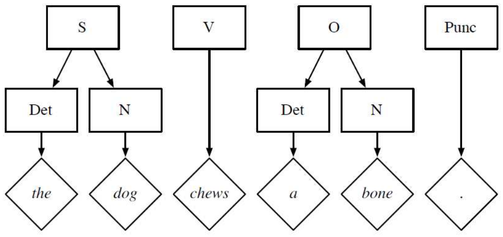
Formal Methöds II Sömmary

Chapter 2: Formal Languages

Natural Language (english)	Formal Language (c++)
+ High expressiveness + No extra learning − Ambiguity − Vagueness − Longish style − Consistency hard to check	+ Well-defined syntax + Unambiguous semantics + Can be processed by a computer + Large problems can be solved − High learning effort − Limited expressiveness − Low acceptance

Syntax:
This figure illustrates how a tree representation is a convenient way of representing the *syntax* structure of a single phrase:



Semantics:

The reason is that verifying the semantics of a phrase is almost impossible, whereas verifying its syntax is much simpler. To draw a comparison with a formal language, each compiler can verify whether a C++ code has a correct syntax (if so, it can be compiled; otherwise, the compiler will generate an error). But no computer will ever be able to verify that the code corresponds to a meaningful program!

Grammar:

One possible set of production rules – i.e. a *generative grammar* –corresponding to our example is the following (" | " stands for "or"):

I	$\rightarrow$	$P\ Punc$
P	$\rightarrow$	$S\ V\ O$
S	$\rightarrow$	$Det\ N$
O	$\rightarrow$	$Det\ N$
V	$\rightarrow$	*chews*
Det	$\rightarrow$	*the* | *a*
N	$\rightarrow$	*dog* | *bone*
$Punc$	$\rightarrow$	.

Table 2.2: A simple generative grammar.

I	$\rightarrow$	$P\ Punc \mid P_i\ Punc_i$
P	$\rightarrow$	$S\ V s\ O$
P_i	$\rightarrow$	$Aux\ S\ V\ O$
S	$\rightarrow$	$Det\ N$
O	$\rightarrow$	$Det\ N$
V	$\rightarrow$	*chew* | *bite* | *lick*
Aux	$\rightarrow$	*does* | *did* | *will*
Det	$\rightarrow$	*the* | *a* | *one* | *my*
N	$\rightarrow$	*dog* | *bone* | *frog* | *child*
$Punc$	$\rightarrow$	. | ; | !
$Punc_i$	$\rightarrow$	? | ?!?

Table 2.3: A slightly more complex generative grammar.

I	$\rightarrow$	$P\ Punc \mid P_i\ Punc_i$
P	$\rightarrow$	$S\ V\ s\ O$
P_i	$\rightarrow$	$Aux\ S\ V\ O$
S	$\rightarrow$	$Det\ N$
O	$\rightarrow$	$Det\ N$
V	$\rightarrow$	*chew* | *bite* | *lick*
Aux	$\rightarrow$	*does* | *did* | *will*
Det	$\rightarrow$	*the* | *a* | *one* | *my*
N	$\rightarrow$	$Adj\ N$ | *dog* | *bone* | *frog* | *child*
Adj	$\rightarrow$	*cute* | *tiny* | *little* | *furry* | *red-haired* | *big* | *tasty* | *white* | ...
$Punc$	$\rightarrow$	. | ; | !
$Punc_i$	$\rightarrow$	? | ?!?

recursion: $N \rightarrow Adj\ N$ =>

Table 2.4: A generative grammar with a recursive production rule.

Definition 2.1 (Kleene Star)**.** Let Σ be an alphabet, i.e. a finite set of distinct symbols. A string is a finite concatenation of elements of Σ. Σ^* is defined as the set of all possible strings over Σ.

Σ^* can equivalently be defined recursively as follows:

1. Basis: $\epsilon \in \Sigma^*$.
2. Recursive step: If $w \in \Sigma^*$ and $a \in \Sigma$, then $wa \in \Sigma^*$.
3. Closure: $w \in \Sigma^*$ only if it can be obtained from ϵ by a finite number of applications of the recursive step.

For any nonempty alphabet Σ, Σ^* contains infinitely many elements.

Σ	Σ^*
$\emptyset$	$\{\epsilon\}$
$\{a\}$	$\{\epsilon, a, aa, aaa, \ldots\}$
$\{0, 1\}$	$\{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011 \ldots\}$
$\{\$, \star, \pounds\}$	$\{\epsilon, \$, \star, \pounds, \$\$, \$\star, \$\pounds, \star\$, \star\star, \star\pounds, \pounds\$, \pounds\star, \pounds\pounds, \$\$\$, \$\$\star, \ldots\}$

Definition 2.2. A language L over an alphabet Σ is a subset of Σ^*:

$$L \subseteq \Sigma^*$$

A language is thus a possibly infinite set of finite-length sequences of elements (strings) drawn from a specified finite set (its alphabet Σ).

The union of two languages L and M over an alphabet Σ is defined as:

$$L \cup M := \{w \in \Sigma^* \mid w \in L \vee w \in M\}$$

The intersection of two languages L and M over an alphabet Σ is defined as:

$$L \cap M := \{w \in \Sigma^* \mid w \in L \wedge w \in M\}$$

The concatenation of two languages L and M is defined as:

$$LM := \{uv \mid u \in L \wedge v \in M\}$$

Example 2.1. L_1 is the language over $\Sigma = \{0, 1\}$ consisting of all strings that begins with 1.

$$L_1 = \{1, 10, 11, 100, 101, \ldots\}$$

L_2 is the language over $\Sigma = \{0, 1\}$ consisting of all strings that contains an even number of 0's.

$$L_2 = \{\epsilon, 1, 00, 11, 001, 010, 100, 0011, \ldots\}$$

L_3 is the language over $\Sigma = \{a, b\}$ consisting of all strings that contains as many a's and b's.

$$L_3 = \{\epsilon, ab, ba, aabb, abab, baba, bbaa, \ldots\}$$

L_4 is the language over $\Sigma = \{x\}$ consisting of all strings that contains a prime number of x's.

$$L_4 = \{xx, xxx, xxxxx, xxxxxxx, xxxxxxxxxxx, \ldots\}$$

Definition 2.3. A grammar G is formally defined as a quadruple

$$G := (\Sigma, V, P, S)$$

with

- Σ: a finite set of terminal symbols (the alphabet)
- V: a finite set of non-terminal symbols (the variables), usually with the condition that $V \cap \Sigma = \emptyset$.
- P: a finite set of production rules
- $S \in V$: the start symbol.

Non-terminal symbols are usually represented by uppercase letters, terminal symbols by lowercase letters, and the start symbol by S.

A sequence of rule applications is called a *derivation*.

Example 2.2. Let us consider the following grammar:

$$\begin{array}{rl} \text{Alphabet } \Sigma: & \{0, 1, +\} \\ \text{Variables } V: & \{S, N\} \\ \text{Production rules } P: & S \rightarrow N \mid N + S \\ & N \rightarrow 0 \mid 1 \mid NN \\ \text{Start symbol:} & S \end{array}$$

The corresponding language contains simple expressions corresponding to the additions of binary numbers, such as $w = 10 + 0 + 1$. One possible derivation of w is as follows:

This derivation can be represented by the following *syntax tree*:

$$
\begin{aligned}
S &\Rightarrow N + S \\
&\Rightarrow NN + S \\
&\Rightarrow 1N + S \\
&\Rightarrow 10 + S \\
&\Rightarrow 10 + N + S \\
&\Rightarrow 10 + N + N \\
&\Rightarrow 10 + N + 1 \\
&\Rightarrow 10 + 0 + 1
\end{aligned}
$$

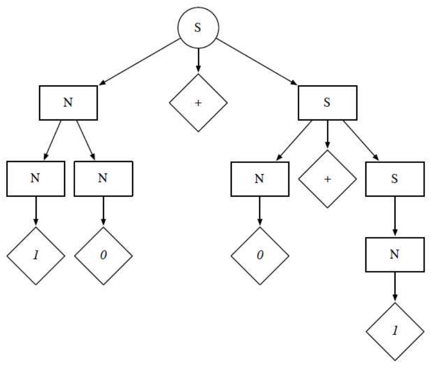


Figure 2.3: Syntax tree corresponding to one possible derivation of “10 + 0 + 1”.

regular grammars

Definition 2.4. All the production rules of a *right regular grammar* are of the form:

$$A \to xyz \ldots X$$

where A is a non-terminal symbol, $xyz\ldots$ zero or more terminal symbols, and X zero or one non-terminal symbol. A *left* regular grammar have productions rules of the form:

$$A \to Xxyz \ldots$$

A regular grammar is either a left regular or a right regular grammar.

=> append the symbols at either the left or the right during derivation

=> a regular language is a language that can be generated by a regular grammar

Example 2.4. An archetypical regular language is:

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

L is the language of all strings over the alphabet $\Sigma = \{a, b\}$ where all the a's precede the b's: $L = \{\epsilon, a, b, aa, ab, bb, aaa, aab, \ldots\}$.

regular expressions

Regular languages are typically described by *regular expressions*:

Symbol	Stands for...
+	at least one occurrence of the preceding symbol
*	zero, one, or more occurrences of the preceding symbol
?	zero or one occurrence of the preceding symbol
|	logical “or”
ϵ	the empty string

Examples of regular expressions

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

context free languages

Context-free languages include most programming languages, and is thus one the central category in the theory of formal languages.

