**2 Grammars as a Generating Device**

**2.1 Languages as Infinite Sets**

In computer science as in everyday parlance, a “grammar” serves to “describe” a “language”. If taken at face value, this correspondence, however, is misleading, since the computer scientist and the naive speaker mean slightly different things by the three terms. To establish our terminology and to demarcate the universe of discourse, we shall examine the above terms, starting with the last one.

**2.1.1 Language**

To the larger part of mankind, language is first and foremost a means of communication, to be used almost unconsciously, certainly so in the heat of a debate. Communication is brought about by sending messages, through air vibrations or through written symbols. Upon a closer look the language message (“utterances”) fall apart into sentences, which are composed of words, which in turn consist of symbol sequences when written. Languages can differ on all three levels of composition. The script can be slightly different, as between English and Irish, or very different, as between English and Chinese. Words tend to differ greatly, and even in closely related languages people call un cheval or ein Pferd, that which is known to others as a horse. Differences in sentence structure are often underestimated; even the closely related Dutch often has an almost Shakespearean word order: “Ik geloof je niet”, “I believe you not”, and more distantly related languages readily come up with constructions like the Hungarian “Pénzem van”, “Money-my is”, where the English say “I have money”.

The computer scientist takes a very abstracted view of all this. Yes, a language has sentences, and these sentences possess structure; whether they communicate something or not is not his concern, but information may possibly be derived from their structure and then it is quite all right to call that information the “meaning” of the sentence. And yes, sentences consist of words, which he calls “tokens”, each possibly carrying a piece of information, which is its contribution to the meaning of the whole sentence. But no, words cannot be broken down any further. This does not worry the computer scientist. With his love of telescoping solutions and multi-level techniques, he blithely claims that if words turn out to have structure after all, they are sentences in a different language, of which the letters are the tokens.

The practitioner of formal linguistics, henceforth called the formal-linguist (to distinguish him from the “formal linguist”, the specification of whom is left to the imagination of the reader) again takes an abstracted view of this. A language is a “set” of sentences, and each sentence is a “sequence” of “symbols”; that is all there is: no meaning, no structure, either a sentence belongs to the language or it does not. The only property of a symbol is that it has an identity; in any language there are a certain number of different symbols, the alphabet, and that number must be finite. Just for convenience we write these symbols as a, b, c, . . . , but ✆, ✈, ❐, . . . would do equally well, as long as there are enough symbols. The word sequence means that the symbols in each sentence are in a fixed order and we should not shuffle them. The word set means an unordered collection with all the duplicates removed. A set can be written down by writing the objects in it, surrounded by curly brackets. All this means that to the formal-linguist the following is a language: a, b, ab, ba, and so is {a, aa, aaa, aaaa, . . . } although the latter has notational problems that will be solved later. In accordance with the correspondence that the computer scientist sees between sentence/word and word/letter, the formal-linguist also calls a sentence a word and he says that “the word ab is in the language {a, b, ab, ba}”.

Now let us consider the implications of these compact but powerful ideas.

To the computer scientist, a language is a probably infinitely large set of sentences, each composed of tokens in such a way that it has structure; the tokens and the structure cooperate to describe the semantics of the sentence, its “meaning” if you will. Both the structure and the semantics are new, that is, were not present in the formal model, and it is his responsibility to provide and manipulate them both. To a computer scientist 3+4×5 is a sentence in the language of “arithmetics on single digits” (“single digits” to avoid having an infinite number of symbols); its structure can be shown by inserting parentheses:

(3+ (4×5)); and its semantics is probably 23.

To the linguist, whose view of languages, it has to be conceded, is much more normal than that of either of the above, a language is an infinite set of possibly interrelated sentences. Each sentence consists, in a structured fashion, of words which have a meaning in the real world. Structure and words together give the sentence a meaning, which it communicates. Words, again, possess structure and are composed of letters; the letters cooperate with some of the structure to give a meaning to the word. The heavy emphasis on semantics, the relation with the real world and the integration of the two levels sentence/word and word/letters are the domain of the linguist. “The circle spins furiously” is a sentence, “The circle sleeps red” is nonsense.

