Formal Methods II:
Automata Theory

October 4, 2013

Rolf Pfeifer

Rudolf M. Füchslin

Zurich University of Applied Sciences

School of Engineering

ZAMP Center for Applied Mathematics and Physics

Recap

Application: Regular Expressions

- **Regular expression mechanism** accept a syntax describing patterns and contain a mechanism that filters matching sequences.
- Originally, the underlying mechanism were based on regular languages.
- In the next lecture, we learn why: strings produced by a regular grammar can by recognized by a simple automata model, which is easy to implement.
- **WARNING:** There are several, slightly different “flavors” of regexp: file globbing à la grep is different from POSIX regexp used in vi, emacs, awk. lex.

Most Common POSIX regexp Operators

- Operators are metacharacters. If they have to be used literally, use a backslash \.

Operator	Stands for
+	At least one occurrence of preceding symbol
*	Zero , one or more occurrence of preceding symbol (NOT like in file globbing)
?	Zero or one occurrence of preceding symbol
|	Separates alternative choices
()	Grouping
\	The next symbol literally (escaping)
.	Any single character (NOT like in file globbing)

Common Operators Regular Expression

Regular Expression	Elements of the Language
abc	abc
a*bc	bc, abc, aabc, aaabc, aaaabc, ...
go+gle	gogle, google, gooogle, ...
pfeiff?er	pfeifer, pfeiffer
pf(a|e)ifer	pfaifer, pfeifer

Used in many UNIX/LINUX – like environments.
NOTE: e.g. grep uses file globbing.

Context Free Languages

Context Free Languages: Motivation

- A program:

```
main() {
  printf("Hello world!");
  return;
}
```

- A bigger program:

```
main() {
  printf("Hello world!\n");
  printf("It's Friday.\n");
  return;
}
```

Context Free Languages: Motivation

- Programming: Replacing SLOT's by content.

  func0(SLOT_A, SLOT_A)

  → func0(func1(SLOT_B), SLOT_A))

  → func0(func1(x), SLOT_A)

  → func0(func1(x), func2(SLOT_A))

  → ….
- SLOTs can be occupied by other expressions, **independent of the surrounding**.
- There may be several types of SLOTs.

Definition: Context Free Grammar

A context-free language is constructed by a context free grammar, which is defined by having replacement rules of the form

$$\alpha \rightarrow \beta$$

$$\alpha \in V$$

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

Context Free Languages: Examples

Example 2.5. A simple context-free grammar is

$$S \to aSb \mid \epsilon$$

It generates the context-free language

$$L = \left\{ a^n b^n \;\middle|\; n \geq 0 \right\}$$

which consists of all strings that contain some number of a's followed by the same number of b's: $L = \{\epsilon, ab, aabb, aaabbb, \ldots\}$.

Context Free Languages: Examples

To illustrate this, consider the following context-free grammar

$$S \rightarrow aS \mid S(S)S \mid \epsilon$$

which generates strings over the alphabet $\Sigma = \{a, (,)\}$ with any number of correctly balanced parenthesis: $(a), ((aa)aaa)a(a)(), \ldots$

Now, we know that context free grammars can construct all paranthesis-balanced expressions. The equivalence between grammars and automata will be exploited for the construction of such a system.

Describing Context Free Languages

Backus-Naur Form

- BNF is metalanguage for the description of context-free languages (almost all programming languages are context free, possible exception: HASKELL)
- The Extended Backus-Naur Form was invented by Niklaus Wirth. EBNF is more handy than BNF, but otherwise of equivalent power of expression.

BNF and EBNF

- An BNF description is an unordered list of BNF rules.
- An BNF-rule consists of a LHS, a “=“ and a RHS. A rule ends with an end-of-line character
- The “::=“ reads as “is defined as”.
- Non-terminals are enclosed in brackets <>.
- A vertical bar indicates a exclusive choice.
- Sequences of characters between “” have to be taken literally.
  - A single literal double quote is expressed as ‘”’
  - A single literal quote is expressed as “’”.

Postal Address in BNF

Non-terminal		
<address>	::=	<name> <street> <city>
<name>	::=	<personal-part> <surname> <EOL> | <personal-part> <name> <EOL>
<personal-part>	::=	<first-name> | <initial> “.”
<street>	::=	<street-name> <number> <EOL>
<city>	::=	“CH-”<zip-code> <city-name> <EOL>

What is lacking?

1. Some non-terminals are not defined (e.g. first-name)
2. <EOL>
3. Treatment of white spaces
4. Titles

BNF in BNF

<syntax> ::= <rule> | <rule> <syntax>

<rule> ::= <opt-whitespace> "<" <rule-name> ">"

<opt-whitespace> "::=" <opt-whitespace> <expression> <line-end>

<opt-whitespace> ::= " " <opt-whitespace> | "" <!-- "" is empty string, i.e. no whitespace -->

<expression> ::= <list> | <list> "|" <expression>

<line-end> ::= <opt-whitespace> <EOL> | <line-end> <line-end>

<list> ::= <term> | <term> <opt-whitespace> <list>

<term> ::= <literal> | "<" <rule-name> ">"

<literal> ::= "" <text> "" | """ <text> """ <!-- actually, the original BNF did not use quotes -->

Extended Backus-Naur Form

- Initally introduced by Wirth, based on WSN (Wirth‘s syntax notation).
- Easier to handle than BNF
- No hyphens, no brackets.
- Six special symbols: =,|,[,],{,}
- [...] options
- {...} repetitions

Example: Integers in EBNF

- Integers: 23, -3454, +453223

digit	=	0 | 1 | 2| 3 | 4 | 5 | 6 | 7 | 8 | 9
integer	=	[+|-] digit {digit}

nonzero	=	1 | 2| 3 | 4 | 5 | 6 | 7 | 8 | 9
digit	=	0 | nonzero
integer	=	0 | [+|-] nonzero {digit}

Parsing: From ASCII - Strings to Executables

Zurich University
of Applied Sciences

School of
Engineering

ZAMP Center for Applied
Mathematics and Physics

Parsing

- EBNF / BNF gives syntax for textfiles representing programs.
- Computer can‘t use text files for executing functions.
- What computers can digest are function calls.

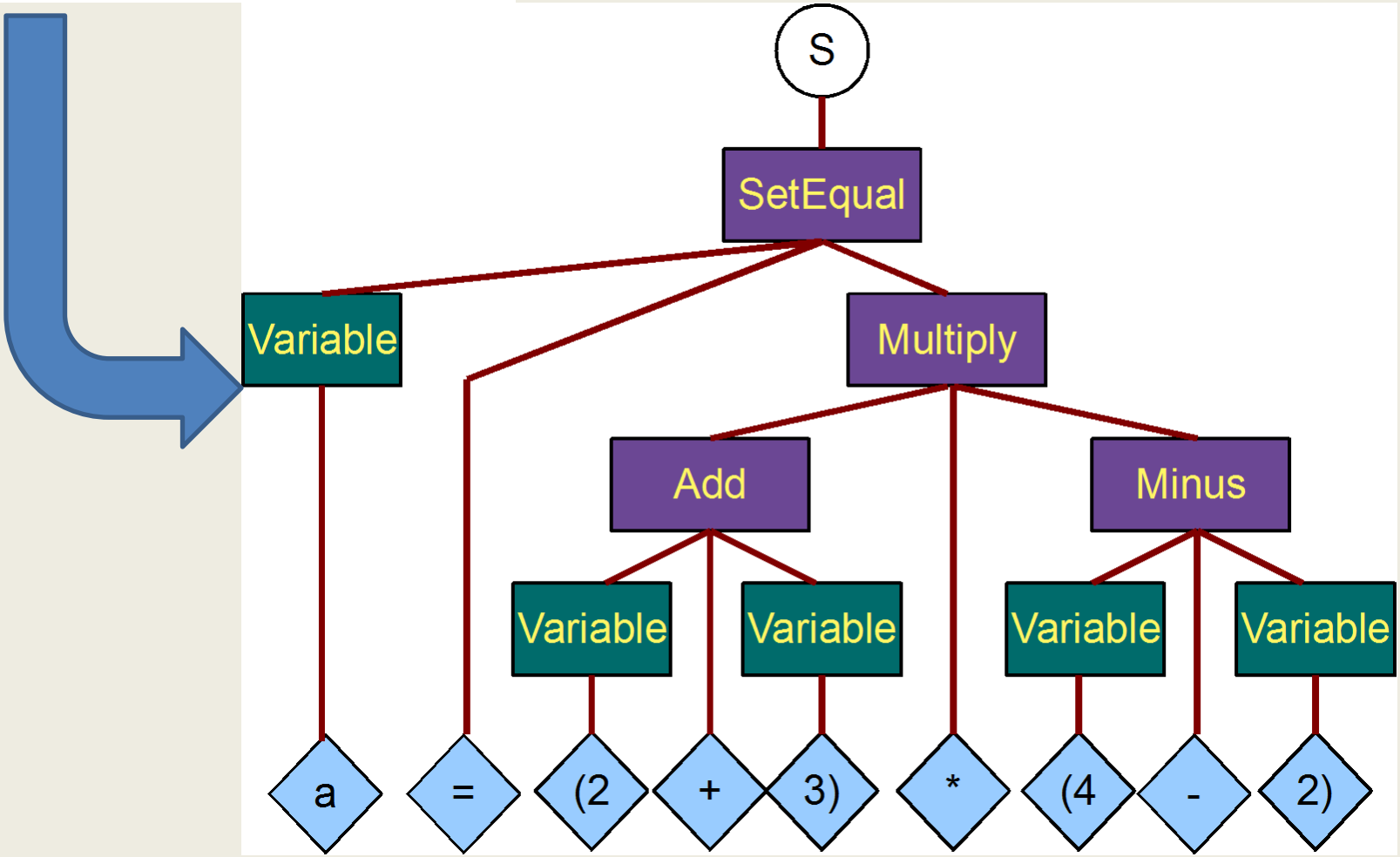
The idea is to use the syntax tree of a sequences for calling internal functions. A **parser** is a program that transforms a sequence in syntax tree.

Remark: Parsers as Syntax Checkers

- Sometimes, one understands by the term parser a syntax checker.
- Today, CPU-time and memory is cheap.
- Checking a program for syntactical correctness is the main task. Constructing an according calling tree is more or less for free. ➔ Parsers include calling – tree generators.

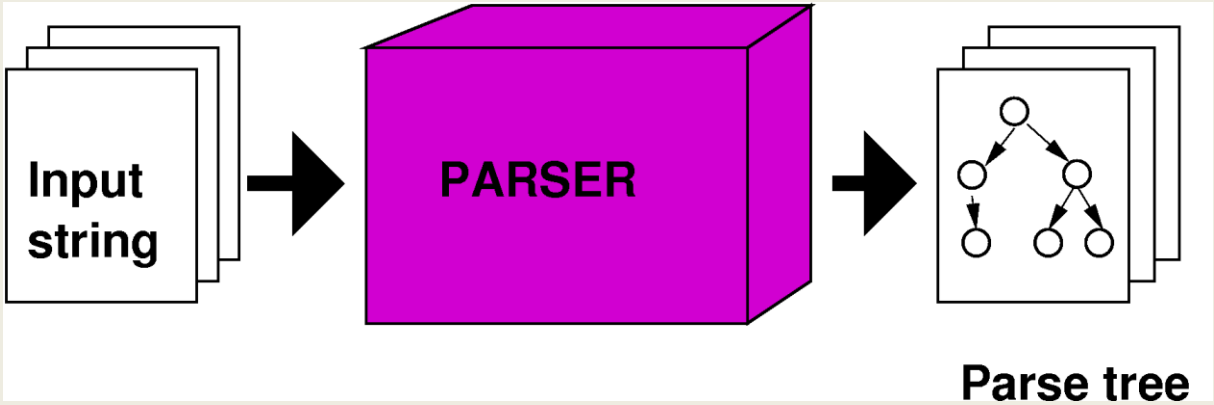
Parsing: From String to Tree

a = (2 + 3) * (4 - 2)



Parsing Programs

- Programming languages are most often context-free languages.
- Their syntax trees can be understood as receipts how to call nested functions.
- A **parser** is a program that **takes a string and reconstructs its syntax tree**. The syntax tree is used for organizing the execution of the program.



Füchslin‘s Simple Lisp (FSLISP)

Non terminals		
expression	=	function | integer
function	=	(functionname , epression , expression)
functionname	=	prod | sum
digit	=	0 | 1 | 2| 3 | 4 | 5 | 6 | 7 | 8 | 9
integer	=	[+|-] digit {digit}

FSLISP represents sums and products of integers.

Examples:

- (prod, 4, 5)
- (sum, (prod, 3, 4), (prod, (sum, 2, 3), 7))

FSLISP Interpreter

Assumption: **`string`** is of the form **`integer`** or **`(string1, string2, string3)`**

```
String1(string) {
returns string1
} ;
String2(string) {
returns string2
} ;
String3(string) {
returns string3
} ;
```

```
IsNumber(string) {
if string is an
integer, return 1
else return 0
}
ToNumber(string) {
converts string in
integer
}
```

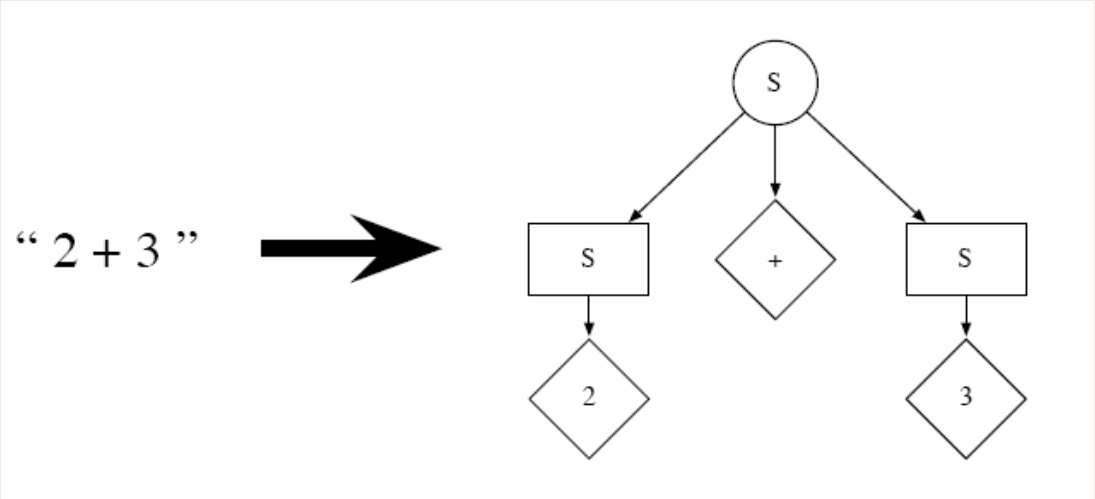
FSLISP Interpreter

```
FSLISP_ip(string) {
If IsNumber(string) == 1,
  return ToNumber(string);
string1 = String1(string);
string2 = String2(string);
string3 = String3(string);
If string1 == „prod“,
  return FSLISP_ip(string2) *
     FSLISP_ip(string3) ;
If string1 == „sum“,
  return FSLISP_ip(string2) +
     FSLISP_ip(string3);
}
```

Recursive calls realize parse tree!

Pitfalls in Parsing

$$
\begin{aligned}
S &\rightarrow S + S \mid S - S \mid S \times S \mid N \\
N &\rightarrow 2 \mid 3 \mid 4
\end{aligned}
$$

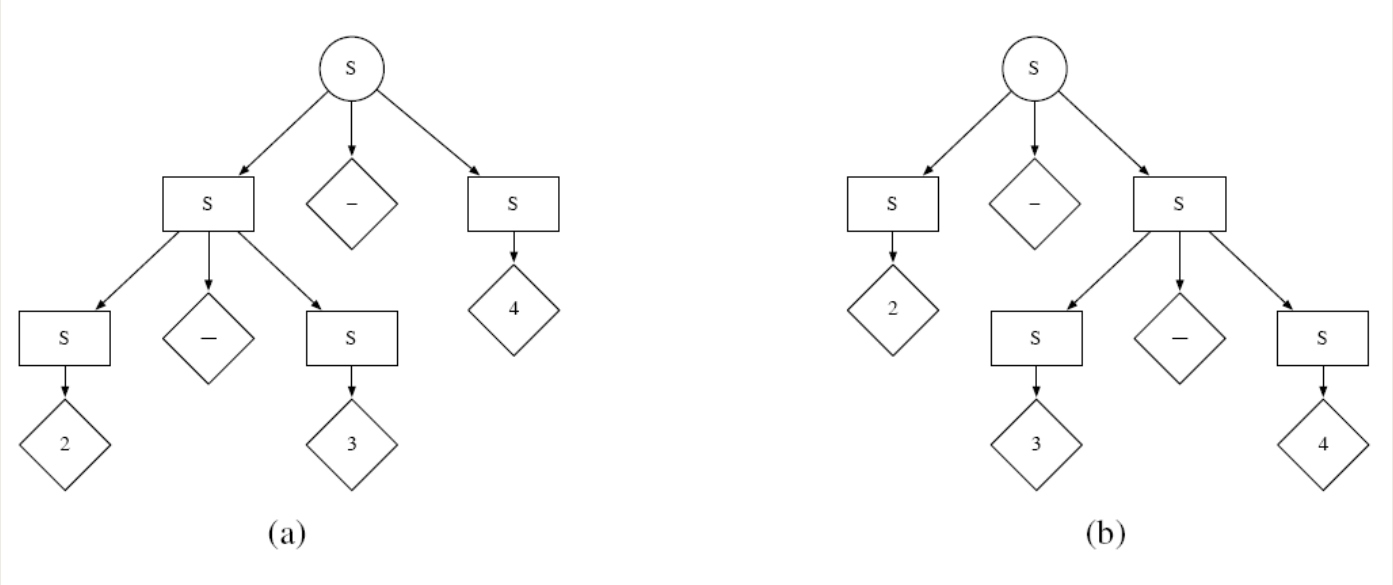


Harmless, easy to understand.