Definition 2.6. A context-free grammar is a grammar whose productions rules are all of the form:

$$A \to \ldots$$

Definition 2.7. A language L is said to be a context-free language if there exists a context-free grammar G, such that $L = L(G)$.

In other words, a context-free language is a language that can be generated by a context-free grammar.

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: $L = \{\epsilon, ab, aabb, aaabbb, \ldots\}$.

Backus-Naur Forms (BNF)

A BNF (Backus-Naur form or Backus normal form) is a particular syntax used to express context-free grammars in a slightly more convenient way. A derivation rule in the original BNF specification is written as

$$< \text{symbol} > ::= < \text{expression with symbols} >$$

Terminal symbols are usually enclosed within quotation marks ("..." or '...'):

$$\begin{aligned} < \text{number} > \quad &::= \quad < \text{digit} > \mid < \text{digit} >< \text{number} > \\ < \text{digit} > \quad &::= \quad \text{"0"} \mid \text{"1"} \mid \text{"2"} \mid \text{"3"} \mid \text{"4"} \mid \text{"5"} \mid \text{"6"} \mid \text{"7"} \mid \text{"8"} \mid \text{"9"} \end{aligned}$$

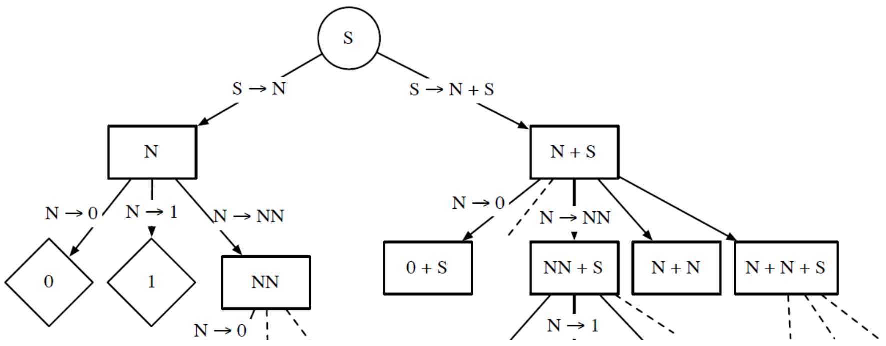
Example 2.8. The following grammar in EBNF notation defines a simplified HTML syntax:

```
   document  =  element ;
    element  =  ( text | list ) * ;
       text  =  ('A' .. 'Z' | 'a' .. 'z' | '0' .. '9' | ' ') + ;
       list  =  '<ul>' listElement * '</ul>'
             |  '<ol>' listElement * '</ol>' ;
listElement  =  '<li>' element ;
```

Figure 2.4 shows the syntax tree corresponding to the following simplified HTML code:

```
Buy<ol><li>Fruits<ul><li>Apple<li>Banana</ul><li>Pasta<li>Water</ol>
```

grammar tree:
A *grammar tree* is a tree where each *link* corresponds to the application of one particular production rule, and where the leafs represent the elements of the language.



parsing:
Parsing (also referred more formally to as “syntactic analysis”) is the process of analyzing a sequence of symbols (or more generally tokens) to determine its grammatical structure with respect to a given formal grammar.

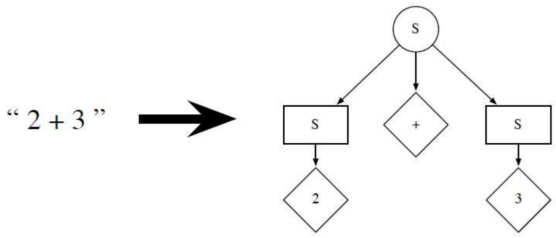


Figure 2.6: The process of parsing. Left: the input string. Right: the corresponding parse tree showing the syntactical structure.

ambiguity:
A grammar is said to be *ambiguous* if the language it generates contains some string that have more than one possible parse tree.
example: “2–3–4” => (2 − 3) − 4 = −5 or 2 − (3 − 4) = 3

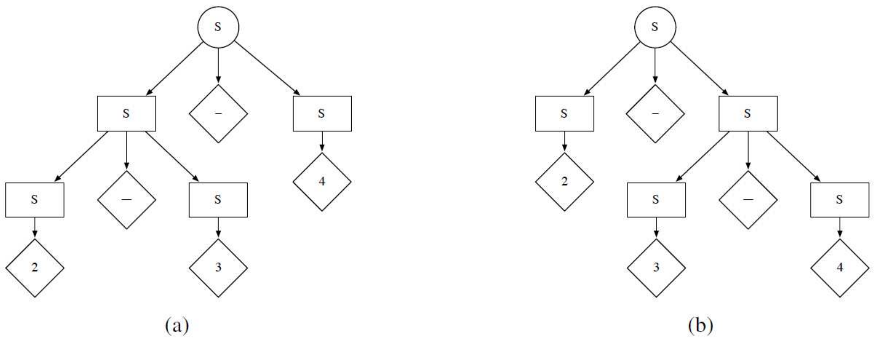


Figure 2.7: Two parse trees showing the two possible derivations of the input string “$2 - 3 - 4$”.
=> leftmost-derivation and rightmost-derivation
make grammar unambiguous (example "2-3*4):

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

Table 2.6: A simple unambiguous context-free grammar. =>

$$\begin{aligned} S &\Rightarrow E - T \Rightarrow T - T \\ &\Rightarrow F - T \Rightarrow N - T \Rightarrow 2 - T \\ &\Rightarrow 2 - T \times F \Rightarrow 2 - F \times F \\ &\Rightarrow 2 - N \times F \Rightarrow 2 - 3 \times F \\ &\Rightarrow 2 - 3 \times N \Rightarrow 2 - 3 \times 4 \end{aligned}$$

context sensitive languages

Definition 2.12. A grammar $G = (\Sigma, V, P, S)$ is called *context-sensitive* if each rule has the form $u \to v$, where $u, v \in (\Sigma \cup V)^+$, and $|u| \leq |v|$.

In addition, a rule of the form $S \to \epsilon$ is permitted provided S does not appear on the right side of any rule. This allows the empty string ϵ to be included in context-sensitive languages.

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:

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

The last rule of this grammar illustrates the reason why such grammars are called "context-sensitive": C can only be replaced by c in a particular context – namely when it is followed by a c.

unrestricted grammar:

Definition 2.14. An *unrestricted grammar* is a formal grammar $G = (\Sigma, V, P, S)$ where each rule has the form $u \to v$, where $u, v \in (\Sigma \cup V)^+$, and $u \neq \epsilon$.

As the name implies, there are no real restrictions on the types of production rules that unrestricted grammars can have.

Chomsky Hierarchy

$$\mathcal{L}_{\text{regular}} \subset \mathcal{L}_{\text{context-free}} \subset \mathcal{L}_{\text{context-sensitive}} \subset \mathcal{L}_{\text{unrestricted}} \subset \mathcal{P}(\Sigma^*)$$

where $\mathcal{L}_{type}$ is the set of all languages generated by a grammar of type *type*, and $\mathcal{P}(\Sigma^*)$ the power set of Σ^*, i.e. the set of all possible languages.

The Chomsky hierarchy is summarized in Table 2.7.

Type	Grammar	Production Rules	Example of Language
0	Unrestricted	$\alpha \to \beta$	
1	Context-Sensitive	$\alpha A \beta \to \alpha \gamma \beta$	$\{a^n b^n c^n \mid n \geq 0\}$
2	Context-Free	$A \to \gamma$	$\{a^n b^n \mid n \geq 0\}$
3	Regular	$A \to \epsilon \mid a \mid aB$	$\{a^m b^n \mid m, n \geq 0\}$

Table 2.7: The Chomsky hierarchy.

undecidable problems

1. **The recognition problem**
   Given a string w and a grammar G, is $w \in L(G)$?

2. **The emptiness problem**
   Given a grammar G, is $L(G) = \emptyset$?

3. **The equivalence problem**
   Given two grammars G_1 and G_2, is $L(G_1) = L(G_2)$?

4. **The ambiguity problem**
   Given a grammar G, is G ambiguous?

Type	Problem: (√ = decidable, □ = undecidable)			
	1. Recognition	2. Emptiness	3. Equivalence	4. Ambiguity
0	□	□	□	□
1	√	□	□	□
2	√	√	□	□
3	√	√	√	√

Table 2.8: Decidability of problems in the Chomsky hierarchy.

Chapter 3 Automata Theory

computation

"Does a particular string w belongs to a given language L or not?"

Definition 3.1. A *decision problem* is a mapping

$$\{0,1\}^* \longmapsto \{0,1\}$$

that takes as input any finite string of 0's and 1's and assign to it an output consisting of either 0 ("no") or 1 ("yes").

Definition 3.2. The $\mathcal{O}$ notation (pronounce: big oh) stands for "order of" and is used to describe an asymptotic upper bound for the magnitude of a function in terms of another, typically simpler, function.

