# Formal Languages and Software Verification

*Alessandro Aldini*

DiSBeF - Sezione STI
University of Urbino "Carlo Bo" – Italy

# Outline

# Prelude

## What is a language?

May refer either to the (human) capacity for acquiring and using complex systems of communication, or to a specific instance of such a system of complex communication

## ...and a formal language?

A language is formal if it is provided with a mathematically rigorous representation of its:

- alphabet of symbols, and
- rules specifying which strings of symbols count as well-formed

# Applications

## Why describing properties of sequences of symbols, formally

Formal languages are studied in:

- **mathematics**, for representing the syntax of axiomatic systems
- **linguistics**, for the scientific study of human language
- **computational biology**, for the description of the genome structure
- **computer science**, for the interpretation of commands and software verification
- ...

# Grammars and formal languages

## A generative view

Formal sentences are generated by applying a finite set of formation rules defining the grammar of the language

## A little bit of history

- Symbolic notations (*Frege*'s predicate logics)
- Combinatorics on words (*Thue*'s pattern representations of the structure of enumerable objects)
- *Chomsky* hierarchy of grammars

# Automata and formal languages

## A recognition view

Formal sentences of a language are recognized by an abtract machine, called automaton, that solves the membership problem

## A little bit of history

- Decidability theory (*Turing* machine, *Kleene* finite-state automata)
- *Backus* studies in the development of compilers (pushdown automata)
- Model theory (*Kripke* structures, timed/probabilistic automata)

# Alphabets, strings, and formal languages

## Definition

Alphabet: a *finite*, non-empty set of symbols

## Example

- $\Sigma_1 = \{0, 1\}$
- $\Sigma_2 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $\Sigma_3 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$
- $\Sigma_4 = \{a, b, c, \ldots, z\}$

## Notation

$a$, $b$, $c$, ... denote *symbols*

# Alphabets, strings, and formal languages

## Definition

String (or word) on alphabet $\Sigma$: a finite sequence of symbols in $\Sigma$

## Example

- $1010 \in \Sigma_1$
- $123 \in \Sigma_2$
- $\text{hello} \in \Sigma_4$

## Notation

- $\varepsilon$ is the empty string
- $v$, $w$, $x$, $y$, $z$, ... denote *strings*
- $|w|$ is the length of $w$ (number of symbols in $w$)

# Alphabets, strings, and formal languages

## Definition

$k$-th power of $\Sigma$: $\Sigma^k \stackrel{\text{def}}{=} \{a_1 \cdots a_k \mid a_1, \ldots, a_k \in \Sigma\}$

## Example

- $\Sigma^0 = \{\varepsilon\}$ for any $\Sigma$
- $\Sigma_1^1 = \{0, 1\}$
- $\Sigma_1^2 = \{00, 01, 10, 11\}$

## Definition

Kleene closures of an alphabet $\Sigma$:

$$\begin{array}{lll} \Sigma^* & \stackrel{\text{def}}{=} & \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \\ \Sigma^+ & \stackrel{\text{def}}{=} & \Sigma^* \setminus \{\varepsilon\} \end{array} \quad \begin{array}{l} = \bigcup_{i=0}^{\infty} \Sigma^i \\ = \bigcup_{i=1}^{\infty} \Sigma^i \end{array}$$

# Alphabets, strings, and formal languages

## So, what is a formal language?

Any subset of $\Sigma^*$

## Example

- English, Chinese, ...
- C, Pascal, Java, HTML, ...
- the set of binary numbers whose value is prime: $\{10, 11, 101, 111, 1011, \ldots\}$
- $\emptyset$ (the empty language)
- $\{\varepsilon\}$

# Alphabets, strings, and formal languages

## Definition

Operations on languages:

- $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$
- $\overline{L_1} = \{w \in \Sigma_1^* \mid w \notin L_1\}$
- $L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$
- $L_1^* = \{\varepsilon\} \cup L_1 \cup L_1^2 \cup \cdots$
- $L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$

# Some interesting result about languages

## About enumerability

- The set of strings of $\Sigma^*$ is **recursively enumerable**
- The set of languages with alphabet $\Sigma$ is **not recursively enumerable**

*Proof*

| | $w_1$ | $w_2$ | $w_3$ | $\cdots$ |
|---|---|---|---|---|
| $L_1$ | 0 | 1 | 1 | $\cdots$ |
| $L_2$ | 1 | 1 | 0 | $\cdots$ |
| $L_3$ | 1 | 0 | 0 | $\cdots$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ |

$$\begin{aligned} D &= \{w_i \mid w_i \in L_i\} = \{w_2, \dots\} \\ \overline{D} &= \{w_i \mid w_i \notin L_i\} = \{w_1, w_3, \dots\} \end{aligned}$$

Suppose by contradiction that $\exists j$ such that $\overline{D} = L_j$:

$$w_j \in \overline{D} \Rightarrow w_j \notin L_j \quad or \quad w_j \notin \overline{D} \Rightarrow w_j \in L_j \quad \text{absurd!}$$

Conclusion: no such $L_j$ exists in the enumeration

# Relation with computability theory

## More on enumerability

- The set of generating grammars (and recognizing automata) is **recursively enumerable**
- Conclusion: there exist languages with no such representations
- Analogy with decidability results by Church ($\lambda$-calculus, general recursive functions) and Turing (abstract machine)

## Turing machine

- In 1936, Alan Turing describes an abstract state machine for reasoning about automatic computation
- Turing machine simulates the logic of any computer algorithm
- It is an abstract model based on an infinite memory in the form of a strip of tape
- Through his abstract machine, Turing shows the undecidability of the *halting problem*

# Turing machine

## Turing machine components

- *Tape*: is divided into cells that may contain symbols of $\Sigma$
- *Control system*: can read/write symbols and move the tape left/right one cell at a time
- Set $Q$: represents the internal states of the control system, among which we distinguish the initial one and the final ones
- Transition function $P$: defines the instructions for the machine (given the current state and symbol, $P$ specifies the new symbol and state and the movement direction)
  *Example:* $P(q, a) = (b, p, left)$

## Turing automaton

A Turing machine used to recognize a formal language[a]

[a] . . . determines the expressiveness boundaries of our investigation

# Where this story begins. . . [1]

*"An organism or robot receives certain stimuli and performs certain actions"*

- In 1951, Kleene defines the simplest deterministic abstract machine for language recognition
- His goal is to represent the stimulus/response behavioral model of nerve nets
- Strings of symbols represent sequences of events, the finite set of states of the automaton represents the internal states of the net

## Example

| | $a$ | $b$ |
|---|---|---|
| $\rightarrow q_0$ | $q_1$ | $q_0$ |
| $* q_1$ | $q_1$ | $q_1$ |

[1] . . . from the expressiveness standpoint

# DFA: formal definition

## Definition

A Deterministic Finite-state Automaton (DFA) is a tuple

$$A = (Q, \Sigma, \delta, q_0, F)$$

where

- $Q$ is a finite set of states
- $\Sigma$ is an alphabet of input symbols
- $\delta : Q \times \Sigma \to Q$ is the transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final (or accepting) states

