).

This is exactly captured by the following definition:

Definition 4.12 (Configuration of a PDA)**.** A *configuration* of a PDA $\langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\rangle$ is a triple

$$\langle q, w, \gamma\rangle \in Q \times \Sigma^* \times \Gamma^*.$$

In particular, the *initial configuration* when reading w is $\langle q_0, w, Z_0\rangle$, i.e., initially, the current state is q_0, w is on the input and Z_0 is on the stack.

Configuration change Let us now formally define how a PDA can move from one configuration to another. As for finite automata, we use the $c \vdash_P c'$ notation to denote the fact that the PDA P can move from configuration c to configuration c'. Thanks to the syntax introduced in Definition 4.11, the definition of $\vdash$ is very simple:

Definition 4.13 (Configuration change of a PDA)**.** Let us consider a PDA $P = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\rangle$. Then, we say that *P can move from configuration*

$\langle q, aw, X\beta\rangle$ *to configuration* $\langle q', w, \alpha\beta\rangle$ (with $a \in \Sigma \cup \{\varepsilon\}$, and $X \in \Gamma$) iff there is $(q', \alpha) \in \delta(q, a, X)$. In this case, we write:

$$\langle q, aw, X\beta\rangle \vdash_P \langle q', w, \alpha\beta\rangle.$$

Let us elucidate this definition to ensure that it captures the intuition we have sketched so far. The original configuration is $\langle q, aw, X\beta\rangle$, where a is a single input letter, or ε; and X is a stack symbol. Let us assume for the moment that $a \neq \varepsilon$. Thus, in the original configuration $\langle q, aw, X\beta\rangle$, the remaining input is non-empty and begins by letter a. Moreover, symbol X is on the top of the stack, and the PDA is in state q. Thus, we should consider $\delta(q, a, X)$, that contains all the possible moves[10] that the PDA can do in this configuration. All these possible moves are pairs (q', α), where q' is the destination state and α is the sequence of symbols that should replace X on the top of the stack after the transition. Thus, the resulting configuration is obtained from $\langle q, aw, X\beta\rangle$ by: (i) changing the current state from q to q'; (ii) reading the a at the beginning of the input, hence only w remains; and (iii) popping the X from the stack and pushing α instead. Thus, the resulting configuration is indeed $\langle q', w, \alpha\beta\rangle$. We can now check that the intuition still works when $a = \varepsilon$. Indeed, in this case, $aw = \varepsilon \cdot w = w$, and the input word is thus not modified by the transition.

[10] Thus, as said above, PDAs can be non-deterministic.

Note that, when the PDA is clear from the context, we might omit the subscript on $\vdash$. We also denote by $\vdash_P^*$ (or, simply, $\vdash^*$) the reflexive and transitive closure♣ of $\vdash$.

Example 4.14. Consider for instance the PDA in Figure 4.11, and the input word $\texttt{01\#10} \in L_{pal\#}$. The initial configuration on this input word is $(q_0, \texttt{01\#10}, Z_0)$. One can check against the definition of $\vdash$ that:

$$(q_0, \texttt{01\#10}, Z_0) \vdash (q_0, \texttt{1\#10}, \texttt{0}Z_0)$$

because $\delta(q_0, \texttt{0}, Z_0) = \{(q_0, \texttt{0}Z_0)\}$

One can further check that the following sequence of configurations can be visited by the PDA until the input is empty:

$$(q_0, \texttt{01\#10}, Z_0) \vdash (q_0, \texttt{1\#10}, \texttt{0}Z_0) \vdash (q_0, \texttt{\#10}, \texttt{10}Z_0) \vdash (q_1, \texttt{10}, \texttt{10}Z_0) \vdash (q_1, \texttt{0}, \texttt{0}Z_0) \vdash (q_1, \varepsilon, Z_0) \vdash (q_2, \varepsilon, Z_0).$$

Hence, we can write that:

$$(q_0, \texttt{01\#10}, Z_0) \vdash^* (q_2, \varepsilon, Z_0)$$

Accepted language Equipped with this notion, we can now define which words are *accepted* by a PDA. As said above, we will actually define two notions of accepted languages. Indeed, one very natural notion of acceptance for PDAs is obtained by adapting the definition we have adopted for finite automata: a word w is accepted iff there is at least one run reading this word and reaching a final state. However, as shown by the example in Figure 4.11, another natural notion of acceptance for PDAs is to accept a word when the *stack is empty*. Intuitively, in many cases, the stack is used as a memory to store some sort of input that still must be treated, so, it is reasonable to accept a word as soon as this pending input is empty. These two notions are captured by the following definition:

Definition 4.15 (Accepted languages of a PDA). Let us consider a PDA $P = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$. Then:

1. Its *final state accepted language*, denoted $L(P)$ is:

$$L(P) = \{w \mid \text{there are } q \in F \text{ and } \gamma \in \Gamma^* \text{ s.t. } \langle q_0, w, Z_0 \rangle \vdash_{\mathrm{P}}^* \langle q, \varepsilon, \gamma \rangle\}$$

2. Its *empty stack accepted language*, denoted $N(P)$ is:

$$N(P) = \{w \mid \text{there is } q \in Q \text{ s.t. } \langle q_0, w, Z_0 \rangle \vdash_{\mathrm{P}}^* \langle q, \varepsilon, \varepsilon \rangle\}$$

In other words:

1. A word w is in $L(P)$ (i.e., it is accepted by final state) iff, from the initial configuration $\langle q_0, w, Z_0 \rangle$ where w is in the input, one can find an execution reaching a configuration $\langle q, \varepsilon, \gamma \rangle$ where w has been read entirely and the current state q is accepting ($q \in F$). Observe that the stack does not need to be empty for w to be accepted (γ is any word in Γ^*).

2. On the other hand, a word w is in $N(P)$ (i.e., it is accepted by empty stack) iff, from the initial configuration $\langle q_0, w, Z_0 \rangle$, one can find an execution reaching a configuration $\langle q, \varepsilon, \varepsilon \rangle$ where w has been read entirely and the stack is empty (observe that, in this case, the current state q does *not* need to be final: $q \in Q$).

Example 4.16. Considering again the sequence of transitions from Example 4.14:

$$(q_0, \mathtt{01\#10}, Z_0) \vdash^* (q_2, \varepsilon, Z_0)$$

we deduce that $\mathtt{01\#10} \in L(P)$, where P is the PDA in Figure 4.11, because, q_2 is an accepting state, and the input is empty in the last configuration. Observe that the stack is *not* empty, but this is not necessary for a word to be in $L(P)$.

Now consider a PDA P' obtained from P by deleting state q_2, and adding, on q_1 a self-loop transition labelled by $\varepsilon, Z_0/\varepsilon$, i.e. a transition that empties the stack once the Z_0 symbol occurs on the top; and where there are no more accepting states. This PDA is shown in Figure 4.12. Then, we have:

$$(q_0, \mathtt{01\#10}, Z_0) \vdash_{\mathrm{P'}}^* (q_1, \varepsilon, \varepsilon)$$

i.e., there is an execution of the PDA that reaches $(q_1, \varepsilon, \varepsilon)$ where the stack is empty (but where q_1 is not accepting). This entails that: $\mathtt{01\#10} \in N(P')$.

Observe, however that $\mathtt{01\#10} \not\in N(P)$, because P never empties its stack; neither that $\mathtt{01\#10} \not\in L(P')$ because P' has no accepting state. That is, we can show that $L(P) = N(P') = L_{pal\#}$, and that $L(P') = N(P) = \emptyset$.

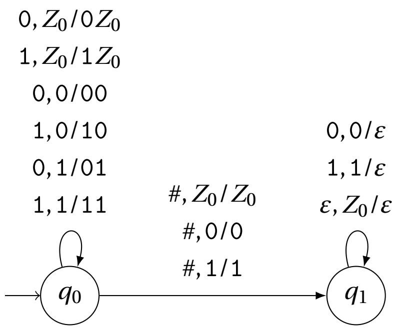


Figure 4.12: An example PDA recognising $L_{pal\#}$ (by empty stack).

4.2.2 *Deterministic* PDAs

When considering finite state automata, we have shown that ε-NFAs and DFAs are *equivalent* in the sense that any ε-NFA can always be turned into a DFA that accepts the same language. Is it the case for PDAs? Unfortunately, the answer is 'no': there are some non-deterministic PDAs that cannot be turned into an equivalent deterministic one. Let us give an example.

Consider the language L_{pal}, which is obtained from $L_{pal\#}$ by *deleting the # in all words*. Formally:

$$L_{pal} = \left\{ww^R \mid w \in \{0,1\}^*\right\}$$

Thus, for instance, $010010 \in L_{pal}$, but $0100 \notin L_{pal}$.

Now, let us consider again the PDA in Figure 4.11, which recognises $L_{pal\#}$. Intuitively, the presence of the # symbol in the middle of all words in $L_{pal\#}$ 'makes the life of the PDA easier' because it marks the end of the prefix w, and 'tells' the PDA when to move to state q_1 where it should start popping from the stack. Unfortunately, this feature is not present anymore in the words of L_{pal}, so a PDA recognising this language (whether by accepting state of by empty stack), cannot determine when the prefix of the palindrome has been read. Instead, it *must* 'guess' it, i.e., by using non-determinism. This yields the PDA in Figure 4.13 (a similar PDA accepting L_{pal} by empty stack can be obtained from the PDA in Figure 4.12).

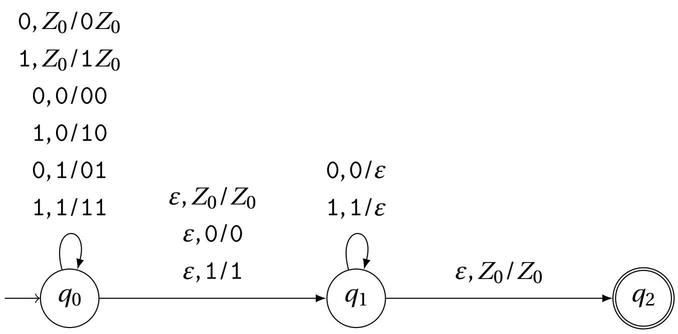


Figure 4.13: An non-deterministic PDA recognising L_{pal} (by accepting state).

Clearly, the PDA in Figure 4.13 is *non-deterministic*, because the transition from q_0 to q_1 does not consume any character on the input, so it can be fired any time. As an example, from the initial configuration

$(q_0, \texttt{0110}, Z_0)$, the two following successors are possible:

$$(q_0, \texttt{0110}, Z_0) \vdash (q_0, \texttt{110}, \texttt{0}Z_0)$$
$$\text{and}$$
$$(q_0, \texttt{0110}, Z_0) \vdash (q_1, \texttt{0110}, Z_0)$$

Moreover, in the latter case, the only possible next configuration change is:

$$(q_1, \texttt{0110}, Z_0) \vdash (q_2, \texttt{0110}, Z_0),$$

i.e., state q_2 is reached without modification of the stack, and the input is not empty. From this configuration $(q_2, \texttt{0110}, Z_0)$, no other configuration is reachable, hence, this *run* of the automaton will not accept the word 0110. Nevertheless, 0110 is accepted by the PDA in Figure 4.13, with the next sequence of transitions (that corresponds, as expected, to pushing the prefix 01 and popping the suffix 10):

$$(q_0, \texttt{0110}, Z_0) \vdash (q_0, \texttt{110}, \texttt{0}Z_0) \vdash (q_0, \texttt{10}, \texttt{10}Z_0) \vdash (q_1, \texttt{10}, \texttt{10}Z_0) \vdash (q_1, \texttt{0}, \texttt{0}Z_0) \vdash (q_1, \varepsilon, Z_0) \vdash (q_2, \varepsilon, Z_0).$$

Now that we have understood why non-determinism is crucial to let the PDA in Figure 4.13 accept L_{pal}, let us try and understand why there is no *deterministic* PDA that accepts it. First, let us define this last notion:

Definition 4.17 (Deterministic Pushdown automaton)**.** A *Deterministic Pushdown automaton* (DPDA) for short is a PDA $\langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\rangle$ s.t.:

1. for all $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ and $\gamma \in \Gamma$: $\delta(q, a, \gamma)$ has at most one element; and
2. for all $q \in Q$ and $\gamma \in \Gamma$: if $\delta(q, \varepsilon, \gamma) \neq \emptyset$, then $\delta(q, a, \gamma) = \emptyset$ for all $a \in \Sigma$.

The intuition behind this definition is as follows. The first condition says that, given a state q, an input letter a and a symbol γ on the top of the stack, $\delta(q, a, \gamma)$ returns at most one move, i.e., the DPDA has no 'choice'. The second condition says that if there is a transition labelled by ε from some state q and with some symbol γ on the top of the stack, then, this is the only transition active in this case. Indeed, if there were q and γ s.t. $\delta(q, \varepsilon, \gamma) \neq \emptyset$ and $\delta(q, a, \gamma) \neq \emptyset$ too (for some letter $a \in \Sigma$), then, in a configuration where the PDA is in state q, γ is on the top of the stack, and a is the first character of the input, the PDA would have the choice between taking the 'a-labelled transition' and the 'ε-labelled transition'.

The fact that there is no DPDA for *some* languages accepted by a (non-deterministic) PDA does not mean that *no* non-deterministic PDA can be determinised. Recall that, in particular, ε-NFAs are PDAs, that all ε-NFAs can be turned into an equivalent DFA, and that DFAs are DPDAs. In general, a PDA that does not use its stack, or stores only a finite amount of data on its stack, whatever the word it accepts, is equivalent to a finite automaton and can thus be determinised.

Example 4.18. The PDA accepting $L_{pal\#}$ in Figure 4.11 is a DPDA. The one in Figure 4.13 is not, because $\delta(q_0, \texttt{0}, \texttt{0}) \neq \emptyset$, and $\delta(q_0, \varepsilon, \texttt{0}) \neq \emptyset$, which violates the second condition of Definition 4.17.

Now, we can argue that there is no DPDA for the language L_{pal}, which shows that, unlike finite automata, PDAs cannot be determinised *in general*:

Theorem 4.2. *There is no* DPDA *that accepts* L_{pal}.

Proof. (Sketch) A full proof is beyond the scope of these lecture notes, so we only give the intuition. Consider two words w_1 and w_2 in L_{pal} that share a common prefix. For instance $w_1 = \texttt{0110}$ and $w_2 = \texttt{011110}$. If there is a DPDA that accepts L_{pal}, then it reaches the same configuration after reading the common prefix `01`, and performs the same configuration change when reading the next `1`. However, the behaviour of the PDA should differ when reading w_1 and w_2, since in the case of w_1 the next `1` belongs to the suffix of the word, and the PDA should check that this suffix is the mirror image of the prefix; while in the case of w_2 the `1` still belongs to the prefix. □

Every DPDA is a PDA just as every DFA is also an ε-NFA, even if the names of those classes provide the opposite intuition...

As a consequence, the class of languages that can be accepted by a DPDA is *stricly included* in the class of languages that can be accepted by a PDA. Indeed, every DPDA *is* a PDA, which proves the inclusion; and we have just identified a language that separates these two classes. We believe it is important to stress out this result since it departs from what we have observed with finite automata, where DFA accept the same languages as NFA and ε-NFA, i.e. the regular languages. We will come back to this at the end of the section.

4.2.3 Equivalence with context-free grammars

Now that we have characterised pushdown automata, let us study the *class of languages* they define, exactly as we have done when we have proved that finite automata accept the class of regular languages (Kleene's theorem). As sketched above, PDAs accept the class of *context free languages* (CFL), that we have defined so far as the *class of languages that are defined by context-free grammars* (CFGs):

Theorem 4.3. *For all* PDA*s* P, $L(P)$ *and* $N(P)$ *are both context-free languages. Conversely, for all* CFL*s* L, *there are* PDA*s* P *and* P' *s.t.* $L(P) = N(P') = L$.

We will prove this theorem in several steps:

1. First, we will show that all CFLs can be accepted by a PDA that accepts on empty stack, by giving a translation from CFGs to PDAs. Hence, we show that for all CFGs G, there is a PDA P s.t. $N(P) = L(G)$;

2. Second, we will show the reverse direction: for all PDAs P, we can build a CFG G_P s.t. $L(G_P) = N(P)$, i.e. $N(P)$ is a CFL. Together with point 1, this shows that the class of languages accepted by empty stack by a PDA is the class of CFLs. It remains to show that this holds also for PDAs that accept with final states;

3. Third, we will show that we can convert any PDA accepting by empty stack into a PDA accepting the same language by accepting state, i.e. for all PDAs P, there is a PDA P' s.t. $N(P) = L(P')$;

4. Finally, we will show that for all PDAs P, there is a PDA P' s.t. $L(P) = N(P')$.

From CFG*s to* PDA*s accepting by empty stack* For this first point, we will only show an example that should be sufficient to convince the reader of the validity of the following Lemma:

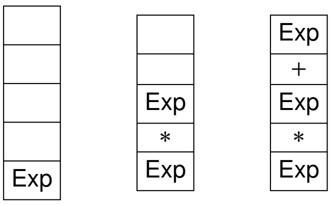
Lemma 4.4. *For all* CFG G*, there is a* PDA P *s.t.* $N(P) = L(G)$

The reason why we restrict ourselves to an example is because, in Chapter 5 and Chapter 6, we will study extensively several techniques to translate CFGs into PDAs. Indeed, as explained in the introduction, this is the key element to build a *parser*, which is the second stage of a compiler, the one that checks that the *syntax* of the input file is correct. This syntax is specified by means of a CFG, and the resulting PDA (the parser) is a device that checks the conformance of the input string to this CFG.

Let us consider again the grammar for arithmetic expressions given in Figure 4.4, and the word $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$ which is accepted by the grammar according to the following *leftmost* derivation:

$$\mathsf{Exp} \overset{2}{\Rightarrow} \mathsf{Exp} * \mathsf{Exp} \overset{1}{\Rightarrow} \mathsf{Exp} + \mathsf{Exp} * \mathsf{Exp} \overset{4}{\Rightarrow} \mathsf{Id} + \mathsf{Exp} * \mathsf{Exp} \overset{4}{\Rightarrow} \mathsf{Id} + \mathsf{Id} * \mathsf{Exp} \overset{4}{\Rightarrow} \mathsf{Id} + \mathsf{Id} * \mathsf{Id}.$$

One way for a device to check that $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$ is indeed accepted by the grammar is to build this derivation. To do so, the device needs to store, at all times, the current sentential form, and must be able to perform rewriting of variables in this sentential form. If this device is a PDA (as we would like to achieve), it is natural to store the current sentential form on the stack, with the leftmost symbol on the top. For instance, the initial content of the stack would be the start symbol only, i.e., we let $Z_0 = S$, where S is the start symbol (in our example, $Z_0 = \mathsf{Exp}$). Then, we should update the stack in order to reflect the derivations of the grammar. Graphically, the expected content of the stack for the three first sentential forms of the derivation should be:



So, now, the question is: how does the PDA update the stack? There are two possibilities to consider:

1. Either the symbol on the top of the stack is a variable V of the grammar, as in the three pictures above. In this case, the PDA must 'simulate' the derivation by finding a rule $V \to \alpha$ in the grammar, popping this variable V and pushing, instead the right-hand side α. This is what happened in our example above, where the two steps correspond to applying the rules $\mathsf{Exp} \to \mathsf{Exp} * \mathsf{Exp}$ and $\mathsf{Exp} \to \mathsf{Exp} + \mathsf{Exp}$, respectively. Observe that these actions can be implemented in a non-deterministic PDA with a single state q: for each rule $V \to \alpha$ of the grammar, we have an element (q, α) in $\delta(q, \varepsilon, V)$, i.e. we add a transition that does not change the state, does not read from the input, but checks that V is on the top of the stack and replaces it by α.

Non-determinism is crucial here: if there are two rules of the form $V \to \alpha_1$ and $V \to \alpha_2$, the PDA must 'guess' which one to apply when seeing V on the top of the stack. The whole point of Chapter 5 and Chapter 6 will be to build a PDA that can make the 'right choices' deterministically in order to obtain a program that can be implemented.

2. Or there is, on the top of the stack, a terminal. In this case, we cannot apply any grammar rule, and we cannot access the other variables that could be deeper in the stack. This is what occurs if we further apply the rule $\mathsf{Exp} \rightarrow \mathsf{Id}$ from the third stack above to obtain:

$$\begin{array}{|c|}\hline \mathsf{Id} \\ \hline + \\ \hline \mathsf{Exp} \\ \hline * \\ \hline \mathsf{Exp} \\ \hline \end{array}$$

   where the Id on the top of the stack 'hides' the Exp variables. However, in this case, we are sure that the word which will be generated by the derivation we are currently simulating *will* start by Id+, i.e. the two terminals which are on the top of the stack. So, we can check that these two terminals are indeed the two first letters on the input. If it is not the case, then, clearly, the derivation we are currently simulating will not allow to recognise the input word. So, the PDA will not be able to execute any further step, and will not reach an accepting state. On the other hand, if Id and + *are* the two next characters on the input, then, they can safely be popped and read from the input. This can be achieved by PDA transitions (still assuming a PDA with a single state q): for all $a \in T$, we have in $\delta(q, a, a)$ an element (q, ε), i.e. we add a transition that does not change the content of the stack, but reads a character a from the input, provided that it is present on the top of the stack, and pops it. Eventually, if the input word is accepted, this will empty the stack, so we obtain a PDA that accepts the language of the grammar by empty stack.

To finish with our example, let us depict the PDA obtained from the grammar in Figure 4.4, using the technique described above. It is shown in Figure 4.14.

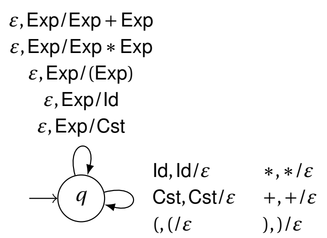


Figure 4.14: A PDA accepting (by empty stack) arithmetic expressions with + and $*$ operators only.

Moreover, one possible execution of this PDA on the input string $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$ is:

$$(q, \mathsf{Id}+\mathsf{Id}*\mathsf{Id}, \mathsf{Exp}) \vdash (q, \mathsf{Id}+\mathsf{Id}*\mathsf{Id}, \mathsf{Exp}*\mathsf{Exp}) \vdash (q, \mathsf{Id}+\mathsf{Id}*\mathsf{Id}, \mathsf{Exp}+\mathsf{Exp}*\mathsf{Exp}) \vdash$$
$$(q, \mathsf{Id}+\mathsf{Id}*\mathsf{Id}, \mathsf{Id}+\mathsf{Exp}*\mathsf{Exp}) \vdash (q, +\mathsf{Id}*\mathsf{Id}, +\mathsf{Exp}*\mathsf{Exp}) \vdash (q, \mathsf{Id}*\mathsf{Id}, \mathsf{Exp}*\mathsf{Exp}) \vdash$$
$$(q, \mathsf{Id}*\mathsf{Id}, \mathsf{Id}*\mathsf{Exp}) \vdash (q, *\mathsf{Id}, *\mathsf{Exp}) \vdash (q, \mathsf{Id}, \mathsf{Exp}) \vdash (q, \mathsf{Id}, \mathsf{Id}) \vdash (q, \varepsilon, \varepsilon)$$

which shows that $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$ is indeed accepted, as $(q, \varepsilon, \varepsilon)$ is an accepting configuration (the stack is empty).

From PDA *with empty stack to* CFG In this case, again, we will restrict ourselves to presenting an example of translation from a PDA that accepts with empty stack to a CFG that accepts the same language. The construction is quite technical, but, fortunately, the intuitions behind the construction are quite simple. For our example, we will consider again the PDA in Figure 4.12 recognising $L_{pal\#}$ by empty stack. Recall that, in a PDA, the execution starts with the Z_0 symbol on the top of stack (so, initially, the stack is not empty), and that the ultimate goal of the PDA is to empty its stack to accept a word. This is why the variables in the CFG we will build are of the form:

$$[p\gamma q]$$

where: q and p are two (possibly equal) states of the PDA; $\gamma \in \Gamma$ is a possible stack symbol; and the intuitive meaning of the variable is that *the set of words that can be generated from* $[p\gamma q]$ *is exactly the set of all words that are accepted by the* PDA *when: (i) it starts its execution in state* p *with* γ *as the content of the stack; and (ii) it ends its execution in state* q. In other words:

$$[p\gamma q] \Rightarrow^* w$$
$$\text{iff}$$
$$(p, w, \gamma) \vdash^* (q, \varepsilon, \varepsilon).$$

In addition to those variables, we will also have an S variable, which is the start symbol of the grammar. So, following the intuition we have given above, the rules of our grammars that have S as the left-hand side must be:

$$S \rightarrow [q_0 Z_0 q_0]$$
$$S \rightarrow [q_0 Z_0 q_1].$$

Indeed, for a word to be accepted by the PDA in Figure 4.12, the PDA must:

- either have an execution that starts in q_0, eventually removes the symbol Z_0 from the stack (so that it becomes empty), and reaches q_0. By the intuition above, the sets of all words accepted by such executions is the set of words generated by $[q_0 Z_0 q_0]$;
- or have an execution that starts in q_0, eventually removes the symbol Z_0 from the stack and reaches q_1. Again, these words are those generated by $[q_0 Z_0 q_1]$.

There are no other possibilities since q_0 and q_1 are the only two states in the PDA.

Now, let us see how we can add to our grammar the rules that have variables of the form $[p\gamma q]$ as the left-hand side. Let us consider the variable $[q_0 \texttt{0} q_1]$. So, we need to understand what are the words that the PDA could accept by an execution that: (i) starts in q_0 with `0` as the only content of the stack; and (ii) ends in q_1 with an empty stack. For that purpose, we look at all the possible transitions that can occur from q_0 when `0` is on the top of the stack. There are three possibilities:

- Either the PDA reads a `1` on the input. In that case, it will push this new `1` to the top of the stack, and stay in q_0. This means that an execution that should empty the stack must, after this transition, pop *two* symbols from the stack: first the `1`, then the `0` that was already on the stack. In-between these two pops, the PDA might either stay in q_0 or move to q_1. Thus, we add to our grammar two rules:

$$[q_0 \texttt{0} q_1] \rightarrow \texttt{1}[q_0 \texttt{1} q_0][q_0 \texttt{0} q_1]$$
$$[q_0 \texttt{0} q_1] \rightarrow \texttt{1}[q_0 \texttt{1} q_1][q_1 \texttt{0} q_1].$$

Intuitively, the first rule says: 'we want the PDA to read a word from a configuration where q_0 is the current state and 0 is on the top of the stack, and we want that after this word is read, the PDA reaches q_1 and the 0 on the top of the stack has been popped ($[q_0 0 q_1]$). Then, one way to do that is to read a 1 from the input (which will push the 1 on the top of the stack), then continue the execution by reaching q_0 again after having removed that 1 from the top of the stack ($[q_0 1 q_0]$), then continue again the execution until reaching q_1 after which the 0 on the top of the stack has been popped ($[q_0 0 q_1]$). The second rule says essentially the same, except that now the intermediary state is q_1.

- Another possibility is that the PDA reads a 0 from the input. Symmetrically to the previous case, we have the two following rules in the grammar:

$$[q_0 0 q_1] \to 0[q_0 0 q_0][q_0 0 q_1]$$
$$[q_0 0 q_1] \to 0[q_0 0 q_1][q_1 0 q_1].$$

- Finally, one possibility is that the PDA reads a # from the input. In this case, it will not modify the stack (so the 0 that we want eventually to pop will remain), and it will move to q_1. Thus, the only rule we have in this case is:

$$[q_0 0 q_1] \to \#[q_1 0 q_1].$$

Now, let us consider variable $[q_1 0 q_1]$. For this variable, we must look for all the possible words that the PDA can accept from a configuration where q_1 is the current state, 0 is on the top of the stack, and the run of the PDA ends in q_1 and the 0 has been popped. Since no push can occur from q_1, the only word that can be accepted this way is 0—the only possible transition from $(q_1, w, 0)$ is the one that pops a 0 and reads a 0 from the input, and thus w is necessarily equal to 0.

If we continue with this intuition, we obtain[11]:

(1)	S	$\to$	$[q_0 Z_0 q_0]$
(2)		$\to$	$[q_0 Z_0 q_1]$
(3)	$[q_0 0 q_0]$	$\to$	$0[q_0 0 q_0][q_0 0 q_0]$
(4)		$\to$	$1[q_0 1 q_0][q_0 0 q_0]$
(5)	$[q_0 1 q_0]$	$\to$	$0[q_0 0 q_0][q_0 1 q_0]$
(6)		$\to$	$1[q_0 1 q_0][q_0 1 q_0]$
(7)	$[q_0 Z_0 q_0]$	$\to$	$0[q_0 0 q_0][q_0 Z_0 q_0]$
(8)		$\to$	$1[q_0 1 q_0][q_0 Z_0 q_0]$
(9)	$[q_0 0 q_1]$	$\to$	$0[q_0 0 q_0][q_0 0 q_1]$
(10)		$\to$	$0[q_0 0 q_1][q_1 0 q_1]$
(11)		$\to$	$1[q_0 1 q_0][q_0 0 q_1]$
(12)		$\to$	$1[q_0 1 q_1][q_1 0 q_1]$
(13)		$\to$	$\#[q_1 0 q_1]$
(14)	$[q_0 1 q_1]$	$\to$	$0[q_0 0 q_0][q_0 1 q_1]$
(15)		$\to$	$0[q_0 0 q_1][q_1 1 q_1]$
(16)		$\to$	$1[q_0 1 q_0][q_0 1 q_1]$
(17)		$\to$	$1[q_0 1 q_1][q_1 1 q_1]$
(18)		$\to$	$\#[q_1 1 q_1]$
(19)	$[q_0 Z_0 q_1]$	$\to$	$0[q_0 0 q_0][q_0 Z_0 q_1]$
(20)		$\to$	$0[q_0 0 q_1][q_1 Z_0 q_1]$
(21)		$\to$	$1[q_0 1 q_0][q_0 Z_0 q_1]$
(22)		$\to$	$1[q_0 1 q_1][q_1 Z_0 q_1]$
(23)		$\to$	$\#[q_1 Z_0 q_1]$
(24)	$[q_1 0 q_1]$	$\to$	0
(25)	$[q_1 1 q_1]$	$\to$	1
(26)	$[q_1 Z_0 q_1]$	$\to$	ε

We finish this example by giving a (leftmost) derivation of this grammar that accepts 01#10:

$$
\begin{aligned}
S &\stackrel{2}{\Rightarrow} [q_0 Z_0 q_1] \\
&\stackrel{20}{\Longrightarrow} 0[q_0 0 q_1][q_1 Z_0 q_1] \\
&\stackrel{12}{\Longrightarrow} 01[q_0 1 q_1][q_1 0 q_1][q_1 Z_0 q_1] \\
&\stackrel{18}{\Longrightarrow} 01\#[q_1 1 q_1][q_1 0 q_1][q_1 Z_0 q_1] \\
&\stackrel{25}{\Longrightarrow} 01\#1[q_1 0 q_1][q_1 Z_0 q_1] \\
&\stackrel{24}{\Longrightarrow} 01\#10[q_1 Z_0 q_1] \\
&\stackrel{26}{\Longrightarrow} 01\#10.
\end{aligned}
$$

From PDA *with empty stack to* PDA *with accepting states* For the third step of our proof, we show how we can transform a PDA P that accepts some language $N(P)$ by empty stack, into a PDA P' that accepts the same language by accepting state.

Lemma 4.5. *For all* PDA P*, there is a* PDA P' *s.t.* $N(P) = L(P')$.

Proof. We sketch the construction of P' from P. It is illustrated in Figure 4.15 Let us assume that $P = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$. Then, we build $P' =$

[11] To keep the grammar as short as possible (!) we have avoided some variables that cannot produce anything for obvious reasons, such as $[q_1 0 q_0]$, as there is no path in the PDA from q_1 to q_0.

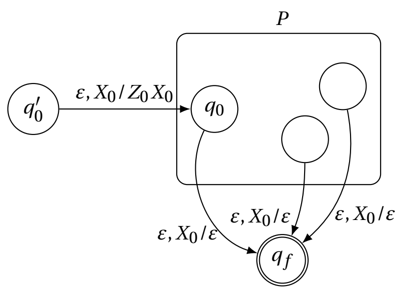


Figure 4.15: An Illustration of the construction that turns a PDA accepting by empty stack into a PDA accepting the same language by final state.

$\langle Q', \Sigma, \Gamma \cup \{Z_0\}, \delta', q_0', X_0, F' \rangle$ where Q', δ' and q_0' are defined according to the following intuition. The idea behind the construction of P' is that P' should somehow 'simulate' P, detect whenever P empties its stack, and then move to an accepting state. To do so, P' will push, on its stack, the symbol Z_0 which is the symbol that is used to mark the bottom of the stack in P; and then move to the initial state of P. That is, the first move of P' will be: $(q_0', w, X_0) \vdash (q_0, w, Z_0 X_0)$. From that configuration, P' can act exactly as P. Indeed, the transitions of P cannot test for the X_0 symbol which is at the bottom of the stack, so this has no influence on the execution of P. Eventually, P will empty its stack by popping Z_0, so we should make sure that P' accepts in this case. When this occurs, the symbol on the top of the stack, in P' will be X_0. So, we can add, from all states of P, a transition that tests for X_0 on the top of the stack, and moves to an accepting state in this case.

Formally:

- $Q' = Q \cup \{q_0', q_f\}$, where q_0' is a new initial state, and q_f is a new accepting state;
- $\delta'(q_0', \varepsilon, X_0) = \{(q_0, Z_0 X_0)\}$, and $\delta'(q_0', a, X_0) = \emptyset$ for all $a \in \Sigma$. That is, P' can only push Z_0 on top of X_0 from its initial state q_0' and then move to q_0;
- for all states $q \in Q$: $\delta'(q, \varepsilon, X_0) = \{(q_f, \varepsilon)\}$, and $\delta'(q, a, X_0) = \emptyset$ for all $a \in \Sigma$. Otherwise, δ' coincide with δ. That is, we add transitions to the accepting state only when X_0 occurs on the top of the stack;
- $\delta'(q_f, a, \gamma) = \emptyset$ for all $a \in \Sigma \cup \{\varepsilon\}$ and $\gamma \in \Gamma$, i.e., there is no transition from the accepting state;
- $F' = \{q_f\}$.

It is easy to check that, we have in P, for some input word w:

$$(q_0, w, Z_0) \vdash^*_{\mathrm{P}} (q, w', \varepsilon)$$

iff we have, in P':

$$(q_0', w, X_0) \vdash_{\mathrm{P'}} (q_0, w, Z_0 X_0) \vdash^*_{\mathrm{P'}} (q, w', X_0).$$

That is, P' can 'simulate' the execution of P with the additional X_0 at the bottom of the stack. Then, from (q, w', X_0), P' can move to the accepting state:

$$(q, w', X_0) \vdash_{\mathrm{P'}} (q_f, w', \varepsilon).$$

Hence, P accepts w (by empty stack) iff P' does (by accepting state), i.e., $N(P) = L(P')$. □

From PDA *with accepting state to* PDA *with empty stack* We close the loop by showing how we can convert a PDA accepting some language $L(P)$ by accepting state into one accepting the same language by empty stack.

Lemma 4.6. *For all* PDA P*, there is a* PDA P' *s.t.* $L(P) = N(P')$.

Proof. The construction is very similar to the one we have used in the previous proof. Given a PDA $P\langle Q,\Sigma,\Gamma,\delta,q_0,Z_0,F\rangle$ that accepts some language $L(P)$ by accepting state, we build a PDA P' that 'simulates' the execution of P, checks when an accepting state is reached, and, in this case, moves, by an ε-transition to a state where it empties its stack.

This can be done as follows[12] (the construction is illustrated in Figure 4.16):

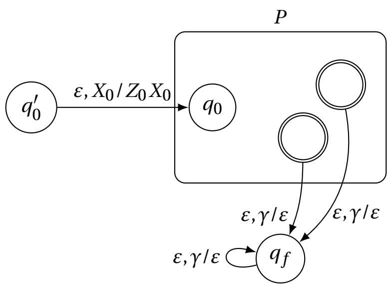


Figure 4.16: An Illustration of the construction that turns a PDA accepting by final state into a PDA accepting the same language by empty stack. Transitions labelled by $\varepsilon,\gamma/\varepsilon$ represent all possible transitions for all possible $\gamma\in\Gamma$.

[12] We skip the formal details, they can be found in classical textbooks such as: John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321455363

1. We add to P two fresh states q_0' and q_f, where q_0' is the new initial state;
2. The bottom of stack symbol of P' is now X_0;
3. From q_0' there is a single transition to q_0 (the initial state of P) that pushes Z_0 on the stack. Thus, after this initial transition, the content of the stack is Z_0X_0, and all transitions of P can be executed, hence P can be 'simulated' by P';
4. From all accepting states of P, there are transitions to q_f labelled by $\varepsilon,\gamma/\varepsilon$ (for all $\gamma\in\Gamma$) to q_f. That is, once an accepting state is reached in P, P' moves to q_f and starts popping the symbols from the stack;
5. On q_f, there are self-loop transitions labelled by $\varepsilon,\gamma/\varepsilon$ (for all $\gamma\in\Gamma$), to empty the stack.

Again, it is easy to check that, there is an execution

$$(q_0,w,Z_0)\vdash^*_{\mathrm{P}}(q,w',x)$$

in P *iff* there is an execution of the form

$$(q_0',w,X_0)\vdash_{\mathrm{P'}}(q_0,w,Z_0X_0)\vdash^*_{\mathrm{P'}}(q,w',xX_0)$$

in P'. In the case where $q\in F$ (that is, q is accepting in P), then, from (q,w',xX_0), P' can move to q_f, where it will empty its stack, i.e.:

$$(q,w',xX_0)\vdash^*_{\mathrm{P'}}(q_f,w',\varepsilon),$$

where this last configuration is accepting for the 'empty stack' condition. Hence, w is in $L(P)$ iff w is in $N(P')$ □

4.2.4 Deterministic Context-Free Languages

We close this section by considering again the special case of DPDA. We start by introducing a name and a notation for the class of languages recognised by DPDA. This naming is due to GINSBURG and GREIBACH[13]:

[13] Seymour Ginsburg and Sheila Greibach. Deterministic context free languages. *Information and Computation (formerly known as Information and Control)*, 9(6):620–648, 1966. ISSN 0019-9958. DOI: 10.1016/S0019-9958(66)80019-0. URL https://www.sciencedirect.com/science/article/pii/S0019995866800190

Definition 4.19 (Deterministic CFL)**.** A CFL L is *deterministic* (DCFL for short) iff there is a DPDA P that accepts it, i.e. $L(P)=L$.

By the discussion at the end of Section 4.2.2, we already know that some CFL *cannot* be recognised by deterministic PDA, so we can state the following about DCFL:

Observe that, thanks to the equivalence between acceptance conditions we have given above, we could have written $N(P)$ instead of $L(P)$.

Lemma 4.7. *The class of deterministic context-free languages is stricly contained in the class of context-free languages:* DCFL $\subsetneq$ CFL.

4.3 Operations and closure properties of context-free languages

Let us now study the operations we can apply to context-free languages, and what are their effect. The operations we are interested in are the ones we have introduced in Section 1.4, namely: union, intersection, concatenation, complement and Kleene closure.

More precisely, we are interested by the *closure properties* of CFLs, i.e., given two CFLs L_1 and L_2 to which we apply one of the operations listed above, is the resulting language still a CFL?

Recall that for the class of *regular languages*, the answer to those questions is always 'yes': the union, the intersection and the concatenation of two regular languages are regular; and the complement and Kleene closure of a regular language are also regular. This is a remarkable property of regular languages.

Union, concatenation and Kleene closure The union, the concatenation and the Kleene closure are three operations that preserve the context-free character of languages. Let us prove this:

Theorem 4.8. *Let L_1 and L_2 be two* CFL*s. Then, $L_1 \cup L_2$, $L_1 \cdot L_2$ and L_1^* are* CFL*s.*

Proof. As L_1 and L_2 are CFLs, there are CFGs $G_1 = \langle V_1, T_1, P_1, S_1 \rangle$ and $G_2 = \langle V_2, T_2, P_2, S_2 \rangle$ that accept them. Then, let us show how we can combine them to produce grammars G, G' and G'' that accept $L_1 \cup L_2$, $L_1 \cdot L_2$ and L_1^* respectively .

Instead of showing how to apply these operations on *grammars* recognising L_1 and L_2, we could also consider two PDA P_1 and P_2 recognising them and show how to combine them to produce PDA accepting $L_1 \cup L_2$, $L_1 \cdot L_2$ and L_1^* respectively. This construction would be in the spirit of the translation from regular expressions to ε-NFA (see Section 2.4.1).

- For the union, we build the grammar

$$G = \langle V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P, S \rangle,$$

  where S is fresh variable and:

$$P = P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}.$$

  That is, all the rules from both grammars are added to G; and the extra rules allow the grammar G to 'chose' between L_1 or L_2. More precisely, if a word is generated from S (i.e. $S \Rightarrow^* w$), then it is necessarily generated either from S_1 (i.e., the derivation is actually $S \Rightarrow S_1 \Rightarrow^* w$) or from S_2 (i.e., the derivation is actually $S \Rightarrow S_2 \Rightarrow^* w$). Thus, w belongs either to L_1 or to L_2, i.e. $w \in L_1 \cup L_2$. We have just shown that $L(G) \subseteq L_1 \cup L_2$.

  Symmetrically, if $w \in L_1 \cup L_2$, then either $w \in L_1$ or $w \in L_2$. In the former case, $w \in L_1$ implies $S_1 \Rightarrow^*_{G_1} w$, hence $S \Rightarrow_G S_1 \Rightarrow^*_G w$ and thus $w \in L(G)$. In the latter case, $w \in L_2$ implies $S_2 \Rightarrow^*_{G_2} w$, hence $S \Rightarrow_G S_2 \Rightarrow^*_G w$ and thus $w \in L(G)$, again. This shows that $L_1 \cup L_2 \subseteq L(G)$. Together with $L(G) \subseteq L_1 \cup L_2$, this implies that $L(G) = L_1 \cup L_2$.

- For the concatenation, we build the grammar

$$G' = \left\langle V_1 \cup V_2 \cup \{S'\}, T_1 \cup T_2, P', S' \right\rangle,$$

  where S' is fresh variable and:

$$P' = P_1 \cup P_2 \cup \{S' \to S_1 S_2\}$$

  It is easy to check that a word is generated by G' iff it is generated from the sentential form $S_1 S_2$, hence w is the concatenation of $w_1 \in L_1$ and $w_2 \in L_2$, and $L(G) = L_1 \cdot L_2$.

- For the Kleene closure, we proceed in a similar fashion, with a recursive rule. We build the grammar

  $$G'' = \langle V_1 \cup \{S''\}, T_1, P'', S'' \rangle,$$

  where S'' is fresh variable and:

  $$P'' = P_1 \cup \{S'' \to S_1 S'', S'' \to \varepsilon\}.$$

  Again, one can easily check that w is accepted by G'' iff it is generated from a sentential form $S_1 S_1 \cdots S_1$. In other words, w is a concatenation of an arbitrary number of words generated from S_1. Hence, $L(G'') = L_1^*$.

