



2. Lexical Analysis

Andrea Polini

Formal Languages and Compilers
MSc in Computer Science
University of Camerino

ToC

- 1 Lexical Analysis: What we wanna do?
- 2 Short Notes on Formal Languages
- 3 Lexical Analysis: How can we do it?
 - Regular Expressions
 - Finite State Automata

Lexical Analysis

```
if (i==j)
    z=0;
else
    z=1;
```

```
\tif (i==j)\n\t\tz=0;\n\telse\n\t\tz=1;
```

Lexical Analysis

```
if (i==j)
    z=0;
else
    z=1;
```

```
\tif (i==j)\n\t\tz=0;\n\telse\n\t\tz=1;
```

Token, Pattern Lexeme

Token

A **token** is a pair consisting of a token name and an optional attribute value. The token names are the input symbols that the parser processes.

Pattern

A **pattern** is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.

Lexeme

A **lexeme** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Lexical Analysis

- **Token Class (or Class)**

- In English: *Noun, Verb, Adjective, Adverb, Article, ...*
- In a programming language: *Identifier, Keywords, “(, “)”, Numbers, ...*

Lexical Analysis

- Token classes corresponds to sets of strings
- Identifier
 - strings of letter or digits starting with a letter
- Integer
 - a non-empty string of digits
- Keyword
 - "else", "if", "while", ...
- Whitespace
 - a non-empty sequence of blanks, newlines, and tabs

Lexical Analysis

- Token classes corresponds to sets of strings
- Identifier
 - strings of letter or digits starting with a letter
- Integer
 - a non-empty string of digits
- Keyword
 - “else”, “if”, “while”, ...
- Whitespace
 - a non-empty sequence of blanks, newlines, and tabs

Lexical Analysis

- Token classes corresponds to sets of strings
- Identifier
 - strings of letter or digits starting with a letter
- Integer
 - a non-empty string of digits
- Keyword
 - “else”, “if”, “while”, ...
- Whitespace
 - a non-empty sequence of blanks, newlines, and tabs

Lexical Analysis

- Token classes corresponds to sets of strings
- Identifier
 - strings of letter or digits starting with a letter
- Integer
 - a non-empty string of digits
- Keyword
 - “else”, “if”, “while”, . . .
- Whitespace
 - a non-empty sequence of blanks, newlines, and tabs

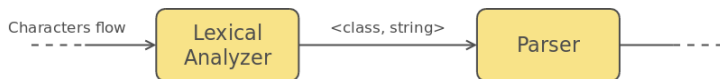
Lexical Analysis

- Token classes corresponds to sets of strings
- Identifier
 - strings of letter or digits starting with a letter
- Integer
 - a non-empty string of digits
- Keyword
 - “else”, “if”, “while”, . . .
- Whitespace
 - a non-empty sequence of blanks, newlines, and tabs

Lexical Analysis

Therefore the role of the lexical analyzer (Lexer) is:

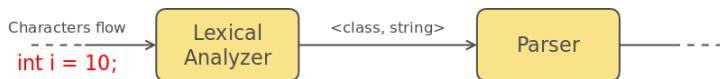
- Classify program substring according to role (token class)
- communicate tokens to parser



Lexical Analysis

Therefore the role of the lexical analyzer (Lexer) is:

- Classify program substring according to role (token class)
- communicate tokens to parser



Lexical Analysis

Let's analyze these lines of code:

```
\tif (i==j)\n\t\t\tz=0;\n\telse\n\t\t\tz=1;
```

```
x=0;\n\twhile (x<10) {\n\t\t\tx++;\n\t}
```

Token Classes: Identifier, Integer, Keyword, Whitespace

Lexical Analysis

Therefore an implementation of a lexical analyzer must do two things:

- Recognize substrings corresponding to tokens
 - the lexemes
- Identify the token class for each lexemes

Lexical Analysis - Tricky problems

- FORTRAN rule: whitespace is insignificant
 - i.e. `VA R1` is the same as `VAR1`

```
DO 5 I = 1,25
```

```
DO 5 I = 1.25
```

In FORTRAN the "5" refers to a label you will find in the following of the program code

Lexical Analysis - Tricky problems

- 1 The goal is to partition the string. This is implemented by reading left-to-right, recognizing one token at a time
- 2 “Lookahead” may be required to decide where one token ends and the next token begins

```
if (i==j)
    z=0;
else
    z=1;
```

Lexical Analysis - Tricky problems

- PL/1 keywords are not reserved

```
IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN
```

```
DECLARE (ARG1, . . . , ARGN)
```

Is `DECLARE` a keyword or an array reference?

Need for an unbounded lookahead

Lexical Analysis - Tricky problems

- PL/1 keywords are not reserved

```
IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN
```

```
DECLARE (ARG1, ..., ARGN)
```

Is `DECLARE` a keyword or an array reference?

Need for an unbounded lookahead

Lexical Analysis - Tricky problems

- PL/1 keywords are not reserved

```
IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN
```

```
DECLARE (ARG1, . . . , ARGN)
```

Is `DECLARE` a keyword or an array reference?