The formal-linguist holds his views of language because he wants to study the fundamental properties of languages in their naked beauty; the computer scientist holds his because he wants a clear, well-understood and unambiguous means of describing objects in the computer and of communication with the computer, a most exacting communication partner, quite unlike a human; and the linguist holds his view of language because it gives him a formal tight grip on a seemingly chaotic and perhaps infinitely complex object: natural language.

**2.1.2 Grammars**

Everyone who has studied a foreign language knows that a grammar is a book of rules and examples which describes and teaches the language. Good grammars make a careful distinction between the sentence/word level, which they often call syntax or syntaxis and the word/letter level, which they call morphology. Syntax contains rules like “pour que is followed by the subjunctive, but parce que is not”. Morphology contains rules like “the plural of an English noun is formed by appending an -s, except when the word ends in -s, -sh, -o, -ch or -x, in which case -es is appended, or when the word has an irregular plural.”

We skip the computer scientist’s view of a grammar for the moment and proceed immediately to that of the formal-linguist. His view is at the same time very abstract and quite similar to the layman’s: a grammar is any exact, finite-size, complete description of the language, i.e., of the set of sentences. This is in fact the school grammar, with the fuzziness removed. Although it will be clear that this definition has full generality, it turns out that it is too general, and therefore relatively powerless. It includes descriptions like “the set of sentences that could have been written by Chaucer”; platonically speaking this defines a set, but we have no way of creating this set or testing whether a given sentence belongs to this language. This particular example, with its “could have been” does not worry the formal-linguist, but there are examples closer to his home that do. “The longest block of consecutive sevens in the decimal expansion of π” describes a language that has at most one word in it (and then that word will consist of sevens only), and as a definition it is exact, of finite-size and complete. One bad thing with it, however, is that one cannot find this word: suppose one finds a block of one hundred sevens after billions and billions of digits, there is always a chance that further on there is an even longer block. And another bad thing is that one cannot even know if this longest block exists at all. It is quite possible that, as one proceeds further and further up the decimal expansion of π, one would find longer and longer stretches of sevens, probably separated by ever-increasing gaps. A comprehensive theory of the decimal expansion of π might answer these questions, but no such theory exists.

For these and other reasons, the formal-linguists have abandoned their static, platonic view of a grammar for a more constructive one, that of the generative grammar: a generative grammar is an exact, fixed-size recipe for constructing the sentences in the language. This means that, following the recipe, it must be possible to construct each sentence of the language (in a finite number of actions) and no others. This does not mean that, given a sentence, the recipe tells us how to construct that particular sentence, only that it is possible to do so. Such recipes can have several forms, of which some are more convenient than others.

The computer scientist essentially subscribes to the same view, often with the additional requirement that the recipe should imply how a sentence can be constructed.

**2.1.3 Problems with Infinite Sets**

The above definition of a language as a possibly infinite set of sequences of symbols and of a grammar as a finite recipe to generate these sentences immediately gives rise to two embarrassing questions:

1. How can finite recipes generate enough infinite sets of sentences?

2. If a sentence is just a sequence and has no structure and if the meaning of a sentence derives, among other things, from its structure, how can we assess the meaning of a sentence?

These questions have long and complicated answers, but they do have answers. We shall first pay some attention to the first question and then devote the main body of this book to the second.