Pitfalls in Parsing: The String 2-3-4

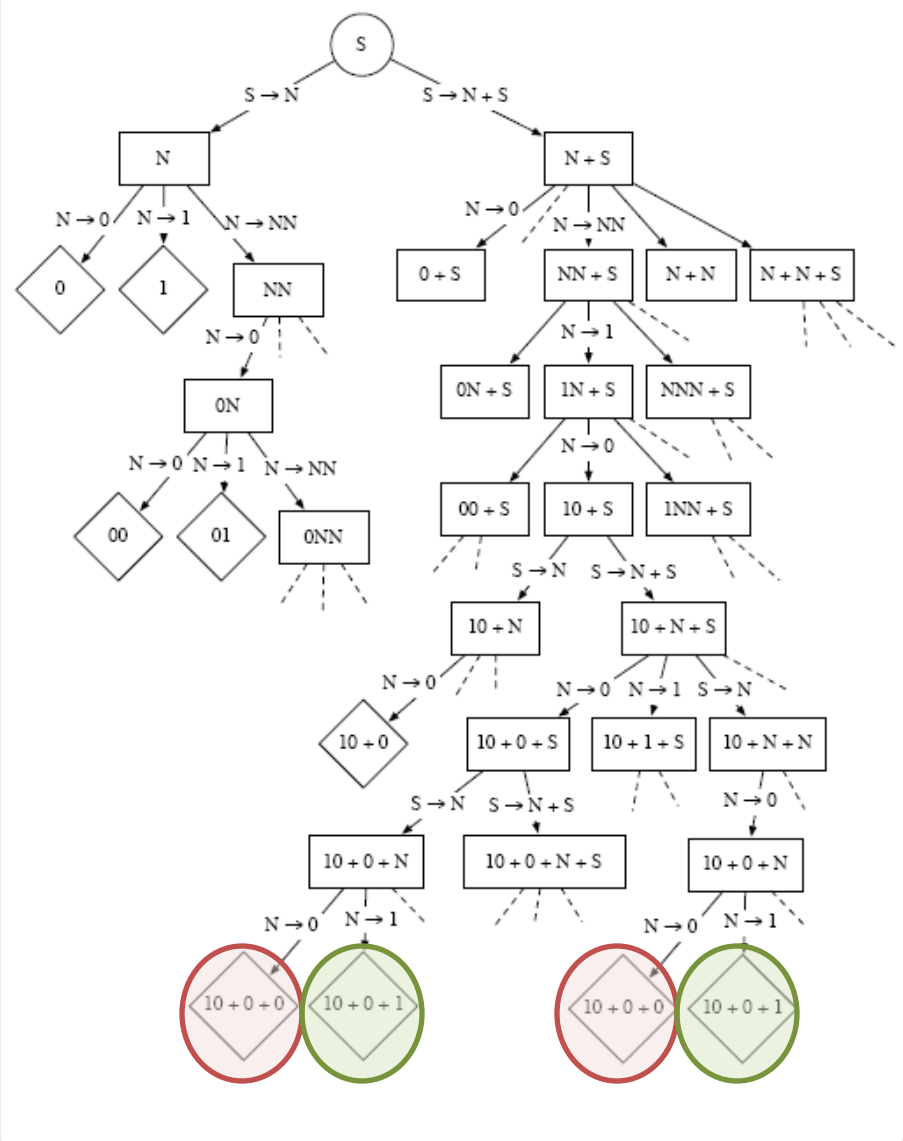
$$(2-3)-4=-5$$

$$2-(3-4)=3$$



One string can have several syntax trees. Which one shall one choose?

Ambiguities in Grammar Trees



$$\Sigma : \{0, 1, +\}$$

$$V : \{S, N\}$$

Start symbol: S

$$S \rightarrow N \mid N + S$$

$$N \rightarrow 0 \mid 1 \mid NN$$

Definition: Left-most Derivation

Definition: Left-most derivation: Always apply replacement rules to the left-most non-terminal symbol. Right-most derivation is defined accordingly.

Ambiguouis Grammars

Definition A: A grammar is said to be **ambiguous** if the language it generates contains some sequence that has more than one possible syntax tree.
Focus is on syntax trees.

Definition B: A grammar is said to be **ambiguous** if the language it generates contains some sequence that can be derived by more than one leftmost derivations.
Focus on derivations.

Ambiguity by Multiple Derivation

Focusing on derivations in definition of ambiguity:

- Disadvantage: It is a bit more strict than absolutely necessary.

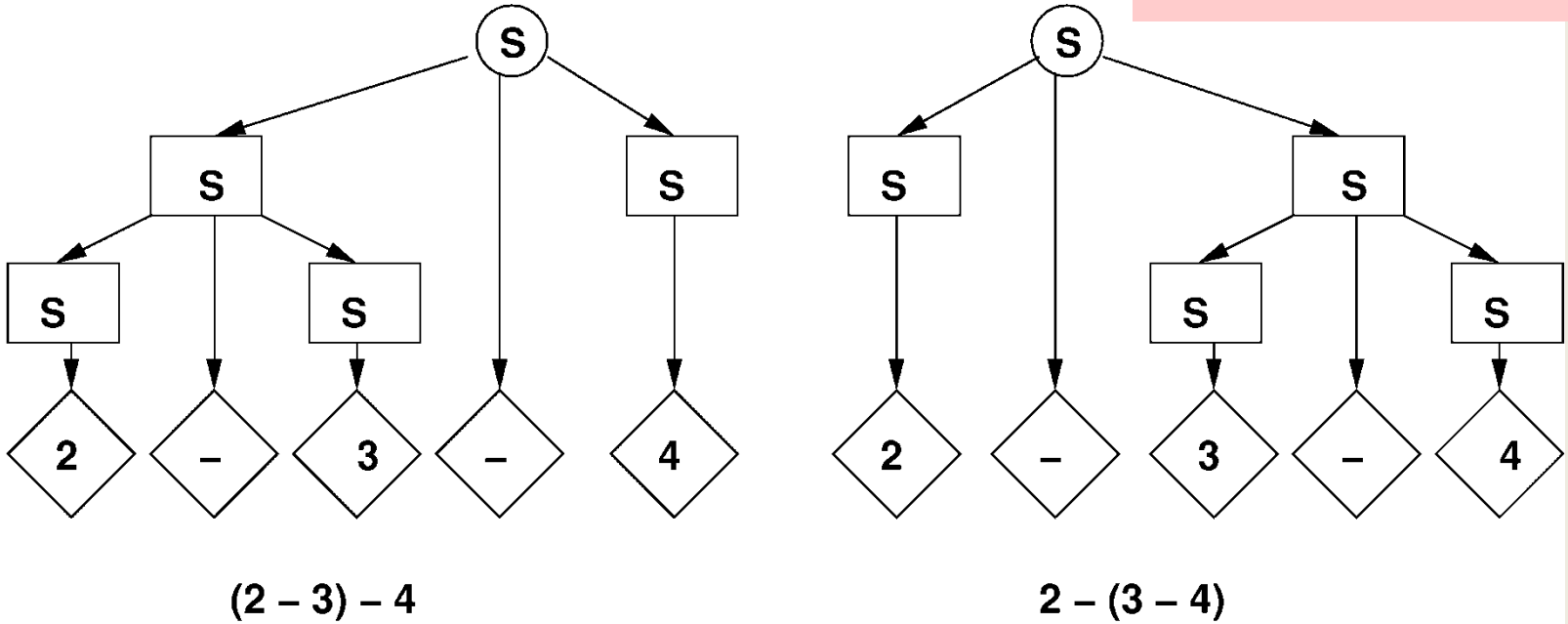
Note: In principle, it wouldn’t be a problem if multiple derivations lead to the same sequence, as long as their according syntax trees are functionally equivalent. But it is hard to say whether two syntax trees are functionally equivalent.

There Are Still Problems

S ➔ S – S ➔ (S – S) – S ➔ (2 – S) – S ➔ (2 – 3) – S ➔ (2 – 3) – 4
S ➔ S – S ➔ 2 – S ➔ 2 – (S – S) ➔ 2 – (3 – S) ➔ 2 – (3 – 4)

Both derivation are „left-most“, but nevertheless different.

"2–3–4" has two possible parse trees



$S \rightarrow S + S \mid S - S \mid S \times S \mid N$
$N \rightarrow 2 \mid 3 \mid 4$

Removing Ambiguities

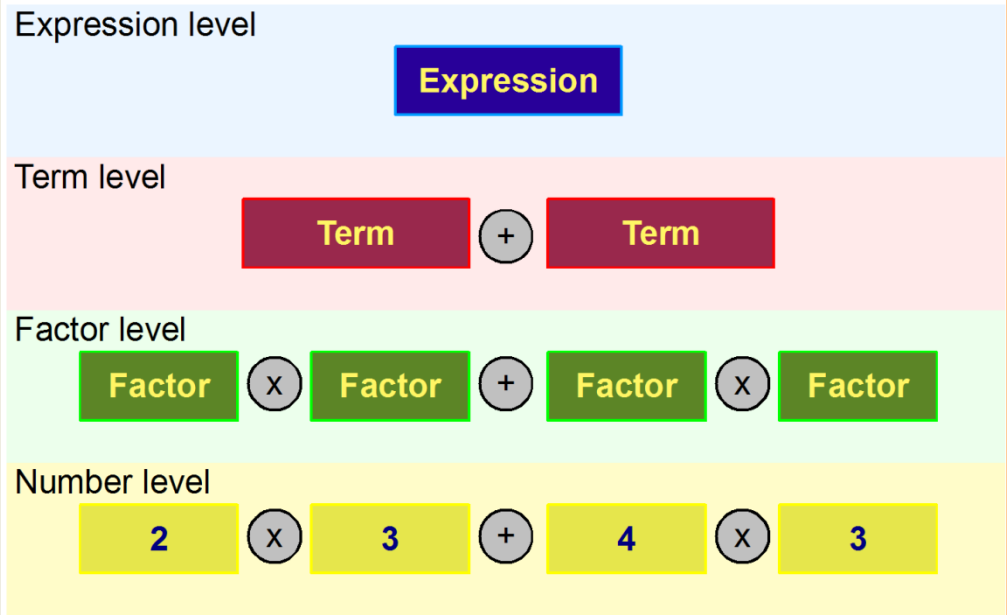
- Bad news first: Whether a context free grammar is unambiguous or not is in general an unsolvable problem.
- Amazing fact: There are context-free grammars which have only ambiguous grammars.
- There is a bunch of techniques working in specific cases.

Techniques for Removing Ambiguities

- In usual arithmetic, ambiguities are removed by requiring
  - Left associativity
  - Precedence *, / over +,-
- Solution: Introduce additional symbols that reflect the binding strength of different elements of the string.

$$S \rightarrow S + S \mid S - S \mid S \times S \mid N$$
$$N \rightarrow 2 \mid 3 \mid 4$$

Techniques for Removing Ambiguities



$$S \rightarrow E$$

$$E \rightarrow T \mid E + T \mid E - T$$

$$T \rightarrow F \mid T \times F$$

$$F \rightarrow N$$

$$N \rightarrow 2 \mid 3 \mid 4$$

- Each string in the language is an expression E.
- Each expression consists either of a term T or the sum or the difference of an expression and a term T.
- Each term consist of one or more factors.

End Recap

Beyond Context-Free Languages

Zurich University
of Applied Sciences
School of
Engineering
ZAMP Center for Applied
Mathematics and Physics

Context-Sensitive Languages

An in-between of context-free and unrestricted grammars are **context –sensitive grammars**.

Definition: A context-sensitive grammar has rules of the form:

$$\alpha \to \beta$$

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

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

$$|\alpha| \le |\beta|$$

$$S \to \varepsilon \text{ is allowed, if } S \text{ doesn't appear on any RHS.}$$

Rarely used, |a| <= |b| for proofs.

Example: Context-Sensitive Language

Example 2.9. A typical example of context-sensitive language that is not context-free is the language

$$L = \{a^n b^n c^n \mid n > 0\}$$

which can be generated by the following context-sensitive grammar:

$$\begin{aligned} S &\rightarrow aAbc \mid abc \\ A &\rightarrow aAbC \mid abC \\ Cb &\rightarrow bC \\ Cc &\rightarrow cc \end{aligned}$$

aaaabCbCbCbc ➔ aaaabbCCbCbc ➔ …. ➔
aaaabbbbCCCc ➔ ….. ➔ aaaabbbbcccc

Unrestricted Grammars

Definition: In an unrestricted grammar, the replacement rules have the form

$$\alpha \to \beta$$

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

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

$$\alpha \neq \varepsilon$$

Chomsky Classification

Grammar	Rules	Examples
Type 0 Unrestricted	$\alpha \to \beta$ $a \in (\Sigma \cup V)^* V (\Sigma \cup V)^*$ $\beta \in (\Sigma \cup V)^*$	
Type 1 Context-sensitive	$\alpha \to \beta$ $a \in (\Sigma \cup V)^* V (\Sigma \cup V)^*$ $\beta \in (\Sigma \cup V)^*$ $\lvert\alpha\rvert \le \lvert\beta\rvert$ $S \to \varepsilon$ if S doesn't appear on any RHS	$\{a^n b^n c^n\}$
Type 2 Context-free	$A \to \beta$ $A \in V, \beta \in (\Sigma \cup V)^*$	$\{a^n b^n\}$
Type 3 Regular	$A \to \alpha B$ $A, B \in V$ $\alpha \in \Sigma^*$	$\{a^m b^n\}$

Chomsky - Hierarchy

$$L_{regular} \subset L_{contextfree} \subset L_{contextsensitive} \subset L_{unrestricted}$$

Why Working With Simple Grammars?

Depending on the grammar type, important questions can be answered:

- **Recognition problem** Given a string w and a grammar G. is w in L(G)?
- **Emptiness problem:** Given G, is L(G)=Ø? That is not trivial, because we have to determine, whether a given set of rules terminates.
- **Equivalence problem:** Given G1 and G2. Is L(G1)=L(G2)?
- **Ambiguity problem:** Is G ambiguous?

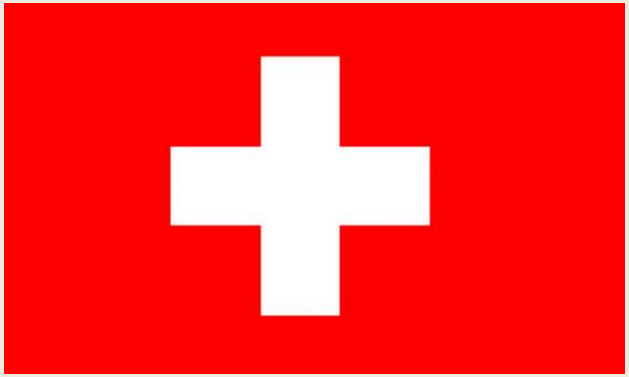
Languages and Problems

Type	Recognition	Emptiness	Equivalence	Ambiguity
0	no	no	no	no
1	yes	no	no	no
2	Yes	yes	no	no
3	yes	Yes	yes	yes

- Though not being solvable in general, a problem may well find a solution in specific cases.
- We see, why we don't use unrestricted grammars: We could not even check whether a sequence is syntactically correct (recognize as element of the language).
- Again: One has to find the equilibrium between expressiveness and ability to answer important question.

Context-Sensitive Natural Languages

A language proven to be context-sensitive is **Swiss German**



- Schieber, Stuart M. „Evidence against the context freeness of natural language", Linguistics and Philosophy 8, p. 333-343 (1985).
- W. Petersen, http://user.phil-fak.uni-duesseldorf.de/~petersen/slides/complexity_PETERSEN02.pdf

Example for a context – sensitive construction: "Mer bliibe deheime, wel mer d'Chind em Hans sis Huus lönd hälfe aaschtriiche.

Automata Theory

Zurich University
of Applied Sciences

School of
Engineering

ZAMP Center for Applied
Mathematics and Physics

Definition: Binary Computation

A **binary computation** is a mapping from a set $X \subset \{0,1\}^*$ to a set $Y \subset \{0,1\}^*$.

Mappings of the form

$$X \subset \Sigma^* \rightarrow Y \subset \Sigma^*$$

For a finite Σ can be translated into binary computations.