□

Intersection and complement Unfortunately, neither intersection, nor complement do preserve the context-free character of languages. That is, one can find two languages L_1 and L_2 that are CFLs, but, s.t. $L_1 \cap L_2$ is not a CFL. From this, we will also be able to deduce that CFLs are not closed under complement either.

To establish those results, we need to admit the following one:

Theorem 4.9. *The language* $L_{\texttt{abc}} = \{\texttt{a}^n\texttt{b}^n\texttt{c}^n \mid n \geq 0\}$ *is not context free.*

Theorem 4.9 can be proved by invoking the *pumping lemma* for context-free languages, which is remotely related to the arguments we have used to show that $L_{()}$ is not regular, at the beginning of the present chapter.

It is not difficult to see that, if we admit PDAs with *two* stacks, then $L_{\texttt{abc}}$ can be recognised. Indeed, such a PDA would push an a on *both* stacks when reading the a's, pop from the first when reading the b's and from the second when reading the c's. However, PDA with two stacks can simulate Turing machines, so adding only a second stack to PDA greatly increases its expressive power, so this model cannot be used for practical purposes in a compiler.

The intuition behind this result is as follows. Assume we have a PDA accepting $L_{\texttt{abc}}$. Clearly, such a PDA needs to *count* the number of a's that are on the input, then check that this number corresponds to the number of b's *and* to the number of c's. To count this number of a's, the PDA has no other choice than pushing all the a's on its stack. Then, to check that the number of b's is equal to the number of a's, the PDA must necessarily pop all the a's from the stack while reading the b's. However, at that point, the stack is empty, so the count of the number of a's has been lost, and the PDA cannot check anymore that the number of c's is correct.

Now, using this result, let us prove that the intersection of two CFLs might *not* be a CFL.

Theorem 4.10. CFL*s are not closed under intersection.*

Proof. Consider the two languages:

$$L_1 = \{\texttt{a}^n\texttt{b}^n\texttt{c}^k \mid k, n \geq 0\}$$
$$L_2 = \{\texttt{a}^k\texttt{b}^n\texttt{c}^n \mid k, n \geq 0\}.$$

So, L_1 contains all words of the form $\texttt{a}\cdots\texttt{ab}\cdots\texttt{bc}\cdots\texttt{c}$ s.t. the number of a's is equal to the number of b's, and L_2 all the words of the same form s.t. the number of b's is equal to the number of c's.

It is easy to check that L_1 and L_2 are CFLs. A PDA accepting L_1 would push on the stack all the a's it reads, then check that the number of b's is equal by popping from the stack, and finally read c's without constraint. Symmetrically, a PDA accepting L_2 would read a prefix of a's, push when reading the b's, then pop when reading the c's[14]

So, L_1 and L_2 are both CFLs, but clearly $L_1 \cap L_2 = L_{\texttt{abc}}$, which is not a CFL by Theorem 4.9. □

[14] Another argument to show that L_1 and L_2 are CFLs is to observe that L_1 is the concatenation of $\{\texttt{a}^n\texttt{b}^n \mid n \geq 0\}$ with $\texttt{c}^*$, while L_2 is the concatenation of $\texttt{a}^*$ with $\{\texttt{b}^n\texttt{c}^n \mid n \geq 0\}$. Clearly, all these languages are CFLs (in particular, $\texttt{a}^*$ and $\texttt{c}^*$ are regular), so their concatenation is also a CFL, by Theorem 4.8.

Finally, we can show that:

Theorem 4.11. CFL*s are not closed under complement*

Proof. The argument is by contradiction and is based on the previous Theorem. Let L_1 and L_2 be two CFLs, and assume, for the sake of contradiction, that, for all CFLs L, $\overline{L} = \Sigma^* \setminus L$ is a CFL. Then, consider the language:

$$\overline{\left(\overline{L_1} \cup \overline{L_2}\right)}.$$

Clearly, since L_1 and L_2 are CFLs, this language is a CFL too: by hypothesis, $\overline{L_1}$ and $\overline{L_2}$ are CFLs; so $\overline{L_1} \cup \overline{L_2}$ is a CFL too by Theorem 4.8, hence, its complement is a CFL. However, classical set theory tells us that:

$$L_1 \cap L_2 = \overline{\left(\overline{L_1} \cup \overline{L_2}\right)}.$$

Thus, we conclude that, if the complement of any CFL L were a CFL too, then the intersection of any pair of CFLs would be a CFL, which we know is not the case by Theorem 4.9. Contradiction. □

To conclude, CFLs do not enjoy all the nice closure properties of regular languages. This is not very surprising: an increase in the expressive power usually comes at the price of a loss of properties.

One can also note that some problems are *undecidable* for CFLs, such as *inclusion* for instance. To mitigate these problems, the class of *visibly pushdown languages* (VPL) has been introduced. They form an intermediary class between regular languages and CFLs, while retaining enough expressive power to have interesting applications. The class of VPL is closed under all classical operations (union, intersection, complement, Kleene star, concatenation), and inclusion is decidable.

Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, STOC '04, pages 202–211, New York, NY, USA, 2004. ACM. ISBN 1-58113-852-0. DOI: 10.1145/1007352.1007390. URL http://doi.acm.org/10.1145/1007352.1007390

4.4 *Grammar transformations*

Let us close this chapter by considering several techniques that will turn to be useful when we will build *parsers* for a given grammar, in order to produce syntactic analysers. Those techniques ensure that the grammars we consider have certain important properties for the kind of parsers we will consider (the importance of these properties will thus become clear in the course of Chapter 5). In particular, the transformation that consists in modifying the grammar to take into account priority and associativity of operators allows one to remove (some) ambiguities of grammars.

4.4.1 *Factoring*

The first transformation is called *factoring* and can be applied when a grammar contains at least two rules with the same left-hand side, and a common prefix in the right-hand side. A typical example is in the specification of an `if` in an imperative language:

(1)	[if]	→	`if` [Cond] `then` [Code] `fi`
(2)	[if]	→	`if` [Cond] `then` [Code] `else` [Code] `fi`

In this case, `if` [Cond] `then` [Code] is the common prefix. Clearly, it is possible to 'factor' this common prefix and transform this grammar into:

(1)	[if]	→	`if` [Cond] `then` [Code] [ifSeq]
(2)	[ifSeq]	→	`fi`
(3)	[ifSeq]	→	`else` [Code] `fi`

This latter grammar accepts the same language as the former, but there are no common prefixes in right-hand sides of rules.

Why is it so important that no two right-hand sides of rules (with the same left-hand side) share a common prefix? The intuition is as follows: when we write a *parser*, we want to produce a program that, given a word w and a grammar G, builds, if possible, a derivation of G that produces w. Assume that the parser has already built a prefix of the derivation; obtained the sentential form $x_1 V x_2$; and needs to decide which rule to apply to rewrite V. Assume further that there are in the grammar two rules, say $V \to \mathtt{a}\alpha_1$ and $V \to \mathtt{b}\alpha_2$. To make a choice between the rules, the parser will look at the next symbol in the input: if it is an `a`, then it will apply the former rule; if it is a `b`, it will apply the latter. This allows to make sure that the parser is *deterministic*. Unfor-

In general, whenever we have, in a grammar, a set of rules of the form:

$$V \to \alpha\beta_1$$
$$V \to \alpha\beta_2$$
$$\vdots$$
$$V \to \alpha\beta_n,$$

we can replace them by:

$$V \to \alpha V'$$
$$V' \to \beta_1$$
$$V' \to \beta_2$$
$$\vdots$$
$$V' \to \beta_n,$$

where V' is a fresh variable. This process can be iterated until there are no more rules to factor.

4.4.2 Removing left-recursion

As already explained, recursion in a CFG refers to the occurrence of the left-hand side of a rule in its right-hand side. For instance, in the following grammar that accepts $\mathtt{a}^*$, the first rule is recursive (and the second is used to stop the recursion):

(1)	S	$\to$	$S\mathtt{a}$
(2)	S	$\to$	ε

One speaks of *left-recursion* when this recursive variable occurs as the first symbol of the right-hand side, as in the example above. This is actually a case of *direct* recursion, but it can also be *indirect*, as in the following example:

(1)	S	$\to$	$A\mathtt{a}$
(2)	A	$\to$	$S\mathtt{b}$
(3)	A	$\to$	ε

For reasons akin to the one that have prompted us to factor rules, left-recursion will be problematic when building parsers, so we need to find a way to remove it. Of course, completely removing recursion will not be possible: recursion is the only way for a grammar to accept an infinite language, and any grammar without recursion necessarily accepts a *finite* language. So, our technique to remove direct and indirect left-recursion will be as follows:

1. First, transform indirect left-recursion into direct left-recursion;

2. then, transform left-recursion into right-recursion.

To turn indirect left-recursion into direct left-recursion, we proceed as follows. Every time there is a rule of the form:

$$A \to B\alpha$$

where A and B are variables, and where all rules with B as left-hand side are:

$$\begin{aligned} B &\to \beta_1 \\ B &\to \beta_2 \\ &\vdots \\ B &\to \beta_n, \end{aligned}$$

we *replace* $A \to B\alpha$ by:

$$\begin{aligned} A &\to \beta_1\alpha \\ A &\to \beta_2\alpha \\ &\vdots \\ A &\to \beta_n\alpha. \end{aligned}$$

Clearly, this preserves the language of the grammar. We repeat this transformation until there is no indirect left-recursion left in the grammar.

Next, we need to remove *direct* left-recursion by turning it into right-recursion. We proceed as follows. Consider a variable V and assume:

$$\begin{aligned} V &\to V\alpha_1 \\ V &\to V\alpha_2 \\ &\vdots \\ V &\to V\alpha_n \end{aligned}$$

is the set of all direct left-recursive rules with V as the left-hand side. Further assume that:

$$\begin{aligned} V &\to \beta_1 \\ V &\to \beta_2 \\ &\vdots \\ V &\to \beta_m \end{aligned}$$

is the set of all other (non-left-recursive) rules that have V as the left-hand side. Observe that a word which is generated from A is necessarily of the form:

$$w w'_1 w'_2 \cdots w'_k$$

where w is generated from one of the β_i's, and each w'_i is generated from one of the α_j's. Following this intuition, we replace all those rules by:

$$
\begin{aligned}
V &\to \beta_1 V' \\
V &\to \beta_2 V' \\
&\vdots \\
V &\to \beta_m V' \\
\\
V' &\to \alpha_1 V' \\
V' &\to \alpha_2 V' \\
&\vdots \\
V' &\to \alpha_n V' \\
V' &\to \varepsilon.
\end{aligned}
$$

As can be seen, V' is now the recursive variable, but we have used *right-recursion* to generate the sequence of α_i's.

4.4.3 Removing useless symbols

Definition of useless symbols The formal definition of CFGs we have given is a *syntactic one* (i.e., there must be exactly one variable, and no terminal, on the left-hand side); but it does not guarantee anything about the possible derivations and the use of the variables and terminals along these derivations. In particular, it is perfectly possible, but rather undesirable, to build grammars that still satisfy this definition but contain *useless symbols* (variables or terminals), as shown by the next examples.

(1)	S	$\to$	a
(2)		$\to$	A
(3)	A	$\to$	Aa

Figure 4.17: A grammar with an unproductive variable (A).

Example 4.20. Let us consider the CFG in Figure 4.17.

Clearly, any derivation that starts by $S \Rightarrow A$ will not allow one to produce any *word*, because all sentential forms derived from one containing an A will also contain an A, that can never be eliminated. In other words, the variable A is *recursive*, but there is no way to *stop* the recursion. This means that A is *useless* in this grammar (it will never allow to produce any word). So, we can safely remove rule 3 from the grammar without modifying its language. But then, we can also remove rule 2, and the grammar becomes:

(1)	S	$\to$	a

This example shows a case where a variable (A) is *unproductive*. More formally:

Definition 4.21 (Unproductive variable)**.** A variable A in a grammar $G = \langle V, T, P, S\rangle$ is *unproductive* iff there is no word $w \in T^*$ s.t. $A \Rightarrow^*_G w$.

Example 4.22. Our second example shows a case of a symbol that *is* productive but is nonetheless useless because no sentential form obtained from the start symbol will ever contain it. Consider the grammar in Figure 4.18.

(1)	S	$\to$	A
(2)	A	$\to$	a
(3)	B	$\to$	b

Figure 4.18: A grammar with an unreachable symbol (B).

In this case, variable B *is* productive because $B \Rightarrow^* b$, but it can never be 'reached' in any sentential form produced from S. Remark that it is also the case with terminal b that occurs only in rule 3 (whereas all terminals are necessarily *productive*).

Observe that symbols can be become *unreachable* because some rules have been removed due to the removal of *unproductive* symbols.

Definition 4.23. Let $G = \langle V, T, P, S\rangle$ be a grammar. A symbol $X \in V \cup T$ is *unreachable* iff there is no sentential form of G that contains an X, i.e. there is no derivation of the form $S \Rightarrow_G^* \alpha_1 X \alpha_2$.

Now, let us devise algorithms to compute symbols that are unproductive or unreachable (i.e., the *useless* symbols). More precisely, we will present algorithms to compute all the productive and reachable symbols, then use this information to remove symbols and rules that are useless.

Unproductive symbols First, for unproductive symbols, remember that all terminals are always productive. Moreover, if we consider a rule of the form $A \to \alpha$, where α contains *only productive symbols*, then A is clearly also productive.

Example 4.24. If we have $A \to aBC$, where $B \Rightarrow^* bb$ and $C \Rightarrow^* cc$, we can compose these two derivations and obtain $aBC \Rightarrow^* abbC \Rightarrow^* abbcc$, and thus $aBC \Rightarrow^* abbcc$ Hence, we also have $A \Rightarrow aBC \Rightarrow^* abbcc$, and A is productive.

Based on this observation, we can devise an iterative algorithm that computes the set of productive symbols of a CFG (it is given in Algorithm 4). This algorithm maintains a set of symbols that are productive for sure. Initially, it contains all the terminals. Then, the algorithm considers iteratively all the rules of the grammar, and, each time it finds a rule of the form $A \to \alpha$ where all symbols in α are productive, it adds A to the set of productive symbols. The algorithm grows the set of productive symbols this way until it reaches a fixed point. Upon termination, all the productive symbols have been computed, so, all the others are unproductive.

Input: A CFG $G = \langle V, T, P, S\rangle$
Output: The set $Prod \subseteq V \cup T$ of productive symbols

$Prod \leftarrow T$;
$Prec \leftarrow \emptyset$;
while $Prec \neq Prod$ **do**
    $Prec \leftarrow Prod$;
    foreach $A \to \alpha \in P$ **do**
        if $\alpha \in Prod^*$ **then**
            $Prod \leftarrow Prod \cup \{A\}$;
return $Prod$;

Algorithm 4: The algorithm to compute productive symbols in a CFG.

Once the set *Prod* of productive symbols has been computed, removing unproductive symbols from $G = \langle V, T, P, S\rangle$ yields $G' = \langle V', T, P', S\rangle$, where $V' = Prod \cap V \cup \{S\}$, and P' contains all the rules of the form $A \to \alpha \in P$ s.t. $\alpha \in Prod^*$, i.e., α contains only productive symbols.

Observe that we keep S in V even when S is not productive, because the syntax of grammars requests that V always contains at least the start symbol. However, if S is unproductive, the set of rules P will not contain any rule of the form $S \to \alpha$.

Unreachable symbols Next, let us devise an algorithm to compute *reachable* symbols. We follow the same kind of inductive reasoning as in the case of productive symbols: clearly the start symbol S is reachable. Then, if a variable A is reachable, and there is a rule $A \rightarrow \alpha$, then all symbols in α are reachable too.

Based on this observation, we obtain an algorithm to compute reachable symbols that maintains at all times a set *Reach* of symbols that are surely reachable. Initially, this set contains only S. Then, each time a rule $A \rightarrow \alpha$ with $A \in \mathit{Reach}$ is found, all the symbols from α are inserted in *Reach*. The algorithm grows the set *Reach* until a fixed point is found. Upon termination, the set *Reach* contains all reachable symbols, and only those. Algorithm 5 presents this algorithm.

This algorithm can be assimilated to a breadth-first search in a graph. Imagine the nodes of the graphs are the terminals and the variables of the grammar, and imagine that a rule of the form $A \rightarrow \alpha$ means that there is an edge between A and each symbol in α. Then, all the reachable symbols are exactly those that are reachable in the graph from node S. This can be computed by a breadth-first search, which is exactly what the algorithm does.

Input: A CFG $G = \langle V, T, P, S \rangle$
Output: The set $\mathit{Reach} \subseteq V \cup T$ of reachable symbols

$\mathit{Reach} \leftarrow \{S\}$;
$\mathit{Prec} \leftarrow \emptyset$;
while $\mathit{Prec} \neq \mathit{Reach}$ **do**
  $\mathit{Prec} \leftarrow \mathit{Reach}$;
  foreach $A \rightarrow \alpha \in P$ **do**
    if $A \in \mathit{Reach}$ **then**
      Add to *Reach* all symbols occurring in α ;
return *Reach* ;

Algorithm 5: The algorithm to compute reachable symbols in a CFG.

Again, removing unreachable symbols from $G = \langle V, T, P, S \rangle$ is easy once the set *Reach* has been computed. We obtain the CFG $G' = \left\langle V', T', P', S \right\rangle$, where $V' = \mathit{Reach} \cap V$; $T' = \mathit{Reach} \cap T$; and P' contains all the rules of the form $A \rightarrow \alpha \in P$ s.t. $A \in \mathit{Reach}$.

Removing all useless symbols Finally, let us show how we can combine the two operations described above to obtain a grammar that contains no useless symbols. Consider the following grammar:

(1)	S	$\rightarrow$	a
(2)		$\rightarrow$	A
(3)	A	$\rightarrow$	AB
(4)	B	$\rightarrow$	b

where, A is *unproductive*; B is productive; and both A and B are reachable. Removing A from the grammar as well as rules 2 and 3 yields the grammar:

(1)	S	$\rightarrow$	a
(2)	B	$\rightarrow$	b

that indeed contains only productive symbols, but where B is not reachable anymore (indeed, it was reachable 'through' A which has been removed). We conclude that *removing unproductive symbols can create unreachable symbols*: after removing unproductive symbols, we will need to run the algorithm to remove unreachable symbols.

Does the reverse hold? That is, is it possible that removing unreachable symbols make some symbols unproductive? We will argue that this is not possible, by contradiction. Assume some variable A in a grammar G which is productive (i.e., $A \Rightarrow_G^* w$ for some $w \in T^*$), and assume we remove the unreachable symbols from the grammar, and that, after this removal, the resulting grammar G' still contains A which is now unproductive, i.e. there is no w s.t. $A \Rightarrow^*_{G'} w$. Clearly, if A cannot produce any word in G', while it could in G, it is because all possible derivations that produce a word from A in G make use of one of the removed unreachable symbols. That is, for all $w \in T^*$: $A \Rightarrow^*_G w$ implies that $A \Rightarrow^*_G \alpha \Rightarrow^*_G w$, where α contains at least one variable B which is *not reachable* in G. However, since we have assumed that A is still present in G' after removal of unreachable symbols, we conclude that A is reachable, hence, $A \Rightarrow^*_G \alpha$ (with B occurring in α) implies that B is reachable too. Contradiction.

The conclusion of this discussion is that *removing unproductive symbols can create unreachable ones*, but that *removing unreachable symbols will not make variables unproductive*. Thus, to remove all useless symbols from a grammar, one should:

1. First, remove unproductive variables;

2. then, remove unreachable symbols.

After that, all variables are guaranteed to be productive, and all symbols to be reachable.

Example 4.25. We close this section by a complete example showing the removal of useless symbols. Consider the grammar $G = \langle V, T, P, S\rangle$, where $V = \{S, A, B, C, D\}$, $T = \{a, b, c\}$ and P contains the set of rules:

(1)	S	$\rightarrow$	A
(2)		$\rightarrow$	CCa
(3)	A	$\rightarrow$	Da
(4)		$\rightarrow$	ABc
(5)	B	$\rightarrow$	b
(6)	C	$\rightarrow$	c

First, we compute the set of productive symbols. Following Algorithm 4, we initialise *Prod* with T, i.e.:

$$Prod = \{a, b, c\}.$$

Considering rule 6, we discover that C is productive too, because the right-hand side contains only $c \in Prod$, so now:

$$Prod = \{a, b, c, C\}.$$

Similarly, rule 5 tells us that B is productive, so:

$$Prod = \{a, b, c, C, B\}.$$

Next, by rule 2, we conclude that S is productive, since the right-hand side of the rule contains only elements from *Prod* (a and C),:

$$Prod = \{a, b, c, C, B, S\}.$$

However, we cannot add A to *Prod*, because the right-hand sides of rules 3 and 4 both contain a symbol (D and A respectively) that does not belong to *Prod*. So the set of productive symbols is exactly $\{a, b, c, B, C, S\}$, and removing the unproductive symbols from the grammar yields $G' = \langle V', T, P', S\rangle$, where $V' = \{S, B, C\}$, and P' contains the rules:

(1)	S	$\rightarrow$	CCa
(2)	B	$\rightarrow$	b
(3)	C	$\rightarrow$	c

Now, we compute the *reachable* symbols in G'. We start with:

$$Reach = \{S\}.$$

By rule 1, we discover that C and a are reachable too, hence:

$$Reach = \{S, C, a\}.$$

Now that C is known to be reachable, we deduce, by rule 3, that terminal c is reachable too:

$$Reach = \{S, C, a, c\}.$$

At that point, we reach a fixed point: there is no rule of the grammar that allows to reach neither B nor b from either S or C, so $\{S, C, a\}$ is exactly the set of reachable symbols. The resulting grammar is $G'' = \langle V'', T'', P'', S\rangle$, where $V'' = \{S, C\}$, $T'' = \{a, c\}$ and P'' contains the rules:

(1)	S	$\rightarrow$	CCa
(2)	C	$\rightarrow$	c

which contains only useful symbols.

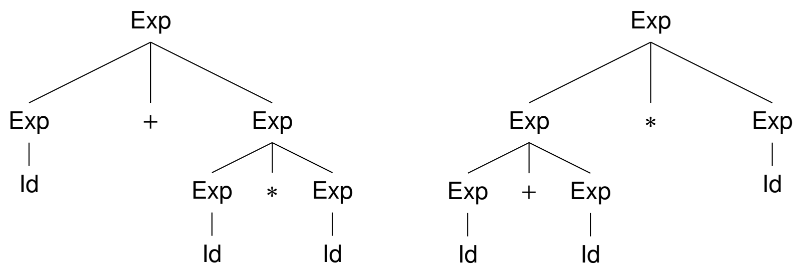
4.4.4 *Priority and associativity of operators*

We close this chapter by explaining a technique that allows to remove ambiguities occurring typically in grammars designed for arithmetic or Boolean expressions. We will consider once again the grammar for arithmetic expressions that we recall in Figure 4.19.

(1)	Exp	$\rightarrow$	Exp + Exp
(2)		$\rightarrow$	Exp ∗ Exp
(3)		$\rightarrow$	Id
(4)		$\rightarrow$	Cst

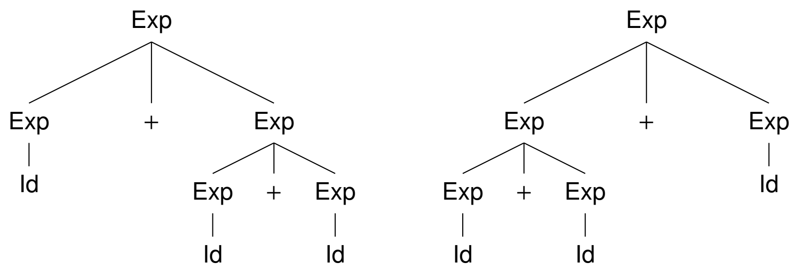
Figure 4.19: A simple grammar for arithmetic expressions.

As we have already discussed in Section 4.1.1 this grammar is *ambiguous*. Consider for instance the word Id + Id ∗ Id, the two following trees are derivation trees of this word:



If we want to build a parser for this grammar, which is a deterministic program, we will need to choose which of these trees will be returned by the parser (and then modify the grammar to make sure that only this tree is returned). To guide our choice, we will take into account the natural *priority* and *associativity* properties of the arithmetic and Boolean operators. In the example above, the tree we want to be returned by the parser is the one on the left. Indeed, this tree clearly show that the expression is the sum of the first identifier, on the one hand; and of the product of the second and third identifiers, on the other hand. Symbolically, this expression is thus equivalent to $\mathsf{Id} + (\mathsf{Id} * \mathsf{Id})$, which is indeed the right priority.

However, ambiguities occur even when operator priority doesn't play any role. Consider for instance the word $\mathsf{Id} + \mathsf{Id} + \mathsf{Id}$. In this case, the two following derivation trees are possible:



Now, the tree that we want to obtain is the one on the right, because it corresponds to the expression $(\mathsf{Id} + \mathsf{Id}) + \mathsf{Id}$, which is indeed the correct associativity for the + operator (the associativity is on the left).

Modifying the grammar Let us now modify the grammar to lift these ambiguities and make sure that the only derivation trees that will be returned by the parser are those that enforce the priority and associativity of the operators. We discuss priority first. Intuitively, for the priority of the $*$ and $+$ operators to be enforced, an expression must be a *sum of products of atoms*, where an *atom* is a basic building block, i.e. either an Id or a Cst. For instance, an expression like $\mathsf{Id} * \mathsf{Id} + \mathsf{Id} * \mathsf{Id}$ must be regarded as the sum of the two products $\mathsf{Id} * \mathsf{Id}$, which means that we will compute the values of those products first, then take their sum. This can be reflected in the grammar, by introducing fresh variables corresponding to the concepts of 'products' and 'atoms', and to modify the rules in order to enforce a hierarchy between these concepts. In the case of the grammar of Figure 4.19, we would first introduce rules:

$$\begin{aligned}\mathsf{Atom} &\to \mathsf{Id}\\ &\to \mathsf{Cst}\end{aligned}$$

to define the notion of 'atom'. Then, we introduce the notion of 'product' (of atoms), using a recursive rule as there can be as many atoms as we want in a product:

$$\begin{aligned}\mathsf{Prod} &\to \mathsf{Prod} * \mathsf{Atom}\\ &\to \mathsf{Atom}.\end{aligned}$$

Using the same canvas, we define an Exp as a sum of products, and we obtain the grammar:

(1)	Exp	$\to$	Exp + Prod
(2)		$\to$	Prod
(3)	Prod	$\to$	Prod $*$ Atom
(4)		$\to$	Atom
(5)	Atom	$\to$	Cst
(6)		$\to$	Id

Observe that the resulting grammar is left-recursive (and we will see hereinafter that this *left* recursion is crucial). However, we can use the techniques of Section 4.4 to remove this left-recursion, if need be.

Let us check that this new grammar is not ambiguous, and that the derivation trees indeed respect the priority of operators. Consider again the word $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$. Its (unique) derivation tree is the one shown in Figure 4.20. Clearly, this tree respects the priority of the operators.

Now, let us consider the word $\mathsf{Id} + \mathsf{Id} + \mathsf{Id}$. Its derivation tree is given in Figure 4.21. Since we have used *left-recursion* in the rules associated to Exp and Prod, the left associativity of the operator is naturally respected.

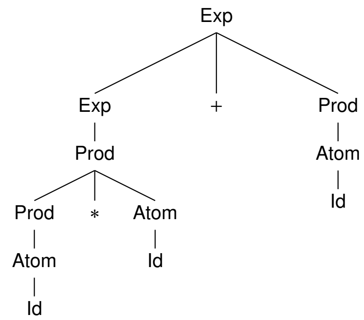


Figure 4.20: The derivation tree of $\mathsf{Id}*\mathsf{Id}+\mathsf{Id}$ taking into account the priority of the operators.

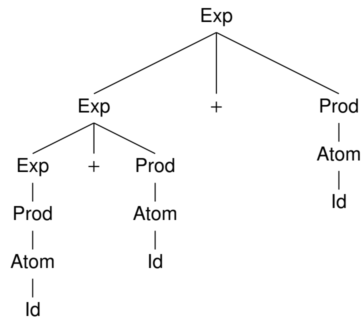


Figure 4.21: The derivation tree of $\mathsf{Id}+\mathsf{Id}+\mathsf{Id}$ taking into account the associativity of the operators.

Unary minus and parenthesis Let us now consider a more complex, yet typical example, where we allow the use of parenthesis and of the unary minus (in addition to the – and / operators that were missing in the previous grammar):

(1)	Exp	→	Exp + Exp
(2)		→	Exp − Exp
(3)		→	Exp ∗ Exp
(4)		→	Exp / Exp
(5)		→	(Exp)
(6)		→	−Exp
(7)		→	Id
(8)		→	Cst

Let us first discuss the case of the unary minus. Clearly, an expression like $-\mathsf{Id} + \mathsf{Id}$ must be understood as $(-\mathsf{Id}) + \mathsf{Id}$, and not as $-(\mathsf{Id} + \mathsf{Id})$, i.e., the minus always ranges on the next atom. Thus, we should incorporate the unary minus to the definition of atom:

$$\begin{aligned}\mathsf{Atom} &\to -\mathsf{Atom}\\ &\to \mathsf{Id}\\ &\to \mathsf{Cst}.\end{aligned}$$

We handle (Exp) similarly. Indeed, the parentheses mean that the expression must be considered as a basic building block, and the priority of the operators within the parenthesis must not interfere with the operators outside the parenthesis. So, we obtain the grammar:

(1)	Exp	→	Exp + Prod
(2)		→	Exp − Prod
(3)		→	Prod
(4)	Prod	→	Prod ∗ Atom
(5)		→	Prod / Atom
(6)		→	Atom
(7)	Atom	→	−Atom
(8)		→	Cst
(9)		→	Id
(10)		→	(Exp)

Finally, after removing left-recursion, we obtain:

(1)	Exp	$\rightarrow$	Prod Exp′
(2)	Exp′	$\rightarrow$	$+$Prod Exp′
(3)		$\rightarrow$	$-$Prod Exp′
(4)		$\rightarrow$	ε
(5)	Prod	$\rightarrow$	Atom Prod′
(6)	Prod′	$\rightarrow$	$*$Atom Prod′
(7)		$\rightarrow$	/Atom Prod′
(8)		$\rightarrow$	ε
(9)	Atom	$\rightarrow$	$-$Atom
(10)		$\rightarrow$	Cst
(11)		$\rightarrow$	Id
(12)		$\rightarrow$	(Exp)

which is a grammar that we will be able to exploit when building parsers, as explained in the next chapter.

4.5 Exercises

4.5.1 Pushdown automata

Exercise 4.1. Give a PDA that accepts the language containing all words of the form ww^R where w is any given word on the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and w^R is the mirror image of w. Test your automaton on the input word `abaaaaba`, by giving an accepting run of your automaton on this word. Does your automaton accept by empty stack or by accepting state?

Exercise 4.2. **(Exam question in 2014)** Give the diagram of a deterministic pushdown automaton, on the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$,that accepts the language $L = \{(\mathtt{ab}).\mathtt{c}(\mathtt{de})^n \mid n \geq 0\}$ using the empty stack acceptance condition.

4.5.2 Grammar transformations

Exercise 4.3. Remove the useless symbols in the following grammars:

The techniques to do so have been described in Section 4.4.3.

(1)	S	$\rightarrow$	a
(2)		$\rightarrow$	A
(3)	A	$\rightarrow$	AB
(4)	B	$\rightarrow$	b

(1)	S	$\rightarrow$	A
(2)		$\rightarrow$	B
(3)	A	$\rightarrow$	aB
(4)		$\rightarrow$	bS
(5)		$\rightarrow$	b
(6)	B	$\rightarrow$	AB
(7)		$\rightarrow$	Ba
(8)	C	$\rightarrow$	AS
(9)		$\rightarrow$	b

Exercise 4.4. Consider the following grammar:

(1)	E	$\rightarrow$	$E\ op\ E$
(2)		$\rightarrow$	$\mathtt{ID}[E]$
(3)		$\rightarrow$	$\mathtt{ID}$
(4)	op	$\rightarrow$	$*$
(5)		$\rightarrow$	$/$
(6)		$\rightarrow$	$+$
(7)		$\rightarrow$	$-$
(8)		$\rightarrow$	$\Rightarrow$

1. Show that it is *ambiguous*.

See Definition 4.5 and Section 4.4.4.

2. The priorities of the various operators are as follows: [] and $\Rightarrow$ have higher priority than $*$ and $/$, which have higher priority than $+$ and $-$. Modify the grammar to take operator precedence into account as well as left associativity.

Exercise 4.5. Left-factor the following production rules:

See Section 4.4 for the techniques.

(1)	stmt	$\rightarrow$	`if` expr `then` stmt − list `end if`
(2)	stmt	$\rightarrow$	`if` expr `then` stmt − list `else` stmt − list `end if`

Exercise 4.6. Apply the left recursion removal algorithm to the following grammar:

See Section 4.4 for the techniques.

(1)	E	$\rightarrow$	$E+T$
(2)		$\rightarrow$	T
(3)	T	$\rightarrow$	$T*P$
(4)		$\rightarrow$	P
(5)	P	$\rightarrow$	$\mathtt{ID}$

Exercise 4.7. (Excerpt from an exam question) Remove unproductive symbols and then inaccessible symbols from the following grammar:

(1)	S	$\rightarrow$	$\mathtt{a}E$
(2)		$\rightarrow$	$\mathtt{b}F$
(3)	E	$\rightarrow$	$\mathtt{b}E$
(4)		$\rightarrow$	ε
(5)	F	$\rightarrow$	$\mathtt{a}F$
(6)		$\rightarrow$	$\mathtt{a}G$
(7)		$\rightarrow$	$\mathtt{a}HD$
(8)	G	$\rightarrow$	$G\mathtt{c}$
(9)		$\rightarrow$	d
(10)	H	$\rightarrow$	$C\mathtt{a}$
(11)	C	$\rightarrow$	$H\mathtt{b}$
(12)	D	$\rightarrow$	$\mathtt{ab}$

Then, remove left-recursion and perform left-factoring whenever possible.

5 Top-down parsers

Parsing is the second step of the compiling process. During this stage, the compiler analyses the syntax of the input program to check its correctness. Just as we have formalised the scanning step using finite automata, we will rely on pushdown automata to define rigorously what a parser does.

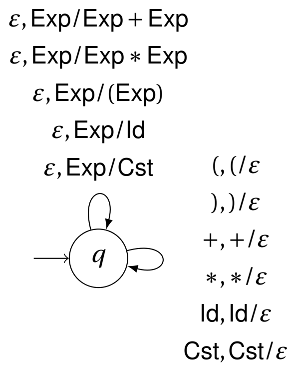
More precisely, in this chapter, we will define a first major family of parsers, namely the *top-down* parsers. In the next chapter, we will study a different family of parsers, the *bottom-up* parsers. As their names indicate, these parsers work in two completely, and actually opposite ways: a top-down parser tries and build a derivation tree for the input string starting from the root, applying the grammar rules one after the other, until the sequence of tree leaves forms the desired string. On the other hand, a bottom-up parser builds a derivation tree starting from the leaves (i.e., the input string), follow *backwards* the derivation rules of the grammar, until it manages to reach the root of the tree. We will see that these two paradigms have their own merits. Top-down parsers are perhaps more intuitive, but bottom-up parsers are more powerful. Historically, compilers such as `gcc` were written by hacking the code of an automatically generated bottom-up parser. Nowadays, recent versions of `gcc` or `clang` use a hand-written top-down parser[1].

For these two main families of parsers, we will present techniques that allow one (when possible) to build *automatically* parsers from grammars, which is exactly what we need in the framework of compiler design.

[1] GCC wiki: new C parser. `https://gcc.gnu.org/wiki/New_C_Parser`, 2008. Online: accessed on December, 29th, 2015; and CLang: features and goals. `http://clang.llvm.org/features.html`. Online: accessed on December, 29th, 2015

5.1 Principle of top-down parsing

We have already explained the main ideas behind top-down parsing when showing how we can turn any CFG into a PDA that accepts the same language by empty stack: see Section 4.2.3, and, in particular, the discussion of Lemma 4.4, that we recall now. Consider the grammar for arithmetic expressions in Figure 4.4. Then, a PDA that accepts (by empty stack) the language of the grammar in Figure 4.4 is the following (where the initial symbol on the stack is the start symbol of the grammar, namely Exp):



This PDA *simulates* a leftmost derivation by maintaining, at all times, on its stack, a suffix of the sentential form. For example, if we consider the word $\mathsf{Id}+\mathsf{Id}*\mathsf{Id}$ and its associated *leftmost* derivation:

$$\mathsf{Exp} \stackrel{2}{\Rightarrow} \mathsf{Exp}*\mathsf{Exp} \stackrel{1}{\Rightarrow} \mathsf{Exp}+\mathsf{Exp}*\mathsf{Exp} \stackrel{4}{\Rightarrow} \mathsf{Id}+\mathsf{Exp}*\mathsf{Exp} \stackrel{4}{\Rightarrow} \mathsf{Id}+\mathsf{Id}*\mathsf{Exp} \stackrel{4}{\Rightarrow} \mathsf{Id}+\mathsf{Id}*\mathsf{Id}$$

Then, the PDA given above will start its execution with the start symbol Exp of the grammar on the top of its stack, i.e., the execution will start in configuration:

$$(q, \mathsf{Id}+\mathsf{Id}*\mathsf{Id}, \mathsf{Exp}).$$

The PDA simulates the first step of the derivation by applying the rule $\mathsf{Exp}\rightarrow\mathsf{Exp}*\mathsf{Exp}$, which consists in *popping* the left-hand side of the rule and *pushing* the right-hand side, yielding the new configuration:

$$(q, \mathsf{Id}+\mathsf{Id}*\mathsf{Id}, \mathsf{Exp}*\mathsf{Exp}).$$

Performing twice the same operations with the rules $\mathsf{Exp}\rightarrow\mathsf{Exp}+\mathsf{Exp}$ and $\mathsf{Exp}\rightarrow\mathsf{Id}$, the PDA reaches the configuration:

$$(q, \mathsf{Id}+\mathsf{Id}*\mathsf{Id}, \mathsf{Id}+\mathsf{Exp}*\mathsf{Exp}),$$

where a *terminal* (Id) is now on the top of the stack. At this point, the PDA can check that the same terminal is on the input, consume it, and pop the terminal. This can be performed twice, and we obtain the new configuration:

$$(q, \mathsf{Id}*\mathsf{Id}, \mathsf{Exp}*\mathsf{Exp}).$$

The simulation of the derivation by the PDA goes on like that up to the point where the stack is empty and the whole input has been read.

5.1.1 *Systematic construction of a top-down parser*

Let us now formalise these ideas, and show how we can build a PDA accepting, by empty stack, the language of a given CFG. Let $G=\langle V,T,P,S\rangle$ be a CFG. We build a PDA P_G with a single state:

$$P_G = \big\langle \{q\}, T, V\cup T, \delta, q, S, \emptyset \big\rangle,$$

where δ is such that:

1. for all $A \in V$: $\delta(q,\varepsilon,A) = \{(q,\alpha) \mid A \to \alpha \in P\}$. That is, for all symbols A of the grammar, for all rules of the form $A \to \alpha$, the PDA has a transition that pops A and pushes α instead (without reading any character from the input). This operation is called a *produce* (of rule $A \to \alpha$).

2. for all $a \in T$: $\delta(q,a,a) = \{(q,\varepsilon)\}$. That is, for all terminals a of the grammar, there is a transition that reads a from the input and pops a from the stack. This operation is called a *match* (of terminal a).

3. in all other cases that have not been covered above, $\delta(a,b,c) = \emptyset$.

We can prove that this construction is indeed correct (which establishes Lemma 4.4). We only give the main ideas of the proof, the details can easily be worked out, and are left as an exercise to the reader:

Lemma 5.1. *For all* CFGs *G, the* PDA *P_G is s.t. $L(G) = N(P_G)$.*

Proof. (Sketch) The proof can be done by showing that: (i) for all words $w \in L(G)$, the leftmost derivation producing w can be simulated by an accepting run of P_G; and that (ii) all accepting runs of P_G (accepting a word w) can be mapped to a leftmost derivation of G that produces w. These two points are easily established by induction (on the lengths of the derivation and of the run, respectively). □

Non-determinism in the parser While this construction allows one to derive a PDA from any CFG, this PDA is not (yet) a parser because it is *non-deterministic*. This can be seen in the example above: when the symbol on the top of the stack is Exp, the PDA can replace it either by Exp + Exp, or by Exp ∗ Exp (among other possibilities) independently of the input string. Observe, however, that when a terminal is present on the top of the stack, the behaviour of the PDA is deterministic: it will *match* this symbol with the same symbol on the input.

This example has allowed us to pinpoint the source of potential non-determinism in the top-down parsers that we have built from CFGs. Such non-determinism *can only occur when a produce must be performed with symbol A on the top of the stack and when there are, in the original grammar, at least two rules $A \to \alpha_1$ and $A \to \alpha_2$ to choose from.* Or, put otherwise, resolving non-determinism in such PDAs amounts to answering the following question: assuming some variable A is on the top of the stack, which rule should we produce?

5.2 Predictive parsers

The rest of this chapter is devoted to identifying classes of grammars for which a deterministic parser can be achieved, if we allow this parser to make use of some extra information, that we call a *look-ahead*. This look-ahead is parametrised by a natural number k and consists of the k next characters on the input, that the parser can now take into account, *without actually reading them*, to decide which transition to take. Parsers that make use of a look-ahead are called *predictive parsers*.

(1)	S	$\to$	a
(2)		$\to$	b

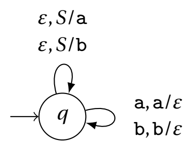


Figure 5.1: A trivial grammar and its corresponding (non-deterministic) parser (where the initial stack symbol is S).

It is important not to confuse a *look-ahead* and a *read* on the input. The look-ahead allows the parser to have a view on the future of the input, without modifying it, while a read *modifies* the input—the characters read on the input cannot be recovered.

The intuition behind the notion of look-ahead is quite simple. Consider for instance the trivial grammar on the top of Figure 5.1. The corresponding parser is displayed below. When running on the input string b, the parser is initially in the configuration $(q, \mathtt{b}, S)$, and has two non-deterministic choices: either perform a produce of the former rule, or of the latter. Clearly, knowing that the next symbol on the input is a b allows the parser to *make the right choice*. So, the grammar above can be parsed deterministically (by a top-down parser) with one character of look-ahead—this is what we will later call an LL(1) grammar.

5.2.1 *Pushdown automata with look-ahead*

In order to formalise these ideas, let us now extend the definition of PDAs by modifying their transition relation so that it can take into account the k next characters on the input (for a look-ahead of k characters). This means that the successors of a configuration will now be computed on the basis of the current state, the current stack content (as in the case of 'regular' PDAs), but also the k first characters on the input.

Observe that the only difference between this definition and that of PDAs, is that the transition function has a fourth parameter, which constitutes the look-ahead. This look-ahead is a word of k characters *at most*, since we have no guarantee that there are always k characters (or more) remaining on the input.

