11-2012

Toward an isomorphic diagram of the Backus-Naur form.

Neil A. Smith
University of Louisville

TOWARD AN ISOMORPHIC DIAGRAM OF THE BACKUS-NAUR FORM

By
Neil A. Smith
B.Sc., University of Louisville, 2009

A Thesis
Submitted to the Faculty of the
University of Louisville
J.B. Speed School of Engineering
as Partial Fulfillment of the Requirements
for the Professional Degree

MASTER OF ENGINEERING

Department of Computer Engineering & Computer Science

November 2012

TOWARD AN ISOMORPHIC DIAGRAM OF THE BACKUS-NAUR FORM

Submitted By: _____

Neil A. Smith

_____

November 30, 2012

by the following Reading and Examination Committee:

_____

Dr. Dar-jen Chang, Thesis Director

_____

Dr. Ming Ouyang

_____

Dr. Tim Hardin

ACKNOWLEDGEMENTS

Prior to the writing of this thesis, I'd "cut my teeth" in the lab of Dr. Patrick Shafto, whose professorship was—and is at present—in the Psychological & Brain Sciences Department, here, at University of Louisville. While Dr. Shafto's work has often crossed the academic divide between the social and computer sciences, my experience *there* would not directly pertain to the field of language and compiler design; those interests of my thesis advisor, Dr. Dar-jen Chang. This circumstance actually proved to be something fruitful for me as a student, since I was afforded the opportunity to undertake research that I, otherwise, would likely not have considered. Upon being asked to do an exposition in the foregoing field, however, I admittedly was uncertain of myself. In fact, I wasn't even aware that compiler design was an active area of research. Moreover, the one course that had exposed me to the subject, simply covered fundamentals, and being that the goal of translation was principally realized decades prior, I wondered: "What problem is left to explore?"

It is often the case, however, that extant solutions can be improved. After all, a solution that is optimal with respect to one dimension, may be inferior with respect to another. Knowing this, my search for *the problem* to solve, took a different approach than what I've come to know to be the "usual"—that is, reading the pertinent papers from leading journals. Instead, I decided to consider the fundamental ways that people have solved problems in the field, such as the use of formal language theory and, particularly the tools that have been devised to explain abstract concepts. Perhaps, my prior experience in a psychology lab had affected my thinking. But, this *thought process* led me to the concept that is the focus of this work: a new way of representation, more specifically, a novel diagram. Upon deciding, once and for all, that this would be my thesis topic, I thought: "OK, so I've created a diagram, but does this concept provide the theoretical depth required for a thesis?" Thus, began a new search, one to answer the preceding question. For me, this led to the discovery

that diagrammatic representation has had a profound impact on the world, and that the effect it has had on science, truly could not be understated.

I was surprised to find that plenty of researchers had published articles, even volumes on diagrams and the mental phenomena associated with them. I recall, one night in particular, scouring the internet for evidence of research dedicated to diagrams. When I came across a link to the website, "Diagrams 2012: The 7th international Conference on the Theory and Application of Diagrams", I became excited and said to my wife, who was sitting nearby: "Hey, Diagrams 2012! There is a conference dedicated to diagrams!" To which, she replied: "Great...Neil, you should have married a nerd." It was at that time, I realized that this idea would be a worthwhile investigation.

This thesis would not have materialized without the support of others along the way. I thank Dr. Chang for his help in gathering my thoughts, and his encouragement in completing this thesis. In addition, I learned so much working in the Computational and Cognitive Science Lab with Dr. Shafto, and am thankful to him for providing that opportunity. Finally, I thank my thesis committee for their insight and the dedication of their time to this endeavor.

ABSTRACT

Computer scientists studying formal languages have made use of a variety of representations to both reason, and communicate their ideas to others. Symbolic representations have proved useful for rigorously defining the theoretical objects of the preceding topics; however, research shows that diagrammatic representations are as fundamental to these subjects. Previous research in this domain has typically been interested in studying the semantics that a particular representation is intended to capture. By contrast, this treatise considers the importance of the *format* of the representations themselves, and how format influences the ability of a person to uncover characteristics, relevant to the problem domain. More specifically, this thesis investigates the established formalisms that have been devised to describe formal languages, and introduces a novel concept, an augmented syntax graph. This graph, an isomorphism of the Backus-Naur form, is shown to have application in visualizing properties that are pertinent to some parsing algorithms.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

In more recent years, issues that were once thought to elucidate academic boundaries have become the concern of subjects as diverse as linguistics, philosophy, psychology, and computer science. Central to these issues is how to represent information. Of particular interest are the classical questions of how humans can better communicate, reason, and make decisions. In addition, there exists the avant-garde theme of how to better communicate with machines. One success given by this interdisciplinary work has been the development of, so-called "high-level" programming languages, which allow people to program computers in a more natural way; relative to that afforded by instruction sets, native to machines.

A high-level programming language is formally specified by a *formal grammar*—almost universally, a *context-free* grammar—which is a linguistic structure. While these linguistic formulations yield a precise description of a given language, the properties of these grammars can be challenging to realize by solely analyzing them in their symbolic form. As such, researchers have introduced methods for uncovering characteristics that are critical for translation programs to successfully derive the valid words of a language. More generally, researchers have discovered alternative ways of representing the information given by these grammars. As a research discipline, *representation*, can be abstract and the terminology used in the literature, often ambiguous; therefore, one goal of this work is to acquaint the reader with the themes and operational definitions requisite to studying systems of representation, with a particular focus on diagrammatic forms. Moreover, this treatise will recapitulate established formalisms for representing language, and explore a novel diagrammatic formalism and its application to compiler design.

The present chapter will identify and briefly address questions that span the corpus, from the philosophical concept of imagery to the role of diagrammatic rea-

soning in science. We begin the discussion by asking an elementary question: "What is representation?". The remainder of this work is outlined as follows: **Chapter 2** and **Chapter 3** offer a review of the concept of representing language in symbolic form, and that in graph representation, respectively; **Chapter 4** explores a novel way of augmenting graph representations and **Chapter 5** builds upon this work, by investigating a pedagogical application that puts to use the foregoing formalism; we conclude with **Chapter 6**, which outlines what we feel are fecund areas of future investigation into this diagrammatic system.

1.1 WHAT IS REPRESENTATION?

We make use of these *things*, countless times, everyday. In fact, we're often not conscious of them; yet, without *representations*, reasoning would be by theory, impossible. What representations are is much more cumbersome to define than what they do, which can be stated rather succinctly. That is, they connect two worlds, one of which is typically more accessible than the other. Adopting the language of Palmer (1978), we may refer to a *representing world* and a *represented world*. The function of the representing world is to act as a surrogate for the represented world by maintaining certain relationships between the two.

More concretely, consider the amount of fuel in your automobile's tank; of course, it is often the case that you need to know this quantity and, generally, it is made accessible by a gauge on your dashboard, such as the mechanical indicator depicted in Figure 1.1. Our represented world, the quantity of fuel, is the product of the dimensions of the liquid. Therefore if your tank is, for instance, rectangular in shape then the amount of fuel can be precisely stated as the product of the length, width, and height of the fuel. Notice that our representing world, the fuel gauge, represents the amount of liquid within the tank by a single dimension, the height. Thus, the dial acts as a surrogate to the quantity of fuel, since that quantity isn't accessible (at least, not readily), and this representing world preserves the relation of height in our represented world.

The foregoing system is just one of many possible ways of representing the amount of available fuel. For instance, digital fuel gauges exist, which report the quantity of fuel as a decimal number in gallon units. Two representations are informationally equivalent if they preserve the same information about the represented world; furthermore, if these are structurally different representations, they are referred to as *isomorphisms*. Otherwise, they are *nonequivalent*. While both mechanical and

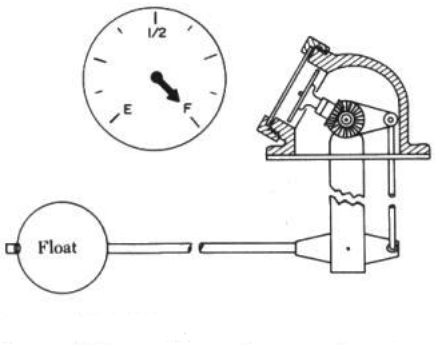

Figure 1.1: *Mechanical fuel gauge* (FAA, 1976). This device functions by representing the angular displacement of the float arm, which rotates along a 90° axis.

digital fuel meters preserve the same dimension (i.e. the height of the fuel), they are nonequivalent due, in part, to the differences in resolution of information. In other words, the mechanical representation is able to take on a continuous range of values, whereas, the digital representation discretizes the information. Even in cases where two representations preserve the same information, they may still differ through the *medium* of representation (e.g. symbolic versus diagrammatic), as well as the way in which the they evolve over time. In other words, for two representations to be considered *completely equivalent*, the same relations must be maintained and represented in precisely the same way.

1.2 SYMBOLIC VERSUS DIAGRAMMATIC REPRESENTATION: DOES A DISTINCTION EXIST?

Intuitively people distinguish between pictures (like diagrams) and symbols (like the elements of written language), and interact with each format in different ways. While people often view these as mutually exclusive categories, does such a distinction truly exist?

Researchers attempting to answer this question have typically identified two forms of representation: *internal representations*, systems that the mind constructs and manipulates in memory, and *external representations*, those created in the physical world, existing "outside of the head." These systems have historically been given separate treatments in literature, with much research into internal representations holding that external representations function, simply, as input for the mind, whereupon reasoning is thought to begin. Thus, the literature on internal representation generally views the boundaries of the mind as the only relevant area of study (Zhang, 1997).

Whether one is recalling their route home from work, determining if a space will accommodate some object, or imagining whether a horse's knee is higher from the ground than the tip of it's tail (Kosslyn et al., 1981), to many it would seem that the "mind's eye" would be requisite for completing the task. In fact, some have ascribed to the hypothesis that mental imagery is the only medium through which the mind operates (Woodworth, 1938). Indeed the experience of mental imagery seems to be a universal phenomenon; yet, some philosophers have argued against its existence. Even among those who concede that imagery exists at some level, there is considerable debate. This is often centered around the nature of internal representation. One camp, the *pictorialists*, holds that the internal representations of an image are given by a format[1] that is distinct from other internal content (e.g.

[1]Some authors suggest that the debate is about representations in analog vs. digital format,

verbal information), and that this representation is a *kind* of picture having, at least, some semblance to a picture in the external world (Kosslyn, 1981). The other camp, the *descriptionalists*, ascribes to the thesis that there is only one format by which the mind represents information—viz., by way of a symbolic "propositional" form—and the experience of a mental image is epiphenomenal (Pylyshyn, 1973)[1].

Experimental investigations of mental imagery have offered support for both philosophical positions. Some of the more remarkable studies supporting the pictorialists' viewpoint have come from R. N. Shepard and his associates. Shepard and Metzler (1971) provided handouts to participants, containing two-dimensional line drawings of three-dimensional objects. The objects depicted on each handout were shown in horizontal pairs, with each object appearing in a different orientation (see Figure 1.2). Subjects were asked to determine whether the drawings in a pair were of the same object. The researchers found a positive, linear relationship between the reaction time and the angular difference of each drawing, suggesting that participants were rotating *mental images* in order to check the equivalence of the drawings. Offering a defense of the descriptionalists' position, Pylyshyn (1981) argued that the participants' *tacit knowledge* was brought to bear on the experimental task. Since people have experienced object rotation in the external world, Pylyshyn's tacit knowledge framework suggests that subjects would attempt to simulate the problem mentally, as they would expect it to be solved in the physical world; thus, calling into question the validity of the explanation posited by the pictorialists.

While the mental imagery debate is most pertinent to those who assume the workings of the mind are exclusive to the boundaries of the brain, "externalists" do not place such limits on cognition. Rather, proponents of this school of thought argue that external facilities or processes are necessary for reasoning. Furthermore, external

such that pictures are assumed to be analog, and symbols are assumed to be digital representations.

