2. Lexical Analysis

Andrea Polini, Luca Tesei

Compilers
MSc in Computer Science
University of Camerino

ToC

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

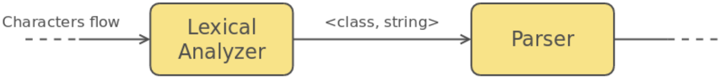
Lexical Analysis

- 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 analyser (Lexer) is:

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

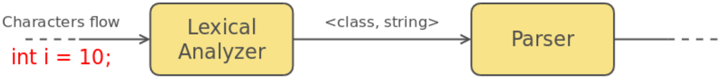


Why is not wise to merge the two components?

Lexical Analysis

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

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

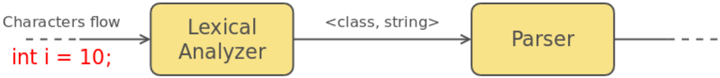


Why is not wise to merge the two components?

Lexical Analysis

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

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



Why is not wise to merge the two components?

Lexical Analysis

Let's analyse these lines of code:

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

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

Token Classes: Identifier, Integer, Keyword, Whitespace

Lexical Analysis

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

- Recognise 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

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

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

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

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 recognise 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

Strings

Parts of a string

Terms related to stings:

- a prefix of a string *s* is the string obtained removing zero or more characters from the end of *s*
- a suffix of a string *s* is the string obtained removing zero or more characters from the beginning of *s*
- a substring of a string *s* is obtained deleting any prefix and any suffix from *s*
- proper prefixes, suffixes and substrings of a string *s* are those prefixes, suffixes and substrings of *s*, respectively, that are not empty (ϵ) or not equal to *s* itself
- a subsequence of a string *s* is any string formed by deleting zero or more not necessarily consecutive positions of *s*

Regular expressions (regexp): Syntax

To form a syntactically correct regexp we have the following rules:

- Single character: 'c' is a regexp for each $c \in \Sigma$;
- Epsilon: ϵ is a regexp;
- Union: $r_1 + r_1$ is a regexp if r_1 and r_1 are regexps (also written $r_1 | r_2$);
- Concatenation: $r_1 \cdot r_2$ is a regexp if r_1 and r_2 are regexps (also written $r_1 r_2$);
- Iteration (Kleene star): r^* is a regexp if r is a regexp;
- Brackets: (r) is a regexp if r is a regexp

Regular expressions (regexp): Syntax

To avoid too much brackets we fix the following precedence and associativity rules:

- $*$ has the highest precedence and is left associative
- $\cdot$ has the second highest precedence and is left associative
- $+$ has the lowest precedence and is left associative
- e.g., $a + bc^*$ means $a + (b(c^*))$; $abc + d + e$ means $(((ab)c) + d) + e$; ...

Moreover we will use the following shorthands:

- At least one: $r^+ \equiv rr^*$
- Option: $r? \equiv r + \epsilon$
- Range: $[a - z] \equiv {}'a' + {}'b' + \cdots + {}'z'$
- Excluded range: $[^\wedge a - z] \equiv$ complement of $[a - z]$

Meaning function $\mathscr{L}$

- The meaning function $\mathscr{L}$ maps syntax to semantics: $\mathscr{L}(e) = \mathscr{M}$ where e is a regexp and $\mathscr{M}$ is a set of strings

Given an alphabet Σ and regular expressions a and b over Σ:

- $\mathscr{L}(\epsilon) = \{\epsilon\}$
- $\mathscr{L}('c') = \{c\}$, where $c \in \Sigma$
- $\mathscr{L}(r_1 + r_2) = \mathscr{L}(r_1) \cup \mathscr{L}(r_2)$
- $\mathscr{L}(r_1 r_2) = \mathscr{L}(r_1) \odot \mathscr{L}(r_2)$
- $\mathscr{L}(r^*) = \bigcup_{i \geq 0} \mathscr{L}(r)^i$ where $\begin{cases} \mathscr{L}(r)^0 = \{\epsilon\} \\ \mathscr{L}(r)^i = \mathscr{L}(r) \odot \mathscr{L}(r)^{i-1} \end{cases}$

$\odot$ is the concatenation of languages:

$$L_1 \odot L_2 = \{s_1 s_2 \mid s_1 \in L_1 \wedge s_2 \in L_2\}$$

Some equivalence laws for regexps

Given regexps r_1 and r_2, they are equivalent, written $r_1 \equiv r_2$, if and only if $\mathscr{L}(r_1) = \mathscr{L}(r_2)$

Let r, r_1, r_2, r_3 be regexps, then:

$r_1 + r_2 \equiv r_2 + r_1$	$+$ is commutative
$r_1 + (r_2 + r_3) \equiv (r_1 + r_2) + r_3$	$+$ is associative
$r + r \equiv r$	$+$ is idempotent
$r_1(r_2 r_3) \equiv (r_1 r_2) r_3$	$\cdot$ is associative
$r_1(r_2 + r_3) \equiv r_1 r_2 + r_1 r_3$	$\cdot$ distributes over $+$ on the left
$(r_1 + r_2) r_3 \equiv r_1 r_3 + r_2 r_3$	$\cdot$ distributes over $+$ on the right
$r\epsilon \equiv \epsilon r \equiv r$	ϵ is the identity for $\cdot$
$(\epsilon + r)^* \equiv r^*$	ϵ is guaranteed in a closure
$r^{**} \equiv r^*$	the Kleene star is idempotent

Regular Languages

Semantics of Regular Expressions

Regular expressions (syntax)
specify regular languages (semantics)