Definition 5.1 (k-look-ahead PDA)**.** A *pushdown automaton with k characters of look-ahead* (k-LPDA for short) is a tuple $\langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\rangle$, where all the components are defined as for PDAs (see Definition 4.11), except for the transition function δ that maps $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times \Sigma^{\leq k}$ to $2^{(Q\times\Gamma^*)}$; where:

$$\Sigma^{\leq k} = \cup_{i=0}^{k} \Sigma^i$$

is the set of all words of length at most k on the alphabet Σ.

When $k = 1$, we note LPDA instead of 1-LPDA.

Let us now define formally the new semantics that takes into account the look-ahead. We lift the notion of configuration from PDAs to k-LPDAs: Definition 4.12 carries on to k-LPDAs. The notion of configuration change, however, must be adapted:

Definition 5.2 (k-LPDA configuration change)**.** Let us consider a k-LPDA $P = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\rangle$. Let $\langle q, auv, X\beta\rangle$ be a configuration of P, where:

- $X \in \Gamma$;
- $a \in \Sigma \cup \{\varepsilon\}$;
- $u \in \Sigma^{\leq k-1}$;
- $v \in \Sigma^*$; and
- if $|auv| \geq k$ then $|au| = k$, otherwise $v = \varepsilon$ (i.e., au is a prefix of length k of the remaining input word if there are at least k characters remaining on the input. Otherwise, au contains all the remaining input).

Then, *P can move from $\langle q, auv, X\beta\rangle$ to $\langle q', uv, \alpha\beta\rangle$* iff there is $(q', \alpha) \in \delta(q, a, X, au)$. In this case, we write:

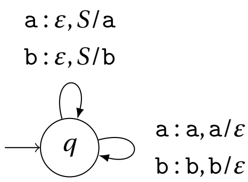
$$\langle q, auv, X\beta\rangle \vdash_P \langle q', uv, \alpha\beta\rangle .$$

The notion of configuration change is the only one we need to adapt for k-LPDAs: all the other notions regarding their semantics (the different notions of accepted language, etc) are lifted from their counterparts on PDAs. Now, let us consider an example that shows how the look-ahead can be used to obtain deterministic automata more easily.

Example 5.3. Let us consider again the trivial grammar in Figure 5.1. We can now extend its corresponding PDA with a look-ahead of one character, to obtain a deterministic LPDA. To this end, we consider the following transition function:

- $\delta(q, \varepsilon, S, \mathtt{a}) = \{(q, \mathtt{a})\}$ for the produce of $S \to \mathtt{a}$. Observe that we perform this produce only when the next character on the input is an a;
- $\delta(q, \varepsilon, S, \mathtt{b}) = \{(q, \mathtt{b})\}$ for the produce of $S \to \mathtt{b}$. Again, the produce is performed only when b is the next character on the input;
- $\delta(q, \mathtt{a}, \mathtt{a}, \mathtt{a}) = \{(q, \varepsilon)\}$ for the match of a; and
- $\delta(q, \mathtt{b}, \mathtt{b}, \mathtt{b}) = \{(q, \varepsilon)\}$ for the match of b.

Graphically, we obtain the following LPDA, assuming a label of the form $u : a, X/\beta$ means: 'if the look-ahead is u, the first character on the input is a, and the top of the stack is X, replace it by β' (in other words, we keep the same convention as for PDAs, prepending the look-ahead followed by a colon).



Observe that this LPDA is now deterministic (and can thus be implemented as a computer program in a straightforward way). Observe further that the look-ahead allows the LPDA to query the next character on the input without reading it: once again, a transition labeled by '$\mathtt{a} : \varepsilon, S/\mathtt{a}$' means that the automaton *checks* that the next character on the input is a, but *does not consume it* (as indicated by the ε), hence does not modify the input in the next configuration. On the other hand, a transition labeled by $\mathtt{a} : \mathtt{a}, \mathtt{a}/\varepsilon$ not only checks that there is an a on the input but also reads it.

Expressive power of k-LPDAs It should now be clear to the reader that LPDAs are a natural and useful class of automata to build deterministic parsers from CFGs. While LPDAs seem like a perfect tool on the practical side, it is not (yet) clear how they fit in the theory we have built so far. In other words: what is the position of k-LPDAs in the Chomsky hierarchy? Clearly, since k-LPDAs *extend* PDAs they accept all the CFLs. But could it be the case that they accept (thanks to the look-ahead) *more languages* than PDAs? The answer, fortunately is: no. We will establish this by showing that all k-LPDAs can be turned into an equivalent PDA (which, however, might be exponentially bigger and non-deterministic). So, at the end of the day, k-LPDAs are nothing more than a more convenient syntax for PDAs (but a very convenient one, as we will see!).

Proposition 5.2. *For all k-LPDAs P, we can build a PDA P' s.t. $L(P) = L(P')$.*

Proof. We will present the proof for the case where $k = 1$, the ideas are easy to generalise to $k > 1$. Let us first sketch the intuition of the proof, by explaining how executions of P' correspond to executions of P, and vice versa. First, we let the set of states of P' be pairs of the form (q, a), where q is a state of q and $a \in \Sigma$ is a single-letter that represents the current look-ahead. Note that this look-ahead will not be *computed* by P', but rather *guessed* using non-determinism, and then checked afterwards. Thus, intuitively, when P' is in (q, a), this corresponds to P being in state q and having guessed that a is the first character on the input. Following this idea, we have to define the transition function of P' so that it properly updates the look-ahead contained in its states, and checks the validity of the non-deterministic guesses, in order to keep P' synchronised with P.

To complement this intuition, recall that a finite automaton can be regarded as a program that uses a *finite amount of memory* only. This memory is encoded in the *states* of the automaton. This is the same idea that we use here. Since the look-ahead is bounded by k, and since the alphabet is finite, there are only finitely many possible values for the look-ahead, which can thus be *stored* in the states, and then queried and updated when need be, using non-determinism.

Initially, P' simply jumps to a state of the form (q_0, x) for some $x \in \Sigma \cup \{\varepsilon\}$ (thus, the x is the first guess performed by P'). Then, Figure 5.2 shows the rest of the construction. If, from state q_1, P can read some character $x \in \Sigma$ (hence, the look-ahead is necessarily equal to x, otherwise the transition cannot be taken), then, in P', we can 'simulate' this transition from (q_1, x) only (because the look-ahead *must* be x). Since the corresponding transition of P' *does* read an x, we are certain that the guess was correct. The different possible successors correspond to the different possible guesses for the next look-ahead. A special case is displayed at the bottom of the figure and occurs when P reads ε from the input. In this case, the look-ahead could be non-empty (otherwise, the look-ahead would have no interest!), and must not be updated in the state of P'.

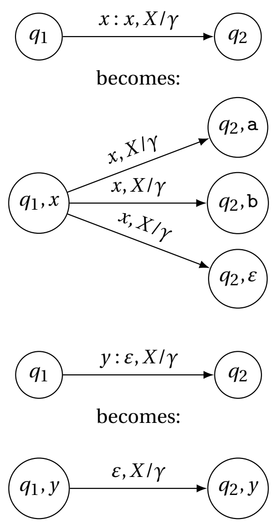


Figure 5.2: Illustration of the transformation of a k-LPDA into an equivalent PDA, assuming $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $x \in \Sigma$ and $y \in \Sigma \cup \{\varepsilon\}$.

Let us formalise this. Given an LPDA $P = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$, we build a PDA $P' = \langle Q', \Sigma, \Gamma, \delta', q'_0, Z_0, F' \rangle$ where:

- $Q' = Q \times (\Sigma \cup \{\varepsilon\}) \uplus \{q'_0\}$ (thus, q'_0 is a fresh initial state);
- $F' = F \times \{\varepsilon\}$; and
- δ' is s.t.:
  1. $\delta'(q'_0, \varepsilon, Z_0) = \left\{((q_0, x), Z_0) \,\middle|\, x \in \Sigma \cup \{\varepsilon\}\right\}$: this corresponds to the initial *guess* of the look-ahead;
  2. for all $q \in Q$, $x \in \Sigma$ and $X \in \Gamma$:
  $$\delta'\big((q, x), x, X\big) = \left\{\big((q', z), \gamma\big) \,\middle|\, (q', \gamma) \in \delta(q, x, X, x), z \in \Sigma \cup \{\varepsilon\}\right\},$$
  which corresponds to the top part of Figure 5.2;
  3. for all $q \in Q$, $y \in \Sigma$ and $X \in \Gamma$:
  $$\delta'\big((q, y), \varepsilon, X\big) = \left\{\big((q', y), \gamma\big) \,\middle|\, (q', \gamma) \in \delta(q, \varepsilon, X, y)\right\},$$
  which corresponds to the bottom part of Figure 5.2; and
  4. $\delta'\big((q, x), y, X\big) = \emptyset$ in all the cases that have not been specified above.

We finish by sketching the arguments to show that the construction is correct. First, we consider an execution of the k-LPDA P on some word w:

$$\langle q_0, w_0, Z_0\rangle \vdash_{P} \langle q_1, w_1, \gamma_1\rangle \vdash_{P} \ldots \vdash_{P} \langle q_k, w_k, \gamma_k\rangle,$$

where $w_0 = w$, $w_k \varepsilon$ and $q_k \in F$. Then, one can show by induction on the length of the execution that it corresponds to an accepting execution in P'. Assuming that for all $0 \le i \le k$, a_i denotes the first character of w_i (with $a_i = \varepsilon$ if $w_i = \varepsilon$), this accepting execution in P' is:

$$\langle q_0', w_0, Z_0\rangle \vdash_{P} \langle (q_0, a_0), w_0, Z_0\rangle \vdash_{P'} \langle (q_1, a_1), w_1, \gamma_1\rangle \vdash_{P'} \ldots \vdash_{P'} \langle (q_k, a_k), w_k, \gamma_k\rangle.$$

That is, it is the execution obtained when P' always 'guesses' correctly the next character on the input. It is easy to check that this is indeed an execution of P' (see the definition of δ' above), which is accepting because (q_k, a_k) is a final state when q_k is.

On the other hand, if

$$\langle q_0', w_0, Z_0\rangle \vdash_{P} \langle (q_0, a_0), w_0, Z_0\rangle \vdash_{P'} \langle (q_1, a_1), w_1, \gamma_1\rangle \vdash_{P'} \ldots \vdash_{P'} \langle (q_\ell, a_\ell), w_\ell, \gamma_\ell\rangle$$

is an accepting execution of P' on some word $w_0 = w$ (thus, with $w_\ell = \varepsilon$ and $q_\ell \in F$), then, one can check that:

$$\langle q_0, w_0, Z_0\rangle \vdash_{P} \langle q_1, w_1, \gamma_1\rangle \vdash_{P} \ldots \vdash_{P} \langle q_\ell, w_\ell, \gamma_\ell\rangle$$

is an accepting execution of P. This again can be done by induction on the length of the execution. This part is a bit more difficult that the reverse direction, because one has to check that all the steps are allowed by the look-ahead in P. The key point to prove this is the fact that, at the end of the execution, $w_\ell = \varepsilon$, otherwise the execution would not be accepting. Considering the definition of δ', this means necessarily that, all look-aheads 'guessed' by P' were eventually checked to be correct. Indeed, when P' performs some step $\langle (q_i, a_i), w_i, \gamma_i\rangle \vdash_{P'} \langle (q_{i+1}, a_{i+1}), w_{i+1}, \gamma_{i+1}\rangle$, then:

1. either it performs an ε-labeled transition in which case $a_i = a_{i+1}$ and $w_i = w_{i+1}$ (i.e., no check of the look-ahead is performed, but neither the input word nor the guessed look-ahead change. So the check is deferred to a later transition);
2. or a transition labeled by a_i is performed. This implies that a_i was indeed the first character of w_i, hence the 'guess' in (q_i, a_i) was correct.

□

Theorem 5.3. *For all k, the class of languages accepted by k-LPDAs is the class of CFLs.*

Proof. Since, for all k, we can translate any k-LPDA into an equivalent PDA (see Proposition 5.2), the class of k-LPDAs accepts no more than the CFLs. On the other hand, all PDAs P can trivially be translated into an equivalent k-LPDA P' (for all k): it suffices to define the transition relation of P' in such a way that it ignores the look-ahead (or, in other words, such that it performs the same actions on the input and on the stack for all possible values of the look-ahead). □

Now that we have k-LPDAs at our disposal, let us show how to transform, when possible, and in a systematic way, CFGs into *deterministic* k-LPDAs that we will be able to translate easily into programs. To this end, we need to introduce some extra definitions.

5.3 First^k *and* Follow^k

In order to introduce these two notions, we start by an extensive example.

Example 5.4. Let us consider the grammar:

(1)	A	$\rightarrow$	aaa
(2)		$\rightarrow$	Bbb
(3)		$\rightarrow$	Cdd
(4)	B	$\rightarrow$	b
(5)	C	$\rightarrow$	c
(6)		$\rightarrow$	ε

and let us assume we want to build a predictive parser with *one* character of look-ahead. There are two sources of non-determinism in this grammar: on variable A, and on variable C.

1. In the case of variable A, we need to choose between rule 1, rule 2 and rule 3. Obviously, *all words generated from A using rule 1 as the first rule in the derivation will start with an* a, so we will apply rule 1 in this case only. What about rules 2 and 3? Clearly, there will never be a B nor a C on the input, as these symbols are variables and not terminals, so we need to examine what B and C can produce:

   (a) Similarly to the case of rule 1, it is easy to see that all words produced from B will start by a b, by rule 4. So, rule 2 should be applied only when a b is on the input.

   (b) The case of C is more complicated, as C can produce either c or ε. In the former case, we expect c to be the first next character on the input to apply rule 3. In the latter case, the derivation is $A \Rightarrow C\texttt{dd} \Rightarrow \texttt{dd}$, so we expect a d as the next character on the input. We conclude that all derivations starting by rule 3 will produce words that start by c or d only.

   The case of variable A are thus summarised in Table 5.1, which gives for each look-ahead, the rule to apply when A is on the top of the stack.

	Look-ahead			
Var	a	b	c	d
A	1	2	3	3

   Table 5.1: The rules to apply when A is on the top of the stack.

   To obtain this information, we have computed, *for each rule of the form* $A \rightarrow \alpha$, the set of all the possible first characters of words that can be derived from α. Indeed, in the case of $A \rightarrow \texttt{aaa}$, all words derived from aaa start by an a; in the case of $A \rightarrow B\texttt{bb}$, all words derived from Bbb start by b; in the case of $A \rightarrow C\texttt{dd}$, all words derived from Cdd start either by c or by d. This captures the intuition behind the First^1, i.e., as we will see hereinafter, $\text{First}^1(\texttt{aaa}) = \{\texttt{a}\}$, $\text{First}^1(B\texttt{aa}) = \{\texttt{b}\}$ and $\text{First}^1(C\texttt{dd}) = \{\texttt{c}, \texttt{d}\}$. We already have the intuition that computing those sets must sometimes be done recursively: for instance $\text{First}^1(B\texttt{aa})$ is equal to $\text{First}^1(B)$, because B is the first symbol in $B\texttt{aa}$.

2. Applying this idea to rule 4 allows us to deduce immediately that we will apply this rule only when B is on the top of the stack and `b` is the next character on the input.

3. The case of variable C is, once again, interesting, as the computation of the First^1 is not sufficient. Indeed, all words that are derived from C (rule 5) necessarily start by `c`, but how do we handle the case of ε? Rule 6 does not give us any clue about the next character that we expect on the input when this rule must be applied.

   Instead, we must consider *the context* in which a C can occur, and what are the characters that could *follow* it. The only place where a C occurs is in rule 3. From this rule we can deduce that *all words generated from C will necessarily be followed by a* `d`. This is better understood by visualising the derivation tree of `dd`, i.e. the only word that one can generate from A by applying rule 6, as shown in Figure 5.3.

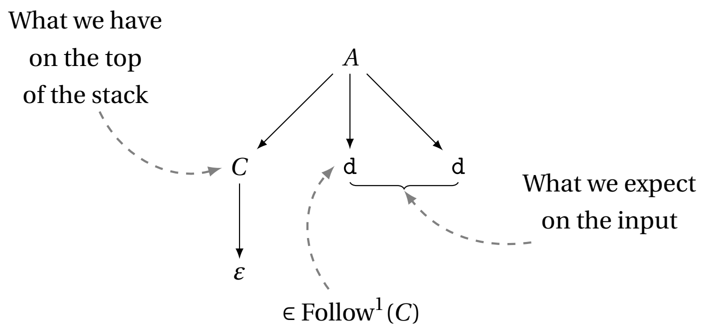


Figure 5.3: The derivation tree of `dd` and the notion of Follow^1.

This intuition is captured by the notion of Follow^1: the set $\text{Follow}^1(X)$, for some variable X will be defined as the set of all possible first characters of some words that are generated immediately after a word derived from X.

We can now complete Table 5.1 and obtain Table 5.2, which tells us exactly which rule to apply for each possible symbol on the top of stack and next character on the input. Since there is at most one rule in each cell of the table, we have now a deterministic parser at our disposal.

	Look-ahead			
Var	`a`	`b`	`c`	`d`
A	1	2	3	3
B		4		
C			5	6

Table 5.2: An *action table* for our example grammar, and a look-ahead of one character: it tells us which rule to produce for each possible symbol on the top of the stack, and each possible first character on the input.

Let us now formalise properly the notions of First^k and Follow^k.

A word w is thus in $\text{First}^k(\alpha)$ iff: (i) it is the prefix of some sentential form generated from α (i.e., $\alpha \Rightarrow^* wx$ for some w); and (ii) it has the *right size*: either it contains exactly k characters ($|w| = k$), or it contains less than k character, but this can occur only if cannot make this prefix any longer, which implies that $x = \varepsilon$ (after all, there is no reason that all sentential forms generated from α contain at least k characters).

Definition 5.5 (First^k)**.** Let $G = \langle V, T, P, S\rangle$ be a CFG, and let α be a sentential form of G (i.e., $\alpha \in (T \cup V)^*$). Then,

$$\mathrm{First}^k(\alpha) = \left\{ w \in T^* \;\middle|\; \begin{array}{c} \alpha \Rightarrow^* wx \\ \text{and} \\ \text{either } |w| = k \text{ or } \big(|w| < k \text{ and } x = \varepsilon\big) \end{array} \right\}.$$

In the case where $k = 1$, we write First(α) instead of First$^1(\alpha)$.

Definition 5.6 (Follow^k)**.** Let $G = \langle V, T, P, S\rangle$ be a CFG, and let α be a sentential form of G (i.e., $\alpha \in (T \cup V)^*$). Then,

$$\mathrm{Follow}^k(\alpha) = \left\{ w \in T^* \;\middle|\; \text{there are } \beta, \gamma \text{ s.t. } S \Rightarrow^* \beta\alpha\gamma \text{ and } w \in \mathrm{First}^k(\gamma) \right\}.$$

In the case where $k = 1$, we write Follow(α) instead of Follow$^1(\alpha)$.

The intuition behind the definition of Follow^k is easier: w is in the Follow$^k(\alpha)$ iff there is some derivation allowed by the grammar that produces α, followed by γ, and w is in the First^k of γ.

Example 5.7. Let us consider the grammar in Figure 5.4, which generates expressions. Remember that this is the grammar that we have obtained after taking into account the priority of the operators and removing left-recursion (see the last pages of Chapter 4). Observe that we have added a rule $S \rightarrow$Exp\$ to the grammar to make sure that all strings end with the marker \$. This will actually make our life easier when computing Follow sets. Let us start by considering some values of First sets:

- $\mathrm{First}(\mathsf{Atom}) = \{-, \mathsf{Cst}, \mathsf{Id}, (\}$;
- $\mathrm{First}^2(\mathsf{Atom}) = \{--, -\mathsf{Cst}, -\mathsf{Id}, -(, (-, ((, (\mathsf{Cst}, (\mathsf{Id}, \mathsf{Cst}, \mathsf{Id}\}$;
- $\mathrm{First}(\mathsf{Prod}') = \{*, /, \varepsilon\}$;
- What is the value of $\mathrm{First}^2(\mathsf{Prod}')$? We see that Prod′ produces either a string starting with $*$ and followed by some string generated by Atom; or a string starting with / and followed by some string generated by Atom; or ε. So, we can rely on First(Atom) to characterise $\mathrm{First}^2(\mathsf{Prod}')$, and we find:

$$\begin{aligned}\mathrm{First}^2(\mathsf{Prod}') &= \{*\} \cdot \mathrm{First}^1(\mathsf{Atom}) \cup \{/\} \cdot \mathrm{First}^1(\mathsf{Atom}) \cup \{\varepsilon\} \\ &= \{*-, *\mathsf{Cst}, *\mathsf{Id}, *(, /-, /\mathsf{Cst}, /\mathsf{Id}, /(, \varepsilon\}.\end{aligned}$$

Now, let us consider some values of Follow sets:

- What is $\mathrm{Follow}(\mathsf{Exp}')$? All strings generated by Exp′ necessarily appear at the end of a string generated by Exp, and all strings generated by Exp are followed by a \$ or by a) in the final output, so $\mathrm{Follow}(\mathsf{Exp}') = \{\$,)\}$.
- What is Follow(Prod)? All strings generated by Prod are followed by a string generated by Exp′. Such a string can: (i) either start by + or −, so these two symbols are in Follow(Prod); (ii) or be the empty word. In this latter case, the Follow of Prod will be the Follow of Exp′. This is sketched in Figure 5.5. Here, Prod eventually generates some string α, and Exp′ generates ε. Then, clearly, the generated word is $\alpha \cdot \varepsilon \cdot \$ = \alpha\$$ (this can be seen by inspecting the tree's leaves). This show that \$ can indeed immediately follow a string (α) generated by Prod. This reasoning holds for all symbols in $\mathrm{Follow}(\mathsf{Exp}') = \mathrm{Follow}(\mathsf{Exp}) = \{\$,)\}$. We conclude that: $\mathrm{Follow}(\mathsf{Prod}) = \{+, -, \$,)\}$.

(1)	S	→	Exp\$
(2)	Exp	→	Prod Exp′
(3)	Exp′	→	+Prod Exp′
(4)		→	−Prod Exp′
(5)		→	ε
(6)	Prod	→	Atom Prod′
(7)	Prod′	→	$*$Atom Prod′
(8)		→	/Atom Prod′
(9)		→	ε
(10)	Atom	→	−Atom
(11)		→	Cst
(12)		→	Id
(13)		→	(Exp)

Figure 5.4: The grammar generating expressions (followed by \$ as an end-of-string marker), where we have taken into account the priority of the operators, and removed left-recursion.

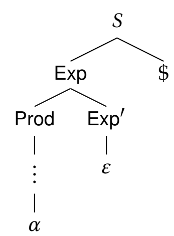


Figure 5.5: An example showing that Follow(Prod) contains \$ in a case where Exp′ generates ε.

5.3.1 Computation of First^k

While the discussion above provides us with a clean definition of First^k, it does not provide us with an algorithm to compute those sets[2]. The algorithm we are about to present is based on the following observation. Assume we want to compute $\text{First}^k(\alpha)$ for some sentential form $\alpha = X_1X_2\cdots X_n$ (with all X_i's individual terminals or variables of the grammar). Then, we can first compute $\text{First}^k(X_1)$. If all words in $\text{First}^k(X_1)$ are of length k, then, we are done. Otherwise (in particular if $\varepsilon \in \text{First}^k(X_1)$), we need to complete those elements of $\text{First}^k(X_1)$ that are not 'long enough' by elements from $\text{First}^k(X_2)$. Again, this might not be sufficient, so we compute $\text{First}^k(X_3)$, etc. This suggests Algorithm 6, a greedy algorithm that computes $\text{First}^k(X)$, for all variables X.

Initially, the algorithm computes $\text{First}^k(a)$ for all terminal a—this is actually trivial since a terminal a can only generate the word a, hence $\text{First}^k(a) = \{a\}$. Then, it initialises the sets $\text{First}^k(A)$ to $\emptyset$ for all variable $A \in V$ and grows those sets in the **repeat** loop. As long as some of the $\text{First}^k(X)$ sets have been updated, the information computed during one iteration of the loop is used to try and enrich other $\text{First}^k(A)$ sets during the next iteration, based on the rules of the grammar: for all rules $A \rightarrow X_1X_2\cdots X_n$, we re-compute the set $\text{First}^k(A)$ by concatenating $\text{First}^k(X_1)$, $\text{First}^k(X_2)$,..., $\text{First}^k(X_n)$, and truncating the resulting words to k characters[3]. This follows exactly the intuition given hereinbefore.

[2] Observe that $\text{First}^k(\alpha)$ is finite for all α since it contains words of length at most k only.

[3] Remark that we could restrict ourselves to rules of the form $A \rightarrow X_1X_2\cdots X_n$ where at least one of the X_i's has been updated in the previous iteration of the loop.

Input: A CFG $G = \langle V, T, P, S\rangle$
Output: The sets $\text{First}^k(X)$ for all $X \in V \cup T$.

foreach $a \in T$ **do**
  $\text{First}^k(a) \leftarrow \{a\}$;
foreach $A \in V$ **do**
  $\text{First}^k(A) \leftarrow \emptyset$;
repeat
  foreach $A \rightarrow X_1X_2\cdots X_n \in P$ **do**
    $\text{First}^k(A) \leftarrow$
    $\text{First}^k(A) \cup \big(\text{First}^k(X_1) \odot^{k} \text{First}^k(X_2) \odot^{k} \cdots \odot^{k} \text{First}^k(X_n)\big)$;
until *no* $\text{First}^k(A)$ *has been updated, for any* $A \in V$;

Algorithm 6: Computation of $\text{First}^k(X)$ for all $X \in V \cup T$.

Example 5.8. Let us close the discussion of Algorithm 6 by an example execution. Consider again the grammar in Figure 5.4. Here are the first few steps of execution of the algorithm (we only detail the computation of $\text{First}^1(X)$ for the variables):

0. Initially, we have: $\text{First}^1(X) = \emptyset$ for all variables $X \in V$.

1. During the first iteration, we go through all rules, and update the First of their left-hand side:

   (a) For the rule $S \rightarrow \text{Exp}\$$, we get the opportunity to update $\text{First}(S)$. We compute:

Remember that $\emptyset \cdot L = \emptyset$ for all languages L. So $\emptyset \odot^{k} L = \emptyset$ for all languages L and all k.

$$\begin{aligned}\mathrm{First}^1\big(Exp\big) \odot^1 \mathrm{First}^1\big(\$\big) &= \emptyset \odot^1 \{\$\}\\ &= \emptyset\end{aligned}$$

So, this first rule does not allow us to infer more information on $\mathrm{First}(S)$ at that point, because we have not computed any word from $\mathrm{First}(\mathsf{Exp})$ yet.

(b) Actually, at this step of computations, all rules that contain at least one variable in the right-hand side will yield a similar result, because all the First's are still empty.

(c) However, rules $Exp' \to \varepsilon$, $\mathsf{Prod}' \to \varepsilon$, $\mathsf{Atom} \to \mathsf{Cst}$ and $\mathsf{Atom} \to \mathsf{Id}$ allow us to update the $\mathrm{First}\big(\mathsf{Exp}'\big)$, $\mathrm{First}\big(\mathsf{Prod}'\big)$ and $\mathrm{First}(\mathsf{Atom})$ respectively.

So, at the end of the first iteration, we have:

X	$\mathrm{First}^1(X)$
Exp′	ε
Prod′	ε
Atom	Cst, Id

and $\mathrm{First}^1(X) = \emptyset$ for all other variables.

2. During the second iteration, we will discover new values of First^1 sets. Thanks to $\mathsf{Prod} \to \mathsf{Atom}\mathsf{Prod}'$, we add to $\mathrm{First}^1(\mathsf{Prod})$ the elements from:

$$\begin{aligned}\mathrm{First}^1(\mathsf{Atom}) \odot^1 \mathrm{First}^1\big(\mathsf{Prod}'\big) &= \{\mathsf{Cst},\mathsf{Id}\} \odot^1 \{\varepsilon\}\\ &= \{\mathsf{Cst},\mathsf{Id}\};\end{aligned}$$

to $\mathrm{First}^1\big(\mathsf{Prod}'\big)$ the elements from:

$$\begin{aligned}\mathrm{First}^1\{*\} \odot^1 \mathrm{First}^1(\mathsf{Atom}) \odot^1 \mathrm{First}^1\big(\mathsf{Prod}'\big) &= \{*\} \odot^1 \{\mathsf{Cst},\mathsf{Id}\} \odot^1 \{\varepsilon\}\\ &= \{*\},\end{aligned}$$

and from:

$$\begin{aligned}\mathrm{First}^1\{/\} \odot^1 \mathrm{First}^1(\mathsf{Atom}) \odot^1 \mathrm{First}^1\big(\mathsf{Prod}'\big) &= \{/\} \odot^1 \{\mathsf{Cst},\mathsf{Id}\} \odot^1 \{\varepsilon\}\\ &= \{/\};\end{aligned}$$

and, finally, to $\mathrm{First}^1(\mathsf{Atom})$, the elements from:

$$\begin{aligned}\mathrm{First}^1\{-\} \odot^1 \mathrm{First}^1(\mathsf{Atom}) &= \{-\} \odot^1 \{\mathsf{Cst},\mathsf{Id}\}\\ &= \{-\}.\end{aligned}$$

So, at the end of this iteration, we have:

X	$\mathrm{First}^1(X)$
Exp′	ε
Prod	Cst, Id
Prod′	$*, /, \varepsilon$
Atom	$-$, Cst, Id

and $\mathrm{First}^1(X) = \emptyset$ for all other variables.

3. The algorithm goes on similarly up to stabilisation, and computes the following values for the First^1 sets:

X	$\mathrm{First}^1(X)$
S	$-$, Cst, Id, (
Exp	$-$, Cst, Id, (
Exp′	$+, -, \varepsilon$
Prod	$-$, Cst, Id, (
Prod′	$*, /, \varepsilon$
Atom	$-$, Cst, Id, (

5.3.2 Computation of Follow^k

Let us now turn our attention to the computation of $\mathrm{Follow}^k(X)$ for all variables X of a CFG. The algorithm is given in Algorithm 7, and is, again, a greedy algorithm that grows the sets $\mathrm{Follow}^k(X)$ up to stabilisation. To do so, we rely on the following intuition: every time we have a rule of the form:

$$A \rightarrow \alpha B \beta,$$

(i.e., a rule that contains variable B in its right-hand side), we can potentially add more information to $\mathrm{Follow}^k(B)$. Indeed, a string generated by B can be followed by a string generated by β, so we can use $\mathrm{First}^k(\beta)$. Observe however, that the words in $\mathrm{First}^k(\beta)$ might be shorter than k, so we might need to complete them with $\mathrm{Follow}^k(A)$.

Input: A CFG $G = \langle V, T, P, S \rangle$
Output: The sets $\mathrm{Follow}^k(X)$ for all $X \in V$.

foreach $A \in V \setminus \{S\}$ **do**
  $\mathrm{Follow}^k(A) \leftarrow \emptyset$;
$\mathrm{Follow}^k(S) \leftarrow \{\varepsilon\}$;
repeat
  foreach $A \rightarrow \alpha B \beta \in P$ *(with $B \in V$ and $\alpha, \beta \in (V \cup T)^*$)* **do**
    $\mathrm{Follow}^k(B) \leftarrow \mathrm{Follow}^k(B) \cup \big(\mathrm{First}^k(\beta) \odot^k \mathrm{Follow}^k(A)\big)$;
until *no* $\mathrm{Follow}^k(A)$ *has been updated, for any* $A \in V$;

Algorithm 7: Computation of $\mathrm{Follow}^k(X)$ for all $X \in V$.

Example 5.9. Let us consider again the grammar from Figure 5.4, and let us apply Algorithm 7 to it, for $k = 1$.

0. Initially, we have $\mathrm{Follow}^1(X) = \emptyset$ for all variables X, *except* $\mathrm{Follow}^1(S)$, which is equal to $\{\varepsilon\}$.

1. During the first iteration, the algorithm adds \$ to $\mathrm{Follow}^1(\mathsf{Exp})$, thanks to the rule $S \rightarrow \mathsf{Exp}\$$. Indeed, this corresponds, in the algorithm, to having $A = S$, $B = \mathsf{Exp}$, $\alpha = \varepsilon$ and $\beta = \$$. So the string:

Observe that initialising $\mathrm{Follow}^1(S)$ to $\{\varepsilon\}$ is crucial here. Otherwise, if we initialise $\mathrm{Follow}(X)$ to $\emptyset$ for *all* variables X, then the algorithm would terminate after one iteration with $\mathrm{Follow}^k(X) = \emptyset$ for all X, since the expression $\mathrm{First}^k(\beta) \odot^k \mathrm{Follow}^k(A) = \mathrm{First}^k(\beta) \odot^k \emptyset$ would always evaluate to $\emptyset$.

$$\begin{aligned}\text{First}^1\big(\$\big)\odot^1\text{Follow}^1(S) &= \{\$\}\odot^1\{\varepsilon\}\\ &= \{\$\}\end{aligned}$$

is indeed added to $\text{Follow}^1(\mathsf{Exp})$.

Then, the rule $\mathsf{Exp}\to\mathsf{Prod}\mathsf{Exp}'$ allows us to grow the sets $\text{Follow}^1(\mathsf{Prod})$ and $\text{Follow}^1\big(\mathsf{Exp}'\big)$. We add to $\text{Follow}^1(\mathsf{Prod})$ the elements from:

$$\begin{aligned}\text{First}^1\big(\mathsf{Exp}'\big)\odot^1\text{Follow}^1(\mathsf{Exp}) &= \{+,-,\varepsilon\}\odot^1\{\$\}\\ &= \{+,-,\$\};\end{aligned}$$

and to $\text{Follow}^1\big(\mathsf{Exp}'\big)$ the elements from:

$$\begin{aligned}\text{First}^1(\varepsilon)\odot^1\text{Follow}^1(\mathsf{Exp}) &= \{\varepsilon\}\odot^1\{\$\}\\ &= \{\$\}.\end{aligned}$$

Etc...

2. The algorithm goes on up to stabilisation, and returns:

X	$\text{Follow}^1(X)$
S	ε
Exp	$\$,)$
Exp'	$\$,)$
Prod	$+,-,\$,)$
Prod'	$+,-,\$,)$
Atom	$*,/,+,-,\$,)$

5.4 LL(k) *grammars*

Using the tools we have just defined (First and Follow sets), we can now identify classes of CFGs for which the predictive parsers using k characters of look-ahead (as sketched above) will be deterministic. Those grammars are called LL(k) grammars, where 'LL' stands for 'Left scanning, Left Parsing', because the input string is read (scanned) from the left to the right; and the parser builds a leftmost derivation when successfully recognising the input word. This class of grammars has first been introduced by Lewis and Stearns in 1968[4], with further important refinements by Rosenkrantz and Stearns in 1970[5] (and many others afterwards...)

What are the conditions we need to impose on the *derivations* of a grammar to make sure that its corresponding parser will be deterministic when it has access to k characters of look-ahead? As we have seen already, the only possible source of non-determinism in the parser stems from the produces, more specifically, when the grammar contains at least two rules of the from $A\to\alpha_1$ and $A\to\alpha_2$. Given that these two rules exist, let us now pinpoint a situation that will confuse a parser that has access to k characters of look-ahead only. Such a situation occur if, at some point in a derivation of the grammar, A is the leftmost symbol (hence, it is on the top of the stack), and the parser must decide whether to apply $A\to\alpha_1$ or $A\to\alpha_2$, but

[4] P. M. Lewis, II and R. E. Stearns. Syntax-directed transduction. *J. ACM*, 15(3):465–488, July 1968. ISSN 0004-5411. DOI: 10.1145/321466.321477

[5] D.J. Rosenkrantz and R.E. Stearns. Properties of deterministic top-down grammars. *Information and Computation (formerly known as Information and Control)*, 17(3):226 – 256, 1970. ISSN 0019-9958. DOI: 10.1016/S0019-9958(70)90446-8

the k characters of look-ahead dot not allow it to discriminate between these two choices. To obtain such a pathological situation, we thus need to expose two different derivations that the parser cannot distinguish. For the first derivation, we can hold the following reasoning:

1. First, since we want to have A as the leftmost symbol at some point, the derivation prefix is:
$$S \Rightarrow^* wA\gamma$$
with $w \in T^*$ and $\gamma \in (V \cup T)^*$.

2. Then, let us assume that in this first derivation, the right choice is to apply $A \rightarrow \alpha_1$, i.e.,
$$wA\gamma \Rightarrow w\alpha_1\gamma.$$

3. Eventually, this derivation will produce a word, which will necessary be of the form wx_1, since w is already a string of terminals (in other words, to finish the derivation, we just need to derive the variables that potentially remain in $\alpha_1\gamma$). So, this first derivation is of the form:
$$S \Rightarrow^* wA\gamma \Rightarrow w\alpha_1\gamma \Rightarrow^* wx_1.$$

Thus, this first derivation generates the word wx_1. Let us consider again the moment in the derivation when the sentential form was $wA\gamma$ and the parser had to decide to apply $A \rightarrow \alpha_1$. As we have already remarked, A is, at that point, on the top of the stack, and w has already been read from the input. Hence, at that point, the string that remains on the input is x_1, and all the parser 'sees' is $\mathrm{First}^k(x_1)$.

Then, it is easy to build a second derivation that will confuse the parser. Assume that, in the grammar we have a derivation of the form:
$$S \Rightarrow^* wA\gamma \Rightarrow w\alpha_2\gamma \Rightarrow^* wx_2.$$

Observe that, now, the right choice to derive A is $A \rightarrow \alpha_2$, and, when the parser must take this choice, he 'sees' a look-ahead of $\mathrm{First}^k(x_2)$. So, the parser will be able to make the right decision regarding the derivation of A iff the look-ahead it has at its disposal is sufficient to tell those two situations apart, i.e.:
$$\mathrm{First}^k(x_1) \neq \mathrm{First}^k(x_2).$$

The definition[6] of LL(k) grammar is based on these intuitions: it says that, whenever a pathological situation such as the one described above occurs (the two derivations and $\mathrm{First}^k(x_1) = \mathrm{First}^k(x_2)$), then, we must have $\alpha_1 = \alpha_2$; which means that there is actually no choice to be made in the grammar. Otherwise, the parser would not be able to take a decision and the grammar would not be LL(k):

Definition 5.10 (LL(k) CFGs)**.** A CFG $\langle P, T, V, S\rangle$ is LL(k) iff for all pairs of derivations:
$$S \Rightarrow^* wA\gamma \Rightarrow w\alpha_1\gamma \Rightarrow^* wx_1$$
$$S \Rightarrow^* wA\gamma \Rightarrow w\alpha_2\gamma \Rightarrow^* wx_2$$
with $w, x_1, x_2 \in T^*$, $A \in V$ and $\gamma \in (V \cup T)^*$, *and* $\mathrm{First}^k(x_1) = \mathrm{First}^k(x_2)$, we have: $\alpha_1 = \alpha_2$.

Remember that x_1 and x_2 are *words,* so their First^k is a singleton containing one string of length at most k. This is why we can write $\mathrm{First}^k(x_1) \neq \mathrm{First}^k(x_2)$ instead of $\mathrm{First}^k(x_1) \cap \mathrm{First}^k(x_2) = \emptyset$, for instance.

[6] P. M. Lewis, II and R. E. Stearns. Syntax-directed transduction. *J. ACM*, 15(3):465–488, July 1968. ISSN 0004-5411. DOI: 10.1145/321466.321477

Observe that, if a grammar is LL(k) for some k, then it is also LL(k') for all $k' \geq k$. This is coherent with our intuition that LL(k) means 'k characters of look-ahead are sufficient'.

Example 5.11. Let us consider the following grammar:

(1)	S	$\rightarrow$	$\mathtt{a}A\mathtt{a}$
(2)	S	$\rightarrow$	$\mathtt{b}AB\mathtt{a}$
(3)	A	$\rightarrow$	$\mathtt{b}$
(4)	A	$\rightarrow$	ε
(5)	B	$\rightarrow$	$\mathtt{b}$
(6)	B	$\rightarrow$	$\mathtt{c}$

This grammar is actually quite simple, since it can generate only 6 words through 10 different derivations:

$$
\begin{aligned}
&S \Rightarrow \mathtt{a}A\mathtt{a} \Rightarrow \mathtt{aba}\\
&S \Rightarrow \mathtt{a}A\mathtt{a} \Rightarrow \mathtt{aa}\\
&S \Rightarrow \mathtt{b}AB\mathtt{a} \Rightarrow \mathtt{bb}B\mathtt{a} \Rightarrow \mathtt{bbba}\\
&S \Rightarrow \mathtt{b}AB\mathtt{a} \Rightarrow \mathtt{bb}B\mathtt{a} \Rightarrow \mathtt{bbca}\\
&S \Rightarrow \mathtt{b}AB\mathtt{a} \Rightarrow \mathtt{b}B\mathtt{a} \Rightarrow \mathtt{bba}\\
&S \Rightarrow \mathtt{b}AB\mathtt{a} \Rightarrow \mathtt{b}B\mathtt{a} \Rightarrow \mathtt{bca}\\
&S \Rightarrow \mathtt{b}AB\mathtt{a} \Rightarrow \mathtt{b}A\mathtt{ba} \Rightarrow \mathtt{bbba}\\
&S \Rightarrow \mathtt{b}AB\mathtt{a} \Rightarrow \mathtt{b}A\mathtt{ca} \Rightarrow \mathtt{bbca}\\
&S \Rightarrow \mathtt{b}AB\mathtt{a} \Rightarrow \mathtt{b}A\mathtt{ba} \Rightarrow \mathtt{bba}\\
&S \Rightarrow \mathtt{b}AB\mathtt{a} \Rightarrow \mathtt{b}A\mathtt{ca} \Rightarrow \mathtt{bca}.
\end{aligned}
$$

One can then check that:

1. this grammar is *not* LL(1). Indeed, consider the pair of derivations:

   $$S \Rightarrow \mathtt{b}AB\mathtt{a} \Rightarrow \mathtt{bb}B\mathtt{a} \Rightarrow \mathtt{bbba}$$

   and

   $$S \Rightarrow \mathtt{b}AB\mathtt{a} \Rightarrow \mathtt{b}B\mathtt{a} \Rightarrow \mathtt{bba}$$

   which both read the sentential form $\mathtt{b}AB\mathtt{a}$ after one step, corresponding to

   $$w = \mathtt{b} \text{ and } \gamma = B\mathtt{a}$$

   in the definition. In the former derivation, one applies $A \rightarrow \mathtt{b}$, while in the latter derivation, $A \rightarrow \varepsilon$ is used, corresponding to:

   $$\alpha_1 = \mathtt{b} \text{ and } \alpha_2 = \varepsilon$$

   in the definition. The resulting words are respectively $\mathtt{bbba}$ and $\mathtt{bba}$, corresponding to:

   $$x_1 = \mathtt{bba} \text{ and } x_2 = \mathtt{ba}$$

   in the definition (since $w = \mathtt{b}$), so we have:

   $$\mathrm{First}^1(x_1) = \mathrm{First}^1(x_2) = \{\mathtt{b}\}.$$

   We are thus in the conditions of the definition, yet $\alpha_1 \neq \alpha_2$, hence the definition is not satisfied, and the grammar is not LL(1).