[1]In other words, although a cognitive process might appear to invoke a mental image and, further, implicate a visual percept, there is nothing *functional* about the phenomena; moreover, if the process were somehow disentangled from the imaging such cognitions would continue un-impeded.

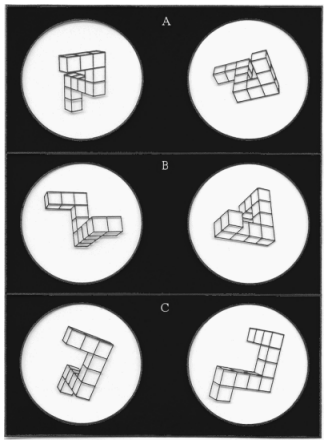

Figure 1.2: *Stimuli used in Shepard and Metzler (1971).*

representations are not considered mere inputs for the mind but are thought to play a role in the reasoning process, as it is assumed to be a process occurring not only in the head, but in the environment and with the medium of representation (e.g. paper and pencil; Blair, 2003). Consequently, such researchers have examined the characteristic features of *external representations* in order to find the basis by which people classify these systems. For instance, Larkin and Simon (1995) suggested that diagrammatical representations are defined by two-dimensions, whereas sentential representations are represented temporally, along a single dimension.

Shimojima (2001) offered what is, perhaps, the most comprehensive work on the "graphic-linguistic" distinction to date, identifying seven candidate hypotheses established in the literature. One of the more popular proposals suggests that diagrams "resemble" that which they are intended to represent, whereas, sentential representations do not. More formally, such representations are referred to as *homomorphic* to the represented world. Shimojima provided compelling arguments against the preceding, each of the other established definitions and, ultimately, argued in favor of his own: Graphical representations satisfy constraints given by natural laws (called "nomic constraints"), whereas, sentential representations are *solely* subject to constraints of convention (called "stipulative constraints"). While Shimojima's definition

is a valuable insight, it too has been argued against (Feeney and Webber, 2003). Indeed, people intuitively *see* a difference between representations of diagrammatic and symbolic form; however, it isn't clear whether there is a single formal distinction that can explain all cases.

1.3 DIAGRAMMATIC REPRESENTATION: BETTER THAN A SYMBOLIC SYSTEM?

No matter where reasoning transpires or what characteristics typify representational objects, people use varied representational forms to communicate and learn, at different times and in different contexts. But what makes one context better suited for a particular representational system than another? Bold claims have been made as to the efficacy of diagrammatic systems in comparison to sentential representations, in all sorts of settings. Thus the aphorism: "A picture is worth a thousand words"; empirical research into the usefulness of "extra-visual" modalities has not necessarily followed this intuition.

Modern logic has generally been concerned with valid reasoning in one form, a symbolic system, while other representational formats have been assumed to lack the characteristics necessary for formal logic. While some limiting constraints of two-dimensions in the plane can easily be seen—for instance, graph theorists may cite, so-called "edge crossing problems"—more recently, researchers have challenged the foregoing assumption, attempting to express logical systems using diagrammatic representations. Inspiration for these systems was initially given by heuristic tools devised before the modern era: Euler diagrams, Venn diagrams, and Existential graphs (Shin, 1994). To illustrate these diagrams, consider the following syllogism[1]:

1. All A are B.
2. No B are C.
3. No C are A.

Figure 1.3: *A Syllogism in sentential form.*

Leonherd Euler, who was arguably the world's greatest mathematician, intro-

[1]Statements (1) and (2) are the *premises* of the argument, statement (3) is the *conclusion*.

duced a diagrammatic system, known as Euler diagrams (or "Eulerian circles") which use inclusion, exclusion, and intersection relations by closed curves to represent syllogisms (cf. Euler, 1768). Figure 1.4 demonstrates the premises of our example argument, using an Euler diagram. John Venn, another notable mathematician, criticized the Euler diagram for being too restrictive in some circumstances but too inconsistent in others. He designed a scheme intended to overcome what he saw as shortcomings of the Eulerian system (cf. Venn, 1881). Both of these diagrams can resemble each other quite closely, often leading students to confuse the two systems. The most distinct characteristic of Venn's system is that the empty set is denoted by shaded regions. A more subtle, but equally important difference between them is that Venn's uses circular forms in a consistent configuration[1].

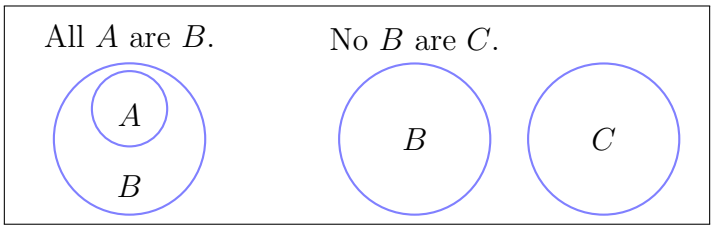

Figure 1.4: *An Euler diagram.*

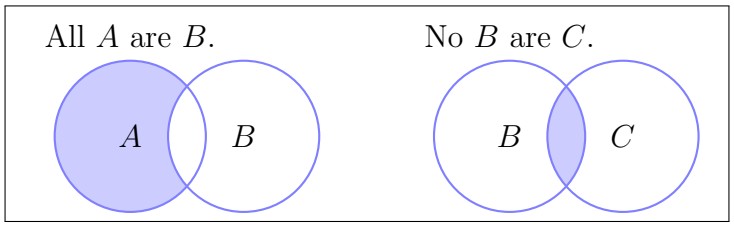

Figure 1.5: *A Venn diagram.*

Both Euler and Venn diagrams have been used extensively as tools for teaching set-theoretic methods and deductive reasoning. However, since their introduction, scholars have identified qualities of these systems that make them incapable of representing certain logical statements, such as disjunctions (Barwise and Etchemendy,

[1]By examining Figure 1.4 and Figure 1.5, you can see that the size and position of the circles may vary in the Eulerian system; whereas, these properties remain unchanged in Venn's system.

1995). Pierce (1933) proposed altering Venn's system by replacing the shaded regions with the symbol 'o', denoting certain existential qualities with 'x', and connecting these symbols with lines, as a way to overcome these limitations. Figure 1.6 demonstrates a syllogism, which is difficult for Euler and Venn diagrams to represent: "All A are B or No A are B". More recently, Shin (1994) gave a proof that showed with certain enhancements, Venn's system could provide a sound and logically complete system of representation, equivalent to symbolic formalisms.

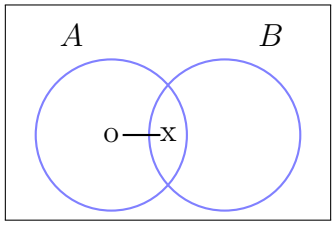

Figure 1.6: A diagram representing the syllogism "All A are B or No A are B" using Pierce's extension.

While the logical status of diagrams has been a major vein of research, other programs have focused on the heuristic power of these representations. Sato et al. (2010) actually tested the usefulness of these systems as a pedagogical tool. This study divided naive participants into three groups: Euler group, Venn group, and Linguistic group. Each participant received a list of syllogistic reasoning questions, which were subject to their group designation. The questionnaires differed by their representational format: Euler diagram (Figure 1.4), Venn diagram (Figure 1.5), or sentential representation (Figure 1.3), respectively. Each question consisted of two premises and multiple conclusions, and based on the given premises, the subjects were asked to determine which was the *valid* conclusion. They found that participants in the Euler and Venn groups performed significantly better than the Linguistic group, with the Euler group outperforming the Venn group.

The Shin (1994) and Sato et al. (2010) studies highlight an important observation: Diagrammatic expressiveness appears to be inversely proportional to visual

clarity. Concretely, as Euler diagrams evolved into Venn diagrams and, further into more formal systems (like Pierce's extension), their utility as a heuristic tool was reduced. Hence, one must balance between clarity and expressiveness when representing information diagrammatically.

1.4 WHAT ROLE DOES DIAGRAMMATIC REPRESENTATION PLAY IN SCIENCE?

Reasoning with diagrams is commonly relied upon to explain abstract concepts, moreover, science is rich with abstract ideas, and thus diagrams pervade the scientific literature. Take, for instance, the subject of mathematics which Galileo spoke of in the 17th century: "The great book of nature can be read only by those who know the language in which it was written. And this language is mathematics." It isn't a stretch to claim that scientific reasoning would be impossible without mathematics and, similarly, it might well be impossible to reason about certain mathematical objects without diagrammatic representations. For example, it would be peculiar for even a single chapter of a modern mathematics textbook to be void of a diagram, or any mathematics lecture failing to make use of a graph or figure at some point. After all, an equation is no more a combination of symbols, than it is a line drawn along a coordinate system; rather, it is a *representation* of an abstract notion existing in the mind [1]. The question explored in this section is: Why does expressing information diagrammatically seem to allow a scientist to realize characteristics of a system, more readily, than when that same information is expressed using other representational forms?

It has been suggested that invariant features of a system are the foundation of naturalistic explanatory models. Gruber argues that through altering representational format, one may discover these invariant properties: "thinking moves from one modality to another, from visual images to sketches, to words and equations explaining (that is, conveying the same meaning as) the visualizations. The thinker is pleased to discover that certain structures remain invariant under these transformations: these are his ideas." Following this notion, it is not necessarily so, then,

[1] Some philosophers of mathematics believe that all abstract objects that exist mathematically, exist physically; however, the existential status of mathematical objects is outside the scope of this work.

that diagrams enable a concept to be more easily understood than another modality. Rather, it seems to suggest that understanding a concept is a process that *evolves* by varying representations and, ultimately, when a certain level of understanding is reached, one is capable of constructing a diagram. Perhaps, at that point understanding proceeds rapidly, creating the *perception* that diagrams offer special clarity over other forms of representation.

Whether or not diagrams are truly superior in fostering scientific creativity, it seems clear that once formulated, a diagram *can* act as heuristic mechanism in conveying complex ideas. Larkin and Simon argue this point from a computational perspective: The usefulness of a representation is determined by how it is coupled with the operations that are applicable to it. They suggest that searching a representation is analogous to a computer algorithm, searching a data structure. And, that a diagram can lead to a speed-up over sentential representations due, in part, to the way in which information is structured and searched. Cheng and Simon (1993) follow up the foregoing work, by showing that some early scientific models of physical phenomena were derived using diagrammatic representations. For instance, they analyze Galileo's solution of Proposition 30, which is found in his Two New Sciences. His solution was derived by combining diagrams, despite the fact that he could have made the same discovery through manipulations of kinematic laws encoded in algebraic formulae.

CHAPTER 2
SYMBOLIC REPRESENTATION OF LANGUAGE

Prior to the introduction of high-level programming languages, programs were written directly in machine code—that is, sequences of *0's* and *1's* or mnemonics—with the intention of executing these instructions on a particular architecture. As such, the task of writing computer programs of appreciable scope was often tedious and difficult to manage. Moreover, it created the challenge of *porting* a program written for a specific architecture to that of another. Thus, computer scientists began working on abstractions to facilitate a more natural way for humans to communicate with machines, and to remedy the issue of portability.

ALGOL 60, the result of an international team—of which, John Backus and Peter Naur were members—was one of the earliest machine independent, high-level programming languages. While the language itself was a success, the notation used to specify the language would have a more lasting impact. In the *Report on the Algorithmic Language ALGOL 60* (Backus et al., 1960), the authors published a formal description of their language, using a notation that would come to be known as *Backus-Naur Form*, or BNF (Backus, 1978). This formalism was remarkable to computer scientists, since it could be used to not only express what language should be accepted by a translation program, but also because the formalism could be used to create a special kind of tool, the *parser generator* (Linz, 2006).

In addition to the practical applications afforded by BNF, it was soon discovered that this formalism was equivalent to a class of grammars, which was previously introduced by the linguist, Noam Chomsky (Ginsburg and Rice, 1962). This discovery sparked the interest of both theoreticians in computer science, as well as those studying natural language.