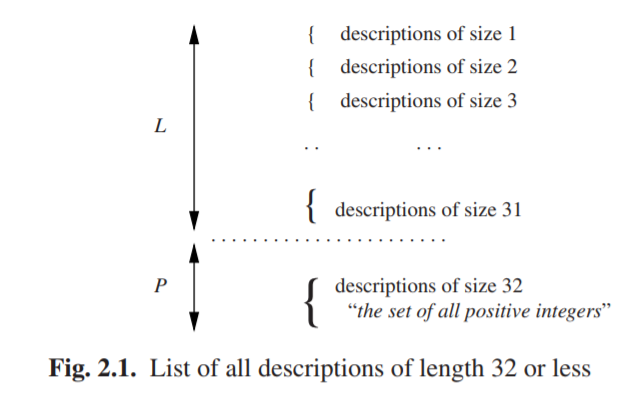
**2.1.3.1 Infinite Sets from Finite Descriptions**

In fact there is nothing wrong with getting a single infinite set from a single finite description: “the set of all positive integers” is a very finite-size description of a definitely infinite-size set. Still, there is something disquieting about the idea, so we shall rephrase our question: “Can all languages be described by finite descriptions?” As the lead-up already suggests, the answer is “No”, but the proof is far from trivial. It is, however, very interesting and famous, and it would be a shame not to present at least an outline of it here.

**2.1.3.2 Descriptions can be Enumerated**

The proof is based on two observations and a trick. The first observation is that descriptions can be listed and given a number. This is done as follows. First, take all descriptions of size one, that is, those of only one letter long, and sort them alphabetically. This is the beginning of our list. Depending on what, exactly, we accept as a description, there may be zero descriptions of size one, or 27 (all letters + space), or 95 (all printable ASCII characters) or something similar; this is immaterial to the discussion which follows.

Second, we take all descriptions of size two, sort them alphabetically to give the second chunk on the list, and so on for lengths 3, 4 and further. This assigns a position on the list to each and every description. Our description “the set of all positive integers”, for example, is of size 32, not counting the quotation marks. To find its position on the list, we have to calculate how many descriptions there are with less than 32 characters, say L. We then have to generate all descriptions of size 32, sort them and determine the position of our description in it, say P, and add the two numbers L and P. This will, of course, give a huge number1 but it does ensure that the description is on the list, in a well-defined position; see Figure 2.1.



Two things should be pointed out here. The first is that just listing all descriptions alphabetically, without reference to their lengths, would not do: there are already infinitely many descriptions starting with an “a” and no description starting with a higher letter could get a number on the list. The second is that there is no need to actually do all this. It is just a thought experiment that allows us to examine and draw conclusions about the behavior of a system in a situation which we cannot possibly examine physically.

Also, there will be many nonsensical descriptions on the list; it will turn out that this is immaterial to the argument. The important thing is that all meaningful descriptions are on the list, and the above argument ensures that.

**2.1.3.3 Languages are Infinite Bit-Strings**

We know that words (sentences) in a language are composed of a finite set of symbols; this set is called quite reasonably the “alphabet”. We will assume that the symbols in the alphabet are ordered. Then the words in the language can be ordered too. We shall indicate the alphabet by Σ.

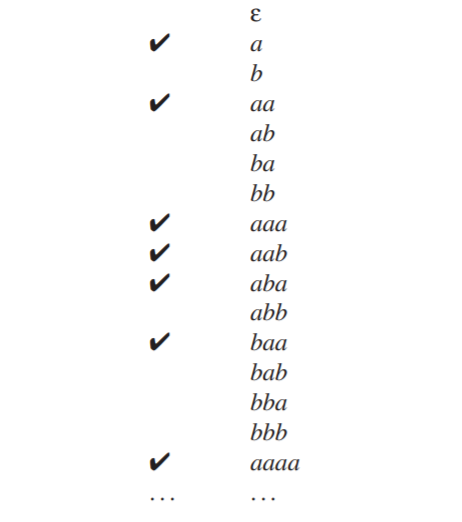
Now the simplest language that uses alphabet Σ is that which consists of all words that can be made by combining letters from the alphabet. For the alphabet Σ ={a, b} we get the language { , a, b, aa, ab, ba, bb, aaa, . . . }. We shall call this language Σ∗, for reasons to be explained later; for the moment it is just a name.