A language L is regular if and only if there exists a regular expression r such that $\mathscr{L}(r) = L$

Closure Properties of Regular Languages

Regular languages are closed with respect to union, intersection, complement

If L_1 and L_2 are regular languages then $L_1 \cup L_2$, $L_1 \cap L_2$ and L_1^c are regular languages

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

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

Exercise

Describe the languages denoted by the following RegExp:

- $a(a|b)^*a$
- $a^*ba^*ba^*ba^*$
- $((\epsilon|a)b^*)^*$

Regular definitions

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

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

where:

- Each d_i is a new symbol, not in Σ, and not the same as any other of the d's
- Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

Using regular definitions

The tokens of a language can be defined as:

- $letter \rightarrow a|b|...|z|A|B|...|Z$
- $letter\_ \rightarrow letter|\_$
  - compact syntax: $[a - zA - B]$
- $digit \rightarrow 0|1|...|9$
  - compact syntax: $[0 - 9]$
- $integers \rightarrow (-|\epsilon)digit \cdot digit^*$
- $identifiers \rightarrow letter\_(letter\_|digit)*$
- $expnot \rightarrow digit(.digit^+E(+|-)digit^+)?$ (Exponential Notation)

Exercise

Write regular definitions for the following languages:

- All strings of lowercase letters that contains the five vowels in order
- All strings of lowercase letters in which the letters are in ascending lexicographic order
- All strings of digits with no repeated digits
- All strings with an even number of a's and and an odd number of b's

How does the lexical analyser work?

Suppose we are given a regular definition $R = \{d_1, \ldots, d_m\}$

1. Let the input be $x_0 \cdots x_n \in \Sigma^*$
   For $0 \leq i \leq n$ check if $x_0 \cdots x_i \in \mathscr{L}(d_k)$ for some $k \in \{1, \ldots, m\}$
2. if success then we know that $x_0 \cdots x_i \in \mathscr{L}(d_k)$ for some k
3. remove $x_0 \cdots x_i$ from input and go to 1

However, things are not so simple. . . consider the following regular definition:

1. $d_1 \rightarrow a$ - token `T1`
2. $d_2 \rightarrow abb$ - token `T2`
3. $d_3 \rightarrow a^* b^+$ - token `T3`

Input: *aaba*, which are the tokens to recognise?

LA matching rules

Suppose that at the same time for $i < j$, $i, j \in \{0, \ldots, n\}$:

- $x_0 \cdots x_i \in \mathscr{L}(d_k)$ for some k, and
- $x_0 \cdots x_i \cdots x_j \in \mathscr{L}(d_h)$ for some h

Which is the match to consider?

longest match rule, i.e., pattern d_h is recognised

Suppose that at the same time for $i \in \{0, \ldots, n\}$ and $k < h$, $k, h \in \{1, \ldots, m\}$:

- $x_0 \cdots x_i \in \mathscr{L}(d_k)$
- $x_0 \cdots x_i \in \mathscr{L}(d_h)$

Which is the match to consider?

first one listed rule, i.e., pattern d_k is recognised

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

LA matching rules

Suppose that at the same time for $i < j$, $i, j \in \{0, \ldots, n\}$:

- $x_0 \cdots x_i \in \mathscr{L}(d_k)$ for some k, and
- $x_0 \cdots x_i \cdots x_j \in \mathscr{L}(d_h)$ for some h

Which is the match to consider?

longest match rule, i.e., pattern d_h is recognised

Suppose that at the same time for $i \in \{0, \ldots, n\}$ and $k < h$, $k, h \in \{1, \ldots, m\}$:

- $x_0 \cdots x_i \in \mathscr{L}(d_k)$
- $x_0 \cdots x_i \in \mathscr{L}(d_h)$

Which is the match to consider?

first one listed rule, i.e., pattern d_k is recognised

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

LA matching rules

Suppose that at the same time for $i < j$, $i, j \in \{0, \ldots, n\}$:

- $x_0 \cdots x_i \in \mathscr{L}(d_k)$ for some k, and
- $x_0 \cdots x_i \cdots x_j \in \mathscr{L}(d_h)$ for some h

Which is the match to consider?

longest match rule, i.e., pattern d_h is recognised

Suppose that at the same time for $i \in \{0, \ldots, n\}$ and $k < h$, $k, h \in \{1, \ldots, m\}$:

- $x_0 \cdots x_i \in \mathscr{L}(d_k)$
- $x_0 \cdots x_i \in \mathscr{L}(d_h)$

Which is the match to consider?

first one listed rule, i.e., pattern d_k is recognised

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

LA matching rules

Suppose that at the same time for $i < j$, $i, j \in \{0, \ldots, n\}$:

- $x_0 \cdots x_i \in \mathscr{L}(d_k)$ for some k, and
- $x_0 \cdots x_i \cdots x_j \in \mathscr{L}(d_h)$ for some h

Which is the match to consider?

longest match rule, i.e., pattern d_h is recognised

Suppose that at the same time for $i \in \{0, \ldots, n\}$ and $k < h$, $k, h \in \{1, \ldots, m\}$:

- $x_0 \cdots x_i \in \mathscr{L}(d_k)$
- $x_0 \cdots x_i \in \mathscr{L}(d_h)$

Which is the match to consider?

first one listed rule, i.e., pattern d_k is recognised

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

LA matching rules

Suppose that at the same time for $i < j$, $i, j \in \{0, \ldots, n\}$:

- $x_0 \cdots x_i \in \mathscr{L}(d_k)$ for some k, and
- $x_0 \cdots x_i \cdots x_j \in \mathscr{L}(d_h)$ for some h