# DFA: how does it work?

**Input string:** $a_1 a_2 \cdots a_n$

Reading such a string corresponds to visit a path along the automaton:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_n} q_n$$

such that:

- $q_0$ is the initial state
- $q_{i+1} = \delta(q_i, a_{i+1})$
- if $q_n \in F$ the string $a_1 a_2 \cdots a_n$ is accepted, otherwise it is rejected

# DFA: example

## Example

Create a DFA that recognizes the language

$$L = \{x \in \{0,1\}^* \mid 01 \text{ is a substring of } x\}$$

- We need 3 states:
  - $q_0$ "I've seen no symbols, or all the symbols I've seen so far were 1"
  - $q_1$ "the last symbol I saw was 0"
  - $q_2$ "I've seen a 01 substring"
- The alphabet is $\Sigma = \{0,1\}$
- The initial state is $q_0$
- The set of final states is $F = \{q_2\}$

# DFA: example

## Example

Create a DFA that recognizes the language

$$L = \{x \in \{0,1\}^* \mid 01 \text{ is a substring of } x\}$$

- We have to specify the transitions:
  - from $q_0$, read 0, go to $q_1$
  - from $q_0$, read 1, stay in $q_0$
  - from $q_1$, read 0, stay in $q_1$
  - from $q_1$, read 1, go to $q_2$
  - from $q_2$, read 0 or 1, stay in $q_2$

# DFA: example

## Example

Create a DFA that recognizes the language

$$L = \{x \in \{0,1\}^* \mid 01 \text{ is a substring of } x\}$$

- Formally, we have defined the automaton:

$$A = (\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_2\})$$

where

$$\begin{aligned}
\delta(q_0, 0) &= q_1 \\
\delta(q_0, 1) &= q_0 \\
\delta(q_1, 0) &= q_1 \\
\delta(q_1, 1) &= q_2 \\
\delta(q_2, 0) &= q_2 \\
\delta(q_2, 1) &= q_2
\end{aligned}$$

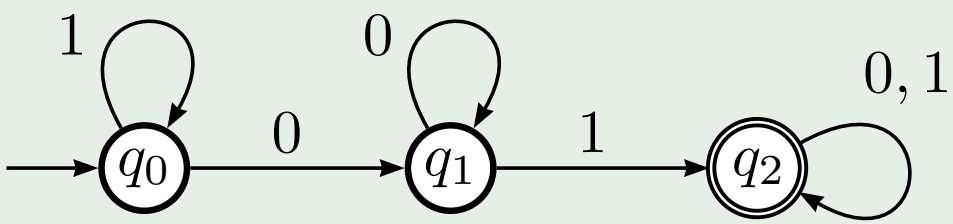
# DFA: graphical representation

## Transition diagram

A more convenient way of representing (small) DFA $(Q, \Sigma, \delta, q_0, F)$:

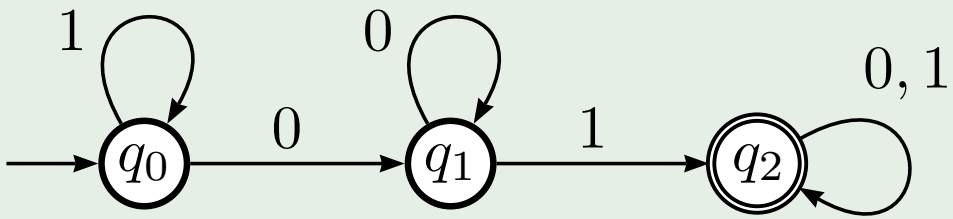
- for each state $q \in Q$ there is a node labeled $q$
- for each state $q \in Q$ and each symbol $a \in \Sigma$ there is an arc labeled $a$ from the node labeled $q$ to the node labeled $\delta(q, a)$
- the initial node $q_0$ has an incoming arrow
- final states $q \in F$ are emphasized

## Example



# DFA: tabular representation

## Example




Transition table:

| | 0 | 1 |
|---|---|---|
| $\rightarrow q_0$ | $q_1$ | $q_0$ |
| $q_1$ | $q_1$ | $q_2$ |
| $* q_2$ | $q_2$ | $q_2$ |

# Language accepted by a DFA: formal definition

## Extending the transition relation

- The transition function $\delta$ is a single-step relation
- It must be extended to deal with sequences of transitions: $\hat{\delta}(q, w)$ denotes the state reached from $q$ after reading $w$

## Definition

$$\begin{aligned} \hat{\delta} : Q \times \Sigma^* &\rightarrow Q \\ \hat{\delta}(q, \varepsilon) &= q \\ \hat{\delta}(q, xa) &= \delta(\hat{\delta}(q, x), a) \end{aligned}$$

Language accepted by DFA $A = (Q, \Sigma, \delta, q_0, F)$:

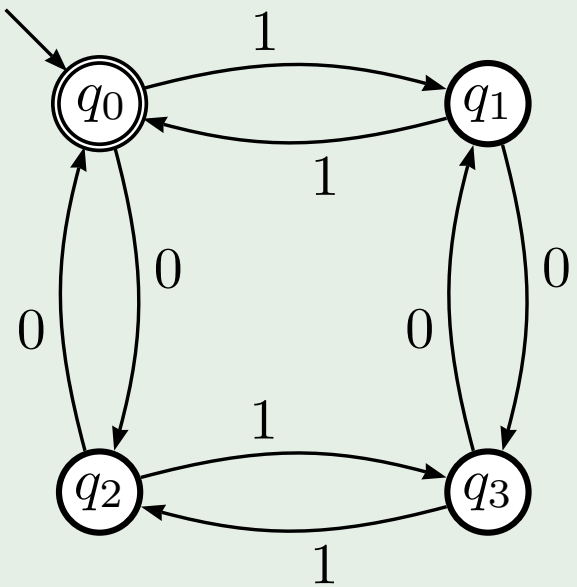
$$L(A) \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$$

# Language accepted by a DFA: example

## Example

$$L = \{w \in \{0,1\}^* \mid w \text{ has an even number of 0s and 1s}\}$$

The automaton that accepts $L$:

