Texts in Computer Science

Stefano Crespi Reghizzi
Luca Breveglieri
Angelo Morzenti

Formal Languages and Compilation

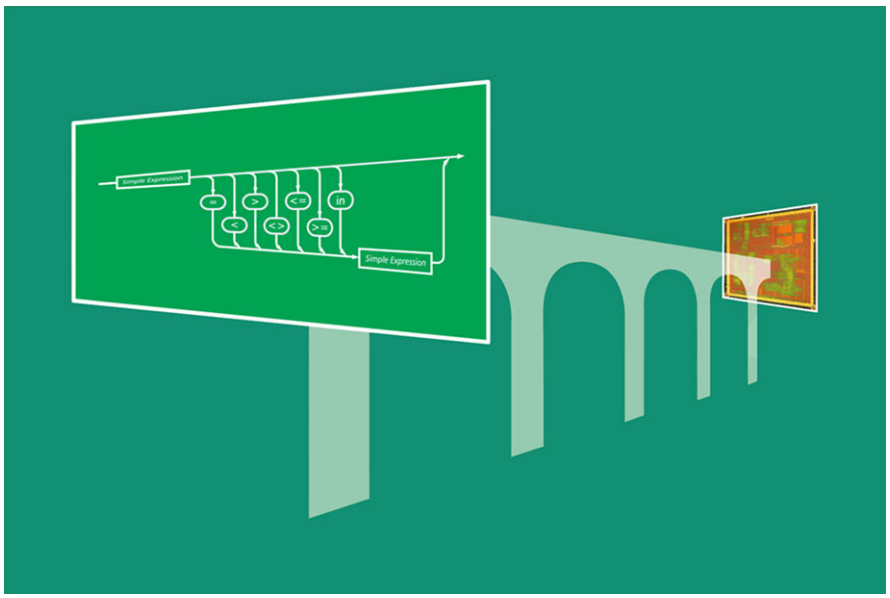
Second Edition

Springer

Texts in Computer Science

Editors
David Gries
Fred B. Schneider

For further volumes:
www.springer.com/series/3191


Simple Expression
=
>
<=
in
<
<>
>=
Simple Expression

Stefano Crespi Reghizzi • Luca Breveglieri •
Angelo Morzenti

Formal Languages and Compilation

Second Edition

Springer

Stefano Crespi Reghizzi
Dipartimento Elettronica
Informazione e Bioingegneria
Politecnico di Milano
Milan, Italy

Angelo Morzenti
Dipartimento Elettronica
Informazione e Bioingegneria
Politecnico di Milano
Milan, Italy

Luca Breveglieri
Dipartimento Elettronica
Informazione e Bioingegneria
Politecnico di Milano
Milan, Italy

ISSN 1868-0941 ISSN 1868-095X (electronic)
Texts in Computer Science
ISBN 978-1-4471-5513-3 ISBN 978-1-4471-5514-0 (eBook)
DOI 10.1007/978-1-4471-5514-0
Springer London Heidelberg New York Dordrecht

Preface

The textbook derives from the well-known identically titled volume[1] published in 2009: two younger co-authors have brought their experience to enrich and streamline the book, without dulling its original structure. The selection of materials and the presentation style have not substantially changed. The book reflects many years of teaching compiler courses and of doing research on formal language theory and formal methods, on compiler and language technology, and to a lesser extent on natural language processing. The more important change concerns the central topic of language parsing. It is a completely new, systematic, and unified presentation of the most important parsing algorithms, including also parallel parsing.

Goals In the turmoil of information technology developments, the subject of the book has kept the same fundamental principles since half a century, and has preserved its conceptual importance and practical relevance. This state of affairs in a topic that is central to computer science and is based on established principles, might lead some people to believe that the corresponding textbooks are by now consolidated, much as the classical books on mathematics and physics. In reality, this is not the case: there exist fine classical books on the mathematical aspects of language and automata theory, but for what concerns the application to compiling, the best books are sort of encyclopedias of algorithms, design methods, and practical tricks used in compiler design. Indeed, a compiler is a microcosm, and features many different aspects ranging from algorithmic wisdom to computer hardware. As a consequence, the textbooks have grown in size, and compete with respect to their coverage of the last developments on programming languages, processor architectures and clever mappings from the former to the latter.