Which is the match to consider?

longest match rule, i.e., pattern d_h is recognised

Suppose that at the same time for $i \in \{0, \ldots, n\}$ and $k < h$, $k, h \in \{1, \ldots, m\}$:

- $x_0 \cdots x_i \in \mathscr{L}(d_k)$
- $x_0 \cdots x_i \in \mathscr{L}(d_h)$

Which is the match to consider?

first one listed rule, i.e., pattern d_k is recognised

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

LA matching rules

Suppose that at the same time for $i < j$, $i, j \in \{0, \ldots, n\}$:

- $x_0 \cdots x_i \in \mathscr{L}(d_k)$ for some k, and
- $x_0 \cdots x_i \cdots x_j \in \mathscr{L}(d_h)$ for some h

Which is the match to consider?

longest match rule, i.e., pattern d_h is recognised

Suppose that at the same time for $i \in \{0, \ldots, n\}$ and $k < h$, $k, h \in \{1, \ldots, m\}$:

- $x_0 \cdots x_i \in \mathscr{L}(d_k)$
- $x_0 \cdots x_i \in \mathscr{L}(d_h)$

Which is the match to consider?

first one listed rule, i.e., pattern d_k is recognised

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

Implementation of LA

- How to implement this algorithm for any given regular definition?
- First, it would be convenient to use a device that is able to recognise automatically the lexemes corresponding to each pattern
- Finite Automata are the devices that are more convenient from an algorithmic point of view
- Then, we should find a way to combine these automata for all the patterns of the given regular definition and to implement the matching rules
- Non-determinism will do the trick
- Finally, we should try to optimise everything, which will be done by eliminating non-determinism and by minimising the resulting deterministic automaton

Finite Automata

- Regular Expressions = specification of tokens
- Finite Automata = recognition of tokens

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 ($\delta : \mathcal{S} \times \Sigma \rightarrow \ldots$)
- 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 flavours: Deterministic Finite Automata (DFA) and Non-Deterministic Finite Automata (NFA)

Finite Automata

- Regular Expressions = specification of tokens
- Finite Automata = recognition of tokens

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 ($\delta : \mathcal{S} \times \Sigma \rightarrow \ldots$)
- 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 flavours: Deterministic Finite Automata (DFA) and Non-Deterministic Finite Automata (NFA)

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) $\delta : \mathcal{S} \times \Sigma \rightarrow \mathscr{P}(\mathcal{S})$

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

$\mathscr{P}(\mathcal{S}) = 2^{\mathcal{S}}$ is the powerset of the set $\mathcal{S}$ of states, i.e., the set of all the subsets of $\mathcal{S}$

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) $\delta : \mathcal{S} \times \Sigma \rightarrow \mathscr{P}(\mathcal{S})$

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

$\mathscr{P}(\mathcal{S}) = 2^{\mathcal{S}}$ is the powerset of the set $\mathcal{S}$ of states, i.e., the set of all the subsets of $\mathcal{S}$

Overview of the graphical notation circle and edges (arrows)

Acceptance of Strings for DFAs

Moves of a DFA

A DFA "consumes" an input character c going from a state s to a state s' if $\delta(s, c) = s'$, written $s \xrightarrow{c} s'$

A DFA "consumes" a string $\mathbf{a} = a_1 a_2 \cdots a_n$ going from a state s_i to a state s_j if there is a sequence of states $s_{i+1}, \ldots, s_{i+n-1}, s_{i+n} = s_j$ s.t. $\forall k \in \{1, \ldots, n\}.\delta(s_{i+k-1}, a_k) = s_{i+k}$, written $s_i \xrightarrow{\mathbf{a}} s_j$

Acceptance of Strings

A DFA accepts a string $\mathbf{a}$ if and only if it consumes $\mathbf{a}$ from the initial state s_0 to a final state s_i, i.e., $s_0 \xrightarrow{\mathbf{a}} s_i$ and $s_i \in \mathcal{F}$

Accepted Language

The language accepted by a DFA is the set of all the strings $\mathbf{a}$ such that $s_0 \xrightarrow{\mathbf{a}} s_i$ and $s_i \in \mathcal{F}$

Acceptance of Strings for DFAs

Moves of a DFA

A DFA "consumes" an input character c going from a state s to a state s' if $\delta(s, c) = s'$, written $s \xrightarrow{c} s'$

A DFA "consumes" a string $\mathbf{a} = a_1 a_2 \cdots a_n$ going from a state s_i to a state s_j if there is a sequence of states $s_{i+1}, \ldots, s_{i+n-1}, s_{i+n} = s_j$ s.t.

$\forall k \in \{1, \ldots, n\}.\delta(s_{i+k-1}, a_k) = s_{i+k}$, written $s_i \xrightarrow{\mathbf{a}} s_j$

Acceptance of Strings

A DFA accepts a string **a** if and only if it consumes **a** from the initial state s_0 to a final state s_i, i.e., $s_0 \xrightarrow{\mathbf{a}} s_i$ and $s_i \in \mathcal{F}$

Accepted Language

The language accepted by a DFA is the set of all the strings **a** such that $s_0 \xrightarrow{\mathbf{a}} s_i$ and $s_i \in \mathcal{F}$

Acceptance of Strings for DFAs

Moves of a DFA

A DFA "consumes" an input character c going from a state s to a state s' if $\delta(s, c) = s'$, written $s \xrightarrow{c} s'$