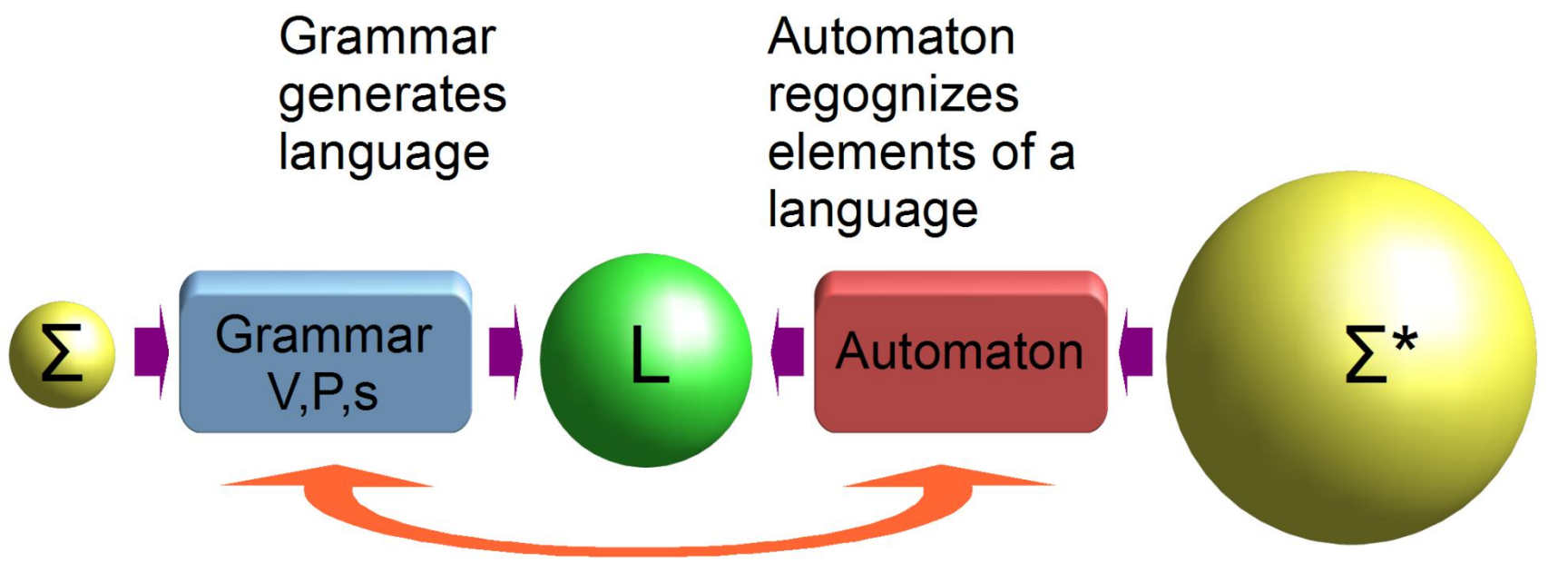
From a theoretical perspective, it is sufficient to study decision problems:

$$X \subset \{0,1\}^* \rightarrow \{0,1\}$$

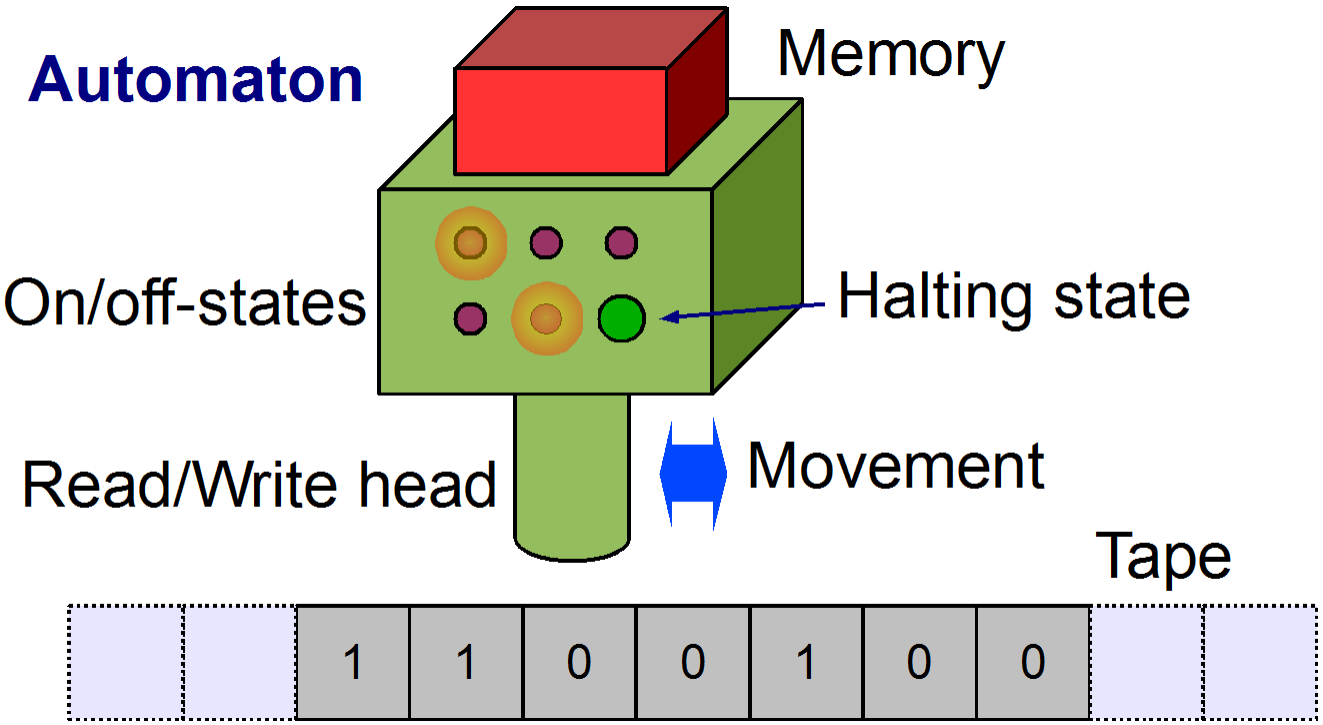
All binary computations can be reduced to sets of questions with yes/no-answer.

Grammars and Automata

- In what follows, we discuss different automata recognizing the elements of different types of languages.

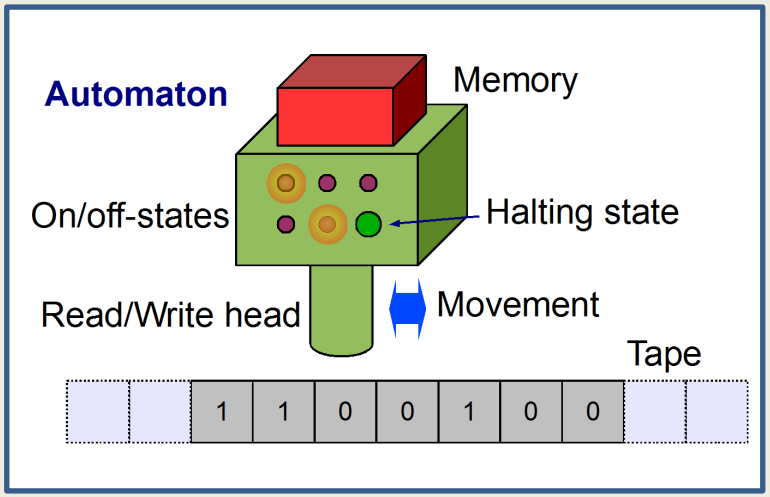


General Automata Paradigm



General Automata Paradigm

Informal: A general automaton is a machine that reads from and writes on a tape.
Restrictions on the tape (both sides infinite or bounded on one side), the memory and the states define subclasses of automata.
Again, the more severe the restrictions, the more can be proven and the less powerful is the machine!



Finite State Automata

The Power of FSA

A regular language

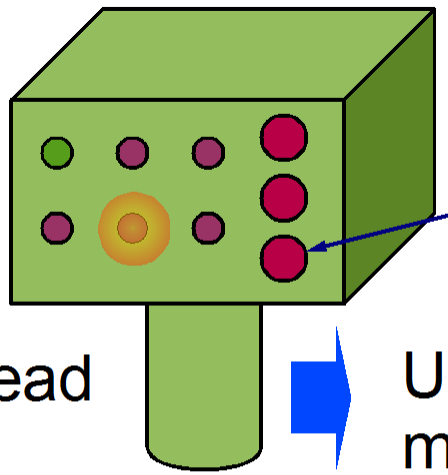
- can be generated by a regular grammar
- can be described by a regular expression
- can be accepted by a deterministic finite state machine.
- can be accepted by a non-deterministic finite state machine

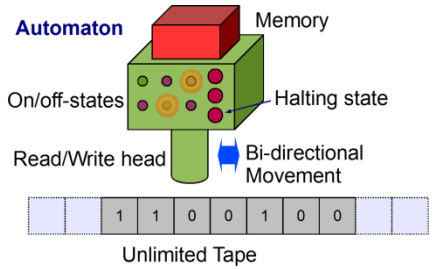
FSA

- Problem: Is there a string “abc” in a given substring?
- Or to state it in a slightly different way: If L is the language of all strings containing abc as substring, is a given string u then an element of L?

FSA

Finite State Automaton





0	1	1	1	0	0	1	0	0	End

Static tape, finite sequence.

- Only one state active
- Tape closed on one side
- No writing
- Only moves to the right
- No memory
- Finite input

FSA: Formal Definition

Definition: A **finite state automaton** is a five-tuple

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

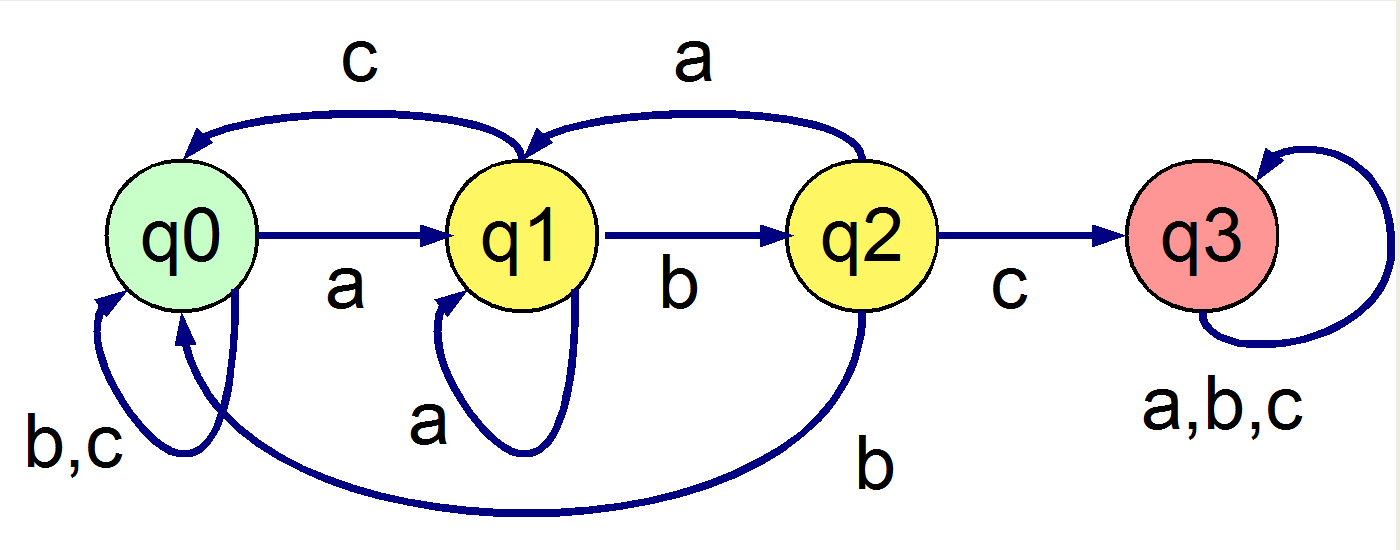
with

1. Q: A finite set of states
2. Σ: A finite set of input symbols.
3. δ: A transition function defining for each pair of state and input signal a successor state

$$\delta : (q, s) \in Q \times \Sigma \rightarrow q' \in Q$$

4. $q_0 \in Q$: An initial state
5. $F \subset Q$: A set of accepting states

FSA: Diagram



	a	b	c
q0	q1	q0	q0
q1	q1	q2	q0
q2	q1	q0	q3
q3	q3	q3	q3

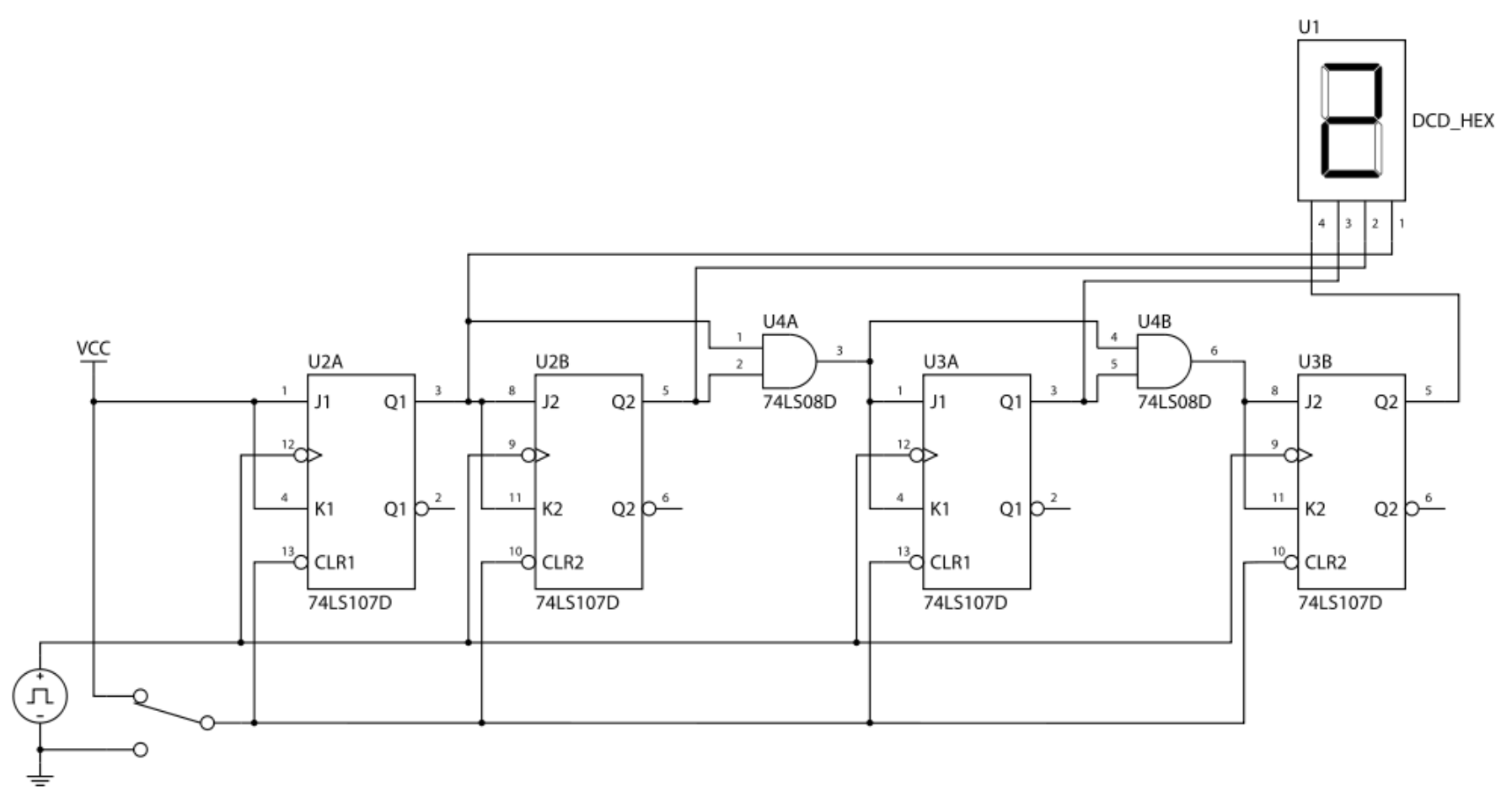
- Nodes reflect states
- Symbols on edges reflect inputs.
- Green: Input state
- Red; Accepting state

What Is Cool About FSA?

- They can „digest“ an input string and don‘t have to go back ➔ they can check an incoming stream of data online, without the necessity to store the string.
- Technically, they can be implemented by a couple of components such as
  - flip-flops,
  - logical gates,
  - multiplexers and, for technical reasons,
  - amplifiers („fan out“ of signals).

These components have been around since the start of the 20th century.