2. this grammar, however, *is* LL(2). Proving this is a painstaking procedure as it requires to check the conditions given by the definition for many pairs of rules, but it can be done on this simple grammar. For instance, the case that we have identified in the previous item is not problematic anymore with a look-ahead of $k = 2$, since now:

   $$\{\mathtt{bb}\} = \mathrm{First}^2(x_1) \neq \mathrm{First}^1(x_2) = \{\mathtt{ba}\}$$

Bear in mind that, to prove that the grammar *is* LL(2) one needs to check *all* the pairs of derivations. This is just an example!

This example shows well that, while the definition of LL(k) grammar makes perfect sense and captures our intuition of what an LL(k) grammar should be, it is of limited use in practice when one wants to check whether a grammar is LL(k) or not. Indeed, Definition 5.10 requires to check *all possible pairs of derivations* in the grammar, and there can be infinitely many such pairs. Instead, we will now identify a stronger condition that we will be able to test, and that will still be relevant in practice.

5.4.1 *Strong* LL(k) *grammars*

Instead of relying on a *semantic* condition, as in Definition 5.10, the condition we will present now is a *syntactic* one as it concerned only with the rules of grammars. Since there are always finitely many rules (contrary to the number of derivations which can be infinite), this will allow us to derive a practical test to check whether a grammar is LL(k) or not. The definition[7] is as follows:

[7] D.J. Rosenkrantz and R.E. Stearns. Properties of deterministic top-down grammars. *Information and Computation (formerly known as Information and Control)*, 17(3):226 – 256, 1970. ISSN 0019-9958. DOI: 10.1016/S0019-9958(70)90446-8

Observe that this definition does not mention derivations, only the (finitely many) rules of the grammar.

Definition 5.12 (Strong LL(k) CFG)**.** A CFG $G = \langle V, T, P, S\rangle$ is *strong* LL(k) iff, for all pairs of rules $A \to \alpha_1$ and $A \to \alpha_2$ in P (with $\alpha_1 \neq \alpha_2$):

$$\mathrm{First}^k\Big(\alpha_1 \mathrm{Follow}^k(A)\Big) \cap \mathrm{First}^k\Big(\alpha_2 \mathrm{Follow}^k(A)\Big) = \emptyset$$

Example 5.13. Let us consider the grammar for arithmetic expression from Figure 5.4, and let us check that it is a *strong* LL(1) grammar. To this end, we can rely on the computation of the First and Follow sets from Example 5.8 and Example 5.9. To apply Definition 5.12, we need to consider all the group of rules that have the same left-hand side. There are three such groups:

1. The three rules that have Exp' as left-hand side are: $\mathsf{Exp}' \to +\mathsf{Prod}\mathsf{Exp}'$, $\mathsf{Exp}' \to -\mathsf{Prod}\mathsf{Exp}'$, and $\mathsf{Exp}' \to \varepsilon$. Hence, we check that there is no common element between the three following sets:

$$\begin{aligned} \mathrm{First}\big(+\mathsf{Prod}\mathsf{Exp}'\mathrm{Follow}(\mathsf{Exp}')\big) &= \{+\} \\ \mathrm{First}\big(-\mathsf{Prod}\mathsf{Exp}'\mathrm{Follow}(\mathsf{Exp}')\big) &= \{-\} \\ \mathrm{First}\big(\varepsilon\mathrm{Follow}(\mathsf{Exp}')\big) &= \mathrm{Follow}(\mathsf{Exp}') \\ &= \{\$,)\}. \end{aligned}$$

   This is indeed the case. Intuitively, this means that, when Exp' is on the top of the parser's stack, it can determine which rule to apply basing its decision on a single character look-ahead: when the look-ahed is $+$, apply the first rule; when the look-ahead is $-$, apply the second; and apply the last only when the look-ahead is \$ or).

2. For the three rules that have Prod' as the left-hand side, we check that

there is no common element between the three following sets:

$$\begin{aligned}
\text{First}\big(*\mathsf{AtomProd}'\text{Follow}(\mathsf{Prod}')\big) &= \{*\}\\
\text{First}\big(/\mathsf{AtomProd}'\text{Follow}(\mathsf{Prod}')\big) &= \{/\}\\
\text{First}\big(\varepsilon\text{Follow}(\mathsf{Prod}')\big) &= \text{Follow}(\mathsf{Prod}')\\
&= \{\$,+,-,)\}.
\end{aligned}$$

3. Finally, for the four rules that have Atom as the left-hand side, we consider the four sets:

$$\begin{aligned}
\text{First}(-\mathsf{Atom}\text{Follow}(\mathsf{Atom})) &= \{-\}\\
\text{First}(\mathsf{Cst}\text{Follow}(\mathsf{Atom})) &= \{\mathsf{Cst}\}\\
\text{First}(\mathsf{Id}\text{Follow}(\mathsf{Atom})) &= \{\mathsf{Id}\}\\
\text{First}((\mathsf{Exp})\text{Follow}(\mathsf{Atom})) &= \{(\},
\end{aligned}$$

which have no element in common. We conclude that the grammar is indeed *strong* LL(1).

The name *strong* LL(k) suggests that the conditions of Definition 5.12 are stronger than those of Definition 5.10. This is indeed the case: all strong LL(k) grammars *are* LL(k) grammars; however, the converse is, in general, not true, as shown by the next example:

Example 5.14. Let us consider again the grammar given in Example 5.11, which is LL(2), and let us show that it is *not* strong LL(2). Indeed, if we consider the two rules $A \to \mathtt{b}$ and $A \to \varepsilon$, we have:

$$\begin{aligned}
\text{First}^2\big(\text{bFollow}^2(A)\big) &= \text{First}^2\big(\{\mathtt{ba},\mathtt{bba},\mathtt{bca}\}\big)\\
&= \{\mathtt{ba},\mathtt{bb},\mathtt{bc}\}
\end{aligned}$$

and:

$$\begin{aligned}
\text{First}^2\big(\varepsilon\text{Follow}^2(A)\big) &= \text{First}^2\big(\{\mathtt{a},\mathtt{ba},\mathtt{ca}\}\big)\\
&= \{\mathtt{a},\mathtt{ba},\mathtt{ca}\}.
\end{aligned}$$

Since these two sets both contain ba, the grammar is *not* strong LL(2).

Nevertheless, in the case where the look-ahead is only one character, it turns out that *strong* LL(1) grammars are not more restrictive than LL(1) grammars. All these results are summarised in the following theorem:

Theorem 5.4.

1. *For all $k \geq 1$, for all* CFG *G: if G is strong* LL(k)*, then it is also* LL(k).
2. *For all $k \geq 2$, there is a* CFG *G which is* LL(k) *but not strong* LL(k).
3. *However, all* LL(1) *grammars are also strong* LL(1)*, i.e. the classes of* LL(1) *and strong* LL(1) *grammars coincide.*

Proof. (Sketch) Points 1. and 3. can be derived from Definition 5.10 and Definition 5.12. Point 2 stems from Example 5.14 that can be generalised to any $k \geq 2$. □

5.4.2 The top-down hierarchy of grammars

Now that we have identified an infinite sequence LL(0), LL(1),...,LL(k),... of families of grammars, one can wonder what are the relationships between them. Obviously LL(k) $\subseteq$ LL($k+1$) for all k. Indeed, if a parser is deterministic with k characters of look-ahead, it will still be deterministic with an extra character of look-ahead. We also know that the grammar from Example 5.11, is LL(2) but not LL(1), so LL(1) $\subsetneq$ LL(2). Is it true in general ? The answer is 'yes', as shown by the following example.

Example 5.15. This example has been proposed by Kurki-Suonio in 1969[8]. We only present here the grammars that are of interest to us, but do not present the formal proof, which is quite involved. For all $k \geq 1$, let us consider the grammar G_k as in Figure 5.6 (where the third rule is parameterised by k). Then, we claim that G_k is an LL($k+1$) grammar but not LL(k).

[8] R. Kurki-Suonio. Notes on top-down languages. *BIT Numerical Mathematics*, 9(3): 225–238, 1969. ISSN 1572-9125. DOI: 10.1007/BF01946814

(1)	S	$\rightarrow$	$\mathtt{a}SA$
(2)		$\rightarrow$	ε
(3)	A	$\rightarrow$	$\mathtt{a}^k\mathtt{b}S$
(4)		$\rightarrow$	$\mathtt{c}$

Figure 5.6: The family of grammars G_k. Each G_k is in LL($k+1$) \ LL(k).

Although we refer the reader to the cited article for the full proof, we can observe that, in G_k:

1. $\text{First}^{k+1}(A) = \{\mathtt{a}^k\mathtt{b}, \mathtt{c}\}$;
2. hence $\text{Follow}^{k+1}(S) = \{\mathtt{a}^k\mathtt{b}, \mathtt{c}\}$ as well, by the first rule of the grammar.
3. However, $\text{First}^{k+1}(S) = \{\varepsilon, \mathtt{a}^{k+1}\}$. Indeed, the recursion in the first rule of the grammar will produce an arbitrarily long prefix of a's, containing at least one a, which will be followed by k more $\mathtt{a}'s$ produced by A.

So, we can already conclude that G_k is *strong* LL($k+1$), hence, it is also LL($k+1$). Using similar arguments, we can show that G_k is not *strong* LL(k). Unfortunately, this does not imply that G_k is not LL(k), and this needs to be proved with other arguments (is the most involved part of the proof from the original article).

So we conclude that:

Theorem 5.5. *The family of* LL(k) grammars *(for all $k \geq 0$) forms a strict hierarchy:*

$$\text{LL}(0) \subsetneq \text{LL}(1) \subsetneq \text{LL}(2) \subsetneq \cdots \subsetneq \text{LL}(k) \subsetneq \text{LL}(k+1) \subsetneq \cdots$$

We call this infinite hierarchy of classes of grammars *the top-down hierarchy (of grammars).*

5.4.3 The top-down hierarchy of languages

Observe that the definitions we have given so far (LL(k), strong LL(k)) are *concerned by grammars*[9], but do not speak explicitly about the *languages* those grammars define. We have already seen, in Example 5.11 that there is at least one grammar which is LL(2) but not LL(1) (hence, not strong LL(1)). However, we also know that there are potentially several different grammars to define the same language. So, instead of considering classes of LL(k) grammars, one could naturally define LL(k) languages:

[9] Actually, the definition of strong LL(k) is a purely syntactical condition on grammars.

Definition 5.16 (LL(k) language)**.** A language L is LL(k) iff there is an LL(k) grammar G_L that accepts it, i.e.: $L(G_L) = L$.

Now, we can compare classes of LL(k) languages. Obviously, all LL(k) languages are also LL($k+1$) for all k. Is the converse true? Let us consider again the grammar from Example 5.11, which is LL(2) but not LL(1), and let us check whether there is *another grammar* that generates the same language. For this grammar, the answer is trivially 'yes' since the language generated by the grammar generates the finite language:

The intuition is always the same: if a parser can recognise a language with k characters of look-ahead, it can also do so with $k+1$ characters of look-ahead.

$$\{\texttt{aba}, \texttt{aa}, \texttt{bbba}, \texttt{bbca}, \texttt{bba}, \texttt{bca}\}.$$

So, an equivalent grammar (which is not yet LL(1)) is:

(1)	S	$\rightarrow$	aba
(2)		$\rightarrow$	aa
(3)		$\rightarrow$	bbba
(4)		$\rightarrow$	bbca
(5)		$\rightarrow$	bba
(6)		$\rightarrow$	bca

We can now use the factoring techniques from Section 4.4, and obtain:

(1)	S	$\rightarrow$	aA
(2)	S	$\rightarrow$	bB
(3)	A	$\rightarrow$	ba
(4)		$\rightarrow$	a
(5)	B	$\rightarrow$	bB'
(6)		$\rightarrow$	ca
(7)	B'	$\rightarrow$	ba
(8)		$\rightarrow$	ca
(9)		$\rightarrow$	a

which one can check is indeed (strong) LL(1), using Definition 5.12.

So, we have been able to turn our 'non-LL(1)' grammar into an equivalent LL(1). Is it always the case? As a matter of fact, it is not. The proof of this statement can be found again in the paper of Kurki-Suonio[10], where they prove that the grammar G_k from Figure 5.6 *generates an* LL(k) *language that is* ***not*** LL($k-1$). Hence:

Observe that this statement is stronger than saying that G_k is LL(k) and not LL($k-1$). The latter statement does not guarantee that there is no LL($k-1$) grammar that generates $L(G_k)$, and this requires a special proof which is in the cited paper. Once again, one should pay attention to the difference between syntax (grammars) and semantics (languages of grammars).

Theorem 5.6. *The families of* LL(k) languages *(for all $k \geq 0$) form a strict hierarchy:*

$$\text{LL}(0)\ \textit{lang.} \subsetneq \text{LL}(1)\ \textit{lang.} \subsetneq \text{LL}(2)\ \textit{lang.} \subsetneq \cdots \subsetneq \text{LL}(k)\ \textit{lang.} \subsetneq \text{LL}(K+1)\ \textit{lang.} \subsetneq \cdots$$

Relationship with DCFL Finally, since the point of considering LL(k) languages is to obtain *deterministic parsers*, one can wonder of LL(k) languages compare to DCFL? Clearly, we have:

$$\text{For all } k \geq 0 : \text{LL}(k) \text{ lang.} \subsetneq \text{DCFL}$$

Indeed, each LL(k) language is recognised by an LL(k) parser which is a *deterministic* PDA, so all those languages are deterministic CFL. The containment needs to be strict since LL(k) lang. $\subsetneq$ LL($K+1$) lang. for all k.

One can actually prove a further result: even the (infinite) union of all LL(k) lang. is still not sufficient to cover all DCFL. This can be proved by considering the language that is obtained by the union of the regular language $\{\texttt{a}^n \mid n \geq 0\}$ and the CFL $\{\texttt{a}^n\texttt{b}^n \mid n \geq 0\}$. Indeed, we can show that:

[10] R. Kurki-Suonio. Notes on top-down languages. *BIT Numerical Mathematics*, 9(3): 225–238, 1969. ISSN 1572-9125. DOI: 10.1007/BF01946814

Lemma 5.7. $L = \{\mathtt{a}^n \mid n \geq 0\} \cup \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$ *is a* DCFL *that is not* LL(k) *for all* $k \geq 0$.

(Sketch). One can easily build a DPDA that accepts L by accepting state. This DPDA pushes all the a it reads on the stack. This will be done by a self-loop on an accepting state, so all the words of the form $\mathtt{a}^n$ are accepted. If a b is read from this state, the DPDA moves deterministically to another state where it will read all the b's and check that there are as many b's as a's by emptying the stack. When the stack becomes empty, the DPDA moves to an accepting state. So, in this last state, all words of the form $\mathtt{a}^n\mathtt{b}^n$ will be accepted.

However, L cannot be LL(k) for any k. Assume it is the case for some value k. Then, consider the two words $\mathtt{a}^k$ and $\mathtt{a}^k\mathtt{b}^k$. We can derive a contradiction from the definition of LL(k) grammars. In its initial state, our hypothetical parser will perform the same action, since the look-ahead $\mathtt{a}^k$ is the same. However, it is clear that there must be two different derivations: in the first case, only a's must be generated from the sentential form that is being built; while in the second case, some symbols must occur in order to show to generate some amount of b's (as many as there are a's). □

So, we can conclude that:

$$\text{LL(0) lang.} \subsetneq \text{LL(1) lang.} \subsetneq \cdots \subsetneq \text{LL}(k) \text{ lang.} \subsetneq \cdots \subsetneq \mathsf{DCFL},$$

and that:

$$\bigcup_{k \geq 0} \text{LL}(k) \text{ lang.} \subsetneq \mathsf{DCFL}.$$

5.5 LL(1) *parsers*

Equipped with this general theory, we are now ready to discuss the construction of *deterministic top-down parsers* for a large and practical class of grammars, namely the LL(1) grammars. Those parser will thus be called LL(1) *parsers*.

5.5.1 *Obtaining an* LL(1) *grammar*

As we have seen before, not all grammars are LL(1), and some languages cannot be defined by an LL(1) grammar. However, for practical matters, when one wants to generate a parser for a typical programming language, obtaining an LL(1) grammar for that language is feasible. Here are the typical obstacles to the LL(1) that can easily be alleviated with the techniques we have seen so far:

Ambiguity First of all, if a grammar is LL(k) for some k, then it is necessarily unambiguous[11]. Thus, to obtain an LL(1) grammar, one must first make sure that it is unambiguous. Consider for example the grammar for arithmetic expression from Figure 4.19. As we have already argued, this grammar *is* ambiguous. However, setting the priority and associativity of operators with the techniques from Section 4.4 yields an unambiguous and equivalent grammar.

[11] D.J. Rosenkrantz and R.E. Stearns. Properties of deterministic top-down grammars. *Information and Computation (formerly known as Information and Control)*, 17(3):226 – 256, 1970. ISSN 0019-9958. DOI: 10.1016/S0019-9958(70)90446-8

Left recursion It is easy to check that no grammar that contains a left-recursive rule can be LL(1). Consider a grammar of the form:

(1)	S	$\rightarrow$	$S\alpha$
(2)		$\rightarrow$	β

where β is a string of terminals. Then, this grammar is obviously *not* LL(1), since the parser cannot decide which rule to apply when S on the top of the stack, and $\mathrm{First}(\beta)$ is seen on the input, i.e.:

$$\mathrm{First}(\beta) \in \mathrm{First}(S) \subseteq \mathrm{First}(S\alpha).$$

However, we have seen in Section 4.4 a technique to turn left-recursion into right-recursion. On the above example, we obtain:

(1)	S	$\rightarrow$	$\beta S'$
(2)	S'	$\rightarrow$	$\alpha S'$
(3)		$\rightarrow$	ε

which is now LL(1).

Common prefixes Another source of trouble is when two rules share the same left-hand side, and a common prefix on their right-hand side, such as in:

(1)	[if]	$\rightarrow$	`if` [Cond] `then` [Code] `fi`
(2)	[if]	$\rightarrow$	`if` [Cond] `then` [Code] `else` [Code] `fi`
			...

Here, if the parser sees variable [if] on the top of the stack, and symbol `if` on the input, it cannot decide which rule to apply, so the grammar is not LL(1). However, factoring (see Section 4.4) solves this issue:

(1)	[if]	$\rightarrow$	`if` [Cond] `then` [Code] [ifSeq]
(2)	[ifSeq]	$\rightarrow$	`fi`
(3)	[ifSeq]	$\rightarrow$	`else` [Code] `fi`
			...

Now, let us assume that we have a proper LL(1) grammar to describe the language we are interested to parse, and let us describe the construction of its associated LL(1) parser.

5.5.2 Action table

The core of the construction will be the building of the so-called '*action table*', which describes what actions the parser must perform (either produce or match), depending on the look-ahead and the top of the stack. We have already sketched an example of such a table at the beginning of Section 5.3. This table describes completely the behaviour of the parser, so, from now on, we will describe a parser with look-ahead by this means only, hiding the fact that the parser is actually a PDA[12]. Here is a more formal definition of the action table:

[12] Actually, a PDA with a single state, which is thus irrelevant. Also, we will hide the fact that there is always a transition that can pop the Z_0 symbol to reach an accepting configuration.

Definition 5.17 (LL(1) action table)**.** Let $G = \langle P, T, V, S\rangle$ be a CFG. Let us assume that:

1. G's rules (elements of P) are indexed from 1 to n; and

2. P contains a rule of the form $S \rightarrow \alpha\$$, where $\$ \in T$ is a terminal that does not occur elsewhere in the grammar (it is an end-of-string marker) and this rule is the only one that has S on the left-hand side.

Then, the LL(1)-*action table* of G is a two-dimensional table M s.t.:

- the lines of M are indexed by elements from $T \cup V$ (the potential tops of stack);
- the rows of M are indexed by elements from T (the potential look-aheads); and
- each cell $M[\alpha, \ell]$ contains a *set of actions* that the parser must perform in configurations where α is the symbol on the top of the stack, and ℓ is the next terminal on the input. These actions can be either:
  - an integer i s.t. $1 \leq i \leq n$, denoting that a produce of rule number i must be performed (i.e., if rule number i is $\alpha \rightarrow \beta$, then pop α from the stack and push β); or
  - Accept, denoting that the string read so far is accepted. This occurs only in cell $M[\$, \$]$, i.e., when $\$$ is on the top of the stack and also the next symbol on the input. In terms of PDA, this consists in reading $\$$, and popping it, to reach an accepting configuration (provided that no characters are left on the input); or
  - Match, denoting that a match action must be performed. This action occurs only in the cases where $\alpha = \ell \in T$. Then, the action consists in popping α and reading α from the input.
  - Error, denoting the fact that the parser has discovered an error and cannot go on with the construction of a derivation. The input should be rejected.

In this definition, we state that each cell of the table can potentially contain *several actions*. Of course, if the grammar is LL(1), then, each cell should contain only one action and the parser will be deterministic.

Before explaining how to build such a table in a systematic way, we present a complete example of such a table, and the execution of the parser on example input strings.

Example 5.18. Let us consider once again the grammar for artihmetic expressions (Figure 5.4), which we reproduce in Figure 5.7 to enhance readability. Its action table is as follows (where M, A and empty cells denote ‘Match’, ‘Accept’ and ‘Error’, respectively):

(1)	S	$\rightarrow$	Exp\$
(2)	Exp	$\rightarrow$	Prod Exp$'$
(3)	Exp$'$	$\rightarrow$	$+$Prod Exp$'$
(4)		$\rightarrow$	$-$Prod Exp$'$
(5)		$\rightarrow$	ε
(6)	Prod	$\rightarrow$	Atom Prod$'$
(7)	Prod$'$	$\rightarrow$	$*$Atom Prod$'$
(8)		$\rightarrow$	/Atom Prod$'$
(9)		$\rightarrow$	ε
(10)	Atom	$\rightarrow$	$-$Atom
(11)		$\rightarrow$	Cst
(12)		$\rightarrow$	Id
(13)		$\rightarrow$	(Exp)

Figure 5.7: The grammar generating expressions (followed by \$ as an end-of-string marker). This is the same grammar as in Figure 5.4, reproduced here for readability.

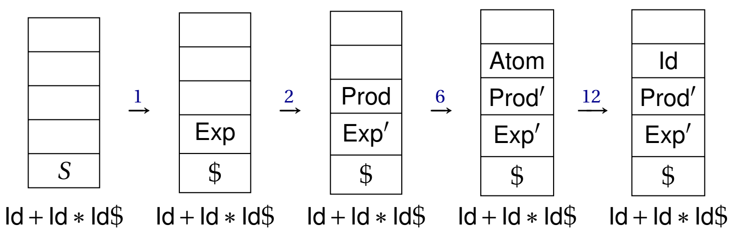
M	\$	+	−	$*$	/	Cst	Id	(	)
S			1			1	1	1	
Exp			2			2	2	2	
Exp′	5	3	4						5
Prod			6			6	6	6	
Prod′	9	9	9	7	8				9
Atom			10			11	12	13	
\$	A								
+		M							
−			M						
$*$				M					
/					M				
Cst						M			
Id							M		
(								M	
)									M

Note that the bottom half of the table is not very informative: it just tells us that we should match terminals when they occur at the top of the stack. This is not surprising: non-determinism can occur only because of the 'Produce' actions. So, in the rest of these notes, we will not show that part of the table anymore.

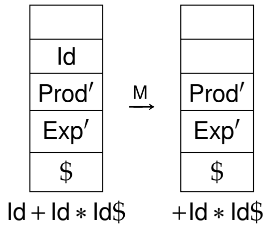
Now, let us consider the input word Id + Id $*$ Id which is accepted by the grammar, and let us build the corresponding run.

1. Initially, we are in a configuration where the stack contains only S and the input contains only Id+Id$*$Id. Thus, we look at $M[S, \text{Id}]$, and see that rule (1) must be produced. This is not a surprise, since rule (1) is the only one that has S as the left-hand side; but at least $M[S, \text{Id}]$ does not raise an error. Had the first character of the input been / for instance, we would have been able to reject the input string immediately. The Produce replaces S by Exp\$ on the the stack but does not modify the input.

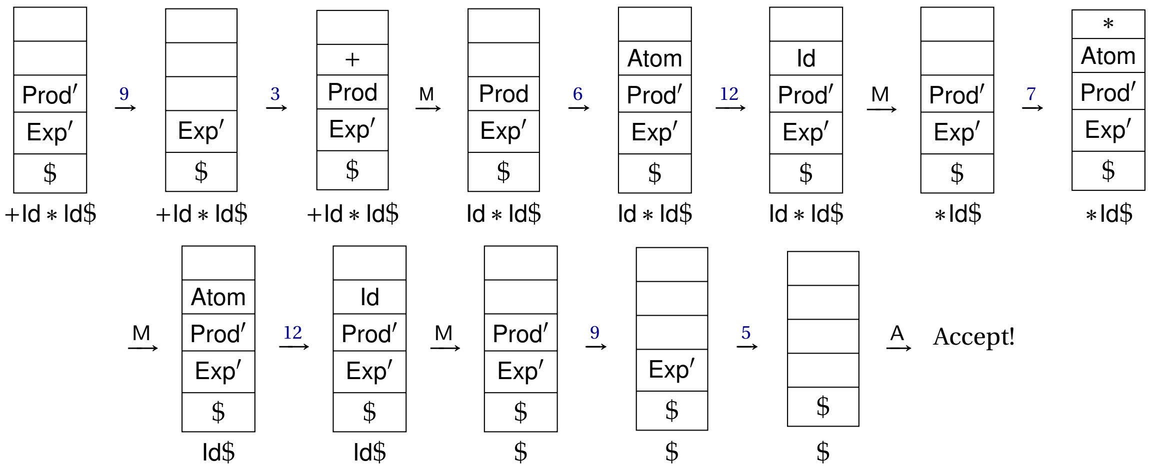
2. Then, we look up $M[\text{Exp}, \text{Id}]$, and Produce rule (2). This replaces Exp by ProdExp′ on the stack (with Prod on top, thus) and does not modify the input. The parsing continues accordingly, for a couple of steps (the remaining inputs are drawn below the stacks):



3. At that point, the terminal Id is present both on the top of the stack and on the input, so a Match occurs, which modifies the input:



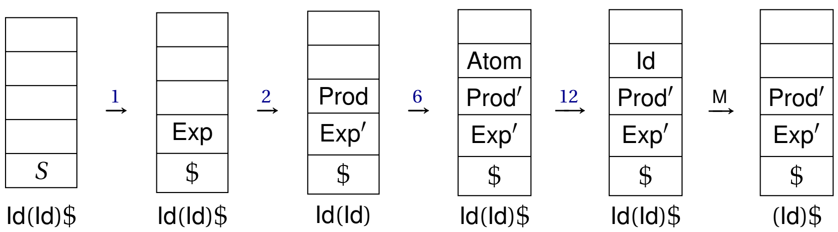
4. Then, the parsing goes on… Observe that the next produce consists in popping the Prod′ variable from the top of the stack (i.e., applying rule Prod′ → ε):



So, the word Id + Id ∗ Id$ is accepted.

Now, let us consider the word Id(Id)$ which is not syntactically correct (it is not accepted by the grammar).

1. The corresponding run starts as in the case of Id + Id ∗ Id$, until the moment where the Id symbol on the top of the stack has to be matched:



2. At this point, the action table returns an error (i.e., M[Prod′, (] = Error, indicated by a blank cell in the table above), and the parsing stops

So the word Id(Id)$ is not accepted by the parser and by the grammar.

Observe that when the error is detected, the information in row Prod′ of the action table could be used to give some feedback to the user, by telling him which symbol could have been correct at that point of the parsing (without guarantee that the parsing could have continued even if that symbol were present). For example, an error message in the case could have been:

Error, unexpected symbol (. I was expecting $, +, −, ∗, / or).

One could also imagine that the compiler skips the error at that point and tries and *re-synchronise*, i.e., tries and compile the remainder of the input (in this case (Id)$, which is correct) in order to inform the user of potential further errors in the input. Error reporting and re-synchronisation are beyond the scope of these notes. The interested reader can refer to the 'Dragon book' .

A. Aho, M. Lam, R. Sethi, and Ullman J. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007

Algorithm to build the action table Let us now formalise these ideas. Algorithm 8 presents the construction of the LL(1) table. The algorithm starts by initialising all the cells $M[A, a]$ (where A is a variable and a a terminal) to the empty set. These are all the cells that can potentially contain one or several '*Produce*' actions. Then all cells $M[a, b]$, where a and b are terminals, are populated: they are all initialised to the empty set, except for the cells $M[a, a]$ (with $a \neq \$$) that are initialised to a '*Match*', and $M[\$, \$]$ that contains the '*Accept*' action.

After this initialisation, all rules $A \to \alpha$ are taken into account: all symbols a from First(αFollow(A)) are computed, and the number i of the rule $A \to \alpha$ is *added* to the corresponding cell $M[A, a]$. Since the algorithm adds the rule number to the cell, each cell can contain several rule numbers, thereby allowing to detect potential conflicts.

It is useful to compare the way the algorithm fills in the table with the definition of *strong* LL(1) grammars. One can then see that there will be no conflict in the table built by the algorithm iff the grammar is strong LL(1), i.e. LL(1) since these two notions are equivalent.

Input: A CFG $G = \langle P, T, V, S \rangle$.
Output: The LL(1) action M table of G

```
/* Initialisation of the table                                */
foreach a ∈ T do
    foreach A ∈ V do
        M[A, a] ← ∅ ;
    foreach b ∈ T \ {a} do
        M[b, a] ← ∅ ;
    M[a, a] ← {M} ;
M[$, $] ← {A} ;
/* Adding the 'Produce' actions                               */
foreach rule A → α ∈ P with number i do
    foreach a ∈ First(α · Follow(A)) do
        M[A, a] ← M[A, a] ∪ {i} ;
return M ;
```

Algorithm 8: Systematic construction of the LL(1) table of a CFG.

5.5.3 *Algorithm of the parser*

Next, we formalise, by means of an algorithm, the execution of the parser, using the action table computed before. We use the syntax of pseudo-code that we have used to describe our algorithms so far instead of the syntax of PDA, because it is more readable, and closer to implementation. Note, however, that we propose, in the next section, another way of implementation such parsers, which is probably more practical.

The algorithm of the parser behaves as illustrated in Example 5.18: it starts by pushing the start symbol of the grammar on the stack, then, at all times, it reads the symbol x on the top of the stack (it can be a variable or a terminal), checks the look-ahead a, and applies the action which is given by the action table in cell $M[x, a]$. It proceeds this way until the '*Accept*' action is executed, or until an error is encountered (which is the case when the cell is empty).

Input: An LL(1) CFG $G = \langle P, T, V, S\rangle$ with its action table M and an input word $w = w_1 w_2 \cdots w_n \in T^*$.

Output: *True* iff $w \in L(G)$. In this case, the sequence of rule numbers in a left-most derivation is printed on the output.

```
/* Position of the 'reading head' in the input word:
   w_j is the look-ahead.                              */
j ← 1 ;
/* Pushing the start symbol.                           */
Push(S) ;
while the stack is not empty do
    x ← Top() ;
    if M[x, w_j] = {i} then
        Assume rule number i is A → α ;       /* Produce i */
        Pop() ;
        Push(α) ;
        Print(i) ;
    else if M[x, w_j] = {M} then
        Pop() ;                                 /* Match */
        j ← j + 1 ;
    else if M[x, w_j] = {A} then
        return True ;                          /* Accept */
    else
        return False ;                          /* Error */
```

5.6 Implementation using recursive descent

To close the section on top-down parsers, let us describe a straightforward way to implement those parsers using recursive functions. To do this, we will build on the intuition we have already given in the introduction of Section 4: consider a rule like $S \to 0S0$, for instance. Such a rule can be interpreted by saying that 'to recognise an S, one should first read a 0 from the input, then recognise an S, then read another 0'. In terms of functions, this could mean that: 'to return *true* the S function should first read a 0 on the input, then make a succesful call (one that returns *true*) to S, then read a 0 from the input'.

While this intuition seems to hold for a *single* rule, and seems to provide an easy way to turn a CFG into a recursive code that implements a parser, it fails when there are at least two rules with the same left-hand side, for instance:

$$S \to A\mathtt{b}$$

and:

$$S \to B\mathtt{c},$$

because this would yield two (or more) different implementations for the same function corresponding to S.

However, in order to resolve this non-determinism, we can rely on the LL(1) techniques we have presented throughout this chapter. In the example above, let us assume that:

$$\begin{aligned}
\text{First}^1\big(A\mathtt{b}\text{Follow}^1(S)\big) &= \{\mathtt{a1}, \mathtt{a2}, \ldots \mathtt{an}\},\\
\text{First}^1\big(B\mathtt{c}\text{Follow}^1(S)\big) &= \{\mathtt{b1}, \mathtt{b2}, \ldots \mathtt{bk}\}.
\end{aligned}$$

Then, we would obtain the code (using the python syntax):

```
def S():
   n = get_next_character() # This is a look-ahead

   # If the look-ahead is in First(A b Follow(S))
   if n == 'a1' or n == 'a2' or ... or n == 'an' :
      read_next_character() # Discard the look-ahead
      r = A()
      if (!r): return False
      n = read_next_character()
      if (n == 'b'): return True
      else: return False

   # If the look-ahead is in First(B c Follow(S))
   if n == 'b1' or n == 'b2' or ... or n == 'bn' :
      read_next_character() # Discard the look-ahead
      r = B()
      if (!r): return False
      n = read_next_character()
      if (n == 'c'): return True
```

```
        else: return False

    # Otherwise, the input cannot be valid.
    return False
```

where we rely on two auxiliary functions:

- `get_next_character()` that returns the next character on the input but does not *consume* it from the input (i.e., several subsequent calls to `get_next_character()` always return the same value); and
- `read_next_character()` that returns the next character on the input and also *consume* it (the read pointer moves in the input stream).

5.7 Exercises

5.7.1 First and Follow sets

(1)	<S>	→	<program>
(2)	<program>	→	`begin` <statement list>
(3)	<statement list>	→	<statement> <statement tail>
(4)	<statement tail>	→	<statement> <statement tail>
(5)	<statement tail>	→	ε
(6)	<statement>	→	`ID :=` <expression> `;`
(7)	<statement>	→	`read (` <id list> `) ;`
(8)	<statement>	→	`write (` <expr list> `) ;`
(9)	<id list>	→	`ID` <id tail>
(10)	<id tail>	→	`, ID` <id tail>
(11)	<id tail>	→	ε
(12)	<expr list>	→	<expression> <expr tail>
(13)	<expr tail>	→	`,` <expression> <expr tail>
(14)	<expr tail>	→	ε
(15)	<expression>	→	<primary> <primary tail>
(16)	<primary tail>	→	<add op> <primary> <primary tail>
(17)	<primary tail>	→	ε
(18)	<primary>	→	(<expression>)
(19)	<primary>	→	`ID`
(20)	<primary>	→	`INTLIT`
(21)	<add op>	→	+
(22)	<add op>	→	−

Exercise 5.1. We consider the grammar given above.

1. Give the values of $\text{First}^1(A)$ and the $\text{Follow}^1(A)$ sets for all variables A of the grammar.
2. Give the values of $\text{First}^2(\text{<expression>})$ and $\text{Follow}^2(\text{<expression>})$.

5.7.2 LL(k) grammars

Exercise 5.2. Consider the four grammars in Figure 5.8. Which of those grammars are LL(1)? Justify your answers.

Exercise 5.3. Give the LL(1) action table for the following grammar:

(1)	<S>	→	<expr>$
(2)	<expr>	→	− <expr>
(3)	<expr>	→	(<expr>)
(4)	<expr>	→	<var> <expr-tail>
(5)	<expr-tail>	→	− <expr>
(6)	<expr-tail>	→	ε
(7)	<var>	→	`ID` <var-tail>
(8)	<var-tail>	→	(<expr>)
(9)	<var-tail>	→	ε

(1)	S	→	$ABBA$
(2)	A	→	`a`
(3)		→	ε
(4)	B	→	`b`
(5)		→	ε

(1)	S	→	`a`S`e`
(2)		→	B
(3)	B	→	`b`B`e`
(4)		→	C
(5)	C	→	`c`C`e`
(6)		→	d

(1)	S	→	AB`c`
(2)	A	→	`a`
(3)		→	ε
(4)	B	→	`b`
(5)		→	ε

(1)	S	→	A`b`
(2)	A	→	`a`
(3)		→	B
(4)		→	ε
(5)	B	→	`b`
(6)		→	ε

Figure 5.8: Which grammars are LL(1)?

6 Bottom-up parsers

IN THIS SECTION, WE WILL CONSIDER A COMPLETELY DIFFERENT FAMILY OF PARSERS, WHICH ARE CALLED BOTTOM-UP PARSERS. As sketched already in the introduction of the previous chapter, bottom-up parsers build a derivation tree starting from the leaves (i.e., the input string), and follow *backwards* the derivation rules of the grammar, until they manage to reach the root of the tree. Those parsers are generally regarded as *more powerful* than their top-down counterparts (we will give formal elements to support this claim in Section 6.9). As such, automatic parser generators such as `yacc`[1], `bison`[2] and `cup`[3] implement bottom-up parsers.

[1] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, AT&T Bell Laboratories, 1975. Readable online at `http://dinosaur.compilertools.net/yacc/`

[2] Gnu bison. `https://www.gnu.org/software/bison/`. Online: accessed on December, 29th, 2015

[3] Cup: Construction of useful parsers. `http://www2.cs.tum.edu/projects/cup/`. Online: accessed on December, 29th, 2015

6.1 Principle of bottom-up parsing

Recall the two main actions that a top-down parser can perform:

1. the *Produce*, which consists in replacing, on the top of the stack, the left-hand side A by the right-hand side α of some production rule $A \rightarrow \alpha$ of the grammar; and

2. the *Match*, that consists in reading from the input some character `a`, which is at the same time popped from the top of the stack.

Such top-down parsers start their execution with the start symbol S on the stack and accept with the empty stack. Doing so, they unravel a parse tree for the input string from the top to the bottom, and produce a *leftmost* derivation.

Bottom-up parsers, on the other hand, work in a complete reverse way. As their name suggest, they build the parse tree from the bottom to the top. As such, they are often regarded as more efficient, since they deduce the nodes of the parse tree based on the actual input, whose elements they recognise as being generated by the grammar; contrary to top-down parsers that have to start from the start symbol and find a way to obtain the input by applying the proper grammar rules.

For these notes, we will consider the most prominent class of bottom-up parsers, which are those that are based on two main actions: the *Shift* and the *Reduce*, as we are about to explain. Those parsers (in particular the LR(k) parsers that we are about to study) have been introduced[4] by Donald E. KNUTH[5] in 1965, in a paper where he generalises previous works from other prominent computer scientists such as Robert FLOYD[6].

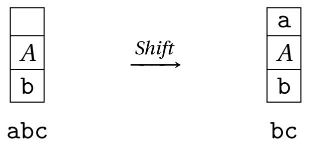
[4] Donald E. Knuth. On the translation of languages from left to right. *Information and Computation (formerly known as Information and Control)*, 8:607–639, 1965. DOI: 10.1016/S0019-9958(65)90426-2

[5] Prominent american computer scientist, born in 1938, former Stanford and CalTech professor, recipient of the TURING award in 1974. He is the author of the famous series of books *The Art of Computer Programming*, and of the TeX typesetting language, which has later been extend to LaTeX by Leslie LAMPORT.

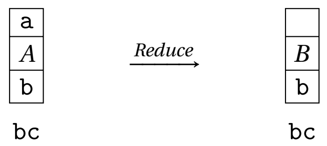
[6] Robert E. FLOYD, american computer scientist (1936–2001). Recipient of the TURING award in 1978, and well-known for several contributions to algorithmic, including the FLOYD-WARSHALL algorithm.

In order to build the parse tree from the leaves to the root, *Shift-Reduce* bottom-up parsers procede as follows:

- First, they rely on the *Shift* action to *move* terminals from the input to the top of the stack. The picture hereunder illustrates a shift, where the terminal a is read from the input and pushed on the top of the stack:



- Second, they apply the grammar rules *in reverse*. The parser looks for a so-called *handle* on the top of the stack, i.e. the right-hand side α of some grammar rule $A \to \alpha$. When such a handle is present (in mirror image) on the top of the stack, the parser can perform a *Reduce*. A Reduce amounts to popping the handle α, and pushing A instead. This way, the parser unravels a parse tree from the bottom to the top (hence the name *bottom-up parser*). The picture hereunder illustrates a Reduce of the rule $B \to A\mathtt{a}$:



Observe that the handle Aa appears with the rightmost character on the top of the stack. That is, the handle is *reversed* wrt to what would have been pushed to the stack by a Produce of the same rule in a top-down parser. This is because the characters that have produced the variable A (through other Reduces, presumably) have been read on the input *before* the a, so the A has been pushed to the stack before the a and is thus *under* the a in the stack.

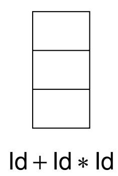
- Finally, the aim of the parser is not to *empty* the stack (by matching all the terminals), but rather to end up with only the start symbol S on the stack (and, of course, an empty input). This means that a derivation has been produced for the whole string, but in the *reverse order* (since rules have been applied in the reverse order too when doing the *Reduces*). Actually, the bottom-up parser builds a *right-most derivation in reverse order*.

Example 6.1. Let us consider once again the grammar for arithmetic expressions of Figure 4.4, and let us consider the string $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$, which is accepted by the grammar. One possible[7] *rightmost* derivation for this string is as follows:

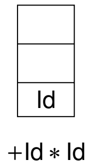
[7] Recall that this grammar is *ambiguous*, so several rightmost derivations are possible.

$$\underline{\mathsf{Exp}} \overset{2}{\Rightarrow} \mathsf{Exp} * \underline{\mathsf{Exp}} \overset{4}{\Rightarrow} \underline{\mathsf{Exp}} * \mathsf{Id} \overset{1}{\Rightarrow} \mathsf{Exp} + \underline{\mathsf{Exp}} * \mathsf{Id} \overset{4}{\Rightarrow} \underline{\mathsf{Exp}} + \mathsf{Id} * \mathsf{Id} \overset{4}{\Rightarrow} \mathsf{Id} + \mathsf{Id} * \mathsf{Id}. \tag{6.1}$$

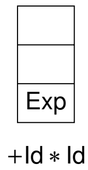
The parser starts its execution in a configuration where: the stack is empty (formally, it contains only the empty stack symbol Z_0, but we will not display it, or the sake of readability); and the input contains $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$:



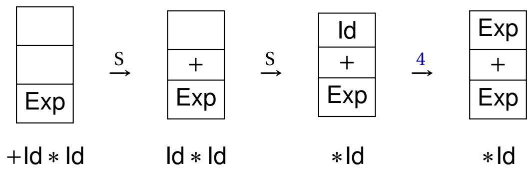
From this configuration, the parser can only *shift* the first character on the input:



Now, the top of the stack constitutes a *handle* for the rule Exp→Id . Remark that, at this point, the parser can decide either to *Reduce* Exp→Id or to *Shift* another character. The former choice is the good one. Indeed, if we shift a + on top of the Id, it means we will need to find a rule whose right-hand part contains Id *and* some other symbols, but there is no such rule in our grammar. The Reduce of Exp→Id yields:

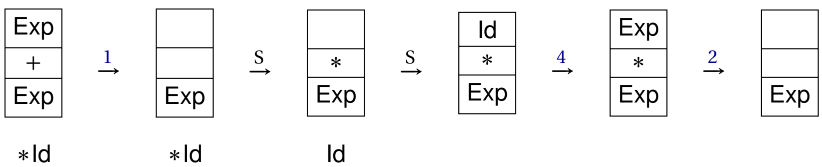


Observe that, in the rightmost derivation (6.1), the *last* rule is indeed Exp→Id . So, this is coherent with our claim that the parser builds the rightmost derivation in a *reverse* manner. Now, the parser can shift two more symbols, and reduce the Id that ends up on the top of the stack:



Next, following the rightmost derivation, Exp + Exp froms a handle of the rule Exp→Exp + Exp , which can be reduced[8]. Then, the parser can shift twice, and finally reduce Exp→Id , then Exp→Exp ∗ Exp :

[8] Formally, this must be performed, in the PDA, by three transitions in a row: one to pop the first Exp, one to pop the +, and one to replace the last Exp by another Exp. This last transition can be skipped however, since we are in the special case where the last symbol that must be popped is also the one than has to be pushed.