Need for an unbounded lookahead

Lexical Analysis - Tricky problems

- C++ template syntax:

```
Foo<Bar>
```

- C++ stream syntax:

```
cin >> var;
```

```
Foo<Bar<Barr>>
```

Lexical Analysis - Tricky problems

- C++ template syntax:

```
Foo<Bar>
```

- C++ stream syntax:

```
cin >> var;
```

```
Foo<Bar<Barr>>
```

ToC

- 1 Lexical Analysis: What we wanna do?
- 2 Short Notes on Formal Languages**
- 3 Lexical Analysis: How can we do it?
 - Regular Expressions
 - Finite State Automata

Languages

Language

Let Σ be a set of characters generally referred as the *alphabet*. A **language** over Σ is a set of strings of characters drawn from Σ

Alphabet = English character \implies Language = English sentences
 Alphabet = ASCII \implies Language = C programs

Given $\Sigma = \{a, b\}$ examples of simple languages are:

- $\mathcal{L}_1 = \{a, ab, aa\}$
- $\mathcal{L}_2 = \{b, ab, aabb\}$
- $\mathcal{L}_3 = \{s \mid s \text{ has an equal number of } a \text{ and } b\}$
- ...

Grammar Definition

Grammar

A **Grammar** is given by a tuple $\mathcal{G} = \langle \mathcal{V}_T, \mathcal{V}_N, \mathcal{S}, \mathcal{P} \rangle$ where:

- ▶ \mathcal{V}_T : finite and non empty set of terminal symbols (alphabet)
- ▶ \mathcal{V}_N : finite set of non terminal symbols s.t. $\mathcal{V}_N \cap \mathcal{V}_T = \emptyset$
- ▶ \mathcal{S} : start symbol of the grammar s.t. $\mathcal{S} \in \mathcal{V}_N$
- ▶ \mathcal{P} : is the set of productions s.t. $\mathcal{P} \subseteq (\mathcal{V}^* \cdot \mathcal{V}_N \cdot \mathcal{V}^*) \times \mathcal{V}^*$ where $\mathcal{V}^* = \mathcal{V}_T \cup \mathcal{V}_N$

Derivations

Derivations

Given a grammar $\mathcal{G} = \langle \mathcal{V}_T, \mathcal{V}_N, \mathcal{S}, \mathcal{P} \rangle$ a derivation is a sequence of strings $\phi_1, \phi_2, \dots, \phi_n$ s.t.

$\forall i \in [1, \dots, n]. \phi_i \in \mathcal{V}^* \wedge \forall i \in [1, \dots, n-1]. \exists p \in \mathcal{P}. \phi_i \rightarrow^p \phi_{i+1}$.

We generally write $\phi_1 \rightarrow^* \phi_n$ to indicate that from ϕ_1 it is possible to derive ϕ_n repeatedly applying productions in \mathcal{P}

Generated Language

The language generated by a grammar $\mathcal{G} = \langle \mathcal{V}_T, \mathcal{V}_N, \mathcal{S}, \mathcal{P} \rangle$ corresponds to: $\mathcal{L}(\mathcal{G}) = \{x \mid x \in \mathcal{V}_T^* \wedge \mathcal{S} \rightarrow^* x\}$

Derivations

Derivations

Given a grammar $\mathcal{G} = \langle \mathcal{V}_T, \mathcal{V}_N, \mathcal{S}, \mathcal{P} \rangle$ a derivation is a sequence of strings $\phi_1, \phi_2, \dots, \phi_n$ s.t.

$\forall i \in [1, \dots, n]. \phi_i \in \mathcal{V}^* \wedge \forall i \in [1, \dots, n-1]. \exists p \in \mathcal{P}. \phi_i \rightarrow^p \phi_{i+1}$.

We generally write $\phi_1 \rightarrow^* \phi_n$ to indicate that from ϕ_1 it is possible to derive ϕ_n repeatedly applying productions in \mathcal{P}

Generated Language

The language generated by a grammar $\mathcal{G} = \langle \mathcal{V}_T, \mathcal{V}_N, \mathcal{S}, \mathcal{P} \rangle$ corresponds to: $\mathcal{L}(\mathcal{G}) = \{x | x \in \mathcal{V}_T^* \wedge \mathcal{S} \rightarrow^* x\}$

Chomsky Hierarchy

A hierarchy of grammars can be defined imposing constraints on the structure of the productions in set \mathcal{P} ($\alpha, \beta, \gamma \in \mathcal{V}^*$, $a \in \mathcal{V}_T$, $A, B \in \mathcal{V}_N$):

T0. Unrestricted Grammars:

- Production Schema: *no constraints*
- Recognizing Automaton: **Turing Machines**

T1. Context Sensitive Grammars:

- Production Schema: $\alpha A \beta \rightarrow \alpha \gamma \beta$
- Recognizing Automaton: **Linear Bound Automaton (LBA)**

T2. Context-Free Grammars:

- Production Schema: $A \rightarrow \gamma$
- Recognizing Automaton: **Non-deterministic Push-down Automaton**

T3. Regular Grammars:

- Production Schema: $A \rightarrow a$ or $A \rightarrow aB$
- Recognizing Automaton: **Finite State Automaton**

Meaning function \mathcal{L}

Meaning Function

Once you defined a way to describe the strings in a language it is important to define a meaning function \mathcal{L} that maps syntax to semantics

- Why using a meaning function?
 - Makes clear what is syntax, what is semantics
 - Allows us to consider notation as a separate issue
 - Because expressions and meanings are not 1 to 1
 - consider the case of arabic number and roman numbers

Warning

It should never happen that the same syntactical structure has more meanings

Meaning function \mathcal{L}

Meaning Function

Once you defined a way to describe the strings in a language it is important to define a meaning function \mathcal{L} that maps syntax to semantics

