# Symbolic Dynamics: A discourse into formal languages

Torben Vaarby Laursen

September 28, 1999

## 1 Introduction

Let $\mathcal{A}$ denote an alphabet, i.e. a finite collection of symbols. The *Kleene closure* of $\mathcal{A}$ is the collection of all finite words made up of symbols from $\mathcal{A}$. The Kleene closure of $\mathcal{A}$ is denoted $\mathcal{A}^*$. A *language* $L$ over $\mathcal{A}$ is some (possibly infinite) subset $L \subset \mathcal{A}^*$.

We will be concerned with various kinds of languages and their properties. Throughout the text we will try to draw parallels to Symbolic Dynamics where appropriate. The main application is syntax analysis of computer languages.

## 2 Languages

For a given alphabet, $\mathcal{A}$, there are uncountably languages over $\mathcal{A}$. To see this note that $\mathcal{A}^*$ must be countable since we can arrange the words according to length. Hence the set of languages over $\mathcal{A}$, that is the powerset of $\mathcal{A}^*$ must be uncountable.

This means that we cannot ever characterise all forms of languages in a positive manner. The best we can hope for is to define a large class of interesting languages in a constructive way.

**Remark 1** The Kleene closure of an alphabet is the same as the language of the full shift, i.e. $\mathcal{A}^* = \mathcal{B}(\mathcal{A}^{\mathbb{Z}})$.

### 2.1 Characterizations of languages

Languages are usually characterised in two ways: *Grammars* and *Finite State Machines* (also called *automata*). Grammars were originally developed by Noah Chomsky as a way of specifying structure in natural languages [2]. Around 1960 his ideas spread to computer science because they offered a natural way of specifying the *syntax* of programming languages. In this paper we will only concern ourselves with grammars, even though the theory of sofic shifts have borrowed heavily from automata theory.

## 2.2 Grammars

Grammars offer a way of specifying the allowed words of a language. In particular we are interested in the set of syntactically correct Pascal (or some other fixed, but arbitrary computer language) programs. A *compiler* is a computer program which reads a source program in some computer language and translates it into object code which can be executed on a particular machine. How does the compiler know whether the program it reads adheres to the rules of the language, i.e. is syntactically correct? Grammars provides a framework for answering such questions.

In Pascal one can assign values to variables using the following syntax:

```
x := 2+8;
```

Here the variable x is assigned the value 10. In a more abstract way we can define an assignment as

$$\textit{Assignment} \longrightarrow \textit{Identifier} \ \texttt{:=} \ \textit{Expression} \ \texttt{;}$$

where the $\longrightarrow$ denotes "can take the form" and *Identifier* and *Expression* have definitions of their own. For instance

$$\begin{array}{lcl}
\textit{Identifier} & \longrightarrow & \textit{Letter String} \\
\textit{Letter} & \longrightarrow & \texttt{\_} \\
\textit{Letter} & \longrightarrow & \texttt{a} \\
 & \vdots & \\
\textit{Letter} & \longrightarrow & \texttt{Z} \\
\textit{String} & \longrightarrow & \epsilon \\
\textit{String} & \longrightarrow & \textit{Letter String} \\
\textit{String} & \longrightarrow & \textit{Digit String} \\
 & \vdots &
\end{array}$$

That is, a identifier is a string of length at least one which begins with either a letter or the underscore character and otherwise can be made up of digits and letters. This also illustrates the use of grammars as a precise way of specifying syntax. The description in natural language of an identifier is not entirely clear (it might not even be correct).

`:=`, `;`, `a`, etc. are called *terminals* whereas *Identifier*, *Expression*, etc. are called *Nonterminals.* A line such as

$$\textit{Assignment} \longrightarrow \textit{Identifier} \ \texttt{:=} \ \textit{Expression} \ \texttt{;}$$

is called a *production rule.* We are now ready to give a formal definition of a grammar.

**Definition 2** A *grammar* $G$ is a tuple $(S, N, T, P)$ where $S \in N$ and $N \cap T = \emptyset$. $S$ is the *start symbol*, $N$ is a set of nonterminals, $T$ is a set of terminal symbols and $P$ is a set of production rules which takes the following form

$$\alpha \longrightarrow \beta \quad \alpha, \beta \in (N \cup T)^*, \ |\alpha| \geq 1.$$

Each of the sets $N, T, P$ has to be finite.

**Remark 3** We can think of $T$ as an alphabet. $T$ is often a set of *tokens*, that is a set of atoms with regards to meaning. For example, in programming languages, the symbols of the alphabet are the ascii characters, these form into tokens, such as the reserved words `if` and `else` for instance. In symbolic dynamics we have a similar construction in the higher block shifts where the symbols in the higher block alphabet are made up of strings of the symbols in the underlying alphabet. The similarity is very shallow though, since tokens can have different length and do not have to overlap progressively.

