Regular Expressions

COMP2600 — Formal Methods for Software Engineering

Katya Lebedeva

Australian National University

Semester 2, 2016

Slides created by Katya Lebedeva

Regular Expressions and Finite State Automata

Regular expressions can define exactly the same languages that finite state automata describe: the regular languages. Regular expressions offer a declarative way to express the strings we want to accept.

That is why many systems that process strings use regular expressions as the input language:

- file search commands (e.g. UNIX `grep`)
- lexical analyzers

these systems convert the regular expression into either a DFA or an NFA, and

- simulate the automaton on the file being searched
- uses the automaton to recognize which token appears next on the input

Example

Consider the expression

$$(0+1)01^*$$

The language described by this expression is the set of all binary strings

- that start with either 0 or 1 as indicated by $(0+1)$,
- for which the second symbol is 0
- that end with zero or more 1s as indicated by 1^*

The language described by this expression is

$$\{00, 001, 0011, 00111, \ldots, 10, 101, 1011, 10111, \ldots\}$$

Definition of a regular expression

Given an alphabet Σ. Regular expressions (RE) over Σ are strings over an alphabet $\Sigma \cup \{+, \cdot, *, (\ ), \varepsilon, \emptyset\}$ defined inductively as follows

1. Base case:
   ε is a RE
   $\emptyset$ is a RE
   for all $a \in \Sigma$, a is a RE

2. Inductive case: if E and F are RE, then
   $E + F$ is a RE — alternation
   $E \cdot F$ is a RE — concatenation
   E^* is a RE — Kleene star
   (E) is a RE

where E^* is the set of all strings that can be made by concatenating any finite number (including zero) of strings from set described by E.

I.e. regular expressions consist of constants that denote sets of strings (base case) and operator symbols that denote operations over these sets (inductive case).

Precedence of operators:

high

$*$

$\cdot$

$+$

low

Definition of a regular language

Language $L(E)$ defined by RE E is also defined inductively:

- $L(\varepsilon) = \varepsilon$
  $L(\emptyset) = \emptyset$
  $L(a) = \{a\}$ for each $a \in \Sigma$

- $L(E + F) = L(E) \cup L(F)$
  $L(E \cdot F) = L(E) \cdot L(F)$
  $L((E)) = L(E)$
  $L(E^*) = (L(E))^*$

Concatenation of L_1 and L_2:

$$L_1 \cdot L_2 = \{w \mid w = x \cdot y, x \in L_1, y \in L_2\}$$

L^* - closure of a language L

We first define the power of L:

- $L^0 = \{\varepsilon\}$
- $L^{n+1} = L^n \cdot L$

Hence

$$L^n = \{w \mid w = x_1 \cdot \dots \cdot x_n, x_i \in L\}$$

Closure L^* of L:

$$L^* = L^0 \cup L^1 \cup \dots$$

Examples

Example 1

$$E = \varepsilon + 1$$
$$L(E) = \{\varepsilon\} \cup \{1\} = \{\varepsilon, 1\}$$

$$F = \varepsilon + 0 + 1$$
$$L(F) = \{\varepsilon, 0, 1\}$$

$$G = (E) \cdot (F) = (\varepsilon + 1) \cdot (\varepsilon + 0 + 1)$$
$$L(G) = \{\varepsilon, 1\} \cdot \{\varepsilon, 0, 1\} = \{\varepsilon, 0, 1, 10, 11\}$$

Example 2

$E = 0 + 1$
$L(E) = \{0, 1\}$
$F = E^* = (0 + 1)^*$
$L(F) = L(E^*) = \{\varepsilon, 0, 1, 01, \dots\}$ set of all binary strings

Example 3

$E = 00$
$L(E) = \{00\}$
$L(E^*) = \{\varepsilon, 00, 0000, 000000, \dots\}$ set of all strings of 0's of even length

Regular expressions specify languages by giving a pattern that the strings must match.

In industry, different notations or even terminologies are used, but the idea behind remains the same. For example:

- $\hat{}[a-z0-9\_-]\{3,16\}\$$ is used to match user names.
- $\hat{}([a-z0-9\_\.-]+)@([a-z0-9\_\.-]+)\.([a-z\.]\{2,6\})\$$ is used to match email addresses.

Compilers

The first two phases of analysing the syntax of (programming) languages, are

1. Lexical analysis (scanning) - converting a sequence of characters into a sequence of tokens (strings with an identified ”meaning”)
2. Syntax analysis (parsing) - checking for correct syntax by converting the input data into a data structure (e.g. parse tree)

Given a string and a RE, a lexical analyser checks whether this string (your code) match the RE.

What do they do?