A Binary Counter



Machines With Output: Mealy - Machines

Definition: A **Mealy – machine** is a finite state automaton given by a seven-tuple (Q, Σ, Λ, δ, G, q_0, F) with

1. Q: A finite set of states
2. Σ: A finite set of input symbols.
3. **Λ: A finite set of output symbols**
4. δ: A transition function defining for each pair of state and input signal a successor state

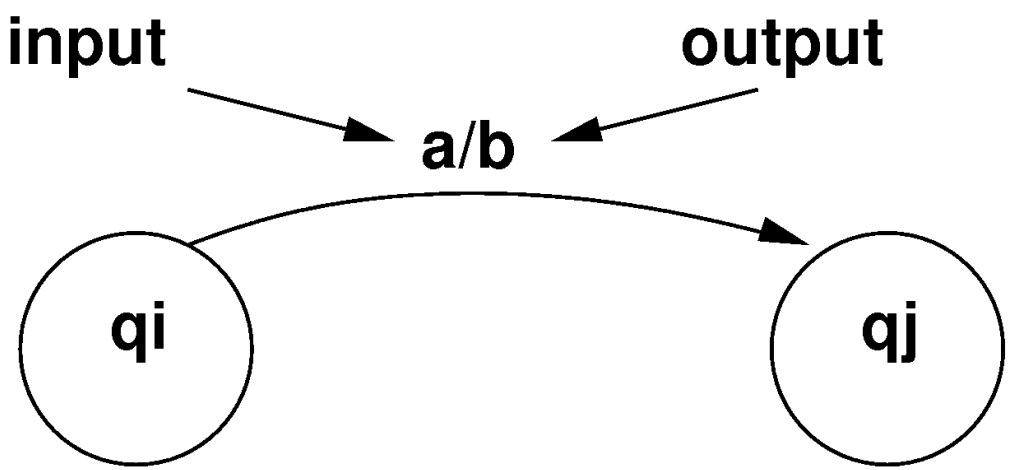
$$\delta : (q,s) \in Q \times \Sigma \rightarrow q' \in Q$$

5. **G: An output function**

$$G : (q,s) \in Q \times \Sigma \rightarrow g \in \Lambda$$

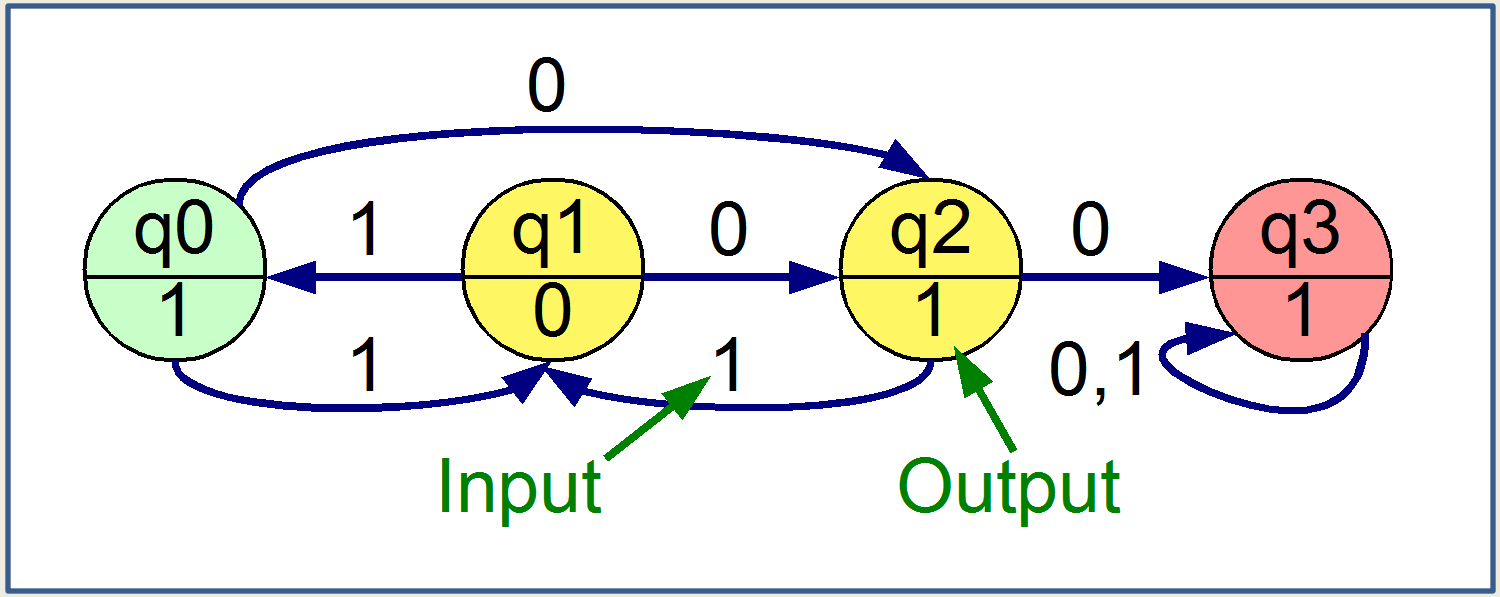
6. $q_0 \in Q$: An initial state
7. $F \subset Q$: A set of accepting states

Mealy - Machines

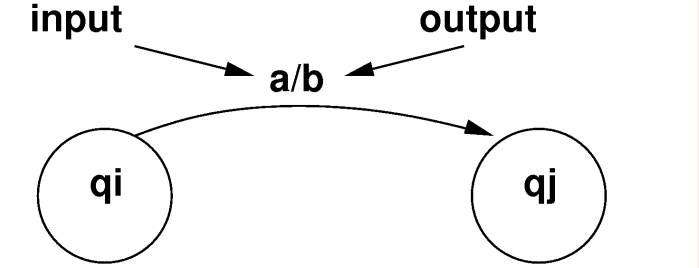


Mealy machine: output is a function of state and input.

Machines with Output: The Moore - Machine

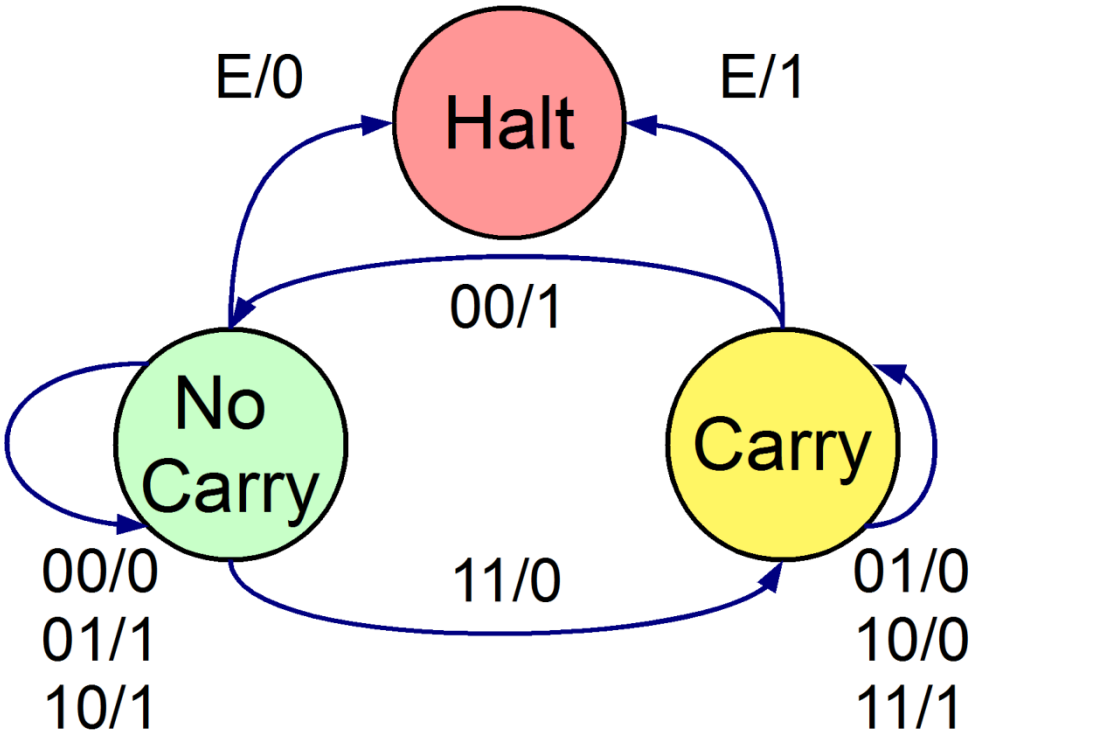


Moore – machine: output is a function of state alone.



Mealy machine: output is a function of state and input.

Mealy - Adder



1	0	0	1	E
0	1	0	1	E

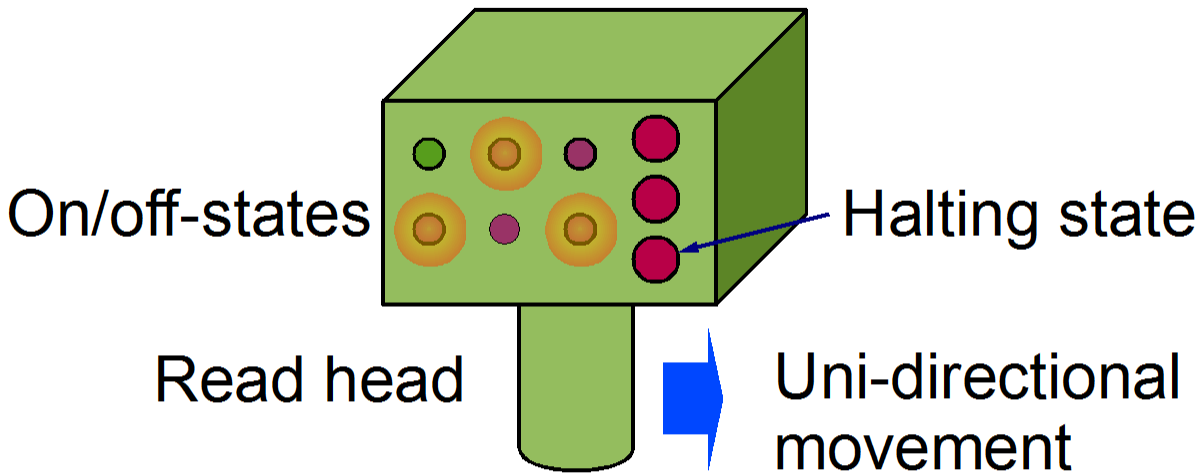
10	01	00	11	E

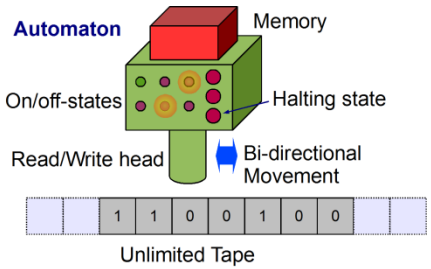
$\Sigma = \{00, 01, 10, 11, E\}$

Non - Deterministic FSA

Non – Deterministic FSA

Non-Deterministic Finite State Automaton





0	1	1	1	0	0	1	0	0	0	End

Static tape, finite sequence.

- **Many states active**
- Tape closed on one side
- No writing
- Only moves to the right
- No memory
- Finite input

Non – Deterministic FSA: Formal Definition

Definition: A **non – deterministic finite state automaton** is a five-tuple

$$(Q, \Sigma, \Delta, q_0, F)$$

with

1. Q: A finite set of states
2. Σ: A finite set of input symbols.
3. Δ: A transition function defining for each pair of state and input signal a set of successor state

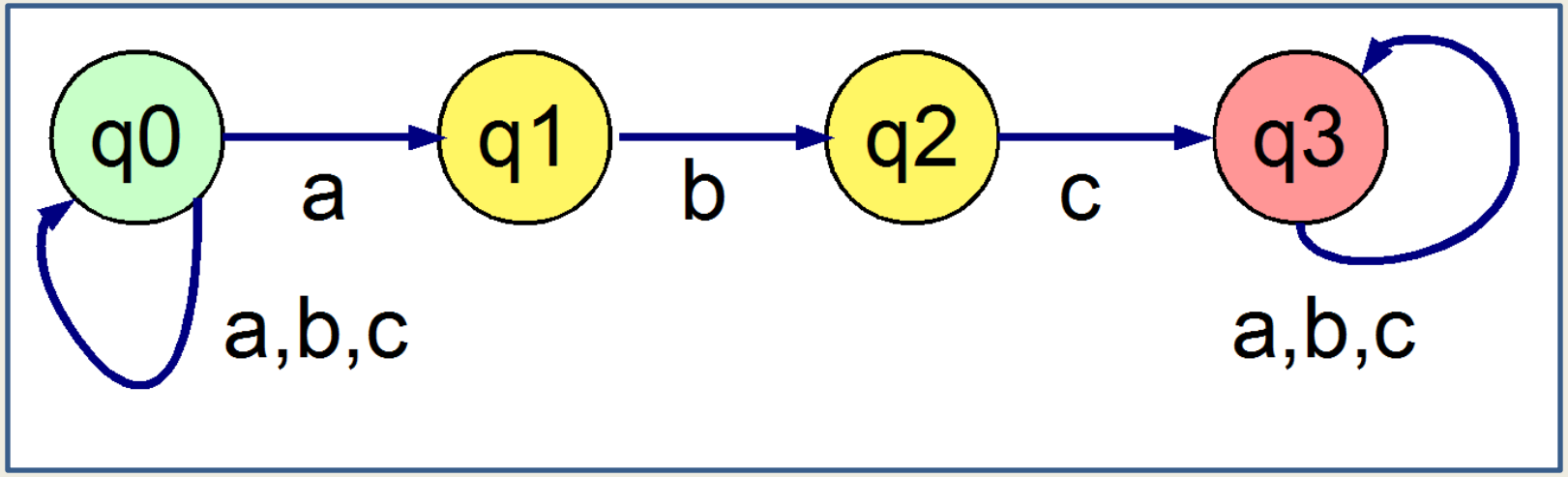
$$\Delta : (q, s) \in Q \times \Sigma \rightarrow \{q'_1, q'_2, \ldots, q'_n\} \subset Q$$

4. $q_0 \in Q$: An initial state
5. $F \subset Q$: A set of accepting states

Non-Deterministic Automata

- **Can be in various states at once**. Another way to state this: all states can be on or off, independent of each other.
- A transition may go lead to more than one state, **means to switch off the original state and to switch on several target state** (the original state can be among them).
- If a state has no transition rule for a given input signal, the state ceases to exist / is switched off.
- One may argue, whether the term “non-deterministic” is a good choice. We are not necessarily in one single state, but all that happens, occurs in a pre-defined way.

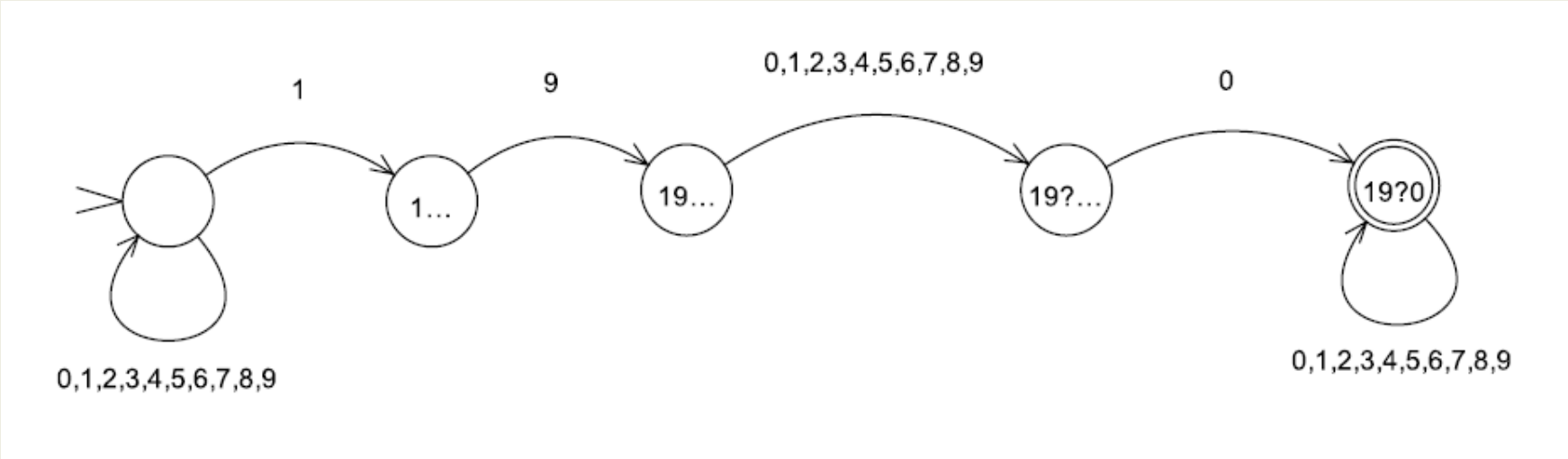
NFA: Diagram



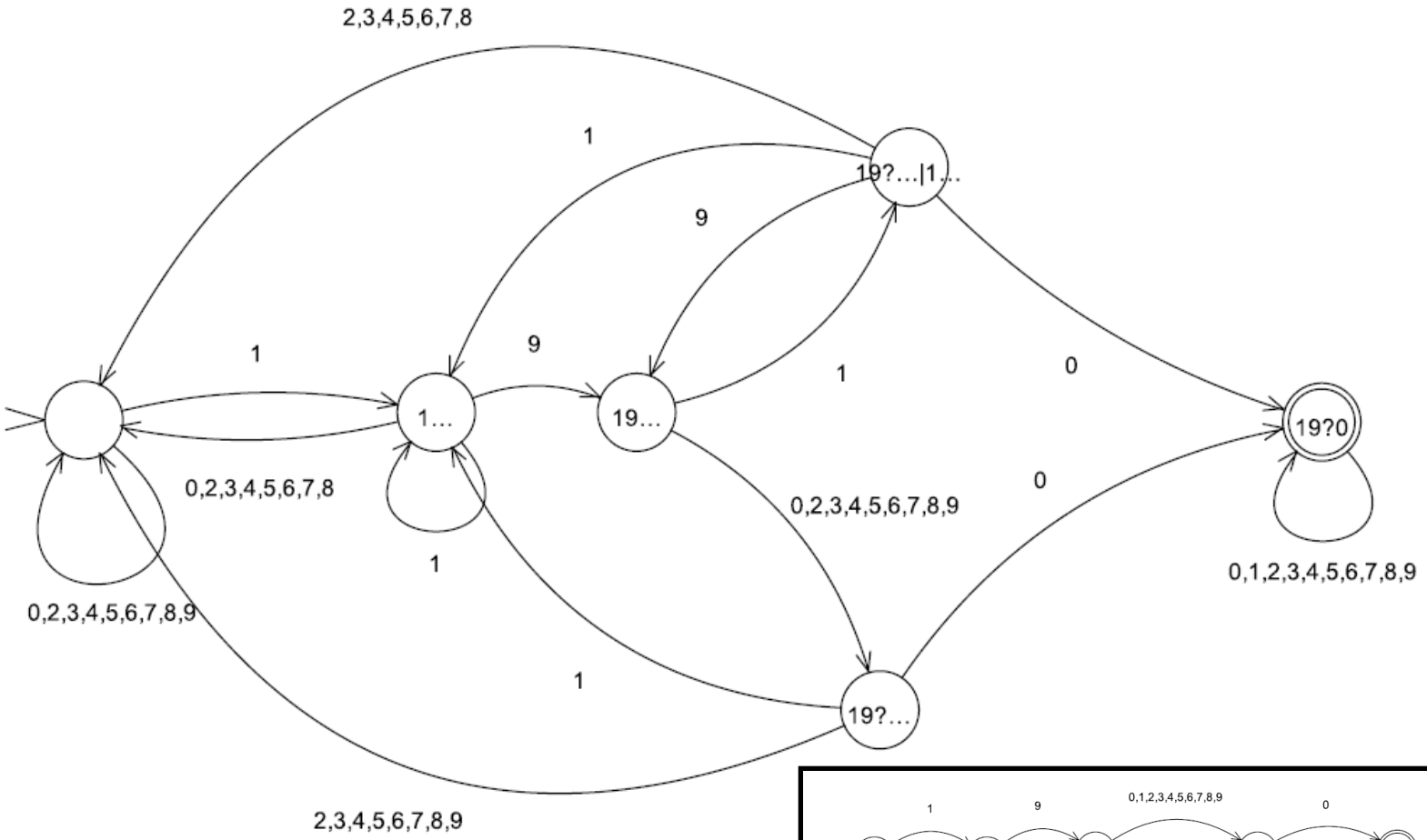
	a	b	c
q0	{q0,q1}	{q0}	{q0}
q1	0	{q2}	0
q2	0	0	{q3}
q3	{q3}	{q3}	{q3}

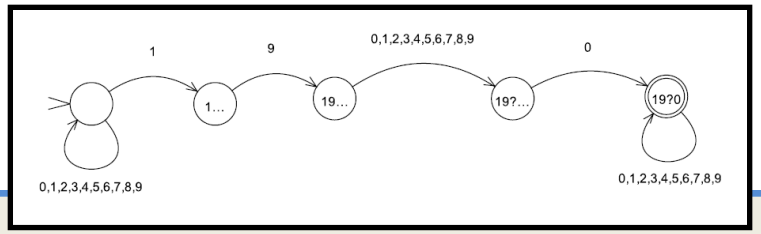
NFA: Example

- An automaton that recognizes dates of the form 19?0.