The last configuration is an *accepting configuration*[9] for this parser, because: (i) the start symbol Exp of the grammar is on the top of the stack; and (ii) the input is empty. Thus, the string Id + Id ∗ Id is accepted. One can check that the sequence of reductions performed by the parser corresponds to the mirror image sequence of rules that have been applied in the rightmost derivation.

[9] Formally, the PDA that corresponds to the parser should contain a transition that pops Exp from the stack and moves to an accepting state.

6.1.1 *Systematic construction of a bottom-up parser*

Following these intuitions, we can now explain how to systematically build a bottom-up parser from a given CFG. We are giving this construction for the sake of completeness. Actually, when we will make those parsers deterministic (as we did for top-down parsers using look-ahead), we will need to slightly alter the behaviour of the basic parser that we are presenting now. Nevertheless, we believe it is a good exercise to show that the actions we have described above can actually be implemented in a PDA.

The parser we are about to describe is unfortunately not as simple as the one-state top-down parser we had obtained in Section 5.1.1. This is due to the fact that a *Reduce* entails a sequence of *pops* from the stack, which cannot be performed by a single transition. Hence, we will need to introduce intermediary states. More precisely, for each rule of the form $A \rightarrow \alpha_1 \cdots \alpha_n$ (where the α_i's are individual variables or terminals), we will have n states, that we call $(A, \alpha_1 \cdots \alpha_j)$ for $1 \leq j \leq n-1$ and (A, ε). Intuitively, the PDA reaches $(A, \alpha_1 \cdots \alpha_j)$ iff: (i) it is in the middle of the reduction of $A \rightarrow \alpha_1 \cdots \alpha_n$; and (ii) it has already popped characters α_n, $\alpha_{n-1}, \ldots,$ α_{j+1} from the stack. Thus, to finish the reduction from this state, the PDA must still: (i) pop α_j, $\alpha_{j-1}, \ldots,$ α_1 (in this order); and (ii) push A. For example, if the grammar contains rule $A \rightarrow \mathtt{bc}$, then, the parser can, from its initial state:

1. take a transition that pops $\mathtt{c}$ and move to $(A, \mathtt{b})$; then,
2. from $(A, \mathtt{b})$, take a transition that pops $\mathtt{b}$ and move to (A, ε); and finally,
3. from (A, ε), take a transition that pushes A, and move back to the initial state of the PDA.

In addition to these transitions, we also need transitions to perform *Shifts*, and one transition to move to a dedicated state called q_a when the start symbol of the grammar occurs on the top of the stack. Note that this state is not *accepting* (we build again a PDA that accepts on empty stack), but its aim is to check that, at this point, the only symbol left on the stack is Z_0, which means that all the symbols on the stack have indeed been reduced to S.

Recall that a PDA with accepting states accepts only when the accepting state is reached *and* the input is empty. There is thus no problem in jumping non-deterministically to the accepting state whenever the start symbol occurs on the top of the stack.

Formally, from a CFG $G = \langle V, T, P, S \rangle$, we build a PDA P'_G as follows:

$$P'_G = \left\langle Q, T, V \cup T, \delta, q_i, Z_0, \emptyset \right\rangle,$$

where:

1. The set of states Q is defined as follows:

$$Q = \left\{q_i, q_a\right\} \cup \left\{(A, \varepsilon) \mid A \in V\right\}$$
$$\cup \left\{(A, \alpha_1 \cdots \alpha_j) \mid A \rightarrow \alpha_1 \cdots \alpha_n \in P \wedge 1 \leq j \leq n-1\right\}.$$

   That is, we have one initial state q_i, one accepting state q_a, and the intermediary states for the reductions, as announced;

2. For the transitions function, we start by describing it from the initial state q_i, then we consider the intermediary states:

   (a) For all $a \in T$, for all $s \in V \cup T \cup \{Z_0\}$:

$$\delta(q_i, a, s) = \{(q_i, as)\} \qquad \text{(Shift)},$$
$$\delta(q_i, \varepsilon, S) = \{(q_a, \varepsilon)\} \qquad \text{(Accept)},$$
$$\delta(q_i, \varepsilon, s) = \left\{\big((A, \alpha), \varepsilon\big) \;\middle|\; A \rightarrow \alpha s \in P\right\} \qquad \text{(Reduce)}.$$

Intuitively, there are self-loops on the initial state that *Shift* any symbol on the input to the stack; when the start symbol S occurs on the top of the stack, the parser can move to the accepting state (*Accept* action); and the parser can decide at any moment to start a *Reduce*, provided that the top of the stack is the right-most character in the handle.

(b) For all states of the form (A,α) with $\alpha \neq \varepsilon$:

$$\delta\big((A,\alpha s),\varepsilon,s\big) = \Big\{\big((A,\alpha),\varepsilon\big)\Big\}$$
$$\delta\big((A,\alpha),a,s\big) = \emptyset \qquad \text{in all other cases.}$$

(c) For all states of the form (A,ε), we have, for all $a \in T$, for all $s \in T \cup V \cup \{Z_0\}$:

$$\delta\big((A,\varepsilon),\varepsilon,s\big) = \big\{(q_i, As)\big\}$$
$$\delta\big((A,\varepsilon),a,s\big) = \emptyset.$$

(d) Finally, the only transition on q_a is a self-loop that removes Z_0, so that the PDA accepts only when all the symbols on the stack have been reduced to S:

$$\delta(q_a,\varepsilon,Z_0) = (q_a,\varepsilon)$$
$$\delta(q_a,a,s) = \emptyset \qquad \text{in all other cases.}$$

The correctness of this construction is given by the following Lemma:

Lemma 6.1. *For all* CFG*s* G*, the* PDA P'_G *is s.t.* $L(G) = N(P'_G)$.

Sketch. The proof is done in two steps. First, one shows that all words w accepted by the grammar G correspond to an accepting run of P'_G, by induction on the length of a (reversed) rightmost derivation producing w in G. Second, one shows that all accepting runs of P'_G on some word w can be translated to a rightmost derivation in G (by induction on the length of the run). □

6.1.2 *Non-determinism in the parser*

As in the case of top-down parsers, the main limitation to the bottom-up parser we have just defined is that it is *non-deterministic*. There are two sources of non-determinism in this parser:

Reduce-Reduce conflict: such conflicts occur when the top of the stack is the handle to two different rules, and the parser cannot decide which rule to reduce;

Shift-Reduce conflicts: such conflicts occur when the top of the stack constitutes a handle to some rule, but the parser cannot determine whether it should continue shifting or whether it should reduce now.

Let us first focus on *Shift-Reduce* conflicts. One (but not the only one) of the difficulties we need to overcome to get rid of such conflicts is to determine when *shifting new symbols might still produce a handle.* This has been illustrated in Example 6.1. After the first *Shift*, the configuration reached by the parser is as shown in Figure 6.1. In this configuration, the non-deterministic parser can either *Reduce* rule Exp→Id , or *Shift* the +. However, as we have already argued, shifting a + in this configuration will not yield an accepting run, as there is no other handle than the one of Exp→Id that contains an Id.

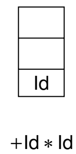


Figure 6.1: A configuration of the bottom-up parser where a Reduce must be performed.

6.1.3 *Viable prefixes*

This example shows that there is a fundamental difference between the two stack contents Id and Id+. This is captured by the notion of *viable prefix*. To define this notion, we must first observe a relationship between the stack contents during an accepting run of the bottom-up parser on input w and a rightmost derivation of w. Looking again at Example 6.1, one can check that **all stack contents are prefixes of some *sentential form* obtained along the right-most derivation (6.1)**. As a matter of fact, one can prove the more precise result which is given by Proposition 6.2 hereinafter.

Recall that a sentential form is a word over $T \cup V$ (i.e., a string containing terminal and variables) that occurs along some derivation of the grammar. When a sentential form has been extracted from a rightmost (leftmost) derivation, we call it a right (respectively left) sentential form.

Proposition 6.2. *Let $G = \langle V, T, P, S\rangle$ be a* CFG*, and let P'_G be its corresponding bottom-up parser. Let $\langle q_i, w, \gamma Z_0\rangle$ be a configuration reached along an accepting run of P'_G (i.e., a configuration where P'_G is neither in one of the intermediate states introduced for the Reduce, nor in q_a). Then:*

$$S \Rightarrow^* \gamma^R \cdot w$$

along a rightmost derivation.

Recall that w^R is the mirror image of w. In this case, γ^R is a word representing the content of the stack with the top on the right-hand side.

Sketch. The proof is by induction on the length of the run. Clearly, the property holds in the initial configuration, since in this case $\gamma = \varepsilon$, hence $\gamma^R w = w$, and w is an accepted word of the grammar (as the run is accepting).

Next, if the property holds on some configuration $\langle q_i, w_1, \gamma_1 Z_0\rangle$, then moving to the next configuration $\langle q_i, w_2, \gamma_2 Z_0\rangle$ where the parser is in q_i can be done either by a Shift or by a Reduce. In the case of a Shift, we have $\gamma_1^R \cdot w_1 = \gamma_2^R \cdot w_2$, because the first letter of w_1 has been transfered to the stack, so the property still holds. In the case of a Reduce, only the top of the stack is modified by the reduction of some handle α into some variable A (because the grammar contains the rule $A \to \alpha$). That is:

$$\begin{aligned} w_1 &= w_2 \\ \gamma_1^R &= \beta\alpha \\ \gamma_2^R &= \beta A \end{aligned}$$

for some stack content β. Since $w_1 = w_2$ is a word of terminals (they contain no grammar variable), A is indeed the rightmost variable in the sentential form $\gamma_2^R w_2$. Since $\gamma_1^R w_1$ can be derived from S (by induction hypothesis) we conclude that $\gamma_2^R w_2$ precedes $\gamma_1^R w_1$ in the rightmost derivation that yields $\gamma_1^R w_1$, hence, $S \Rightarrow^* \gamma_2^R \cdot w_2$. □

Example 6.2. To illustrate Proposition 6.2, we can check that it holds at all times along the run shown in Example 6.1. For example, when the configuration is $\langle q_i, \gamma, w\rangle = \langle q_i, \mathsf{Id} + \mathsf{Exp}, *\mathsf{Id}\rangle$, we have $\gamma^R = \mathsf{Exp} + \mathsf{Id}$, which is indeed a prefix of the sentential form $\mathsf{Exp}+\mathsf{Id}*\mathsf{Id}$ obtained after 4 derivations in (6.1). Observe that this sentential form is exactly $\gamma^R \cdot w$.

Thus, the content of the stack in a successful run of the bottom up parser should always be a prefix of a right-sentential form. Unfortunately, not all such prefixes allow us to build a succesful run. Continuing Example 6.1, $\mathsf{Id}+$ is a prefix of a right-sentential form in (6.1), but, as we have already argued, it does not allow us to build a successful run because the parser has shifted past the handle Id that should have been reduced to Exp.

This discussion brings us to a central concept of bottom-up parsers, that of *viable prefix*. A prefix is *viable* iff it can lead to a successful run. Formally:

In some references, *viable prefixes* are called *feasible prefixes*.

Definition 6.3 (Viable prefix)**.** A *viable prefix* of a CFG $G = \langle V, T, P, S\rangle$ is a prefix of a right-sentential form that can occur (in reverse) on the stack during an accepting run of the associated bottom-up parser P'_G.

That is, $p \in (V \cup T)^*$ is a viable prefix iff there is a word $w \in L(G)$ and an accepting run

$$(q_i, w, Z_0) = (q_1, w_1, \gamma_1 Z_0) \vdash_{P'_G} \cdots \vdash_{P'_G} (q_j, w_j, \gamma_j Z_0) \vdash_{P'_G} \cdots \vdash_{P'_G} (q_n, w_n, \gamma_n Z_0) = (q_a, \varepsilon, \varepsilon)$$

of P'_G s.t. $p = \gamma_j^R$ for some j s.t. $q_j = q_i$ (i.e. P'_G is in the initial state at step j and not in one of the intermediary states used for the Reduce).

Observe that the set of viable prefixes of a grammar can be infinite, and actually constitutes a language on $V \cup T$. If we consider once again the grammar of Example 6.1, we can see that all words of the form $\mathsf{Exp}+\mathsf{Exp}+\cdots+\mathsf{Exp}$ are viable prefixes of the grammar. Indeed, for every such word of the form:

$$\mathsf{Exp}\underbrace{+\mathsf{Exp}+\cdots+\mathsf{Exp}}_{n \text{ times}},$$

one can build a derivation obtained by applying n times the first grammar rule on the rightmost Exp of the sentential form:

$$\mathsf{Exp} \Rightarrow \mathsf{Exp}+\mathsf{Exp} \Rightarrow \cdots \Rightarrow \mathsf{Exp}\underbrace{+\mathsf{Exp}+\cdots+\mathsf{Exp}}_{n \text{ times}},$$

which can then yield the word:

$$\mathsf{Id}\underbrace{+\mathsf{Id}+\cdots+\mathsf{Id}}_{n \text{ times}}.$$

Then, an accepting run of the bottom-up parser consists in systematically shifting all the Id and $+$ tokens on the stack and reducing the Id to Exp as soon as they are shifted. Finally, the parser reduces the rule $\mathsf{Exp} \rightarrow \mathsf{Exp}+\mathsf{Exp}$ n times and accepts. Along this run all the prefixes Exp, $\mathsf{Exp}+\mathsf{Exp}$, $\mathsf{Exp}+\mathsf{Exp}+\mathsf{Exp}$ have been present on the stack.

From this discussion, it should be clear that identifying viable prefixes will be a central condition to build deterministic bottom-up parsers. The point of the next section is to introduce a tool to do so.

6.2 The canonical finite state machine

In this section, we explain how to build, from a CFG G, a finite automaton recognising exactly the viable prefixes of G. As explained, this automaton will be instrumental in building deterministic bottom-up parsers, and it is called the *canonical finite state machine*, or CFSM.

Not to be confused with the Church of the Flying Spaghetti Monster...

In order to introduce the construction of the CFSM, we will consider an example grammar.

Example 6.4. Consider the grammar in Figure 6.2. The set of viable prefixes of this grammar is:

$$\{\varepsilon, A, A\$, \mathtt{a}, \mathtt{a}C, \mathtt{ac}, \mathtt{a}CD, \mathtt{a}Cd, \mathtt{ab}\}.$$

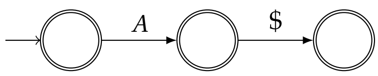
In particular, observe that $\mathtt{ac}D$ is *not* a viable prefix, because the parser has missed the handle $\mathtt{c}$ of $C \to \mathtt{c}$. Hence, we cannot reduce the rule $A \to \mathtt{a}CD$. Instead, the parser had to reduce the $\mathtt{c}$ into C *before* shifting and reducing the $\mathtt{d}$.

Observe that this grammar is not LL(1), because of the two rules having A as the left-hand side. Nevertheless, we will manage to parse it with a bottom-up parser *without* any symbol of prevision.

(1)	S	$\to$	$A\$$
(2)	A	$\to$	$\mathtt{a}CD$
(3)		$\to$	$\mathtt{ab}$
(4)	C	$\to$	$\mathtt{c}$
(5)	D	$\to$	$\mathtt{d}$

Figure 6.2: An example grammar to demonstrate the notion of viable prefix.

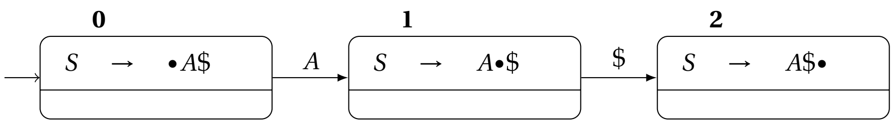
Now, let's see how to build the CFSM. If we consider the first rule of the grammar $S \to A\$$, we see immediately that A and $A\$$ are viable prefixes. So, we could start building our CFSM by having a three-state automaton of the form:



However, to track the progress of the automataon along the rule $S \to A\$$, we will associate each state with so-called *items*. An *item* is simply a grammar rule where we have inserted a $\bullet$ at some point in the right-hand part, in order to makr the current progress of the CFSM (and, as we will see later, of the bottom-up parser):

Definition 6.5 (CFSM item)**.** Given a grammar $G = \langle V, T, P, S\rangle$, an *item* of G is a rule of the form $A \to \alpha_1 \bullet \alpha_2$, where $A \to \alpha_1\alpha_2 \in P$ is a rule of G. We denote by Items(G) the set of items of G.

So, intuitively the item $S \to A\bullet\$$ means: (i) that the automaton is trying to recognise the *handle* $A\$$ of the rule $S \to A\$$, (ii) that it has recognised an A so far, and (iii) that it still expects to read a \$ to complete the handle (and hence, complete the viable prefix). Using this notation, our automaton becomes:



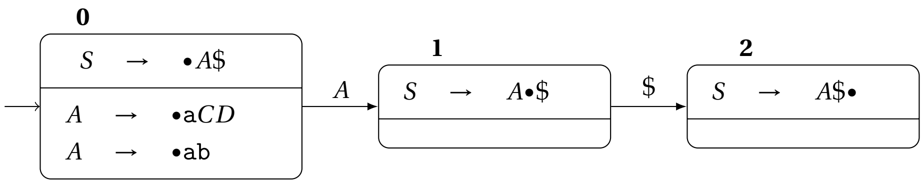
Observe that we have slightly altered our conventions for depicting automata in this figure (in order to reflect common practice of the literature). First, we do not mark explicitly the states as 'accepting' since they are all accepting anyway. Indeed, the prefix of a viable prefix is itself a viable prefix. Second, our states are now divided into two parts: we will understand why in a moment. Finally, we are now numbering the states, to be able to refer to them easily (see the bold numbers on top left of states).

Note that, for the moment, our automaton contains only one item per state. In general states of the CFSM will be *sets* of prefixes. Observe that, for a given grammar, there are finitely many items, because there are finitely many rules, which have all a finite right-hand side. So the CFSM is guaranteed to be finite.

The fact that states of the CFSM are *sets* of items should not be surprising, since we are building a *deterministic* automaton, but there can be *non-determinism* in the grammar. This is reminiscent of the subset construction technique to determinise finite automata from Section 2.4.2.

Clearly, this automaton is not sufficient to accept all *viable prefixes*. Indeed, $S \Rightarrow A\$ \Rightarrow \mathtt{a}CD\$$, for instance, is a possible prefix of derivation in our grammar, so we should also accept the viable prefixes a, aC, *etc*.

How can we obtain this? We need to incorporate in our CFSM the fact that A can be derived as $\mathtt{a}CD$, but where to add this information? If we observe the initial state, it contains the item $S \rightarrow \bullet A\$$, in which the $\bullet$ symbol is immediately followed by the *variable* A. This is a sign that we need to add more information to the initial state: when the automaton tries to recognise a viable prefix generated using rule $S \rightarrow A\$$, it might need to read an a, because the rule $A \rightarrow \mathtt{a}CD$ might be applied next. Thus, we will add to the initial states two new items: $A \rightarrow \bullet \mathtt{a}CD$ and $A \rightarrow \bullet \mathtt{ab}$.



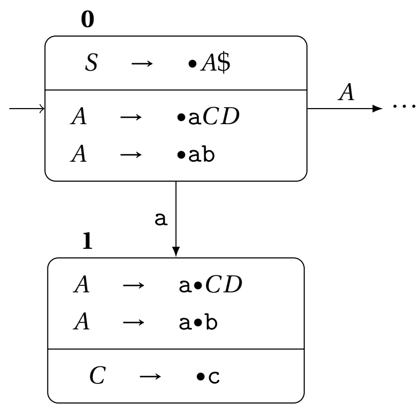
The operation that consists in looking, in a state, for all the items of the form $A \rightarrow \alpha_1 \bullet B \alpha_2$, where B is a variable; and adding to the state all the items of the form $B \rightarrow \bullet \alpha$ is called the *closure* operation. It needs to be applied to all states[10] of the CFSM (possibly several times until no more items can be added).

[10] The closure operation does not add items to states **1** and **2**, because the $\bullet$ is not followed by a variable in the corresponding items.

In our depiction, the items that result from the closure operation appear in the lower part of the states. The items from the top part of the states form the *kernel* of the state. In our depiction we keep them separated because it makes it easier to identify states, since the closure of a given kernel will always be the same.

This new information in the initial state allows us to complete our automaton. Since the state now contains items $A \rightarrow \bullet \mathtt{a}CD$ and $A \rightarrow \bullet \mathtt{ab}$, the automaton can progress in the recognition of a valid prefix by reading an a from the initial state. This will yield a new state where the automaton has progressed in *both* items, so the kernel of the new state will contain both $A \rightarrow \mathtt{a} \bullet CD$ and $A \rightarrow \mathtt{a} \bullet \mathtt{b}$. This means that we don't know, at this point, whether the handle that will be read will be $\mathtt{a}CD$ or ab, but both are still possible so far. In addition, the closure operation will add the item $C \rightarrow \bullet \mathtt{c}$ to the state, because the $\bullet$ is directly followed by a C in $A \rightarrow \mathtt{a} \bullet CD$:

Again, compare with the subset construction for determinising finite automata.



Continuing this systematic construction, we obtain the automaton which is shown in Figure 6.3.

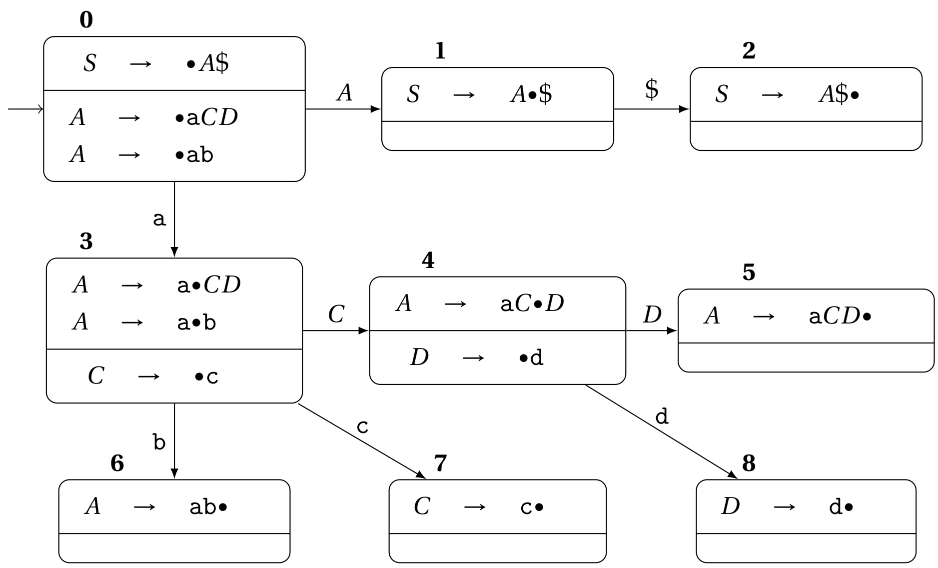


Figure 6.3: The CFSM for our example grammar. All states shown on the figure are accepting.

With these intuitions in mind, we can now describe formally the construction of the CFSM for a given CFL G.

The Closure *operation* We start by giving an algorithm to compute the closure that we have informally discussed above. It is given in Algorithm 9. As can be seen, it is a fixed point algorithm, to adds items of the form $B \to \bullet\beta$ for every item of the form $A \to \alpha_1 \bullet B \alpha_2$ where the $\bullet$ is followed by some variable B. This operation is carried on up to stabilisation, i.e. until no more items can be added to the set, which is then returned.

Computing the successors of items The next step in the construction is to define formally the notion of *successor* in a CFSM. For that, we define a function that receives a set of items $I \subseteq \text{Items}(G)$ and a terminal $a \in T$, and returns the set $\text{CFSMSucc}(I, a)$ of all the items obtained from I after reading a. This is obtained by selecting, from I, all the items of the form $A \to \alpha_1 \bullet a \alpha_2$, where the $\bullet$ is immediately followed by a; and by moving the $\bullet$ one position ot the right, in order to reflect the progress of the

Thus, I is a state of the CFSM.

Input: A set $I \subseteq \text{Items}(G)$ of items of some CFG $G = \langle V, T, P, S\rangle$.
Output: The closure of I

Closure $\leftarrow I$;
repeat
  PrecClosure $\leftarrow$ Closure ;
  foreach $A \rightarrow \alpha_1 \bullet B\alpha_2 \in$ Closure *s.t.* $B \in V$ **do**
    Closure $\leftarrow$ Closure $\cup \{B \rightarrow \bullet\beta \mid B \rightarrow \beta \in P\}$;
until PrecClosure = Closure;
return Closure ;

Algorithm 9: Computing the Closure of a set I of items for the LR(0) CFSM.

automaton. Formally:

$$\text{CFSMSucc}(I, a) = \{A \rightarrow \alpha_1 a \bullet \alpha_2 \,|\, A \rightarrow \alpha_1 \bullet a\alpha_2 \in I\}.$$

Construction of the CFSM Then, the construction of the CFSM associated to a CFL G is rather straightforward.

Definition 6.6 (Canonical Finite State Machine)**.** Let $G = \langle V, T, P, S\rangle$ be a CFG. Its *Canonical Finite State Machine* (or CFSM for short) is the DFA

$$\text{CFSM}(G) = \langle Q, V \cup T, \delta, q_0, F\rangle,$$

where:

1. the set of states Q is the set of all sets of G's items: $Q = 2^{\text{Items}(G)}$;
2. the initial state $q_0 = \text{Closure}(\{S \rightarrow \bullet\alpha \mid S \rightarrow \alpha \in P\})$ is obtained by taking the closure of all the items obtained from the rules where S is the left-hand side, and where the $\bullet$ appears at the initial position (on the left of the right-hand side);
3. for all $q \in Q$, for all $a \in V \cup T$:

$$\delta(q, a) = \text{Closure}(\text{CFSMSucc}(q, a)).$$

   That is, the transition function is obtained by first computing the successors of q by a (hence by advancing the $\bullet$ one position to the right in the relevant items), and then taking the closure of the resulting set.
4. all the states are accepting but the empty set, which is considered as an error state: $F = Q \setminus \{\emptyset\}$.

The main property of the CFSM is given by the following theorem (which we will not prove here):

Theorem 6.3. *For all CFG G:* CFSM(G) *accepts the set of all viables prefixes of G.*

Now that we have the CFSM in our toolbox, how can we exploit it to make our bottom-up parsers deterministic? In the next sections, we will introduce the LR(k) parsers, which are a family of deterministic bottom-up parsers that use k characters of look-ahead. Similarly to the case of LL(k), the 'LR' in LR(k) stands for *Left scanning, Right parsing*.

Observe that the automaton is indeed *deterministic*: the Closure function returns a *set* of items, which is indeed a *state* of the CFSM.

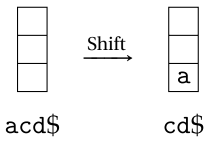
Observe that the automaton is actually complete: there are transitions labelled by all alphabet symbols from all states. However, these transitions can lead to $\emptyset$, which is the error state, and which we usually do *not* depict (see for instance Figure 6.3) to keep figures compact and readable.

6.3 LR(0) *parsers*

We start with the LR(0) parsers, which use *no look-ahead*. This might sound surprising, since it is hard to imagine that a *top-down* parser would be able to parse anything meaningful without look-ahead. As a matter of fact, a grammar is LL(0) iff it does not contain two rules with the same left-hand side, which is a very strong restriction, since such grammars would accept only one word at most! Nevertheless, LR parsers are usually regarded as *more powerful* than LL parsers, and there are non-trivial grammars (such as the one in Example 6.4) that can be parsed by an LR(0) parser, as shown by the next example, which will help us build our intuition.

Example 6.7. Let us consider again the grammar G of Example 6.4, and the word `acd`\$ which is accepted by this grammar. How does CFSM (G), as given in Figure 6.3, helps us parse `acd`\$ in a deterministic fashion.

Let us observe the run of the bottom-up parser on `acd`\$. Initially, the stack is empty[11], and the only thing the parser can do is to shift `a` (since there is no rule which has ε as its right-hand side, no Reduce is possible):

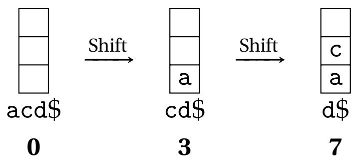


[11] It contains only the Z_0 character, that we will *not* depict in this example in order to keep it readable.

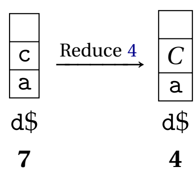
As usual, we represent the sequence of stack contents, with the current input under the stack.

Don't forget that the viable prefixes are *mirror images* of the stack contents, see Definition 6.3. So, the stack content must be read from the bottom to the top by the CFSM!

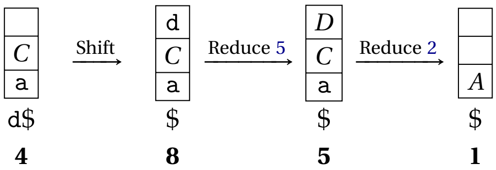
Now, let us consider the associated CFSM. Remember that its goal is to accept viable prefixes, which are *stack contents*, so let us run it on the two stack contents obtained so far. In the initial configuration of the parser, the stack was empty, so the CFSM reaches state **0**. Observe that in this state, there are no items of the form $A \rightarrow \alpha\bullet$, which indicates that no handle has been read completely. This is coherent with our decision of *Shifting* and not *Reducing* at this point. In the second configuration, the CFSM reaches state **1**. Again, the items contained in this state indicate that a *Shift* must occur, and the parser reaches a configuration where the stack contains (from the bottom to the top) `ac`, which moves the CFSM to state **7**. We depict this as follows, indicating the current CFSM state under the input:



Now, the parser is in a situation of Shift/Reduce conflict: indeed, it could reduce the `c` on the top of the stack to C, or it could shift the `d`. The CFSM helps us lift this conflict: in state **7**, the only item is $C \rightarrow \mathtt{c}\bullet$, which indicates that the handle for $C \rightarrow \mathtt{c}$ has been read completely and must be reduced. This yields:



Following the same reasoning, we now *Shift* the d, and the CFSM reaches state **8** from which rule 5 must be reduced (item $D \rightarrow \mathtt{d}\bullet$). After this reduce, the CFSM is in state **5**, where a *Reduce* of rule 2 occurs, which moves the CFSM to state **1**, as the stack now contains A only:



Finally, the parser shifts the remaining character $, which moves the CFSM to state **2**. In this state, the parser reduces rule 1 which leaves only the start symbol S on the top of the stack. This amounts to accepting the input string, and is denoted by the *Accept* action. The full run is displayed in Figure 6.4 (top of the figure).

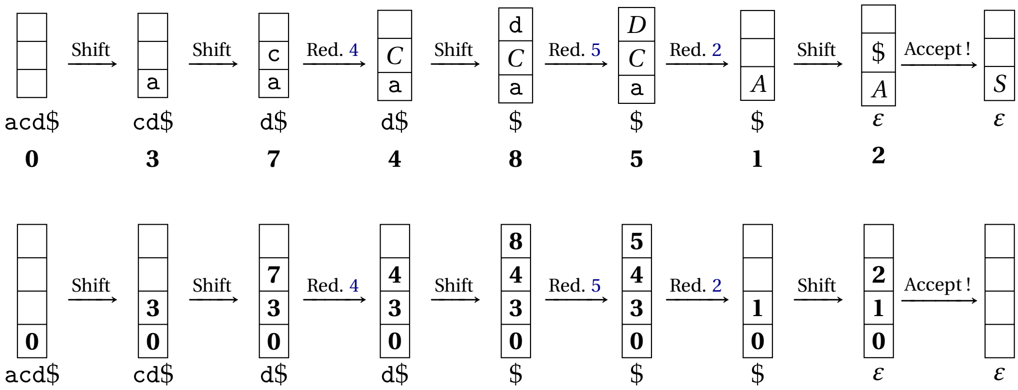


Figure 6.4: Two versions of the run of the LR(0) parser on acd$. On the top: the run where we push grammar symbols on the stack. Bottom: the same run where we push CFSM states instead (which is the actual run of the LR(0) parser).

Let us summarise what we have learned from this example. First, we can always associate a state of the CFSM to the current stack content, and this state is sufficient to determine the action that the parser must perform. These actions are to:

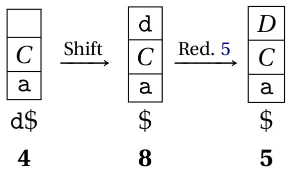
- *Accept* in all states containing an item of the form $S \rightarrow \alpha\bullet$;
- *Shift* whenever the current state does *not* contain an item of the form $A \rightarrow \alpha\bullet$ because we have not yet shifted a complete handle on the stack; and
- *Reduce* rule $A \rightarrow \alpha$ whenever the current state contains the item $A \rightarrow \alpha\bullet$ (except if this is a state where an *Accept* is performed).

As in the case of top-down parsers, this can be summed up in an *action table*, as follows, where an 'A' stands for *Accept*, an 'S' stands for *Shift*, a number i stands for '*Reduce* rule number i', and states are identified by their numbers:

State	Action
0	S
1	S
2	A
3	S
4	S
5	2
6	3
7	4
8	5

Observe that in our example, each line of the table contains one and only one action. This need not be the case in general: one could imagine a single state of the CFSM containing both $A \to \alpha\bullet$ and $B \to \beta\bullet$ (a Reduce/Reduce conflict) or both $A \to \alpha_1 \bullet \alpha_2$ and $B \to \beta\bullet$ (a Shift/Reduce conflict). We will see such examples later and how to solve them using look-ahead.

Now, since the CFSM state is the only meaningful information for determining the behaviour of the parser, one could wonder what is the point of actually pushing symbols on the stack? It turns out that we can, instead, push the sequence of CFSM states. To better understand this, consider the excerpt of the run above:



In this example, the CFSM state is **4** when the stack content is Ca, then a *Shift* occurs, which moves the CFSM to state **8**. To obtain this state **8**, we only need to know that we were previously in state **4** and that we have read symbol d. Then, a *Reduce* occurs that pops one character (because the right-hand side of rule 5 has length 1), pushes a D instead, and moves the CFSM to state **5**. It is here crucial to observe that, in order to determine that the CFSM ends up in state **5**, all we need to know is:

1. the current state of the CFSM *before* the reduced handle (here: d) was on the stack (here this state was **4**); and
2. the fact that we are pushing a D into the stack.

Indeed, in the CFSM: $\delta(\mathbf{4}, D) = \mathbf{5}$. In other words, to determine that the CFSM reaches state **5** when the stack contains DCa, one does not need to re-run the CFSM on the whole stack content from scratch: instead, we can *remember* on the stack the state reached with stack content Ca (here, state **4**), then look for the (unique) successor of **4** by letter D. This holds because the CFSM is *deterministic*.

We abuse notations and denote CFSM states by their numbers.

So, in practice, the run of the LR(0) parser on acd is the one displayed in Figure 6.4 (bottom). Let us now formalise these ideas.

Observe that, since the stack now contains the initial state of the CFSM from the beginning, we do not even need the Z_0 symbol anymore. Or, in other words, we let $Z_0 = \mathbf{0}$, and the *Accept* will pop **0** to empty the stack.

6.3.1 *Action table of the* LR(0) *parser*

The action table of the LR(0) parser associates one or several actions to each state:

Definition 6.8 (LR(0) action table)**.** Let $G = \langle V, T, P, S\rangle$ be a CFG with its associated canonical finite state machine $\text{CFSM}(G) = \left\langle Q, q_0, V \cup T, \delta, F \right\rangle$, and let us assume that:

- G's rules (i.e. the elements of P) are indexed from 1 to n; and
- The non-empty states of CFSM(G) (i.e. the elements of $Q \setminus \emptyset$) are indexed from $\mathbf{0}$ to $\mathbf{m-1}$, so that $Q = \{\mathbf{0}, \mathbf{1} \ldots, \mathbf{m-1}, \emptyset\}$.

Then, the LR(0) *action table* is a table M with $|Q| = m+1$ lines s.t., for all $0 \le i \le m$, $M[i]$ contains a *set of actions*. The actions can be:

- either an integer $1 \le j \le n$ denoting a *Reduce* of rule number j;
- or Shift denoting that the parser must *Shift* the next character of the input to the stack ;
- or Accept denoting that the string read so far is accepted and belongs to $L(G)$;
- or Error denoting that the string must be rejected. The only state which is associated with this action (in the case of LR(0)) is $\emptyset$.

As in the case of LL(1) our definition allows for *several* actions in a given cell, but the parser will be deterministic only when there is exactly one action per cell.

When we depict the action tables, we often denote the *Shift*, the *Accept* and the *Error* by S, A and an empty cell, respectively.

To build this action table, we look for items of the form $A \rightarrow \alpha \bullet$ in the states of the CFSM and we associate a *Reduce* of the corresponding rule to those states. When the state contains $A \rightarrow \alpha_1 \bullet \alpha_2$, we associate a *Shift* to the state. This is detailed in Algorithm 10. Observe that after the execution of this algorithm, no cell will be empty, because each state of the CFSM always contains at least one item that either will satisfy one of the two **If**'s of the main loop, or will allow for at least one execution of the innermost **foreach** loop.

6.3.2 *Running the* LR(0) *parser*

Now that we have described the construction of the action table of the LR(0) parser, let us formalise how it runs on a given input string. For the sake of clarity, we will not describe this parser as a PDA, but rather as an algorithm that can access a stack $\mathscr{S}$ from which it can push and pop. The parser is given in Algorithm 11 and assumes that the action table contains exactly one action per cell. The NextSymbol() function reads and returns the next symbol on the input (i.e. in the word w).

It is easy to check that the algorithm follows exactly the intuitions that we have described so far. Namely, in the main **while** loop, the parser checks the content of the LR(0) table in the cell given by the top of the stack, to obtain the action that must be executed. Then:

- if this action is an Error or an Accept, the execution finishes immediately;
- if the action is a Shift, the next character is read and stored in variable c; and

```
Input: A CFG G = ⟨V, T, P, S⟩ whose rules are indexed from 1 to n;
       and CFSM(G) = ⟨Q, 0, V ∪ T, δ, F⟩ where
       Q = {0, 1 ..., m − 1, ∅}.
Output: LR(0) action table of G.

/* Initialisation                                          */
M[∅] ← Error;
foreach q ∈ {0, 1, ..., m − 1} do
    M[q] ← ∅ ;
/* Populating the table                                    */
foreach q ∈ {0, 1, ..., m − 1} do
    if q contains an item of the form S→α• then
        M[q] ← M[q] ∪ {Accept} ;
    foreach item in q that is of the form A→α•  with A ≠ S do
        M[q] ← M[q] ∪ {j} where rule number j is  A→α ;
    if q contains an item of the form A→α1•α2  with α2 ≠ ε then
        M[q] ← M[q] ∪ {Shift} ;
return M ;
```

Algorithm 10: The algorithm to compute the LR(0) action table of a CFG G using CFSM(G).

```
Input: The LR(0) action table M of some CFG G and its associated
       CFSM(G) = ⟨Q, 0, V ∪ T, δ, F⟩; and an input word w.
Output: True iff w ∈ L(G)

Let 𝒮 be an empty stack ;
Push(𝒮, 0) ;            // Push of the initial CFSM state
while M[Top(𝒮)] ≠ Error and M[Top(𝒮)] ≠ Accept do
    if M[Top(𝒮)] = {Shift} then
        c ← NextSymbol() ;
    else if M[Top(𝒮)] = {i} then
        Let  A→α  be rule number i in G;
        Pop() |α| times from 𝒮 ;
        c ← A ;
    /* Compute the next CFSM state and push it             */
    nextState ← δ(Top(𝒮), c) ;
    Push(𝒮, nextState) ;
```

Algorithm 11: The LR(0) top-down parser. This algorithm assumes that each cell of action table M contains exactly one action.

- if the action is a rule number i (meaning that a *Reduce* of rule number i must be performed), and if rule number i is $A \to \alpha$, then $|\alpha|$ state numbers are popped from the stack, and grammar variable A is stored in c.

Finally, the parser computes the next state based on the current state (which is now on top of stack) and the symbol c that the CFSM must read. This new state is pushed onto the stack. After all these actions have been performed, the state of the parser has been correctly updated, and it can continue its execution the same way.

In practice, only the transition relation of the CFSM is needed for the parser. It can be given by a simple table that returns, for all states s and terminal a, the successor state of s by a.

6.3.3 *Another example*

We close this section by discussing a final example of LR(0) grammar, one where the set of viable prefixes is infinite (yet, still regular, as it always is).

Example 6.9. Let us consider the simple grammar in Figure 6.5. This grammar has one recursive rule, and the rule $S \to \varepsilon$, which is particularly interesting here. Indeed, a bottom-up parser parsing this grammar will necessarily need to *Reduce* it at some point (it is needed to terminate the recursion). But reducing this rule has a perhaps unexpected effect: since the right-hand side is ε, there is nothing to pop from the stack when performing the reduction. So, reducing $S \to \varepsilon$ amounts to pushing S on the top of the stack, and this can occur basically at any moment in the run of the parser. This might give use the impression that we will end up with a non-deterministic parser, yet our LR(0) parser will be deterministic, even without look-ahead!

(1)	S'	$\to$	$S\$$
(2)	S	$\to$	$S\mathtt{a}S\mathtt{b}$
(3)		$\to$	ε

Figure 6.5: An LR(0) grammar whose set of viable prefixes is infinite.