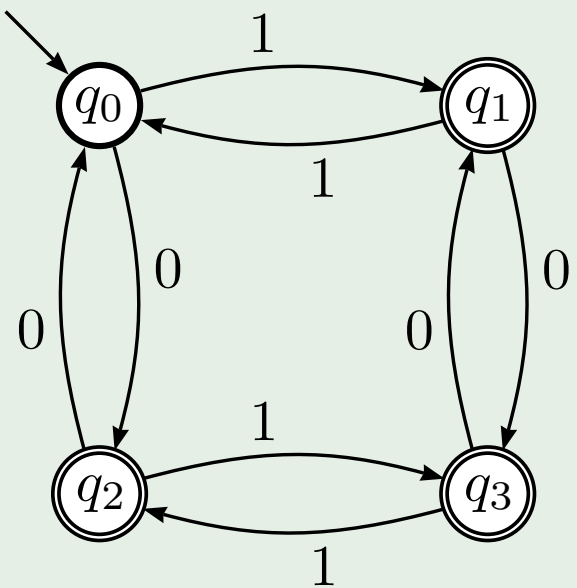


# Language accepted by a DFA: example

## Example

$$L = \{w \in \{0,1\}^* \mid w \text{ has an even number of 0s and 1s}\}$$

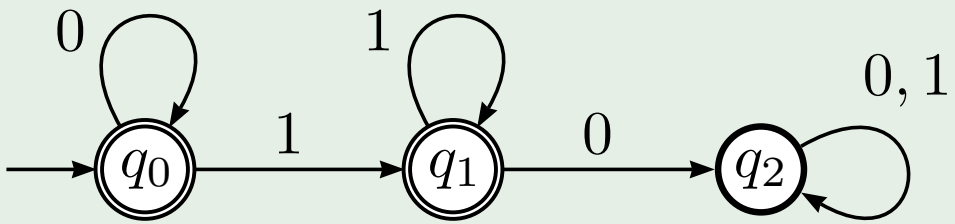
The automaton that accepts $\overline{L}$:



# Language accepted by a DFA: example

## Example

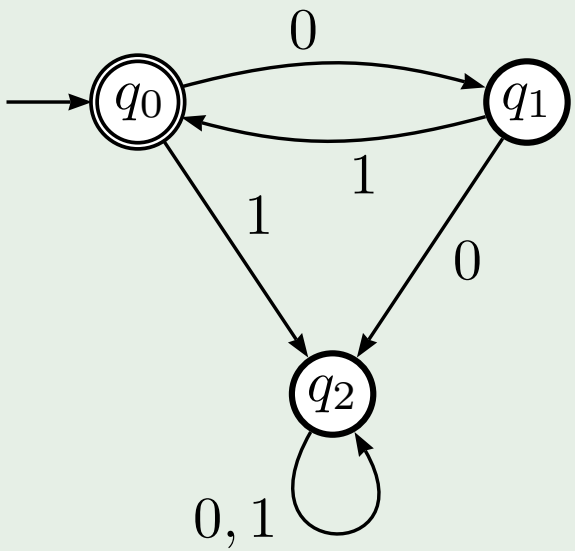
$$L = \{0^n 1^m \mid n \geq 0, m \geq 0\}$$



# Language accepted by a DFA: example

## Example

$$L = \{(01)^n \mid n \geq 0\}$$



# Regular expressions

## Alternative characterization

- Kleene offers another way of representing languages recognized by DFA
- The alternative model employs syntactic expressions, called regular expressions, to construct *compositionally* strings of symbols
- Kleene defines three composition operators: union (operator $+$), concatenation, and star (operator $^*$)
- The semantics of a regular expression is a set of strings

# Regular expressions: syntax and semantics

## Definition

| Syntax $E$ | Semantics $L(E)$ | |
|---|---|---|
| $\varepsilon$ | $\{\varepsilon\}$ | |
| $\emptyset$ | $\emptyset$ | |
| $\mathbf{a}$ | $\{a\}$ | $a \in \Sigma$ |
| $E_1 + E_2$ | $L(E_1) \cup L(E_2)$ | |
| $E_1 E_2$ | $L(E_1)L(E_2)$ | |
| $E^*$ | $L(E)^*$ | |
| $(E)$ | $L(E)$ | |

# Regular expressions: examples

## Example

**ab**

$$L(\mathbf{ab}) = L(\mathbf{a})L(\mathbf{b}) = \{ab\}$$

$\mathbf{a} + \mathbf{b}$

$$L(\mathbf{a} + \mathbf{b}) = L(\mathbf{a}) \cup L(\mathbf{b}) = \{a, b\}$$

$\mathbf{a} + \varepsilon$

$$L(\mathbf{a} + \varepsilon) = L(\mathbf{a}) \cup L(\varepsilon) = \{a, \varepsilon\}$$

$\mathbf{a}^*$

$$L(\mathbf{a}^*) = \{\varepsilon\} \cup L(\mathbf{a}) \cup L(\mathbf{a})L(\mathbf{a}) \cup \cdots = \{a^n \mid n \geq 0\}$$

# Regular expressions: examples

## Example

$\mathbf{a}^*\mathbf{b}^*$

$$\begin{aligned} L(\mathbf{a}^*\mathbf{b}^*) = L(\mathbf{a}^*)L(\mathbf{b}^*) &= \{a^m \mid m \geq 0\}\{b^n \mid n \geq 0\} \\ &= \{a^m b^n \mid m, n \geq 0\} \end{aligned}$$

$(\mathbf{ab})^*$

$$\begin{aligned} L((\mathbf{ab})^*) = L(\mathbf{ab})^* &= \{\varepsilon, ab, abab, ababab, \dots\} \\ &= \{(ab)^n \mid n \geq 0\} \end{aligned}$$

$(\mathbf{a} + \mathbf{b})^*$

$$L((\mathbf{a} + \mathbf{b})^*) = L(\mathbf{a} + \mathbf{b})^* = \{a, b\}^*$$

# Regular expressions: algebraic properties

## Theorem

$$
\begin{aligned}
E + F &= F + E \\
(E + F) + G &= E + (F + G) \\
(EF)G &= E(FG) \\
\emptyset + E = E + \emptyset &= E \\
\varepsilon E = E\varepsilon &= E \\
\emptyset E = E\emptyset &= \emptyset
\end{aligned}
\qquad
\begin{aligned}
E + E &= E \\
(E^*)^* &= E^* \\
\emptyset^* &= \varepsilon \\
\varepsilon^* &= \varepsilon \\
E(F + G) &= EF + EG \\
(F + G)E &= FE + GE
\end{aligned}
$$

# RE and DFA define regular languages

**Theorem (Kleene main result)**

*Regular expressions and finite-state automata characterize the same class of languages*

Proof sketch

- From RE to DFA: by induction on the structure of the RE, define first the automata for the atomic regular expressions, and then exploit the inductive hypothesis to show the structure of the DFA of composite regular expressions
- From DFA to RE (much more challenging): by induction on the number of distinct states visited by the accepting paths of the automaton, which are translated into regular expressions

# Expressiveness issues

*A machine of Turing (1937) which is supplied with an unlimited amount of tape, is not a finite automaton in our present sense, since, although in its operation only a finite number of squares of tape are printed upon at any time, there is no preassigned bound to this number*

Kleene, 1956

# Some properties

## Theorem (Closure properties)

*If $L_1$ and $L_2$ are regular languages over $\Sigma_1$ and $\Sigma_2$, then:*

- $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$
- $\overline{L_1} = \{w \in \Sigma_1^* \mid w \notin L_1\}$
- $L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$
- $L_1^* = \{\varepsilon\} \cup L_1 \cup L_1^2 \cup \cdots$
- $L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$