A DFA "consumes" a string $\mathbf{a} = a_1 a_2 \cdots a_n$ going from a state s_i to a state s_j if there is a sequence of states $s_{i+1}, \ldots, s_{i+n-1}, s_{i+n} = s_j$ s.t. $\forall k \in \{1, \ldots, n\}.\delta(s_{i+k-1}, a_k) = s_{i+k}$, written $s_i \xrightarrow{\mathbf{a}} s_j$

Acceptance of Strings

A DFA accepts a string **a** if and only if it consumes **a** from the initial state s_0 to a final state s_i, i.e., $s_0 \xrightarrow{\mathbf{a}} s_i$ and $s_i \in \mathcal{F}$

Accepted Language

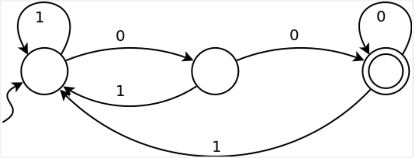
The language accepted by a DFA is the set of all the strings **a** such that $s_0 \xrightarrow{\mathbf{a}} s_i$ and $s_i \in \mathcal{F}$

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
  - at most one transition for one input in a given state
  - no ϵ-moves
- NFA
  - can have multiple transitions for one input in a given state
  - can have ϵ-moves, i.e., $\delta : \mathcal{S} \times (\Sigma \cup \{\epsilon\}) \to \mathscr{P}(\mathcal{S})$
  - smaller (exponentially)

Acceptance of Strings for NFAs

Moves of an NFA

An NFA "consumes" an input character c going from a state s to a state s' if $s' \in \delta(s, c)$, written $s \xrightarrow{c} s'$

An NFA can move from a state s to a state s' without consuming any input character, written $s \xrightarrow{\epsilon} s'$

An NFA "consumes" a string $\mathbf{a} = a_1 a_2 \cdots a_n$ going from a state s_i to a state s_j if there is a sequence of moves $s_i \xrightarrow{x_0} s_{i+1} \xrightarrow{x_1} \ldots s_{i+m-1} \xrightarrow{x_{m-1}} s_{i+m} = s_j$ s.t. $\forall k \in \{0, \ldots, m-1\}. s_{i+k} \in \delta(s_{i+k}, x_k)$ and $x_0 x_1 \cdots x_{m-1} = \mathbf{a}$, written $s_i \xRightarrow{\mathbf{a}} s_j$

Acceptance of Strings

An NFA accepts a string $\mathbf{a}$ if and only if there exists at least one sequence of moves from the initial state s_0 to a state s_i such that s_i is a final state, i.e., $\exists s_i \in \mathcal{F} : s_0 \xRightarrow{\mathbf{a}} s_i$

Accepted Language

The language accepted by an NFA is the set of all the strings $\mathbf{a}$ such that $\exists s_i \in \mathcal{F} : s_0 \xRightarrow{\mathbf{a}} s_i$

Acceptance of Strings for NFAs

Moves of an NFA

An NFA "consumes" an input character c going from a state s to a state s' if $s' \in \delta(s, c)$, written $s \xrightarrow{c} s'$

An NFA can move from a state s to a state s' without consuming any input character, written $s \xrightarrow{\epsilon} s'$

An NFA "consumes" a string $\mathbf{a} = a_1 a_2 \cdots a_n$ going from a state s_i to a state s_j if there is a sequence of moves $s_i \xrightarrow{x_0} s_{i+1} \xrightarrow{x_1} \ldots s_{i+m-1} \xrightarrow{x_{m-1}} s_{i+m} = s_j$ s.t. $\forall k \in \{0, \ldots, m-1\}. s_{i+k} \in \delta(s_{i+k}, x_k)$ and $x_0 x_1 \cdots x_{m-1} = \mathbf{a}$, written $s_i \stackrel{\mathbf{a}}{\Longrightarrow} s_j$

Acceptance of Strings

An NFA accepts a string **a** if and only if there exists at least one sequence of moves from the initial state s_0 to a state s_i such that s_i is a final state, i.e., $\exists s_i \in \mathcal{F} \colon s_0 \stackrel{\mathbf{a}}{\Longrightarrow} s_i$

Accepted Language

The language accepted by an NFA is the set of all the strings **a** such that $\exists s_i \in \mathcal{F} \colon s_0 \stackrel{\mathbf{a}}{\Longrightarrow} s_i$

Acceptance of Strings for NFAs

Moves of an NFA

An NFA "consumes" an input character c going from a state s to a state s' if $s' \in \delta(s, c)$, written $s \xrightarrow{c} s'$

An NFA can move from a state s to a state s' without consuming any input character, written $s \xrightarrow{\epsilon} s'$

An NFA "consumes" a string $\mathbf{a} = a_1 a_2 \cdots a_n$ going from a state s_i to a state s_j if there is a sequence of moves $s_i \xrightarrow{x_0} s_{i+1} \xrightarrow{x_1} \ldots s_{i+m-1} \xrightarrow{x_{m-1}} s_{i+m} = s_j$ s.t. $\forall k \in \{0, \ldots, m-1\}. s_{i+k} \in \delta(s_{i+k}, x_k)$ and $x_0 x_1 \cdots x_{m-1} = \mathbf{a}$, written $s_i \xRightarrow{\mathbf{a}} s_j$

Acceptance of Strings

An NFA accepts a string $\mathbf{a}$ if and only if there exists at least one sequence of moves from the initial state s_0 to a state s_i such that s_i is a final state, i.e., $\exists s_i \in \mathcal{F} \colon s_0 \xRightarrow{\mathbf{a}} s_i$

Accepted Language

The language accepted by an NFA is the set of all the strings $\mathbf{a}$ such that $\exists s_i \in \mathcal{F} \colon s_0 \xRightarrow{\mathbf{a}} s_i$