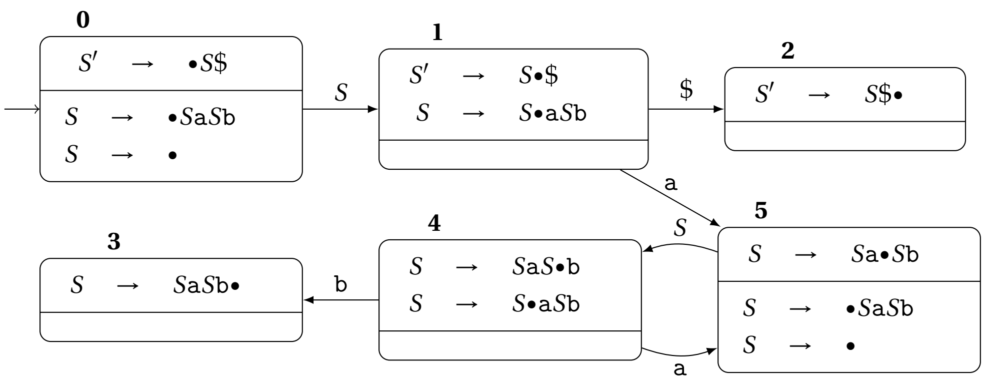


Figure 6.6: The CFSM of the previous grammar. Its language is infinite.

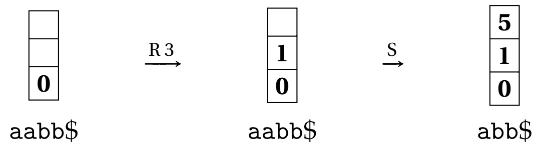
The (LR(0)) CFSM of the grammar is given in Figure 6.6. On can check that its language is indeed infinite since it contains a loop between states **4** and **5**. One can also remark the presence of the item $S \to \bullet$ in states **0** and **5**, that triggers the reduce of rule number 3 (i.e., pushing an S on the top of the stack as discussed above).

Then, the LR(0) action table is given in Table 6.1. Observe that states **0** and **5**, again, prompt a *Reduce* of $S \to \varepsilon$, and this *Reduce* only. There is no *Shift* in these states since there is no outgoing transition labeled by a terminal.

Table 6.1: The LR(0) action table of the previous grammar.

State	Action
0	3
1	S
2	A
3	2
4	S
5	3

Now, let us build the run of the LR(0) parser on aabb\$. As always, we start with the initial state **0** on the stack. In state **0**, a *Reduce* of $S \to \varepsilon$ must be performed. This amounts to pushing an S on the stack, or rather pushing the CFSM state which is the successor of **0** by S, i.e. **1**. From **1**, a *Shift* (of a) is performed, and we move to state **5**. So the three first steps of the run are:



After that, the run continues as usual. The full run is given in the following table (in order to save space), where the stack content is displayed from the bottom to the top:

Input	Stack content	Action
aabb$	**0**	*Reduce* 3
aabb$	**01**	*Shift*
abb$	**015**	*Reduce* 3
abb$	**0154**	*Shift*
bb$	**01545**	*Reduce* 3
bb$	**015454**	*Shift*
b$	**0154543**	*Reduce* 2
b$	**0154**	*Shift*
$	**01543**	*Reduce* 2
$	**01**	*Shift*
ε	**012**	*Accept*

6.4 SLR(1) *parsers*

One thing which is remarkable about LR(0) parsers is their ability to parse grammars that are not entirely trivial, *without using any look-ahead*, as the previous examples have clearly shown. However, as it often happens with life and computing, *there is no free lunch*, and there are grammars of practical interest that we cannot parse *deterministically* with LR(0) parsers! Let us have a look at such an example...

Example 6.10. Consider the grammar in Figure 6.7. It is a simplification of the grammar to generate arithmetic expressions that we have considered several times before (the point of the simplification here is to keep the example short, but everything we are about to discuss extends to the grammar with all operators). Notice that this grammar implements the priority of operator as discussed at the end of Chapter 4.

(1)	S	→	Exp$
(2)	Exp	→	Exp + Prod
(3)		→	Prod
(4)	Prod	→	Prod ∗ Atom
(5)		→	Atom
(6)	Atom	→	Id
(7)		→	(Exp)

Figure 6.7: A simple grammar for generating arithmetic expressions. This grammar is not LR(0). It is also not LL(k) for any k since it contains left-recursive rules.

We claim that this grammar is not LR(0). Let us build its CFSM to check this. It is given in Figure 6.8. We can immediately spot conflicts in state **1** and in state **12**. In state **1** the parser cannot decide between performing a shift (which will necessarily be of symbol ∗), or reducing Exp → Prod. Similarly, in state **12**, there is a conflict between a shift (of ∗ again) and a reduce of Exp → Exp + Prod.

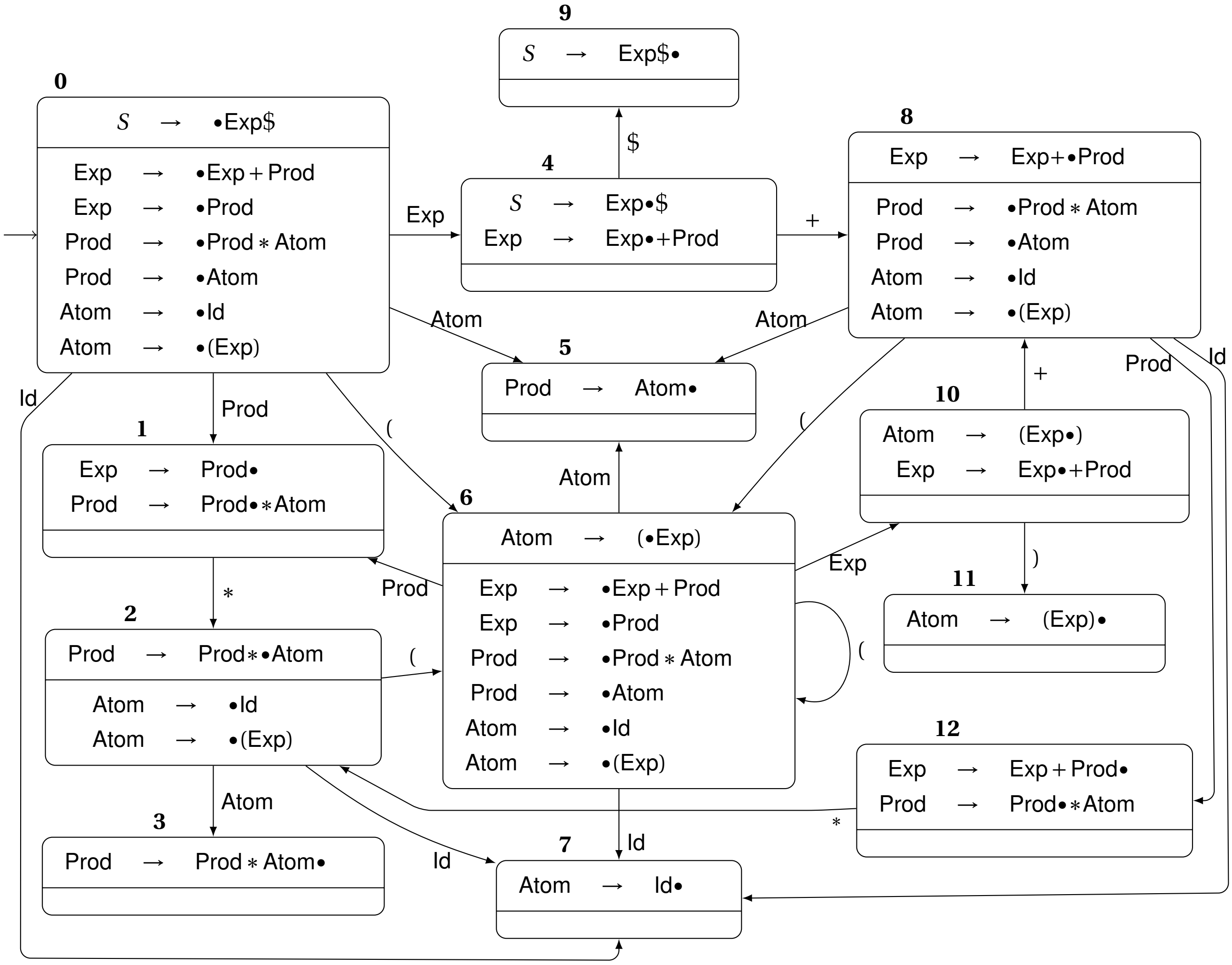


Figure 6.8: The CFSM for the grammar generating expressions.

While this example shows that some grammars cannot be parsed by a deterministic bottom-up parser without any symbol of look-ahead, it also suggests an easy way to lift this conflict. Let us again consider state **1**. Clearly, in this state, the *Shift* action should occur only when a $*$ symbol is the next character on the input (all other shifts lead to the error state that is not shown). But how can we find a subset of symbols that contain at least all the symbols for which a *Reduce* of $\mathsf{Exp} \rightarrow \mathsf{Prod}$ should occur?

We will later introduce formally the notion of LR(k) grammar, and we will be able to check that the grammar of the previous example is *not* LR(0).

Another way to express this question is the following: 'what are the next symbols that we expect on the input when the top-down parser is supposed to reduce $A \rightarrow \alpha$?' Clearly, if the top-down parser is to reduce $A \rightarrow \alpha$, this means that is has already read (i.e., shifted on the stack) *a string of symbols that have been derived from A*. So, the next symbol on the input (i.e., the first one that we have not shifted yet), must be a symbol that we expect *after* a word which is derived from A; and this is exactly the definition of $\mathrm{Follow}(A)$.

We have thus found a practical way to lift the conflicts, at least in our example:

1. whenever a state contains an item of the form $A \rightarrow \alpha\bullet$, perform a *Reduce* of $A \rightarrow \alpha$ if and only if the look-ahead is a symbol from $\mathrm{Follow}(A)$;

2. whenever a state contains an item of the form $A \rightarrow \alpha_1 \bullet \alpha_2$ where α_2 starts with a terminal, perform a *Shift* if and only if the look-ahead is a symbol from $\mathrm{First}(\alpha_2)$.

The technique we have just sketched has been introduced in 1969 by Franklin DeRemer in his PhD thesis[12], and is called *Simple* LR(1) or SLR(1) for short, because it uses one character of look-ahead and it can be regarded as a simplification of the more general LR(1) technique that we will introduce next. In order to formalise it, we have to modify the algorithm building the action table, and the parsing algorithm of LR(0). The construction of the CFSM stays the same. Those new algorithms are given in Algorithm 12 (for the construction of the action table) and in Algorithm 13 (for the actual parser).

[12] Franklin Lewis DeRemer. *Practical Translators for LR(k) Languages*. PhD thesis, Massachusetts Institute of Technology, 1969. URL https://web.archive.org/web/20130819012838/http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-065.pdf; and Franklin Lewis DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7), 1971. DOI: 10.1145/362619.362625

If this 'simple', what will come next? You have no idea...

On a more serious note, we point out that this technique can be generalised to k characters of look-ahead.

Observe that the action table M is now computed as a *two-dimensional* table. For all states q of the CFSM, and for all symbols $a \in T \cup \{\varepsilon\}$: $M[q, a]$ provides the parser with the action(s) that it should perform when the current state of the CFSM is q and the next symbol on the input (i.e. the look-ahead) is a.

Input: A CFG $G = \langle V, T, P, S\rangle$ whose rules are indexed from 1 to n;
and CFSM$(G) = \langle Q, \mathbf{0}, V \cup T, \delta, F\rangle$ where
$Q = \{\mathbf{0}, \mathbf{1} \ldots, \mathbf{m-1}, \emptyset\}$.

Output: SLR(1) action table of G.

```
/* Initialisation                                              */
foreach a ∈ T do
    M[∅, a] ← Error;
    foreach q ∈ {0, 1, ..., m − 1} do
        M[q, a] ← ∅ ;
/* Populating the table                                        */
foreach q ∈ {0, 1, ..., m − 1} do
    if q contains an item of the form S→α• then
        M[q, ε] ← M[q, ε] ∪ {Accept} ;
    foreach item in q that is of the form A→α•  with A ≠ S do
        foreach a ∈ Follow(A) do
            M[q, a] ← M[q, a] ∪ {j} where rule number j is  A→α  ;
    foreach item in q that is of the form A→α1•aα2  with a ∈ T do
        M[q, a] ← M[q, a] ∪ {Shift} ;
return M ;
```

Algorithm 12: The algorithm to compute the SLR(1) action table of a CFG G using CFSM(G).

Input: The SLR(1) or LR(1) action table M of some CFG G and its associated CFSM$(G) = \langle Q, \mathbf{0}, V \cup T, \delta, F\rangle$; and an input word w.

Output: **True** iff $w \in L(G)$

```
Let 𝒮 be an empty stack ;
/* Push of the initial CFSM state                              */
Push(𝒮, 0) ;
/* Initialisation of the look-ahead                            */
ℓ ← first symbol on the input ;
while M[Top(𝒮), ℓ] ≠ Error and M[Top(𝒮), ℓ] ≠ Accept do
    if M[Top(𝒮), ℓ] = {Shift} then
        c ← NextSymbol() ;
    else if M[Top(𝒮), ℓ] = {i} then
        Let  A→α  be rule number i in G;
        Pop() |α| times from 𝒮 ;
        c ← A ;
    /* Compute the next CFSM state and push it                 */
    nextState ← δ(Top(𝒮), c) ;
    Push(𝒮, nextState) ;
    /* Update the look-ahead                                   */
    ℓ ← first symbol on the remaining input ;
```

Algorithm 13: The SLR(1) and LR(1) top-down parser. This algorithm assumes that each cell of action table M contains exactly one action.

Table 6.2: The SLR(1) action table for our simple grammar of expressions. The first column gives the state number, the first row lists the possible look-ahead symbols.

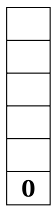
M	$+$	$*$	Id	(	)	\$	ε
0			S	S			
1	3	S			3	3	
2			S	S			
3	4	4			4	4	
4	S					S	
5	5	5			5	5	
6			S	S			
7	6	6			6	6	
8			S	S			
9							A
10	S				S		
11	7	7			7	7	
12	2	S			2	2	

Example 6.11. We continue Example 6.10, and build the action table of the grammar in Figure 6.7. To this end, we first compute:

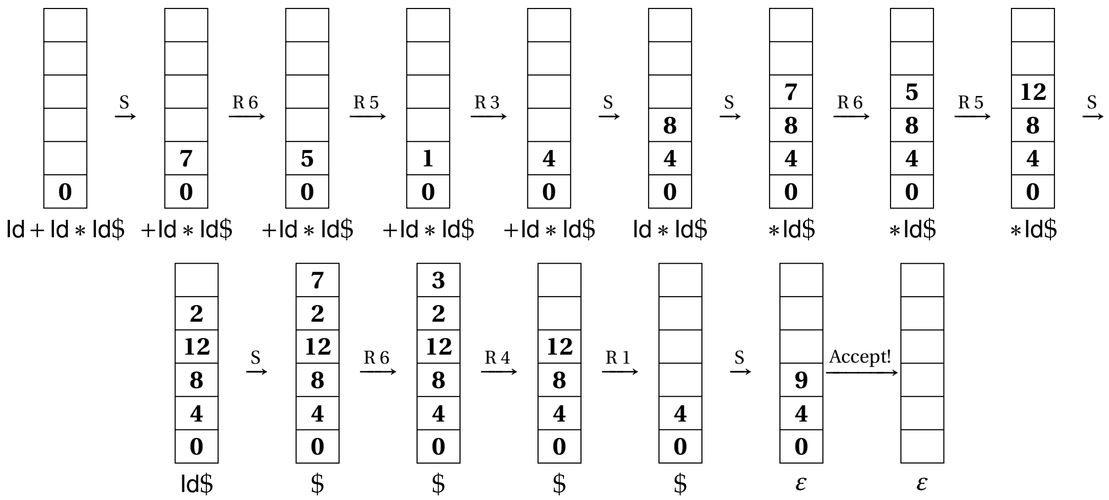
1. $\mathsf{Follow}(\mathsf{Exp}) = \{\$, +,)\}$;
2. $\mathsf{Follow}(\mathsf{Prod}) = \{*\} \cup \mathsf{Follow}(\mathsf{Exp}) = \{*, \$, +,)\}$;
3. $\mathsf{Follow}(\mathsf{Atom}) = \mathsf{Follow}(\mathsf{Prod}) = \{*, \$, +,)\}$.

Then, we obtain the table given in Figure 6.2. Observe that this table contains no conflict now.

Now, let us consider the word $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}\$$, which is in the language of the grammar, and let us observe the corresponding run of the SLR(1) parser. The run starts as usual with:



When the CFSM is in state **0** and the look-ahead is Id, the action table says to perform a *Shift*, which leads to state **7**. In this new state, and with a new look-ahead equal to $+$, the parser must *Reduce* rule 6, which leads to sate **5**, and so on… The full run is then:



6.5 LR(k) parsers

While SLR(1)1 parsers offer a solution to some of the conflicts that exist in LR(0) parsers, they are not paramount as we will see in the next example:

(1)	S'	$\rightarrow$	$S\$$
(2)	S	$\rightarrow$	$L = R$
(3)	S	$\rightarrow$	R
(4)	L	$\rightarrow$	$*R$
(5)	L	$\rightarrow$	Id
(6)	R	$\rightarrow$	L

Figure 6.9: A grammar which is not SLR(1).

Example 6.12. Let us consider the grammar from Figure 6.9. It can generate assignments where both the left-hand and the right-hand sides can contain references. For example, the assignment $**\mathsf{Id} = ***\mathsf{Id}\$$ can be generated by this grammar. Observe that we have the following values for the Follows:

- Follow(S) = {\$};
- Follow(L) = {=, \$}; and
- Follow(R) = {=, \$}.

Now, let us try and build the CFSM for this grammar. An excerpt is given in Figure 6.10. On this small excerpt, we can clearly see that there is a shift/reduce conflict in state **1**, if we build an LR(0) parser. Unfortunately, this conflict subsides if we build an SLR(1) parser. Clearly, when the look-ahead is =, a *Shift* can be performed. But since $= \in$ Follow(S), we would also perform a *Reduce* in this case.

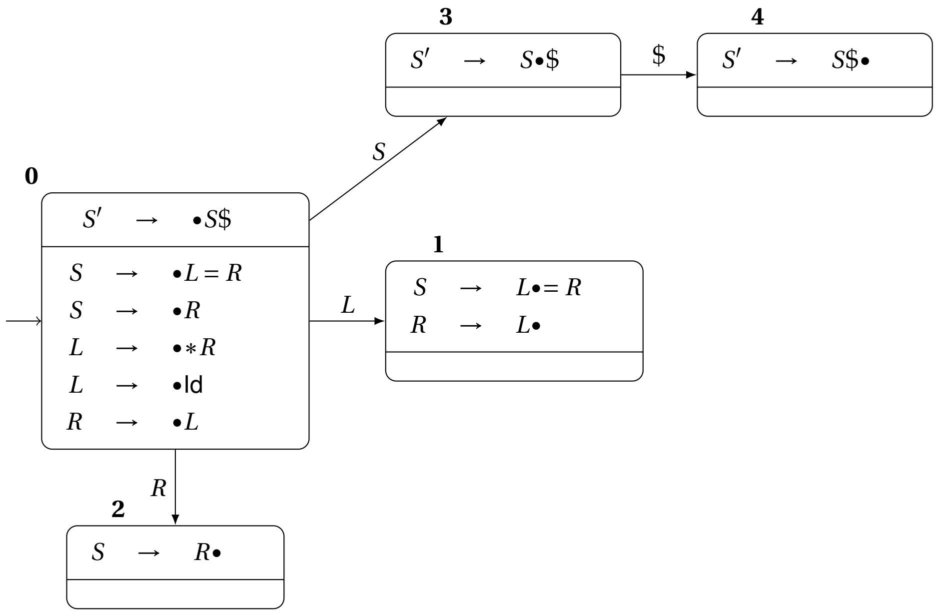


Figure 6.10: An excerpt of the CFSM for the previous grammar.

This example clearly shows that using Follow(R) to lift the conflict does not work in this example. Observe, however that Follow(R) contains *two* elements: = and \$, but only one of them causes the conflict. Could it be the case that we can refine the notion of Follow in order to find a way to avoid this conflict? In the present example, it turns out to be possible!

Indeed, let us check how we came to have the item $R \rightarrow L\bullet$ in state **1**. First, starting from $S' \rightarrow \bullet S\$$ in state **0**, we have applied the closure operation to yield item $S \rightarrow \bullet R$, then another iteration of the closure to obtain $R \rightarrow \bullet L$, both in state **0**.

Now, observe that we have introduced $S \rightarrow \bullet R$, because we expect to reduce an R into an S, in order to finally reduce $S\$$ into S' (item $S' \rightarrow \bullet S\$$). But this means that we expect to have an R on the top of the stack in a particular *context*, which is when it is *followed by a* \$. And then, since we expect to reduce the R from an L (item $R \rightarrow \bullet L$), it means, in turn, that we expect to find an L on the top of the stack *only in the context where it is followed by a* \$.

A contrario, imagine that we perform a *Reduce* of $R \rightarrow L$ in state **1**, with an = as look-ahead. Then, the parser will move to state **2**, where it will reduce an S (still with = as look-ahead), and further move to state **3**. However, from this state, it is not possible to *Shift* the = and to continue the run of the parser.

With this discussion, we conclude that, in state **1**, an = in the look-ahead must trigger a *Shift*; while we must do a *Reduce* only when \$ is on the look-ahead. By doing so, we actually *refined* the notion of Follow. While $\text{Follow}(R) = \{=, \$\}$, this can be regarded as a *global* Follow, because it is computed over the whole grammar. With the finer analysis we have sketched here, we have obtained some sort of *local* Follow: in state **1**, this local Follow of R contains only \$, which allows us to lift the conflict. The point of this section is to formalise those notions.

6.5.1 *The* LR(k) *CFSM*

In order to formalise the ideas we have just sketched, we need to extend the notion of CFSM, to incorporate *local Follows*. We start by redefining the notion of item:

Definition 6.13 (LR(k) CFSM item)**.** Given a grammar $G = \langle V, T, P, S\rangle$, an *item* of G is a rule of the form $A \to \alpha_1 \bullet \alpha_2,\ u$, where $A \to \alpha_1\alpha_2 \in P$ is a rule of G and $u \in T^{\leq k}$. We denote by $\text{LR}(k) - \text{Items}(G)$ the set of LR(k) items of G.

Remember that the notation $T^{\leq k}$ means $\cup_{i=0}^{k} T^i$, i.e., the set of all strings of length at most k on T.

As can be seen, we have augmented the notion of item with a *local Follows* u (of size at most k). The intuition behind an item of the form $A \to \alpha_1 \bullet \alpha_2,\ u$ is that:

- the parser is trying to reduce an A from the handle $\alpha_1\alpha_2$;
- it has already reduced α_1 on the top of the stack;
- this occurs in the context where A is followed by u. In other words, if $\alpha_2 = \varepsilon$, the look-ahead should be u, otherwise the run of the parser *cannot succeed.*

Example 6.14. Continuing the example of Figure 6.9 and Figure 6.10, the second item in state **1** should be $R \to L\bullet$, \$.

In order to compute an LR(k)-CFSM with such items, we need to redefine the initial state, and the closure operation.

For the initial state, we compute it by taking the closure (which we define hereunder) of the set of items:

$$\{S \to \bullet\alpha,\ \varepsilon \mid S \to \alpha \in P\}$$

where P is the set of rules of the grammar. That is, we take the closure of all the items of the form $S \to \bullet\alpha$ extended with the local Follow ε (and where S is, as usual, the start symbol).

Now, for the closure operation, assume we have an item of the form $A \to \alpha_1 \bullet B\alpha_2,\ u$. We will perform the closure by creating items based on all rules of the form $B \to \beta$. The local Follows will be computed on the basis of what we expect on the input after B. Since we start from the item $A \to \alpha_1 \bullet B\alpha_2,\ u$, we expect on the input the k first characters of α_2, which we can complete with u if need be. That is, we will have all the items of

Input: A set $I \subseteq \text{Items}(G)$ of LR(k)-items of some CFG
$G = \langle V, T, P, S\rangle$.
Output: The closure of I

Closure $\leftarrow I$;
repeat
  PrecClosure $\leftarrow$ Closure ;
  foreach $A \rightarrow \alpha_1 \bullet B\alpha_2,\ u \in$ Closure *s.t.* $B \in V$ **do**
    Closure $\leftarrow$
      Closure $\cup \left\{B \rightarrow \bullet\beta,\ u' \middle| B \rightarrow \beta \in P \text{ and } u' \in \text{First}^k(\alpha_2 \cdot u)\right\}$;
until PrecClosure = Closure;
return Closure ;

Algorithm 14: Computing the Closure of a set I of items for the LR(k) CFSM.

the form $B \rightarrow \bullet\beta,\ u'$, where $u' \in \text{First}^k(\alpha_2 \cdot u)$. Algorithm 14 gives the formalised algorithm for computing this closure operation.

Finally, the transitions of the CFSM are computed by simply propagating the local follows. That is, we redefine the successor function as:

$$\text{CFSMSucc}(I, a) = \left\{A \rightarrow \alpha_1 a \bullet \alpha_2,\ u \middle| A \rightarrow \alpha_1 \bullet a\alpha_2,\ u \in I\right\}.$$

Let us illustrate these notions by continuing our example:

Example 6.15. Let us build on the example of Figure 6.9 and Figure 6.10. Let us first compute the initial state of the LR(1) CFSM for the grammar of Figure 6.9. The kernel of the state will be the item $S' \rightarrow \bullet S\$,\ \varepsilon$. Then, we compute the closure of this item step by step:

- First, we add the items $S \rightarrow \bullet L = R,\ \$$ and $S \rightarrow \bullet R,\ \$$. The local Follows are obtained by computing:

$$\begin{aligned}\text{First}\big(\$ \cdot \varepsilon\big) &= \text{First}\big(\$\big)\\ &= \{\$\}.\end{aligned}$$

- Then, from the item $S \rightarrow \bullet L = R,\ \$$, we obtain the items $L \rightarrow \bullet * R,\ =$ and $L \rightarrow \bullet \mathsf{Id},\ =$.

- Next, from $S \rightarrow \bullet R,\ \$$ we add the item $R \rightarrow \bullet L,\ \$$.

- Finally, from this last item, we obtain two new items $L \rightarrow \bullet * R,\ \$$ and $L \rightarrow \bullet \mathsf{Id},\ \$$. Observe that the local Follows we have obtained in these items are different from the one computed before. In order to reduce the size of the representation of the CFSM, we will often merge items that differ only by the local Follows into sets of possible local Follows: we will thus write $L \rightarrow \bullet * R,\ \{=, \$\}$ and $L \rightarrow \bullet \mathsf{Id},\ \{=, \$\}$. For the sake of clarity, we will thus systematically represent the possible local Follows as *sets* even when they are singletons.

This state is displayed in Figure 6.11, along with its L-successor, state **1**. This shows that the conflict on state **1** has been lifted, since the local Follow for item $R \rightarrow L\bullet$ is $\$$.

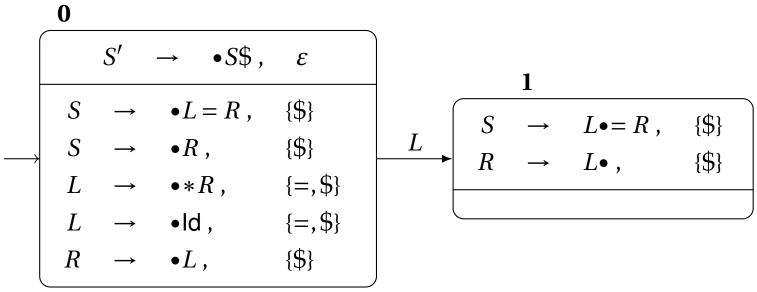


Figure 6.11: An excerpt of the LR(1) CFSM for the example grammar.

Example 6.16. For our second example, we will build the LR(1) CFSM for the grammar in Figure 6.7. We already know that this grammar is SLR(1), so we will be able to compare the SLR(1) and LR(1) CFSM for this grammar. The CFSM is given in Figure 6.12 and Figure 6.13. Observe that the states in these two figures are very similar, but that the local follows in the items are different. As an example, compare state **6** in Figure 6.12 and state **16** in Figure 6.13, where the difference lies in the first item only. This clearly shows that these local follows allow for a finer analysis, but this comes at the cost of the number of states.

LR(k) *Action table* Now that we can compute the LR(k) CFSM, let us see how we can exploit it in a parser. We will adapt the techniques we have introduced for SLR(1). There are mainly two cases to consider:

1. When a state s contains an item of the form $A \to \alpha \bullet a\beta,\ u$, where a is a terminal (i.e. the dot is directely followed by a terminal), then we must perform a *Shift*. This will occur in state s, and when the look-ahead is some element $y \in \mathrm{First}^k(a\beta u)$. Observe that we complete $a\beta$ with the *local Follow* that is associeted to the item, in order to obtaint the look-ahead.

2. When a state s contains an item of the form $A \to \alpha\bullet,\ u$, we need to perform a *Reduce* of the rule $A \to \alpha$. This action will occur in state s and when the look-ahead is u.

   For example, if we rely on the CFSM of Figure 6.12, in state **9**, we perform a *Reduce* when the look-ahead is either \$, + or $*$. Note that we do not readuce when the look-ahead is), although $) \in \mathrm{Follow}(\mathsf{Prod})$. The *Reduce* will occur on this look-ahead in state **14** (but, in this state, it will not occur on the look-ahead \$).

Remember that in '$A \to \alpha\bullet, u$', the element u is a word of at most k characters. In the CFSM, we adopt a compact notation like $A \to \alpha\bullet, \{w_1, w_2, \ldots, w_n\}$ to represent the n items $A \to \alpha\bullet,\ w_1 ,\ldots,\ A \to \alpha\bullet,\ w_n$.

It is worth comparing the rule for performaing a *Reduce* in the case of LR(1) to what we have done for SLR(1) (see Algorithm 12). In the case of SLR(1), the look-ahead we use is $\mathrm{Follow}(A)$. Here, we use u, that is, we use a *local* Follow instead of a *global* one.

We can now exploit those ideas to adapt Algorithm 12 to the LR(k) case. This new algorithm is given in Algorithm 15. Apart from the use of the local Follows that we have already discussed, the most notable addition to Algorithm 15 is the use of k characters of look-ahead. This means that the action table is now indexed by a state q and a word u of length *at most* k.

We are indexing by words of length *at most* k and not *exactly* k because the actual look-ahead available on the input might be shorter than k (for example when we reach the end of the input).

Example 6.17. We apply Algorithm 15 to the grammar in Figure 6.7. Its LR(1) CFSM is given in Figure 6.12 and Figure 6.13. Its LR(1) action table is in Table 6.3.

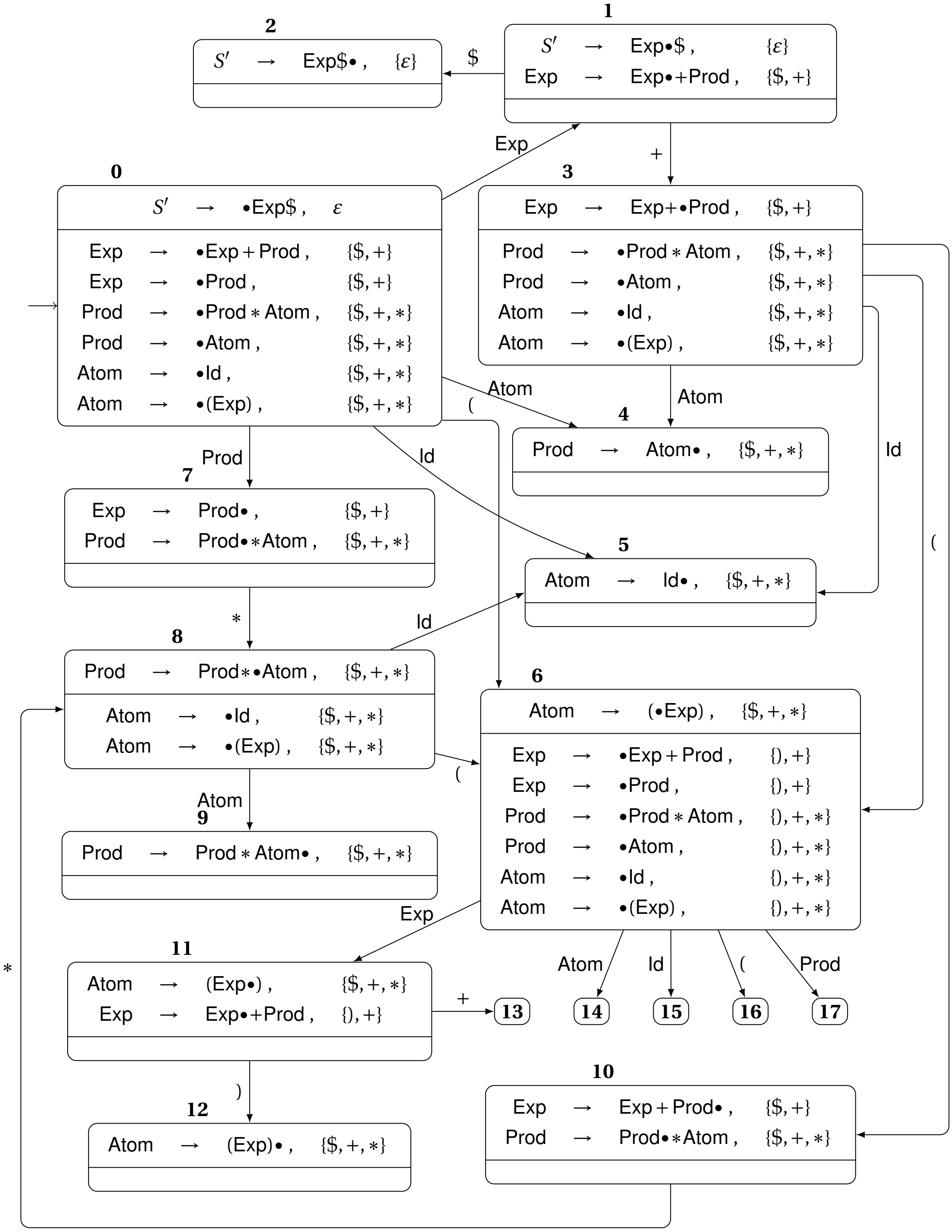


Figure 6.12: The LR(1) CFSM for the grammar for arithmetic expressions, first part. States **13** through **17** (and their successors) are given in the next figure.

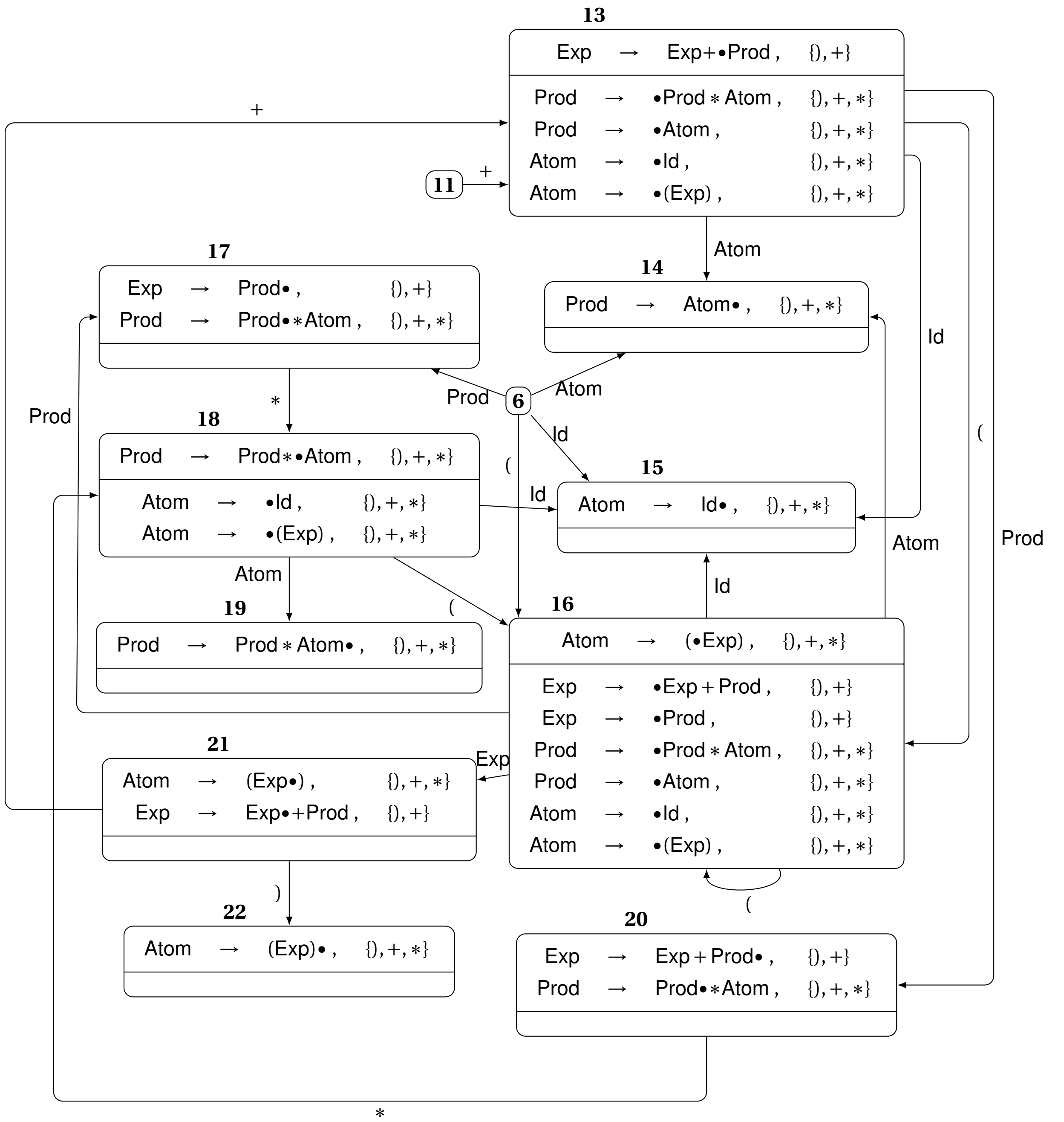


Figure 6.13: The LR(1) CFSM for the grammar for arithmetic expressions, (continued). States **6** and **11** are displayed on the previous figure.

Input: A CFG $G = \langle V, T, P, S\rangle$ whose rules are indexed from 1 to n;
and CFSM $(G) = \langle Q, \mathbf{0}, V \cup T, \delta, F\rangle$ where
$Q = \{\mathbf{0}, \mathbf{1} \ldots, \mathbf{m-1}, \emptyset\}$.
Output: LR(k) action table of G.

`/* Initialisation */`
foreach $u \in T^{\leq k}$ **do**
  $M[\emptyset, u] \leftarrow$ Error;
  foreach $q \in \{\mathbf{0}, \mathbf{1}, \ldots, \mathbf{m-1}\}$ **do**
    $M[q, u] \leftarrow \emptyset$;

`/* Populating the table */`
foreach $q \in \{\mathbf{0}, \mathbf{1}, \ldots, \mathbf{m-1}\}$ **do**
  if *q contains an item of the form* $S \to \alpha \bullet, \varepsilon$ **then**
    $M[q, \varepsilon] \leftarrow M[q, \varepsilon] \cup \{\text{Accept}\}$;
  foreach *item in q that is of the form* $A \to \alpha \bullet, u$ *with* $A \neq S$ **do**
    $M[q, u] \leftarrow M[q, u] \cup \{j\}$ where rule number j is $A \to \alpha$;
  foreach *item in q that is of the form* $A \to \alpha_1 \bullet a\alpha_2, u$ *with* $a \in T$ **do**
    foreach $y \in \text{First}^k(a\alpha_2 u)$ **do**
      $M[q, y] \leftarrow M[q, y] \cup \{\text{Shift}\}$;

return M ;

Algorithm 15: The algorithm to compute the LR(k) action table of a CFG G using CFSM (G).

M	+	$*$	Id	(	)	\$	ε
0			S	S			
1	S					S	
2							A
3			S	S			
4	5	5				5	
5	6	6				6	
6			S	S			
7	3	S				3	
8			S	S			
9	4	4				4	
10	2	S			2		
11	S				S		
12	7	7			7		
13			S	S			
14	5	5			5		
15	6	6			6		
16			S	S			
17	3	S			3		
18			S	S			
19	4	4		4			
20	2				2		
21	)			S			
22	7	7			7		

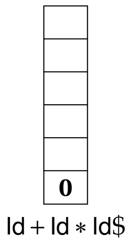
Table 6.3: The LR(1) action table for our simple grammar of expressions.

6.5.2 *Running the* LR(k) *parser*

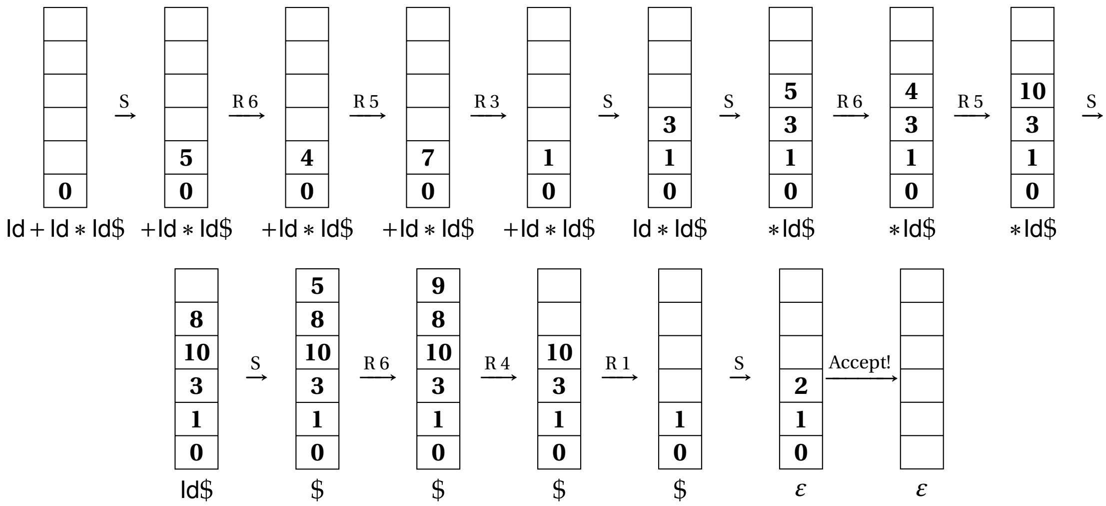
Now that we have managed to build the action table of an LR(k) parser, we can discuss how it *runs* on a given input word. Actually, the run of an LR(1) parser will not be different from that of an SLR(1)1 parser: the origin of the action table and of the CFSM does not matter for Algorithm 13 to run. All this algorithm needs is an action table indexed by states (for the rows) and look-aheads (for the columns), and a way to compute the successor states of the CFSM (which can be given as a table, as well, and is implicitly represented by the CFSM itself). While we have restricted our presentation of Algorithm 13 to the case $k = 1$ (i.e., and LR(1) parser), it is straightforward to adapt it to the case where $k > 1$: it suffices to replace the look-ahead ℓ with a word of size (at most) k.

We can thus close our discussion on the construction of LR(k) parsers by giving an example of run.