To put things in order, it is better to separate such complex topics into two parts, *basic* and *advanced*, which to a large extent correspond to the two subsystems that make a compiler: the user-language specific front-end, and the machine-language specific back-end. The basic part is the subject of this book. It covers the principles and algorithms to be used for defining the syntax of languages and for implementing simple translators. It does not include: the specialized know-how needed for various classes of programming languages (imperative, functional, object oriented, etc.), the computer architecture related aspects, and the optimization methods used to improve the machine code produced by the compiler.

[1] S. Crespi Reghizzi, *Formal Languages and Compilation* (Springer, London, 2009).

Organization and Features In other textbooks the bias towards practical aspects has reduced the attention to fundamental concepts. This has prevented their authors from taking advantage of the improvements and simplifications made possible by decades of extensive use, and from avoiding the irritating variants and repetitions that are found in the original papers. Moving from these premises, we decided to present, in a simple minimalist way, the principles and methods used in designing language syntax and syntax-directed translators.

Chapter 2 covers regular expressions and context-free grammars, with emphasis on the structural adequacy of syntactic definitions and a careful analysis of ambiguity and how to avoid it.

Chapter 3 presents finite-state recognizers and their conversion back and forth to regular expressions and grammars.

Chapter 4 presents push-down machines and parsing algorithms, focusing attention on the *LL*, *LR* and Earley methods. We have substantially improved the standard presentation of such algorithms, by unifying the concepts and notations used in various approaches, and by extending the method coverage with a reduced definitional apparatus. An example that expert readers and instructors should appreciate, is the unification of the top-down (*LL*) and bottom-up (*LR*) parsing algorithms, as well as the tabular (Early) one, within a novel practical framework. In this way, the effort and space spared have made room for advanced methods typically not present in similar textbooks. First, our parsing algorithms apply to the Extended *BNF* grammars, which are the de facto standard in the language reference manuals. Second, we provide a parallel parsing algorithm that takes advantage of the many processing units of modern microprocessors, to speed-up the analysis of large files.

The book is not restricted to syntax: Chap. 5, the last one, studies translations, semantic functions (attribute grammars), and the static program analysis by data flow equations. This provides a comprehensive understanding of the compilation process, and covers the essential elements of a syntax-directed translator.

The presentation is illustrated by many small yet realistic examples and pictures, to ease the understanding of the theory and the transfer to application. Theoretical models of automata, transducers and formal grammars are extensively used, whenever practical motivations exist, without insisting too much on their formal definition. Algorithms are described in a pseudo-code to avoid the disturbing details of a programming language, yet they are straightforward to convert to executable procedures.

This book should be welcome by those willing to teach or to learn the essential concepts of syntax-directed compilation, without the need of relying on software tools and implementations. We believe that learning by doing is not always the best approach, and that over-commitment to practical work may sometimes obscure the conceptual foundations. In the case of formal languages, the elegance and simplicity of the underlying theory allows the students to acquire the fundamental paradigms of language structures, to avoid pitfalls such as ambiguity, and to adequately map structure to meaning. In this field, the relevant algorithms are simple enough to be practiced by paper and pencil. Of course, students should be encouraged to enroll in a hands-on laboratory and to experiment syntax-directed tools (like *flex* and *bison*) on realistic cases.

Intended Audiences This is primarily a textbook targeted to *graduate* (or upper-division undergraduate) *students* in computer science or computer engineering. We list as prerequisites: familiarity with some programming language models and with algorithm design; and the ability to use elementary mathematical and logical notation. If the reader or student has already taken a course on theoretical computer science including a few elements of formal language and automata theory, the time and effort needed to learn the first chapters can be reduced. But it is fair to say that this book does not compete with the many available books on the theory of formal languages and computation: we usually do not include mathematical proofs of properties, and we rely instead on examples and informal arguments. Yet mathematically oriented readers will find here many motivating examples and applications.

A large collection of *problems and solutions* complementing the numerous examples in the book is available on the authors' course web site at Politecnico di Milano. Similarly, a comprehensive set of *lecture slides* is also available on the course web site.