*are regular languages*

## Note

Proofs are given for *nondeterministic* finite-state automata

# Some properties

## Theorem (Decidability properties)

*These problems:*

- *membership* ($w \in L$)
- *emptyness* ($L = \emptyset$)
- *inclusion* ($L_1 \subseteq L_2$)
- *equivalence* ($L_1 = L_2$)
- *finiteness* ($|L| \neq \infty$)

*are decidable for regular languages*

## Note

Proofs are given for *nondeterministic* finite-state automata

# Nondeterministic finite-state automata

## Intuition

In some state, when presented with an input symbol, the automaton may decide nondeterministically to choose among different states to move to, or to reject the symbol, or to evolve internally without reading the symbol

## Definition

A Non-deterministic Finite-state Automaton (NFA) is a tuple

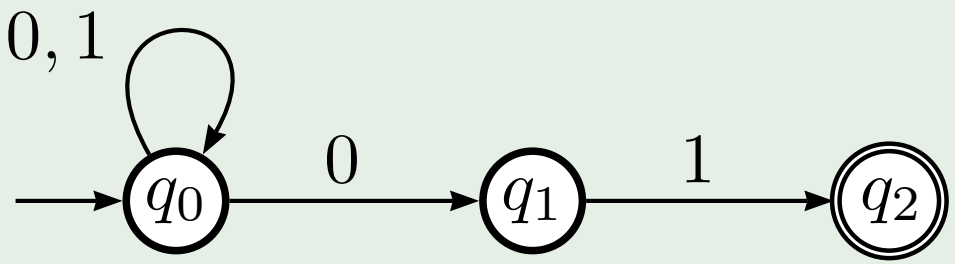
$$A = (Q, \Sigma, \delta, q_0, F)$$

where

- $Q$ is a finite set of states
- $\Sigma$ is an alphabet of input symbols
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \wp(Q)$ is the transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final (or accepting) states

# NFA: how does it work?

## Example




There are several paths labeled with the same string:

$$q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_0$$
$$q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2$$

The automaton accepts $w$ if *there exists* a path labeled $w$ that leads to a final state

# Interpretations of nondeterminism

**When the automaton can choose among two or more paths**

- **Oracle:** it "knows" the right one
- **Parallelism:** it clones itself and tries all of them
- **Backtracking:** it saves its status (the state and the position in the input string) and tries one of them

**When the automaton has no way to proceed**

- **Oracle:** there was no way to accept the string
- **Parallelism:** the clone "dies"
- **Backtracking:** it backtracks to one of the saved state and tries another path

# Expressiveness of NFA

**Theorem**

*NFA and DFA are equivalent*

# Why DFA expressiveness is not enough . . .

## A digression about grammars

- A grammar is a rewriting system using syntactic transformation rules to manipulate strings

## Example

Consider the rewriting rules:

- Any **Sentence** is made of a **Subject**, a **Verb**, and an **Object**
- Possible subjects are **I** and **You**
- Possible verbs are **Eat** and **Buy**
- Possible objects are **Pen** and **Apple**

By applying the rules, we can derive any well-formed sentence:

$$\textbf{Sentence} \rightarrow \textbf{Subject Verb Object} \rightarrow \textbf{You Verb Object} \rightarrow \textbf{You Eat Object} \rightarrow \textbf{You Eat Apple}$$

# Grammar: formal definition

## Definition

A grammar is a tuple $G = (V, T, S, P)$ where

- $V$ is a finite, non-empty set of symbols called variables (or non-terminals or syntactic categories)
- $T$ is an alphabet of symbols called terminals
- $S \in V$ is the start (or initial) symbol of the grammar
- $P$ is a finite set of productions $\alpha \to \beta$ where $\alpha \in (V \cup T)^+$ and $\beta \in (V \cup T)^*$

## Example

$V = \{\textbf{Sentence}, \textbf{Subject}, \textbf{Verb}, \textbf{Object}\}$,
$T = \{\textbf{I}, \textbf{You}, \textbf{Eat}, \textbf{Buy}, \textbf{Pen}, \textbf{Apple}\}$, $S = \{\textbf{Sentence}\}$, and
$P = \{\textbf{Sentence} \to \textbf{Subject Verb Object}, \textbf{Subject} \to \textbf{I} \mid \textbf{You}$,
$\textbf{Verb} \to \textbf{Eat} \mid \textbf{Buy}, \textbf{Object} \to \textbf{Pen} \mid \textbf{Apple}\}$

# Grammar: derivations

## Definition

$\mu \rightarrow_G \gamma$ is a single-step derivation iff:

1. $\mu = \sigma\alpha\tau$
2. $\gamma = \sigma\beta\tau$
3. and $\alpha \rightarrow \beta \in P$

The reflexive and transitive closure of $\rightarrow_G$ is the multi-step derivation $\mu \rightarrow^*_G \gamma$ iff:

1. $\mu = \gamma$, or
2. $\exists\delta$ such that $\mu \rightarrow_G \delta$ and $\delta \rightarrow^*_G \gamma$

## Example

**Sentence** $\rightarrow^*_G$ **You Eat Apple**

# Language generated by a grammar

## Definition

The language generated by $G$, called $L(G)$, is:

$$L(G) \stackrel{\text{def}}{=} \{w \in T^* \mid S \rightarrow_G^* w\}$$

# Language generated by a grammar

## Example

- A fragment for a programming language:

$$\begin{aligned} \text{Statement} &\rightarrow \textit{if}\ \text{Boolean\_condition Statement}\ \textit{else} \\ &\quad\ \text{Statement} \mid \\ &\quad\ \textit{while}\ \text{Boolean\_condition Statement} \\ \text{Boolean\_condition} &\rightarrow \textit{true} \mid \textit{false} \mid \ldots \end{aligned}$$

- Try to describe the language generated by the grammar

$$G = (\{S\}, \{a, b\}, S, \{S \rightarrow aSb, S \rightarrow ab\})$$

- Find a grammar that generates the language of strings made of 0s and 1s such that all the 0s occur before all the 1s

# Chomsky hierarchy

By strict decreasing expressive power:

| | |
|---|---|
| General (Type-0) | no restrictions |
| Context-dependent (Type-1) | $\gamma A \delta \rightarrow \gamma \beta \delta$ |
| Context-free (Type-2)[a] | $A \rightarrow \beta$ |
| Right-Linear (Type-3)[b] | $A \rightarrow wB$ |

[a] Sufficient to model programming languages syntax
[b] Sufficient to model programming languages morphology

# About expressiveness, again

## Are DFA expressive enough?

- Type-2 generated languages, called *free* languages, are important!
- Regular languages $\subset$ free languages
- Limitation of DFA: memory finiteness

## Example

$$\begin{aligned} G &= \{\{S\}, \{a, b\}, S, \{S \to aSb \mid ab\}\} \\ L(G) &= \{a^n b^n \mid n \geq 1\} \end{aligned}$$

## Conclusion

- We need a class of automata that recognize *free* languages (Ginsburg, Greibach, and Harrison, 1967)
- And what about DFA: they are as expressive as Type-3 grammars