Finite State Automata

A *finite state automaton* (plural: finite state automata) is an abstract machine that successively reads each symbols of the input string, and changes its state according to a particular control mechanism. If the machine, after reading the last symbol of the input string, is in one of a set of particular states, then the machine is said to *accept* the input string. It can be illustrated as follows:

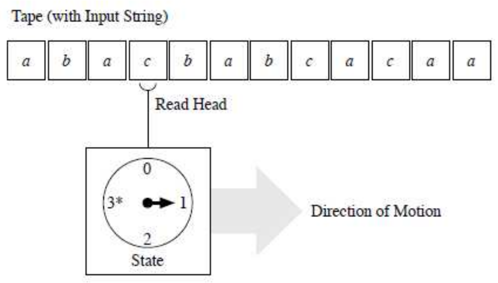


Figure 3.1: Illustration of a finite state automaton.

Definition 3.3. A *finite state automaton* is a five-tuple

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

consisting of:

1. Q: a finite set of states.
2. Σ: a finite set of input symbols.
3. $\delta : (q, s) \in Q \times \Sigma \longmapsto q' \in Q$: a transition function (or transition table) specifying, for each state q and each input symbol s, the next state q' of the automaton.
4. $q_0 \in Q$: the initial state.
5. $F \subset Q$: a set of accepting states.

Example 3.3. Consider the example used in the introduction of this section, namely the search for the sequence "abc" (see Listing 3.1). The corresponding finite state automaton is:

	a	b	c
$\rightarrow q_0$	q_1	q_0	q_0
q_1	q_1	q_2	q_0
q_2	q_1	q_0	q_3
$^*q_3$	q_3	q_3	q_3

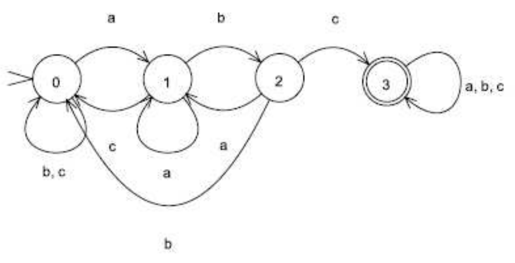


Figure 3.2: State diagram of a finite state automaton accepting the strings containing "abc".

Nondeterministic Finite Automata

Definition 3.4. A *nondeterministic finite automaton* is a five-tuple

$$(Q, \Sigma, \hat{\delta}, q_0, F)$$

consisting of:

1. Q: a finite set of states.
2. Σ: a finite set of input symbols.
3. $\hat{\delta} : (q, s) \in Q \times \Sigma \longmapsto \{q_i, q_j, \ldots\} \subseteq Q$: a transition function (or transition table) specifying, for each state q and each input symbol s, the next state(s) $\{q_i, q_j, \ldots\}$ of the automaton.
4. $q_0 \in Q$: the initial state.
5. $F \subset Q$: a set of accepting states. An input w is accepted if the automaton, after reading the last symbol of the input, is in *at least one* accepting state.

Example 3.5. The transition table $\hat{\delta}$ of the nondeterministic finite automaton shown in Figure 3.3 is the following:

	a	b	c
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$	$\emptyset$
q_2	$\emptyset$	$\emptyset$	$\{q_3\}$
$^*q_3$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$

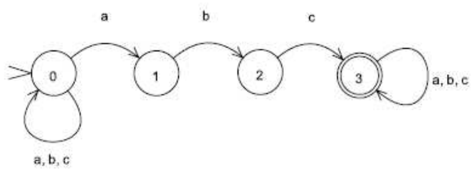


Figure 3.3: State diagram of a nondeterministic finite automaton accepting the strings containing "abc".

Theorem 3.1. *Every language that can be described by an NFA (i.e. the set of all strings accepted by the automaton) can also be described by some DFA.*

regular languages

In summary, it can be shown that any regular language satisfies the following equivalent properties:

- It can be generated by a *regular grammar*.
- It can be described by a *regular expression*.
- It can be accepted by a *deterministic finite automaton*.
- It can be accepted by a *nondeterministic finite finite automaton*.

The "Pumping Lemma" for Regular Languages

Lemma 3.1. *Let L be a regular language. Then, there exists a constant n (which depends on L) such that for every string $w \in L$ with n or more symbols ($|w| \geq n$), we can break w into three strings, $w = xyz$, such that:*

1. $|y| > 0$ *(i.e. $y \neq \epsilon$)*
2. $|xy| \leq n$
3. *For all $k \geq 0$, the string xy^kz is still in L. ($y^k := \underbrace{yy \dots y}_{k\ times}$)*

In other words, for every string w of a certain minimal length, we can always find a nonempty substring y of w that can be "pumped"; that is, repeating y any number of times, or deleting it (the case of $k = 0$), keeps the resulting string in the language L.

proof is in the script... not really important

examples for applications of finite state automata:

- Regular expressions
- Software for scanning large bodies of text, such as collections of web pages, to find occurrences of words or patterns
- Communication protocols (such as TCP)
- Protocols for the secure exchange of information
- Stateful firewalls (which build on the 3-way handshake of the TCP protocol)
- Lexical analyzer of compilers

Push-Down Automata

=> In general the same like FSA, but with a external memory (stack) to remember states:

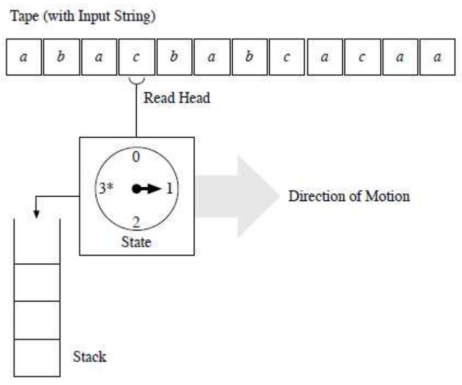


Figure 3.6: Illustration of a push-down automaton.

Parser Generator

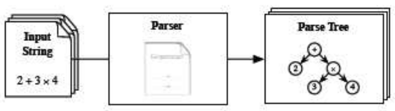


Figure 3.7: A parser transforms input strings into parse trees according to a given grammar.

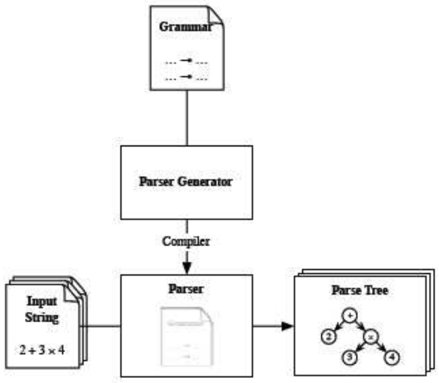


Figure 3.8: A parser generator creates a parser from a given grammar.

Example 3.7. `yacc` is a parser generator that uses a syntax close to the EBNF. The following (partial) listing illustrates an input grammar that can be transformed into a parser for simple mathematical expressions using integers, the operators "$+$", "$-$" and "$\times$" as well as parenthesis "(" and ")":

```
%token DIGIT

  line : expr
       ;

  expr : term
       | expr '+' term
       | expr '-' term
       ;

  term : factor
       | term 'x' factor
       ;

factor : number
       | '(' expr ')'
       ;

number : DIGIT
       | number DIGIT
```

Compiler Compilers

A *compiler compiler* is a program that

- takes as input a grammar complemented with atomic *actions* for each of its production rule, and
- generates a compiler (or to be more exact, a interpreter) that not only checks for any input string the correctness of its syntax, but also evaluates it.

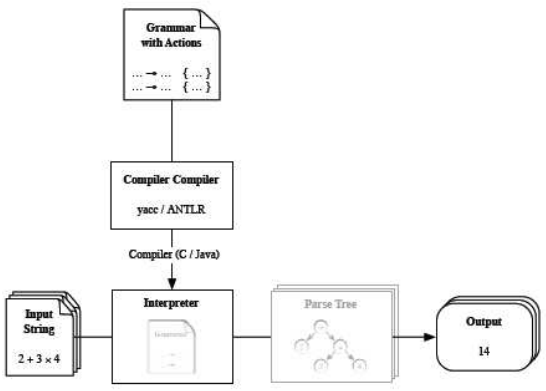


Figure 3.9: Overview of what a compiler compiler does. From a given grammar complemented with atomic actions (the "what"), it automatically generates a interpreter that checks the syntax and evaluates any number of input string (the "how").

Example 3.8. `yacc` is in fact also a compiler compiler (it is the acronym of "yet another compiler compiler"). The following lines illustrates how each production rule of the grammar provided in Example 3.7 can be complemented with atomic actions:

```
  line : expr               { printf("result: %i\n", $1); }
       ;

  expr : term               { $$ = $1; }
       | expr '+' term      { $$ = $1 + $3; }
       | expr '-' term      { $$ = $1 - $3; }
       ;

  term : factor             { $$ = $1; }


       | term 'x' factor    { $$ = $1 * $3; }
       ;

factor : number             { $$ = $1; }
       | '(' expr ')'       { $$ = $2; }
       ;

number : DIGIT              { $$ = $1; }
       | number DIGIT       { $$ = 10 * $1 + $2; }
```

The point of this example is that the above listing provides in essence all what is required to create a calculator program that correctly evaluate any input string corresponding to a mathematical expression, such as "$2 + 3 \times 4$" or "$((12 - 4) \times 7 + 42) \times 9$".