The Authors Thank The colleagues who have taught this book to computer engineering students, Giampaolo Agosta and Licia Sbattella; the Italian National Research Group on Formal Languages, in particular Antonio Restivo, Alessandra Cherubini, Dino Mandrioli, Matteo Pradella and Pierluigi San Pietro; our past and present Ph.D. students and teaching assistants, Alessandro Barenghi, Marcello Bersani, Andrea Di Biagio, Simone Campanoni, Silvia Lovergine, Michele Scandale, Ettore Speziale, Martino Sykora and Michele Tartara. We also acknowledge the support of ST Microelectronics Company for R.&D. projects on compilers for advanced microprocessor architectures.

The first author remembers the late Antonio Grasselli, a pioneer of computer science studies, who first fascinated him with a subject combining linguistic, mathematical, and technological aspects.

Milan, Italy
July 2013

Stefano Crespi Reghizzi
Luca Breveglieri
Angelo Morzenti

Contents

Introduction 1

1.1 Intended Scope and Audience

The information technology revolution was made possible by the invention of electronic digital machines, but without programming languages their use would have been restricted to the few people able to write binary machine code. A programming language as a text contains features coming both from human languages and from mathematical logic. The translation from a programming language to machine code is known as *compilation*.[1] Language compilation is a very complex process, which would be impossible to master without systematic design methods. Such methods and their theoretical foundations are the argument of this book. They make up a consistent and largely consolidated body of concepts and algorithms, which are applied not just in compilers, but also in other fields. Automata theory is pervasively used in all branches of informatics to model situations or phenomena classifiable as time and space discrete systems. Formal grammars on the other hand originated in linguistic research and are widely applied in document processing, in particular for the Web.

Coming to the prerequisites, the reader should have a good background in programming, but the detailed knowledge of a specific programming language is not required, because our presentation of algorithms uses self-explanatory pseudo-code. The reader is expected to be familiar with basic mathematical theories and notation, namely set theory, algebra, and logic. The above prerequisites are typically met by computer science/engineering or mathematics students with two or more years of university education.

The selection of topics and the presentation based on rigorous definitions and algorithms illustrated by many motivating examples should qualify the book for a university course, aiming to expose students to the importance of good theories and of efficient algorithms for designing effective systems. In our experience about 50 hours of lecturing suffice to cover the entire book.

[1] This term may sound strange; it originates in the early approach to the compilation of tables of correspondence between a command in the language and a series of machine operations.

S. Crespi Reghizzi et al., *Formal Languages and Compilation*,
Texts in Computer Science, DOI 10.1007/978-1-4471-5514-0_1,

The authors' long experience in teaching the subject to different audiences brings out the importance of combining theoretical concepts and examples. Moreover it is advisable that the students take advantage of well-known and documented software tools (such as classical *Flex* and *Bison*), to implement and experiment the main algorithm on realistic case studies.

With regard to the reach and limits, the book covers the essential concepts and methods needed to design simple translators based on the syntax-directed paradigm. It goes without saying that a real compiler for a programming language includes other technological aspects and know-how, in particular related to processor and computer architecture, which are not covered. Such know-how is essential for automatically translating a program to machine instructions and for transforming a program in order to make the best use of the computational resources of a computer. The study of program transformation and optimization methods is a more advanced topic, which follows the present introduction to compiler methods. The next section outlines the contents of the book.

1.2 Compiler Parts and Corresponding Concepts

There are two external interfaces to a compiler: the source language to be analyzed and translated, and the target language produced by the translator.

Chapter 2 describes the so-called syntactic methods that are generally adopted in order to provide a rigorous definition of the texts (or character strings) written in the source language. The methods to be presented are regular expressions and context-free grammars. Both belong to formal language theory, a well-established chapter of theoretical computer science.

The first task of a compiler is to check the correctness of the source text, that is, whether it complies with the syntactic definition of the source language by certain grammar rules. In order to perform the check, the algorithm scans the source text character by character and at the end it rejects or accepts the input depending on the result of the analysis. By a minimalist approach, such recognition algorithms can be conveniently described as mathematical machines or automata, in the tradition of the well-known Turing machine.