## 2.3 Language of a grammar

The allowed words of a grammar can be found by starting with the startsymbol and use the production rules repeatedly until one ends up with a string consisting only of terminal symbols. Note that we naturally insist that the left hand side of a production rule is nonempty. Let $\Rightarrow$ be the relation on $(N \cup T)^*$ defined by

$$\gamma\alpha\delta \Rightarrow \gamma\beta\delta \text{ whenever } \alpha \longrightarrow \beta \in P \text{ and } \gamma, \delta \in (N \cup T)^*$$

Let $\Rightarrow^*$ denote the reflexive and transitive closure of $\Rightarrow$, that is

$$\begin{array}{ll} \gamma & \Rightarrow^* \gamma \\ \gamma & \Rightarrow^* \eta \text{ if } \gamma \Rightarrow \delta \text{ and } \delta \Rightarrow^* \eta \end{array}$$

**Definition 4** The language $L(G)$ of a grammar $G$ is the set

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

# 3 Types of languages

We are interested in two questions. Given a language $L$ can we find a grammar $G$ which *generates* $L$, that is $L = L(G)$? And for a given grammar $G$ are we able to decide whether a word $w \in L(G)$? According to the introduction the answer to the first question is no. The set of languages which can be generated by a grammar is called *type* 0 languages, but since grammars are finite constructions, there are only countable many type 0 languages.

The answer to the second question is also somewhat depressingly negative. It is in general undecidable whether a word can be generated by a type 0 grammar. That means we cannot necessarily make a compiler for a type 0 language, since we are not in general able to discern between correct and incorrect programs.

If we place restrictions on the form production rules may take, things are looking a little better.

We define type 1 languages to be the languages generated by grammars where the production rules all have the form

$$\alpha \longrightarrow \beta \quad \alpha, \beta \in (N \cup T)^*, \ |\beta| \geq |\alpha| \geq 1.$$

We note that this means we cannot generate the empty string, since the right hand side of a production has to be longer than the left hand side, which is non-empty. To allow the empty string as a word in the language we can add the production rule

$$S \longrightarrow \epsilon$$

along with the condition that the startsymbol $S$ must not occur on any right hand sides of production rules.

Let $w \in T^*$ with $|w| = n$. Then it can be shown that deciding whether $w \in L(G)$ in the worst case takes $O(2^n)$ time.

While this is clearly computable it is far too slow to have any use in practice. Hence we are looking for further restrictions on the grammar. It turns out that if we restrict ourselves to having only nonterminals on the left hand side of productions we get what we need.

**Definition 5** A *contextfree* grammar is a grammar where the productions have the following form:

$$\alpha \longrightarrow \beta \quad \alpha \in N, \ \beta \in (N \cup T)^*.$$

A *contextfree language* is one generated by a contextfree grammar.

**Example 6** The language $\{a^n b^n \mid n \in \mathbb{N}\}$ over $\{a, b\}$ is contextfree since it is generated by the grammar

$$\begin{array}{lcl} S & \longrightarrow & N \\ N & \longrightarrow & ab \\ N & \longrightarrow & aNb \end{array}$$

Note that we do not allow the empty string, hence the introduction of the nonterminal $N$.

**Remark 7** Contextfree languages are also called type 2 languages. Technically we should not allow empty right hand sides, but it can be shown [3] that all it does is allowing the empty string as part of the language.

It takes $O(n^{\sqrt{8}})$ time to determine whether a word belongs to a contextfree language, which still is a bit slow. If we further restrict the production rules we can achieve linear time:

**Definition 8** A *regular* language is one generated by a grammar where the productions only can have the following two forms:

$$\begin{aligned} \alpha &\longrightarrow w\beta \\ \alpha &\longrightarrow w \end{aligned}$$

where $\alpha, \beta \in N$ and $w \in T^*$.

Unfortunately, this class is too restrictive to describe the constructs of most programming languages. As an example regular languages only have finite memory. That means we cannot describe arbitrarily deep nesting (for instance balanced parentheses).

But how does one go about determining whether a given language is contextfree or regular?

## 3.1 Characterisation of regular and contextfree languages

**Theorem 9** *Let $L$ be a regular language. There exists $n \in \mathbb{N}$ such that if $z \in L$ with $|z| \geq n$ then $z$ has the form $uvw$, where $|uv| < n$ and $|v| \geq 1$. Furthermore $uv^iw \in L$ for all $i \geq 0$.*

For a proof see [3]. That means that all "long" words in a regular language contains a substring which can be repeated arbitrarily many times. Hence if one wants to show that a language is not regular one assumes that it has the property described in the theorem. Then for some long word $z$ one shows that regardless of the form $z = uvw$ has, $uv^iw$ for some $i$ violates the rules of the grammar. For instance it can be shown that the language $\{a^nb^n \mid n \in \mathbb{N}\}$ over $\{a, b\}$ is not regular in this manner.

**Theorem 10** *Let $L$ be a contextfree language. There exists $n \in \mathbb{N}$ such that if $z \in L$ with $|z| \geq n$ then $z$ has the form $uvwxy$ where $|vwx| \leq n$ and $|vx| \geq 1$. Furthermore $uv^iwx^iy \in L$ for all $i \geq 0$.*

That is, words longer than a certain fixed length contains two parts which can be repeated arbitrarily (but equally) many times. Note that one of the parts may be empty.

The theorem can be used to show that the language $\{ww \mid w \in \{a, b\}^*\}$ is not contextfree.

## 4 Further reading

Chomsky's original account can be found in [2]. Today it is mostly of historical interest. The use of formal languages and automata in compiler design is splendidly covered by the book [1]. The style is informal, but thorough. A more formal account of the issues touched upon in this paper (but without any references to symbolic dynamics) can be found in [3]. The interplay between languages and various models of computation (i.e. the automata) is the main theme of the book.

## References

[1] Alfred V. Aho, Revi Sethi and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley 1986.

[2] Noah Chomsky, *Three Models for the description of Language*, IRE Transactions on Information Theory, 2:3 (1956), pp. 118–124.

[3] J. Hopcroft and J Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley 1979.