=> ANTLR better than yacc, because it provides very powerful framework for constructing recognizers, interpreters, compilers and translators from grammatical description containing actions, but also generates a very transparent code.

Turing Machines

Definition 3.5. A *Turing machine* is an automaton with the following properties:

- A **tape** with the input string initially written on it. Note that the tape can be potentially infinite on both sides, but the number of symbols written at any time on the tape is always finite.
- A **read-write head**. After reading the symbol on the tape and overwriting it with another symbol (which can be the same), the head moves to the next character, either on the left or on the right.
- A **finite controller** that specifies the behavior of the machine (for each state of the automaton and each symbol read from the tape, what symbol to write on the tape and which direction to move next).
- A **halting state**. In addition to moving left or right, the machine may also *halt*. In this case, the Turing machine is usually said to *accept* the input. (A Turing machine is thus an automaton with only one accepting state H.)

The initial position of the Turing machine is usually explicitly stated (otherwise, the machine can for instance start by moving first to the left-most symbol).

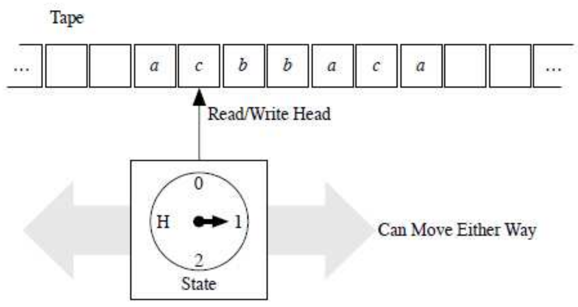


Figure 3.10: Illustration of a Turing machine.

Example 3.10. The reader is left with the exercise of finding out what the following Turing machine does on a tape containing symbols from the alphabet $\{a, b, c\}$.

The rows correspond to the state of the machine. Each cell indicates the symbol to write on the tape, the direction in which to move (L or R) and the next state of the machine.

	a	b	c	␣
0	a L 0	b L 0	c L 0	␣ R 1
1	b R 1	a R 1	c R 1	Halt

Recursively Enumerable Languages

Definition 3.6. A *recursively enumerable set* is a set whose members can simply be numbered. More formally, a recursively enumerable set is a set for which there exist a mapping between every of its elements and the integer numbers.

Linear Bounded Turing Machines

The automaton that corresponds to context-sensitive grammars (i.e. that can recognize elements of context-sensitive languages) are so-called *linear bounded Turing machines*. Such a machine is like a Turing machine, but with one restriction: the length of the tape is only $k \cdot n$ cells, where n is the length of the input, and k a constant associated with the machine.

Universal Turing Machines

Definition 3.7. A *universal Turing machine* U is a Turing machine which, when supplied with a tape on which the code of some Turing machine M is written (together with some input string), will produce the same output as the machine M.

Multitape Turing Machines

=> important, because it is easier to see how a multitape Turing machine can simulate real computers, compared with a single-tape model.

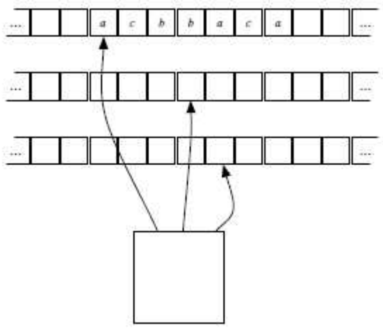


Figure 3.11: Illustration of a multitape Turing machine. At the beginning, the input is written on one of the tape. Each head can move independently.

Nondeterministic Turing Machines

=> the NTM are to standard TM what NFA are to DFA

Theorem 3.2. *If M_N is a nondeterministic Turing machine, then there is a deterministic Turing machine M_D such that $L(\mathrm{M}_N) = L(\mathrm{M}_D)$.*

The P = NP Problem

=> how the number of steps to perform a computation grows with input increasing length

Example 3.11. The *subset-sum problem* is an example of a problem which is easy to verify, but whose answer is believed (but not proven) to be difficult to compute.

Given a set of integers, does some nonempty subset of them sum to 0? For instance, does a subset of the set $\{-2, -3, 15, 14, 7, -10\}$ add up to 0? The answer is yes, though it may take a while to find a subset that does, depending on its size. On the other hand, if someone claims that the answer is "yes, because $\{-2, -3, -10, 15\}$ add up to zero", then we can quickly check that with a few additions.

Definition 3.8. P is the complexity class containing decision problems which can be solved by a deterministic Turing machine using a polynomial amount of computation time, or polynomial time.

Definition 3.9. NP – nondeterministic, polynomial time – is the set of decision problems solvable in polynomial time on a nondeterministic Turing machine. Equivalently, it is the set of problems whose solutions can be "verified" by a deterministic Turing machine in polynomial time.

The relationship between the complexity classes P and NP is a famous unsolved question in theoretical computer science. It is generally agreed to be the most important such unsolved problem; the Clay Mathematics Institute has offered a US$1 million prize for the first correct proof.

In essence, the P = NP question asks: if positive solutions to a "yes or no" problem can be verified quickly (where "quickly" means "in polynomial time"), can the answers also be computed quickly?

The Church-Turing Thesis

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

Due to the vagueness of the concept of effective calculability, the Church–Turing thesis cannot formally be proven. There exist several variations of this thesis, such as the following *Physical Church–Turing thesis*:

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

The Halting Problem

It is nevertheless possible to formally define functions that are not computable. One of the best known example is the halting problem: given the code of a Turing machine as well as its input, decide whether the machine will halt at some point or loop forever.

The Chomsky Hierarchy Revisited

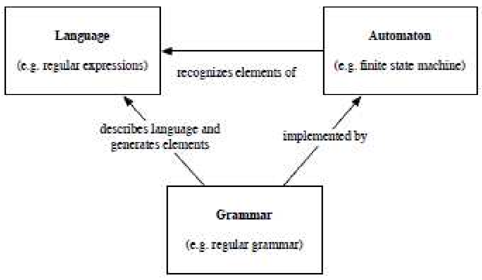

Language
(e.g. regular expressions)
recognizes elements of
Automaton
(e.g. finite state machine)
describes language and generates elements
implemented by
Grammar
(e.g. regular grammar)

Figure 3.12: Relation between languages, grammars and automata.

Type	Grammar	Language	Automaton
0	Unrestricted $\alpha \to \beta$	Recursively Enumerable e.g. $\{a^n \mid n \in \mathbb{N}, n \text{ perfect}\}$	Turing Machine
1	Context-Sensitive $\alpha A \beta \to \alpha \gamma \beta$	Context-Sensitive e.g. $\{a^n b^n c^n \mid n \geq 0\}$	Linear Bounded Turing Machine
2	Context-Free $A \to \gamma$	Context-Free e.g. $\{a^n b^n \mid n \geq 0\}$	Push-Down Automaton
3	Regular $A \to \epsilon \mid a \mid aB$	Regular e.g. $\{a^m b^n \mid m, n \geq 0\}$	Finite State Automaton

Table 3.1: The Chomsky hierarchy.

Chapter 4: Markov Processes

=> a Markov process is a stochastic extension of a FSA. In a Markov process, state transitions are probabilistic, and there is -in contrast to a FSA- no input to the system. The system is only in one state at each time step.

Process Diagrams

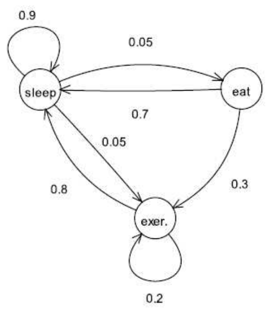


Figure 4.1: Process diagram of a Markov process.

Formal Definitions

Definition 4.1. A *Markov chain* is a sequence of random variables $X_1, X_2, X_3, \ldots$ with the *Markov property*, namely that the probability of any given state X_n only depends on its immediate previous state X_{n-1}. Formally:

$$P(X_n = x \mid X_{n-1} = x_{n-1}, \ldots, X_1 = x_1) = P(X_n = x \mid X_{n-1} = x_{n-1})$$

where $P(A \mid B)$ is the probability of A given B.

transition matrix

Example. The state space of the "hamster in a cage" Markov process is:

$$S = \{\text{sleep}, \text{eat}, \text{exercise}\}$$

and the transition matrix:

$$\mathbf{P} = \begin{pmatrix} 0.9 & 0.7 & 0.8 \\ 0.05 & 0 & 0 \\ 0.05 & 0.3 & 0.2 \end{pmatrix}$$

The transition matrix can be used to predict the probability distribution $\mathbf{x}^{(n)}$ at each time step n. For instance, let us assume that Cheezit is initially sleeping:

$$\mathbf{x}^{(0)} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

After one minute, we can predict:

$$\mathbf{x}^{(1)} = \mathbf{P} \cdot \mathbf{x}^{(0)} = \begin{pmatrix} 0.9 \\ 0.05 \\ 0.05 \end{pmatrix}$$

Thus, after one minute, there is a 90% chance that the hamster is still sleeping, 5% chance that he's eating and 5% that he's running in the wheel.