- Why using a meaning function?
 - Makes clear what is syntax, what is semantics
 - Allows us to consider notation as a separate issue
 - Because expressions and meanings are not 1 to 1
 - consider the case of arabic number and roman numbers

Warning

It should never happen that the same syntactical structure has more meanings

Meaning function \mathcal{L}

Meaning Function

Once you defined a way to describe the strings in a language it is important to define a meaning function \mathcal{L} that maps syntax to semantics

- Why using a meaning function?
 - Makes clear what is syntax, what is semantics
 - Allows us to consider notation as a separate issue
 - Because expressions and meanings are not 1 to 1
 - consider the case of arabic number and roman numbers

Warning

It should never happen that the same syntactical structure has more meanings

ToC

- 1 Lexical Analysis: What we wanna do?
- 2 Short Notes on Formal Languages
- 3 Lexical Analysis: How can we do it?**
 - Regular Expressions
 - Finite State Automata

Languages

We need to define which is the set of strings in any token class. Therefore we need to choose the right mechanisms to describe such sets:

- Reducing at minimum the complexity needed to recognize lexemes
 - Identifying effective and simple ways to describe the patterns
-
- Regular languages seem to be enough powerful to define all the lexemes in any token class
 - Regular expressions are a suitable way to syntactically identify strings belonging to a regular language

Regular expressions

- Single character: $'c' = \{“c”\}$
- Epsilon: $\epsilon = \{“ ”\}$
- Union: $A+B = \{a|a \in A\} \cup \{b|b \in B\}$
- Concatenation: $AB = \{ab|a \in A \wedge b \in B\}$
- Iteration: $A^* = \cup_{i \geq 0} A^i$

The regular expressions over Σ are the smallest set including ϵ , all the character 'c' in Σ and that is closed with respect to union, concatenation and iteration.

- Algebraic laws for RE:
 - + is commutative and associative
 - concatenation is associative
 - concatenation distributes over +
 - ϵ is the identity for concatenation
 - ϵ is guaranteed in a closure
 - the Kleene star is idempotent

Regular expressions

- Single character: $'c' = \{“c”\}$
- Epsilon: $\epsilon = \{“ ”\}$
- Union: $A+B = \{a|a \in A\} \cup \{b|b \in B\}$
- Concatenation: $AB = \{ab|a \in A \wedge b \in B\}$
- Iteration: $A^* = \cup_{i \geq 0} A^i$

The regular expressions over Σ are the smallest set including ϵ , all the character 'c' in Σ and that is closed with respect to union, concatenation and iteration.

- Algebraic laws for RE:
 - + is commutative and associative
 - concatenation is associative
 - concatenation distributes over +
 - ϵ is the identity for concatenation
 - ϵ is guaranteed in a closure
 - the Kleene star is idempotent

Regular expressions

- Single character: $'c' = \{“c”\}$
- Epsilon: $\epsilon = \{“ ”\}$
- Union: $A+B = \{a|a \in A\} \cup \{b|b \in B\}$
- Concatenation: $AB = \{ab|a \in A \wedge b \in B\}$
- Iteration: $A^* = \cup_{i \geq 0} A^i$

The regular expressions over Σ are the smallest set including ϵ , all the character 'c' in Σ and that is closed with respect to union, concatenation and iteration.

- Algebraic laws for RE:
 - + is commutative and associative
 - concatenation is associative
 - concatenation distributes over +
 - ϵ is the identity for concatenation
 - ϵ is guaranteed in a closure
 - the Kleene star is idempotent

Exercise

Consider $\Sigma = \{0, 1\}$. What are the sets defined by the following REs?

- ▶ 1^*
- ▶ $(1 + 0)1$
- ▶ $0^* + 1^*$
- ▶ $(0 + 1)^*$

Exercise

Given the regular language identified by $(0 + 1)^*1(0 + 1)^*$ which are the regular expressions identifying the same language among the following one:

- ▶ $(01 + 11)^*(0 + 1)^*$
- ▶ $(0 + 1)^*(10 + 11 + 1)(0 + 1)^*$
- ▶ $(1 + 0)^*1(1 + 0)^*$
- ▶ $(0 + 1)^*(0 + 1)(0 + 1)^*$

Exercise

Consider $\Sigma = \{0, 1\}$. What are the sets defined by the following REs?

- ▶ 1^*
- ▶ $(1 + 0)1$
- ▶ $0^* + 1^*$
- ▶ $(0 + 1)^*$

Exercise

Given the regular language identified by $(0 + 1)^*1(0 + 1)^*$ which are the regular expressions identifying the same language among the following one:

- ▶ $(01 + 11)^*(0 + 1)^*$
- ▶ $(0 + 1)^*(10 + 11 + 1)(0 + 1)^*$
- ▶ $(1 + 0)^*1(1 + 0)^*$
- ▶ $(0 + 1)^*(0 + 1)(0 + 1)^*$

Exercise

Choose the regular languages that are correct specifications of the following English-language description:

Twelve-hour times of the form "04:13PM". Minutes should always be a two digit number, but hours may be a single digit