Chapter 3 covers finite automata, which are machines with a finite random access memory. They are the recognizers of the languages defined by regular expressions. Within compilation they are used for lexical analysis or scanning, to extract from the source text keywords, numbers, and in general the pieces of text corresponding to the lexical units or lexemes (or tokens) of the language.

Chapter 4 is devoted to the recognition problem for languages defined by context-free grammars. Recognition algorithms are first modeled as finite automata equipped with unbounded last-in-first-out memory or pushdown stack. For a compiler, the language recognizer is an essential component known as the syntax analyzer or parser. Its job is to check the syntactic correctness of a source text already subdivided into lexemes, and to construct a structural representation called a syntax tree. To reduce the parsing time for very long texts, the parser can be organized as a parallel program.

The ultimate job of a compiler is to translate a source text to another language. The module responsible for completing the verification of the source language rules and for producing the translation is called the semantic analyzer. It operates on the structural representation produced by the parser.

The formal models of translation and the methods used to implement semantic analyzers are in Chap. 5, which describes two kinds of transformations. Pure syntactic translations are modeled by finite or pushdown transducers. Semantic translations are performed by functions or methods that operate on the syntax tree of the source text. Such translations will be specified by a practical extension to context-free grammars, called attribute grammars. This approach, by combining the accuracy of formal syntax and the flexibility of programming, conveniently expresses the analysis and translation of syntax trees.

To give a concrete idea of compilation, typical simple examples are included: the type consistency check between variables declared and used in a programming language, the translation of high-level statements to machine instructions, and semantic-directed parsing.

For sure compilers do much more than syntax-directed translation. Static program analysis is an important example, consisting in examining a program to determine, ahead of execution, some properties, or to detect errors not covered by semantic analysis. The purpose is to improve the robustness, reliability, and efficiency of the program. An example of error detection is the identification of uninitialized variables. For code improvement, an example is the elimination of useless assignment statements.

Chapter 5 terminates with an introduction to the static analysis of programs modeled by their control-flow graph, viewed as a finite automaton. Several interesting problems can be formalized and statically analyzed by a common approach based on flow equations, and their solution by iterative approximations converging to the least fixed point.

Syntax 2

2.1 Introduction

2.1.1 Artificial and Formal Languages

Many centuries after the spontaneous emergence of natural language for human communication, mankind has purposively constructed other communication systems and languages, to be called artificial, intended for very specific tasks. A few artificial languages, like the logical propositions of Aristotle or the music sheet notation of Guittone d'Arezzo, are very ancient, but their number has exploded with the invention of computers. Many of them are intended for man-machine communication, to instruct a programmable machine to do some task: to perform a computation, to prepare a document, to search a database, to control a robot, and so on. Other languages serve as interfaces between devices, e.g., Postscript is a language produced by a text processor commanding a printer.

Any designed language is artificial by definition, but not all artificial languages are formalized: thus a programming language like Java is formalized, but Esperanto, although designed by man, is not.

For a language to be formalized (or formal), the form of sentences (or syntax) and their meaning (or semantics) must be precisely and algorithmically defined. In other words, it should be possible for a computer to check that sentences are grammatically correct, and to determine their meaning.

Meaning is a difficult and controversial notion. For our purposes, the meaning of a sentence can be taken to be the translation to another language which is known to the computer or the operator. For instance, the meaning of a Java program is its translation to the machine language of the computer executing the program.

In this book the term *formal language* is used in a narrower sense that excludes semantics. In the field of syntax, a formal language is a mathematical structure, defined on top of an alphabet, by means of certain axiomatic rules (formal grammar) or by using abstract machines such as the famous one due to A. Turing. The notions and methods of formal language are analogous to those used in number theory and in logic.

S. Crespi Reghizzi et al., *Formal Languages and Compilation*,
Texts in Computer Science, DOI 10.1007/978-1-4471-5514-0_2,

Thus formal language theory is only concerned with the form or syntax of sentences, not with meaning. A string (or text) is either valid or illegal, that is, it either belongs to the formal language or does not. Such theory makes a first important step towards the ultimate goal: the study of language translation and meaning, which will require additional methods.