Similarly, we can predict that after two minutes:

$$\mathbf{x}^{(2)} = \mathbf{P} \cdot \mathbf{x}^{(1)} = \begin{pmatrix} 0.885 \\ 0.045 \\ 0.07 \end{pmatrix}$$

Definition 4.2. The *process diagram* of a Markov chain is a directed graph describing the Markov process. Each node represent a state from the state space. The edges are labeled by the probabilities of going from one state to the other states. Edges with zero transition probability are usually discarded.

Stationary Distribution

=> after certain time the probability distribution converges towards a stationary distribution

Example. The stationary distribution of the hamster

$$\mathbf{x}^* = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

can be obtained using Equation 4.1, as well as the fact that the probabilities add up to $x_1 + x_2 + x_3 = 1$. We obtain:

$$\mathbf{x}^* = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ 1 - x_1 - x_2 \end{pmatrix} = \begin{pmatrix} 0.9 & 0.7 & 0.8 \\ 0.05 & 0 & 0 \\ 0.05 & 0.3 & 0.2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ 1 - x_1 - x_2 \end{pmatrix}$$

From the first two components, we get:

$$\begin{aligned} x_1 &= 0.9x_1 + 0.7x_2 + 0.8(1 - x_1 - x_2) \\ x_2 &= 0.05x_1 \end{aligned}$$

Combining the two equations gives:

$$0.905x_1 = 0.8$$

so that:

$$\begin{aligned} x_1 &= \frac{0.8}{0.905} \approx 0.89 \\ x_2 &= 0.05x_1 \approx 0.044 \\ x_3 &= 1 - x_1 - x_2 \approx 0.072 \\ \mathbf{x}^* &\approx \begin{pmatrix} 0.89 \\ 0.044 \\ 0.072 \end{pmatrix} \end{aligned}$$

In other words, if we observe the hamster long enough, the probability that it will be sleeping is $x_1 = 89\%$, that it will be eating $x_2 = 4\%$, and that it will be doing some exercise $x_3 = 7\%$.

Hidden Markov Models

=> Markov process with unknown parameters. In HMM states are not visible like in MM => the observer sees an observable (or output) token. Each hidden state has a probability distribution, called *emission probability*, over the possible observable tokens.

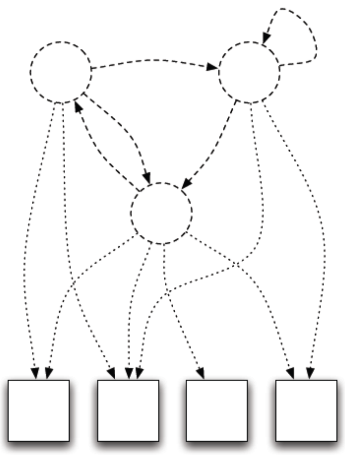


Figure 4.2: Illustration of a hidden Markov model. The upper part (dashed) represents the underlying hidden Markov process. The lower part (shadowed) represents the observable outputs. Dotted arrows represent the emission probabilities.

Example 4.1. Assume you have a friend who lives far away and to whom you talk daily over the telephone about what he did that day. Your friend is only interested in three activities: walking in the park, shopping, and cleaning his apartment. The choice of what to do is determined exclusively by the weather on a given day. You have no definite information about the weather where your friend lives, but you know general trends. Based on what he tells you he did each day, you try to guess what the weather must have been like.

You believe that the weather operates as a discrete Markov process (i.e. a Markov chain). There are two states, "Rainy" and "Sunny", but you cannot observe them directly – they are hidden from you. On each day, there is a certain chance that your friend will perform one of the following activities, depending on the weather: "walk", "shop', or "clean". Since your friend tells you about his activities, those are the observations.

Furthermore, you know the general weather trends in the area, and what your friend likes to do on average.

Concerning the weather, you known that:

- If one day is rainy, there is a 70% chance that the next day will be rainy too.
- If one day is sunny, there is 40% chance that the weather will degrade on the next day.

Your friend's general habits can be summarized as follows:

- If it is rainy, there is a 50% chance that he is cleaning his apartment, and only 10% chance that he goes out for a walk.
- If it is sunny, there is a 60% chance that he is outside for a walk, 30% chance that he decides to go shopping and 10% that he stays home to clean his apartment.

The entire system is that of a hidden Markov model (HMM). The hidden state space is $S = \{\text{Rainy}, \text{Sunny}\}$, and the possible observable output state $O = \{\text{walk}, \text{shop}, \text{clean}\}$. The transition probability matrix (between hidden states) is:

$$\mathbf{P} = \begin{pmatrix} 0.7 & 0.4 \\ 0.3 & 0.6 \end{pmatrix}$$

Finally, the emission probabilities are:

	Rainy	Sunny
walk	0.1	0.6
shop	0.4	0.3
clean	0.5	0.1

The whole model can also be summarized with the following diagram:

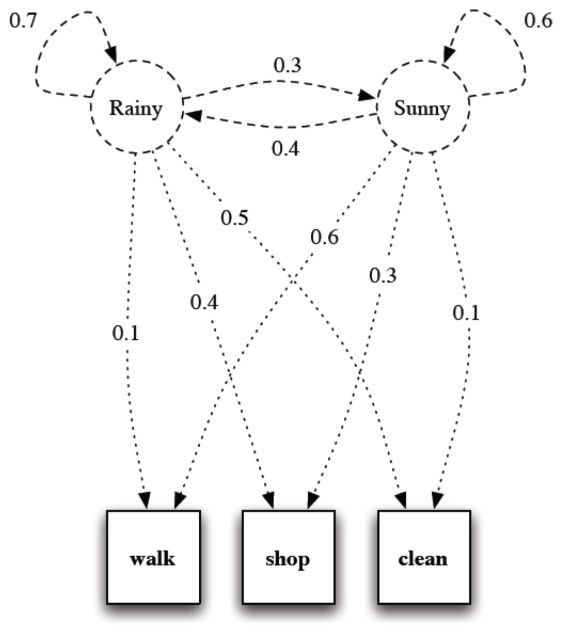


Figure 4.3: Diagram of a hidden Markov model.

Applications of Hidden Markov models include text recognition, predictive text input systems of portable devices and speech-to-text software.

Viterbi Algorithm => I don't think that this is part of the relevant stuff... for a good summary take the script.

The *Viterbi algorithm* is an efficient algorithm that finds the most likely
sequence of hidden states given a sequence of observations.

Chapter 5: Logic

Definition of a Formal System

Definition 5.1. In logic, a *formal system* consists of:

(a) a formal language,

(b) a set of axioms,

(c) a set of inference (or transformation) rules.

The choice of these defines the power of the system (which assertions can be expressed) and its properties (completeness, decidability, etc.).

Propositional Logic

language

The language of propositional logic consists of:

- a set P of atomic formulas, consisting of e.g. symbols such as p, q, r...
- the following logical connectives:

$\neg$	negation
$\wedge$	logical "and"
$\vee$	logical "or"
$\rightarrow$	logical implication ("if ... then")
$\leftrightarrow$	equivalence ("if and only if")

- auxiliary symbols: "(" and ")"

Definition 5.2. *Propositional formulas* can then be constructed from the symbols of the language by a recursive definition:

(i) Every atomic formula $p \in P$ is a propositonal formula.

(ii) If the expressions A and B are propositional formulas, then the expressions $\neg A, (A \wedge B), (A \vee B), (A \rightarrow B), (A \leftrightarrow B)$ are propositional formulas.

(iii) Every propositional formula arises from a finite number of application of (i) and (ii).

Example 5.1. If $P = \{p, q, r\}$ is a set of atomic formulas, then the expressions

- p, q and r are propositional formulas according to (i)
- $\neg p$ and $(q \wedge r)$ are propositional formulas according to (ii)
- $(\neg p \rightarrow (q \wedge r))$ is a propositional formula according to another application of (ii)

Alternatively, propositional (or well-formed) formulas can also be generated by means of a grammar:

$$\begin{array}{rcl}
\langle\text{formula}\rangle & \rightarrow & \langle\text{atomic formula}\rangle \,|\langle\text{ propositional formula}\rangle \\
\langle\text{atomic formula}\rangle & \rightarrow & T \mid F \mid p \mid q \mid r \mid \ldots \\
\langle\text{propositional formula}\rangle & \rightarrow & (\ \langle\text{formula}\rangle\ ) \\
 & | & \langle\text{formula}\rangle\langle\text{connector}\rangle\langle\text{formula}\rangle \\
 & | & \neg\langle\text{formula}\rangle \\
\langle\text{connector}\rangle & \rightarrow & \wedge \mid \vee \mid \rightarrow \mid \leftrightarrow
\end{array}$$

where T and F stand for the logical "true" and "false" values.

This grammar still contains some ambiguity as long as the priority of connectors is not defined. Priority is defined as follows:

1. The negation $\neg$ has the highest priority,
2. followed by the logical "and" ($\wedge$),
3. the logical "or" ($\vee$),
4. and the logical implications ($\rightarrow$, $\leftrightarrow$).

Alternatively, parenthesis can be used, which also makes a logical expression more readable. For instance, $A \wedge \neg B \vee C \rightarrow D$ can also be written as $((A \wedge (\neg B)) \vee C) \rightarrow D$.

Now we are able to construct propositional (or well-formed) formulas.

Semantics

Definition 5.3. A propositional formula A is *satisfiable*, if there exists an interpretation of its atomic formulas (assigning truth values to all of them) such that A is true.