2.1 FUNDAMENTALS

Colloquially, *language* is most often used to refer to a specific kind of linguistic system (e.g. English) and, generally, people have a vague definition of the term. By contrast, in this treatise we prescribe a strict definition to *language*: the set of all grammatically correct sentences, where a *sentence* consists of a set of words, and each *word* is derived from a finite set of *symbols* (Chomsky, 1957). Notice, by the foregoing definition that natural languages are ambiguous in that, given a particular sentence, it is debatable as to whether the sentence is "grammatically correct"; thus, a member of the language. For formal languages, this is an undesirable property and, as we'll see, having a formal method to provably say which sentences are members of a given language is offered by an important grammatical formalism, called *formal grammar*.

Formal language theory concerns itself with sets of strings, or *words* which are constructed from *symbols* drawn from some *alphabet*. The theory strips away the meaning, or *semantics*, of language in order to uncover its generative properties.

Definition 2.1.1 *An alphabet,* $\mathbf{\Sigma}$*, is a finite set of symbols.*

For example, the English alphabet consists of 52 lowercase and uppercase letters and, if we are to consider sentences, will contain spaces and punctuation marks. The most important aspects of formal languages, however, can be modeled using a much simpler alphabet, such as $\mathbf{\Sigma} = \{0, 1\}$ or $\mathbf{\Sigma} = \{\text{a, b}\}$.

Definition 2.1.2 *A word (or string) over an alphabet* $\mathbf{\Sigma}$ *is a finite-length* $\mathbf{\Sigma}$*-sequence.*

A word can be written as $x = a_i...a_k$, where $a \in \mathbf{\Sigma}$, $k \geq 0$, and $1 \leq i \leq k$. The length of the *word* is given by $|k|$. When $k = 0$ (i.e. the word contains no symbols), the result is referred to as the *null* or *empty word*, which we will denote as λ.

We now cover two operators that are essential to our definition of a language: *Kleene Closure* (or *star*) and *Kleene plus*.

Definition 2.1.3 *Let* $\boldsymbol{\Sigma^*} = \bigcup_{i=0} \boldsymbol{\Sigma^i}$ *denote the Kleene closure (or star) over alphabet* $\boldsymbol{\Sigma}$.

Definition 2.1.4 *Let* $\boldsymbol{\Sigma^+} = \bigcup_{i=1} \boldsymbol{\Sigma^i}$ *denote the Kleene plus over alphabet* $\boldsymbol{\Sigma}$.

While $\boldsymbol{\Sigma}$ is finite by definition, note that $\boldsymbol{\Sigma^*}$ and $\boldsymbol{\Sigma^+}$ are always infinite, since there is no limit on the length of the words in these sets. Often, we are not concerned with all of the words that can possibly be generated from an alphabet; rather, we want to consider only a subset of those words. For instance, ... This leads to the general definition of *language*:

Definition 2.1.5 *Any subset of* $\boldsymbol{\Sigma^*}$ *is called a language over* $\boldsymbol{\Sigma}$.

Example

Let $\boldsymbol{\Sigma} = \{0, 1\}$. Then

$\boldsymbol{\Sigma^*} = \{\lambda, 0, 1, 01, 10, \dots\}$.

$\boldsymbol{\Sigma^+} = \{0, 1, 01, 10, \dots\}$.

The set $\{00, 11, 01, 10\}$ is a language over $\boldsymbol{\Sigma}$. Since it has a finite number of sentences then it is a finite language. Whereas, the set $\mathbf{L} = \{0^n 1^m : m, n \geq 0\}$ over $\boldsymbol{\Sigma}$ is an infinite language.

Since we define a language to be a set, not only are the foregoing set operations valid for languages, but also the standard operations: *union*, *intersection*, *difference*, and *complement*. Furthermore, definitions will be reviewed, which are often seen in the literature on formal language: *concatenation*, *reversal*, and the *power of a language*.

Definition 2.1.6 *Let* $\boldsymbol{L_0}$, $\boldsymbol{L_1}$ *be languages over the alphabet* $\boldsymbol{\Sigma}$. *Let* $\boldsymbol{L_2} = \boldsymbol{L_1} \cup \boldsymbol{L_0}$, *s.t.* $\cup$ *is defined to be the union operator. Then* $\boldsymbol{L_2}$ *contains each* $x \in \boldsymbol{L_1}$ *or* $x \in \boldsymbol{L_0}$.

Definition 2.1.7 *Let $\boldsymbol{L}_2 = \boldsymbol{L}_1 \cap \boldsymbol{L}_0$, s.t. $\cap$ is defined as the intersection operator. Then $\boldsymbol{L}_2$ contains each $x \in \boldsymbol{L}_1$ and $x \in \boldsymbol{L}_0$.*

Definition 2.1.8 *Let $\boldsymbol{L}_2 = \boldsymbol{L}_1 \setminus \boldsymbol{L}_0$, s.t. $\setminus$ is defined as the difference operator. Then $\boldsymbol{L}_2$ contains each $x \in \boldsymbol{L}_1$ and $x \notin \boldsymbol{L}_0$.*

Example

Let $\mathbf{L}_0 = \{\text{a, b, c}\}$ and $\mathbf{L}_1 = \{\text{c, d, e}\}$. Then

$\mathbf{L}_0 \cup \mathbf{L}_1 = \{\text{a, b, c, c, d, e}\}$.

$\mathbf{L}_0 \cap \mathbf{L}_1 = \{\text{c}\}$.

$\mathbf{L}_0 \setminus \mathbf{L}_1 = \{\text{a, b}\}$.

Definition 2.1.9 *The complement of the language $\boldsymbol{L}$ is defined as $\bar{\boldsymbol{L}} = \boldsymbol{\Sigma}^* \setminus \boldsymbol{L}$.*

Example

Let $\mathbf{L} = \{\text{all words starting with 0}\}$ over $\boldsymbol{\Sigma} = \{0, 1\}$. Then
$\bar{\boldsymbol{L}} = \{\text{all words starting with 1}\} = \{1x : x \in \boldsymbol{\Sigma}^*\}$.

Definition 2.1.10 *The concatenation of two languages $\boldsymbol{L}_0$ and $\boldsymbol{L}_1$ is the set of all words obtained by concatenating each element of $\boldsymbol{L}_0$ with $\boldsymbol{L}_1$.*

Example

Let $\boldsymbol{L_0} = \{\text{a, b}\}$ and $\boldsymbol{L_1} = \{\text{c, d}\}$. Then

$\boldsymbol{L_0 L_1} = \{\text{ac, ad, bc, bd}\}$.

Definition 2.1.11 *The reversal, $\boldsymbol{L}^R$ is recursively defined as:*

1. *if $x = \{\lambda\}$, then $\boldsymbol{L}^R = x$.*

2. *if* $x = yz$ *for some* $y \in \mathbf{\Sigma^*}$, $z \in \mathbf{\Sigma}$. *Then* $x^R = zy^R$ *(the word* zy^R *is the concatenation of the string* z *with the reversal of the string* y*).*

Definition 2.1.12 *Let the* n^{th} *power of a language* $\boldsymbol{L^n}$ *be defined as the concatenation of* $\boldsymbol{L}$ *with itself,* n *times.*

2.2 GRAMMAR

As demonstrated in the previous section, set-theoretic methods provide a level of rigor for specifying some languages. However, languages exist that cannot be adequately represented using this formalism. Consider, for instance, the alphabet $\Sigma = \{0, 1\}$ and suppose one wants to specify the language: "all words beginning with two *0's*, and ending with two *1's*". This language could be represented with set-predicate notation, like $\boldsymbol{L} = \{x : x \in \Sigma^* \text{ and } x \text{ begins with "00" and ends with "11" }\}$; or, perhaps, one might prefer $\boldsymbol{L} = \{xyz : x \in \{00\}, y \in \Sigma^*, \text{ and } z \in \{11\}\}$. Notice, that there exists a multitude of ways—some, less formal than others—in which this language could be specified using the set-theoretic method; thus, the foregoing approach does not provide enough constraint, and in many cases can introduce ambiguity in our specifications. Therefore, scholars have devised alternative approaches that provide a more rigorous treatment to language representation, one of the more successful is *grammar*.

Grammar may be thought of as a mechanism that operates on symbols, of which we distinguish two kinds: *terminals*, which will be denoted using lowercase letters, and *non-terminals* which will be represented by uppercase letters. Terminals are the actual elements of the language—that is, the symbols making up the *words* of a given language—whereas, *non-terminals* are auxiliary objects, used to facilitate the specification of the language. More formally, a grammar is a finite set of rules defined as a 4-tuple $\mathbf{G} = \{\mathbf{N}, \mathbf{T}, \mathbf{P}, S\}$, where $\mathbf{N}$ is a finite set of non-terminal symbols, $\mathbf{T}$ is a finite set of terminal symbols, $\mathbf{P}$ is a finite set of *production rules*, and $S \in \mathbf{N}$ is a distinguished production rule, known as the *start symbol*.

Definition 2.2.1 *Let the gammer* $\boldsymbol{G}$ *be a 4-tuple,* $\boldsymbol{G} = \{\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, S\}$*, s.t.*

1. $\boldsymbol{N}$ *is a finite set of non-terminal symbols.*

2. $\boldsymbol{T}$ *is a finite set of terminal symbols.*

3. $\boldsymbol{P}$ *is a finite set of productions rules, each of the form:* $A \to \alpha$, *s.t.* $A \in (\boldsymbol{N} \cup \boldsymbol{T})^+$ *and* $\alpha \in (\boldsymbol{N} \cup \boldsymbol{T})^*$.

4. S *is a special non-terminal, called the start symbol, and* $S \in \boldsymbol{N}$.

The production rules are used to specify the structure of a given language and are applied, intuitively, beginning with the start symbol S; thereafter, the rules may be applied in arbitrary order. To demonstrate, consider the word $w_0 = xz$, and a production rule $x \to y$, which is interpreted as "x may be replaced by y". After the application of the rule, a new word is formed, $w_1 = yz$, and we say that w_0 *derives* w_1. In order to state the derivation formally, we use the symbol $\Rightarrow$, for example, $w_0 \Rightarrow w_1$. Moreover, to concisely represent multiple derivations, for instance, given $S \Rightarrow w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n$, we can write $S \Rightarrow^* w_n$. The derivation of all the words consisting of only terminal symbols is the language *generated* by the grammar, $\boldsymbol{L}(\boldsymbol{G})$.

Definition 2.2.2

Let $\boldsymbol{G} = \{\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, S\}$ *be a grammar. Then*

$\boldsymbol{L(G)} = \{w \in \boldsymbol{T}^* : S \Rightarrow^* w\}$ *is the language generated by* $\boldsymbol{G}$.

Note that there may exist many different grammars that can generate the same language.

Example

Let $\boldsymbol{G}$ be a grammar with start symbol E and production rules $\boldsymbol{P}$:

$E \to EOE|N$

$O \to +|-|*|\div$

$N \to 0|1|2|3|4|5|6|7|8|9$

Note the symbol '|' is the exclusive disjunctive operator, for instance, the first production rule can be read "E can be replaced by EOE or N".

Given $\boldsymbol{G}$, we may derive a word starting at E, such as $E \Rightarrow EOE \Rightarrow NOE \Rightarrow 2OE \Rightarrow 2 + E \Rightarrow 2 + N \Rightarrow 2 + 2$. Notice that the order with which the word $2 + 2$ was derived always replaced the leftmost non-terminal symbol. Thus, we've demonstrated the *leftmost derivation* of the word $2 + 2$. By contrast, we could have derived the word using a different technique: $E \Rightarrow EOE \Rightarrow EON \Rightarrow EO2 \Rightarrow E + 2 \Rightarrow N + 2 \Rightarrow 2 + 2$, which is the *rightmost derivation* of the word $2 + 2$.

The foregoing example demonstrated a leftmost and rightmost derivation. In this case, the order of replacement was of significance because the first rule—$E \rightarrow EOE|N$—contained more than one non-terminal on the right-hand side. Some grammars, however, have restrictions that do not allow for this, namely *linear grammars*.