2.1.2 Language Types

A language in this book is a one-dimensional communication medium, made by sequences of symbolic elements of an alphabet, called terminal characters. Actually people often refer to language as other not textual communication media, which are more or less formalized by means of rules. Thus iconic languages focus on road traffic signs or video display icons. Musical language is concerned with sounds, rhythm, and harmony. Architects and designers of buildings and objects are interested in their spatial relations, which they describe as the language of design. Early child drawings are often considered as sentences of a pictorial language, which can be partially formalized in accordance with psychological theories. The formal approach to the syntax of this chapter has some interest for non-textual languages too.

Within computer science, the term language applies to a text made by a set of characters orderly written from, say, left to right. In addition the term is used to refer to other discrete structures, such as graphs, trees, or arrays of pixels describing a discrete picture. Formal language theories have been proposed and used to various degrees also for such non-textual languages.[1]

Reverting to the main stream of textual languages, a frequent request directed to the specialist is to define and specify an artificial language. The specification may have several uses: as a language reference manual for future users, as an official standard definition, or as a contractual document for compiler designers to ensure consistency of specification and implementation.

It is not an easy task to write a complete and rigorous definition of a language. Clearly the exhaustive approach, to list all possible sentences or phrases, is unfeasible because the possibilities are infinite, since the length of sentences is usually unbounded. As a native language speaker, a programmer is not constrained by any strict limit on the length of phrases to be written. The problem to represent an infinite number of cases by a finite description can be addressed by an enumeration procedure, as in logic. When executed, the procedure generates longer and longer sentences, in an unending process if the language to be modeled is not finite.

This chapter presents a simple and established manner to express the rules of the enumeration procedure in the form of rules of a generative grammar (or syntax).

[1] Just two examples and references: tree languages [5] and picture (or two-dimensional) languages [4, 6].

2.1.3 Chapter Outline

The chapter starts with the basic components of language theory: alphabet, string, and operations, such as concatenation and repetition, on strings and sets of strings.

The definition of the family of regular languages comes next.

Then the lists are introduced as a fundamental and pervasive syntax structure in all kinds of languages. From the exemplification of list variants, the idea of linguistic abstraction grows out naturally. This is a powerful reasoning tool to reduce the varieties of existing languages to a few paradigms.

After discussing the limits of regular languages, the chapter moves to context-free grammars. Following the basic definitions, the presentation focuses on structural properties, namely, equivalence, ambiguity, and recursion.

Exemplification continues with important linguistic paradigms such as: hierarchical lists, parenthesized structures, polish notations, and operator precedence expressions. Their combination produces the variety of forms to be found in artificial languages.

Then the classification of some common forms of ambiguity and corresponding remedies is offered as a practical guide for grammar designers.

Various transformations of rules (normal forms) are introduced, which should familiarize the reader with the modifications often needed for technical applications, to adjust a grammar without affecting the language it defines.

Returning to regular languages from the grammar perspective, the chapter evidences the greater descriptive capacity of context-free grammars.

The comparison of regular and context-free languages continues by considering the operations that may cause a language to exit or remain in one or the other family. Alphabetical transformations anticipate the operations studied in Chap. 5 as translations.

A discussion of unavoidable regularities found in very long strings, completes the theoretical picture.

The last section mentions the Chomsky classification of grammar types and exemplifies context-sensitive grammars, stressing the difficulty of this rarely used model.

2.2 Formal Language Theory

Formal language theory starts from the elementary notions of alphabet, string operations, and aggregate operations on sets of strings. By such operations complex languages can be obtained starting from simpler ones.

2.2.1 Alphabet and Language

An *alphabet* is a finite set of elements called *terminal symbols* or *characters*. Let $\Sigma = \{a_1, a_2, \ldots, a_k\}$ be an alphabet with k elements, i.e., its *cardinality* is $|\Sigma| = k$.

A *string* (also called a *word*) is a sequence (i.e., an ordered set possibly with repetitions) of characters.

Example 2.1 Let $\Sigma = \{a, b\}$ be the alphabet. Some strings are: $aaba$, aaa, $abaa$, b.

A *language* is a set of strings on a specified alphabet.