Definition 5.4. Let M be a set of formulas. If a fomula A is true in every interpretation in which all formulas of M are true, then A is a *tautologic consequence* of M and we write $M \models A$.

Definition 5.5. If A is true in all interpretations, it is called a *tautology*. This is written as $\models A$. In other words, $\models A$ implies that $M \models A$ is valid for any M. An example of a tautology is $P \vee \neg P$.

Definition 5.6. *Contradiction*: If a statement is always false for all interpretations, it is called a contradiction. An example of contradiction is $P \wedge \neg P$.

Formal System

axioms

A particularly compact and well-known axiom system for propositional logic is the following (after Jan Lukasiewicz):

$$p \rightarrow (q \rightarrow p) \quad \text{(A1)}$$
$$(p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r)) \quad \text{(A2)}$$
$$(\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p) \quad \text{(A3)}$$

Note that other axiom systems are also possible.

Inference Rules

Propositional logic has a single inference rule: Modus ponens.

Modus Ponens

$$\begin{array}{l} p \rightarrow q \\ p \\ \hline q \end{array}$$

Example 5.2. Let us replace p with `bad_weather` and q with `I_stay_home`. If we assume the axioms `bad_weather` $\rightarrow$ `I_stay_home` as well as `bad_weather`, we can derive, using modus ponens, the formula `I_stay_home`:

$$\begin{array}{l} \texttt{bad\_weather} \rightarrow \texttt{I\_stay\_home} \\ \texttt{bad\_weather} \\ \hline \texttt{I\_stay\_home} \end{array}$$

All theorems of propositional logic can be derived from the three axioms and the single inference rule.

proofs

Definition 5.7.

(i) A finite sequence of formulas $A_1, A_2, \ldots, A_n$ is the *proof* of a formula A, if A_n is the formula A and for any i, formula A_i is either an axiom or it is derived from previous formulas $A_j \ (j < i)$ by modus ponens.

(ii) If there exists a proof of formula A, we say that A is provable in propositional logic and we write $\vdash A$.

Definition 5.8. *Proof from assumptions.* Let T be a set of formulas. We say that formula A is provable from T and write $T \vdash A$ if A can be proven from axioms and formulas in T, by using the inference rule. That is we have basically enriched the set of axioms by the formulas in T.

theorems

de Morgan's Rules

1. $(\neg(P \vee Q)) \leftrightarrow (\neg P) \wedge (\neg Q)$

2. $(\neg(P \wedge Q)) \leftrightarrow (\neg P) \vee (\neg Q)$

Distributive Laws

1. $(P \vee (Q \wedge R)) \leftrightarrow ((P \vee Q) \wedge (P \vee R))$

2. $(P \wedge (Q \vee R)) \leftrightarrow ((P \wedge Q) \vee (P \wedge R))$

completeness

Theorem 5.1 (Post)**.** *For every propositional formula A*

$$\vdash A \quad \textit{if and only if} \quad \models A$$

This means that, in propositional logic, tautologies are provable, and what is provable is a tautology. Thus, the formal system of propositional logic is not only sound (i.e. generates only valid formulas) but also generates all of them.

Theorem 5.2 (completeness of propositional logic)**.** *Let T be a set of formulas and A a formula. Then*

$$T \vdash A \quad \textit{if and only if} \quad T \models A$$

This is a more general version of Post's theorem. In a sense, completeness implies that loosely speaking syntax and semantics are equivalent in this case. This is by no means true for any formalism (see below).

Normal Forms of Propositional Formulas

Every propositional formula can be expressed in two standard or normal forms:

Definition 5.9 (Conjunctive normal form – CNF)**.** A formula in CNF has the following form:

$$(A_1 \vee \ldots \vee A_N) \wedge (B_1 \vee \ldots \vee B_M) \wedge \ldots$$

Definition 5.10 (Disjunctive normal form – DNF)**.** A formula in DNF has the following form:

$$(A_1 \wedge \ldots \wedge A_N) \vee (B_1 \wedge \ldots \wedge B_M) \vee \ldots$$

Predicate Calculus (First Order Logic)

language

Definition 5.11. *First order language* contains:

(i) An unlimited number of symbols for variables: $x, y, z, \ldots$

(ii) Symbols for logical connectives: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$.

(iii) Symbols for quantifiers: the *universal* quantifier $\forall$ ("for all"), and the *existential* quantifier $\exists$ ("there exists").

(iv) Symbols for predicates: $p, q, \ldots$

Predicates are relations. The *arity* of a predicate symbol specifies the number of arguments of the predicate. For example, equality "=" is a binary predicate (its arity is 2).

(v) Symbols for functions: $f, g, \ldots$

The *arity* of a function symbol specifies the number of arguments of the function. Function symbols of arity 0 are constants.

(vi) Auxiliary symbols: "(", ")"

This language is called first order because one can only quantify over variables (for individuals), such as in $(\forall P)\texttt{inhabitant\_of\_Prague}(P) \rightarrow \texttt{mortal}(P)$. However, one cannot quantify over predicates, i.e. one cannot say "$\forall$ inhabitants of Prague".

Definition 5.12 (Expression)**.** An *expression* is any sequence of symbols of a particular language.

Definition 5.13 (Term)**.** An *term* is an expression defined recursively as follows:

(i) Every variable is a term.

(ii) If the expressions $t_1, \dots, t_n$ are terms and f is an n-ary function symbol, then $f(t_1, \dots, t_n)$ is a term.

(iii) Every term arises from a finite number of applications of (i) and (ii).

Example 5.6. In number theory $x, x + y$, are terms. The latter term could also be written as $+(x, y)$, where + is our f (but it is conventional to write these binary predicates in infix notation).

Definition 5.14 (Formula)**.** A *formula* is defined recursively as follows:

(i) If p is an n-ary predicate symbol and the expressions $t_1, \dots, t_n$ are terms, then the expression $p(t_1, \dots, t_n)$ is an atomic formula.

(ii) If the expressions A and B are formulas, then the expressions $\neg A, (A \wedge B), (A \vee B), (A \rightarrow B), (A \leftrightarrow B)$ are formulas.

(iii) If x is a variable and A a formula, then $(\forall x)A, (\exists x)A$ are formulas.

(iv) Every formula arises from a finite number of applications of (i) to (iii).

Definition 5.15 (Theorem)**.** A *theorem* is a formula that is valid, i.e. a formula that is logically true in the given formal system.

Semantics

Scope of a Quantifier

The definition of the scope of a quantifier is illustrated in the following example.

Example 5.8. For every human x there exists a human y that loves x. Stated formally:

$$\forall x, \underbrace{(\text{human}(x) \rightarrow \exists y \underbrace{(\text{human}(y) \wedge \text{loves}(x, y)))}_{\text{scope of } y}}_{\text{scope of } x}$$

Definition 5.16.

(i) A given occurrence of a variable x in a formula A is *bounded*, if it is part of a subformula of A (i.e. a substring of A that is also a formula) of the form $(\exists x)B$ or $(\forall x)B$. If an occurrence is not bounded, it is *free*.

(ii) A variable is *free* in A, if it has a free occurrence there.

A variable is *bounded* in A, if it has a bounded occurence there.

(iii) Formula A is *open*, if it does not contain any bounded variable.

Formula A is *closed*, if it does not contain any free variable.

Example 5.9. Formula A:

$$(\forall x)(x \rightarrow y)$$

In formula A, x has a bounded occurrence by the quantifier $\forall$, and hence it is bounded in A, whereas y is not quantified and hence it has a free occurrence and thus is free in A. Formula A is neither open nor closed.

Formal System

1a) Axioms for logical connectives

(A1) – (A3) from propositional calculus

Thus, the whole propositional logic becomes a 'subset' of predicate logic. Tautologies of propositional calculus are automatically theorems of predicate calculus.

1b) Inference rule: Modus ponens

2) Axioms for quantifiers

2a) Specification scheme: Let A be a formula, x a variable and t a term that can be substituted for x into A

$$(\forall x)A \rightarrow A_x[t]$$

2b) "Jump scheme:" A, B are formulas, x a variable which is not free in A, then

$$(\forall x)(A \rightarrow B) \rightarrow (A \rightarrow (\forall x)B)$$

Comment: This is a rather technical axiom, to be used in prenex operations.

3) Inference rule: Universal generalization For an arbitrary variable x, from a formula A, derive $(\forall x)A$.

Comment: This shows the role of free variables in theorems. Whenever you can prove a formula A with a free variable x, then you can prove also a formula $(\forall x)A$. This is because, from a semantic point of view, for free variables you would have to check all possible interpretations anyway.

Rules of Manipulation

Permutation

$$\forall x(\forall y(P(x,y))) \leftrightarrow \forall y(\forall x(P(x,y)))$$

A similar rule can be shown for the existential quantifier.

Negation

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

For the negation of the universal quanitifer it suffices to show that there exists one case for which $\neg P(x)$.

Nesting/Applicability

$$(\forall x : P(x)) \wedge Q \leftrightarrow \forall x : (P(x) \wedge Q)$$

Here, x appears in P, but not in Q. Therefore it does not affect the truth value of Q when it is grouped with P with respect to x. Similar argumentation holds true for the existential quantifier.