- ▶ $(0 + 1)?[0 - 9] : [0 - 5][0 - 9](AM + PM)$
- ▶ $((0 + \epsilon)[0 - 9] + 1[0 - 2]) : [0 - 5][0 - 9](AM + PM)$
- ▶ $(0^*[0 - 9] + 1[0 - 2]) : [0 - 5][0 - 9](AM + PM)$
- ▶ $(0?[0 - 9] + 1(0 + 1 + 2)) : [0 - 5][0 - 9](a + P)M$

Regular expressions (syntax)
specify regular languages (semantics)

Exercise

Choose the regular languages that are correct specifications of the following English-language description:

Twelve-hour times of the form "04:13PM". Minutes should always be a two digit number, but hours may be a single digit

- ▶ $(0 + 1)?[0 - 9] : [0 - 5][0 - 9](AM + PM)$
- ▶ $((0 + \epsilon)[0 - 9] + 1[0 - 2]) : [0 - 5][0 - 9](AM + PM)$
- ▶ $(0^*[0 - 9] + 1[0 - 2]) : [0 - 5][0 - 9](AM + PM)$
- ▶ $(0?[0 - 9] + 1(0 + 1 + 2)) : [0 - 5][0 - 9](a + P)M$

Regular expressions (syntax)
specify regular languages (semantics)

Meaning function \mathcal{L}

- The meaning function L maps syntax to semantics

$\mathcal{L}(e) = \mathcal{M}$ where e is a RE and \mathcal{M} is a set of strings

Therefore:

- $\mathcal{L}(\epsilon) = \{“ ”\}$
- $\mathcal{L}('c') = \{“c”\}$
- $\mathcal{L}(A + B) = \mathcal{L}(A) \cup \mathcal{L}(B)$
- $\mathcal{L}(AB) = \{ab \mid a \in \mathcal{L}(A) \wedge b \in \mathcal{L}(B)\}$
- $\mathcal{L}(A^*) = \{\cup_{i \geq 0} \mathcal{L}(A^i)\}$

Regular definitions

For notational convention we give names to certain regular expressions. A regular definition, on the alphabet Σ is sequence of definition of the form:

- $d_1 \rightarrow r_1$
- $d_2 \rightarrow r_2$
- ...
- $d_n \rightarrow r_n$

So token of a language can be defined as:

- *letter* $\rightarrow a|b|\dots|z|A|B|\dots|Z$
 - compact syntax: $[a - zA - B]$
- *digit* $\rightarrow 0|1|\dots|9$
 - compact syntax: $[0 - 9]$
- *Identifier* $\rightarrow \textit{letter}(\textit{letter}|\textit{digit})^*$
- *ExpNot* $\rightarrow \textit{digit}(\textit{digit}^+ E(+|-)\textit{digit}^+)?$ (*Exponential Notation*)

Lexical Specification

- At least one: $A^+ \equiv AA^*$
- Union: $A|B \equiv A + B$
- Option: $A? \equiv A + \epsilon$
- Range: $'a' + 'b' + \dots + 'z' \equiv [a - z]$
- Excluded range: **complement of** $[a - z] \equiv [^a - z]$

Lexical Specification

We want to derive a regular expression for all tokens of a language:

$s \in \mathcal{L}(R)$ – where R is the RegExp resulting from the sum of the
RegExp for all the different kinds of token

How can we define it?

- write a regexp for the lexemes of each token class (number, keyword, identifier, ...)

Lexical Specification

We want to derive a regular expression for all tokens of a language:

$s \in \mathcal{L}(R)$ – where R is the RegExp resulting from the sum of the
RegExp for all the different kinds of token

How can we define it?

- 1 write a regexp for the lexemes of each token class (number, keyword, identifier, . . .)
- 2 Constructs R matching all lexemes for all tokens
- 3 Let input be $x_1 \dots x_n$
For $1 \leq i \leq n$ check if $x_1 \dots x_i \in \mathcal{L}(R_j)$ for some j
- 4 if success then we know that $x_1 \dots x_i \in \mathcal{L}(R_j)$ for some j
- 5 remove $x_1 \dots x_i$ from input and go to (3)

Lexical Specification

We want to derive a regular expression for all tokens of a language:

$s \in \mathcal{L}(R)$ – where R is the RegExp resulting from the sum of the
RegExp for all the different kinds of token

How can we define it?

- 1 write a regexp for the lexemes of each token class (number, keyword, identifier, . . .)
- 2 Constructs R matching all lexemes for all tokens
- 3 Let input be $x_1 \dots x_n$
For $1 \leq i \leq n$ check if $x_1 \dots x_i \in \mathcal{L}(R_j)$ for some j
- 4 if success then we know that $x_1 \dots x_i \in \mathcal{L}(R_j)$ for some j
- 5 remove $x_1 \dots x_i$ from input and go to (3)

Lexical Specification

We want to derive a regular expression for all tokens of a language:

$s \in \mathcal{L}(R)$ – where R is the RegExp resulting from the sum of the
RegExp for all the different kinds of token

How can we define it?

- 1 write a regexp for the lexemes of each token class (number, keyword, identifier, . . .)
- 2 Constructs R matching all lexemes for all tokens
- 3 Let input be $x_1 \dots x_n$
For $1 \leq i \leq n$ check if $x_1 \dots x_i \in \mathcal{L}(R_j)$ for some j
- 4 if success then we know that $x_1 \dots x_i \in \mathcal{L}(R_j)$ for some j
- 5 remove $x_1 \dots x_i$ from input and go to (3)

Lexical Specification

We want to derive a regular expression for all tokens of a language:

$s \in \mathcal{L}(R)$ – where R is the RegExp resulting from the sum of the
 RegExp for all the different kinds of token

How can we define it?