Example 6.18. We repeat Example 6.11, this time using the action table of the LR(k) parser, and its associated LR(k) CFSM (see Example 6.15 and Example 6.17). We consider the input word $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}\$$, which is in the language of the grammar (see Figure 6.7). As always, we start our run in the following configuration:



In this configuration, the look-ahead is Id, the current state if **0** and the action table tells us to perform a *Shift*, which leads to state **5** in the CFSM, *etc.* The full run is then:



As can be seen, this run is very similar to the run of the SLR(1) parser. This should not be a surprise, since both parsers build the same *rightmost derivation.*

6.5.3 LR(k) *grammars*

In the previous section, we have managed to build LR(k) parsers, which are bottom-up parsers that use k characters of look-ahead. As their name suggests, they are some sort of bottom-up relatives to the LL(k) parsers. In the case of LL(k) parsers we had identified the class of LL(k) *grammars* for which we had the guarantee that their corresponding LR(k) parser is deterministic (see Definition 5.10).

In this section, we will consider a similar definition of LR(k) *grammars*, which captures exactly the class of all grammars for which the LR(k) parser is deterministic (i.e., has no conflict in its action table).

Remember the remark on page 178? There we are...

As for LL(k), the formal definition might seem daunting, so we will start with some intuition. We will proceed as for LL(k): we will try and identify situations that will be 'confusing' for the LR(k) parser, and the definition will basically state that such situations are not allowed.

Let us assume that we have a first rightmost derivation starting with:

$$S \Rightarrow^* \gamma A x \Rightarrow \gamma\alpha x \Rightarrow \cdots$$

In this prefix we have pinpointed the derivation of rule $A \to \alpha$. Since this is a *rightmost* derivation, we have the guarantee that $x \in T^*$, i.e. x contains terminals only. On the other hand α and γ are in $(T \cup V)^*$: they can contain terminals and variables. In this derivation, α is the *handle* of $A \to \alpha$, and the task of the parser will be to correctly identify this handle and this rule to perform the reduction.

Thus, in order to build a situation that *confuses* the parser, we have to remember what is the information that the parser uses to take a decision. From the discussion before, we know that it uses two pieces of information: the current CFSM state and the look-ahead. Where can we find this information in the derivation? To answer this question, we must find out what is on the stack at the moment where the parser must reduce $A \to \alpha$, and what remains on the input. Since α is reduced as A, α must necessarily be on the top of the stack, so the stack content is $(\gamma\alpha)^R$; and x remains on the input (see Proposition 6.2). Then, *when the Reduce of $A \to \alpha$ must occur*:

We have $(\gamma\alpha)^R$ and not $\gamma\alpha$ because we have taken the convention to read the stack from the top to the bottom.

1. the current CFSM state is the one which is reached when the CFSM reads the mirror image of the stack content, i.e. $\gamma\alpha$. Since the CFSM is deterministic, we can identify the state with $\gamma\alpha$ (i.e., there will be one and only one state reached when reading $\gamma\alpha$, so we can abuse notations slightly and say that $\gamma\alpha$ 'is the state'); and
2. the look-ahead contains the k first symbols of what remains on the input (if they exist), i.e. $\text{First}^k(x)$.

Now, let us build another derivation that will create a situation where the parser is 'confused', i.e. a derivation where the parser cannot tell the difference between the reduce of $A \to \alpha$ described above, and another reduce. Let us thus assume that we have now two rightmost derivations (we recall the previous one for the sake of comparison):

$$S \Rightarrow^* \gamma A x \Rightarrow \gamma\alpha x \Rightarrow \cdots$$
$$S \Rightarrow^* \delta B y \Rightarrow \delta\beta y \Rightarrow \cdots$$

In the latter derivation, the parser must reduce $B \rightarrow \beta$. This occurs when $(\delta\beta)^R$ is on the stack and y remains on the input. So the situation where the parser gets 'confused' between the *Reduce* of $A \rightarrow \alpha$ and $B \rightarrow \beta$ is when the CFSM state (or equivalently the stack content) is the same in both cases, and when the look-aheads are the same. This occurs when:

$$\delta\beta y = \gamma\alpha x'$$

(i.e. the stack content is the same) for some x' s.t.:

$$\text{First}^k(x) = \text{First}^k\left(x'\right)$$

(the look-ahead is the same).

Then, the definition of LR(k) grammar says essentially that such a situation cannot occur. So, if the above conditions are met, we must have $\gamma = \delta$, $A = B$ and $x' = y$ for the grammar to be LR(k). Here it is[13]:

Observe that this is a *semantic* definition that talks about all the pairs of derivations (like the definition of LL(k)).

[13] Donald E. Knuth. On the translation of languages from left to right. *Information and Computation (formerly known as Information and Control)*, 8:607–639, 1965. DOI: 10.1016/S0019-9958(65)90426-2

Definition 6.19 (LR(k) grammar)**.** A CFG $G = \langle P, T, V, S\rangle$ is LR(k) iff for all pairs of rightmost derivations:

$$S \Rightarrow^* \gamma A x \Rightarrow \gamma\alpha x \Rightarrow^* w_1 x$$
$$S \Rightarrow^* \delta B y \Rightarrow \delta\beta y \Rightarrow^* w_2 y$$

s.t.: (i) $\delta\beta y = \gamma\alpha x'$; and (ii) $\text{First}^k(x) = \text{First}^k\left(x'\right)$, we have: $\gamma A x' = \delta B y$.

Testing that a given grammar *is* LR(k) (for a given k) can be challenging with this definition. The best way is still to build the LR(k) parser and check that it is deterministic.

While testing whether a grammar is LR(k) for a given k (for example: 'is this grammar LR(5)?') is clearly decidable, KNUTH shows in his paper cite above that testing whether *there exists some* k s.t. a given grammar is LR(k) is an undecidable problem.

Let us illustrate Definition 6.19 with an example:

Example 6.20. Consider the simple grammar in Figure 6.14. It is clearly not LR(0), since there is a *Shift*/*Reduce* conflict in state **1** of the CFSM (see Figure 6.15).

(1)	S	→	a
(2)		→	ab

Figure 6.14: A simple grammar which is not LR(0).

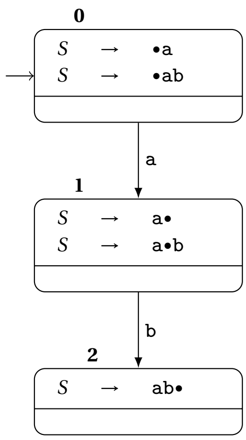


Figure 6.15: The CFSM of the previous grammar.

Let us show why it does not satisfy Definition 6.19 for $k = 0$. To do so, we must find a pair of derivations that do not satisfy the conditions of the definition. In our case, this will be pretty straightforward, since the grammar has two derivations only:

$$S \Rightarrow \texttt{a}$$
$$S \Rightarrow \texttt{ab}$$

Matching these derivations with the notations of Definition 6.19, we obtain:

$$A = B = S$$
$$\alpha = \texttt{a}$$
$$\beta = \texttt{ab}$$
$$\gamma = \delta = \varepsilon$$
$$x = y = \varepsilon$$

Thus, $\delta\beta y = \texttt{ab}$. Since, $\gamma\alpha = \texttt{a}$, we have $\delta\beta y = \gamma\alpha x'$ with $x' = \texttt{b}$. So, all the conditions of the definition are satisfied. In particular, note that

Intuitively, $\gamma A x'$ is the situation where we have the stack content $(\gamma A)^R$ from the first derivation but with the input x' which is a suffix of the remaining input in the second derivation, and the parser should be able to tell the difference between these two situations and take the right decision. In our example, when the parser decides to reduce $S \rightarrow \texttt{a}$ in the first derivation, it should still do this even if we add $x' = \texttt{b}$ to the input,

$\text{First}^0(x) = \emptyset = \text{First}^0(x')$ (we have no look-ahead in LR(0)). In this case, the definition tells us that we should have $\gamma A x' = \delta B y$, however:

$$\gamma A x' = A\mathtt{b}$$
$$\delta B y = A$$

So these two values are clearly different.

The grammar, however, *is* LR(1) since the look-ahead allows us to decide whether to *Shift* or to *Reduce* in state **1** of the CFSM. This can also be checked with the definition: for $k = 1$, there is no way to arrange the values α, β, γ, δ, A, B, x, x' and y to satisfy the conditions of the definition and have $\gamma A x' \neq \delta B y$. In particular, if we re-use the values as above, we fail to satisfy the condition $\text{First}^1(x) = \text{First}^1(x')$, since $\text{First}^1(x) = \{\varepsilon\}$ and $\text{First}^1(x') = \{\mathtt{b}\}$.

6.6 LALR(k) *parsers*

We close this section on bottom-up techniques by discussing (briefly) a last family of parsers that are essential in practice since they are the ones that are actually implemented in tools such as `yacc`, `bison` and `cup`.

One of the main criticism of LR(k) parsers is that their associated CFSM can be very big (because the number of potential items is multiplied by the number of potential local look-aheads), hence, the action and successor tables needed by the parser can become unmanageable. As a consequence, researchers realised quickly that some techniques were needed to make LR(k) parsers practical.

One such solution was introduced[14] by DeRemer along with SLR(1), under the name of LALR(k) parsers. LALR(k) stands for 'Look-Ahead LR(k)' parsers. While DeRemer introduced the idea of LALR(k) parsers, a practical algorithm to build them (efficiently) was given a few years later[15] only. Since then, several algorithms have been presented[16].

The idea of the LALR(k) parser is based on an observation we have already made when considering LR(k) parsers: in LR(k) parsers, many states are similar in the sense that they contain the same LR(0) items but with different 'local follows'. For example, in the CFSM of Figure 6.12 and Figure 6.13, states **6** and **16** only differ on the local follows. So, a natural idea consists in 'merging' those states, i.e., computing the union of the sets of items of those states, and building an action table and a successor table based on this new automaton. Of course, by doing so, we are losing some precision, we cannot expect that LR(k) grammars will also be LALR(k), but if the resulting parser is still deterministic, we have obtained a CFSM which has the same size as the LR(0) CFSM, while using look-aheads for lifting ambiguities.

[14] Franklin Lewis DeRemer. *Practical Translators for LR(k) Languages.* PhD thesis, Massachusetts Institute of Technology, 1969. URL https://web.archive.org/web/20130819012838/http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-065.pdf

[15] W. R. LaLonde, E. S. Lee, and J. Horning. An LALR(k) parser generator. In *Proceedings of IFIP congress*, pages 151–153. Elsevier Science, New York, 1971

[16] Thomas J. Penello and Franklin Lewis DeRemer. Efficient computation of LALR(1) look-ahead sets. *ACM SIGPLAN Notices*, 39(4), 2004. DOI: 10.1145/69622.357187

6.6.1 *Building the* LALR(k) *parser from the* LR(k) *CFSM*

More formally, we will first explain how to obtain an LALR(k) parser from the LR(k) CFSM of a given grammar. We start with the definition of the *heart* of a CFSM state

Definition 6.21 (Heart of a CFSM state)**.** Let $\{A_1 \to \alpha_1 \bullet \beta_1,\ u_1; \ldots; A_n \to \alpha_n \bullet \beta_n,\ u_n\}$ be a set of LR(k) items, i.e. a state of some LR(k) CFSM. Then, its *heart*[17] is the set $\{A_1 \to \alpha_1 \bullet \beta_1; \ldots; A_n \to \alpha_n \bullet \beta_n\}$ of corresponding LR(0) items.

[17] Some references call it the *core* of the state.

Then, we say that two states of the LR(k) automaton are *equivalent* if they have the same heart:

Definition 6.22 (LALR(k) equivalence)**.** Two states q_1 and q_2 of some LR(k) CFSM are *equivalent* iff they have the same heart. We denote this by $q_1 \equiv q_2$.

One can check that $\equiv$ is indeed an *equivalence relation.* Hence, it partitions the set of states of the LR(k) CFSM into equivalence classes. For all states q of the LR(k) CFSM, we denote by $[q]$ the equivalence class that contains it. This partition will help us to build a so-called LALR(k) CFSM as we are about to see. But let us first have a look at an example for $\equiv$.

Thus, the *equivalence classe* $[q]$ of a state q is set of all states q' that are equivalent to q, i.e. such that $q' \equiv q$.

Example 6.23. Let us consider again the LR(1) CFSM of Figure 6.12 and Figure 6.13. Here is the list of equivalence classes of its states:

$$
\begin{aligned}
[\mathbf{0}] &= \{\mathbf{0}\}\\
[\mathbf{1}] &= \{\mathbf{1}\}\\
[\mathbf{2}] &= \{\mathbf{2}\}\\
[\mathbf{3}] = [\mathbf{13}] &= \{\mathbf{3},\mathbf{13}\}\\
[\mathbf{4}] = [\mathbf{14}] &= \{\mathbf{4},\mathbf{14}\}\\
[\mathbf{5}] = [\mathbf{15}] &= \{\mathbf{5},\mathbf{15}\}\\
[\mathbf{6}] = [\mathbf{16}] &= \{\mathbf{6},\mathbf{16}\}\\
[\mathbf{7}] = [\mathbf{17}] &= \{\mathbf{7},\mathbf{17}\}\\
[\mathbf{8}] = [\mathbf{18}] &= \{\mathbf{8},\mathbf{18}\}\\
[\mathbf{9}] = [\mathbf{19}] &= \{\mathbf{9},\mathbf{19}\}\\
[\mathbf{10}] = [\mathbf{20}] &= \{\mathbf{10},\mathbf{20}\}\\
[\mathbf{11}] = [\mathbf{21}] &= \{\mathbf{11},\mathbf{21}\}\\
[\mathbf{12}] = [\mathbf{22}] &= \{\mathbf{12},\mathbf{22}\}
\end{aligned}
$$

Then, based on this equivalence relation, and from the LR(k) CFSM, we can define the LALR(k) CFSM. To this end, we need to define the new set of states and the new transition relation of this automaton:

Note that the set of states of the LALR(k) CFSM is *not* the set of equivalence classes. Otherwise, the states of the new automaton would be sets of states of the LR(k) CFSM, i.e. *sets of sets of items*; while we want the states of the new automaton to bet *sets of items* to obtain a CFSM.

1. For each equivalence class of $\equiv$, we will have one state in the LALR(k) CFSM. More precisely, for each equivalence class $[q]$, there will be a new state which contains all the items that are in all the states in $[q]$. In other words, the new state (which we denote $S_{[q]}$) is:

$$S_{[q]} = \bigcup_{q' \in [q]} q'.$$

2. Then, we need to adapt the transition relation. This is easy, if we make the following observation. Let $[q]$ be some equivalence class, and let

$q' \in [q]$ be one of these states. By definition of the 'merging' of LR(k) states into LALR(k) states, whenever there is an item of the form

$$A \to \alpha \bullet \beta,\ u$$

in q', then there is an item of the form

$$A \to \alpha \bullet \beta,\ v$$

in all other states q'' of the equivalence class (i.e., with the same LR(0) item, but a different context). This means that the transitions from all states q'' of the equivalence class will be *similar* in the following sense. Whenever there is, in the LR(k) CFSM, a transition from q_1, labeled by η and going to q_2, then there is also a transition labeled by η from all $q_1' \in [q_1]$, and this transition leads to a state q_2' which is equivalent to q_2 (i.e. $q_2' \in [q_2]$). It is thus safe to have an η-labeled transition in the LALR(k) CFSM from $S_{[q_1]}$ to $S_{[q_2]}$. Remark that this construction preserves the determinism of the automaton.

Example 6.24. Let us illustrate this on our running example. Consider for example the equivalence class $\{\mathbf{8}, \mathbf{18}\}$. Observe that the sets of corresponding LR(0) items of these two states are the same, which is not surprising since this is the definition of the heart:

$$\Big\{\mathsf{Prod} \to \mathsf{Prod} * \bullet \mathsf{Atom}, \mathsf{Atom} \to \bullet \mathsf{Id}, \mathsf{Atom} \to \bullet (\mathsf{Exp})\Big\}.$$

This is why we have identically-labeled transitions from both states: one labeled by Atom, one by Id and one by (. Moreover, these respective transitions reach states that are $\equiv$-equivalent. For example, reading Atom from state **18** leads to state **19**; while reading Atom from **8** leads to **9**, with $\mathbf{19} \equiv \mathbf{9}$.

Based on this discussion, we are now ready to formally define the LALR(k) CFSM from the LR(k) CFSM:

Definition 6.25 (LALR(k) CFSM)**.** Let $G = \langle V, T, P, S\rangle$ be a CFG, and let $\big\langle Q, V \cup T, \delta, q_0, Q \setminus \{\emptyset\}\big\rangle$ be its LR(k) CFSM. Then, the LALR(k) CFSM is the DFA $\big\langle Q', V \cup T, \delta', q_0', Q' \setminus \{\emptyset\}\big\rangle$ where:

- $Q' = \{S_{[q]} \mid q \in Q\}$, where, for all $q \in Q$:

$$S_{[q]} = \bigcup_{q' \in [q]} q';$$

- $q_0' = S_{[q_0]}$; and
- for all $S_{[q]} \in Q'$, for all $a \in V \cup T$:

$$\delta'(S_{[q]}, a) = S_{[q']} \text{ iff there are } q_1 \in q, q_2 \in q' \text{ s.t. } \delta(q_1, a) = q_2.$$

We compute $S_{[q]}$ for all states q in Q, however note that, unless all equivalence classes are singleton, there will be at least two states q_1 and q_2 in Q s.t. $S_{[q_1]} = S_{[q_2]}$. That is not a problem for the definition of Q', since it is a *set*, and thus each element appears at most once.

Once the LALR(k) CFSM is built, we build the action table as in the case of an LR(k) parser, and the runs of the parser follow this action table similarly.

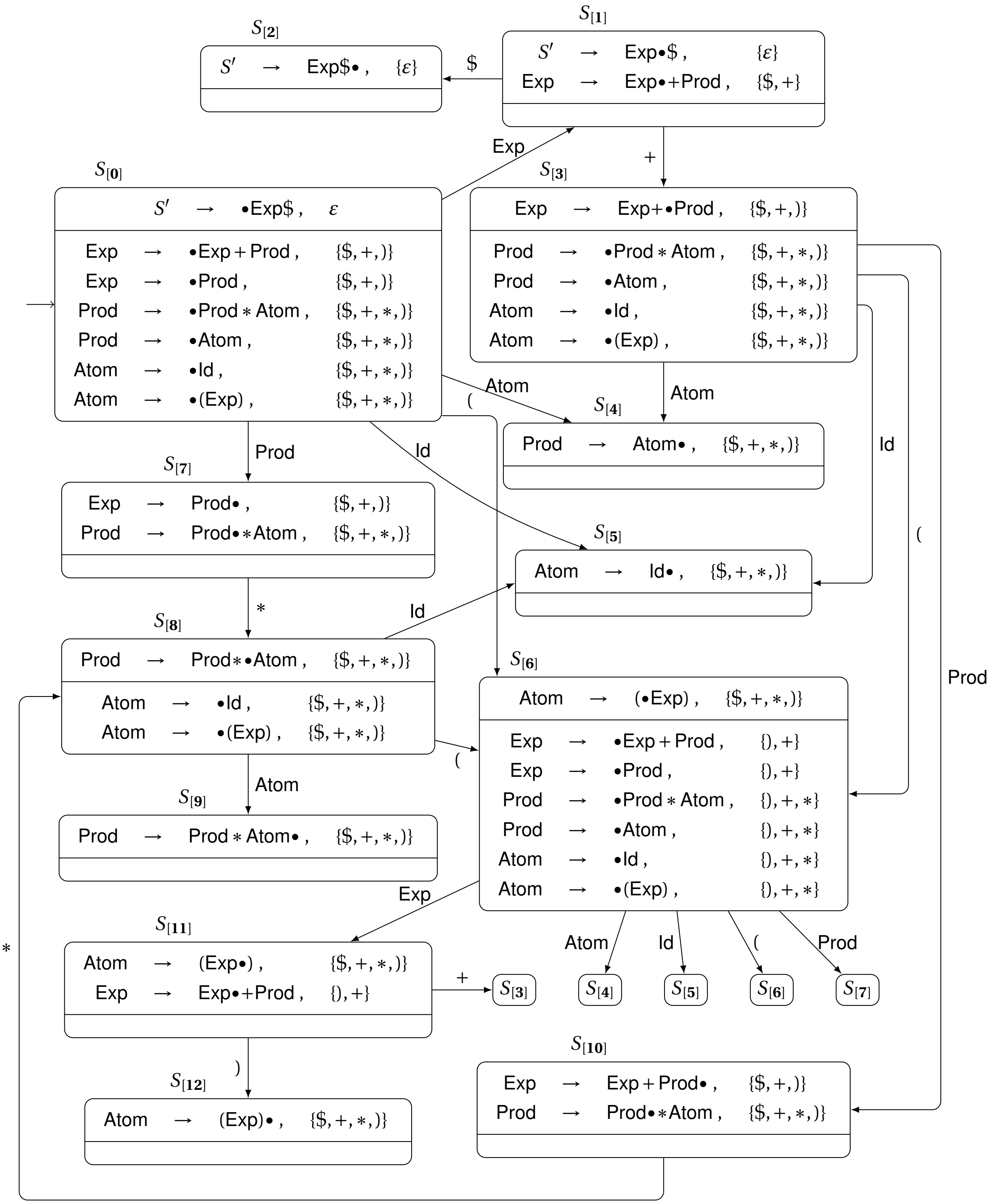


Figure 6.16: The LALR(k) CFSM for the grammar for arithmetic expressions.

Example 6.26. Let us close this discussion by building the LALR(k) parser for our running example. We start by building the LALR(k) CFSM. It is given in Figure 6.16. We have purportedly chosen to keep the same presentation as in Figure 6.12, to allow the reader to compare them.

Then, the action table computed from this CFSM is given in Table 6.4.

M	+	$*$	Id	(	)	\$	ε
$S_{[\mathbf{0}]}$			S	S			
$S_{[\mathbf{1}]}$						S	
$S_{[\mathbf{2}]}$							A
$S_{[\mathbf{3}]}$			S	S			
$S_{[\mathbf{4}]}$	5	5			5	5	
$S_{[\mathbf{5}]}$	6	6			6	6	
$S_{[\mathbf{6}]}$			S	S			
$S_{[\mathbf{7}]}$	3	S			3	3	
$S_{[\mathbf{8}]}$			S	S			
$S_{[\mathbf{9}]}$	4	4			4	4	
$S_{[\mathbf{10}]}$	2	S			2	2	
$S_{[\mathbf{11}]}$	S				S		
$S_{[\mathbf{12}]}$	6	6			6	6	

Table 6.4: The LALR(1) action table for the simple grammar of expressions.

6.6.2 *Some remarks on the* LALR(k) *parser*

We close this section on LALR(k) parsers by two remarks. First of all, since our goal was to obtain a parser which is more efficient than LR(0) but more compact than LR(k), we should ask ourselves whether it is indeed the case. As a matter of fact, it is easy to see that the size (in terms of number of states) of the LALR(k) CFSM is the same as the size of the LR(0) CFSM.

Proposition 6.4. *For all CFG, the* LALR(k) *CFSM has the same number of states than the* LR(0) *CFSM.*

Proof. This property stems directly from the definition of LALR(k) states. Let us first consider the states of the LR(k) CFSM, from which the states of the LALR(k) CFSM are extracted. We can observe that the set of hearts of the LR(k) CFSM is exactly the set of LR(0) states. This can be seen by comparing the respective effects of the closure operation in the cases of LR(0) (see Algorithm 9) and LR(k) (Algorithm 14). More precisely, assume that, when applied to an item $A \rightarrow \alpha \bullet \beta$, the LR(0) closure produces $A_1 \rightarrow \alpha_1 \bullet \beta_1$;...; $A_n \rightarrow \alpha_n \bullet \beta_n$. Then, when applied to an item $A \rightarrow \alpha \bullet \beta, u$ with the same heart, the LR(k) closure will produce a set of items $A_1 \rightarrow \alpha_1 \bullet \beta_1\ u_1$;...; $A_n \rightarrow \alpha_n \bullet \beta_n\ u_n$ with the same heart as well. Thus, the set of hearts of the LR(k) CFSM is exactly the set of LR(0) states. However, the set of LALR(k) states has the same size as the set of LR(k) hearts, by construction. Hence our proposition. □

Moreover, since the SLR(1) parser is essentially the LR(0) parser with some look-ahead, we also have that:

Corollary 6.5. *For all CFG, the* LALR(k) *parser has the same number of states as the* SLR(1) *parser.*

So, from these two results, it seems that we have managed to obtain a pretty good reduction in the number of states that the LALR(k) parser must use. Now, the question of the usefulness of LALR(k) remains open, as we haven't shown yet that there are some grammars that can be parsed with LALR(k) but not LR(0) or SLR(1). We will address this in more details in Section 6.6. Observe, however that, in the example we have discussed above, the obtained LALR(1) parser happens to be the same as the SLR(1) parser, but this is not always the case.

Finally, we observe that the technique we have described here to build an LALR(k) parser is pretty inefficient since it requires to first build the LR(k) parser, then merge states. More efficient techniques, that perform this merge on-the-fly exist[18], but we will not discuss them here. The description of such a technique can be found in Section 4.7.5 of the 'Dragon Book[19]'.

[18] T. Anderson, J. Eve, and J. Horning. Efficient LR(1) parsers. *Acta Informatica*, 2: 2–39, 1973. DOI: 10.1007/BF00571461

[19] A. Aho, M. Lam, R. Sethi, and Ullman J. *Compilers: Principles, Techniques, & Tools.* Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007

6.7 The bottom-up hierarchy of grammars

Now that we have such a wide range of (bottom-up) parsers at our disposal, one might wonder which one to chose from? Clearly, increasing the size of the look-ahead increases the number of grammars that a parser can recognise deterministically; but we also know that such an increase has a cost in the complexity of the parser. So, it seems that choosing a parser will be a trade-off between its expressive power and its complexity.

As we did in the case of LL(k) grammars (see Section 5.4.2 and Section 5.4.3), we will now establish bottom-up hierarchies of grammars and languages. We will first compare the *families of grammars* that can be parsed deterministically with the different bottom-up techniques we have seen, i.e., establish a *syntactic hierarchy*. However, the fact that a grammar G is, for instance, LR($k+1$) but not LR(k) does not necessarily mean that $L(G)$ *cannot* be recognised by an LR(k) parser, because there might be an LR(k) grammar G' with the same language... Thus, in Section 6.8, we will be comparing the *families of languages* that can be recognised by the different parsers that we have at our disposal, i.e. to establish a *semantic hierarchy*. Finally, in Section 6.9 we will compare the top-down and bottom-up hierarchies.

We will start this discussion with the *syntactic* comparison. We have already defined LL(k), strong LL(k) and LR(k) grammars (see Definition 5.10, Definition 5.12 and Definition 6.19). We can further define SLR(1) and LALR(k) grammars:

One could discuss whether the LR(k) and LL(k) classes are truly *syntactic*. Clearly, the definition of *strong* LL(k) is purely syntactic since it concerns the rules of the grammar. The definitions of LL(k) and LR(k) are more *semantic* since they constrain the *derivations* that the grammar generates. However, those definition do no constraint the ultimate semantic object that a grammar generates, i.e. its language. So, it is fair to say that comparing classes of *grammars* (which are syntactic objects) is a syntactic comparison, while comparing the *languages* that these grammars generate is *semantic*.
We will not venture further into this (pseudo-)philosophical discussion... Let us just quote this beautiful piece of coffee-machine wisdom that we have overheard during the break of a scientific conference: 'You know, one man's semantic is another man's syntax'...

Definition 6.27 (SLR(1) and LALR(k) grammars)**.** A CFG is SLR(1) iff the SLR(1) parser generated for this grammar is deterministic. A CFG is LALR(k) iff the LALR(k) parser generated for this grammar is deterministic.

Now, let us compare these different families of grammars. One can establish the following inclusions:

Theorem 6.6. *The following (strict) inclusions hold:*

$$\text{LR}(0) \subsetneq \text{SLR}(1) \subsetneq \text{LALR}(1) \subsetneq \text{LR}(1) \subsetneq \text{LR}(2) \subsetneq \cdots \subsetneq \text{LR}(k) \subsetneq \cdots$$

Proof. We look at all these inclusions one after the other:

- LR(0) $\subsetneq$ SLR(1). We know that all LR(0) grammars are SLR(1) grammars, since SLR(1) is an extension of LR(0) with a look-ahead. The strict inclusion stems from Example 6.10, which shows that the grammar in Figure 6.7 is SLR(1) but not LR(0).

- SLR(1) $\subsetneq$ LALR(1). The fact that SLR(1) $\subseteq$ LALR(1) stems from the definitions of SLR(1) and LALR(1). Both have states with the same hearts, which are the LR(0) states by construction, but the contexts are more precise in the case of LALR(1) since they are unions of local Follows. Hence, SLR(1) does not offer more opportunities to lift conflicts than LALR(1), so all SLR(1) grammars are also LALR(1).

  To separate strictly LALR(1) and SLR(1), we consider the grammar in Figure 6.17 (adapted from exercise 20, Chapter 3.5 of SEIDL, WHILHELM and HACK's book[20]). It is easy to check that it is LALR(1) but not SLR(1). Indeed, the state that is reached after reading d in the LR(0) CFSM is:

S	$\rightarrow$	d•c
A	$\rightarrow$	d•

  and SLR(1) is not able to lift this Shift/Reduce conflict since c $\in$ Follow(A). However, in the LALR(1) CFSM, this state becomes:

S	$\rightarrow$	d•c ,	$\{\varepsilon\}$
A	$\rightarrow$	d• ,	{a}

(1)	S	$\rightarrow$	Aa
(2)		$\rightarrow$	bAc
(3)		$\rightarrow$	dc
(4)	A	$\rightarrow$	d

Figure 6.17: An LALR(1) grammar that is not SLR(1) and not LL(1) but LL(2).

[20] Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. *Compiler Design, Syntactic and Semantic Analysis*. Springer-Verlag, 2013. ISBN 978-3-642-17539-8. DOI: 10.1007/978-3-642-17540-4

which lifts the conflict (and no other conflicts exist in the LALR(1) automaton).

- LALR(1) $\subsetneq$ LR(1). The fact that LALR(1) $\subseteq$ LR(1) stems from the construction of LALR(1): if a grammar can be parsed deterministically by an LALR(k) parser, it will also be the case by an LR(k) parser since the latter is a refinement of the former. To prove that the inclusion is strict, all we need is to exhibit an example of grammar that is LR(1) but not LALR(1). Such a grammar is given in Figure 6.18 and can be found as Example 4.58 in the 'Dragon book[21]'.

  In particular, building the LR(1) CFSM creates the two following states:

$$\begin{array}{|llll|} A & \rightarrow & \mathtt{c}\bullet, & \{\mathtt{d}\} \\ B & \rightarrow & \mathtt{c}\bullet, & \{\mathtt{e}\} \\ \hline \end{array} \qquad \begin{array}{|llll|} A & \rightarrow & \mathtt{c}\bullet, & \{\mathtt{e}\} \\ B & \rightarrow & \mathtt{c}\bullet, & \{\mathtt{d}\} \\ \hline \end{array}$$

  which have no conflict since the look-ahead always allows one to decide which Reduce to perform. However, they have the same heart, and their merge in the LALR(1) CFSM will yield a state with a Reduce/Reduce conflict:

$$\begin{array}{|llll|} A & \rightarrow & \mathtt{c}\bullet, & \{\mathtt{d},\mathtt{e}\} \\ B & \rightarrow & \mathtt{c}\bullet, & \{\mathtt{d},\mathtt{e}\} \\ \hline \end{array}$$

- For all k: LR(k) $\subsetneq$ LR($k+1$). The fact that LR(k) $\subseteq$ LR($k+1$) is trivial, since having a longer look-ahead allows one to parse more grammars deterministically. Then, for all $k \geq 1$, the grammar in Figure 6.19 is LR($k+1$) but not LR(k). Indeed, after reading a, its LR(k) CFSM reaches the state:

$$\begin{array}{|llll|} A & \rightarrow & \mathtt{a}\bullet, & \{\mathtt{b}^k\} \\ B & \rightarrow & \mathtt{a}\bullet, & \{\mathtt{b}^k\} \\ \hline \end{array}$$

  which clearly contains a Reduce/Reduce conflict. This conflict is lifted with an additional symbol of look-ahead (and it is easy to check that the other states contain no conflit):

$$\begin{array}{|llll|} A & \rightarrow & \mathtt{a}\bullet, & \{\mathtt{b}^k\mathtt{c}\} \\ B & \rightarrow & \mathtt{a}\bullet, & \{\mathtt{b}^k\mathtt{d}\} \\ \hline \end{array}$$

□

[21] A. Aho, M. Lam, R. Sethi, and Ullman J. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007

(1)	S	$\rightarrow$	$\mathtt{a}A\mathtt{d}$
(2)		$\rightarrow$	$\mathtt{b}B\mathtt{d}$
(3)		$\rightarrow$	$\mathtt{a}B\mathtt{e}$
(4)		$\rightarrow$	$\mathtt{b}A\mathtt{e}$
(5)	A	$\rightarrow$	$\mathtt{c}$
(6)	B	$\rightarrow$	$\mathtt{c}$

Figure 6.18: An LR(1) grammar that is not LALR(1) and not LL(1).

(1)	S	$\rightarrow$	$A\mathtt{b}^k\mathtt{c}$
(2)		$\rightarrow$	$B\mathtt{b}^k\mathtt{d}$
(3)	A	$\rightarrow$	$\mathtt{a}$
(4)	B	$\rightarrow$	$\mathtt{a}$

Figure 6.19: An LR($k+1$) grammar that is not LR(k) and not LL($k+1$).

Finally, we address a final question regarding the hierarchy of LR(k) languages: since the sequence LR(0), LR(1), ..., LR(k), ... is growing, one could think that each CFG can fit snuggly in one of those classes. It turns out that this is not the case. Clearly, any *ambiguous* grammar will not be

Recall that an ambiguous CFG is one that admits at least two different parse trees on a given word, hence, at least two different rightmost derivations on this word.

LR(k) since the parser cannot decide which parse tree to use to produce the rightmost derivation, hence it will necessarily be non-deterministic. However, there are unambiguous CFG which are not LR(k) for any k. Here is an example:

Example 6.28. This example has been proposed by KNUTH in his original paper[22] on LR(k), it is given in Figure 6.20.

To be convinced that this grammar is not LR(k) for any k, we need to consider its accepted language, which is $\{\texttt{a}\texttt{b}^{2n+1}\texttt{c} \mid n \geq 0\}$. That is, the grammar accepts all the words of the form $\texttt{abb}\cdots\texttt{bc}$ where the number of b's is odd. To do so, the grammar relies on the recursive rule $A \rightarrow \texttt{b}A\texttt{b}$, as shown in the following derivation:

$$S \Rightarrow \texttt{a}A\texttt{c} \Rightarrow \texttt{ab}A\texttt{bc} \Rightarrow \texttt{abb}A\texttt{bbc} \Rightarrow \texttt{abbb}A\texttt{bbbc} \Rightarrow \texttt{abbb}\underline{\texttt{b}}\texttt{bbbc}$$

So, the crucial difficulty here will be for the parser to decide when to Reduce the 'middle b' (the one that is underlined) as A.

Assume the word to be accepted is $\texttt{ab}^{2n+1}\texttt{c}$ for some n. Then, when this particular Reduce must happens, an LR(k) parser 'sees' the look-ahead $\mathrm{First}^k(\texttt{b}^n\texttt{c})$, and needs to 'see' the c in order to realise that it is time to Reduce the b into A. In other words, if the look-ahead is long enough (i.e. $k \geq n+1$), the parser will 'see' the c and can decide to perform the Reduce at the right moment *on that particular word*. Otherwise, the look-ahead will contain only b's which does not allow the parser to decide whether to Reduce or to keep shifting b's. Unfortunately, the size of the look-ahead is fixed, and there will always be a word which is too long for this look-ahead to be sufficient. More precisely, if the size of the look-ahead is k, the parser will not have enough information to parse deterministically all words $\texttt{ab}^{2n+1}\texttt{c}$ with $k \leq n$. So the grammar cannot be LR(k) for any k (and it is clearly unambiguous).

22 Donald E. Knuth. On the translation of languages from left to right. *Information and Computation (formerly known as Information and Control)*, 8:607–639, 1965. DOI: 10.1016/S0019-9958(65)90426-2

(1)	S	→	aAc
(2)	A	→	bAb
(3)		→	b

Figure 6.20: A grammar which is not LR(k) for any k.

6.8 The bottom-up hierarchy of languages

The next step in our comparison is to compare the bottom-up parsers from the point of view of the *families of languages* that they accept. To this end, we start with the following definition:

Definition 6.29 (LR(k), SLR(1) and LALR(k) languages)**.** A language L is LR(k) (SLR(1), LALR(k)) for some $k \geq 0$ iff there is an LR(k) grammar (respectively an SLR(1) or an LALR(k) grammar) G s.t. $L(G) = L$.

We denote by 'LR(k) lang.' ('SLR(1) lang.' and 'LALR(k) lang.') the classes of LR(k) languages (respectively, SLR(1) languages and LALR(k) languages). Clearly:

$$\mathrm{LR}(0)\ \text{lang.} \subseteq \mathrm{SLR}(1)\ \text{lang.} \subseteq \mathrm{LALR}(1)\ \text{lang.} \subseteq \mathrm{LR}(1)\ \text{lang.} \subseteq \cdots \mathrm{LR}(k)\ \text{lang.} \subseteq \mathrm{LR}(K+1)\ \text{lang.} \subseteq \cdots$$

This is a direct consequence of the hierarchy of bottom-up grammars (Theorem 6.6) and of Definition 6.29. Let us now check whether these inclusions are strict or not.

Most of the results of this section can be found in the seminal paper on LR(k) by Donald KNUTH[23]. We start by observing, as KNUTH does, that

23 Donald E. Knuth. On the translation of languages from left to right. *Information and Computation (formerly known as Information and Control)*, 8:607–639, 1965. DOI: 10.1016/S0019-9958(65)90426-2

there is at least one language which is an LR(1) language but not an LR(0) language:

Proposition 6.7. *The language $L = \{\mathtt{a}, \varepsilon\}$ is an* LR(1) *language but not an* LR(0) *language.*

Proof. The fact that L is an LR(1) language can be established by finding an LR(1) grammar for it. The CFG that contains only the rules $S \to \mathtt{a}$ and $S \to \varepsilon$ is such a grammar. In the initial state of the CFSM, the character of look-ahead allows the parser to decide between: (i) shifting the a from the input when it is in the look-ahead; or (ii) reducing the S immediately when the look-ahead is empty.

For similar reasons, we can conclude that the language L is not LR(0). This is a bit more difficult, from the conceptual point of view, since we need to make a reasoning on *all possible grammars* that can define L and show that *none of them* can be LR(0). Assume we a have a grammar G s.t. $L(G) = L$, and assume we build the LR(0) CFSM of this grammar. Recall that the CFSM is a *deterministic finite state automaton* that reads all the *viable prefixes* of the grammar in order to identify the *handles*. In the initial state, the CFSM has read ε which is a handle (as ε is accepted); so, in the initial state, *Accept* must be one action that the parser can perform (otherwise, the parser would shift and miss the handle that allows it to accept $\varepsilon \in L$). However, a *Shift must* also be one of the actions of the parser, in order to obtain the handle a. So, there is necessarily a *Shift*/*Accept* conflict in the initial state of the LR(0) CFSM, that only one character of look-ahead can lift, as we have seen. □

So, clearly, LR(0) ⊂ LR(1), this inclusion is strict. But, surprisingly, such a strict inclusion does not carry out to the next levels of the hierarchy! Indeed, KNUTH, again, proves the following beautiful result, linking the LR(k) grammars for $k \geq 1$ and the *deterministic* PDA that we have studied in Section 4.2.2.

Theorem 6.8. *If L is the language of a* DPDA*, then there is an* LR(1) *grammar G that recognises it, i.e. $L(G) = L$*

We will refer the reader to the previously mentioned paper[24] for the proof of this theorem, and rather discuss here its implications.

First, let us recall that we denote by DCFL the class of languages accepted by *deterministic* PDA (DPDA). So, we can rephrase the theorem above by writing:

$$\mathsf{DCFL} \subseteq \mathrm{LR}(1) \text{ lang.}$$

However, we already know that all the LR(k) classes of languages *can* be recognised by a *deterministic* PDA[25], by definition! So, we conclude now that:

$$\text{for all } k \geq 1 : \mathrm{LR}(k) \text{ lang.} \subseteq \mathsf{DCFL}.$$

Putting everything together, we obtain:

$$\mathsf{DCFL} \subseteq \mathrm{LR}(1) \text{ lang.} \subseteq \mathrm{LR}(2) \text{ lang.} \subseteq \cdots \subseteq \mathrm{LR}(k) \text{ lang.} \subseteq \mathsf{DCFL},$$

which implies that:

$$\text{for all } k \geq 1 : \mathrm{LR}(k) \text{ lang.} = \mathsf{DCFL}.$$

The fact that DPDA pop up once again when establishing the expressive powers of parsers should not be a total surprise to the attentive reader (!) Indeed, the whole purpose of studying parsers was to find a way to parse (thus, by means of a PDA) CFL in a *deterministic way*. In some sense, this theorem tells us that we have reached our Graal with the LR(k) grammars!

[24] Donald E. Knuth. On the translation of languages from left to right. *Information and Computation (formerly known as Information and Control)*, 8:607–639, 1965. DOI: 10.1016/S0019-9958(65)90426-2

[25] Actually, by a deterministic PDA with look-ahead, but we have seen in Section 5.2.1 that the look-ahead is essentially syntactic sugar and can be incorporated in a regular (deterministic) PDA

6.8.1 *Case of the languages suffixed by* \$

Is there a way to extend this result to LR(0)? We have already argued in Proposition 6.7 that $\mathrm{LR}(0) \subset \mathrm{LR}(1)$. However, observe that the example we have used to separate LR(0) and LR(1) can be slightly modified to yield a 'very similar' language that is now LR(0): one just needs to concatenate the language with a fresh special character, such as \$. Indeed, the language $\{\mathtt{a}\$, \$\}$ can now be checked to be LR(0): in the initial state of the CFSM, a Shift is the only possible action. Depending whether an a or a \$ is shifted, we will end in two different states, where the proper action (Shift after reading an a, Reduce after reading a \$) can be performed.

It turns out that this result can be generalised when we restrict ourselves to languages of the form $L \cdot \{\$\}$, i.e. languages where *all* the words end by the *same* fixed letter. This is again a result due to KNUTH[26]:

Here, we have chosen to use the letter \$ to end words in the language, but any fixed symbol that does not occur anywhere in words can be used.

[26] Donald E. Knuth. On the translation of languages from left to right. *Information and Computation (formerly known as Information and Control)*, 8:607–639, 1965. DOI: 10.1016/S0019-9958(65)90426-2

Theorem 6.9. *For all languages L in $\mathsf{DCFL} \cap \Sigma^* \cdot \{\$\}$ (where $\$ \not\in \Sigma$): there is an* LR(0) *grammar to define L.*

As a consequence, if we restrict ourselves to languages where all words end by a given character, we can state that $\mathrm{LR}(k)$ lang. = DCFL for all $k \geq 0$.