The set notation Σ∗ above started with “ { , a,”, a remarkable construction; the first word in the language is the empty word, the word consisting of zero as and zero bs. There is no reason to exclude it, but, if written down, it may easily be overlooked, so we shall write it as ε (epsilon), regardless of the alphabet. So, Σ∗ = { ε, a, b, aa, ab, ba, bb, aaa, . . . }. In some natural languages, forms of the present tense of the verb “to be” are empty words, giving rise to sentences of the form “I student”, meaning “I am a student.” Russian and Hebrew are examples of this.

Since the symbols in the alphabet Σ are ordered, we can list the words in the language Σ∗, using the same technique as in the previous section: First, all words of size zero, sorted; then all words of size one, sorted; and so on. This is actually the order already used in our set notation for Σ∗.

The language Σ∗ has the interesting property that all languages using alphabet Σ are subsets of it. That means that, given another possibly less trivial language over Σ, called L, we can go through the list of words in Σ∗ and put ticks on all words that are in L. This will cover all words in L, since Σ∗ contains any possible word over Σ.

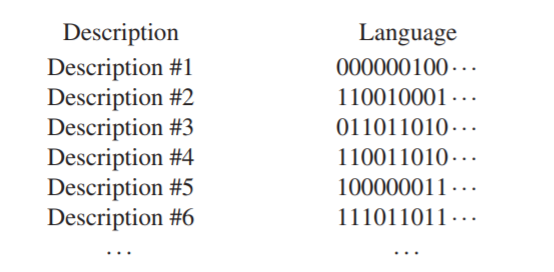
Suppose our language L is “the set of all words that contain more as than bs”. L is the set {a, aa, aab, aba, baa, . . . }. The beginning of our list, with ticks, will look as follows:



Given the alphabet with its ordering, the list of blanks and ticks alone is entirely sufficient to identify and describe the language. For convenience we write the blank as a 0 and the tick as a 1 as if they were bits in a computer, and we can now write L = 0101000111010001··· (and Σ∗ = 1111111111111111···). It should be noted that this is true for any language, be it a formal language like L, a programming language like Java or a natural language like English. In English, the 1s in the bitstring will be very scarce, since hardly any arbitrary sequence of words is a good English sentence (and hardly any arbitrary sequence of letters is a good English word, depending on whether we address the sentence/word level or the word/letter level).

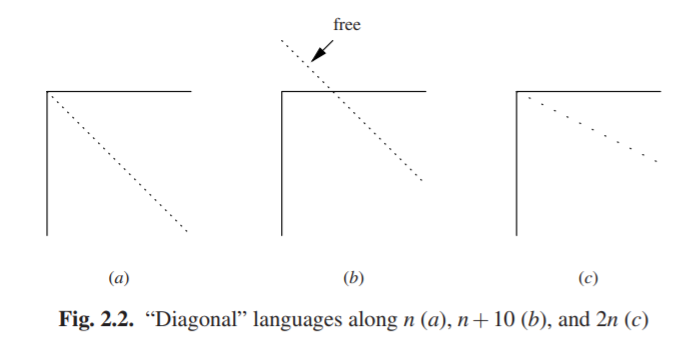
**2.1.3.4 Diagonalization**

The previous section attaches the infinite bit-string 0101000111010001··· to the description “the set of all the words that contain more as than bs”. In the same vein we can attach such bit-strings to all descriptions. Some descriptions may not yield a language, in which case we can attach an arbitrary infinite bit-string to it. Since all descriptions can be put on a single numbered list, we get, for example, the following picture:



At the left we have all descriptions, at the right all languages they describe. We now claim that many languages exist that are not on the list of languages above: the above list is far from complete, although the list of descriptions is complete. We shall prove this by using the diagonalization process (“Diagonalverfahren”) of Cantor.