From regexp to NFA

Equivalent NFA for a regexp

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

1. for ϵ
2. for 'c'
3. for ab
4. for $a + b$
5. for a^*

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

Implementation of Lexical Analyser

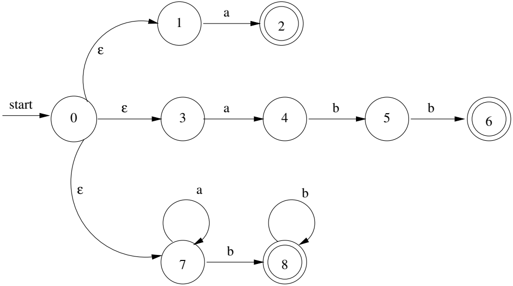
- Recall the matching rules, i.e., the way in which the LA should work to recognise the tokens of a given regular definition $R = \{d_1, \ldots, d_m\}$
- We can use Thompson's algorithm to create NFAs A_1 for d_1, ..., A_m for d_m
- We can create a fresh new initial state s_0 and connect it with an ϵ transition to all the (unique) initial states of A_1, ..., A_m
- The (unique) final state f_i of A_i recognises the lexemes of token i for all i
- We can then use this combined NFA to implement the matching rules

Implementation of LA: Example

- Let R be :

$$\begin{array}{lll} d_1 = & a & \{\texttt{TOKEN1}\} \\ d_2 = & abb & \{\texttt{TOKEN2}\} \\ d_3 = & a^*b^+ & \{\texttt{TOKEN3}\} \end{array}$$

- The combined NFA of the three NFAs obtained from d_1, d_2 and d_3 is the following (the NFA for d_3 is simplified, actually made deterministic):



Implementation of LA: Example cont'd

- The LA must record the last time in which the automaton was in a final state (null at the beginning)
- To do this it implements a lookahead with two variables:
  - `Last_Final`: it is the set of the last occurred final states (empty at the beginning)
  - `Input_Pos_at_Last_Final`: it records the position on the input corresponding to the last occurred final state
- These positions must be reset when the the lookahead is "too ahead", i.e., the input is terminated or the automaton is blocked
- Simulation of ϵ-transitions will be handled by ϵ-*closure*(s) (s single state); and
- ϵ-*closure*$(\mathcal{T}) = \bigcup_{s \in \mathcal{T}} \epsilon$-*closure*$(s)$ ($\mathcal{T}$ set of states)

Implementation of LA: Example cont'd

- Let's apply this idea to the input *aaba*
- Initially, the automaton is in the set of states ϵ-*closure*$(0) = \{0, 1, 3, 7\}$
- The first input character *a* is read and the automaton moves to states ϵ-*closure*$(\delta(\{0, 1, 3, 7\}, a)) = \{2, 3, 7\}$
- Now 2 is a final state, so we set `Last_Final` $= \{2\}$ and `Input_Pos_at_Last_Final` $= 1$. This must be considered a partial result, we need to go ahead because there could be a longer input prefix that corresponds to a lexeme
- The second character *a* is read making the automaton reach the set of states $\{7\}$, which does not contain final states, so we go on
- The third character *b* is read and the set of states $\{8\}$ is reached, and 8 is final state. Thus we update: `Last_Final` $= \{8\}$ and `Input_Pos_at_Last_Final` $= 3$. We go on

Implementation of LA: Example cont'd

- The fourth character *a* is read and the automaton is blocked because there are no transitions labelled with *a* from state 8.
- The LA outputs `TOKEN3` with lexeme *aab* and resets the variables to the the initial state with the remaining input *a*

The LA restarts with input *a*:

- Initially, the automaton is in the set of states ϵ-*closure*$(0) = \{0, 1, 3, 7\}$
- The first input character *a* is read and the automaton moves to states ϵ-*closure*$(\delta(\{0, 1, 3, 7\}, a)) = \{2, 3, 7\}$
- Now 2 is a final state, so we set `Last_Final` $= \{2\}$ and `Input_Pos_at_Last_Final` $= 1$. This must be considered a partial result, we need to go ahead because there could be a longer input prefix that corresponds to a lexeme
- The automaton is blocked because the input is terminated. The LA outputs `TOKEN1` with lexeme *a* and terminates.

Implementation of LA: Example cont'd

- The pattern matching algorithm that we have just given correctly implements the longest match rule
- Note that `Last_Final` is a set of states
- If it contains more than one state and the LA decides to output the token, the final state corresponding to the highest d_i in R must be considered to correctly implement the first one listed rule

The automaton that is used by the LA is non-deterministic, thus it must simulate the non-determinism and the ϵ-closure:

- A real LA would be more efficient if the given automaton was deterministic
- $\rightarrow$ we can transform the NFA into an equivalent DFA (possible exponential blow up of states)
- A real LA would be more efficient if the given deterministic automaton had a minimal number of states
- $\rightarrow$ we can minimise the obtained DFA

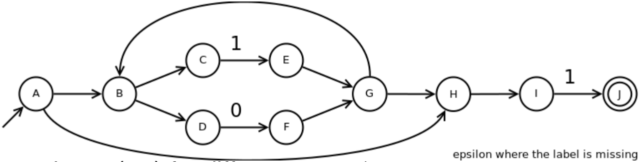
NFA to DFA

NFA 2 DFA

Given an NFA accepting a language $\mathscr{L}$ 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 subset construction algorithm makes the transformation using the following operations:
  - ϵ-*closure*(s) with $s \in \mathcal{S}$
  - ϵ-*closure*$(\mathcal{T}) = \bigcup_{s \in \mathcal{T}} \epsilon$-*closure*$(s)$ where $\mathcal{T} \subseteq \mathcal{S}$
  - *move*$(\mathcal{T}, a)$ with $\mathcal{T} \subseteq \mathcal{S}$ and $a \in \Sigma$