Example 2.2 For the same alphabet $\Sigma = \{a, b\}$ three examples of languages follow:

$$\begin{aligned} L_1 &= \{aa, aaa\} \\ L_2 &= \{aba, aab\} \\ L_3 &= \{ab, ba, aabb, abab, \ldots, aaabbb, \ldots\} \\ &= \text{set of strings having as many } a\text{'s as } b\text{'s} \end{aligned}$$

Notice that a formal language viewed as a set has two layers: at the first level there is an unordered set of non-elementary elements, the strings. At the second level, each string is an ordered set of atomic elements, the terminal characters.

Given a language, a string belonging to it is called a *sentence* or *phrase*. Thus $bbaa \in L_3$ is a sentence of L_3, whereas $abb \notin L_3$ is an *incorrect* string.

The *cardinality* or size of a language is the number of sentences it contains. For instance, $|L_2| = |\{aba, aab\}| = 2$. If the cardinality is finite, the language is called *finite*, too. Otherwise, there is no finite bound on the number of sentences, and the language is termed *infinite*. To illustrate, L_1 and L_2 are finite, but L_3 is infinite.

One can observe a finite language is essentially a collection of words[2] sometimes called a *vocabulary*. A special finite language is the *empty* set or *language* $\emptyset$, which contains no sentence, $|\emptyset| = 0$. Usually, when a language contains just one element, the set braces are omitted writing e.g., abb instead of $\{abb\}$.

It is convenient to introduce the notation $|x|_b$ for the number of characters b present in a string x. For instance:

$$|aab|_a = 2, \qquad |aba|_a = 2, \qquad |baa|_c = 0$$

The *length* $|x|$ of a string x is the number of characters it contains, e.g.: $|ab| = 2$; $|abaa| = 4$.

Two strings

$$x = a_1 a_2 \ldots a_h, \qquad y = b_1 b_2 \ldots b_k$$

are *equal* if $h = k$ and $a_i = b_i$, for every $i = 1, \ldots, h$. In words, examining the strings from left to right their respective characters coincide. Thus we obtain

$$aba \neq baa, \qquad baa \neq ba$$

[2] In mathematical writings the terms *word* and *string* are synonymous, in linguistics a word is a string having a meaning.

2.2.1.1 String Operations

In order to manipulate strings it is convenient to introduce several operations. For strings

$$x = a_1 a_2 \ldots a_h, \qquad y = b_1 b_2 \ldots b_k$$

concatenation[3] is defined as

$$x.y = a_1 a_2 \ldots a_h b_1 b_2 \ldots b_k$$

The dot may be dropped, writing xy in place of $x.y$. This essential operation for formal languages plays the role addition has in number theory.

Example 2.3 For strings

$$x = \text{well}, \qquad y = \text{in}, \qquad z = \text{formed}$$

we obtain

$$xy = \text{wellin}, \qquad yx = \text{inwell} \neq xy$$
$$(xy)z = \text{wellin.formed} = x(yz) = \text{well.informed} = \text{wellinformed}$$

Concatenation is clearly non-commutative, that is, the identity $xy \neq yx$ does not hold in general. The *associative* property holds:

$$(xy)z = x(yz)$$

This permits to write without parentheses the concatenation of three or more strings. The length of the result is the sum of the lengths of the concatenated strings:

$$|xy| = |x| + |y| \tag{2.1}$$

2.2.1.2 Empty String

It is useful to introduce the concept of *empty* (or null) *string*, denoted by Greek epsilon ε, as the only string satisfying the identity

$$x\varepsilon = \varepsilon x = x$$

for every string x. From equality (2.1) it follows the empty string has length zero:

$$|\varepsilon| = 0$$

From an algebraic perspective, the empty string is the neutral element with respect to concatenation, because any string is unaffected by concatenating ε to the left or right.

[3] Also termed *product* in mathematical works.

The empty string should not be confused with the empty set: in fact $\emptyset$ as a language contains no string, whereas the set $\{\varepsilon\}$ contains one, the empty string.

A language L is said to be *nullable* iff it includes the empty string, i.e., iff $\varepsilon \in L$.