Consider the language C = 100110···, which has the property that its n-th bit is unequal to the n-th bit of the language described by Description #n. The first bit of C is a 1, because the first bit for Description #1 is a 0; the second bit of C is a 0, because the second bit for Description #2 is a 1, and so on. C is made by walking the NW to SE diagonal of the language field and copying the opposites of the bits we meet. This is the diagonal in Figure 2.2(a). The language C cannot be on the list! It cannot be on line 1, since its first bit differs (is made to differ, one should say) from that on line 1, and in general it cannot be on line n, since its n-th bit will differ from that on line n, by definition.



So, in spite of the fact that we have exhaustively listed all possible finite descriptions, we have at least one language that has no description on the list. But there exist more languages that are not on the list. Construct, for example, the language whose n+10-th bit differs from the n+10-th bit in Description #n. Again it cannot be on the list since for every n > 0 it differs from line n in the n+10-th bit. But that means that bits 1. . . 9 play no role, and can be chosen arbitrarily, as shown in Figure 2.2(b); this yields another 2^9 = 512 languages that are not on the list. And we can do even much better than that! Suppose we construct a language whose 2n-th bit differs from the 2n-th bit in Description #n (c). Again it is clear that it cannot be on the list, but now every odd bit is left unspecified and can be chosen freely! This allows us to create freely an infinite number of languages none of which allows a finite description; see the slanting diagonal in Figure 2.2. In short, for every language that can be described there are infinitely many that cannot.

The diagonalization technique is described more formally in most books on theoretical computer science; see e.g., Rayward-Smith [393, pp. 5-6], or Sudkamp [397, Section 1.4].

**2.1.3.5 Discussion**

The above demonstration shows us several things. First, it shows the power of treating languages as formal objects. Although the above outline clearly needs considerable amplification and substantiation to qualify as a proof (for one thing it still has to be clarified why the above explanation, which defines the language C, is not itself on the list of descriptions; see Problem 2.1, it allows us to obtain insight into properties not otherwise assessable.

Secondly, it shows that we can only describe a tiny subset (not even a fraction) of all possible languages: there is an infinity of languages out there, forever beyond our reach.

Thirdly, we have proved that, although there are infinitely many descriptions and infinitely many languages, these infinities are not equal to each other, and the latter is larger than the former. These infinities are called ℵ0 and ℵ1 by Cantor, and the above is just a special case of his proof that ℵ0 < ℵ1.

**2.1.4 Describing a Language through a Finite Recipe**

A good way to build a set of objects is to start with a small object and to give rules for how to add to it and construct new objects from it. “Two is an even number and the sum of two even numbers is again an even number” effectively generates the set of all even numbers. Formalists will add “and no other numbers are even”, but we will take that as understood.

Suppose we want to generate the set of all enumerations of names, of the type “Tom, Dick and Harry”, in which all names but the last two are separated by commas. We will not accept “Tom, Dick, Harry” nor “Tom and Dick and Harry”, but we shall not object to duplicates: “Grubb, Grubb and Burrowes”2 is all right. Although these are not complete sentences in normal English, we shall still call them “sentences” since that is what they are in our midget language of name enumerations. A simpleminded recipe would be:

0. Tom is a name, Dick is a name, Harry is a name;

1. a name is a sentence;

2. a sentence followed by a comma and a name is again a sentence;

3. before finishing, if the sentence ends in “, name”, replace it by “and name”.

Although this will work for a cooperative reader, there are several things wrong with it. Clause 3 is especially wrought with trouble. For example, the sentence does not really end in “, name”, it ends in “, Dick” or such, and “name” is just a symbol that stands for a real name; such symbols cannot occur in a real sentence and must in the end be replaced by a real name as given in clause 0. Likewise, the word “sentence” in the recipe is a symbol that stands for all the actual sentences. So there are two kinds of symbols involved here: real symbols, which occur in finished sentences, like “Tom”, “Dick”, a comma and the word “and”; and there are intermediate symbols, like “sentence” and “name” that cannot occur in a finished sentence. The first kind corresponds to the words or tokens explained above; the technical term for them is terminal symbols(or terminals for short). The intermediate symbols are called non-terminals, a singularly uninspired term. To distinguish them, we write terminals in lower case letters and start non-terminals with an upper case letter. Non-terminals are called (grammar) variables or syntactic categories in linguistic contexts.