- 1 write a regexp for the lexemes of each token class (number, keyword, identifier, . . .)
- 2 Constructs R matching all lexemes for all tokens
- 3 Let input be $x_1 \dots x_n$
 For $1 \leq i \leq n$ check if $x_1 \dots x_i \in \mathcal{L}(R_j)$ for some j
- 4 if success then we know that $x_1 \dots x_i \in \mathcal{L}(R_j)$ for some j
- 5 remove $x_1 \dots x_i$ from input and go to (3)

Lexical Specification

We want to derive a regular expression for all tokens of a language:

$s \in \mathcal{L}(R)$ – where R is the RegExp resulting from the sum of the
 RegExp for all the different kinds of token

How can we define it?

- 1 write a regexp for the lexemes of each token class (number, keyword, identifier, . . .)
- 2 Constructs R matching all lexemes for all tokens
- 3 Let input be $x_1 \dots x_n$
 For $1 \leq i \leq n$ check if $x_1 \dots x_i \in \mathcal{L}(R_j)$ for some j
- 4 if success then we know that $x_1 \dots x_i \in \mathcal{L}(R_j)$ for some j
- 5 remove $x_1 \dots x_i$ from input and go to (3)

Lexical Specification

We want to derive a regular expression for all tokens of a language:

$s \in \mathcal{L}(R)$ – where R is the RegExp resulting from the sum of the
RegExp for all the different kinds of token

How can we define it?

- 1 write a regexp for the lexemes of each token class (number, keyword, identifier, . . .)
- 2 Constructs R matching all lexemes for all tokens
- 3 Let input be $x_1 \dots x_n$
For $1 \leq i \leq n$ check if $x_1 \dots x_i \in \mathcal{L}(R_j)$ for some j
- 4 if success then we know that $x_1 \dots x_i \in \mathcal{L}(R_j)$ for some j
- 5 remove $x_1 \dots x_i$ from input and go to (3)

LA matching rules

Suppose that at the same time for $i \neq j$:

- $x_1 \dots x_j \in \mathcal{L}(R)$
- $x_1 \dots x_i \in \mathcal{L}(R)$

Which is the match to consider?

longest match rule

Suppose that at the same time for $i \neq j \in [1..n]$ and $R = R_1 | R_2 | \dots | R_n$:

- $x_1 \dots x_k \in \mathcal{L}(R_i)$
- $x_1 \dots x_k \in \mathcal{L}(R_j)$

Which is the match to consider?

first one listed rule

Errors: to manage errors put as last match in the list a rexp for all lexemes not in the language

LA matching rules

Suppose that at the same time for $i \neq j$:

- $x_1 \dots x_j \in \mathcal{L}(R)$
- $x_1 \dots x_i \in \mathcal{L}(R)$

Which is the match to consider?

longest match rule

Suppose that at the same time for $i \neq j \in [1..n]$ and $R = R_1 | R_2 | \dots | R_n$:

- $x_1 \dots x_k \in \mathcal{L}(R_i)$
- $x_1 \dots x_k \in \mathcal{L}(R_j)$

Which is the match to consider?

first one listed rule

Errors: to manage errors put as last match in the list a rexp for all lexemes not in the language

LA matching rules

Suppose that at the same time for $i \neq j$:

- $x_1 \dots x_j \in \mathcal{L}(R)$
- $x_1 \dots x_i \in \mathcal{L}(R)$

Which is the match to consider?

longest match rule

Suppose that at the same time for $i \neq j \in [1..n]$ and $R = R_1 | R_2 | \dots | R_n$:

- $x_1 \dots x_k \in \mathcal{L}(R_i)$
- $x_1 \dots x_k \in \mathcal{L}(R_j)$

Which is the match to consider?

first one listed rule

Errors: to manage errors put as last match in the list a rexp for all lexemes not in the language

LA matching rules

Suppose that at the same time for $i \neq j$:

- $x_1 \dots x_j \in \mathcal{L}(R)$
- $x_1 \dots x_i \in \mathcal{L}(R)$

Which is the match to consider?

longest match rule

Suppose that at the same time for $i \neq j \in [1..n]$ and $R = R_1 | R_2 | \dots | R_n$:

- $x_1 \dots x_k \in \mathcal{L}(R_i)$
- $x_1 \dots x_k \in \mathcal{L}(R_j)$

Which is the match to consider?

first one listed rule

Errors: to manage errors put as last match in the list a rexp for all lexemes not in the language

LA matching rules

Suppose that at the same time for $i \neq j$:

- $x_1 \dots x_j \in \mathcal{L}(R)$
- $x_1 \dots x_i \in \mathcal{L}(R)$

Which is the match to consider?

longest match rule

Suppose that at the same time for $i \neq j \in [1..n]$ and $R = R_1 | R_2 | \dots | R_n$:

- $x_1 \dots x_k \in \mathcal{L}(R_i)$
- $x_1 \dots x_k \in \mathcal{L}(R_j)$

Which is the match to consider?

first one listed rule

Errors: to manage errors put as last match in the list a rexp for all lexemes not in the language

LA matching rules

Suppose that at the same time for $i \neq j$:

- $x_1 \dots x_j \in \mathcal{L}(R)$
- $x_1 \dots x_i \in \mathcal{L}(R)$

Which is the match to consider?