Linear grammars are those whose production rules $\boldsymbol{P}$ are of the form $A \rightarrow x\beta y$, where $x, y \in \boldsymbol{T}^*$ and $\beta \in \boldsymbol{N}$. Therefore, linear grammars have at most one non-terminal that can occur on the right-hand side of any production rule, without restriction on the position of this non-terminal in relation to the terminal symbols (Linz, 2006). However, there do exist special cases of linear grammars that maintain a constraint on the position of the non-terminal symbol. *Right-linear* grammars are linear grammars, whose non-terminal symbols—if present—appear in the rightmost position on the right-hand side of each production rule. Similarly, each production rule in a *left-linear* can specify only one non-terminal symbol, such that it appears in the leftmost position on the right-hand side.

Definition 2.2.3

A gammer $\boldsymbol{G} = \{\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, S\}$ *is said to be right-linear if all productions* $\boldsymbol{P}$ *are of the form:* $A \rightarrow x\beta$ *or* $A \rightarrow x$.

where $x \in \boldsymbol{T}^*$ *and* $\beta \in \boldsymbol{N}$*. A grammar is said to be left-linear if all productions are of the form:* $A \to \beta x$ *or* $A \to x$*.*

2.2 Chomsky's hierarchy

Given a word w and a grammar $\boldsymbol{G}$, the question of whether $\boldsymbol{G}$ can generate w is called *the membership problem* (i.e. is $w \in L(\boldsymbol{G})$?). It is *decidable* if there exists a formalism (e.g. a Turing machine), which can provide a sufficient answer—i.e. "yes" or "no"—for the question in finite time. In his studies on formal language theory, Noam Chomsky introduced a classification system of grammar, whose assignment to a particular *type* is given by the formalism required to cede the question *decidable*. Furthermore, he recognized that each type of grammar was related in such fashion, as to form a containment hierarchy. Specifically, this relationship can be realized by viewing the hierarchy top-down: the top-level grammar has no restrictions placed on its productions, however, each successive level down, accumulates restrictions upon them. This gradual increase in constraints, yields a nested hierarchical structure.

The *Chomsky Hierarchy* assigns a numerical label to each type of grammar, *type 0* through *type 3*. Moreover, the language family that is described by each type is given the respective name: *recursively enumerable*, *context-sensitive*, *context-free*, or *regular*. Table 2.1 provides a brief description for each type of grammar. This treatise will not recapitulate the details of each and every type. However, the next section will provide a discussion of the *context-free languages*, which have special significance to the representation of programming languages.

2.2 Context-free grammar

Imagine that you've just received a message from a friend that reads: "How are *are* you?". Obviously, the preceding sentence isn't valid English, so you assume

Type	Name	Productions	Parsing Complexity
0	Recursively enumerable	Unrestricted	Undecidable
1	Context-sensitive	$x \rightarrow \alpha$ $x, \alpha \in (\boldsymbol{N} \cup \boldsymbol{T})^*$	NP Complete
2	Context-free	$A \rightarrow \alpha$ $A \in \boldsymbol{N}$, $\alpha \in (\boldsymbol{N} \cup \boldsymbol{T})^*$	$O(n^3)$
3	Regular	$A \rightarrow x\beta$ $x \in \boldsymbol{T}^*$ and $\beta \in \boldsymbol{N}$	$O(n)$

Table 2.1: The Chomsky Hierarchy

that the message contains a typo. In this sense, you are acting as a *recognizer* for the English language, and have solved *the membership problem* from section 2.3.1. Building formal recognizers for natural languages, such as English, is challenging and has been an active research area for several decades. Whereas, high-level programming languages, specified by context-free grammars possess characteristics that make them more amenable to algorithmic interpretation.

Definition 2.2.4

Let $\boldsymbol{G} = \{\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, S\}$ *be a grammar, as defined in Definition 2.3.1. Then* $\boldsymbol{G}$ *is context-free if all productions,* $p \in \boldsymbol{P}$*, of are of the form:* $A \rightarrow \alpha$ *where* $A \in \boldsymbol{N}$ *and* $\alpha \in (\boldsymbol{N} \cup \boldsymbol{T})^*$*.*

Consider the word aAb, an intermediate representation of the word aab, given the condition that the generative grammar has the production rule: $A \rightarrow a$. The *context* of the non-terminal symbol in the word aAb, is intuitively the surrounding terminal symbols: a and b. The rule $A \rightarrow a$ says "replace non-terminal A, with terminal a"—regardless of the "neighborhood" of symbols within which one may find A—thus, motivating the name "context-free grammar".

2.2 Backus-Naur form

Backus-Naur Form or BNF is a formalism that has been used to successfully model programming language syntax. In fact, grammars specified using BNF are essentially the same as context-free grammars, though their appearance is different. To demonstrate the denotational differences, consider the grammar in the example from Section 2.3:

$E \rightarrow EOE|N$

$O \rightarrow +|-|*|\div$

$N \rightarrow 0|1|2|3|4|5|6|7|8|9$

Using the BNF notation to rewrite the above grammar, we replace all $\rightarrow$ symbols with ::=. The nonterminal symbols are designated using triangular brackets and terminals are symbols without such brackets. Moreover, the *lettercase* of a symbol has no bearing on the interpretation. Typically, the symbols are denoted with the intent to elucidate the grammar's purpose, which may involve using longer names for symbols. Thus, the grammar may appear as:

$< Expr >::=< Expr >< Op >< Expr > | < Num >$

$< Op >::= +|-|*|\div$

$< Num >::= 0|1|2|3|4|5|6|7|8|9$

2.3 SYNTACTIC ANALYSIS

It has been mentioned that the question of whether a word can be recognized as a member of a language is of importance. This section builds on the foregoing by asking "exactly *how* is $w \in \boldsymbol{L}(\boldsymbol{G})$?". We'll discuss how an algorithm can provide a sequence of productions by which w can be derived, a process formally known as *syntactic analysis*, or parsing.

Algorithms for uncovering the rules by which a particular word can be derived from a grammar, have been classified into types: "bottom-up" and "top-down." The former methodology attempts to recover the sequence of production rules by working in reverse—essentially, using the production rules from right-to-left. While the bottom-up methods can recognize more languages than the top-down algorithms, it is generally easier to work with the class of grammars that can be recognized by top-down algorithms.

A top-down parsing algorithm assumes that the input word matches the start symbol, and given this assertion, works its way toward uncovering the structure. To demonstrate, consider the grammar given in Section 2.2.3, and imagine that the word $2 + 2$ is input. The first assumption, the word $2 + 2$ matches the symbol $< Expr >$, implies that either the string matches to $< Expr >< Op >< Expr >$, or $< Num >$. Whichever of the two foregoing choices is followed, leads to further implications until the base assertions are found to be true. Notice that this strategy attempts to uncover a leftmost derivation of the input, and it leads to non-terminating errors for left-recursive grammars.

CHAPTER 3
DIAGRAMMATIC REPRESENTATION OF LANGUAGE

Representational objects that people refer to as *diagrams*, may be labeled more granularly, for instance, as "maps", "charts", and so on. The current chapter investigates a category of diagram, called *graphs*, which are pertinent to reasoning about the structures discussed in **Chapter 2**, in addition to a novel formalism introduced in **Chapter 4**.

While anyone who has taken a high school algebra course should be familiar with the "graph of a function", the graphs discussed in this chapter, bear no relation to the preceding and are, instead, the study of a field known as *graph theory*; wherein, graphs are used to describe more natural situations than those of *functions*, or other entities of an inherently mathematical nature. It is, perhaps, due to this applied history that texts on graph theory do not necessarily follow a canonical language and, as such, the current treatise will rely upon a single resource from which to derive the definitions of the objects of interest, Ross and Wright (2003).

Graph theory finds its roots in a famous puzzle, the so-called "Seven Bridges of Königsberg." The city of Königsberg in Prussia (modern-day Kaliningrad, Russia) was partitioned by the Pregel River (see the map in Figure 3.1), such that there existed no route to the resulting islands, but for seven bridges. Thus, the challenge was to find a single walk through the city that would involve crossing each bridge, once and only once, before returning back to the starting position.

In 1735, Euler presented a theorem satisfying the puzzle to the Petersburg Academy; he proved that there existed no such path with the given constraints. His analysis considered the only relevant features to be the land masses and bridges connecting them, and the relationships between those entities. This allowed for a more abstract representation of the problem, and eventually led to a more rigorous language with which to describe *it*, and similar problems (Alexanderson, 2006).

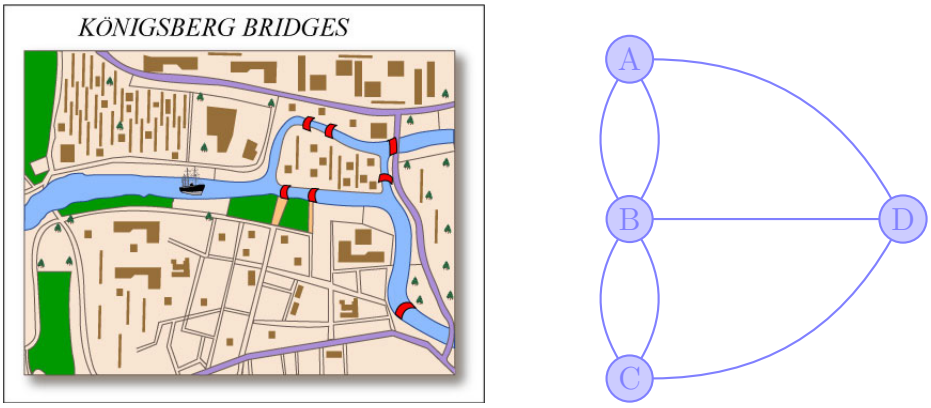

Figure 3.1: The diagram on the left is a map of Königsberg (MIT, 2007). On the right is a graph representation of the Seven Bridges of Königsberg puzzle.

3.1 GRAPHS

Every graph is given by a vertex set $V(G)$ and an edge set $E(G)$. The diagram appearing on the right side of Figure 3.1, demonstrates a graph representation of the "Seven Bridges of Königsberg" puzzle. In the puzzle, each land mass is referred to as a *vertex*, which is represented by a point in the plane, and each bridge connecting them an *edge*, represented by a line segment connecting vertices. As it turns out, the graph of Figure 3.1 is a distinguished kind of graph, known as a *connected graph*, which means that there is a path from any vertex to any other vertex.

Definition 3.1.1 *A graph G is an ordered pair $G = (V, E)$; where $V(G)$ is the set of all vertices and $E(G)$, the set of all edges in G.*

Definition 3.1.2 *A vertex $v_i \in V(G)$ is the fundamental unit of a graph.*

Example The graph in Figure 3.1 has the following vertex set $V = \{A, B, C, D\}$.

Definition 3.1.3 *An edge $e_i \in E(G)$ is an unordered pair of vertices $e_i = \{v_n, v_m\}$, and is a fundamental unit of a graph that represents a relation between vertices.*

> **Example** The graph in Figure 3.1 has the following edge set $E = \{\{A,B\}$, $\{A,B\}$, $\{A,D\}$, $\{B,C\}$, $\{B,C\}$, $\{B,D\}$, $\{C,D\}\}$.

Definition 3.1.4 *The number of edges at a vertex is called the degree of the vertex.*

> **Example** The degree vertex A, B, C, and D in the graph of Figure 3.1 is: 3, 5, 3, and 3, respectively.

A route around a graph in which every edge is visited, at most once, is called an *Euler path*. Similarly, an *Euler circuit* is an Euler path, such that the beginning and ending vertex is the same point. Given the foregoing definitions, the question then becomes: Given graph G, is there a path P that passes through every vertex

$v \in V(G)$, and uses every edge $e \in E(G)$ exactly once? More succinctly, is P an Euler circuit?