To stress the generative character of the recipe, we shall replace “X is a Y” by “Y may be replaced by X”: if “tom” is an instance of a Name, then everywhere we have a Name we may narrow it down to “tom”. This gives us:

0. Name may be replaced by “tom”

Name may be replaced by “dick”

Name may be replaced by “harry”

1. Sentence may be replaced by Name

2. Sentence may be replaced by Sentence, Name

3. “, Name” at the end of a Sentence must be replaced by “and Name” before Name is replaced by any of its replacements

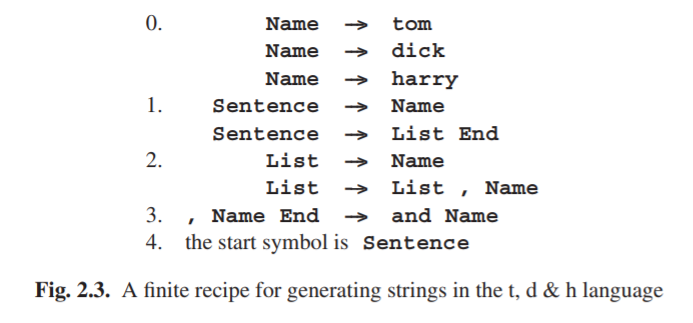
4. a sentence is finished only when it no longer contains non-terminals

5. we start our replacement procedure with Sentence

Clause 0 through 3 describe replacements, but 4 and 5 are different. Clause 4 is not specific to this grammar. It is valid generally and is one of the rules of the game. Clause 5 tells us where to start generating. This name is quite naturally called the start symbol, and it is required for every grammar.

Clause 3 still looks worrisome; most rules have “may be replaced”, but this one has “must be replaced”, and it refers to the “end of a Sentence”. The rest of the rules work through replacement, but the problem remains how we can use replacement to test for the end of a Sentence. This can be solved by adding an end marker after it. And if we make the end marker a non-terminal which cannot be used anywhere except in the required replacement from “, Name” to “and Name”, we automatically enforce the restriction that no sentence is finished unless the replacement test has taken place. For brevity we write ---> instead of “may be replaced by”; since terminal and non-terminal symbols are now identified as technical objects we shall write them in a typewriter-like typeface. The part before the ---> is called the left-hand side, the part after it the right-hand side. This results in the recipe in Figure 2.3.

This is a simple and relatively precise form for a recipe, and the rules are equally straightforward: start with the start symbol, and keep replacing until there are no non-terminals left.

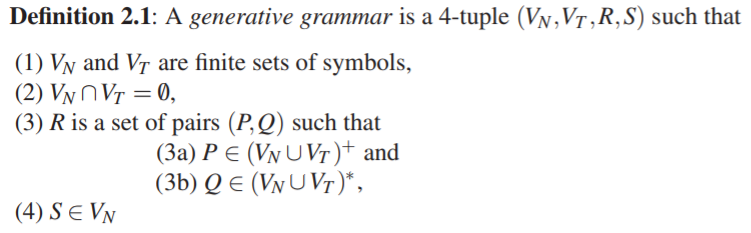


**2.2 Formal Grammars**

The above recipe form, based on replacement according to rules, is strong enough to serve as a basis for formal grammars. Similar forms, often called “rewriting systems”, have a long history among mathematicians, and were already in use several centuries B.C. in India (see, for example, Bhate and Kak [411]). The specific form of Figure 2.3 was first studied extensively by Chomsky [385]. His analysis has been the foundation for almost all research and progress in formal languages, parsers and a considerable part of compiler construction and linguistics.