2.2.1.3 Substrings

Let $x = uyv$ be the concatenation of some, possibly empty, strings u, y, v. Then y is a *substring* of x; moreover, u is a *prefix* of x, and v is a *suffix* of x. A substring (prefix, suffix) is called *proper* if it does not coincide with string x.

Let x be a string of length at least k, $|x| \geq k \geq 1$. The notation $Ini_k(x)$ denotes the prefix u of x having length k, to be termed the *initial* of length k.

Example 2.4 The string $x = aabacba$ contains the following components:
prefixes: $a, aa, aab, aaba, aabac, aabacb, aabacba$
suffixes: $a, ba, cba, acba, bacba, abacba, aabacba$
substrings: all prefixes and suffixes and the internal strings such as $a, ab, ba, bacb, \ldots$
Notice that bc is not a substring of x, although both b and c occur in x. The initial of length two is $Ini_2(aabacba) = aa$.

2.2.1.4 Mirror Reflection

The characters of a string are usually read from left to right, but it is sometimes requested to reverse the order. The *reflection* of a string $x = a_1 a_2 \ldots a_h$ is the string $x^R = a_h a_{h-1} \ldots a_1$. For instance, it is

$$x = \text{roma} \qquad x^R = \text{amor}$$

The following identities are immediate:

$$\left(x^R\right)^R = x \qquad (xy)^R = y^R x^R \qquad \varepsilon^R = \varepsilon$$

2.2.1.5 Repetitions

When a string contains repetitions it is handy to have an operator denoting them. The mth *power* ($m \geq 1$, integer) of a string x is the concatenation of x with itself $m - 1$ times:

$$x^m = \underbrace{xx \ldots x}_{m \text{ times}}$$

By stipulation the zero power of any string is defined to be the empty string.

The complete definition is

$$\begin{cases} x^m = x^{m-1}x, & m > 0 \\ x^0 = \varepsilon \end{cases}$$

Examples:

$x = ab$ $\quad x^0 = \varepsilon$ $\quad x^1 = x = ab$ $\quad x^2 = (ab)^2 = abab$
$y = a^2 = aa$ $\quad y^3 = a^2a^2a^2 = a^6$
$\varepsilon^0 = \varepsilon$ $\quad \varepsilon^2 = \varepsilon$

When writing formulas, the string to be repeated must be parenthesized, if longer than one. Thus to express the second power of ab, i.e., $abab$, one should write $(ab)^2$, not ab^2, which is the string abb.

Expressed differently, we assume the power operation takes *precedence* over concatenation. Similarly reflection takes precedence over concatenation: e.g., ab^R returns ab, since $b^R = b$, while $(ab)^R = ba$.

2.2.2 Language Operations

It is straightforward to extend an operation, originally defined on strings, to an entire language: just apply the operation to all the sentences. By means of this general principle, previously defined string operations can be revisited, starting from those having one argument.

The reflection of a language L is the set of strings that are the reflection of a sentence:

$$L^R = \{x \mid \underbrace{x = y^R \wedge y \in L}_{\text{characteristic predicate}}\}$$

Here the strings x are specified by the property expressed in the so-called characteristic predicate.

Similarly the set of proper prefixes of a language L is

$$\text{Prefixes}(L) = \{y \mid x = yz \wedge x \in L \wedge y \neq \varepsilon \wedge z \neq \varepsilon\}$$

Example 2.5 (Prefix-free language) In some applications the loss of one or more final characters of a language sentence is required to produce an incorrect string. The motivation is that the compiler is then able to detect inadvertent truncation of a sentence.

A language is *prefix-free* if none of the proper prefixes of sentences is in the language; i.e., if the set Prefixes(L) is disjoint from L.

Thus the language $L_1 = \{x \mid x = a^n b^n \wedge n \geq 1\}$ is prefix-free since every prefix takes the form $a^n b^m, n > m \geq 0$ and does not satisfy the characteristic predicate.

On the other hand, the language $L_2 = \{a^m b^n \mid m > n \geq 1\}$ contains a^3b^2 as well as its prefix a^3b.