Formally, a path in a graph is a sequence of edges $e_1, e_2, ..., e_n$ together with a sequence of vertices $v_1, v_2, ..., v_{n+1}$, where $e_i = \{v_i, v_{i+1}\}$ for $i = 1, ..., n$. Euler observed that in order to visit a vertex by a novel path, and be able to leave that vertex, there must be a novel entering edge, and a novel outgoing edge. Thus, the degree of each vertex must be *even*. In the case of a terminal node, the degree of the vertex is not relevant to the existence of an Euler path; however, for an Euler circuit to exist, no vertex may be of odd degree. Figure 3.1 demonstrates the utility of graph representation, as it can easily be seen that no Euler circuit (or, even Euler path) exists.

Definition 3.1.5 *A path $P \in G$ is itself a graph $P = (V, E)$ having edges $e_1, e_2, ..., e_n$ together with a sequence of vertices $v_1, v_2, ..., v_{n+1}$; where $e_i = \{v_i, v_{i+1}\}$ for $i = 1, ..., n$. If $v_i = v_{i+1}$ then e_i is a loop. Moreover, if $v_{n+1} = v_1$ then P is said to be closed. Finally, if all of the edges $e_1, e_2, ..., e_n$ are distinct then P is called simple.*

Theorem 3.1.1 *(Euler's Theorem) A finite connected graph in which every vertex has even degree has an Euler circuit.*

The closed simple path with vertices A, B, C, D, A is known as a *cycle*. The cycle concept is important in graph theory and, as we'll later see, has application to the specification of programming languages. A graph is said to be *acyclic* if it does not possess a cycle. Moreover, D is said to be *reachable* from A, since there is a path from vertex A to vertex D.

Definition 3.1.6 *A closed simple path with vertex sequence $v_1, v_2, ..., v_n, v_1$ for $n \geq 1$ is called a cycle if the vertices $v_1, ..., v_n$ are distinct.*

Definition 3.1.7 *Given graph G, the reachability relation R is defined by the following: $R(u) = \{u \in V(G) : u \text{ is reachable from } v\}$; where $u, v \in V(G)$.*

3.1 Directed graphs

It doesn't take much to imagine a path that can be traversed in only one direction. After all, situations like this are a common experience of drivers the world over, as they navigate a grid pattern defined by one-way streets. Intuitively, such situations have been modeled with graphs by augmenting their edges with arrows that point in permissible directions. Figure 3.2 demonstrates an example of such a graph or, more formally, *digraph*.

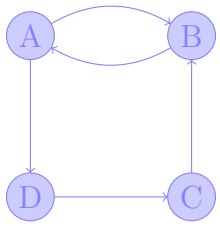

Figure 3.2: The edges of a digraph are augmented with arrows, indicating the direction of the relation between vertices.

The definitions given in the preceding section were for undirected graphs, but these hold for digraphs, excluding the definition given for edges (Definition 3.1.3). In the case of undirected graphs, each edge is defined by an unordered pair of vertices. By contrast, a digraph's edges are given by ordered pairs of vertices.

Definition 3.1.8 *An edge $e_i = (v_n, v_m)$ is an ordered pair of vertices, and is a fundamental unit of a digraph. Each edge represents a relation between vertices, and a relation has a direction property, going from v_n to v_m.*

3.1 Trees

As previously shown, researchers studying graph theory or relying upon graph structures to reason about problems in other fields, have uncovered a plethora of distinct forms of graph. True to their history, these graphs have taken inspiration from situations and entities that exist materially. A further example, the *tree*, is a type of

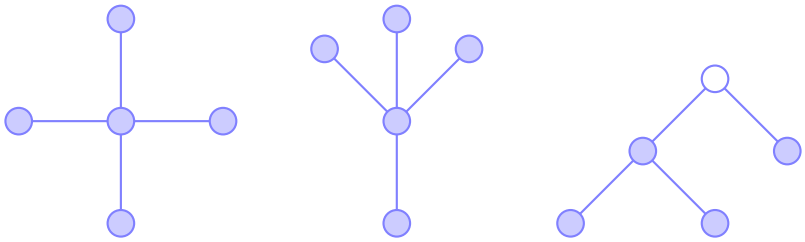

Figure 3.3: A tree is a connected acyclic graph.

graph that students of computer science are likely to be most familiar, as they are often used to model data structures. Figure 3.3 demonstrates a few examples of these graphs.

A tree is formally defined as a connected acyclic graph. These graphs have a nomenclature, adopted from the terminology used to describe both real-world trees, and the relationships found in genealogy trees. To demonstrate, consider graph S in Figure 3.3. The hollow vertex is singled out as the *root node*. This root node is the *parent* of two *child nodes*, one of which is a *leaf node* (it has no descendants), the other is itself a parent node. As with other forms of graph, trees may have direction associated with their edges; however, for rooted trees it is common to leave off the arrow heads, and assume that the edges are directed downward. Additionally, like other graphs, trees may have extra information attached to their edges and vertices in the form of labels.

There are a few points to be made regarding the trees depicted in Figure 3.3. Firstly, graphs Q and R are actually equivalent representations, or isomorphisms; however, they're laid out in the plane differently—the latter graph, R, is drawn in such a way as to motivate its category label (i.e. "tree"). Secondly, graph S is a special kind of tree called a *binary tree*, a rooted tree that can have at most two child nodes: a left and/or right child, or no child nodes; this concept generalizes to *m-ary* trees, those with at most m child nodes. Lastly, any of tree may be transformed into a rooted tree, by simply assigning the root status to one of the vertices in the tree.

3.2 DIAGRAMS FOR DESCRIBING LANGUAGE

The material presented in this section should begin to "bridge the gap" between the symbolic formalisms that were introduced in **Chapter 2**, and what may seem, thus far, to be unrelated structures presented in the current chapter. While there is surely a vast array of diagrams that could find application to specifying language, this section will focus on a few of the more established forms: *finite automata* and *parse trees* which use graph structures that show how to recognize elements of a formal language, *dependency graphs* that are used for many tasks, including identifying useless production rules of formal grammars, and *syntax trees*, which are generally used in compiler technology.

3.2 Finite automata

The ability to rigorously specify language through formal grammar has already been discussed to a sufficient degree; the concept of a language accepter, however, has only briefly been mentioned (cf. Section 2.2.2). Indeed, this circumstance was by design, as formal language theory has been distinguished from that of *automata theory*, the study of language recognizers. Moreover, although formal grammar and automata (the objects of interest in automata theory) can be shown to be equivalent systems of representation, it seems fitting that a discussion of *finite automata* be covered in a chapter about diagrammatic representation, a point that will become clear in the remaining text.

A finite automaton is an idealized device that can be in *one* of a finite set of states. Given a distinguished starting position, the machine will read an input symbol from an imagined memory cell and, depending upon the character and the device's configuration, may *transition* to a new state. The pattern of reading a character from memory and transitioning to a different state, will continue until the machine

finds itself in a peculiar state, known as a *final state*; whereupon, the machine ceases operation. The resulting sequence of characters, permitting the machine to transition to a final state, is said to be a "string accepted by the machine." For expository purposes, this treatise will cover one form of automaton, the *deterministic finite automaton*, or DFA.

Definition 3.2.1 *A deterministic finite accepter or DFA is defined by the 5-tuple:* $M = (Q, \Sigma, \delta, q_0, F)$, *s.t.*

1. Q *is a finite set of internal states.*
2. Σ *a finite set of symbols called the input alphabet.*
3. $\delta : Q \times \Sigma \to Q$ *is a total function called the transition function.*
4. $q_0 \in Q$ *is the initial state.*
5. $F \subseteq Q$ *is a set of final states.*

Example. Let $M = (\{q_0, q_1, 0_2.., q_5\}, \{a, b, c\}, \delta, q_0, \{q_5\})$; where, the transition function δ is defined as:

$\delta(q_0, a) = q_1, \qquad \delta(q_0, c) = q_2,$

$\delta(q_1, b) = q_3, \qquad \delta(q_2, b) = q_4,$

$\delta(q_3, c) = q_5, \qquad \delta(q_4, a) = q_5$

Hence, this DFA accepts a simple language of two strings: "abc" and its reverse "cba".

Although, finite automata (like most other representations) can be specified by a system of symbols, an automaton described by a graph can be easier to work with in practice. Traditionally, digraphs, augmented with edge and vertex labels have been used to more concretely represent these highly abstract machines.

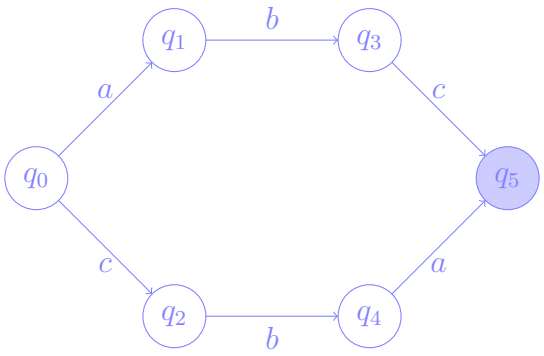

Figure 3.4: *A deterministic finite automaton.*

More formally, given a DFA, M, then the graph representation of M, G_M, will have a total of $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled a. The vertex associated with q_0 is called the *initial vertex*, while those labeled with $q_f \in F$ are the *final vertices* (Linz, 2006). Figure 3.4 shows the graph of an automaton, which is formally called a *transition graph*. In the figure, the final state is distinguished by a filled circle.

Theorem 3.2.1 *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every $q_i, q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is a walk in G_M with label w from q_i to q_j.*

As shown, the transition graph provides an intuitive representation for uncovering properties of automata; however, symbolic systems might better serve some situations. Nevertheless, by Theorem 3.2.1, these representational forms are isomorphic and can be used interchangeably.

3.2 Parse trees

In section 2.3, we reviewed algorithms that can take as input a word w, and show "exactly *how* is $w \in \boldsymbol{L}(G)$." The current section will review a related concept, a form of diagram known as a *parse tree* (or derivation tree). In words, a parse tree offers

a way to visualize the steps followed in order to derive a given sentential form; thus, proving that the sentential form is a member of a particular language.

For example, given the following context-free grammar, a derivation tree for the word, $w = abab$, is demonstrated in Figure 3.6.

$G = (\{S, B\}, \{a, b, \lambda\}, S, \boldsymbol{P})$; where $\boldsymbol{P}$ is:

$S \rightarrow aA$

$A \rightarrow bBb$

$B \rightarrow a|\lambda$

Hopcroft and Ullman (1979), provide a formal definition for systematically constructing a parse tree. That is, given a grammar $G = (\boldsymbol{N}, \boldsymbol{T}, S, \boldsymbol{P})$, a parse tree is a tree where:

1. Every vertex has a label, which is a symbol of $\boldsymbol{N} \cup \boldsymbol{T} \cup \{\lambda\}$.
2. The label of the root is S, the start symbol.
3. If a vertex is interior and has a label A, then A must be in $\boldsymbol{N}$.
4. if n has label A and $n_1, n_2, ..., n_k$ are children of vertex n, in order from the left, with labels $X_1, X_2, ..., X_k$, respectively, then $A \rightarrow X_1X_2...X_k$ must be a production rule in $\boldsymbol{P}$.
5. If vertex n has label λ, then n is a leaf and it is the only child of its parent node.

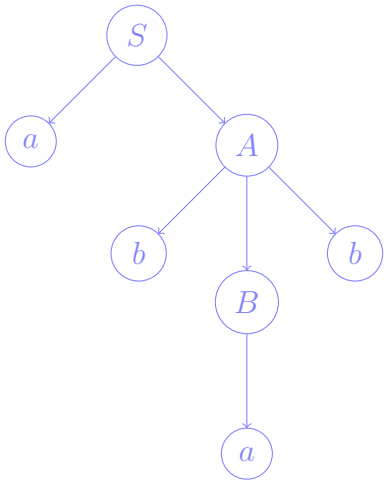

Figure 3.5: *A parse tree.*

3.2 Dependency graphs

The current section will review a kind of graph, called a *dependency graph*, a tool used for visualizing complex relationships. These graphs are used in many different contexts across a number of fields; here, graphs that are pertinent to context-free grammars will be discussed.