**2.2.1 The Formalism of Formal Grammars**

Since formal languages are a branch of mathematics, work in this field is done in a special notation. To show some of its flavor, we shall give the formal definition of a grammar and then explain why it describes a grammar like the one in Figure 2.3. The formalism used is indispensable for correctness proofs, etc., but not for understanding the principles; it is shown here only to give an impression and, perhaps, to bridge a gap.



A 4-tuple is just an object consisting of 4 identifiable parts; they are the non-terminals, the terminals, the rules and the start symbol, in that order. The above definition does not tell this, so this is for the teacher to explain. The set of non-terminals is named VN and the set of terminals VT . For our grammar we have:

VN = {Name, Sentence, List, End}

VT = {tom, dick, harry, ,, and}

(note the , in the set of terminal symbols).

The intersection of VN and VT (2) must be empty, indicated by the symbol for the empty set, ∅. So the non-terminals and the terminals may not have a symbol in common, which is understandable.

R is the set of all rules (3), and P and Q are the left-hand sides and right-hand sides, respectively. Each P must consist of sequences of one or more non-terminals and terminals and each Q must consist of sequences of zero or more non-terminals and terminals. For our grammar we have:

R = {(Name, tom), (Name, dick), (Name, harry),

(Sentence, Name), (Sentence, List End), (List, Name),

(List, List , Name), (, Name End, and Name)}

Note again the two different commas.

The start symbol S must be an element of VN, that is, it must be a non-terminal:

S = Sentence

This concludes our field trip into formal linguistics. In short, the mathematics of formal languages is a language, a language that has to be learned; it allows very concise expression of what and how but gives very little information on why. Consider this book a translation and an exegesis.

**2.2.2 Generating Sentences from a Formal Grammar**

The grammar in Figure 2.3 is what is known as a phrase structure grammar for our t,d&h language (often abbreviated to PS grammar). There is a more compact notation, in which several right-hand sides for one and the same left-hand side are grouped together and then separated by vertical bars, |. This bar belongs to the formalism, just as the arrow --->, and can be read “or else”. The right-hand sides separated by vertical bars are also called alternatives. In this more concise form our grammar becomes

0. Name ---> tom | dick | harry

1. Sentences ---> Name | List End

2. List ---> Name | Name , List

3. , Name End ---> and Name

where the non-terminal with the subscript s is the start symbol. (The subscript identifies the symbol, not the rule.)

Now let us generate our initial example from this grammar, using replacement according to the above rules only. We obtain the following successive forms for Sentence:

Intermediate form Rule used Explanation

Sentence the start symbol

List End Sentence ---> List End rule 1

Name , List End List ---> Name , List rule 2

Name , Name , List End List ---> Name , List rule 2

Name , Name , Name End List ---> Name rule 2

Name , Name and Name , Name End ---> and Name rule 3

tom , dick and harry rule 0, three times

The intermediate forms are called sentential forms. If a sentential form contains no non-terminals it is called a sentence and belongs to the generated language. The transitions from one line to the next are called production steps and the rules are called production rules, for obvious reasons.

The production process can be made more visual by drawing connective lines between corresponding symbols, using a “graph”. A graph is a set of nodes connected by a set of edges. A node can be thought of as a point on paper, and an edge as a line, where each line connects two points; one point may be the end point of more than one line. The nodes in a graph are usually “labeled”, which means that they have been given names, and it is convenient to draw the nodes on paper as bubbles with their names in them, rather than as points. If the edges are arrows, the graph is a directed graph; if they are lines, the graph is undirected. Almost all graphs used in parsing techniques are directed.

The graph corresponding to the above production process is shown in Figure 2.4. Such a picture is called a production graph or syntactic graph and depicts the syntactic structure (with regard to the given grammar) of the final sentence. We see that the production graph normally fans out downwards, but occasionally we may see starlike constructions, which result from rewriting a group of symbols.

A cycle in a graph is a path from a node N following the arrows, leading back to N. A production graph cannot contain cycles; we can see that as follows. To get a cycle we would need a non-terminal node N in the production graph that has produced children that are directly or indirectly N again. But since the production process always makes new copies for the nodes it produces, it cannot produce an already existing node. So a production graph is always “acyclic”; directed acyclic graphs are called dags.