Prenex normal form

is the normal form for predicate calculus (like CNF and DNF for propositional calculus)

Definition 5.17. We say that formula A is in *prenex form*, if it has the following form:

$$(Q_1 x_1) \ldots (Q_n x_n) B$$

where

1. Q_i are either $\forall$ or $\exists$
2. B is an open formula (i.e. all variables are free in it)
3. $x_1 \ldots x_n$ are all different

B is called an open core of A and the sequence of quantifiers is called prefix.

Chapter 6: Cellular Automata

Cell Lattice

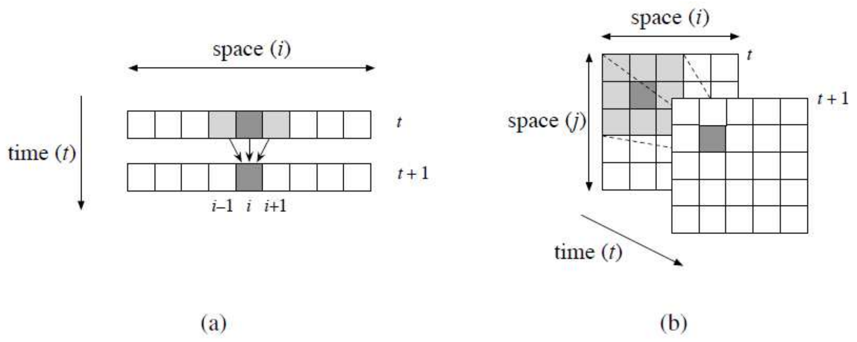


Figure 6.1: Illustration of a (a) 1-dimensional CA and (b) 2-dimensional CA. The light gray cells are the neighboring cells of the dark gray cell.

Local Rules

$a_{i-1}(t)$	$a_i(t)$	$a_{i+1}(t)$	$a_i(t+1)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Table 6.1: Example of a set of local rules for a 1-dimensional CA with a neighboring radius of $r = 1$ and $k = 2$ possible states.

Initial and Boundary Condition

initial conditions:
- seed (all cells are in the state 0 except one)
- random (initial state of each state is chosen randomly)

boundary conditions:
- fixed (It is assumed that there are “invisible” cells next to the border-cells which are in a given predefined state)
- cyclic (It is assumed that the cells on the edge are neighbors of the cells on the opposite edge as depicted)

One-Dimensional Cellular Automata

We will now explore[3] the simplest possible kind of cellular automata, namely 1-dimensional CA with a neighborhood radius of $r = 1$ and a binary state space ($k = 2$).

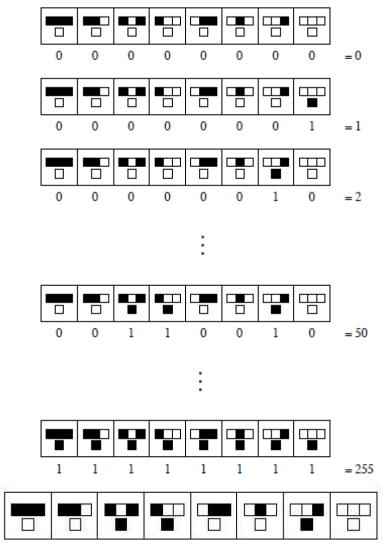


Figure 6.3: Graphical representation of the rule for the cellular automaton specified by Table 6.1.

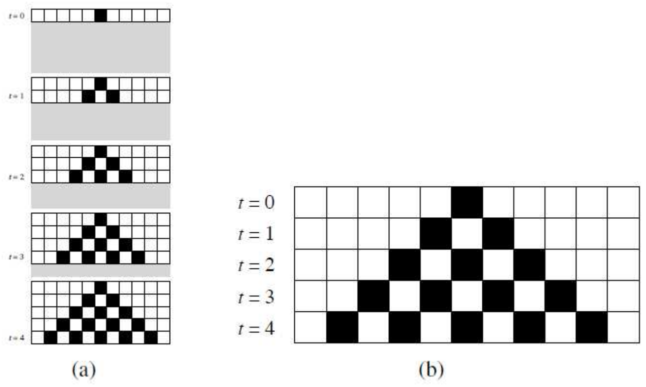


Figure 6.4: Time evolution of the cellular automaton. The individual time steps (a) can be shown with one array of cells (b).

Simple Patterns

A certain number of rules lead to simple patterns, as shown in Figure 6.6.

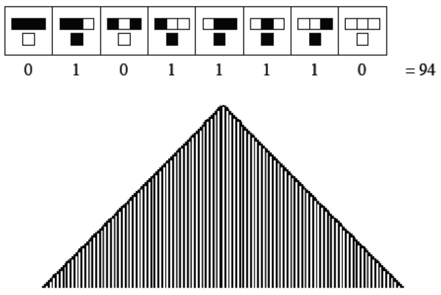


Figure 6.6: A simple repetitive pattern.

Fractals

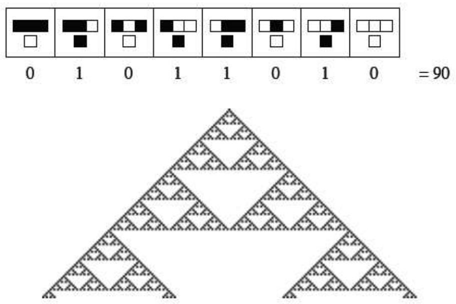


Figure 6.7: A fractal pattern.

Each of these pieces is essentially just a smaller copy of the whole pattern, with still smaller copies nested in a very regular way inside it.

Pattern with nested structure of this kind are often called *fractals* or *self-similar.*

Chaos

=> neither simple nor fractal pattern

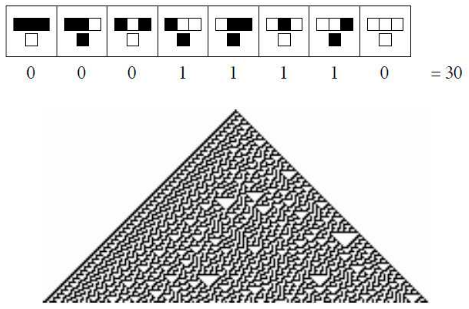


Figure 6.9: A chaotic pattern.

Edge of Chaos

=> between chaos and regularity

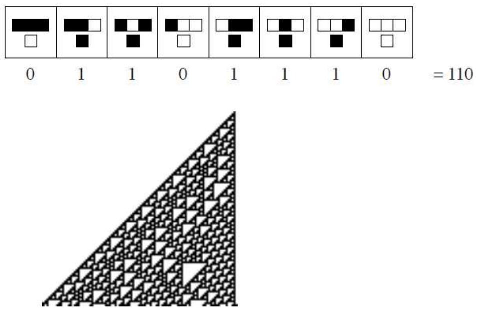


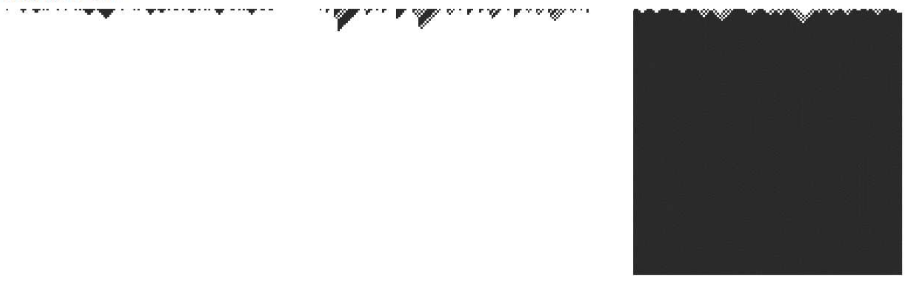
Figure 6.12: A pattern which seems neither highly regular nor completely random.

The Four Classes of Cellular Automata (by Wolfram)

6.4.1 Class 1

Figure 6.14 illustrates different patterns produced by cellular automata of class 1. In this class, the behavior is very simple, and almost all initial conditions lead to exactly the same uniform final state.

Such systems evolve to simple "limit points" in phase space (a point being one particular configuration, such as all cell in state 0), and are thus said to have *point attractors*.



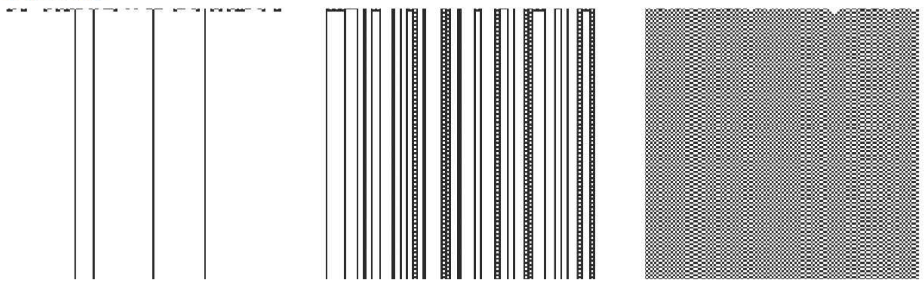
(a) rule 128 (b) rule 168 (c) rule 250

Figure 6.14: Cellular automata of class 1.

6.4.2 Class 2

Patterns produced by cellular automata belong to class 2 are shown in Figure 6.15. In this class, there are many different possible final states, but all of them consist just of a certain set of simple structures that either remain the same forever, or repeat every few steps.