NFA to DFA



- build the ϵ-*closure*(...) for different states/sets
- build *move*($\mathcal{T}$, *a*) for different sets and elements

NFA to 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 an NFA $\langle \mathcal{N}, \Sigma, \delta_N, n_0, \mathcal{F}_N \rangle$

$s_0 \leftarrow \epsilon\text{-}closure(\{n_0\})$; $\mathcal{S} \leftarrow \{s_0\}$; $\mathcal{F}_D \leftarrow \emptyset$; worklist $\leftarrow \{s_0\}$;
if $(s_0 \cap \mathcal{F}_N \neq \emptyset)$ **then** $\mathcal{F}_D \leftarrow \mathcal{F}_D \cup s_0$;
end if
while (worklist $\neq \emptyset$) **do**
  take and remove q from worklist;
  for all $(c \in \Sigma)$ **do**
    $t \leftarrow \epsilon\text{-}closure(move(q, c))$;
    $\delta_D[q, c] \leftarrow t$;
    if $(t \notin \mathcal{S})$ **then**
      $\mathcal{S} \leftarrow \mathcal{S} \cup t$; worklist $\leftarrow$ worklist $\cup\, t$;
    end if
    if $(t \cap \mathcal{F}_N \neq \emptyset)$ **then** $\mathcal{F}_D \leftarrow \mathcal{F}_D \cup t$;
    end if
  end for
end while

Simulating DFA and NFA

DFA

$s = s_0$;
$c = nextChar()$;
while (c $\neq$ **eof**) **do**
  $s = move(s, c)$;
  $c = nextChar()$;
end while
if ($s \in \mathcal{F}$) **then return** “yes”;
else return “no”;
end if

NFA

$S = \epsilon\text{-}closure(s_0)$;
$c = nextChar()$;
while (c $\neq$ **eof**) **do**
  $S = \epsilon\text{-}closure(move(S, c))$;
  $c = nextChar()$;
end while
if ($S \cap \mathcal{F} \neq \emptyset$) **then return** “yes”;
else return “no”;
end if

Exercises NFA to DFA

- Derive an NFA for the regexp: $(a|b)^*abb$
- NFA to DFA for the obtained NFA

Exercises NFA to DFA

- Derive an NFA for the regexp: $(a|b)^* abb$
- NFA to DFA for the obtained NFA

DFA to Minimal DFA

Note

Reducing the size of the automaton does not reduce the number of moves needed to recognise 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.

Let $\mathcal{D} = \langle \mathcal{S}, \Sigma, \delta, q_0, \mathcal{F} \rangle$ be a DFA. Two states s_i and s_j of $\mathcal{D}$ are considered equivalent, written $s_i \equiv s_j$, **iff**

$$\forall \mathbf{x} \in \Sigma^*.(s_i \xrightarrow{\mathbf{x}} s_i' \wedge s_i' \in \mathcal{F}) \iff (s_j \xrightarrow{\mathbf{x}} s_j' \wedge s_j' \in \mathcal{F})$$

DFA to Minimal DFA – Partition Refinement Algorithm

Deriving a minimal DFA

Transform a DFA $\langle \mathcal{S}, \Sigma, \delta_D, s_0, \mathcal{F}_D \rangle$ into a minimal *DFA* $\langle \mathcal{S}', \Sigma, \delta'_D, s'_0, \mathcal{F}'_D \rangle$

// Π *is a partition of the set of states* $\mathcal{S}$
$\Pi \leftarrow \{\mathcal{F}_D, \mathcal{S} - \mathcal{F}_D\}$ // *Initially there are only two groups of states: final states and non-final states*
repeat
    $\Pi_{\text{new}} \leftarrow \Pi$ // *create a working copy* Π_{new}
    for all groups G in Π **do**
        partition G in subgroups $G_1, \ldots, G_n$ ($n \geq 1$) such that two states s and t are in the same subgroup G_i iff
        $\forall c \in \Sigma\ ((s \overset{c}{\not\longrightarrow}) \wedge (t \overset{c}{\not\longrightarrow})) \vee ((s \overset{c}{\longrightarrow} s') \wedge (t \overset{c}{\longrightarrow} t') \wedge (s', t' \in \bar{G})$ for some group $\bar{G}$ in Π)
        // *subgroups* G_i*'s may be composed of only one state*
        $\Pi_{\text{new}} \leftarrow \Pi_{\text{new}} - G \cup \{G_1, \ldots, G_n\}$ // *Replace G with the obtained subgroups in* Π_{new}
        // *the partition is refined: the group G is possibly replaced with a finer partition* $G_1, \ldots, G_n$
    end for
until $\Pi_{\text{new}} = \Pi$ // *exit when the partition cannot be refined further*
// *Now* Π *contains a set of groups that are a partition of the states* $\mathcal{S}$
// *The algorithm continues with the construction of the minimal DFA . . .*

DFA to Minimal DFA – Partition Refinement Algorithm

// *Continues from the previous slide . . .*
// *the states of the minimal DFA are representatives of groups of equivalent states, those that are in* Π
$\mathcal{S}' \leftarrow \varnothing$ and $\mathcal{F}'_D \leftarrow \varnothing$
for all groups G in Π **do**
    choose a state in G as the *representative* for G and add it to $\mathcal{S}'$
    if $G \cap \mathcal{F}_D \neq \varnothing$ // *G contains either all final states or all non-final states* **then**
        add the representative state for G also to $\mathcal{F}'_D$
    end if
end for
$s'_0 \leftarrow$ the representative state of the group G containing s_0
for all states $s \in \mathcal{S}'$ **do**
    for all charachters $c \in \Sigma$ **do**
        if $\delta_D[s, c]$ is defined **then**
            $\delta'_D[s, c] \leftarrow$ the representative state of the group G containing the state $\delta_D[s, c]$
        end if
    end for
end for

Uniqueness of the minimal DFA

There exists a unique DFA, up to isomorphism, that recognises a regular language $\mathscr{L}$ and has minimal number of states. Two DFA are isomorphic iff they are equal by neglecting the labels of the states.