# Pushdown automata

## Pushdown automata, intuitively

They consist of:

- a nondeterministic finite-state automaton
- a stack of unlimited size (LIFO access policy)

The automaton can change state depending on:

- the current symbol in the input string
- the topmost symbol on the stack

# Pushdown automata: formal definition

## Definition

A pushdown automaton is a tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

1. $Q$ is a finite set of states
2. $\Sigma$ is the input alphabet
3. $\Gamma$ is the stack alphabet
4. $\delta : Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \to \wp(Q \times \Gamma^*)$ is the transition function
5. $q_0$ is the initial state
6. $Z_0$ is the initial symbol that appears on the stack
7. $F \subseteq Q$ is the set of final states

# Pushdown automata: instantaneous description

## PDA behavior

- The state of a PDA is fully described by:
  1. the current state $q$
  2. the string $w$ still to be read
  3. the content $\gamma$ of the stack
- The triple $(q, w, \gamma)$ is called **instantaneous description** (or **ID**)
- Transition: if $(p, \alpha) \in \delta(q, a, X)$ then $(q, aw, X\beta) \vdash (p, w, \alpha\beta)$
- The reflexive, transitive closure of $\vdash$ is relation $\vdash^*$
- Computation:

$$(q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha)$$

# Pushdown automata: example

## Example

$$L = \{a^n b^n \mid n \geq 1\}$$

- $Q = \{q, q_f\}$
- $\Sigma = \{a, b\}$
- $\Gamma = \Sigma \cup \{Z_0\}$
- $q_0 = q$
- $F = \{q_f\}$
- $\delta$ is such that $\delta(q, a, X) = (q, aX)$, $\delta(q, b, a) = (q, \varepsilon)$, $\delta(q, \varepsilon, Z_0) = (q_f, \varepsilon)$

# Pushdown automata: language accepted

## Language accepted by final state

$$L(P) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha) \wedge q \in F\}$$

## Language accepted by empty stack

$$N(P) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}$$

## Theorem

*Both formulations characterize the class of free languages*[a]

[a] . . . the subclass of deterministic PDA is strictly less expressive

## From CFG to PDA

From a CFG $G = (V, T, S, P)$ we create the PDA

$$M = (\{q\}, T, V \cup T, \delta, q, S)$$

such that for each variable $A$:

$$\delta(q, \varepsilon, A) = \{(q, \beta) \mid A \to \beta \in P\}$$

and for each terminal $a$:

$$\delta(q, a, a) = \{(q, \varepsilon)\}$$

# Some properties

## Theorem (Closure properties)

*If* $G_1 = (V_1, T_1, S_1, P_1)$ *and* $G_2 = (V_2, T_2, S_2, P_2)$ *are CFGs for two languages* $L_1 = L(G_1)$ *and* $L_2 = L(G_2)$*, and*

$$\begin{aligned} G_a &= (\{S\} \cup V_1 \cup V_2, T_1 \cup T_2, S, \{S \to S_1 S_2\} \cup P_1 \cup P_2) \\ G_b &= (\{S\} \cup V_1 \cup V_2, T_1 \cup T_2, S, \{S \to S_1 \mid S_2\} \cup P_1 \cup P_2) \\ G_c &= (\{S\} \cup V_1, T_1, S, \{S \to \varepsilon \mid S_1 S\} \cup P_1) \end{aligned}$$

*then*

$$\begin{aligned} L_1 L_2 &= L(G_a) \\ L_1 \cup L_2 &= L(G_b) \\ L_1^* &= L(G_c) \end{aligned}$$

# Some properties

## Theorem (Decidability properties)

*These problems:*

- *emptyness*
- *membership*
- *finiteness*
- *equivalence (for languages generated by deterministic PDA)*

*are decidable for free languages*

# Linear bounded automata

**In between PDA and TMs**

- The stack in a PDA obeys the LIFO access policy
- If we relax such a condition, by treating the auxiliary memory structure as a finite tape, we obtain the class of *linear bounded automata*
- Historically, LBA have been introduced as a practical variant of Turing machines (Myhill, 1960)

# Linear bounded automata: formal definition

## Definition

A linear bounded automaton is a tuple $M = (Q, \Sigma, B, \delta, q_0, F)$

1. $Q$ is a finite set of states
2. $\Sigma$ is the input alphabet ($\Sigma \subset B$)
3. $B$ is the tape alphabet, which includes $\perp$ (empty), $X_l$ (left boundary) and $X_r$ (right boundary)
4. $\delta : Q \times B \to \wp(Q \times B \times \{r, l\})$ is the transition function[a]
5. $q_0$ is the initial state
6. $F \subseteq Q$ is the set of final states

---

[a] $(q, a, m) \in \delta(p, X_l) \Rightarrow a = X_l \wedge m = r$ and
$(q, a, m) \in \delta(p, X_r) \Rightarrow a = X_r \wedge m = l$

# LBA: instantaneous description

## LBA behavior

- The state of a LBA is fully described by:
  - the content of the portion of tape on the left of the current position
  - the current state
  - the symbol in the current position
  - the remaining portion of tape on the right
- The triple $(\alpha q \beta)$ denotes the ID, where the current symbol is represented by the first symbol of $\beta$
- If $(p, b, r) \in \delta(q, a)$ then $(\alpha q a \gamma) \vdash (\alpha b p \gamma)$
- if $(p, b, l) \in \delta(q, a)$ then $(\alpha c q a \gamma) \vdash (\alpha p c b \gamma)$
- The reflexive, transitive closure of $\vdash$ is relation $\vdash^*$
- Computation:

$$(q_0 X_l w X_r) \vdash^* (\alpha q \gamma)$$

# Language accepted by LBA

## Language accepted by LBA

$$L(M) = \{w \in \Sigma^* \mid (q_0 X_l w X_r) \vdash^* (\alpha q X_r) \wedge q \in F\}$$

# LBA: example

## Example

$L = \{a^n b^n c^n \mid n \geq 1\}$

$$
\begin{aligned}
Q &= \{q_0, q_1, q_a, q_b, q_c\} \\
\Sigma &= \{a, b, c\} \\
B &= \Sigma \cup \{\bot, X_l, X_r\} \cup \{x, y, z\} \\
F &= \{q_1\}
\end{aligned}
$$

| $\delta$ | $X_l$ | $X_r$ | $a$ | $b$ | $c$ | $x$ | $y$ | $z$ |
|---|---|---|---|---|---|---|---|---|
| $q_0$ | $q_1 X_l r$ | | $q_0 a l$ | $q_0 b l$ | $q_0 c l$ | $q_1 x r$ | $q_0 y l$ | $q_0 z l$ |
| $q_a$ | | | $q_a a r$ | $q_b y r$ | | | $q_a y r$ | |
| $q_b$ | | | | $q_b b r$ | $q_c z r$ | | | $q_b z r$ |
| $q_c$ | | $X_r q_0 l$ | | | $q_c c r$ | | | |
| $q_1$ | | $\surd$ | $q_a x r$ | | | | $q_1 y r$ | $q_1 z r$ |