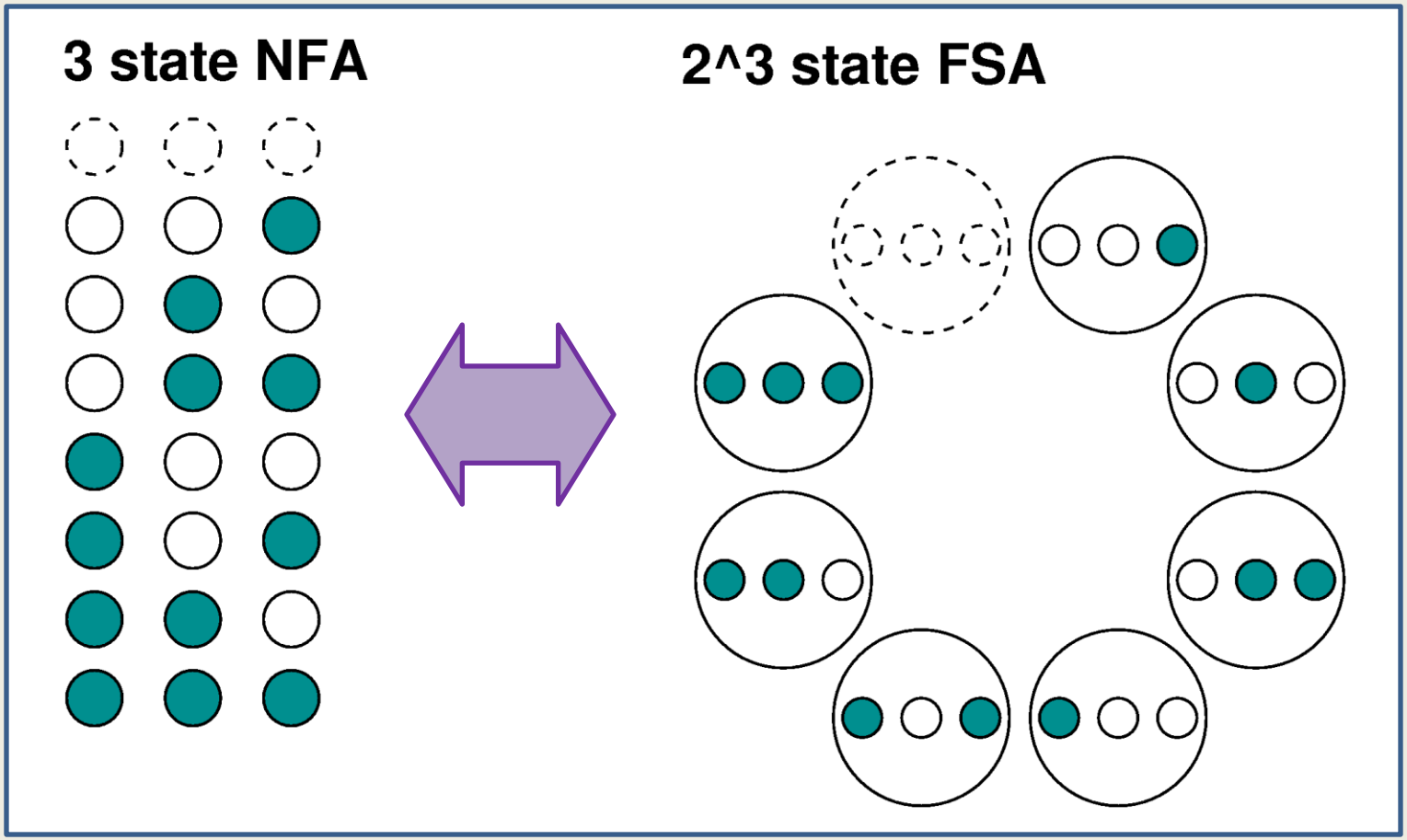


The According FSA





NFA and FSA



States of a n-state NFA can be mapped on a 2^n-state FSA.

Why NFA?

- Regular languages have the form

$$A \rightarrow \omega_1 B_1 \mid \omega_2 B_2 \mid \ldots$$
$$A \in V, \omega_i \in \Sigma^*, B_i \in V$$

- Rules = transitions in a NFA with output ➔ production of language elements.
- Production of languages ➔ recognition of languages.
- Each NFA can be converted into an FSA.

Regular language ➔ NFA ➔ FSA
Regular language ⇔ FSA

A Pumping Lemma

Why Pumping Lemmata?

- Pumping lemmata serve for proving that certain sequences cannot be recognized by a specific type of machine.
- Proving that there are strings which cannot be recognized by a given type of automata implies that we have to define a more general type!

Comment on Regular Languages and NFA

- In what follows, we take it for granted that if a language is regular, it can be recognized by an NFA and vice versa.
- We have only motivated but not proven this!
- Strictly following the definitions, FSA or NFA just recognize languages.
- We already introduced Mealy-machines. They can be used to produce strings.
- On the next slide, you will see how to construct a "producer" given a recognizer and vice versa. NOTE: The given argumentation only holds for regular languages!

Recognition = Production

- **Recognition to production:**
  1. Start with n = 0 and go up.
  2. Produce all possible strings with length n, given an alphabet ∑ List will be finite)
  3. Use the "recognizer" to filter the strings that belong to the given language
  4. ➔ You have produced the language elements of length n ➔ go to Step 1.
- **Production to recognition**
  1. Given a string of length n that should be recognized.
  2. Take the the producer to produce all strings up to length n.
  3. Check whether the given string is element of that list.

The Pumping Lemma

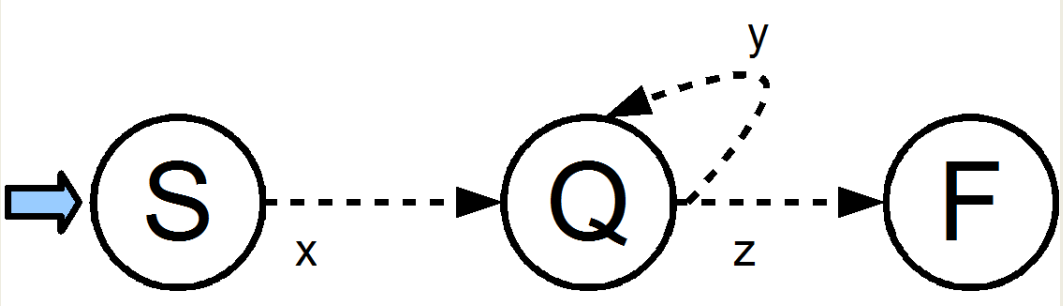
Let L be a regular language. Then, there exists a constant n (which depends on L) such that for every possible string $w \in L$ with n or more symbols ($|w| \geq n$) we can break w into three substrings $w = xyz$ in a manner that

1. $|y| > 0$, i.e. $y \neq \varepsilon$
2. $|xy| \leq n$
3. For all $k \geq 0$, it holds that $xy^k z \in L$

Means: For every string above a certain length, we can find a non-empty substring that can be “pumped”; that is, it can be repeated an arbitrary number of times (incl. zero) and the result is still element of L.

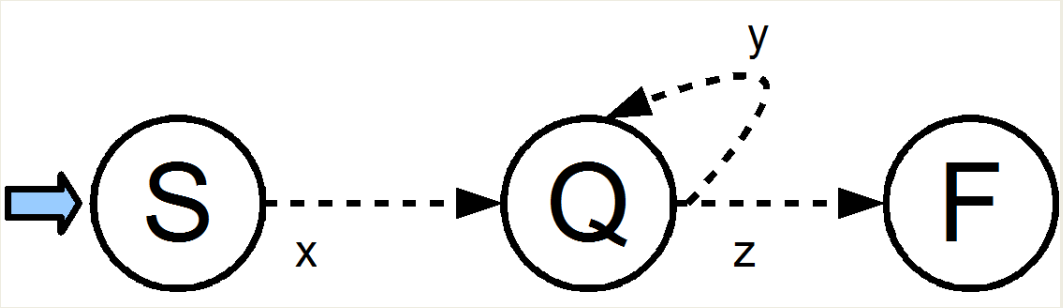
Pumping Lemma: Proof

- We assume that for every regular language, there is a NFA that generates/accepts this language.
- This NFA shall have **n** states. IMPORTANT: One of these states is the initial state, and one the final state. ➔ If one visits **n** states, on has produced a string of length **n – 1**.
- Now, for all **w ∈ L, |w| > n - 1** (equivalently: **|w| ≥ n**), it holds that at least one state **Q** has been visited twice, and this during the production of the first **n** symbols of **w** (which may b much longer than **n**).
- The string produced between the first and the second visit in **Q** is called **y**. **y** is necessarily non-empty. Note that **y** may have length 1, but can be longer.



Pumping Lemma: Proof

- The whole string produced from the start to the second visit of **Q** is set to **xy.** It necessarily holds that **|xy| ≤ n**.
- **w** may be longer than **xy.** We set **z** such that **w = xyz.**
- Now, if **xyz** is element in the language, **xyyz**, **xyyyz**, … is also in the language; and the same holds for **xz**. This, because the loop constructing **y** is a choice that can be chosen an arbitrary number of times, including zero.



Pumping Lemma: FAQ

Q: Why is it that if the machine has **n** states, there needs to be a loop if **|w| ≥ n**?

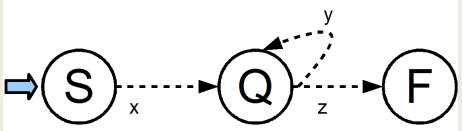
A: Because there are **n – 1** arrows (➔ symbols) between **n** states. So, if the sequence under consideration consists of **n** symbols, it is the result of **n + 1** visits in various states. There are only **n** states, so at least on state has been visited twice. ➔ **Pigeon hole principle**

Q: In the proof, one assumes that **|y| > 0**. But how can it be that **xz** is then necessarily an element of **L**?

A: The state **Q**, being visited twice, is either a final state or not. If it is a final state, one may just stop. If its not a final state, there is certainly a next state. But then, the y-producing loop may be skipped.

Q: Why is the **n** in the lemma equal to the number of states?

A: It isn't. The pumping lemma is, according to the proof, true for **n** = number of states, although it may hold form smaller **n** as well.



Pumping Lemma: Application

- Pumping lemma is used to prove that specific languages are non-regular.
- Strategy:

We know: (Regular lang. $\Rightarrow$ pumping lemma) $\Rightarrow$ (Not pumping lemma $\Rightarrow$ not regular language)

Pumping Lemma Formally

- The pumping lemma formally (n,k$\in$ N, w $\in$ L, L $\in$ $\boldsymbol{L_{reg}}$) :

$$\exists n > 0 \forall w \exists x, y, z \forall k \left(|w| \geq n \rightarrow \left(w = xyz \wedge |y| > 0 \wedge |xy| \leq n \wedge xy^k z \in L \right) \right)$$

- In order to formulate the negation of the pumping lemma, we have to know a couple of facts about logic:

$$(A \rightarrow B) \leftrightarrow (\neg A \vee B)$$

$$\neg(\neg A \vee B) \leftrightarrow (A \wedge \neg B)$$

$$\neg(\forall x P) \leftrightarrow \exists x(\neg P)$$

$$\neg(\exists x P) \leftrightarrow \forall x(\neg P)$$

Pumping Lemma Formally

$$(A \to B) \leftrightarrow (\neg A \vee B)$$
$$\neg(\neg A \vee B) \leftrightarrow (A \wedge \neg B)$$

$$\neg(\forall x P) \leftrightarrow \exists x(\neg P)$$
$$\neg(\exists x P) \leftrightarrow \forall x(\neg P)$$

$$\neg\left(\exists n > 0 \forall w \exists x, y, z \forall k \left(|w| \geq n \to \left(w = xyz \wedge |y| > 0 \wedge |xy| \leq n \wedge xy^k z \in L\right)\right)\right)$$

$$\forall n > 0 \exists w \forall x, y, z \exists k \neg\left(|w| \geq n \to \left(w = xyz \wedge |y| > 0 \wedge |xy| \leq n \wedge xy^k z \in L\right)\right)$$

$$\forall n > 0 \exists w \forall x, y, z \exists k \neg\left(\neg\left(|w| \geq n\right) \vee \left(w = xyz \wedge |y| > 0 \wedge |xy| \leq n \wedge xy^k z \in L\right)\right)$$

$$\forall n > 0 \exists w \forall x, y, z \exists k \left(\left(|w| \geq n\right) \wedge \neg\left(w = xyz \wedge |y| > 0 \wedge |xy| \leq n \wedge xy^k z \in L\right)\right)$$

Pumping Lemma Formally

$$(A \to B) \leftrightarrow (\neg A \vee B)$$

$$\neg(\neg A \vee B) \leftrightarrow (A \wedge \neg B)$$

$$\neg(\forall x P) \leftrightarrow \exists x(\neg P)$$

$$\neg(\exists x P) \leftrightarrow \forall x(\neg P)$$

$$\neg\left(A \wedge B \wedge C \wedge D\right)$$

$$\neg\left(\left(A \wedge B \wedge C\right) \wedge D\right)$$

$$\left(\neg\left(A \wedge B \wedge C\right) \vee \neg D\right)$$

$$\left(\left(A \wedge B \wedge C\right) \to \neg D\right)$$

$$\neg\left(\exists n > 0 \forall w \exists x, y, z \forall k \left(|w| \geq n \to \left(w = xyz \wedge |y| > 0 \wedge |xy| \leq n \wedge xy^k z \in L\right)\right)\right)$$

$$\forall n > 0 \exists w \forall x, y, z \exists k \neg\left(|w| \geq n \to \left(w = xyz \wedge |y| > 0 \wedge |xy| \leq n \wedge xy^k z \in L\right)\right)$$

$$\forall n > 0 \exists w \forall x, y, z \exists k \neg\left(\neg\left(|w| \geq n\right) \vee \left(w = xyz \wedge |y| > 0 \wedge |xy| \leq n \wedge xy^k z \in L\right)\right)$$

$$\forall n > 0 \exists w \forall x, y, z \exists k \left(\left(|w| \geq n\right) \wedge \neg\left(w = xyz \wedge |y| > 0 \wedge |xy| \leq n \wedge xy^k z \in L\right)\right)$$

Pumping Lemma Formally

$$\neg(A \wedge B \wedge C \wedge D)$$
$$\neg((A \wedge B \wedge C) \wedge D)$$
$$(\neg(A \wedge B \wedge C) \vee \neg D)$$
$$((A \wedge B \wedge C) \rightarrow \neg D)$$

$$(A \rightarrow B) \leftrightarrow (\neg A \vee B)$$
$$\neg(\neg A \vee B) \leftrightarrow (A \wedge \neg B)$$

$$\neg(\forall x P) \leftrightarrow \exists x(\neg P)$$
$$\neg(\exists x P) \leftrightarrow \forall x(\neg P)$$

$$\forall n > 0 \exists w \forall x, y, z \exists k \left((|w| \geq n) \wedge \neg \left(w = xyz \wedge |y| > 0 \wedge |xy| \leq n \wedge xy^k z \in L \right) \right)$$
$$\forall n > 0 \exists w \forall x, y, z \exists k \left((|w| \geq n) \wedge \left((w = xyz \wedge |y| > 0 \wedge |xy| \leq n) \rightarrow xy^k z \notin L \right) \right)$$

Negation of the pumping lemma (check carefully!!!)

$$\exists n > 0 \forall w \exists x, y, z \forall k \left(|w| \geq n \rightarrow \left(w = xyz \wedge |y| > 0 \wedge |xy| \leq n \wedge xy^k z \in L \right) \right)$$

Pumping Lemma: Application

We analyze the language **L = {0^k1^k, k≥ 0}**. We play a game:

- You can give me an **n** (It is "forall", take what you want, I'm not picky).
- I give you **w = 0^n1^n** (an "existence move", I give you a **w** of my choice, you have to take it).
- In whatever decomposition (You chose) **xyz** with **|xy|** $\leq$ **n, y** $\neq$**ε, y** will consist only of zeros.
- But, say for **k = 2, xy^2z** will have more zeros than ones ➔ not element of **L**.
- ➔ the conditions for the negation of the pumping lemma hold
- ➔ **L** is not a regular language.

$$\forall n > 0 \exists w \forall x, y, z \exists k \left(\left(|w| \geq n \right) \wedge \left(\left(w = xyz \wedge |y| > 0 \wedge |xy| \leq n \right) \rightarrow xy^k z \notin L \right) \right)$$

