Formal Languages and Grammars

1 Formal Languages

Basic concepts of any formal language:

- Alphabet Σ = set of all possible symbols
- Sequence (or word) α = string of symbols
- Language = Particular set of sequences

Further concepts:

- Syntax: which sequences belong to the language
- Semantics: the meaning of these sequences

In the context of programming languages:

- Sequences in the formal language are all valid program codes

If a language L has a finite number of sequences it is described by listing all sequences
If it has an infinite number of sequences it is represented with more complex set notation or with a set of substitution rules

2 Formal Grammars

Definition: Formal Grammar

A finite way of describing an infinite number of strings
Given by a start symbol and a set of production rules

Definition: Grammar

A quadruple (V_T, V_N, P, S) where:

- V_T is a set of terminal symbols (or terminals)
- V_N is a set of non-terminal symbols (or terminals)
- P is a set of productions (or rules)
- S is the start symbol (which is a non-terminal)

$V = V_T \cup V_N$ is the set of all symbols where $V_T \cap V_N = \varnothing$

2.1 Notation

- Non terminals: Capital letters
- Terminals: Lower case letters
- Sequences (strings): Greek letters
- Productions have the form: $\alpha \to \beta$
  - α and β are strings of both terminals and non-terminals
  - α lies in VY^+ (it has at least one non-terminal)
  - β lies in V^* (i.e. can also be the empty string ϵ)
- Whenever we have two rules $\alpha \to \beta_1, \alpha \to \beta_2$ we write equivalently $\alpha \to \beta_1 | \beta_2$

2.2 Example

$$L_4 = \{a^m b^n | m, n \geqslant 0\}$$

Grammar for this language: $G_4 = \{\{a, b\}, \{S, A, B\}, P, S\}$ where P includes:
$S \rightarrow AB$
$A \rightarrow aA$
$A \rightarrow \epsilon$
$B \rightarrow bB$
$B \rightarrow \epsilon$
Or equivalently $S \rightarrow AB$
$A \rightarrow aA|\epsilon$
$B \rightarrow bB|\epsilon$

$$S \rightarrow AB \rightarrow aAB \rightarrow aaAB \rightarrow aaaAB \rightarrow aaaB \rightarrow aaabB \rightarrow aaabbB \rightarrow aaabb$$

Or in short:

$$S \xrightarrow[+]{G_4} aaabb$$

3 Language generated by a grammar

$$L(G) = \{\alpha \in V_T^* | S \xrightarrow[+]{G} \alpha\}$$

A non-terminal N is called

- **Left-recursive** if starting with N, we can produce a string that starts again with N
  - e.g. $N \rightarrow Nab$
- **Nullable** if starting with N, we can produce ϵ
  - e.g. $N \rightarrow aNb|\epsilon 0$
- **Useless** if starting with N, we can never produce a string consisting only of terminals
  - e.g. $N \rightarrow aN|bN$

4 The Chomsky Hierarchy

Definition: Chomsky Hierarchy

The classes of grammars we obtain by restricting the types of productions that can appear

Type 0 grammars - The set of all grammars
Type 1 (or context sensitive) grammars

- The grammars that have only productions of the form $\alpha \rightarrow \beta$, where $|\alpha| \geqslant |\beta|$

Type 2 (or context free) grammars:

- The context sensitive grammars, in which the left part of each production has one non terminal
- e.g. $A \rightarrow BaA$

Type 3 (or regular) grammars

- The context free grammars, in which all productions have one of the forms:

$$A \rightarrow a$$

$$A \rightarrow aB$$

- where B can also be equal to A. They are also called right linear grammars

Equivalent definition of a type 3 (regular) grammars:

- All productions have one of the forms
$$A \to a$$
$$A \to Ba$$
- They are also called left linear grammars

The chomsky hierarchy also classifies the languages generated by these grammars

5 Regular Expressions

Regular expressions over an alphabet Σ (recursive definition):

- If any element of Σ is a regular expression (also the empty string ϵ)
- If P and Q are regular expressions, then the following are as well:
  - PQ, i.e. the concatenation of P and Q
  - $P|Q$ i.e. the disjunction of P and Q (that is: P or Q)
  - P^* i.e. Kleene star of P (zero or more copies of P)

Theorem 1 *Regular grammars generate exactly regular expressions*

Context free and regular languages are very useful in describing programming languages

Regular grammars:

- Many ”local” features of a programming language can be expressed by regular expressions, e.g. constants, strings, identifiers
- A regular expression describing an identifier:
$$L(L'|D)^*$$
  Here:
  - L is the set of all letters
  - $L' = L \cup \{\_\}$ is the set of all letters, including '_'
  - D is the set of all digits

6 Context Free Grammars

- Many syntactic properties of a programming language can be expressed by a context free grammar
- e.g. the pattern of matching brackets of arbitrary length
- This can't be expressed by a regular expression but by this context free grammar
$$S \to (S)$$
$$S \to SS$$
$$S \to \epsilon$$

7 Calling Graph

Many algorithms in compiler construction first collect some basic data items then apply some rules to them to:

- Extend the known information about these data items
- Draw more general conclusions about them

These ”information-improving” algorithms:

- Share a common structure
- Although they seem different

Basic example: the calling graph of a program

- Directed graph with a node for each procedure
- Edge from A to B, whenever A calls

An example of why we need to find the calling graph is to find out which procedures:

- Are recursive
- Can be expanded in-line inside another procedure

8 Regular Definition

Definition: Regular Definition

A sequence of definitions of the form

$$\begin{aligned} d_1 &\to r_1 \\ d_2 &\to r_2 \\ &\dots \\ d_k &\to r_k \end{aligned}$$

Where:

- Each d_i is different and not in Σ
- Each r_i is a regular expression in $\Sigma \cup \{r_1, r_2, ..., r_{i-1}\}$

We avoid recursive definitions by sequentially ”eliminating” all d_i’s, thus if we can write a set of regular expressions as a regular definition, then we have no recursive definitions