longest match rule

Suppose that at the same time for $i \neq j \in [1..n]$ and $R = R_1 | R_2 | \dots | R_n$:

- $x_1 \dots x_k \in \mathcal{L}(R_i)$
- $x_1 \dots x_k \in \mathcal{L}(R_j)$

Which is the match to consider?

first one listed rule

Errors: to manage errors put as last match in the list a rexp for all lexemes not in the language

LA matching rules

Suppose that at the same time for $i \neq j$:

- $x_1 \dots x_j \in \mathcal{L}(R)$
- $x_1 \dots x_i \in \mathcal{L}(R)$

Which is the match to consider?

longest match rule

Suppose that at the same time for $i \neq j \in [1..n]$ and $R = R_1 | R_2 | \dots | R_n$:

- $x_1 \dots x_k \in \mathcal{L}(R_i)$
- $x_1 \dots x_k \in \mathcal{L}(R_j)$

Which is the match to consider?

first one listed rule

Errors: to manage errors put as last match in the list a rexp for all lexemes not in the language

Finite Automata

- Regular Expressions = specification
- Finite Automata = implementation

Finite Automaton

A Finite Automaton \mathcal{A} is a tuple $\langle \mathcal{S}, \Sigma, \delta, s_0, \mathcal{F} \rangle$ where:

- ▶ \mathcal{S} represents the set of states
- ▶ Σ represents a set of symbols (alphabet)
- ▶ δ represents the transition function ($\delta : \mathcal{S} \times \Sigma \rightarrow \dots$)
- ▶ s_0 represents the start state ($s_0 \in \mathcal{S}$)
- ▶ \mathcal{F} represents the set of accepting states ($\mathcal{F} \subseteq \mathcal{S}$)

In two flavors: Deterministic Finite Automata (DFA) and Non-Deterministic Finite Automata (NFA)

Finite Automata

- Regular Expressions = specification
- Finite Automata = implementation

Finite Automaton

A Finite Automaton \mathcal{A} is a tuple $\langle \mathcal{S}, \Sigma, \delta, s_0, \mathcal{F} \rangle$ where:

- ▶ \mathcal{S} represents the set of states
- ▶ Σ represents a set of symbols (alphabet)
- ▶ δ represents the transition function ($\delta : \mathcal{S} \times \Sigma \rightarrow \dots$)
- ▶ s_0 represents the start state ($s_0 \in \mathcal{S}$)
- ▶ \mathcal{F} represents the set of accepting states ($\mathcal{F} \subseteq \mathcal{S}$)

In two flavors: **Deterministic Finite Automata (DFA)** and **Non-Deterministic Finite Automata (NFA)**

Acceptance of Strings for Finite Automaton

Derivations

A DFA goes from state s_i to state s_{i+1} consuming from the input the character a if $s_{i+1} = \delta(s_i, a)$. A DFA can go from state s_i to s_j consuming the string $a = a_1 a_2 \dots a_n$ if there is a sequence of states $s_{i+1}, \dots, s_{i+n-1}$ and $s_j = s_{i+n}$ s.t.

$\forall k \in [1..n]. s_{i+k} = \delta(s_{i+k-1}, a_k)$, then we write $s_i \xrightarrow{a} s_j$

Equivalently the **extended transition function** $\bar{\delta} : \mathcal{S} \times \Sigma^* \rightarrow \mathcal{S}$ is defined, i.e.

$$\delta(\delta(\dots\delta(s_i, a_1)\dots, a_{n-1}), a_n) = \bar{\delta}(s_i, a) = s_j$$

Acceptance of Strings

A DFA accepts a strings a in the alphabet Σ if there is a derivation from s_0 to a state s_i consuming the string a (i.e. $s_0 \xrightarrow{a} s_i$) and $s_i \in \mathcal{F}$

Accepted Language

The language accepted by a FSA is constituted by all the strings for which there is a derivation ending in a state in \mathcal{F} .

Acceptance of Strings for Finite Automaton

Derivations

A DFA goes from state s_i to state s_{i+1} consuming from the input the character a if $s_{i+1} = \delta(s_i, a)$. A DFA can go from state s_i to s_j consuming the string $a = a_1 a_2 \dots a_n$ if there is a sequence of states $s_{i+1}, \dots, s_{i+n-1}$ and $s_j = s_{i+n}$ s.t.

$\forall k \in [1..n]. s_{i+k} = \delta(s_{i+k-1}, a_k)$, then we write $s_i \xrightarrow{a} s_j$

Equivalently the **extended transition function** $\bar{\delta} : \mathcal{S} \times \Sigma^* \rightarrow \mathcal{S}$ is defined, i.e.

$\delta(\delta(\dots\delta(s_i, a_1)\dots, a_{n-1}), a_n) = \bar{\delta}(s_i, a) = s_j$

Acceptance of Strings

A DFA accepts a strings a in the alphabet Σ if there is a derivation from s_0 to a state s_i consuming the string a (i.e. $s_0 \xrightarrow{a} s_i$) and $s_i \in \mathcal{F}$

Accepted Language

The language accepted by a FSA is constituted by all the strings for which there is a derivation ending in a state in \mathcal{F} .