We know: (Regular lang. $\Rightarrow$ pumping lemma) $\Rightarrow$ (Not pumping lemma $\Rightarrow$ not regular language)

Consequence of the Pumping Lemma

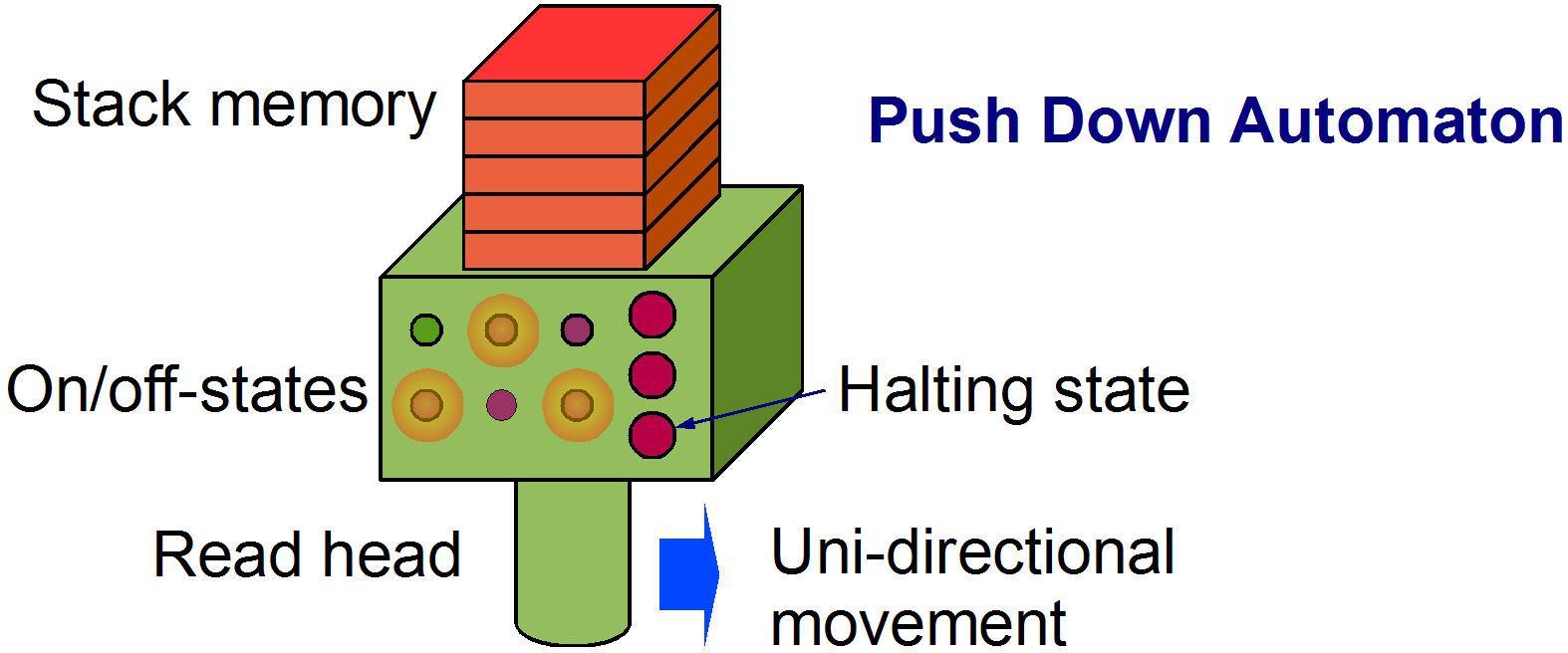
With the help of the pumping lemma, we can prove:

There are languages which cannot be recognized by FSA

Push – Down Automata

Zurich University
of Applied Sciences
School of
Engineering
ZAMP Center for Applied
Mathematics and Physics

Push Down Automata



0	1	1	1	0	0	1	0	0	0	END

Static tape, finite sequence.

PDA

Definition: A **push-down automaton** is a seven-tuple $(Q, \Sigma, \Phi, \Lambda, q_0, \phi_0, F)$

with

1. Q : A finite set of states
2. Σ : A finite set of input symbols
3. Φ : A finite set of stack symbols
4. Λ : A transition function given by a set

$$\Lambda \subset \left(Q \times \Sigma \times \Phi \times Q \times \Phi^*\right)$$

5. q_0 : An initial state
6. ϕ_0 : An initial stack symbol
7. $F \subset Q$: A set of accepting states.

PDA: The Transition Function

The transition function has to be read as follows:

If the PDA

1. Is in state q
2. Reads the symbol s from the tape
3. Reads a symbol ϕ from the stack
4. And it holds that $(q,s,\phi,q',f \in \Phi^*) \in \Lambda$

then actions as follows have to be performed:

- Go into state q'
- Pop s from the stack
- Write f sequentially on the stack. Note: $f \in \Phi^*$ may be a string!

Note well that in the above statement, q',f are not given by the tape or the state of the machine. It may well be that $(q,s,\phi,q',f \in \Phi^*) \in \Lambda$ as well as $(q,s,\phi,\hat{q}',\hat{f} \in \Phi^*) \in \Lambda, q' \neq \hat{q}', f \neq \hat{f}$. PDA's are usually assumed to be **non-deterministic**.

PDA

- The stack is always finite, but potentially unlimited.
- The transitions they undergo depend on the state, the input and the top symbol in the stack.
- transitions include actions on the stack, reading, removing, adding a symbol to the stack.

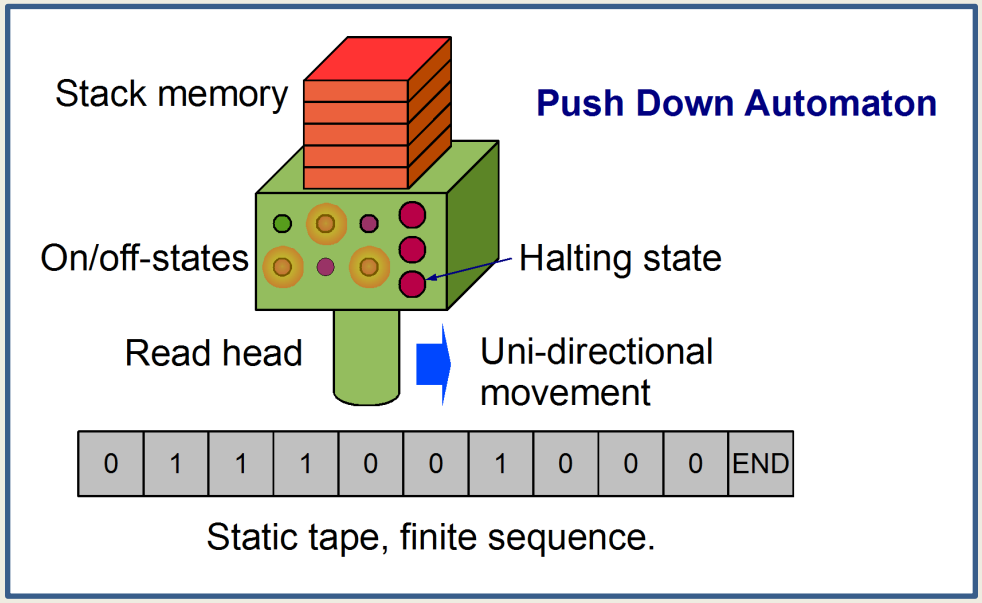
PDA and CFL

- For each context-free language L, there is a **non-deterministic PDA** that recognizes each element of L.
- Most conventional programming languages are CFL ➔ recognizing their elements is parsing.
- From parser to interpreter is only a small step.

Deterministic and Non-Deterministic PDA

NPDA $\neq$ DPDA, because the stack influences the transition function / relation. The stack is potentially unlimited.

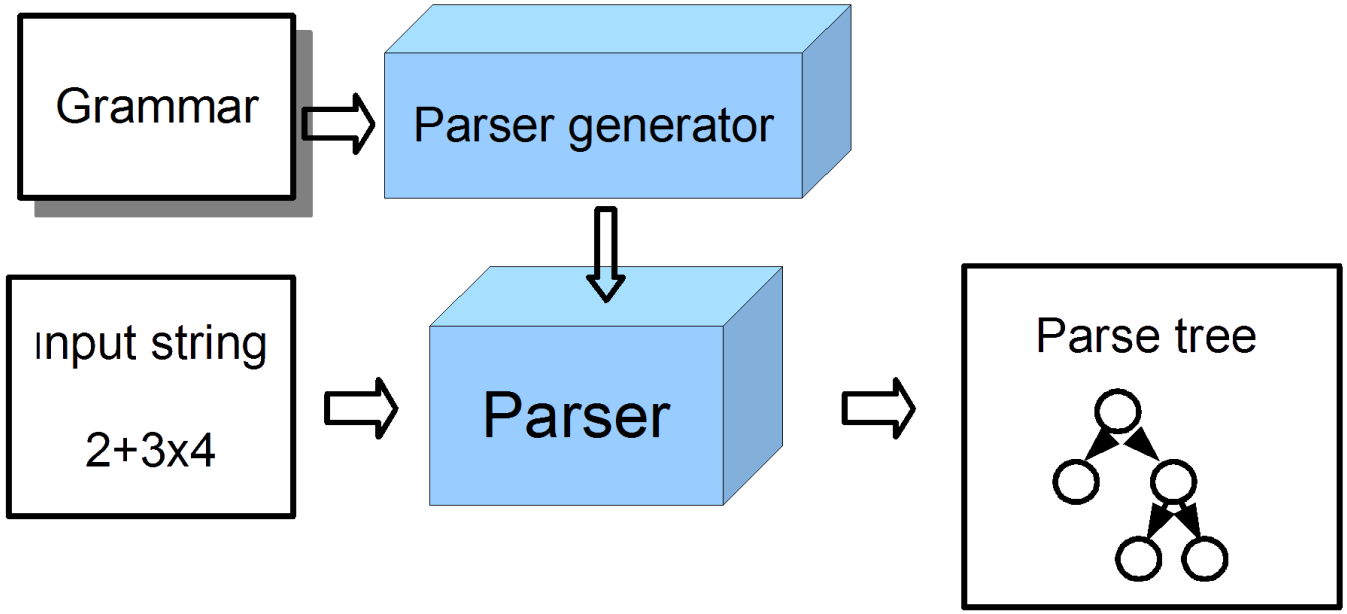
PDA refers in general to NPDA.



Parser and Parser Generators

Parser and Parser Generators

For context free languages, parsers can be generated in an automated way.



Compilers / Interpreters

- Parsing only tell you whether a string is syntactically correct.
- Running program needs **semantics.**
- A compiler converts the input string into a parse tree by realizing this parse tree as nested function calls. These functions are the atomic actions defined by the grammar!

Füchslin‘s Simple Lisp (FSLISP)

Non terminals		
expression	=	function | integer
Function	=	(functionname , epression , expression)
Functionname	=	prod | sum
digit	=	0 | 1 | 2| 3 | 4 | 5 | 6 | 7 | 8 | 9
integer	=	[+|-] digit {digit}

FSLISP represents sums and products of integers.

Examples:

- (prod, 4, 5)
- (sum, (prod, 3, 4), (prod, (sum, 2, 3), 7))

FSLISP Interpreter

Assumption: **`string`** is of the form **`integer`** or **`(string1, string2, string3)`**

```
String1(string) {
 returns string1
} ;

String2(string) {
 returns string2
} ;
String3(string) {
 returns string3
} ;
```

```
IsNumber(string) {
  if string is an
  integer, return 1
  else return 0
}
  ToNumber(string) {
  converts string
  integer
}
```

FSLISP Interpreter

```
FSLISP_ip(string) {
If IsNumber(string) == 1,
  return ToNumber(string);
string1 = String1(string);
string2 = String2(string);
string3 = String3(string);
If string1 == „prod“,
  return FSLISP_ip(string2) *
     FSLISP_ip(string3) ;
If string1 == „sum“,
  return FSLISP_ip(string2) +
     FSLISP_ip(string3);
}
```

Recursive calls of FSLISP_ip realize parse tree!

Basic Structure of a Compiler Compilers

A compiler compiler produces a program able of transforming an input string into a set of function calls.

- the start function gets the whole input string.
- It proceeds with “consuming” symbols, until it detects a pattern that requires calling (an)other function(s).
- These other function(s) get the rest of the string.

Interpreter

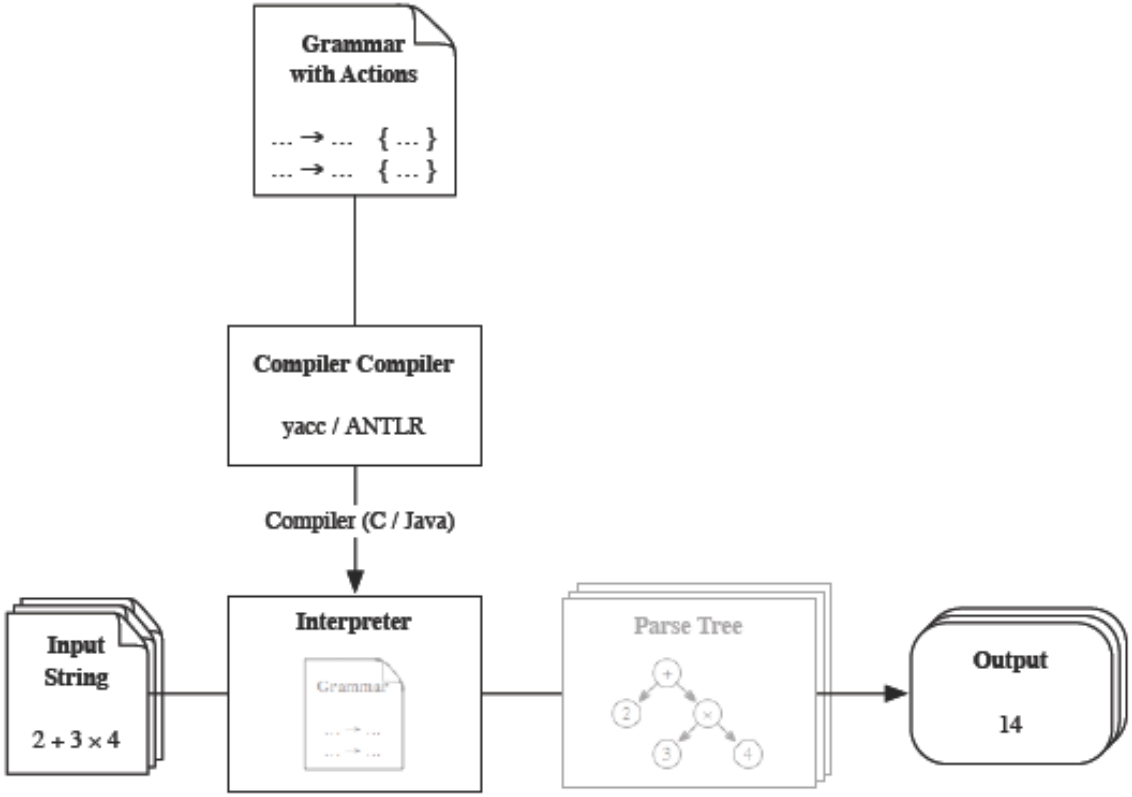


Figure 3.9: Overview of what a compiler compiler does. From a given grammar complemented with atomic actions (the "what"), it automatically generates a interpreter that checks the syntax and evaluates any number of input string (the "how").

Compiler Compilers and CFL

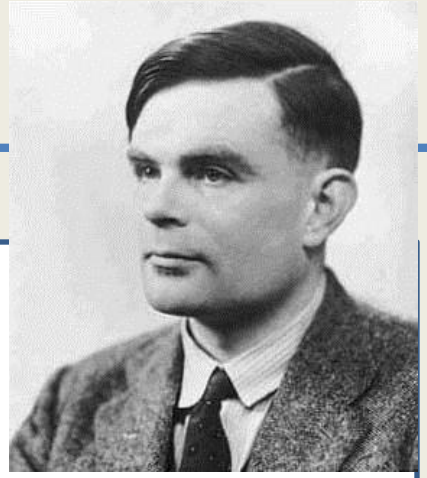
- The fact that we can speack about compiler compilers in general is due to the fact that

For context free languages, parsers can be generated in an automated way by the use of PDA.