# LBA: example

## Example

$L = \{a^n b^n c^n \mid n \geq 1\}$

- scan the tape left to right by turning one $a$ into $x$, one $b$ into $y$, and one $c$ into $z$
- go back and repeat the scan
- if no $a$ is found and you can reach the end of the tape, then success!

| $\delta$ | $X_l$ | $X_r$ | $a$ | $b$ | $c$ | $x$ | $y$ | $z$ |
|---|---|---|---|---|---|---|---|---|
| $q_0$ | $q_1 X_l r$ | | $q_0 a l$ | $q_0 b l$ | $q_0 c l$ | $q_1 x r$ | $q_0 y l$ | $q_0 z l$ |
| $q_a$ | | | $q_a a r$ | $q_b y r$ | | | $q_a y r$ | |
| $q_b$ | | | | $q_b b r$ | $q_c z r$ | | | $q_b z r$ |
| $q_c$ | | $X_r q_0 l$ | | | $q_c c r$ | | | |
| $q_1$ | | $\surd$ | $q_a x r$ | | | | $q_1 y r$ | $q_1 z r$ |

# Some properties

## What we can do with LBA

- Expressiveness:
  - Same as Type-1 grammars
  - Are deterministic LBA a strict subclass of LBA? *Open problem*
- Closure: union, concatenation, Kleene star, intersection, complement
- Decidability: membership

# Back to Turing automata

## From the boundaries of LBA to those of decidability

- Formal definition: take LBA and remove the tape finiteness constraints
- Expressiveness:
  - Same as Type-0 grammars (they characterize the *recursively enumerable* languages)
  - Same as TA with semi-infinite tape, two or more tapes, 2-dimensional tape
  - Deterministic and nondeterministic TA have the same expressive power
- Closure: union, concatenation, Kleene star, intersection
- Decidability:
  - Any non-trivial property of RE languages is not decidable (*Rice* thm.)
  - Membership problem is semi-decidable

# References


- A. Aldini: Teoria degli automi per linguaggi formali (APhEx journal, vol. 9, 2014) http://www.aphex.it/

# Algorithmic verification

## Correctness problem

- Is the software/hardware system correct with respect to the expected requirements?
- Hard to determine whenever the system is concurrent
- Concurrency theory studies interacting parallel and distributed systems, called *reactive* systems
- Foundational ingredients:
  - automata for describing reactive systems
  - logics for expressing properties of reactive systems

# State-transition systems

## State

- . . . describes a configuration of the system in a given instant of time
- . . . may be labeled with additional information, like a formal statement specifying the conditions that hold in the state

## Transition

- . . . describes a state change due to the execution of a given event
- . . . may be labeled with additional information, like the name of the action representing the executed event

# Kripke structure

## Definition

Let $\mathrm{AP}$ be a set of atomic propositions
A Kripke structure is a tuple

$$K = (S, S_0, R, L)$$

where

- $S$ is a finite set of states
- $S_0 \subseteq S$ is the set of initial states
- $R \subseteq S \times S$ is a total transition relation
- $L : S \to 2^{\mathrm{AP}}$ is a labeling function that associates each state with the subset of atomic propositions that are true in that state

# Modeling reactive systems through Kripke structures

## First-order logic representation

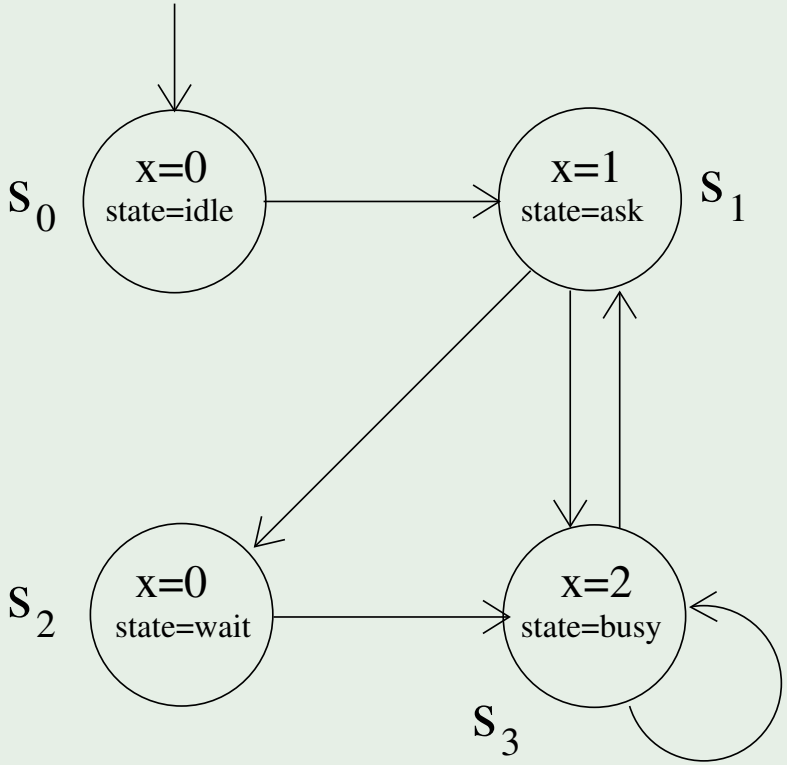
What does $\mathrm{AP}$ represent?

Let $V = \{v_1, \ldots, v_n\}$ be the set of system variables ranging over a finite set $D$ of values, called *domain*, and a valuation for $V$ be a function associating each $v$ in $V$ with a value in $D$

- An atomic proposition has the form $v = d$
- A state $s$ is a valuation $s : V \to D$ and can be expressed as a conjunction of atomic propositions
- The transition relation $R$ is a mapping from valuations to valuations and can be expressed as a disjunctive normal form

# Graphical representation

## Example




Example of Kripke structure $(S, S_0, R, L)$

- $V = \{x, s\}$
- $S_0 = \{x = 0 \wedge s = i\}$
- $R$ is such that:

$(x = 0 \wedge s = i \wedge x' = 1 \wedge s' = a) \in R$
$(x = 1 \wedge s = a \wedge x' = 0 \wedge s' = w) \in R$
$(x = 1 \wedge s = a \wedge x' = 2 \wedge s' = b) \in R$
$(x = 0 \wedge s = w \wedge x' = 2 \wedge s' = b) \in R$
$(x = 2 \wedge s = b \wedge x' = 1 \wedge s' = b) \in R$
$(x = 2 \wedge s = b \wedge x' = 2 \wedge s' = a) \in R$

- $L$ is such that:

$L(s_0) = \{x = 0 \wedge s = i\}$
$L(s_1) = \{x = 1 \wedge s = a\}$
$L(s_2) = \{x = 0 \wedge s = w\}$
$L(s_3) = \{x = 1 \wedge s = b\}$

# Kripke structures and logics

## Definition

For any Kripke structure $M = (S, S_0, R, L)$ and formula $\psi$ of any logic:

- $Sat(\psi) = \{s \in S \mid s \models \psi\}$
- $M \models \psi$ iff $S_0 \subseteq Sat(\psi)$

## Example

Note: it could be that $M \not\models \psi$ and $M \not\models \neg\psi$

# Kripke structures and logics

## Temporal logics

- Modal operators are added to classical logics for reasoning about time
- Well-formed formulas specify properties of *computations*
- The process of checking the model against the property is called model checking

## Example

Along a computation . . .