Similarly, operations on two strings can be extended to two languages, by letting the first and second argument span the respective language, for instance *concatena-*

tion of languages L' and L'' is defined as

$$L'L'' = \left\{xy \mid x \in L' \wedge y \in L''\right\}$$

From this the extension of the mth *power* operation on a language is straightforward:

$$L^m = L^{m-1}L, \quad m > 0$$
$$L^0 = \{\varepsilon\}$$

Some special cases follow from previous definitions:

$$\emptyset^0 = \{\varepsilon\}$$
$$L.\emptyset = \emptyset.L = \emptyset$$
$$L.\{\varepsilon\} = \{\varepsilon\}.L = L$$

Example 2.6 Consider the languages:

$$L_1 = \left\{a^i \mid i \geq 0, \text{even}\right\} = \{\varepsilon, aa, aaaa, \ldots\}$$
$$L_2 = \left\{b^j a \mid j \geq 1, \text{odd}\right\} = \{ba, bbba, \ldots\}$$

We obtain

$$\begin{aligned} L_1L_2 &= \left\{a^i.b^j a \mid (i \geq 0, \text{even}) \wedge (j \geq 1, \text{odd})\right\} \\ &= \left\{\varepsilon ba, a^2ba, a^4ba, \ldots, \varepsilon b^3a, a^2b^3a, a^4b^3a, \ldots\right\} \end{aligned}$$

A common error when computing the power is to take m times the *same* string. The result is a different set, included in the power:

$$\left\{x \mid x = y^m \wedge y \in L\right\} \subseteq L^m, \quad m \geq 2$$

Thus for $L = \{a, b\}$ with $m = 2$ the left part is $\{aa, bb\}$ and the right part is $\{aa, ab, ba, bb\}$.

Example 2.7 (Strings of finite length) The power operation allows a concise definition of the strings of length not exceeding some integer k. Consider the alphabet $\Sigma = \{a, b\}$. For $k = 3$ the language

$$\begin{aligned} L &= \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb\} \\ &= \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \end{aligned}$$

may also be defined as

$$L = \{\varepsilon, a, b\}^3$$

Notice that sentences shorter than k are obtained using the empty string of the base language.

Slightly changing the example, the language $\{x \mid 1 \leq |x| \leq 3\}$ is defined, using concatenation and power, by the formula

$$L = \{a, b\}\{\varepsilon, a, b\}^2$$

2.2.3 Set Operations

Since a language is a set, the classical set operations, union ($\cup$), intersection ($\cap$), and difference ($\setminus$), apply to languages; set relations, inclusion ($\subseteq$), strict inclusion ($\subset$), and equality ($=$) apply as well.

Before introducing the complement of a language, the notion of *universal language* is needed: it is defined as the set of all strings of alphabet Σ, of any length, including zero.

Clearly the universal language is infinite and can be viewed as the union of all the powers of the alphabet:

$$L_{\text{universal}} = \Sigma^0 \cup \Sigma \cup \Sigma^2 \cup \cdots$$

The *complement* of a language L of alphabet Σ, denoted by $\neg L$, is the set difference:

$$\neg L = L_{\text{universal}} \setminus L$$

that is, the set of the strings of alphabet Σ that are not in L. When the alphabet is understood, the universal language can be expressed as the complement of the empty language,

$$L_{\text{universal}} = \neg\emptyset$$

Example 2.8 The complement of a finite language is always infinite, for instance the set of strings of any length except two is

$$\neg\big(\{a, b\}^2\big) = \varepsilon \cup \{a, b\} \cup \{a, b\}^3 \cup \cdots$$

On the other hand, the complement of an infinite language may or may not be finite, as shown on one side by the complement of the universal language, on the other side by the complement of the set of even-length strings with alphabet $\{a\}$:

$$L = \big\{a^{2n} \mid n \geq 0\big\} \qquad \neg L = \big\{a^{2n+1} \mid n \geq 0\big\}$$

Moving to set difference, consider alphabet $\Sigma = \{a, b, c\}$ and languages

$$L_1 = \big\{x \mid |x|_a = |x|_b = |x|_c \geq 0\big\}$$
$$L_2 = \big\{x \mid |x|_a = |x|_b \wedge |x|_c = 1\big\}$$

Then the differences are

$$L_1 \setminus L_2 = \varepsilon \cup \big\{x \mid |x|_a = |x|_b = |x|_c \geq 2\big\}$$