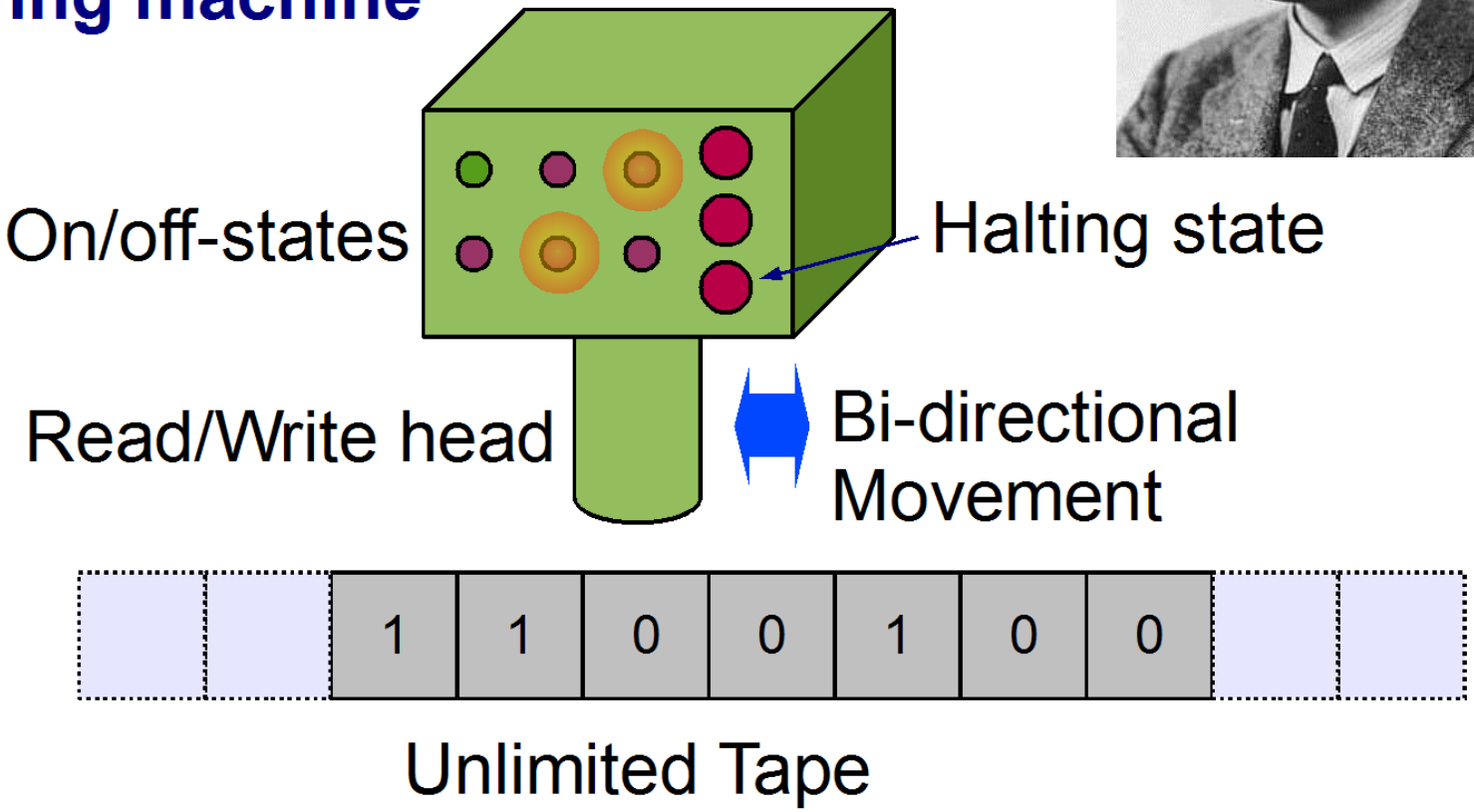
DEMO YACC and ANTLR

Turing Machines

Turing Machines



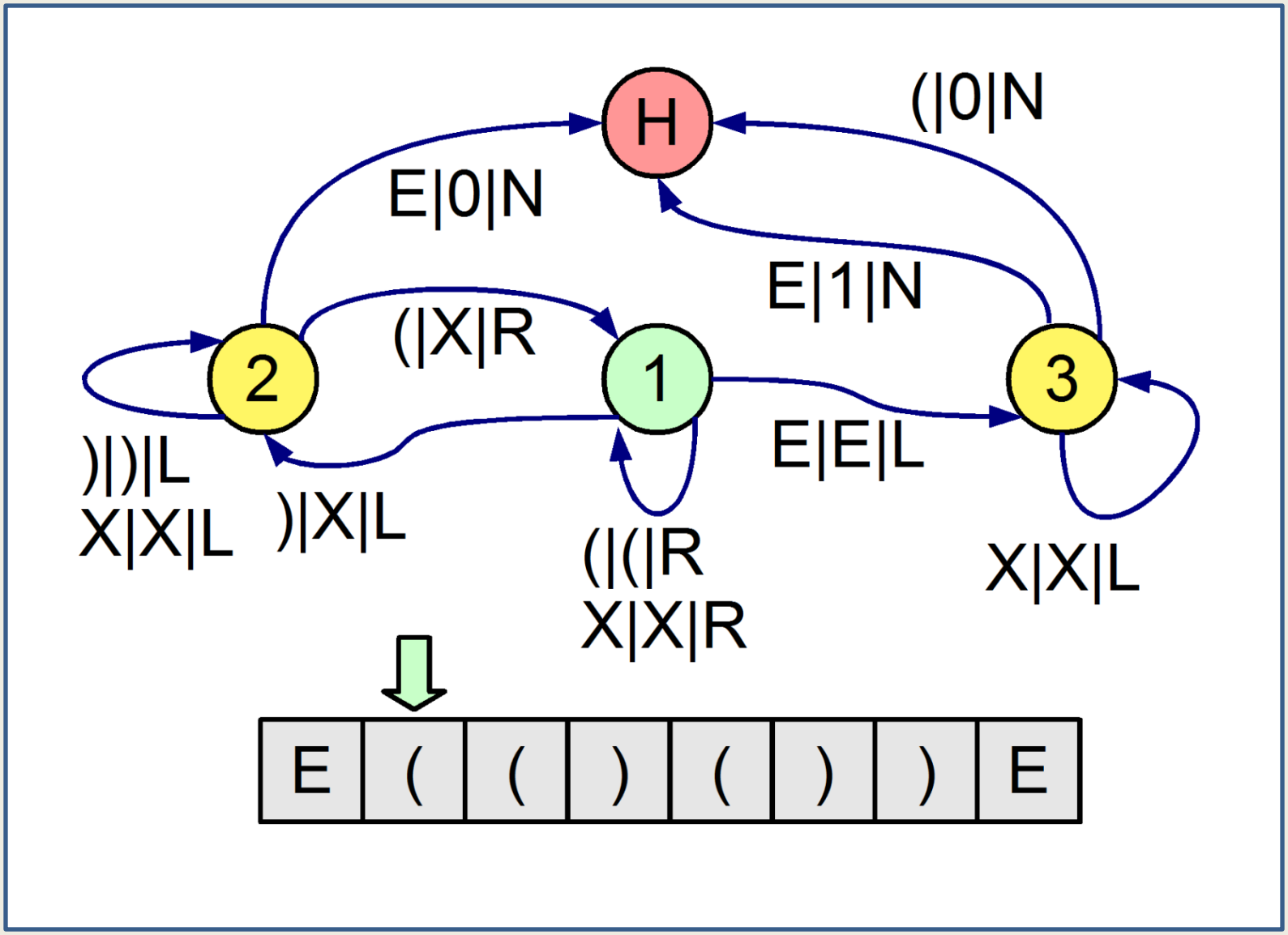
Turing machine



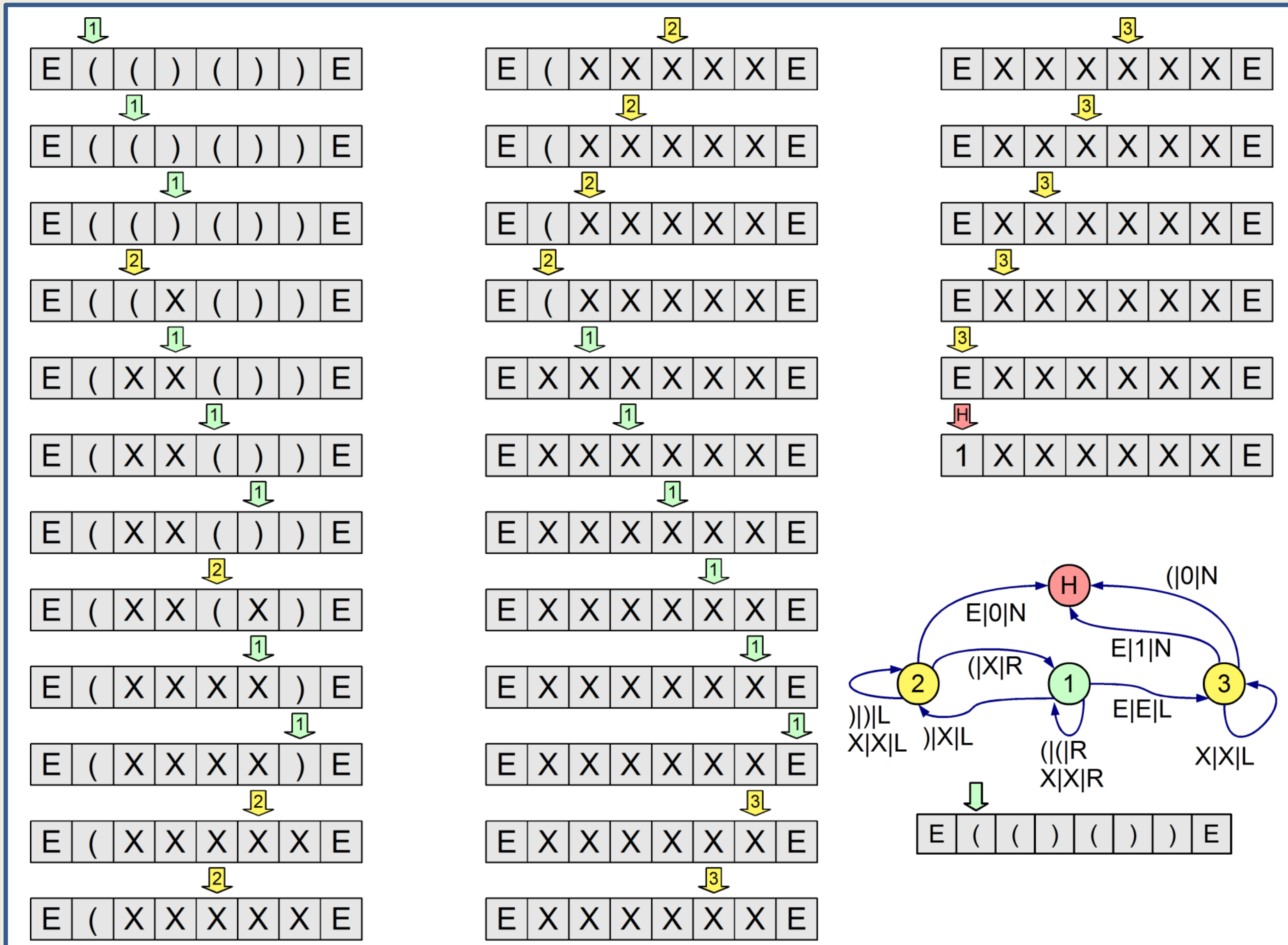
Components of a Turing Machines

- Tape with finite input string. Tape is potentially infinite, but the number of symbols is always finite.
- Read/Write head
- A finite state controller that defines, as a function of the actual state and the read symbol, which symbol is written and in which direction one has to move.
- A halting state.

Example: Minsky's Parenthesis Checker

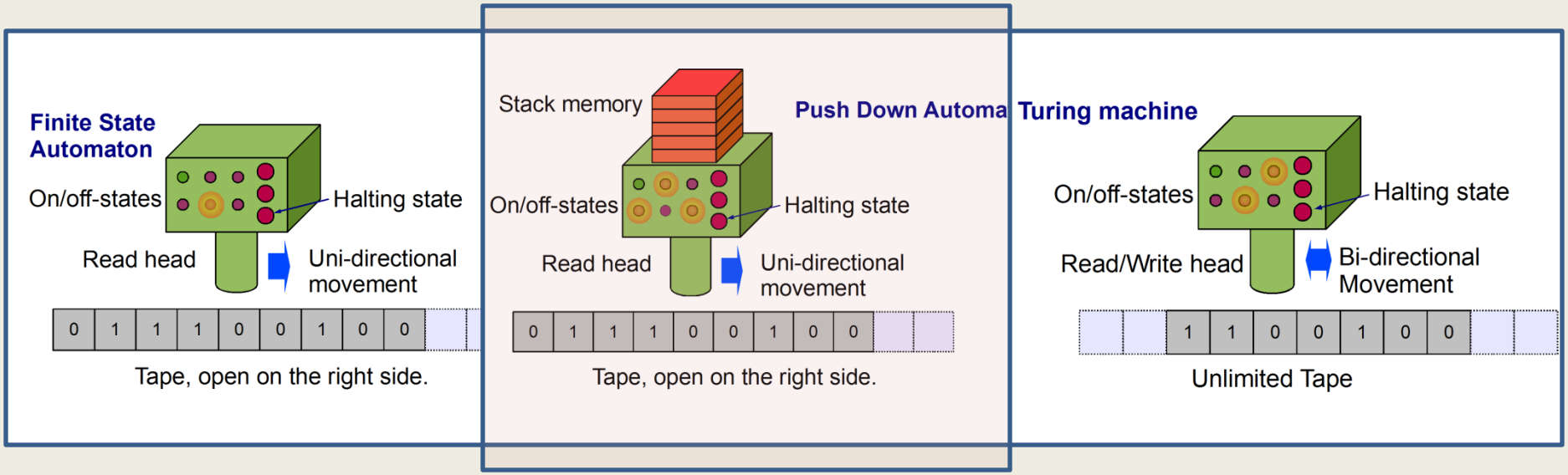


Example: Minsky's Parenthesis Checker



Detrministic and Non-Deterministic TM

- DTM and NDTM are equivalent.



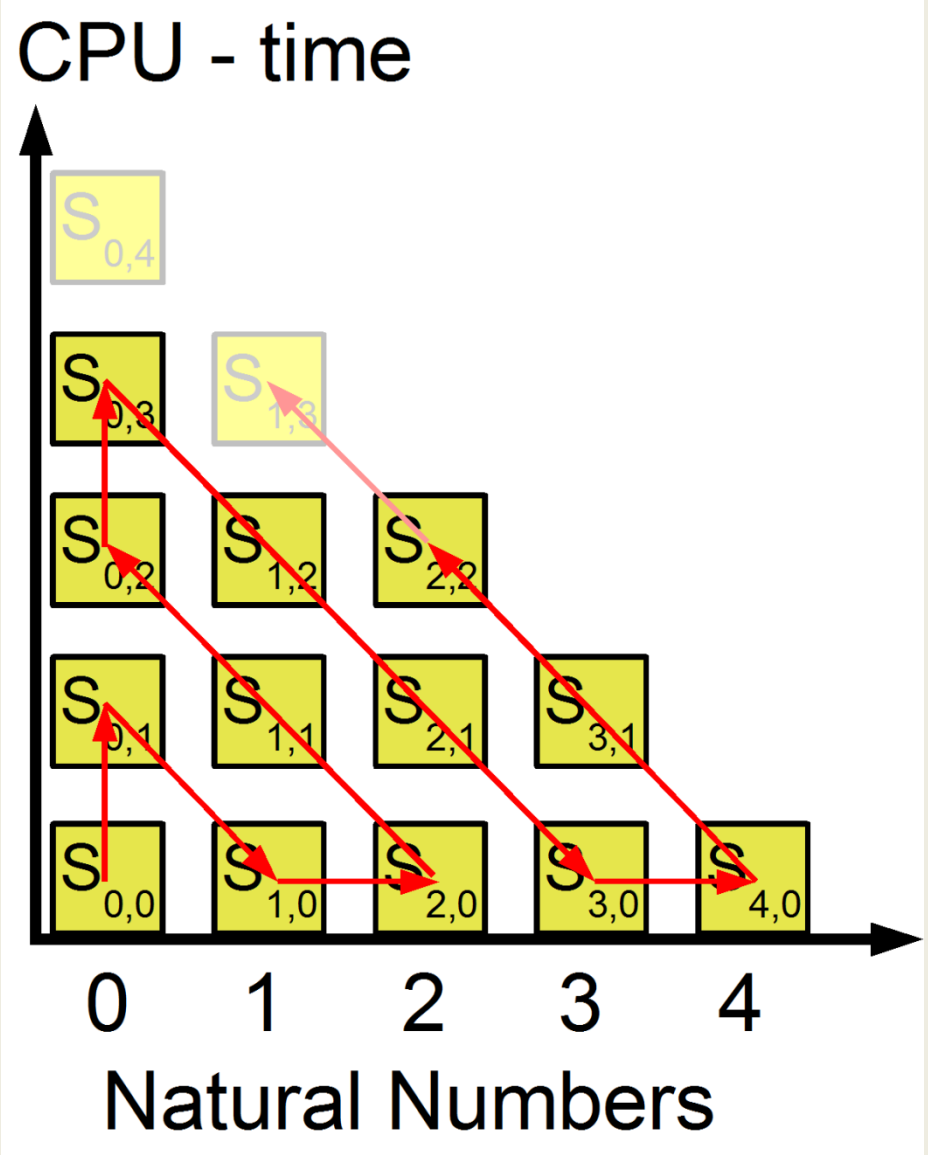
- FSA equivalent NFA
- NPDA $\neq$ PDA, because the stack influences the transition function / relation. The stack is potentially unlimited.

Recursive and Recursively Enumerable Sets

Definition: A set M is called **decidable, recursive** or **computable,** if there is a Turing machine that for each $m \in \mathbb{N}$ can compute in finitely many steps whether or not $m \in M$.

Definition: A set is called **recursively enumerable** if there is a Turing machine that for each $m \in \mathbb{N}$ can compute in finitely many steps whether $m \in M$. The machine may not reach an answer in finitely many steps in case of $m \notin M$.

Recursively Enumerable Sets



- How can we enumerate a set if we can't identify its non-members in a finite number of steps?
- By distributing the task: $S_{m,n}$ is the n-th package of steps for number m.