1. Derive an NFA from the regular expression
2. Convert the NFA to a DFA
3. Minimize the obtained DFA
4. Use the DFA as data structure (for recognising tokens that appear next on the input)

Formal Grammars

COMP2600 — Formal Methods for Software Engineering

Katya Lebedeva

Australian National University

Semester 2, 2016

Slides created by Katya Lebedeva

In the 1950s Noam Chomsky started his study of natural languages. His goal was to formally define syntax of languages.

Chomsky introduced generative grammar. Later it was found that the syntax of programming languages can be described Chomsky's grammatical models (context-free grammars).

Formal Grammars

Informal Definition

Alphabet is a finite, nonempty set of symbols

String (word) is a finite sequence of symbols chosen from some alphabet

Language L over Σ is any subset of Σ^* (i.e. $L \subseteq \Sigma^*$)

A formal grammar is a set of rules by which strings of a language are constructed.

I.e., a grammar is a set of rules that describe a language.

The rules are called production rules.

They describe how to form valid strings from the language's alphabet.

Formal Grammars as Language "Generators"

$$S \to A\,N$$

$$A \to \text{good}$$

$$A \to \text{diligent}$$

$$A \to \text{dedicated}$$

$$N \to \text{student}$$

A formal grammar is a set of rules for rewriting strings, along with a "start symbol" from which rewriting starts. In this way a grammar can be seen as a language generator.

$S \Rightarrow A\,N \Rightarrow \text{good}\,N \Rightarrow \text{good student}$

$S \Rightarrow A\,N \Rightarrow \text{diligent}\,N \Rightarrow \text{diligent student}$

$S \Rightarrow A\,N \Rightarrow \text{dedicated}\,N \Rightarrow \text{dedicated student}$

Formal Grammars as Language "Recognizers"

The process of recognizing a string conforming to the rules of a formal grammar is parsing (also called syntactic analysis).

The string is parsed by breaking it down to symbols and analysing each one against the grammar of the language.

Hence, a grammar allows us to write a computer program (called syntax analyser or parser in a compiler) to determine whether a string is syntactically correct.

dedicated␣student

Unrestricted Grammar

Unrestricted grammar is a formal grammar on which no restrictions are made on the left and right sides of the grammar's productions.

An unrestricted grammar is a quadruple $\langle \Sigma, N, S, P \rangle$ where:

- Σ is a finite set of terminal symbols (the alphabet)
- N is a finite set of nonterminal symbols

$$\Sigma \cap N = \emptyset$$

- S is a distinguished non-terminal symbol called the start symbol, $S \in N$
- P is a finite set of production rules of the form $\alpha \to \beta$, where

$$\alpha \in (\Sigma \cup N)^* N (\Sigma \cup N)^*$$

(notice that there has to be at least 1 nonterminal symbol in α)

$$\beta \in (\Sigma \cup N)^*$$

$(\Sigma \cup N)^*$ - all possible strings over $\Sigma \cup N$

Example

$$G = \langle \{a,b\},\ \{S,A\},\ S,\ \{S \to aAb,\ aA \to aaAb,\ A \to \varepsilon\} \rangle$$

- Terminals: $\{a,b\}$
- Non-terminals: $\{S,A\}$
- Start symbol: S
- Production rules:

$$S \to aAb$$

$$aA \to aaAb$$

$$A \to \varepsilon$$

Conventions

Usually:

- Nonterminal symbols are denoted by capital letters: S, A, B
- Terminal symbols are denoted by lower case letters: a, b, c

And

$$\alpha \to \beta$$

$$\alpha \to \gamma$$

is often abbreviated as

$$\alpha \to \beta \mid \gamma$$

Derivation

Production rules are substitution rules:
if there is a production $\alpha \to \beta$ then we can rewrite any string $\gamma\alpha\rho$ as $\gamma\beta\rho$

This is denoted as $\gamma\alpha\rho \Rightarrow \gamma\beta\rho$

Derivation, $\overset{*}{\Rightarrow}$, is a reflexive transitive closure of $\Rightarrow$:

$\alpha \overset{*}{\Rightarrow} \beta$ (β is derived from α using 0 or more steps)

The language generated by a grammar is the set of strings over Σ – i.e. terminals only – that can be derived from the start symbol:

$$\{\alpha \mid S \overset{*}{\Rightarrow} \alpha \wedge \alpha \in \Sigma^*\}$$

Strings produced by substitutions are sentential forms.
I.e. the sentential forms are $\{\alpha \mid S \overset{*}{\Rightarrow} \alpha \wedge \alpha \in (\Sigma \cup N)^*\}$

$$S \to aAb, \qquad aA \to aaAb, \qquad A \to \varepsilon.$$

A sample derivation:

$$S \Rightarrow aAb \Rightarrow aaAbb \Rightarrow aaaAbbb \Rightarrow aaabbb$$

Each of strings obtained at each derivation step, including the last string, is a sentential form. The last string is also called a sentence.