Acceptance of Strings for Finite Automaton

Derivations

A DFA goes from state s_i to state s_{i+1} consuming from the input the character a if $s_{i+1} = \delta(s_i, a)$. A DFA can go from state s_i to s_j consuming the string $a = a_1 a_2 \dots a_n$ if there is a sequence of states $s_{i+1}, \dots, s_{i+n-1}$ and $s_j = s_{i+n}$ s.t.

$\forall k \in [1..n]. s_{i+k} = \delta(s_{i+k-1}, a_k)$, then we write $s_i \xrightarrow{a} s_j$

Equivalently the **extended transition function** $\bar{\delta} : \mathcal{S} \times \Sigma^* \rightarrow \mathcal{S}$ is defined, i.e.

$\delta(\delta(\dots\delta(s_i, a_1)\dots, a_{n-1}), a_n) = \bar{\delta}(s_i, a) = s_j$

Acceptance of Strings

A DFA accepts a strings a in the alphabet Σ if there is a derivation from s_0 to a state s_i consuming the string a (i.e. $s_0 \xrightarrow{a} s_i$) and $s_i \in \mathcal{F}$

Accepted Language

The language accepted by a FSA is constituted by all the strings for which there is a derivation ending in a state in \mathcal{F} .

Finite Automata

DFA vs. NFA

Depending on the definition of δ we distinguish between:

- ▶ **Deterministic** Finite Automata (**DFA**) - $\delta : \mathcal{S} \times \Sigma \rightarrow \mathcal{S}$
- ▶ **Nondeterministic** Finite Automata (**NFA**) $\delta : \mathcal{S} \times \Sigma \rightarrow \mathcal{P}(\mathcal{S})$

The transition relation δ can be represented in a table (transition table)

Overview of the graphical notation circle and edges (arrows)

Finite Automata

DFA vs. NFA

Depending on the definition of δ we distinguish between:

- ▶ **Deterministic** Finite Automata (**DFA**) - $\delta : \mathcal{S} \times \Sigma \rightarrow \mathcal{S}$
- ▶ **Nondeterministic** Finite Automata (**NFA**) $\delta : \mathcal{S} \times \Sigma \rightarrow \mathcal{P}(\mathcal{S})$

The transition relation δ can be represented in a table (transition table)

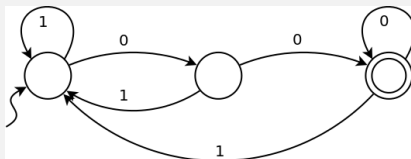
Overview of the graphical notation **circle and edges (arrows)**

Exercise

Define the following automata:

- ▶ DFA for a single 1
- ▶ DFA for accepting any number of 1's followed by a single 0
- ▶ DFA for any sequence of a or b (possibly empty) followed by 'abb'

Exercise



Which regular expression corresponds to the automaton?

- 1 $(0|1)^*$
- 2 $(1^*|0)(1|0)$
- 3 $1^*|(01)^*|(001)^*|(000^*1)^*$
- 4 $(0|1)^*00$

ϵ -moves

DFA, NFA and ϵ -moves

• DFA

- one transition per input per state
- no ϵ -moves
- faster

• NFA

- can have multiple transitions for one input in a given state
- can have ϵ -moves
- smaller (exponentially)

ϵ -moves

DFA, NFA and ϵ -moves

- DFA
 - one transition per input per state
 - no ϵ -moves
 - faster
- NFA
 - can have multiple transitions for one input in a given state
 - can have ϵ -moves
 - smaller (exponentially)

From regexp to NFA

Equivalent NFA for a regexp

The **Thompson's algorithm** permits to automatically derive a NFA from the specification of a regexp. It defines basic NFA for the basic regexp and **rules to compose** them:

- 1 for ϵ
- 2 for 'a'
- 3 for AB
- 4 for A|B
- 5 for A*

Now consider the regexp for $(1|0)^*1$

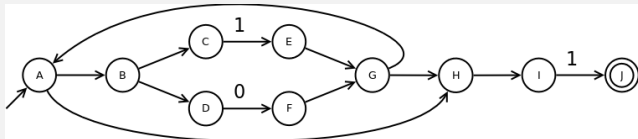
NFA to DFA

NFA to DFA

Given a NFA accepting a language \mathcal{L} there exists a DFA accepting the same language

The derivation of a DFA from an NFA is based on the concept of ϵ -closure. The algorithm to make the transformation is based on:

- ϵ -closure(s) with $s \in \mathcal{S}$
- ϵ -closure(\mathcal{T}) with $\mathcal{T} \subseteq \mathcal{S}$ i.e. $= \{\cup_{s \in \mathcal{T}} \epsilon$ -closure(s)\}
- move(\mathcal{T}, a) with $\mathcal{T} \subseteq \mathcal{S}$ and $a \in \mathcal{L}$



1 $(a|b)^*abb$

NFA 2 DFA

Subset Construction Algorithm

The Subset construction algorithm permits to derive a DFA $\langle \mathcal{S}, \Sigma, \delta_D, s_0, \mathcal{F}_D \rangle$ from a NFA $\langle \mathcal{N}, \Sigma, \delta_N, n_0, \mathcal{F}_N \rangle$

```

 $q_0 \leftarrow \epsilon - \text{closure}(\{n_0\});$ 
 $\mathcal{Q} \leftarrow q_0;$ 
Worklist  $\leftarrow \{q_0\};$ 
while (Worklist  $\neq \emptyset$ ) do
  take and remove  $q$  from Worklist;
  for all ( $c \in \Sigma$ ) do
     $t \leftarrow \epsilon - \text{closure}(\text{move}(q, c));$ 
     $T[q, c] \leftarrow t;$ 
    if ( $t \notin \mathcal{Q}$ ) then
       $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{t\};$ 
      Worklist  $\leftarrow \text{Worklist} \cup \{t\};$ 
    end if
  end for
end while