Exercises

RegExp 2 DFA

- Minimise the DFA for the regexp $(a|b)^*abb$
- Consider the regexp $a(b|c)^*$ and derive the minimal 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

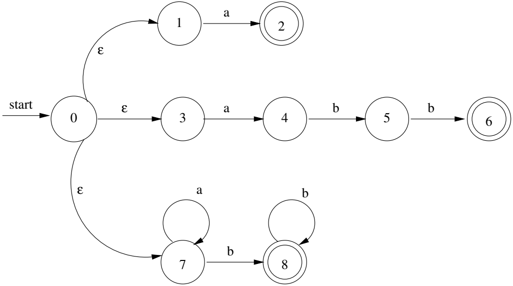
- Specify a DFA accepting all strings of *a*'s and *b*'s that do not contain the substring *aab*
- Show that the complement of a regular language, on alphabet Σ, is still a regular language
- Show that the intersection of two regular languages, on alphabet Σ, is still a regular language

Recall of Implementation of LA: Example

- Let R be :

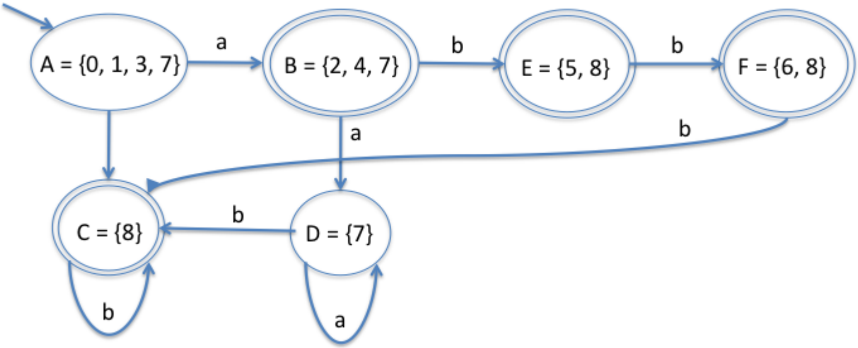
$$\begin{array}{lll} d_1 = & a & \{\texttt{TOKEN1}\} \\ d_2 = & abb & \{\texttt{TOKEN2}\} \\ d_3 = & a^*b^+ & \{\texttt{TOKEN3}\} \end{array}$$

- The combined NFA of the three NFAs obtained from d_1, d_2 and d_3 is the following (the NFA for d_3 is simplified, actually made deterministic):



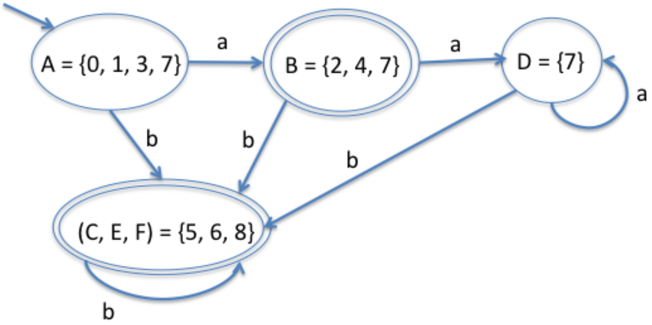
Implementation of LA: Optimisation

- The behaviour of the LA can be optimised by determinizing the NFA and then by minimising the states
- The DFA obtained from the combined NFA for *R* is:



Implementation of LA: Optimisation

- By performing a standard minimisation the following minimal DFA is obtained:



Implementation of LA: Optimisation

- Let's scan the input *aaba*
- $A \xrightarrow{a} B$, $\texttt{Last\_Final} = \{2\}$, $\texttt{Input\_Pos\_at\_Last\_Final} = 1$
- $B \xrightarrow{a} D$
- $D \xrightarrow{b} (C, E, F)$, $\texttt{Last\_Final} = \{6, 8\}$, $\texttt{Input\_Pos\_at\_Last\_Final} = 3$
- $(C, E, F) \not\xrightarrow{a}$
- The LA cannot decide which token to output! Final state 6 would call for TOKEN 2 (incorrect!) and final state 8 would call for TOKEN 3!

We need to retain the information on the final states!