- a predicate is *always* true
- the system will *eventually* satisfy a predicate
- the system exhibits a certain behavior *until* a predicate holds

# Computation tree

## Definition

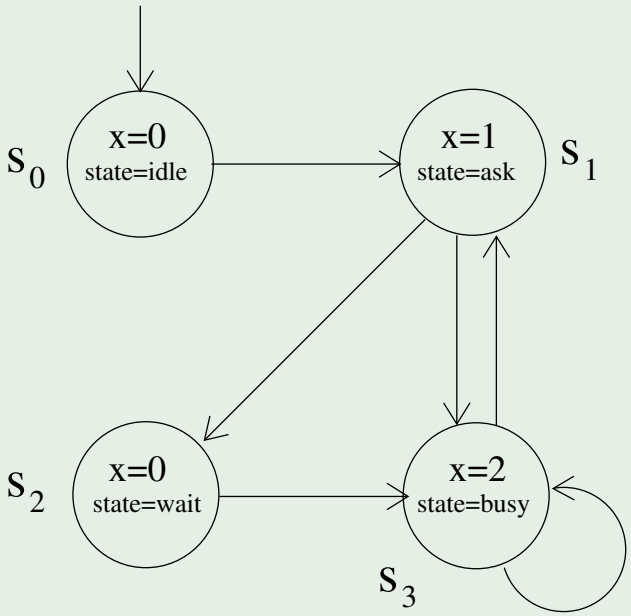
For any Kripke structure $M = (S, S_0, R, L)$ and $s \in S$, the computation tree for $M$ starting at $s$ is a tree such that:

- the root of the tree is $s$
- the nodes of the tree are states in $S$
- there exists an arc from node $s$ to node $s'$ in the tree if and only if $(s, s') \in R$

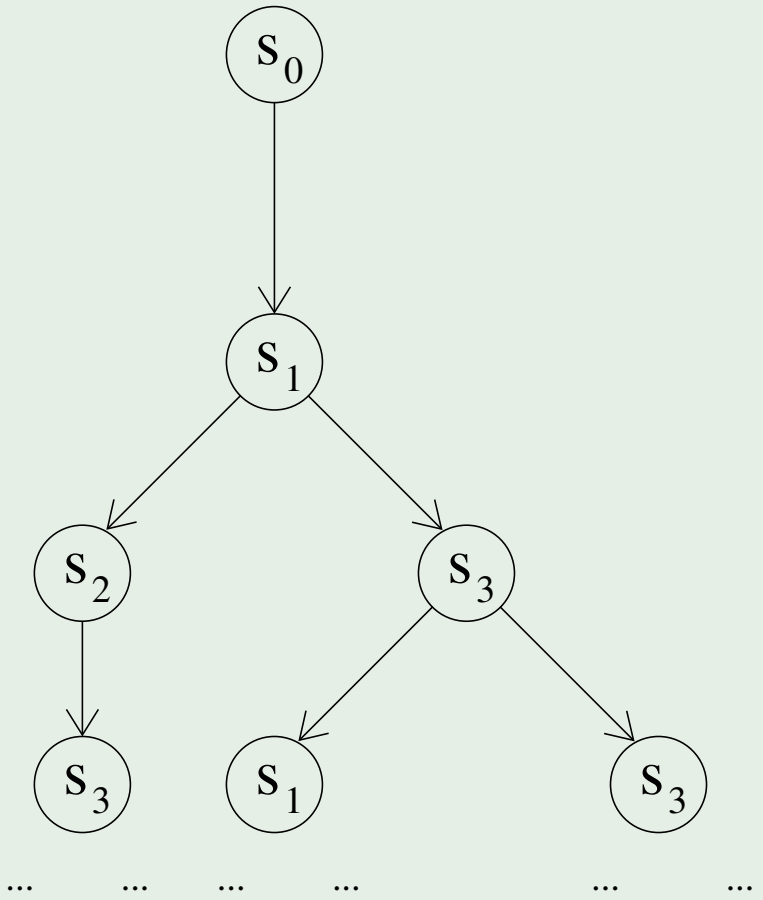
# Graphical representation

## Example

### Kripke structure




### Finite prefix of the computation tree rooted at $s_0$



# Modal operators of temporal logics

## Temporal operators

| | |
|---|---|
| **X** (*next*) | $\bigcirc \rightarrow \circledast \rightarrow \ldots$ |
| **F** (*eventually*) | $\bigcirc \rightarrow \cdots \rightarrow \bigcirc \rightarrow \circledast \rightarrow \bigcirc \rightarrow \ldots$ |
| **G** (*always*) | $\circledast \rightarrow \cdots \rightarrow \circledast \rightarrow \ldots$ |
| **U** (*until*) | $\odot \rightarrow \cdots \rightarrow \odot \rightarrow \circledast \rightarrow \bigcirc \rightarrow \ldots$ |

# Temporal logics: two flavors

## Expressiveness

- Shall the logic abstract from the branching structure of the computation tree?
- If yes, the approach is analogous to the semantics of NFA ...
- ... each computation is considered in isolation ...
- ... at each instant of time, only one future is possible (*linear time*)
- If not, a *branching* notion of time is considered ...
- at each instant of time, there may be several different possible futures

# Linear Temporal Logic: syntax

## Definition (LTL grammar)

$$\varphi \rightarrow \mathit{true} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \varphi\mathbf{U}\varphi$$

Note: $p$ is any atomic proposition

# Linear Temporal Logic: semantics

## Definition

The satisfaction relation $\models$ for computations $\pi = s_0, s_1, \ldots, s_i, \ldots$ of a Kripke structure $M = (S, S_0, R, L)$ is defined by:[a]

1. $\pi \models \mathit{true}$ always holds
2. $\pi \models p$ iff $p \in L(\pi[0])$
3. $\pi \models \neg\varphi$ iff $\pi \not\models \varphi$
4. $\pi \models \varphi_1 \vee \varphi_2$ iff $\pi \models \varphi_1$ or $\pi \models \varphi_2$
5. $\pi \models \mathbf{X}\,\varphi$ iff $\pi^1 \models \varphi$
6. $\pi \models \mathbf{F}\,\varphi$ iff $\exists k \geq 0.\, M, \pi^k \models \varphi$
7. $\pi \models \mathbf{G}\,\varphi$ iff $\forall k \geq 0.\, M, \pi^k \models \varphi$
8. $\pi \models \varphi_1\,\mathbf{U}\,\varphi_2$ iff there exists $k \geq 0$ such that $\pi^k \models \varphi_2$ and for all $0 \leq j < k$ it holds that $\pi^j \models \varphi_1$