The language generated by this grammar can also be described as

$$\{a^n b^n \mid n \in \mathbb{N}, n \geq 1\}$$

We can generate this language by a simpler grammar:

$$S \to aSb, \qquad S \to ab.$$

Grammars are not 1-to-1 with languages.

The Chomsky Hierarchy

Chomsky classified grammars on the basis of the *form of their productions*:

Unrestricted: (type 0) no constraints.

Context-sensitive: (type 1) Rules are of the form
$\alpha A \beta \to \alpha \gamma \beta$ and $\gamma \in (N \cup \Sigma)^+$, i.e. γ is not empty.

Context-free: (type 2) the left side of each production must be a *single non-terminal*.

Regular: (type 3) as for type 2, and the right side of each production is also constrained (details to come).

There are many interesting intermediate types of grammar also.

`http://en.wikipedia.org/wiki/Template:Formal_languages_and_grammars`

Classification of Languages

A language is *type* n if it **can** be generated by a type n grammar.

Going down the hierarchy of grammars, there are more restrictions placed on the form of production rules that are permitted.

For example, if there is a type 2 grammar for some language then there are also type 1 and type 0 grammars for that language.

To show that a *language* is type 2 we must provide a type 2 grammar for it.

To show that a *language* is not type 2 we must show that there *cannot* be a type 2 grammar for it.

Example

We have seen two grammars for $\{a^n b^n \mid n \in \mathbb{N},\ n \geq 1\}$:

- Unrestricted (type 0):

$$S \to aAb$$
$$aA \to aaAb$$
$$A \to \varepsilon$$

- Context-free (type 2):

$$S \to ab$$
$$S \to aSb$$

Last week we proved that there is no FSA for this language (and therefore, as we will see, no regular grammar), so the language is not of type 3 and must be of type 2.

Regular Grammars (type 3)

Productions are all of the form (for a right-linear grammar):

$$A \to aB \qquad \text{or} \qquad A \to a \qquad \text{or} \qquad A \to \varepsilon$$

or all of the form (for a left-linear grammar):

$$A \to Ba \qquad \text{or} \qquad A \to a \qquad \text{or} \qquad A \to \varepsilon$$

Their essential feature is that they generate sentences one symbol at a time.

The languages they generate are the **regular languages**

There is no deep difference between right and left linear grammar. We will make the arbitrary choice to stick to **right-linear**.

Regular Languages - Many Views

The following are equivalent:

- L is the language generated by a right-linear grammar
- L is the language generated by a left-linear grammar
- L is the language accepted by some DFA
- L is the language accepted by some NFA
- L is the language specified by a regular expression

We have proven that the equivalent DFA and NFA recognise the same languages.

We will now show that NFA and (right-)linear grammars specify the same languages.

From NFAs to Right-linear Grammars

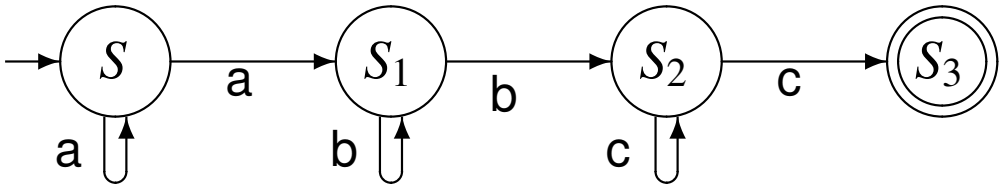
Take an NFA $(\Sigma, S, s_0, F, \delta)$

(alphabet, states, start state, final states, transition function)

Our equivalent right-linear grammar will have

- as terminal symbols symbols of the alphabet Σ
- as non-terminal symbols the states of S
- as start symbol the start state s_0
- as production rules
  - for all $S' \in S$ and $a \in \Sigma$, if $S'' \in \delta(S', a)$, add $S' \to aS''$
  - for each final state $Q \in F$ add $Q \to \varepsilon$

Example



A right-linear grammar accepting the same language:

$$S \to aS$$
$$S \to aS_1$$
$$S_1 \to bS_1$$
$$S_1 \to bS_2$$
$$S_2 \to cS_2$$
$$S_2 \to cS_3$$
$$S_3 \to \varepsilon$$