It is patently impossible to have the grammar generate tom, dick, harry, since any attempt to produce more than one name will drag in an End and the only way to get rid of it again (and get rid of it we must, since it is a non-terminal) is to have it absorbed by rule 3, which will produce the and. Amazingly, we have succeeded in implementing the notion “must replace” in a system that only uses “may replace”; looking more closely, we see that we have split “must replace” into “may replace” and “must not be a non-terminal”.

Apart from our standard example, the grammar will of course also produce many other sentences; examples are:

harry and tom

harry

tom, tom, tom and tom

and an infinity of others. A determined and foolhardy attempt to generate the incorrect form without the and will lead us to sentential forms like:

tom, dick, harry End

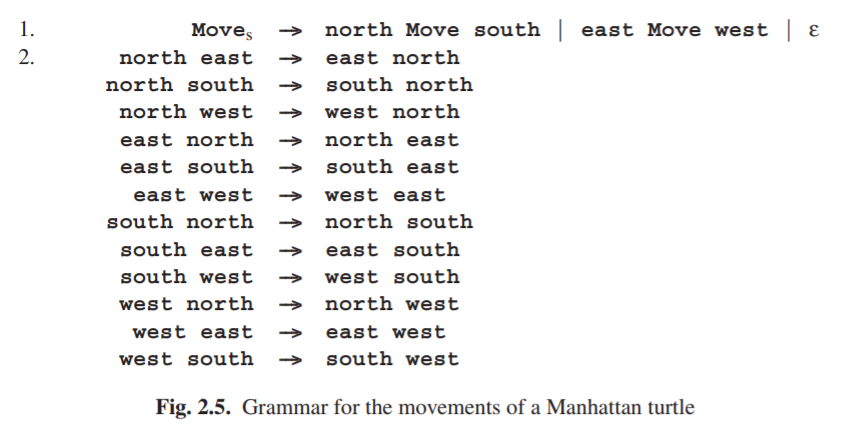
which are not sentences and to which no production rule applies. Such forms are called blind alleys. As the right arrow in a production rule already suggests, the rule may not be applied in the reverse direction.

**2.2.3 The Expressive Power of Formal Grammars**

The main property of a formal grammar is that it has production rules, which may be used for rewriting part of the sentential form (= sentence under construction) and a starting symbol which is the mother of all sentential forms. In the production rules we find non-terminals and terminals; finished sentences contain terminals only. That is about it: the rest is up to the creativity of the grammar writer and the sentence producer.

This is a framework of impressive frugality and the question immediately rises: Is it sufficient? That is hard to say, but if it is not, we do not have anything more expressive. Strange as it may sound, all other methods known to mankind for generating sets have been proved to be equivalent to or less powerful than a phrase structure grammar. One obvious method for generating a set is, of course, to write a program generating it, but it has been proved that any set that can be generated by a program can be generated by a phrase structure grammar. There are even more arcane methods, but all of them have been proved not to be more expressive. On the other hand there is no proof that no such stronger method can exist. But in view of the fact that many quite different methods all turn out to halt at the same barrier, it is highly unlikely3 that a stronger method will ever be found. See, e.g. Révész [394, pp 100-102].

As a further example of the expressive power we shall give a grammar for the movements of a Manhattan turtle. A Manhattan turtle moves in a plane and can only move north, east, south or west in distances of one block. The grammar of Figure 2.5 produces all paths that return to their own starting point. As to rule 2, it should be



noted that many authors require at least one of the symbols in the left-hand side to be a non-terminal. This restriction can always be enforced by adding new non-terminals.

The simple round trip north east south west is produced as shown in Figure 2.6 (names abbreviated to their first letter). Note the empty alternative in rule 1 (the ε), which results in the dying out of the third M in the above production graph.