Finally, $s \models \varphi$ iff $\pi \models \varphi$ for all $\pi$ such that $\pi[0] = s$

---

[a] Note: $\pi[i] = s_i$ and $\pi^i = s_i, \ldots$

# Computation Tree Logic: syntax

## Definition (CTL grammar)

$$\begin{array}{lcl} \varphi & \rightarrow & \mathit{true} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{E}\,\phi \mid \mathbf{A}\,\phi \\ \phi & \rightarrow & \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \varphi\mathbf{U}\varphi \end{array}$$

Note: $p$ is any atomic proposition

# Computation Tree Logic: semantics

## Definition

1. $s \models p$ iff $p \in L(s)$
2. $s \models \neg\phi$ iff $s \not\models \phi$
3. $s \models \phi_1 \vee \phi_2$ iff $s \models \phi_1$ or $s \models \phi_2$
4. $s \models \mathbf{E}\,\phi$ iff there is a path $\pi$ starting from $s$ such that $\pi \models \phi$
5. $s \models \mathbf{A}\,\phi$ iff for every path $\pi$ starting from $s$ it holds that $\pi \models \phi$
6. $\pi \models \mathbf{X}\,\phi$ iff $\pi[1] \models \phi$
7. $\pi \models \phi_1\,\mathbf{U}\,\phi_2$ iff there exists $k \geq 0$ such that $\pi[k] \models \phi_2$ and for all $0 \leq j < k$ it holds that $\pi[j] \models \phi_1$

# Temporal Logics: properties of operators

## Expressiveness

$$\begin{aligned} \mathbf{F}\,\psi &\equiv \mathtt{true}\,\mathbf{U}\,\psi \\ \mathbf{G}\,\psi &\equiv \neg\mathbf{F}\,\neg\psi \\ \mathbf{A}\,\psi &\equiv \neg\mathbf{E}\,\neg\psi \end{aligned}$$

## Duality

$$\begin{aligned} \neg\mathbf{X}\,\psi &\equiv \mathbf{X}\,\neg\psi \\ \neg\mathbf{F}\,\psi &\equiv \mathbf{G}\,\neg\psi \\ \neg\mathbf{G}\,\psi &\equiv \mathbf{F}\,\neg\psi \end{aligned}$$

# Temporal Logics: equivalence

## Definition

A CTL formula $\varphi$ is equivalent to a LTL formula $\varphi'$ ($\varphi \equiv \varphi'$) if for all Kripke structures $M$:

$$M \models \varphi \textit{ iff } M \models \varphi'$$

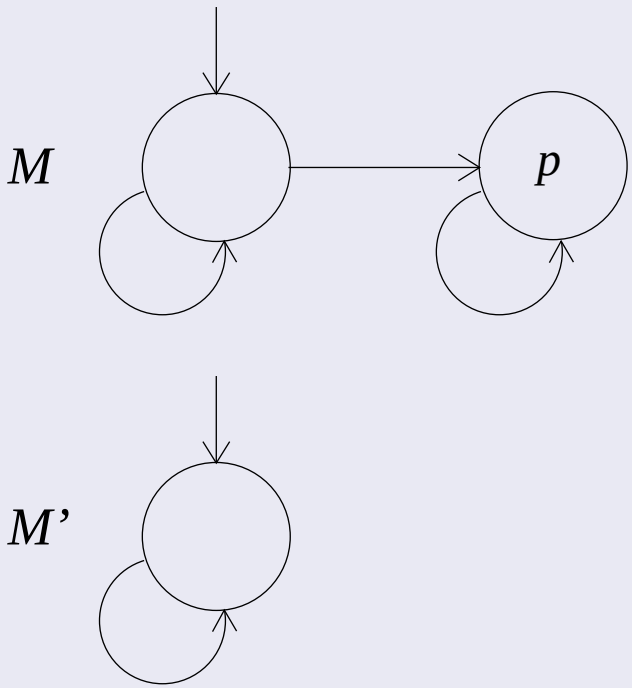
## Results about equivalence

- Given a CTL formula $\varphi$ and the LTL formula $\varphi'$ obtained by eliminating any path quantifier from $\varphi$, we have:
  1. either $\varphi \equiv \varphi'$
  2. or there does not exist any LTL formula equivalent to $\varphi$
- CTL and LTL are incomparable:
  1. the CTL formula $\mathbf{A}\,\mathbf{G}\,(\mathbf{E}\,\mathbf{F}\,p)$ cannot be expressed in LTL
  2. the LTL formula $\mathbf{F}\,\mathbf{G}\,p$ cannot be expressed in CTL

# Why LTL cannot express $\mathbf{A}\,\mathbf{G}\,(\mathbf{E}\,\mathbf{F}\,p)$

## Proof.

Assume, by contradiction, $\varphi \equiv \mathbf{A}\,\mathbf{G}\,(\mathbf{E}\,\mathbf{F}\,p)$ such that $\varphi$ is a LTL formula




- $M \models \mathbf{A}\,\mathbf{G}\,(\mathbf{E}\,\mathbf{F}\,p)$, hence $M \models \varphi$
- Hence, since the computations of $M'$ are a subset of those of $M$, it must hold $M' \models \varphi$, which is absurd!

# Property specification

## Some formula schema

| | LTL | CTL |
|---|---|---|
| Reachability | $M \not\models \mathbf{G}\,\neg p$ | $M \models \mathbf{E}\,\mathbf{F}\,p$ |
| Unreachability (Safety) | $\mathbf{G}\,\neg p$ | $\mathbf{A}\,\mathbf{G}\,\neg p$ |
| Conditional reachability | $q\,\mathbf{U}\,p$ | $\mathbf{A}\,(q\,\mathbf{U}\,p)$ |
| Extended reachability | No | $\mathbf{A}\,\mathbf{G}\,\mathbf{E}\,\mathbf{F}\,p$ |
| Composite reachability | $\mathbf{F}(q \wedge \mathbf{X}\,p)$ | No |
| Liveness | $\mathbf{G}(q \rightarrow \mathbf{F}\,p)$ | $\mathbf{A}\,\mathbf{G}(q \rightarrow \mathbf{A}\,\mathbf{F}\,p)$ |
| Unconditional fairness | $\mathbf{G}\,\mathbf{F}\,p$ | $\mathbf{A}\,\mathbf{G}\,\mathbf{A}\,\mathbf{F}\,p$ |
| Strong fairness | $\mathbf{G}\,\mathbf{F}\,p \rightarrow \mathbf{G}\,\mathbf{F}\,q$ | No |
| Weak fairness | $\mathbf{F}\,\mathbf{G}\,p \rightarrow \mathbf{G}\,\mathbf{F}\,q$ | No |

# References


- Christel Baier and Joost-Peter Katoen: Principles of Model Checking, The MIT Press, 2008
- Michael Huth and Mark Ryan: Logic in Computer Science - Modelling and Reasoning about Systems, Cambridge University Press, 2004