```

DFA 2 Minimal DFA

Note

Reducing the size of the Automaton does not reduce the number of moves needed to recognize a string, nevertheless it reduces the size of the transition table that could more easily fit the **size of a cache**

Equivalent states

Two states of a DFA are equivalent if they produce the same “behaviour” on any input string. Formally two states s_i and s_j of a DFA $\mathcal{D} = \langle S, \Sigma, \delta, q_0, \mathcal{F} \rangle$ are considered **equivalent** ($s_i \equiv s_j$) **iff** $\forall x \in \Sigma^*. (s_i \rightarrow^x s'_i \wedge s'_i \in \mathcal{F}) \iff (s_j \rightarrow^x s'_j \wedge s'_j \in \mathcal{F})$

DFA 2 Minimal DFA – Hopcroft's Algorithm

Let T a matrix containing information about the equivalence of two states and

let L a matrix containing sets (initially empty) of pairs of states

for all $s_x \in S \wedge s_y \in S$ **do**

$T[s_x, s_y] \leftarrow 0;$ // All pairs of states are initially marked as equivalent

end for

for all $s_x \in F \wedge s_y \in S/F$ **do**

$T[s_x, s_y] \leftarrow 1;$ $T[s_y, s_x] \leftarrow 1;$

end for

for all $\langle s_x, s_y \rangle$ s.t. $T[s_x, s_y] = 0 \wedge s_x \neq s_y$ **do**

if $(\exists c \in \Sigma. T[\delta(s_x, c), \delta(s_y, c)] = 1)$ **then**

$T[s_x, s_y] \leftarrow 1;$ $T[s_y, s_x] \leftarrow 1;$

for all $\langle s_w, s_z \rangle \in L[s_x, s_y]$ **do**

$T[s_w, s_z] \leftarrow 1;$ $T[s_z, s_w] \leftarrow 1;$

end for

else

for all $c \in \Sigma$ **do**

if $(\delta(s_x, c) \neq \delta(s_y, c) \wedge (s_x, s_y) \neq (\delta(s_x, c), \delta(s_y, c)))$ **then**

$L[\delta(s_x, c), \delta(s_y, c)] \leftarrow L[\delta(s_x, c), \delta(s_y, c)] \cup \langle s_x, s_y \rangle;$

$L[\delta(s_y, c), \delta(s_x, c)] \leftarrow L[\delta(s_y, c), \delta(s_x, c)] \cup \langle s_x, s_y \rangle;$

end if

end for

end if

end for

Uniqueness of the minimal DFA

∃! DFA that recognizes a regular language \mathcal{L} and has minimal number of states

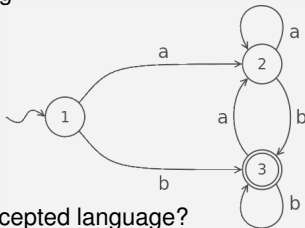
Minimizing Transition Table

The easiest way to represent a DFA is to have a matrix with state and characters. Alternative representations:

- Lists of pairs for each state (character,states)
- hardcoded table into case statements

Example

Consider the following DFA:



Transition Table

		characters	
		a	b
states	1	2	3
	2	2	3
	3	2	3

- ▶ Which is the accepted language?
- ▶ How can the table be represented as a list of pairs?

Exercises

Regular Expressions

Write a regular expression for each of the following languages:

- ▶ Given an alphabet $\Sigma = \{0, 1\}$, L is the set of strings composed by pairs of 0 and pairs of 1
- ▶ Given an alphabet $\Sigma = \{1, b, c, d\}$, L is the set of strings $xyzwy$, where x and w are strings of one or more characters in Σ , y is any single character in Σ and z is the character 'z', taken from outside the alphabet
- ▶ Floating-point numbers

Finite Automata

Construct a FA accepting the following languages:

- ▶ $\{w \in \{a, b\}^* \mid w \text{ starts with 'a' and contains the substring 'baba'}\}$
- ▶ $\{w \in \{a, b, c\}^* \mid \text{in } w \text{ the number of 'a's modulo 2 is equal to the number of 'b's modulo 3}\}$

Exercises

RegExp 2 DFA

- ▶ Consider the RegExp $a(b|c)^*$ and derive the accepting DFA.
- ▶ Define an automated strategy to decide if two regular expressions define the same language combining the algorithms defined so far

Regular Languages properties

- ▶ Show that the complement of a regular language, on alphabet Σ , is still a regular language
- ▶ Show that the intersection of two regular languages, on alphabet Σ , is still a regular language

Scanner issues

Describe the behaviour of a scanner when the two tokens described by the following patterns are considered: ab and $(ab)^*c$. Why a simple scanner is particularly inefficient on a string like 'abababababab'?

Summary

Lexical Analysis

Relevant concepts we have encountered:

- Tokens, Patterns, Lexemes
- Chomsky hierarchy and regular languages
- Regular expressions
- Problems and solutions in matching strings
- DFA and NFA
- Transformations
 - RegExp \rightarrow NFA
 - NFA \rightarrow DFA
 - DFA \rightarrow Minimal DFA