Implementation of LA: Optimisation

- We must start the minimisation of the DFA by initially splitting the group of final states into subgroups
- A subgroup for each set of reached final states must be created
- subgroup 1 = $\{B\}$ for TOKEN 1 - only final state 2
- subgroup 2 = $\{C, E\}$ for TOKEN 3 - only final state 8
- subgroup 3 = $\{F\}$ for TOKEN 2 and TOKEN 3 - final states $\{6, 8\}$
- The other non-final states can be grouped together as usual

$$\Pi_1 = \{(A, D), (B), (C, E), (F)\}$$

Implementation of LA: Optimisation

- The group (A, D) can be refined: $A \xrightarrow{a} B$ and $D \xrightarrow{a} D$
- $\Pi_2 = \{(A), (D), (B), (C, E), (F)\}$
- The group (C, E) can be refined: $C \xrightarrow{b} C$ and $E \xrightarrow{b} F$
- $\Pi_3 = \{(A), (D), (B), (C), (E), (F)\}$
- Π_3 cannot be refined further!
- The minimal DFA to use for the LA scanning is just the same DFA

Implementation of LA: Optimisation

- Let's scan the input *aaba*
- $A \stackrel{a}{\longrightarrow} B$, $\texttt{Last\_Final} = \{2\}$, $\texttt{Input\_Pos\_at\_Last\_Final} = 1$
- $B \stackrel{a}{\longrightarrow} D$
- $D \stackrel{b}{\longrightarrow} C$, $\texttt{Last\_Final} = \{8\}$, $\texttt{Input\_Pos\_at\_Last\_Final} = 3$
- $C \stackrel{a}{\not\longrightarrow}$
- The LA outputs TOKEN 3 with lexeme *aab*, then clear the recognised input and restart
- $A \stackrel{a}{\longrightarrow} B$, $\texttt{Last\_Final} = \{2\}$, $\texttt{Input\_Pos\_at\_Last\_Final} = 1$
- $B \not\longrightarrow$ end of input
- The LA outputs TOKEN 1 with lexeme *a*, then stops.

ToC

Languages

Language

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

Alphabet = English character $\Longrightarrow$ Language = English sentences
Alphabet = ASCII $\Longrightarrow$ Language = C programs

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

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

Grammar Definition

Grammar

A Grammar $\mathcal{G}$ is a tuple $\langle \mathcal{V}_\mathcal{T}, \mathcal{V}_\mathcal{N}, \mathcal{S}, \mathcal{P} \rangle$ where:

- $\mathcal{V}_\mathcal{T}$ is a finite and non empty set of terminal symbols (alphabet)
- $\mathcal{V}_\mathcal{N}$ is a finite set of non-terminal symbols s.t. $\mathcal{V}_\mathcal{N} \cap \mathcal{V}_\mathcal{T} = \varnothing$
- $\mathcal{S} \in \mathcal{V}_\mathcal{N}$ is the start symbol
- $\mathcal{P}$ is a finite set of productions s.t. $\mathcal{P} \subseteq (\mathcal{V}^* \cdot \mathcal{V}_\mathcal{N} \cdot \mathcal{V}^*) \times \mathcal{V}^*$ where $\mathcal{V} = \mathcal{V}_\mathcal{T} \cup \mathcal{V}_\mathcal{N}$

Derivations

Derivations

Given a grammar $\mathcal{G} = \langle \mathcal{V}_{\mathcal{T}}, \mathcal{V}_{\mathcal{N}}, \mathcal{S}, \mathcal{P} \rangle$ a derivation is a sequence of strings $\phi_1, \phi_2, ..., \phi_n$ s.t.
$\forall i \in \{1, .., n\}.\phi_i \in \mathcal{V}^* \wedge \forall i \in \{1, ..., n-1\}.\exists p \in \mathcal{P}: \phi_i \rightarrow^p \phi_{i+1}$
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 $\mathcal{G} = \langle \mathcal{V}_{\mathcal{T}}, \mathcal{V}_{\mathcal{N}}, \mathcal{S}, \mathcal{P} \rangle$ corresponds to: $\mathcal{L}(\mathcal{G}) = \{x \mid x \in \mathcal{V}_{\mathcal{T}}^* \wedge \mathcal{S} \rightarrow^* x\}$

Derivations

Derivations

Given a grammar $\mathcal{G} = \langle \mathcal{V}_{\mathcal{T}}, \mathcal{V}_{\mathcal{N}}, \mathcal{S}, \mathcal{P} \rangle$ a derivation is a sequence of strings $\phi_1, \phi_2, ..., \phi_n$ s.t.
$\forall i \in \{1, .., n\}.\phi_i \in \mathcal{V}^* \wedge \forall i \in \{1, ..., n-1\}.\exists p \in \mathcal{P} : \phi_i \rightarrow^p \phi_{i+1}$
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 $\mathcal{G} = \langle \mathcal{V}_{\mathcal{T}}, \mathcal{V}_{\mathcal{N}}, \mathcal{S}, \mathcal{P} \rangle$ corresponds to: $\mathcal{L}(\mathcal{G}) = \{x \mid x \in \mathcal{V}_T^* \wedge \mathcal{S} \rightarrow^* x\}$

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: $\alpha A \beta \to \alpha \gamma \beta$
- Recognizing Automaton: Linear Bound Automaton (LBA)

T2. Context-Free Grammars:

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

T3. Regular Grammars:

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

Meaning function $\mathscr{L}$

Meaning Function

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

- e.g. the case for numbers

- Why using a meaning function?
  - Makes clear what is syntax, what is semantics
  - Allows us to consider notation as a separate issue
  - Expressions and meanings are not 1 to 1

Warning

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

Meaning function $\mathscr{L}$

Meaning Function

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

- e.g. the case for numbers

- Why using a meaning function?
  - Makes clear what is syntax, what is semantics
  - Allows us to consider notation as a separate issue
  - Expressions and meanings are not 1 to 1

Warning

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

Meaning function $\mathscr{L}$

Meaning Function

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

- e.g. the case for numbers

- Why using a meaning function?
  - Makes clear what is syntax, what is semantics
  - Allows us to consider notation as a separate issue
  - Expressions and meanings are not 1 to 1

Warning

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

Summary

Lexical Analysis

Relevant concepts we have encountered:

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