From Right-linear Grammars to NFAs

Given a right-linear grammar (Σ, N, S, P), the equivalent NFA has

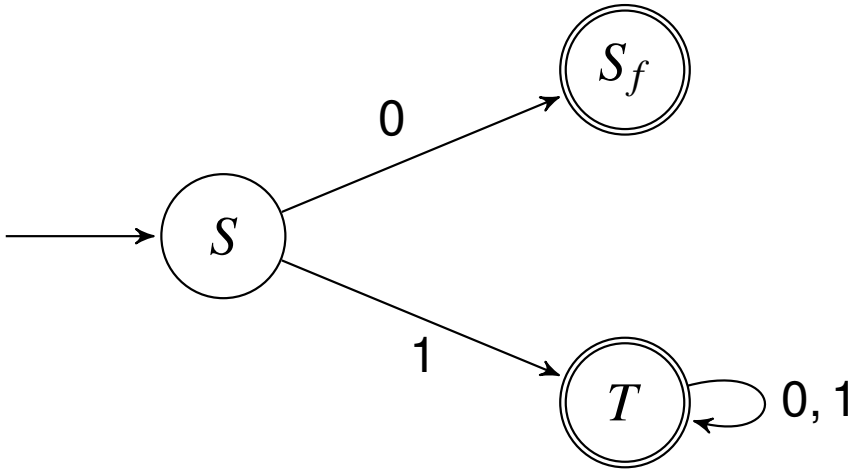
- as alphabet the terminal symbols Σ
- as states the non-terminal symbols N along with a new state S_f (for Final)
- as start state the start symbol S
- as final states S_f and all non-terminals $T \in N$ such that there exists a production $T \to \varepsilon$.
- as transition function
  - for each $T \to aU$ add U to the set defined by $\delta(T, a)$
  - for each $T \to a$ add S_f to the set defined by $\delta(T, a)$

Example

$$S \to 0 \qquad S \to 1T$$

$$T \to \varepsilon \qquad T \to 0T \qquad T \to 1T$$

generates the language of binary integers, and the automaton



Context-Free Grammars (type 2)

Productions are all of the form: $A \to \omega$ where $A \in N$ and $\omega \in (\Sigma \cup N)^*$.

- the left hand side of each production must be one non-terminal on its own (the same as in regular grammars)
- the right side of a production is a string of zero or more non-terminal and terminal symbols

Therefore A can be replaced by ω independently of their context.

Hence the name "context free".

This contrasts with context-**sensitive** grammars which may have productions of the form $\alpha A \beta \to \alpha \omega \beta$, where A may be replaced by ω only in the context α_β.

Example

Design a CFG for the language $\{a^m b^n c^{m-n} \mid m \geq n \geq 0\}$?

Strategy: Split the words in this language into sections:

- a^{m-n}, followed by
- $a^n b^n$, followed by
- c^{m-n}

Use different nonterminals for generating the first and third substrings (simultaneously) and for generating the second substring:

$$S \rightarrow aSc \mid T$$
$$T \rightarrow aTb \mid \varepsilon$$

$$S \rightarrow aSc \mid T$$
$$T \rightarrow aTb \mid \varepsilon$$

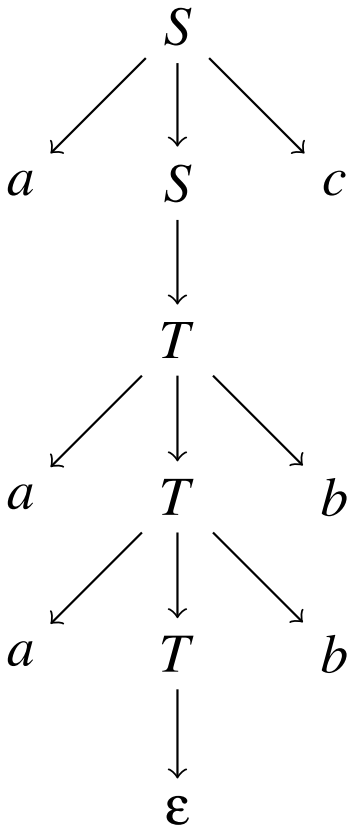
An example derivation of the word $aaabbc$:

$$\begin{aligned} S &\Rightarrow aSc \\ &\Rightarrow aTc \\ &\Rightarrow aaTbc \\ &\Rightarrow aaaTbbc \\ &\Rightarrow aaabbc \end{aligned}$$

Parse trees

$$S \rightarrow aSc \mid T$$
$$T \rightarrow aTb \mid \varepsilon$$

$$\begin{aligned} S &\Rightarrow aSc \\ &\Rightarrow aTc \\ &\Rightarrow aaTbc \\ &\Rightarrow aaaTbbc \\ &\Rightarrow aaabbc \end{aligned}$$



A parse tree is a tree whose nodes are labeled with $N \cup \Sigma \cup \{\varepsilon\}$ such that

- each interior node is labeled with an element of N
- each leaf is labeled with a terminal, non-terminal or ε
- if an interior node is labeled with A and its children from left to right are labeled with $X_1, \ldots, X_k$ then there is a production $A \to X_1 \ldots X_k$
- the root of the tree is labeled with S

The Power of Context-Free Grammars

A fun example:

`http://pdos.csail.mit.edu/scigen`