In practice, when considering compiler construction, the restriction that 'all words need to end with \$' is not really a problem. Indeed, one can interpret the \$ as the 'end of file' special symbol that is usually added at the end of files by operating systems, for example.

6.8.2 *Practical interest of* LR(k)

Since 'all LR(k) parsers can accept the same families of languages', one can reasonably wonder why it was necessary to define them all? The motivation is of course purely practical: even if we can recognise all languages of DPDA, even with an LR(0) parser (under the light restriction we have seen above), this does not mean that it is always easy or desirable[27] to obtain an LR(0) grammar for a given language. So, we need parsers and parser generators that can exploit broad classes of grammars. The experience shows that many grammars that one uses in practice fall into the LALR(1) class, which explains why tools such as `yacc`[28], `bison`[29] and `cup`[30] target this particular class of grammars.

[27] After all, grammars are also designed to specify the syntax of computer languages to human beings who have to program. So they should be readable.

[28] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, AT&T Bell Laboratories, 1975. Readable online at http://dinosaur.compilertools.net/yacc/

[29] Gnu bison. https://www.gnu.org/software/bison/. Online: accessed on December, 29th, 2015

[30] Cup: Construction of useful parsers. http://www2.cs.tum.edu/projects/cup/. Online: accessed on December, 29th, 2015

6.9 *Comparison of the top-down and bottom-up hierarchies*

The next step, in our discussion of the different families of context-free grammars and languages that can be parsed deterministically, is to compare the bottom-up hierarchies we have established in the two previous sections, to the results of Section 5.4.2 and Section 5.4.3 that concern the top-down hierarchy.

6.9.1 *Comparing the hierarchies of grammars*

We start by comparing the top-down and bottom-up hierarchies of grammars. Recall from Theorem 5.5 that the families of LL(k) grammars form a strict hierarchy, i.e.: $\mathrm{LL}(0) \subsetneq \mathrm{LL}(1) \subsetneq \mathrm{LL}(2) \subsetneq \cdots \subsetneq \mathrm{LL}(k) \subsetneq \mathrm{LL}(k+1) \subsetneq \cdots$.

Recall also that, from Theorem 6.6, the same holds the families of LR(k) grammars, i.e.: $\mathrm{LR}(0) \subsetneq \mathrm{SLR}(1) \subsetneq \mathrm{LALR}(1) \subsetneq \mathrm{LR}(1) \subsetneq \mathrm{LR}(2) \subsetneq \cdots \subsetneq \mathrm{LR}(k) \subsetneq \mathrm{LR}(k+1) \subsetneq \cdots$

The main result for this comparison consists in checking that *all* LL(k) *grammars are* LR(k), while there are LR(k) grammars that are not LL(k). In other words: $\mathrm{LL}(k) \subsetneq \mathrm{LR}(k)$ for all $k \geq 0$. An history of attempts to establish this result together with a comprehensive proof can be found in a 1982 paper by Anton NIJHOLT[31].

[31] Anton Nijholt. On the relationship between LL(k) and LR(k) grammars. *Information Processing Letters*, 15(3):97–101, 1982. DOI: 10.1016/0020-0190(82)90038-2. URL https://www.researchgate.net/publication/222460902_On_the_relationship_between_the_LLk_and_LRk_grammars

Theorem 6.10. *For all* $k \geq 0$*:* $\mathrm{LL}(k) \subsetneq \mathrm{LR}(k)$.

Proof. For the proof that $\mathrm{LL}(k) \subseteq \mathrm{LR}(k)$, we refer the reader to the paper by NIJHOLT. The argument is rather short and consists in showing that: "if the leftmost derivations of a grammar satisfy the LL(k) conditions, then the rightmost derivations satisfy the LR(k) conditions" (see page 98 of the article).

To prove that the inclusion is strict, one can rely on the grammar of Figure 6.19, which is in LR($k+1$) but not in LL($k+1$) (for all $k \geq 0$) since $\mathrm{First}^{k+1}\left(A\mathtt{b}^k\mathtt{c}\right) = \mathrm{First}^{k+1}\left(B\mathtt{b}^k\mathtt{d}\right) = \mathtt{ab}^k$. □

This results can be seen as the ground behind the folklore assertion that 'bottom-up parsers are more powerful than top-down parsers' (although we will provide other arguments to support this in Section 6.9.2). In other words, when we have ascertained that k characters of look-ahead are sufficient to parse a grammar top-down, we are sure that k characters of look-ahead will be sufficient for a bottom-up parser as well.

With this result in mind, we can summarise the relationships between the different (syntactic) classes of grammars in Figure 6.21. On this figure, we have placed several points that correspond to grammars that we are about to describe now. These examples are interesting because they provide a nice insight into the characteristics that make a grammar belong to a particular class.

We also refer to the *LL and the LR hierarchy* to denote the classes of grammars that belong respectively to LL(k) and LR(k) for some k. In other words, the LL hierarchy refers to the infinite union:

$$\bigcup_{k\geq 0} \mathrm{LL}(k)\,,$$

and, symmetrically, the LR hierarchy refers to:

$$\bigcup_{k\geq 0} \mathrm{LR}(k)\,.$$

In order to discuss all these examples, we first introduce two grammar transformations.

1. The **first transformation (henceforth called T1)** turns a grammar G_1 which is assumed to be LR(k_1) (for some $k_1 \geq 0$) into a grammar G_2 that is still LR(k_1) but *not* LL(k_2) for a chosen $k_2 \geq 0$. Let G_1 be an LR(k_1) grammar, and let a and b be two *fresh* terminals (i.e., terminals that do not appear in G_1). Then, let k_2 be a natural number (possibly equal to k_1) and let G_2 be the grammar obtained from G_1 by adding

Observe that this argument also applies when G_1 is SLR(1) or LALR(1). Further observe that this transformation does *not* preserve the language of G_1, so it can only be used to prove the *existence* of a grammar that belongs to the same bottom-up class than G_1 but is not LL(k_2) for a chosen k_2.

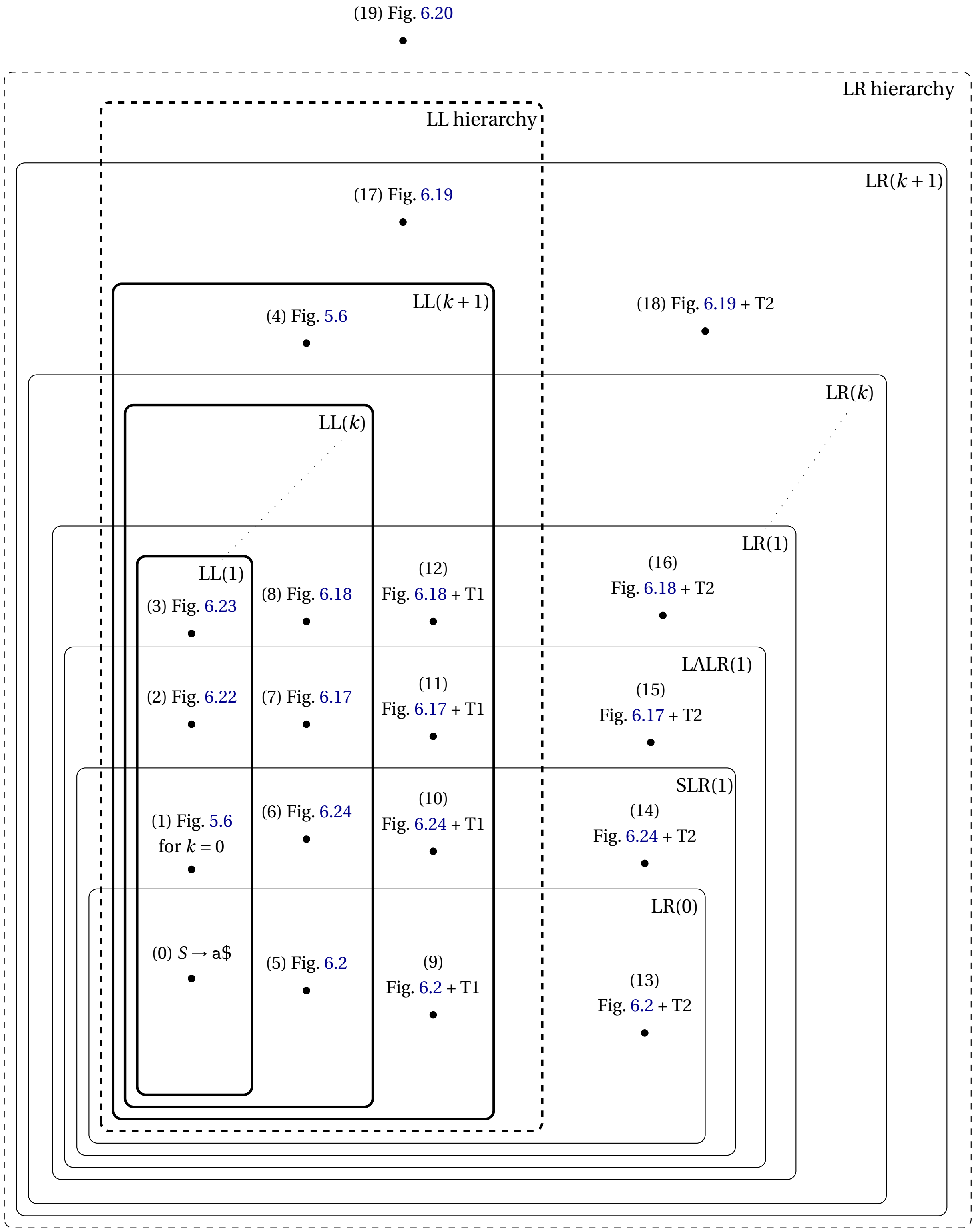


Figure 6.21: Comparison of different (syntactic) classes of grammars. The top-down (LL) hierarchy is in bold, the other classes are from the bottom-up (LR) hierarchy. All subsets are non-empty.

rules $S \rightarrow \mathtt{a}^{k_2}\mathtt{b}$ as well as $S \rightarrow \mathtt{a}^{(k_2+1)}\mathtt{b}$. Clearly, those rules imply that $G_2 \not\in \mathrm{LL}(k_2)$. However, those new rules do not change the $\mathrm{LR}(k_1)$ character of the grammar, because a and b are fresh symbols. Indeed, if we build the LR(0) CFSM for grammar G_2, we will obtain, in comparaison with the LR(0) CFSM of G_1:

- k_2 extra states:

$$\{S \rightarrow \bullet \mathtt{a}^{k_2}\mathtt{b}, S \rightarrow \bullet \mathtt{a}^{(k_2+1)}\mathtt{b}\}, \{S \rightarrow \mathtt{a} \bullet \mathtt{a}^{(k_2-1)}\mathtt{b}, S \rightarrow \mathtt{a} \bullet \mathtt{a}^{(k_2)}\mathtt{b}\}, \ldots, \{S \rightarrow \mathtt{a}^{k_2} \bullet \mathtt{b} S \rightarrow \mathtt{a}^{k_2} \bullet \mathtt{ab}\},$$

  in which the parser will shift the a's to the stack; and
- three extra states $\{S \rightarrow \mathtt{a}^{k_2}\mathtt{b} \bullet\}$, $\{S \rightarrow \mathtt{a}^{(k_2+1)} \bullet \mathtt{b}\}$ and $\{S \rightarrow \mathtt{a}^{(k_2+1)}\mathtt{b} \bullet\}$ where there are no conflicts, even with a look-ahead of size 0.

2. The **second transformation (henceforth called T2)** is similar to the previous one but allows one to make sure that the resulting grammar G_2 is not $\mathrm{LL}(k)$ *for any* $k \geq 1$, while retaining its properties in the bottom-up hierarchy. Again, let G_1 be an $\mathrm{LR}(k_1)$ grammar, and let G_2 be the grammar obtained by adding to G_1 the rules $S \rightarrow A$, $A \rightarrow A\mathtt{a}$ and $A \rightarrow \mathtt{a}$ (where A is a fresh variable and a is a fresh terminal). Clearly, G_2 is *not* $\mathrm{LL}(k)$ for any k since it is now left-recursive. However, one can check here as well that this transformation does not introduce conflicts in the LR(0) CFSM of the grammar, so G_2 retains the same bottom-up properties than G_1.

Now, we can discuss the grammars that appear as elements of the sets in Figure 6.21:

(0) A simple grammar containing only the rule $S \rightarrow \mathtt{a\$}$ for example is clearly both LL(1) and LR(0). It is actually LL(0) as well.

(1) The grammar from Figure 5.6, when we let $k = 0$ is an LL(1) grammar that can be checked to be SLR(1) as well. It is however *not* LR(0) since the two first rules introduce a Shift/Reduce conflict in the initial state of the CFSM.

(2) The grammar in Figure 6.22 is an LL(1) grammar that can be checked to be LALR(1) as well. It is however not SLR(1) since:

$$\mathrm{Follow}(A) = \mathrm{Follow}(B) = \{\mathtt{a}, \mathtt{b}\}.$$

(1)	S	$\rightarrow$	$A\mathtt{a}A\mathtt{b}$
(2)		$\rightarrow$	$B\mathtt{b}B\mathtt{a}$
(3)	A	$\rightarrow$	ε
(4)	B	$\rightarrow$	ε

Figure 6.22: An grammar which is LL(1) and LALR(1) but not SLR(1).

(3) The grammar in Figure 6.23 can be checked to be LL(1) but not LALR(1). This grammar is adapted from an example of book of Andrew APPEL [32]. The grammar is not LALR(1) because the LR(1) CFSM contains the two states:

E	$\rightarrow$	$A\bullet$,	$\{\mathtt{b}\}$
F	$\rightarrow$	$A\bullet$,	$\{\mathtt{c}\}$

and

[32] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998. ISBN 0-521-58274-1

E	$\to$	$A\bullet$,	{c}
F	$\to$	$A\bullet$,	{b}

whose merge in the LALR(1) will create a Reduce/Reduce conflict.

(4) In general, grammar G_k as found in Figure 5.6 (with parameter set to $k+1$), gives a grammar that is LL($k+1$) but not LR(k).

(5) The grammar we have used at the beginning of the chapter (in Example 6.4) is LR(0) but not LL(1) as we have already seen. It is however still in the LL hierarchy, as it is LL(2).

(6) The grammar in Figure 6.24 is a simple example of grammar that is not LL(1) and not LR(0), but which is LL(2) and SLR(1).

(7) The grammar in Figure 6.17 has already been shown to be LALR(1) but not SLR(1). We can also observe that it is not LL(1) because $\mathtt{d} \in \mathrm{First}(A\mathtt{a}) \cap \mathrm{First}(\mathtt{dc})$. However, it is LL(2).

(8) The grammar in Figure 6.18 has also been discussed already, and we know that it is LR(1) but not LALR(1). We can check that it is not LL(1) nor LL(2): for example, a look-ahead of ac does not allow one to chose between $S \to \mathtt{a}A\mathtt{d}$ and $S \to \mathtt{a}B\mathtt{e}$ since both A and B produce c. However, it is still in the LL hierarchy, as it is LL(3).

Alternatively, the grammar in Figure 6.23 can be modified using transformation T1 above to obtain a grammar that is LR(1) and LL(2) but not LALR(1)

(1)	S	$\to$	aX
(2)		$\to$	Eb
(3)		$\to$	Fc
(4)	X	$\to$	Ec
(5)		$\to$	Fb
(6)	E	$\to$	A
(7)	F	$\to$	A
(8)	A	$\to$	ε

Figure 6.23: An grammar which is LL(1) and not LALR(1)

(1)	S	$\to$	ab
(2)	S	$\to$	ac
(3)	S	$\to$	a

Figure 6.24: A grammar that is LL(2) and SLR(1) but neither LL(1) nor LR(0).

(9)–(12) The grammars which have been used in the points (5) to (8) above can be modified using transformation T1 in order to make them LL(k) but not LL($k+1$) for any k.

(13)–(16) Similarly, transformation T2 can be used to make all these grammars be outside of the LL hierarchy.

(17) We have already seen that the grammar in Figure 6.19 is LR($k+1$) but not LR(k). One can also check that this grammar is not LL($k+1$), since $\mathrm{First}^{k+1}(A\mathtt{b}^k\mathtt{c}) = \mathrm{First}^{k+1}(B\mathtt{b}^k\mathtt{d}) = \{\mathtt{ab}^k\}$. It is however, LL($k+2$), so still in the LL hierarchy.

(18) Then, applying transformation T2 to the grammar of Figure 6.19 allows one to obtain a grammar that is in LR($k+1$) \ LR(k) but not LL(k) for any k.

(19) Finally, we have already shown that the grammar in Figure 6.20 is not LR(k) for any k, so outside the LR hierarchy (while still non-ambiguous).

Don't forget that a grammar which is ambiguous cannot be LL(k) nor LR(k) for any k. However, that does not necessarily imply that all non-ambiguous CFG fall into one of those categories.

6.9.2 *Comparing the hierarchies of languages*

Let us now compare the top-down and bottom-up hierarchies of languages. This is will turn out to be much easier than the comparison of grammar classes, since we already know that the LR(k) hierarchy of languages 'collapses' to correspond to the languages of DPDA. Associating this result with the hierarchy we have obtained in Section 5.4.3, we obtain:

$$\text{LL}(1)\ \text{lang.} \subsetneq \text{LL}(1)\ \text{lang.} \subsetneq \cdots \subsetneq \text{LL}(k)\ \text{lang.} \subsetneq \cdots \subsetneq \text{LR}(1)\ \text{lang.} = \text{LR}(2)\ \text{lang.} = \cdots = \text{LR}(k)\ \text{lang.} = \cdots = \mathsf{DCFL}$$

6.10 Exercises

6.10.1 LR(0)

Exercise 6.1. Build the CFSM corresponding to the following grammar:

(1)	S'	$\rightarrow$	$S\$$
(2)	S	$\rightarrow$	aCd
(3)		$\rightarrow$	bD
(4)		$\rightarrow$	Cf
(5)	C	$\rightarrow$	eD
(6)		$\rightarrow$	Fg
(7)		$\rightarrow$	CF
(8)	F	$\rightarrow$	z
(9)	D	$\rightarrow$	y

Then, give the action table of the LR(0) parser on this grammar.

The algorithms to build a CFSM are found in Section 6.2.

The building techniques for an LR(0) are in Section 6.3.

Exercise 6.2. Simulate the run of the LR(0) parser for the grammar of the previous exercise, on the word `aeyzzd`.

See Section 6.3 again.

6.10.2 SLR(1)

Exercise 6.3. Build the SLR(1) parser for the following grammar (i.e., build the appropriate CFSM and give the SLR(1) action table):

(1)	S'	$\rightarrow$	$S\$$
(2)	S	$\rightarrow$	A
(3)	A	$\rightarrow$	bB
(4)		$\rightarrow$	a
(5)	B	$\rightarrow$	cC
(6)		$\rightarrow$	cCe
(7)	C	$\rightarrow$	dAf

Is the above grammar LR(0)? Justify your answer.

SLR(1) parsers are covered in Section 6.4.

6.10.3 LR(k)

Exercise 6.4. Build the LR(1) parser for the following grammar (i.e., build the appropriate CFSM and give the LR(1) action table):

(1)	S'	$\rightarrow$	$S\$$
(2)	S	$\rightarrow$	SaSb
(3)		$\rightarrow$	c
(4)		$\rightarrow$	ε

Is this grammar LR(0)? Is it SLR(1) 1? Justify your answers.

Section 6.5 is devoted to LR(k) parsers.

Exercise 6.5. Simulate the run of the parser you built at the previous exercise on the word `abacb`.

6.10.4 LALR(1)

Exercise 6.6. Build the LALR(1) parser for the grammar of exercise 6.4, using the LR(1) parser you have built for the same exercise.

The definition of LALR(1) parsers in Section 6.6 shows how to build them from LR(1) parsers.

Exercise 6.7. Find a grammar which is LR(1) but not LALR(1).

Since LALR(1) parsers can be built from LR(1) parsers, try to come up with states of an LR(1) parser that would be generate a conflict when the LALR(1) parser is built, and infer a grammar from that.

A Some reminders of mathematics

This section is a quick reminder of some mathematical concepts that are pervasive in these lecture notes. It is intended as a refresher, not as an in-depth explanation from which one could study from. Readers are advised to read this section first to make sure they are familiar with all the concepts. If not, they are advised to refer to a good textbook on discrete mathematics, such as the free online book[1] by DOER and LEVASSEUR, for example.

[1] A. Doer and K. Levasseur. *Applied Discrete Structures*. 2012. Available online with supplementary material at: https://discretemath.org/

A.1 Greek letters

Although not strictly mathematical content, we believe that a reminder of the names of the greek letters (as they are used everywhere in these notes) would be useful. This alphabet is given in Table A.1.

Did you know where the word 'alphabet' comes from? If not, have a look at the two first lines of the table...

Table A.1: Greek letters.

Capital	Lowercase	Name
A	α	alpha
B	β	beta
Γ	γ	gamma
Δ	δ	delta
E	ε	epsilon
Z	ζ	zeta
H	η	eta
Θ	θ	theta
I	ι	iota
K	κ	kappa
Λ	λ	lambda
M	μ	mu
N	ν	nu
Ξ	ξ	xi
O	o	omicron
Π	π	pi
P	ρ	rho
Σ	σ	sigma
T	τ	tau
Υ	υ	upsilon
Φ	φ	phi
X	χ	chi
Ψ	ψ	psi
Ω	ω	omega

The names we are used in the table are the 'ancient greek' names. In contemporary Greece some of these letters are called differently. For example, μ and ν are now called 'mi' and 'ni' respectively. Also, some lettres can be written differently: Epsilon can be written ε (as in those notes) or ϵ. Phi can be written φ (as we do here) or ϕ.

One should pay attention not to confuse ε (epsilon) with ξ (xi). One should also remark that the alternate glyph ϵ for epsilon is not the same as the mathematical symbol $\in$ which means 'belongs to' (as in $x \in X$: the element x belongs to the set X). Finally, the letters Φ and ϕ (phi) should not be confused with the empty set symbol $\emptyset$.

A.2 Sets and relations

For the sake of completeness, we recall the very basic definitions about sets, although we assume the reader must be pretty familiar with them.

A.2.1 Sets

Definition A.1 (Set (informal))**.** A *set* is a (finite or infinite) collection of *elements.*

We rely on the classical notation $x \in X$ to say that x is an element of set X. We also use $A \subseteq B$ to denote that 'set A is a *subset* of B', which means that all elements of set A are also elements of B. We use $A \subset B$ to denote the fact that set A is a proper subset of B. This means that $A \subseteq B$ *and* that there is at least one element of B that is not in A (in other words $A \neq B$). We sometimes use the alternate notation $A \subsetneq B$ to emphasise the fact that $A \neq B$.

Example A.2. We can denote the set S containing all natural numbers between 2 and 7 (included) as:

$$S = \{2, 3, 4, 5, 6, 7\}.$$

Here, we use an *enumeration* to list all the elements of the set. Although we have chosen to enumerate the elements in increasing order, we could have written:

$$S = \{3, 6, 5, 2, 7, 4\}$$

instead. Indeed, sets by themselves do not carry a notion of order.

Other sets of interest to us are the sets of natural numbers and of integers:

$$\mathbb{N} = \{0, 1, 2, 3, \ldots\}$$
$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}.$$

These sets, however, are examples of infinite sets and cannot be completely enumerated[2]. Moreover, even some sets which are finite can be more conveniently represented using the so-called *set-builder notation.* For example, we could define the previous set S like so:

[2] In the sense that we cannot write all their elements on a sheet of paper. Note that there exists a mathematical notion of 'enumerable set' for which the natural and integer numbers *are* enumerable

$$S = \{x \mid 2 \leq x \leq 7\}.$$

Similarly, the set of all even natural numbers can be expressed as:

$$\{x \in \mathbb{N} \mid x = 2.y \text{ for some } y \in \mathbb{N}\}.$$

Finally, let us note that the *elements* of sets can be *anything*, including:

Note how we use brackets () for denoting pairs and curly brackets {} for denoting sets, as is standard practice.

1. *pairs.* A pair is an *ordered* collection of two elements. For example, $(1,2)$ is the pair where the first element is 1 and the second is 2. Pairs are commonly used to denote coordinates in a plane for instance, and clearly, one should not confuse the x and the y coordinates, so the order matters. Then:
$$\{(1,2),(2,3),(1,3)\}$$
and
$$\{(x,y) \mid x \in \mathbb{N} \text{ and } y \in \mathbb{N} \text{ and } y = 2.x\}$$
are two examples of sets containing pairs.

2. *other sets.*

A.2.2 *Relations*

Before we can introduce relations, we need the notion of *cartesian product*:

Definition A.3 (Cartesian product)**.** Given two sets A and B (which might be equal), their *cartesian product*, denoted $A \times B$ is the set:

$$\{(a,b) \mid a \in A \text{ and } b \in B\}$$

As can be seen from the definition, the cartesian product of A and B is a set of *pairs* of elements from A and B.

Example A.4. The cartesian product of $A = \{1,2,3\}$ and $B = \{\mathtt{a},\mathtt{b}\}$ is:

$$A \times B = \big\{(1,\mathtt{a}),(2,\mathtt{a}),(3,\mathtt{a}),(1,\mathtt{b}),(2,\mathtt{b}),(3,\mathtt{b})\big\}.$$

The cartesian product of $\mathbb{Z}$ by itself is the set of all possible integer coordinates in a two-dimensional plane. Instead of writing $\mathbb{Z} \times \mathbb{Z}$, we rather write $\mathbb{Z}^2$.

We can now introduce the notion of (binary) relation:

Definition A.5 (Binary relation)**.** Given two sets A and B, a binary relation over A and B is a subset of $A \times B$.

Since we will only consider binary relations here, we will simply call them *relations*. Put simply, a relation over A and B is a set of pairs (a,b) where $a \in A$ and $b \in B$. Let R be such a relation. Instead of writing $(a,b) \in R$, we will adopt the common shorthand aRb, as shown by the following examples.

Example A.6. Let $S = \{1,2,3\}$. Then, the set:

$$\big\{(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)\big\}$$

is a relation over S^2 (in this case, both sets A and B from the definition are equal to S). It defines the 'smaller than or equal to' concept over S. So, we can call this relation '$\leq$', and write $(a,b) \in \leq$ iff a is smaller than or equal to b. With the shorthand notation, we write $a \leq b$.

The previous example highligths a particular case of relation, which is among the *homogenous relations*:

Definition A.7 (Homogenous relation)**.** A relation $R \subseteq A \times B$ is *homogenous* iff $A = B$.

In other words, a relation is homogenous if it is over A^2 for some set A.

Properties of relations The notion of relation is, as we have seen very general. It can be used to formalise several concepts. For instance, the previous example shows that the natural notion of *order* can be formalised as a binary relation. In order to identify certain relations of interest, we need to define some properties that relations can have. We start we two different properties of relations in general.

First, we look at *functional* relations. As the name indicates, this property captures the notion of *function.* Consider for example the function sin. We know that, for all possible value of x, $\sin(x)$ is a unique value. However, several values of x can be mapped by the function to the same value. For example, $\sin\left(\frac{\pi}{2}\right) = \sin\left(\frac{5\pi}{2}\right) = 1$. Now, if we see the function sin as a relation containing all the pairs $(x, \sin(x))$, we know that: (i) there can't be two pairs (x, y_1) and (x, y_2) with $y_1 \neq y_2$, since both y_1 and y_2 must be equal to $\sin(x)$, which is unique; however (ii) there can be several pairs (x_1, y) and (x_2, y) with $x_1 \neq x_2$. For example $\left(\frac{\pi}{2}, 1\right)$ and $\left(\frac{5\pi}{2}, 1\right)$ both exist in the relation. This is captured in the following definition:

Definition A.8 (Functional relation)**.** A relation $R \subseteq X \times Y$ over X and Y is *functional* iff for all $x \in X$, for all $y_1, y_2 \in Y$: xRy_1 and xRy_2 implies $y_1 = y_2$.

Now, if we look closely at this definition, we remark that it captures the notion of *partial* function. Indeed, the defintion says that, for all $x \in X$, there can be *at most* one pair of the form (x, y). When we 'always want the function to return something', we need an additional condition:

Definition A.9 (Total relation)**.** A relation $R \subseteq X \times Y$ over X and Y is *total* iff there exists a pair $(x, y) \in R$ for all $x \in X$.

Then, a *complete function* can be defined:

Definition A.10 (Complete function)**.** A function $f : X \to Y$ is *complete* (or total) iff the set $\{(x, f(x)) \mid x \in X\}$ is a functional and total relation.

Finally, the notion of *injective* relation can be defined symmetrically to that of a functional relation:

Definition A.11 (Injective relation)**.** A relation $R \subseteq X \times Y$ over X and Y is *injective* iff for all $y \in Y$, for all $x_1, x_2 \in X$: x_1Ry and x_2Ry implies $x_1 = x_2$.

Properties of homogenous relations Let us now focus on relevant properties of *homogeneous relations*, i.e. those relations that are subsets of A^2 for some set A.

Definition A.12 (Properties of homogeneous relations)**.** Let $R \subseteq A^2$ be a relation over A^2. Then, we say that:

1. R is *reflexive* iff, for all $a \in A$: $(a,a) \in R$. That is, all elements are always put in relation with themselves.

2. R is *symmetric* iff, for all $(a,b) \in R$: we also have $(b,a) \in R$. That is, every time a is put into relation with b through R, then b is also put in relation to a through R.

3. R is *antisymmetric* iff, for all $a,b \in A$: if $(a,b) \in R$ and $(b,a) \in R$, then $a = b$. In some sense, the antisymmetric property does not allo symmetry to happen on distinct elements: every time we have (a,b) and (b,a) in R, it must happens on the same elements, i.e. $a = b$.

4. R is *transitive* iff, whenenver $(a,b) \in R$ and $(b,c) \in R$: we also have $(a,c) \in R$. This is the classical definition of transitivity.

5. R is *strongly connected* iff, for all $a \in A$, for all $b \in B$: either $(a,b) \in R$ or $(b,a) \in R$. That is, for all pairs of elements in A, R always put them into relation one way or the order.

Observe that a relation which is not symmetric is not necessarily antisymmetric and vice-versa. It is possible that $(a,b) \in R$ and $(b,a) \in R$ for *some* pairs (a,b) but not all (hence, R is not symmetric), and that there exists at least such a pair with $a \neq b$ (hence, it is not antisymmetric).

Further observe that some authors use the word 'total' instead of 'strongly connected'. However, this seems to be deprecated, and we have adopted the modern notion of 'total relation', see Definition A.2.2.

While these properties might sound very abstract, they are the basic building blocks that allow one to define the classical concepts that we are used to manipulate, like partial orders, orders or equivalences, as we are about to see.

Partial orders and orders Let's start with the classical notion of order.

Definition A.13 ((Partial) orders)**.** A *partial order* is a transitive and antisymmetric homogeneous relation. An *order* (sometimes called *total order*) is a partial order which is also reflexive and strongly connected.

Example A.14. For example, the classical ordering relation $\leq$ on the integers is both an order and a partial order. Indeed, if $x \leq y$ and $y \leq z$, then $x \leq z$, so $\leq$ is *transitive*. It is also antisymmetric, because if $x \leq y$ and $y \leq x$, it can only be that $x = y$. So, $\leq$ is a partial order. Moreover, $x \leq x$ for all integer x, so $\leq$ is also reflexive. Finally, we can always compare two integers through $\leq$, so it is also strongly connected. Hence, $\leq$ is indeed an order.

Now let's lift the $\leq$ relation to pairs of integers: $(x_1, y_1) \leq (x_2, y_2)$ iff $x_1 \leq x_2$ and $y_1 \leq y_2$. In this case, we have elements that are incomparable. As a concrete example, assume we need to buy a washing machine, and that we rate the models according to their yearly energy consumption and their price. So each machine is characterised by (e,p). Assume we have a machine that consumes 100 kWh and costs 500 euros. We assign it the pair $(100,500)$. So, a machine with the pair $(80,400)$ is clearly better, and we have $(80,400) \leq (100,500)$. However, a machine with characteristics $(75,700)$ is not comparable to our $(100,500)$ machine: $(75,700) \not\leq (100,500)$ and $(100,500) \not\leq (75,700)$.

This new relation $\leq$ is still transitive and antisymmetric, so it is indeed a *partial order*. It is also reflexive, but *not* strongly connected, and is thus *not* an order.

Equivalence relations Another classical concept is that of *equivalence relation*. It can also be defined on top of the properties listed above.

Definition A.15 (Equivalence relation)**.** An equivalence relation is an homogeneous relation which is reflexive, transitive and symmetric.

Example A.16. As an example, let us say that fruits are *equivalent* when they have the same color (assuming they have only one color). So, for instance, tomatoes are equivalent to cherries because they are both red, cherries are also equivalent to strawberries because strawberries are red as well. So, clearly, strawberries must be equivalent to tomatoes. This show why an equivalence relation must be *transitive*. Of course, if cherries are the same color as tomatoes, then tomatoes are the same color as cherries (!) so our relation is symmetric. Finally, tomatoes are the same color as tomatoes (!!) so our relation is also reflexive. We conclude that our 'has the same color' relation is indeed an equivalence relation.

We can continue this example and see that equivalence relations naturally induce a splitting of the fruits between so-called *classes*: while tomatoes, cherries and strawberries are all red; bananas and lemons belong to their own gang of yellow fruits. All yellow fruits are equivalent to each other, but no yellow fruit can be equivalent to a red one. This is further formalised in the next definitions.

Definition A.17 (Partition)**.** A *partition* of a set A is a set of subsets A_1, A_2,..., A_n of A s.t.:

1. All elements of A occur in some subset A_i. For all $a \in A$, there exists i s.t. $a \in A_i$.

2. There is no overlapping between the subsets: for all $i \neq j$: $A_i \cap A_j = \emptyset$.

So, the notion of partition consists in 'splitting' the whole set A into different subsets, much like we do when we cut a cake. Such a 'cut' of the set can be done through an equivalence relation, when we put all equivalent elements together in a subset:

Definition A.18 (Equivalence classes)**.** Given a set A and an equivalence relation R on A, the *equivalence classes* of R are all the non-empty subsets $A_1, A_2, \ldots A_n$ of A s.t. for all $1 \leq i \leq n$: $a, b \in A_i$ iff aRb.

The equivalence classes of R form a *partition* of A.

Example A.19. One can now check that these definitions match the intuitions given in the example above. Given the set

$$A = \{\text{tomatoes}, \text{cherries}, \text{strawberries}, \text{lemons}, \text{bananas}\};$$

the subsets $\{\text{tomatoes}, \text{cherries}, \text{strawberries}\}$ and $\{\text{lemons}, \text{bananas}\}$ are the two equivalence classes of our 'has the same color' relation, and they indeed form a partition of A, since all fruits end up in either equivalence class, and there is no intersection between these classes.

Transitive closure Finally, an important concept regarding relations is that of *transitive closure*. Roughly speaking, computing the transitive closure of a given relation amounts to adding the minimal number of pairs that are necessary to make the relation transitive. Let us illustrate this on an example.

Example A.20. Let us consider the five cities: Antwerp (A), Brussels (B), Paris (P), New York (NY) and Miami (M). Let us assume we are given some information about the possibility to travel from one city to the other by road, as a relation:

$$R = \big\{(A,B),(B,P),(NY,M)\big\}.$$

That is, we know there is a road from Antwerp to Brussels, from Brussels to Paris and from New York to Miami. Let is now assume we want to know what are all the possible road connections we can deduce from this information. Clearly, if we can go from Antwerp to Brussels and from Brussels to Paris, then we can also go from Antwerp to Paris, so we can add the pair (A,P), but no further pair based on the information which is given to us.

This is exactly the transitive closure of the above relation:

$$R' = \big\{(A,B),(B,P),(NY,M),(A,P)\big\}.$$

Observe that this relation is indeed transitive, and that it contains R. It is also the *smallest* such relation. Indeed, the pair (A,P) must be added to make the relation transitive, and no other pair needs to be added to this aim. For example, if we had further added the pairs (A,NY) and (A,M), we would also have a transitive relation that contains R, but it is not the smallest that has these properties.

Let us formalise this:

Observe that the transitive closure of a transitive relation R is R itself.

Definition A.21 (Transitive closure of a relation)**.** Given a relation R, its transitive closure is the smallest relation R' s.t. (i) $R \subseteq R'$; and (ii) R' is transitive.

Finally, let us note that the notion of *transitive closure* can be extended to other properties of relations, such as: the *transitive and reflexive closure* of R is the smallest transitive and reflexive relation R^* that contain R, and so forth.

B Bibliography

Gnu bison. https://www.gnu.org/software/bison/. Online: accessed on December, 29th, 2015.

CLang: features and goals. http://clang.llvm.org/features.html. Online: accessed on December, 29th, 2015.

Cup: Construction of useful parsers. http://www2.cs.tum.edu/projects/cup/. Online: accessed on December, 29th, 2015.

GCC wiki: new C parser. https://gcc.gnu.org/wiki/New_C_Parser, 2008. Online: accessed on December, 29th, 2015.

A. Aho, M. Lam, R. Sethi, and Ullman J. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007.

Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970. ISSN 0362-1340. DOI: 10.1145/390013.808479.

Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, STOC '04, pages 202–211, New York, NY, USA, 2004. ACM. ISBN 1-58113-852-0. DOI: 10.1145/1007352.1007390. URL http://doi.acm.org/10.1145/1007352.1007390.

T. Anderson, J. Eve, and J. Horning. Efficient LR(1) parsers. *Acta Informatica*, 2:2–39, 1973. DOI: 10.1007/BF00571461.

Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998. ISBN 0-521-58274-1.

J.A. Brzozowski and Jr. McCluskey, E.J. Signal flow graph techniques for sequential circuit state diagrams. *Electronic Computers, IEEE Transactions on*, EC-12(2):67–76, April 1963. ISSN 0367-7508. DOI: 10.1109/PGEC.1963.263416.

N. Chomsky. *Syntactic Structures*. Mouton and Co, The Hague, 1957.

N. Chomsky. On certain formal properties of grammars. *Information and Computation (formerly known as Information and Control)*, 2(2):137 – 167, 1959. DOI: 10.1016/S0019-9958(59)90362-6.

Franklin Lewis DeRemer. *Practical Translators for LR(k) Languages*. PhD thesis, Massachusetts Institute of Technology, 1969. URL https://web.archive.org/web/20130819012838/http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-065.pdf.

Franklin Lewis DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7), 1971. DOI: 10.1145/362619.362625.

A. Doer and K. Levasseur. *Applied Discrete Structures*. 2012. Available online with supplementary material at: https://discretemath.org/.

Python Software Foundation. re – Regular expression operations. https://docs.python.org/3/library/re.html. Online: accessed on April 12th, 2023.

H.W. Fowler, J.B. Sykes, and F.G. Fowler. *The Concise Oxford dictionary of current English*. Clarendon Press, 1976.

Seymour Ginsburg and Sheila Greibach. Deterministic context free languages. *Information and Computation (formerly known as Information and Control)*, 9(6):620–648, 1966. ISSN 0019-9958. DOI: 10.1016/S0019-9958(66)80019-0. URL https://www.sciencedirect.com/science/article/pii/S0019995866800190.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321455363.

Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, AT&T Bell Laboratories, 1975. Readable online at http://dinosaur.compilertools.net/yacc/.

B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall software series. Prentice Hall, 1988.

Stephen C. Kleene. Representation of events in nerve nets and finite automata. Technical Report RM-704, The RAND Corporation, 1951. URL http://minicomplexity.org/pubr.php?t=2&id=2.

Gerwin Klein, Steve Rowe, and Régis Décamps. Jflex user's manual. https://jflex.de/manual.html, March 2023. Version 1.9.1. Online: accessed on April, 12th, 2023.

Donald E. Knuth. On the translation of languages from left to right. *Information and Computation (formerly known as Information and Control)*, 8:607–639, 1965. DOI: 10.1016/S0019-9958(65)90426-2.

Dexter Kozen. On Kleene algebras and closed semirings. In *Mathematical foundations of computer science, Proceedings of the 15th Symposium, MFCS '90, Banská Bystrica/Czech. 1990*, volume 452 of *Lecture notes in computer science*, pages 26–47, 1990. URL http://www.cs.cornell.edu/~kozen/Papers/kacs.pdf.

R. Kurki-Suonio. Notes on top-down languages. *BIT Numerical Mathematics*, 9(3):225–238, 1969. ISSN 1572-9125. DOI: 10.1007/BF01946814.

W. R. LaLonde, E. S. Lee, and J. Horning. An LALR(k) parser generator. In *Proceedings of IFIP congress*, pages 151–153. Elsevier Science, New York, 1971.

Leslie Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley, 1986. ISBN 0-201-15790-X.

P. M. Lewis, II and R. E. Stearns. Syntax-directed transduction. *J. ACM*, 15 (3):465–488, July 1968. ISSN 0004-5411. DOI: 10.1145/321466.321477.

R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *Electronic Computers, IRE Transactions on*, EC-9(1):39–47, March 1960. ISSN 0367-9950. DOI: 10.1109/TEC.1960.5221603.

A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):pp. 541–544, 1958. ISSN 00029939. URL http://www.jstor.org/stable/2033204.

Anton Nijholt. On the relationship between LL(k) and LR(k) grammars. *Information Processing Letters*, 15(3):97–101, 1982. DOI: 10.1016/0020-0190(82)90038-2. URL https://www.researchgate.net/publication/222460902_On_the_relationship_between_the_LLk_and_LRk_grammars.

Damian Niwińsky and Wojciech Rytter. *200 Problems in Formal Languages and Automata Theory*. University of Warsaw, 2017.

Thomas J. Penello and Franklin Lewis DeRemer. Efficient computation of LALR(1) look-ahead sets. *ACM SIGPLAN Notices*, 39(4), 2004. DOI: 10.1145/69622.357187.

M.O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, April 1959. ISSN 0018-8646. DOI: 10.1147/rd.32.0114. URL https://www.researchgate.net/publication/230876408_Finite_Automata_and_Their_Decision_Problems.

R.M. Ritter. *The Oxford Guide to Style*. Language Reference Series. Oxford University Press, 2002.

D.J. Rosenkrantz and R.E. Stearns. Properties of deterministic top-down grammars. *Information and Computation (formerly known as Information and Control)*, 17(3):226 – 256, 1970. ISSN 0019-9958. DOI: 10.1016/S0019-9958(70)90446-8.

Itiiro Sakai. Syntax in universal translation. In *International Conference on Machine Translation of Languages and Applied Language Analysis*, pages 593–608. London: Her Majesty's Stationery Office, 1961.

Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996. ISBN 053494728X.

L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 1–9, New York, NY, USA, 1973. ACM. DOI: 10.1145/800125.804029.

Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. ISSN 0001-0782. DOI: 10.1145/363347.363387.

Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. *Compiler Design, Syntactic and Semantic Analysis*. Springer-Verlag, 2013. ISBN 978-3-642-17539-8. DOI: 10.1007/978-3-642-17540-4.

Niklaus Wirth and Helmut Weber. EULER: A generalization of ALGOL, and its formal definition: Part II. *Commun. ACM*, 9(2):89–99, February 1966. ISSN 0001-0782. DOI: 10.1145/365170.365202.