Consider the variation of the example CFG, given in Section 2.2.2. In this version, the first production rule was changed by removing the alternate derivation rule of non-terminal E.

$\boldsymbol{G} = (\{E, O, N\}, \{+, -, *, \div, 0, 1, ..., 9\}, E, \boldsymbol{P})$; where $\boldsymbol{P}$ is:

$E \rightarrow EOE$

$O \rightarrow +| - | * |\div$

$N \rightarrow 0|1|2|3|4|5|6|7|8|9$

Understanding the language generated by a grammar, like the foregoing, can often be mitigated by certain simplifications on the rules of the grammar. For instance, some grammars may contain, so-called "useless productions", production rules that do not

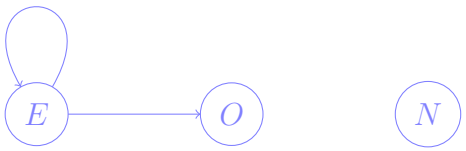

Figure 3.6: *A dependency graph.*

result in at least one derivation. In order to realize where these useless productions are specified in the grammar, a dependency graph can be constructed from the variables.

A dependency graph is constructed from a grammar by creating a vertex for each variable, and connecting vertices with an edge where there is a relationship between variables via a production rule. To demonstrate, Figure 3.5 shows a dependency graph for the above CFG. In the figure, you can see that non-terminal N is not reachable from any other vertex, so can be removed from $\boldsymbol{G}$.

While it might seem a trivial task to identify the useless production in the example grammar used here, grammars that describe program language syntax are much larger in scope and, therefore, the use of a dependency graph can indeed be helpful for such cases.

3.2 Syntax diagrams

A further concept related specifically to context-free grammar, and the final graph structure that will be reviewed, is known as a *syntax diagram*. These diagrams are used to visualize the production rules of CFGs, historically, specified in BNF form. They were introduced in texts on programming languages; for example, *The programming language Pascal* (Wirth, 1973).

To demonstrate these graphs, consider the BNF grammar:

$< Expr >::=< Expr >< Op >< Expr > | < Num >$

$< Op >::= +| - | * |\div$

$< Num >::= 0|1|2|3|4|5|6|7|8|9$

The way in which to construct a syntax diagram is to translate each BNF rule into a graph representation, such that given $|\boldsymbol{V}|$ production rules, the plane will possess that same number of syntax diagrams. Each sub-diagram will contain a set of rectangular boxes to represent the terminal and non-terminal symbols (e.g. $< Expr >$, '+', and so forth), and a set of directed edges, where each edge between symbols, denotes that one symbol may be replaced by the other. Moreover, an alternative derivation for a given non-terminal is represented by an orthogonal edge. The following figure (Figure 3.7) illustrates the complete set of syntax diagrams for the above grammar.

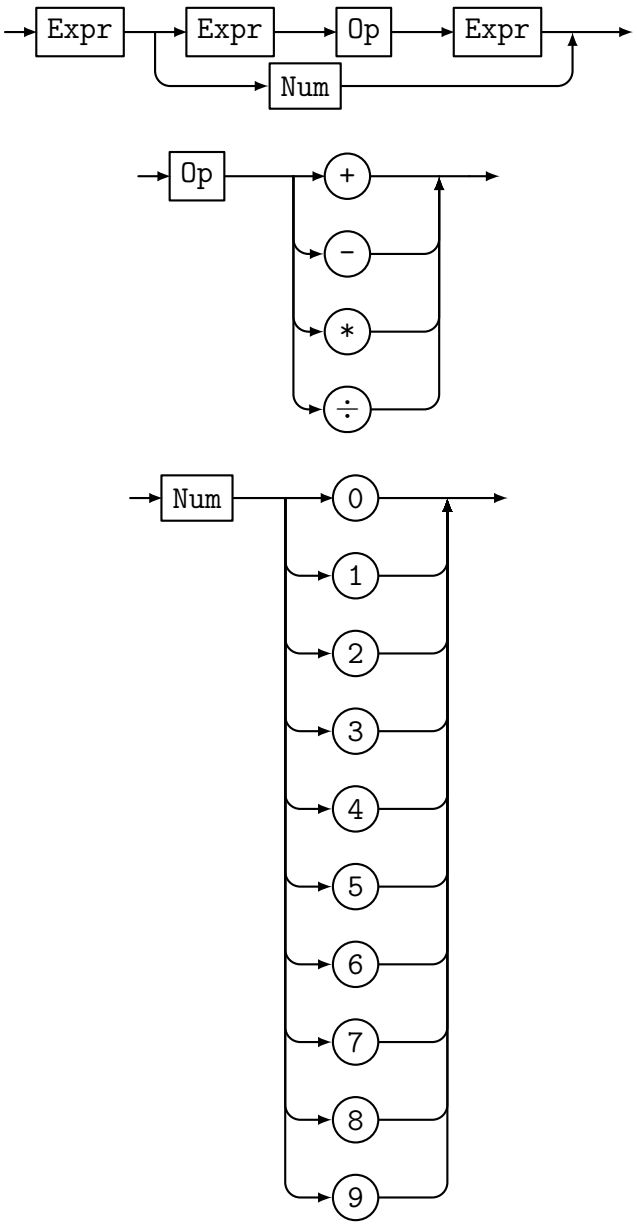

Figure 3.7: *Syntax diagrams.*

CHAPTER 4
AUGMENTED SYNTAX GRAPHS

The idea that representational format plays a non-trivial role in scientific reasoning was discussed in the introduction of this treatise. Following from this idea, the current section will analyze the representational forms that were covered in **Chapter 2** and **Chapter 3** and will, further, consider how the constraints of these formalisms, constrain reasoning about the *represented world*—that which these representations intend to stand for. Moreover, a new diagrammatic representation will be introduced, one that offers a different way of viewing the information of the target domain, the syntactical structure of language.

4.1 BACKGROUND

There exist situations where information can act as noise; for example, **Chapter 3** reviewed the concept of a *dependency graph* used for uncovering so-called "useless productions" in formal grammars. Rather than translating all of the information about the grammar, this particular formalism includes only information that would be pertinent to finding non-terminal symbols that are not reachable, avoiding extraneous information.

While, in some cases, it may be beneficial to focus on a subset of the information of some target, there are scenarios where maintaining all of the information is useful. This can be argued, *prima facie*, by noting the established use of such formalisms. For instance, a finite state automaton is often represented diagrammatically, and has an equivalent symbolic representation. In other words, the difference between these systems is not informational, but structural. And while both representations maintain the same information about the same target, one format has not replaced the other. Instead, a reasoner might often use these systems in conjunction, switching between the two representations as needed. Hence, it not only matters what knowledge about a *represented world* is captured, but how the information is structured. Indeed, although representations can be informationally equivalent, they can be computationally different (Larkin and Simon, 1995), potentially leading to different insights.

4.2 A NOVEL DIAGRAM OF THE BACKUS-NAUR FORM

While each of the diagrammatic representations that were reviewed in Section 3.2, can serve multiple purposes, the fundamental use of each one may be thought of as follows: Parse trees are used to visualize the way in which a particular *word* is a member of some language, while finite state automata (represented diagrammatically) can be used to visualize how all of the words are given by some formal grammar; dependency graphs are billed as a mechanism for uncovering reachable production rules, and syntax diagrams are formalisms used to visualize the individual production rules of a BNF (or EBNF) grammar (Linz, 2006).

Here we introduce the *augmented syntax graph*, a diagrammatic representation which has semblances to the aforementioned formalisms. In particular, these structures most closely resemble the concept of a syntax diagram. However, while syntax diagrams provide a way of visualizing the production rules of a grammar, one can only investigate the rules individually. That is, one cannot externally envisage the grammar as a *whole*, since, given that a language is specified by n production rules, there will be n syntax diagrams used to represent them. On the other hand, the augmented syntax graph provides a way of capturing all production rules of the grammar in a single diagram.

An augmented syntax graph, or *ASG* is a directed graph, where each production rule is represented within a single instance of the graph. The nodes of the graph are labeled with the symbols of each production rule, and edges are used to demonstrate which symbols derive to other symbols. In order to maintain the arrangement of the symbols, and "depth" of the alternative rules, a numbering convention is used to label each of the edges.

Formally, given a BNF grammar an ASG has a labeled vertex for each symbol $v \in (N \cup T)$, where N and T and the sets of non-terminal and terminal symbols,

respectively. For each production rule of the form $C \rightarrow x$, where $x \in (N \cup T)$, an edge appears between C and every element in x. Moreover, each edge $e_{(i,j)} \in E(G)$ is labeled by the ordered pair (i, j); such that $i \in \mathbb{N}$ is the order of each alternative derivation of the rule, and $j \in \mathbb{N}$ is the positional index of the symbol in the given production rule. Figure 4.1 gives a concrete example of a syntax graph, representing the following grammar, which is a right-associative grammar for arithmetic expressions:

$$< E >::=< T > + < E > \mid < T > - < E > \mid < T >$$

$$< T >::=< F > * < T > \mid < F > \div < T > \mid < F >$$

$$< F >::= (< E >)$$

$$< F >::= number$$

Looking at the graph, one noteworthy point is that the edge labels distinguish the beginning and ending of alternative derivations for each non-terminal symbol. This property is of note because the BNF rules are maintained, and the grammar can be completely recovered after being translated to the graph formalism. To demonstrate, consider the first rule that is listed in the grammar: $< E >::=< T > + < E >$ which consists of three edges: $e_{1,1} = \{E, T\}$, $e_{1,2} = \{E, +\}$, $e_{1,3} = \{E, E\}$ in the graph. Compare the preceding to the next alternative derivation listed for $< E >$, $< E >::=< T > - < E >$. The graph demonstrates the rule with edges: $e_{2,1} = \{E, T\}$, $e_{2,2} = \{E, -\}$, and $e_{2,3} = \{E, E\}$.

A consequence of using the numbering system, rather than a qualitative notation (e.g. coloring scheme) is that the knowledge of ordering of alternative derivations in the BNF is maintained in the ASG. This property is generally irrelevant in theory, however, for some applications of the diagram the information can be of use. In the next chapter some of these applications will be discussed.