Universal Turing Machines

Every Turing machine can be encoded such that it can be emulated by a **universal Turing machine.**

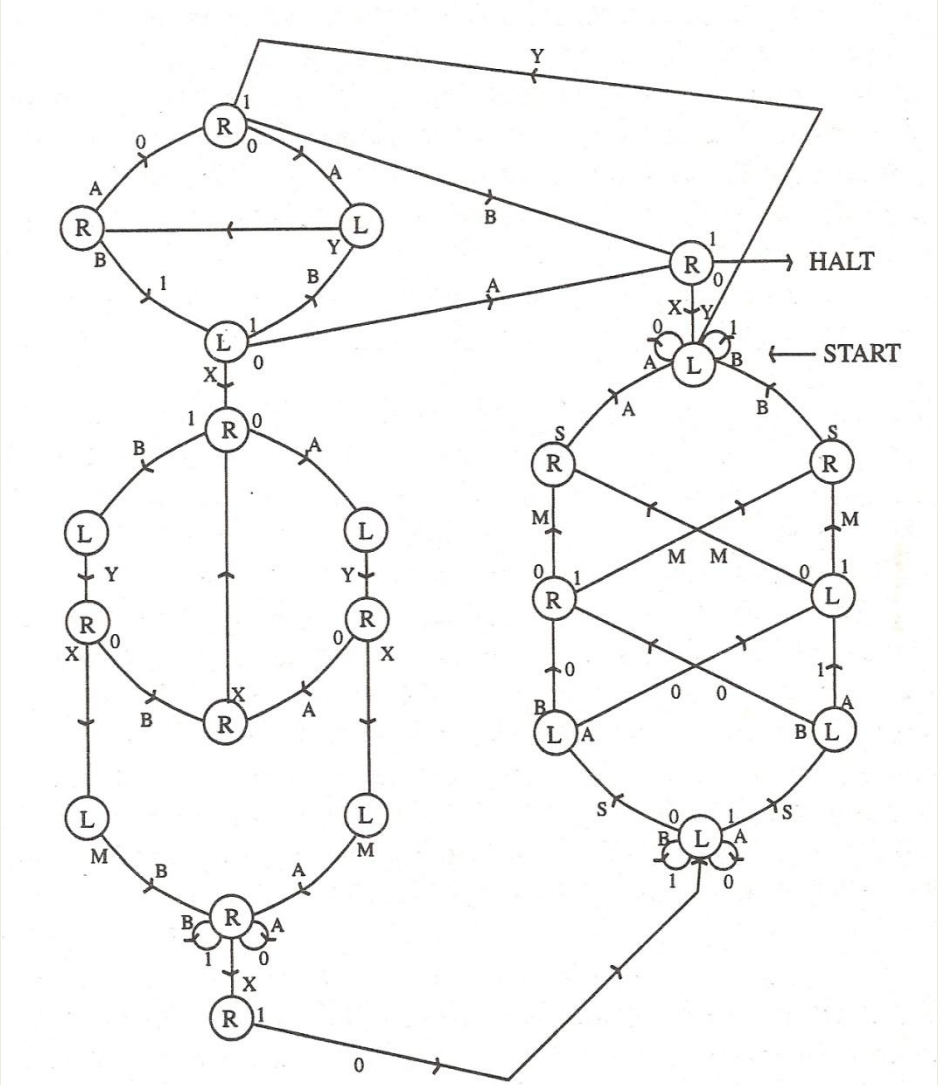


Fig. 3.23 A Universal Turing Machine

From R. P. Feynman: Lectures on computation

The Power of Turing Machines

- For each unrestricted grammar, there is a Turing machine that recognizes its elements. There is not necessarily a way to recognize the non-members.
- Turing machines / unresticted grammars represent the most general concept of formal manipulations we (widely) know.
- Everything your computer does, can be done by a Turing-machine.

The Halting Problem

The Halting Problem

- There are mathematically well defined functions that cannot be computed by a Turing machine.
- ➔ There are problems that cannot be solved by any computer, indept. how big and and fast it may be.
- There is a connection between formal mathematics and TM established by Robinson, Davis, Putnam and Matiyasevich:
  - TM can emulate all manipulations on Diophanbtine equations.
  - Diophantine equations can represent TM
  - ➔ If there are problems that cannot be solved by any TM, there must be an according mathematical problem that cannot be solved by formal means.

The Halting Problem 1

- Each computer program is represented by a number N. The same holds for its input M: N(M).
- One may ask, whether N(M) halts or continues forever.
- Assume, we have a program SProg(N), which answers this question for a specific situation

$$\mathrm{SProg}(N) = \begin{cases} 0, & N(N) \text{ halts} \\ 1, & N(N) \text{ does not halt} \end{cases}$$

The Halting Problem 2

We define

$$H(N) = \begin{cases} \text{halt}, & \text{SProg}(N) = 1 \;\Leftrightarrow\; N(N) \text{ goes on forever} \\ \text{loop}, & \text{SProg}(N) = 0 \;\Leftrightarrow\; N(N) \text{ stops eventually} \end{cases}$$

BUT

$$H(H) = \begin{cases} \text{halt}, & \text{SProg}(H) = 1 \;\Leftrightarrow\; H(H) \text{ goes on forever} \\ \text{loop}, & \text{SProg}(H) = 0 \;\Leftrightarrow\; H(H) \text{ stops eventually} \end{cases}$$

CONTRADICTION! This means, SProg(N) cannot exist!
The halting problem cannot be solved.

The Halting Problem: FAQ

- Q: How can we speak about something that doesn‘t exist?
- A: You have to distinguish between the syntactic and the semantic level. It is well possible to speak about the truth of logical relations between propositions about flying carpets (syntactic level) without having to say a word about the existence of flying carpets.

“If the existence of dragons implies the existence of unicorns, the non-existence of unicorns implies the non-existence of dragons.” is a **statement about the truth value of a relation between propositions**, not about the truth of the propositions themselves.

TM Variants: Linear Bounded TM

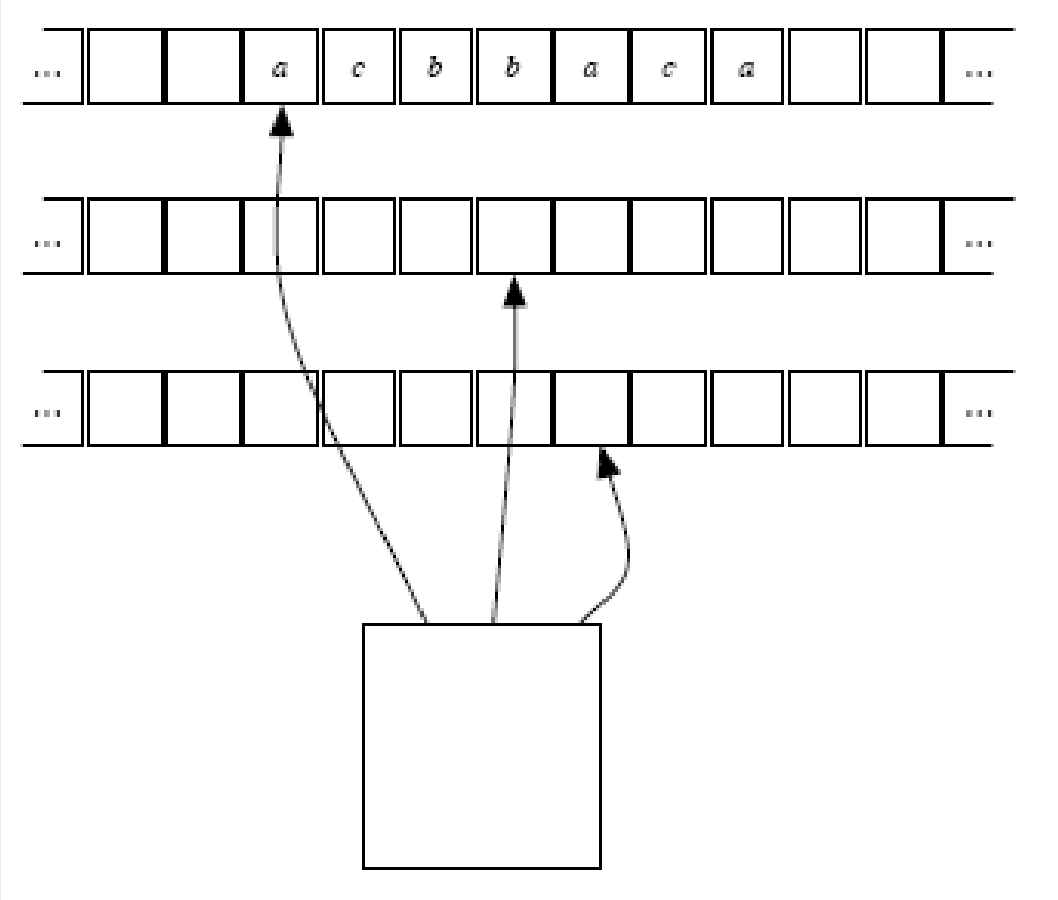
Definition: A TM with a tape of length k * n, where n is the length of the input string and k is a machine dependent constant is called **a linear bounded TM.**

Linear bounded TM can recognize context-sensitive but not general unrestricted languages.

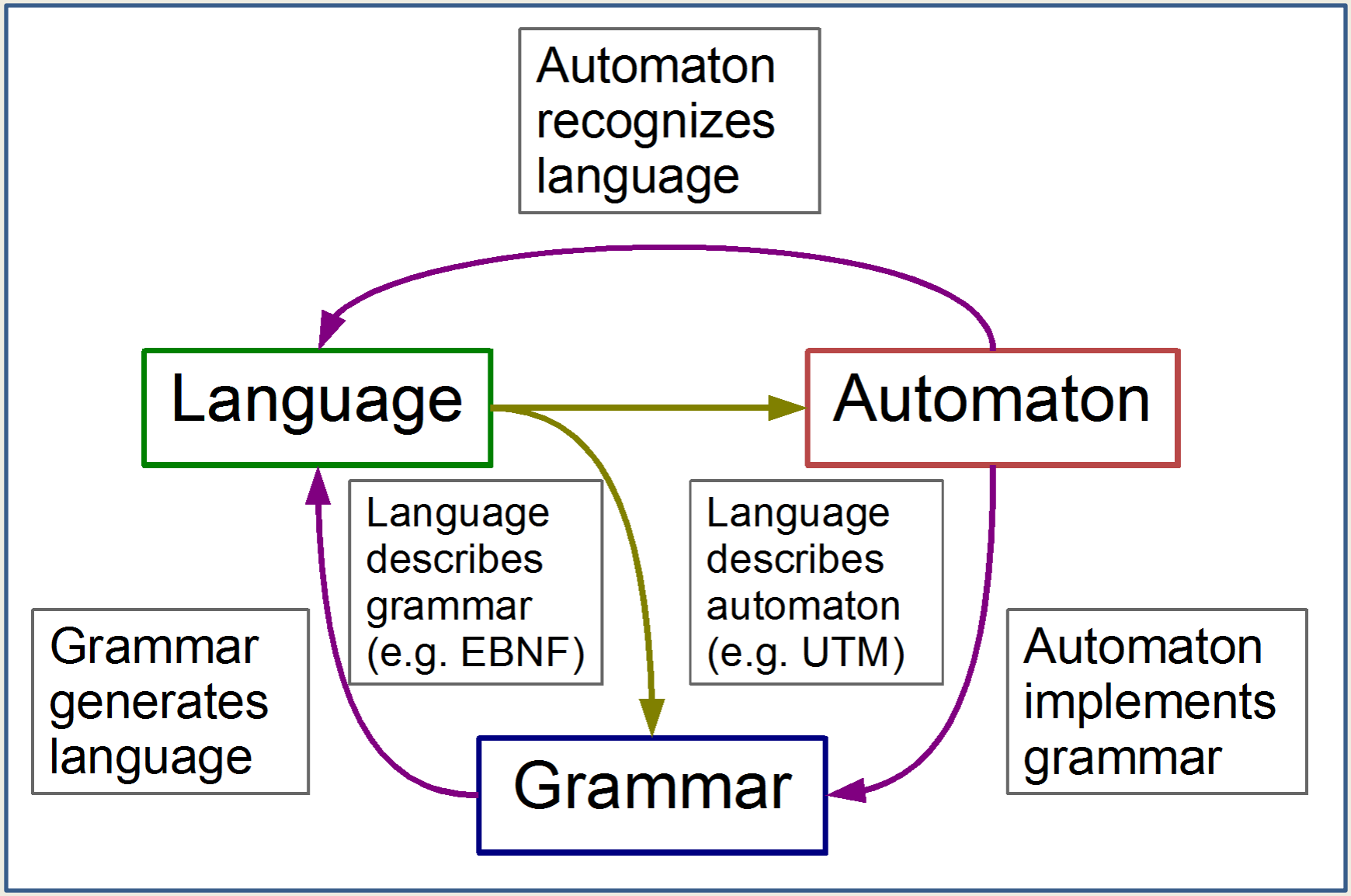
TM Variants: Multitape Machines

Definition: A **multitape TM** is a TM with multiple tapes.

Multitape TM are equivalent to single tape TM



Languages – Grammars - Automata



Church-Turing Thesis

The Church Turing Thesis

Every function which would naturally be regarded as computable can be computed by a Turing machine.

The term “naturally” is not well-defined.

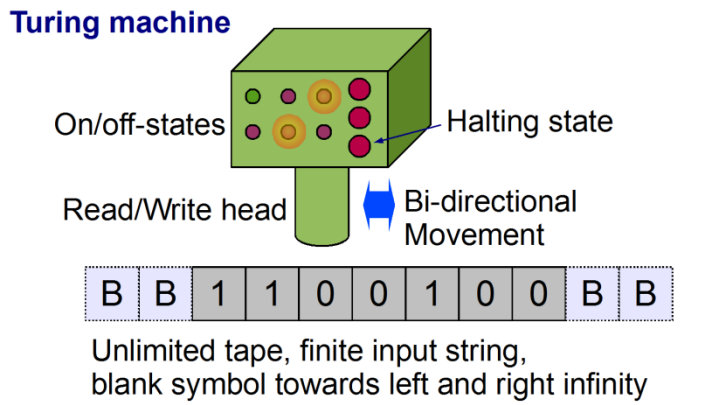
Of big interest is the **physical Church Turing Thesis**

Every function that can be physically computed can be computed by a Turing machine.

Morphological Computation will bring new insights.

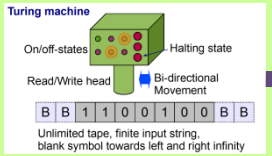
Other Computing Paradigms

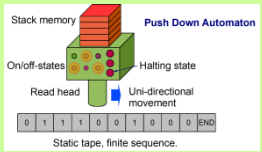
- Today, “computation” means Turing-computation.
- Your computer can emulate a TM and vice versa.
- Concerning novel paradigms: Distinguish two questions:
  - Does a novel paradigm enable the solution of so far unsolvable problems?
  - Does a novel paradigm enable more efficiently solving problems also solvable by TM?



The Quest for Infinity

Discrete systems

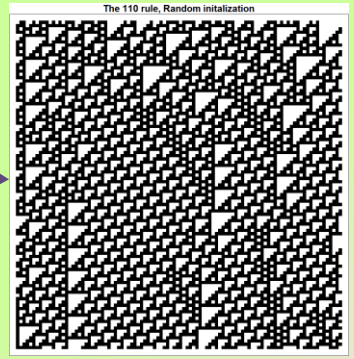


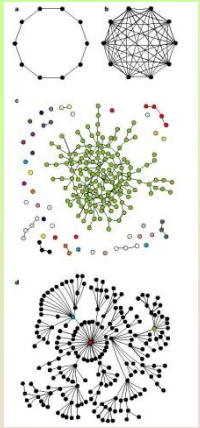


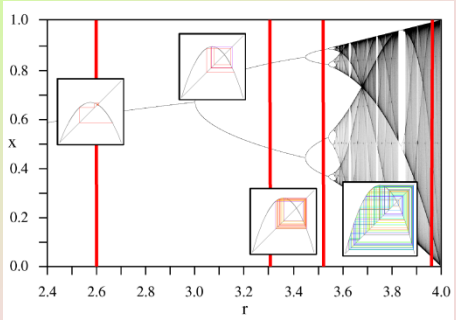
EQUIVALENCE!

Infinity via memory or tape

Infinity by space and number of finite automata

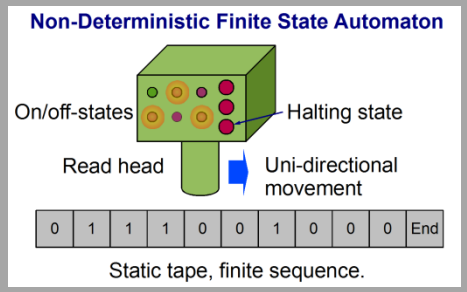


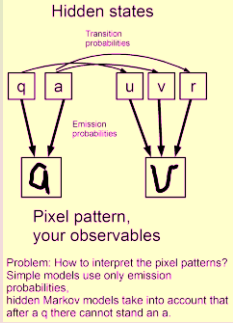


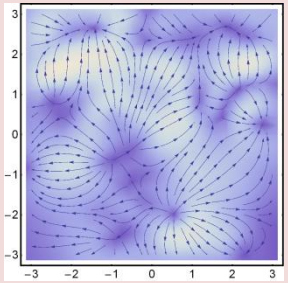


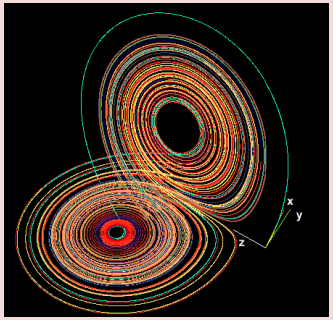
Continous parameter(s)

Infinity by continuous probabilities

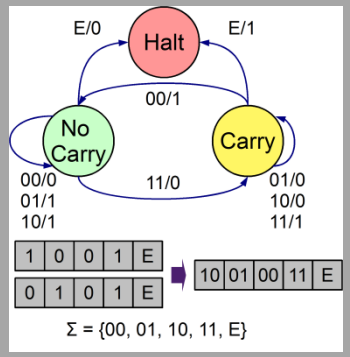




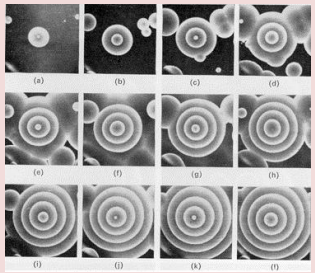




Finite systems



Continuous systems



Infinity by spatial and temporal continuity.