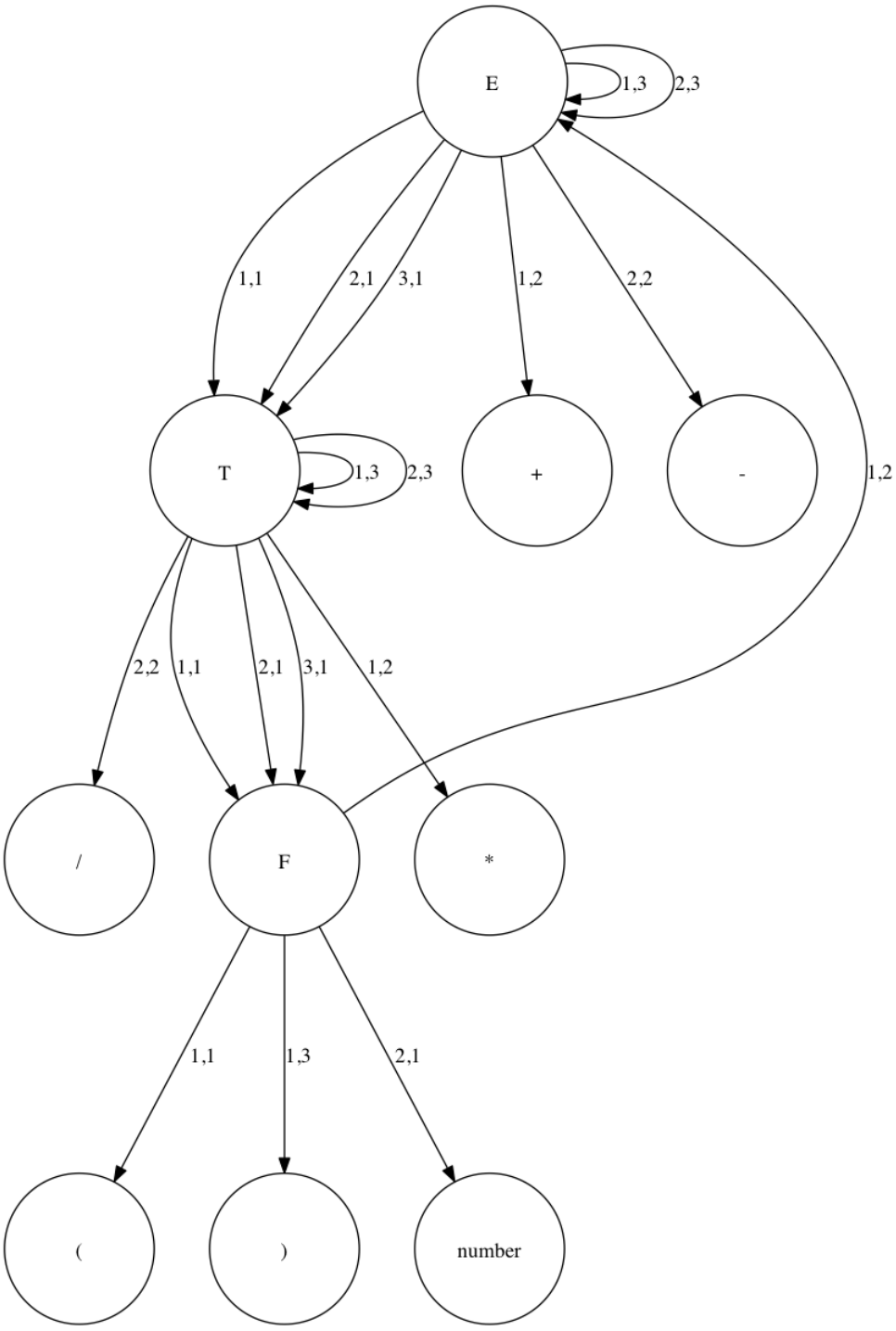

Figure 4.1: *An augmented syntax graph.*

CHAPTER 5

AN APPLICATION FOR VISUALIZING SYNTACTIC ANALYSIS

The previous chapter introduced a new type of diagram, an isomorph of a grammar denoted in BNF. The current chapter will introduce the implementation of the foregoing diagram in a pedagogical application, known as the *CFGAnalyzer*. The intent of the application is to teach the properties of grammars that are used to specify programming languages. The tool was developed using what is known as a compiler generator, *Coco/R*; therefore, the essentials of this toolchain will be reviewed. Additionally, the software package *GraphViz* will be introducted.

5.1 AN INTRODUCTION TO THE COMPILER GENERATOR *COCO/R*

A compiler generator is a program that takes as input a formal description of a programming language—historically, a BNF grammar. More specifically, Coco/R takes as input a BNF or EBNF grammar (an extended version of the BNF notation), and produces a *parser* and a *scanner*—a DFA that is used to recognize the valid words of a language.

More specifically, the parser that is created by Coco/R is known as a *recursive descent* parser. This class of parser uses a top-down methodology to parse words of the language, using a set of mutually recursive procedures. Typically, each procedure implements one of the production rules of the input grammar. The general class of languages that can be parsed using Coco/R can be described by LL(1) grammars. However, Coco/R allows one to implement ad hoc code snippets—it calls *conflict resolvers*—which can be used to accept languages described by LL(k) production rules.

To demonstrate, consider an example from Mos (2006). Given the *Sample* grammar, Coco/R will create a scanner that associates an identifier with each of the valid words of the grammar. For this particular grammar, the words are "red", "apple", and "orange" and their respective identifiers are '1', '2', and '3'. Listing 5.1 shows an example of the parser for the Sample language. Here, *la* represents the "look-ahead" token, which is a value given by the Scanner as it incrementally reads each token of an input program, from left to right.

Concretely, the conditional $la.kind = 1$ tests that the look-ahead token is equivalent to the string value "red." Given a program, *ParseSample* will check that the first word is either *orange* or *red*. If the token is found to be *orange*, then the character is retrieved and the look-ahead is advanced to the next token in the program. If the first token is *red*, the look-ahead token is advanced, and the parser will then

expect the following token to be *apple*; if that is not the case, then a syntax error results. Similarly, a syntax error results when neither *orange* nor *red* is the first token appearing in the input program.

```
COMPILER Sample
PRODUCTIONS
Sample = "red" "apple" | "orange".
END Sample.

function ParseSample
    if la.kind = 3 then
        la = Scanner.scan();
    else if la.kind = 1 then
        la = Scanner.scan();
        if la.kind = 2 then
            la = Scanner.scan();
        else
            SyntaxError(); // Expected kind 2
        end if
    else
        SyntaxError(); // Expected kind 1 or kind 3
    end if
end function
```

Listing 5.1: Coco/R outputs a parser for the language described by the given grammar.

5.1 The syntax graph structure

The Coco/R compiler generator is supported by several different data structures. In particular, there is a so-called *syntax graph* structure, that is used to represent each of the production rules of a grammar.

The graph data structure is composed of a set of *nodes*, where each node may be one of several different types; for instance, given the production rule $S = (ab|c)d$, the graph will consist of five nodes: four terminal nodes, representing 'a', 'b', 'c', and 'd', and an *alt* node (meaning "alternative") to represent the option symbol '|'. Moreover, each node has the properties *next*, *sub*, and *down* which are used to maintain the positional relationships of the given node with those of its neighboring nodes. The graph data structure can be visualized as depicted in Figure 5.1.

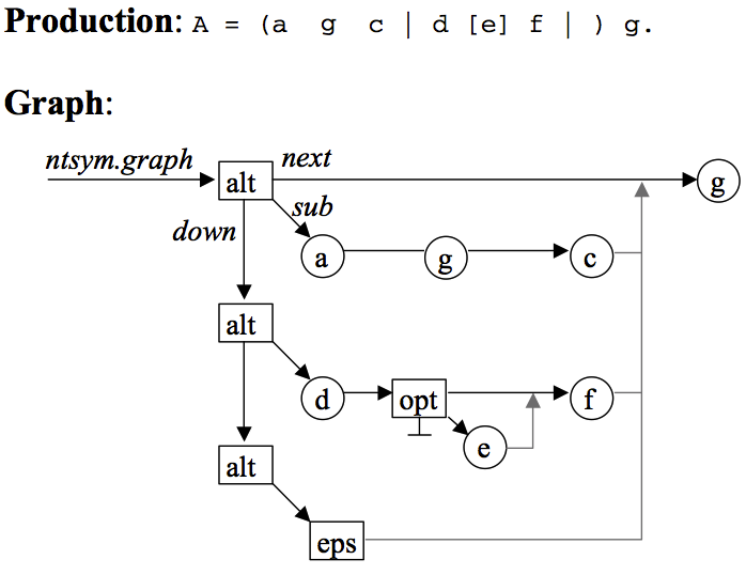

Figure 5.1: *Coco/R syntax graph data structure.*

5.2 AN INTRODUCTION TO GRAPHVIZ

GraphViz is open source software that is designed to easily construct high-quality diagrams. Given a formal description of a graph diagram, specified in the *DOT* language, a diagrammatic representation of the symbolic description is produced.

One interesting aspect of GraphViz descriptions are that they follow a declarative paradigm; wherein the user of the language need only declare nodes of the graph and their relations (i.e. which nodes are connected via edges), and define any associated labels. With those declarations, the software determines how to layout the elements of the graph with the given constraints in the plane. The DOT language offers some mechanisms for dictating the layout, in cases where one wants to override it. The aesthetics of the graphs are governed by the following principles Ganser et al. (1993):

1. Expose hierarchical structure in the graph. In particular, aim edges in the same general direction if possible.
2. Avoid visual anomalies that do not convey information about the underlying graph. For example, avoid edge crossings and shard bends.
3. Keep edges short.
4. Favor symmetry and balance.

5.2 The DOT language

The DOT language follows a declarative paradigm, meaning that the instructions describe the goal of the model while not explicitly stating *how* to meet that goal—that is left up to GraphViz's interpreter. It defines three kinds of objects: graphs, nodes, and edges. Graphs may be declared as directed or undirected. To demonstrate, consider the following listing:

Listing 5.1: DOT Language Example

```
digraph
{
   parent_node -> left_child_node [label=left_edge]
   parent_node -> right_child_node [label=right_edge]
}
```

This output is a binary tree consisting of three nodes and two edges, one for each child node. The *digraph* keyword tells GraphViz that the graph is of type digraph; moreover, the nodes follow on the left and right sides of the edge declarations, which are specified with the character sequence "$->$".

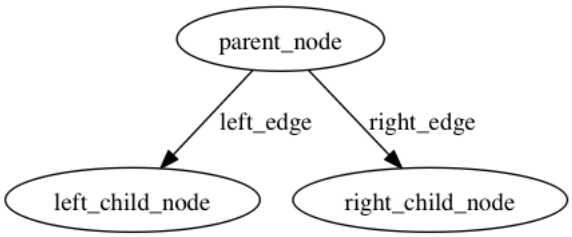

Figure 5.2: *An example of a graph defined using DOT.*

5.3 CFGANALYZER

CFGAnalyzer is an application that was developed for an unrelated project. All features of the application existed prior to this work, except for the ability to draw an augmented syntax graph given a grammar. The purpose of the application is to aid in the pedagogy of language design. More specifically, it allows a user to visualize important properties of context-free grammars (CFGs) that are used to describe programming languages. Conceptually, the software works by taking as input, a text file containing multiple CFGs, represented in BNF or EBNF notation. The input requires that terminal symbols be distinguished from non-terminals by the lettercasing—terminals are lowercase, non-terminals are uppercase; moreover, the end of each production rule is indicated by a period. The following listing demonstrates a valid CFG input file:

Listing 5.2: CFGAnalyzer input file

```
BNF Circular
Circular = X .
X = Y | x .
Y = Circular | y.
END
```

5.3 Interface design

The interface of the CFGAnalyzer uses a tabbed document interface (TDI), wherein each tab is used to display a different program feature. In particular, there are four tabs with the following titles: “CFG”, “Symbols in CFG”, “Parse Table”, and “Parse.” Figure 5.2 shows an example of the interface.

Upon executing the program, the first tab that is displayed is the “CFG” tab. This view contains a module on the left-hand side of the program window, that

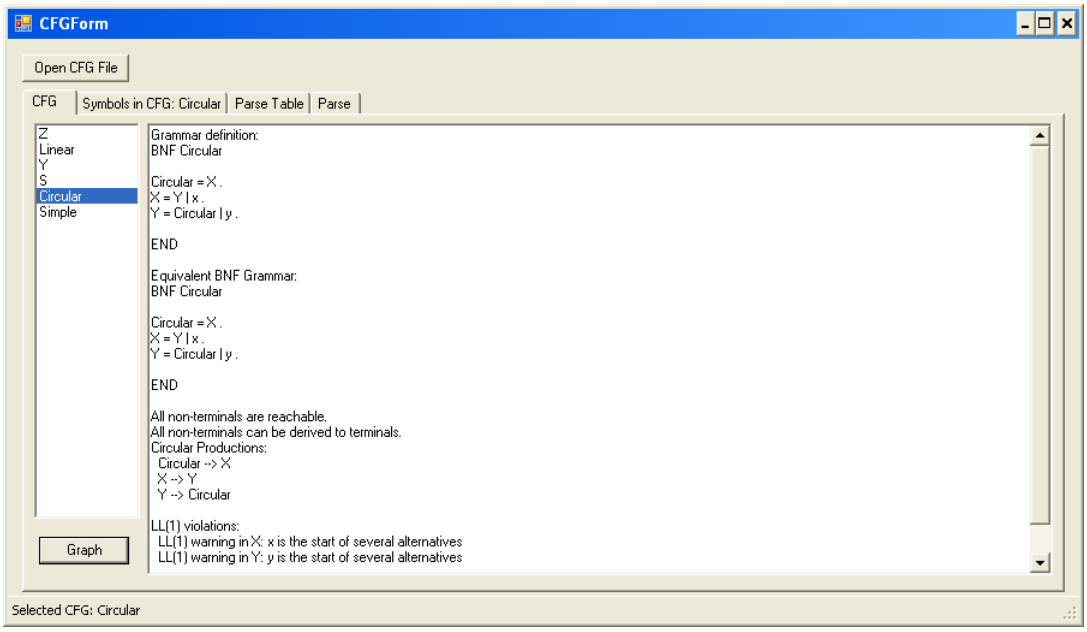

Figure 5.3: *Interface of the CFGAnalyzer.*

is used to list each of the CFGs defined in the input file. Each CFG is assigned a reference name, which is derived from the start symbol of each grammar. For instance, if the input file has two grammars with start symbol S and R, respectively, then the left-hand module will vertically list S and R. Selecting a grammar reference from the left-hand menu will result in a right-hand module being populated with that grammar's details. Specifically, if the pertinent grammar was denoted in EBNF in the input file, the application will display the grammar in its original form, and display its equivalent BNF formulation as well. Moreover, the module will report if there are unreachable non-terminal symbols, and if the grammar contains any circular production rules.

The tab appearing to the right of the "CFG" tab, the "Symbols in CFG" tab, also contains a left-hand and right-hand module. The left-hand side of the TDI is used to list all of the terminal symbols of a particular grammar, and the right-hand module is used to list the non-terminal symbols and attributes of those symbols: whether the symbol is nullable, reachable, useful (i.e. derives a terminal symbol), the

symbol's first set, and the symbol's follow set.

Finally, the "Parse Table" tab demonstrates a parse table for the selected grammar, and the "Parse" tab offers an interactive interface that allows a user to enter a word to test if it is accepted by the grammar.

The foregoing features of the CFGAnalyzer were created prior to this work. The feature introduced here, the new extension that was added to the application, is an implementation of the graph formalism from **Chapter 4**. This feature can be executed from the "CFG" tab, by selecting a grammar from the left-hand menu, and pressing the "Graph" button. Once the preceding button is pressed, a window will appear with the diagram representation of the grammar. In Figure 5.2, the input file has six grammars defined within it, and the selected grammar is referred to as "Circular" which is the first non-terminal symbol in that grammar. By pressing the "Graph" button, the ASG in Listing 5.2 is output to the screen.

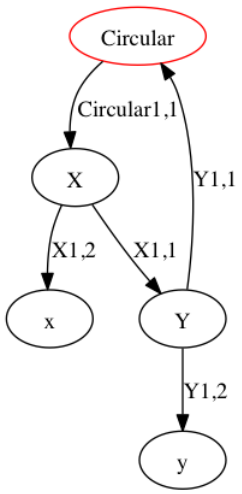

Figure 5.4: *An example of the CFGAnalyzer's ASG.*

5.3 Implementing the augmented syntax graph

In order to implement the graph, the CFGAnalyzer uses Coco/R to perform syntactic analysis, and utilizes the GraphViz software as a service for graphing the augmented

syntax graph (ASG). The Coco/R data structure, reviewed in section 5.1.1, undergoes a translation process which yields a DOT description of the ASG. The CFGAnalyzer invokes the GraphViz interpreter and uses the DOT description as input; the product of the process is the ASG. The following listing demonstrates the translation algorithm.

```
1: procedure DrawASG(N)
2:     for all nt ∈ N do
3:         x ← 1
4:         y ← 1
5:         node ← GetNode(nt)              ▷ reference the node of this non-terminal
6:         DrawNode(node, x, y)
7:     end for
8: end procedure
1: procedure DrawNode(node, x, y)
2:     if node ≠ ∅ then
3:         if HasSubNode(node) then
4:             subNode ← GetSubNode(node)
5:             symbol ← GetSymbol(subNode)
6:             DrawEdgeWithLabel(nt, symbol, (x, y))
7:             nextNode ← GetNextNode(node)
8:             while nextNode ≠ ∅ do
9:                 y ← y + 1
10:                symbol ← GetSymbol(nextNode)
11:                DrawEdgeWithLabel(nt, symbol, (x, y))
12:                nextNode ← GetNextNode(nextNode)
13:            end while
```

```
14:             x ← x + 1
15:             y ← 1
16:         else if HasSymbol(node) then
17:             symbol ← GetSymbol(node)
18:             DrawEdgeWithLabel(nt, symbol, (x, y))
19:         end if
20:         if HasDownNode(node) then
21:             downNode ← GetDownNode(node)
22:             DrawNode(downNode, x, y)
23:         end if
24:         if HasNextNode(node) then
25:             y ← y + 1
26:             nextNode ← GetDownNode(node)
27:             DrawNode(nextNode, x, y)
28:         end if
29:     end if
30: end procedure
```

Listing 5.2: Coco/R syntax graph to ASG DOT representation.

As previously discussed, the productions of a grammar in Coco/R are represented by a syntax graph data structure. Moreover, each non-terminal symbol has a pointer associated with it, which is used to point to the beginning of the syntax graph for the production rule of the given non-terminal symbol. In the above listing, GetNode(nt) is intended to represent the operation of accessing the first node of the syntax graph of the non-terminal symbol nt. The remaining pseudo functions represent the following operations: GetSubNode($node$), GetNextNode($node$), and GetDownNode($node$) are used to traverse through the syntax graph data structure;

and DrawEdgeWithLabel(nt, $symbol$, (x, y)) represents the operation of outputting the DOT instructions to place an edge between symbols nt and $symbol$, labelled with the ordered pair (x, y).

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

The current thesis has argued that representational format matters; yet, symbolic systems of representation have dominated the scientific literature; in part, this is due to the precision and expressiveness afforded by these formulae. However, it was shown that diagrammatic representations play a significant role in reasoning as well, and can often be as rigorous as a symbolic representation. Such examples can be found in the literature of formal language and automata theory; wherein, a system like a finite automaton can be formally specified using either a symbolic or diagrammatic representation. And, while diagrams can be as expressive as sentential formalisms, they can be superior in another way: Diagrams can make the abstract, concrete. One example of this is the *syntax diagram*.

Since Wirth (1973), a BNF grammar, specifying some programming languages has often been accompanied by a system of syntax diagrams, in order to better communicate the information given by the grammar. While syntax diagrams can help elucidate the syntactical patterns given by each production rule, there is, at least one limitation of the formalism: They do not capture all of the information about the grammar in one representation, and therefore do not allow one to externally envisage the grammar as a *whole*. "To bridge the gap", this work has investigated a novel idea, that of a diagrammatic representation that captures all of the production rules of a grammar in one formalism. The notational rules of these *augmented syntax graphs* (ASGs), yield diagrammatic representations that are isomorphs of BNF specifications. As such, one may translate the grammar from BNF to ASG—or, vice versa—without losing pertinent information.

The ASG is, indeed, a novel representation and, hence worth investigation in its own right. However, as computer science is an *applied* science, there is a prepossession given to objects that are of practical service in solving extant problems. Therefore,

this thesis has explored the application of the formalism, choosing to do so from a human-centered perspective. While the ASG was shown to have some favorability over an established formalism—the syntax diagram—this was argued from an rationally. One potential direction for the work could be to test the representation empirically, comparing it to other systems of representation. Other areas of future work might continue the human-centered theme by varying graph properties (e.g. replacing the numbering scheme with a coloring scheme), or could include exploring the ASG from a computational perspective. For instance, we are currently experimenting with the representation as a data structure in solving the *circular productions problem*.

CHAPTER 6

REFERENCES

(1976). *Airframe and Powerplant Mechanic General Handbook*. Federal Aviation Administration, first edition. 4

(2006). *Coco/R Tutorial*. Joint Modular Languages Conference. 47

Alexanderson, G. L. (2006). About the cover: Euler and konigsberg's bridges: A historical view. *Bulletin of the American Mathematical Society*, 43(4):567–573. 27

Backus, J. (1978). Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641. 15

Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., van Wijngaarden, A., and Woodger, M. (1960). Report on the algorithmic language algol 60. *Commun. ACM*, 3(5):299–314. 15

Barwise, J. and Etchemendy, J. (1995). Heterogeneous logic. In Chandrasekaran, B., Glasgow, J., and Narayanan, N. H., editors, *Diagrammatic Reasoning: Cognitive and Computational Perspectives*, pages 211–234. AAAI Press/The MIT Press. 10

Blair, D. C. (2003). Information retrieval and the philosophy of language. *Annual Review of Information Science and Technology*, 37:3–50. 7

Cheng, P. C.-H. and Simon, H. A. (1993). Scientific discovery and creative reasoning with diagrams. In Smith, S., W.-T. and Finke, R., editors, *The Creative Cognition Approach*. MIT Press. 14

Chomsky, N. (1957). *Syntactic Structures*. The Hague/Paris: Mouton. 16

Euler, L. (1768). *Lettres à une Princesse d'Allemagne*. l'Academie Imperiale des Sciences. 10

Feeney, A. and Webber, L. J. (2003). Analogical representation and graph comprehension. *Lecture Notes in Computer Science*, 2733:212–221. 8

Ganser, E. R., Koutsofios, E., North, S. C., and Vo, K.-P. (1993). A technique for drawing directed graphs. *IEEE Trans. Software Engineering*. 50

Ginsburg, S. and Rice, H. G. (1962). Two families of languages related to algol. *Journal of the ACM*, 9(3):350–371. 15

Hopcroft, J. and Ullman, J. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts. 36

Kosslyn, S. M. (1981). The medium and the message in mental imagery: A theory. *Psychological Review*, 88:46–66. 6

Kosslyn, S. M., Pinker, S., Smith, G. E., and Schwartz, S. P. (1981). On the demystification of mental imagery. In Block, N., editor, *Imagery*. The MIT Press. 5

Larkin, J. H. and Simon, H. A. (1995). Why a diagram is (sometimes) worth ten thousand words. In Chandrasekaran, B., Glasgow, J., and Narayanan, N. H., editors, *Diagrammatic Reasoning: Cognitive and Computational Perspectives*, pages 69–109. AAAI Press/The MIT Press. 7, 42

Linz, P. (2006). *An Introduction to Formal Language and Automata*. Jones and Bartlett Publishers, Inc. 15, 22, 35, 43

MIT (2007). Königsberg bridges. 28

Palmer, S. E. (1978). Fundamental aspects of cognitive representation. In Rosch, E. and Lloyd, B. L., editors, *Cognition and categorization*, pages 259–302. Lawrence Erlbaum. 3

Pierce, C. S. (1933). *Collected Papers*. Harvard University Press. 11

Pylyshyn, Z. (1973). What the mind's eye tells the mind's brain: A critique of mental imagery. *Psychological Bulletin*, 80:1–24. 6

Pylyshyn, Z. (1981). The imagery debate. In Block, N., editor, *Imagery*. The MIT Press. 6

Ross, K. A. and Wright, C. R. B. (2003). *Discrete Mathematics*. Prentice Hall, Upper Saddle River, New Jersey 07458, fifth edition. 27

Sato, Y., Mineshima, K., and Takemura, R. (2010). The efficacy of euler and venn diagrams in deductive reasoning: Empirical findings. In *Proceedings of the 6th International Conference on the Theory and Application of Diagrams*, pages 6–22. 11

Shepard, R. N. and Metzler, J. (1971). Mental rotation of three-dimensional objects. *Science*, 171:701–703. 6

Shimojima, A. (2001). The graphic-linguistic distinction. *Artificial Intelligence Review*, 15:5–27. 7

Shin, S. j. (1994). *The logical Status of Diagrams*. Cambridge University Press. 9, 11

Venn, J. (1881). *Symbolic Logic*. Macmillan. 10

Wirth, N. (1973). The programming language pascal. Technical report, ETH Zurich. 58

Woodworth, R. S. (1938). *Experimental Psychology*. Holt. 5

Zhang, J. (1997). The nature of external representations in problem solving. *Cognitive Science*, 21(2):179–217. 5