Such systems evolve to "limit cycles", and are thus said to have *periodic attractors*.



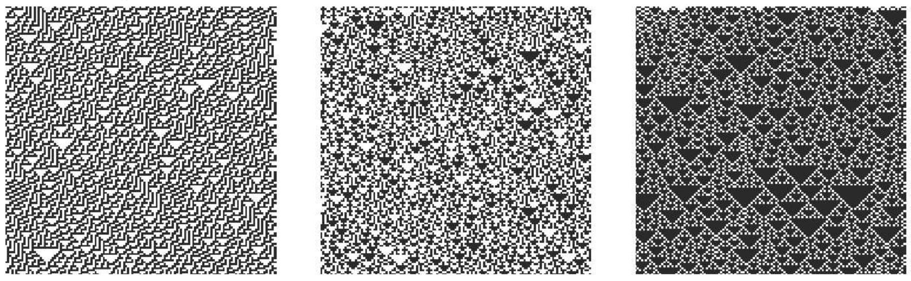
(a) rule 36 (b) rule 108 (c) rule 178

Figure 6.15: Cellular automata of class 2.

6.4.3 Class 3

In class 3, illustrated in Figure 6.16, the behavior is more complicated, and seems in mayn respects random, although triangles and other small-scale structures are essentially always at some level seen.

Such systems are said to have *strange* or *chaotic attractors*.



(a) rule 30 (b) rule 150 (c) rule 182

Figure 6.16: Cellular automata of class 3.

6.4.4 Class 4

Class 4 correspond to the "edge of chaos" between class 2 and class 3. It involves a mixture of order and randomness: localized structures are produced which on their own are failry simple, but these structures move around and interact with each other in very complicated ways (see Figure 6.17).

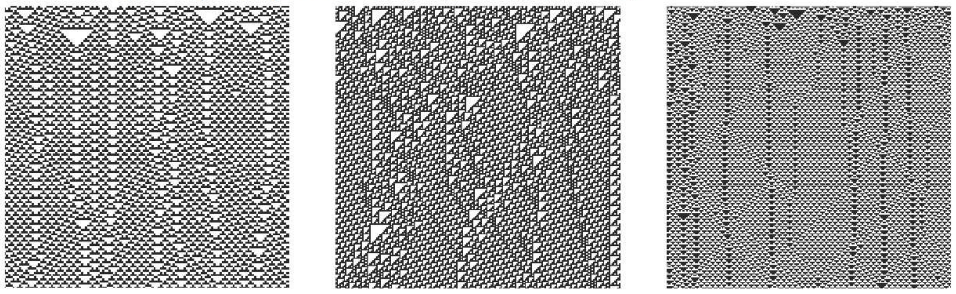

(a) rule 54 (b) rule 110 (c) rule 147

Figure 6.17: Cellular automata of class 4.

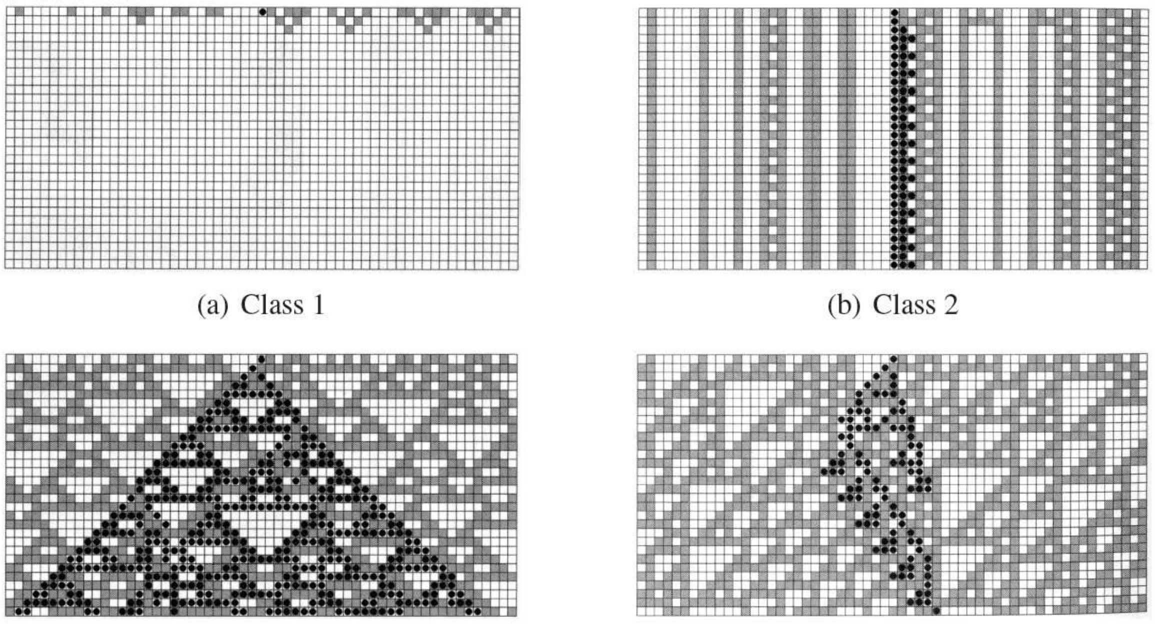
Sensitivity to Initial Conditions

Class 1: Small changes always die out, and there is no change in the final state.

Class 2: Small changes may persist, but always remain localized in small region of the system.

Class 3: Any change typically spreads at a uniform rate, eventually affecting every part of the system.

Class 4: Changes spread only when they are in effect carried by localized structures that propagate across the system. Class 4 systems are once again somewhat intermediate between class 2 and class 3.


(a) Class 1 (b) Class 2

(c) Class 3 (d) Class 4

Figure 6.18: The effect of changing the color of a single cell in the initial conditions for typical cellular automata from each of the four classes. The black dots indicate all the cells that change. (Wolfram, 2002, p. 250)

Langton's Landa-Parameter

=> If most of the entries in a rule table of a cellular automata are zero, then the pattern will most probably always converge to the empty configuration. The more entries in the rule table are different from zero, higher is the probability of obtaining complex or chaotic patterns.

$$\lambda = \frac{N - n_0}{N}$$

N = number of entries in the rule table of a CA, n0 = number of zeros in the rule table

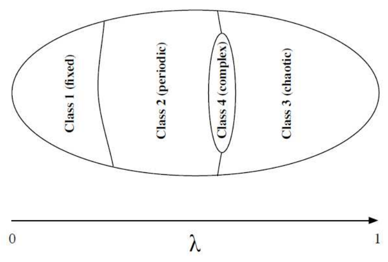


Figure 6.20: Langton's λ parameter classifies cellular automata into the four different classes. Class 4 lies at the edge of the region with chaotic regime.

Computation at the Edge of Chaos

We have seen so far the cellular automata belonging to class 4 have the interesting property of being neither too simple, nor too complex. Only in this class could be observe patterns to move, to interact with each other as well as the effect of small changes to be localized to a certain region. By viewing such moving patterns as bits of information being transmitted, stored and modified, Langton (1990) showed that the optimal conditions for the support of *computation* is found at the phase transition between the non-chaotic regime and the chaotic regime. In fact, Cook (2004) has proven that rule 110 is capable of universal computation, i.e. has the same computational power as a (universal) Turing machine!

2-Dimensional CAs (Game of Life)

Each cell of this cellular automaton can be in either one of two states, called "dead" or "alive", and obeys the following very simple set of rules:

Loneliness:
If a living cell has less than two neighbors, then it dies.

Overcrowding:
If a living cell has more than three neighbors, then it dies.

Reproduction:
If a dead cell has exactly three living neighbors, then it comes to life.

Stasis:
Otherwise, a cell stays as it is.

Many different types of patterns occur in the Game of Life, including static patterns ("still lifes"), repeating patterns ("oscillators"), and patterns that repeat themselves after a fixed sequence of states and translate themselves across the board ("spaceships").

Simple Patterns: Many different types of patterns occur in the Game of Life, including static patterns ("still lifes"), repeating patterns ("oscillators"), and patterns that repeat themselves after a fixed sequence of states and translate themselves across the board ("spaceships"). Common examples of these three classes are illustrated in Figure 6.22.

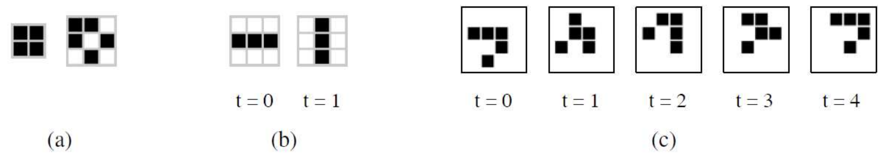


Figure 6.22: Simple patterns occurring in the Game of Life. (a) Still lifes. (b) A 2-periodic oscillator. (c) A "glider", one of the simplest spaceships.

Growing Patterns: Patterns called "Methuselahs" can evolve for long periods before disappear or stabilize (see Figure 6.23). Conway originally conjectured that no pattern can grow indefinitely – i.e., that for any initial configuration with a finite number of living cells, the population cannot grow beyond some finite upper limit. In the game's original appearance in "Mathematical Games", Conway offered a $50 (!) prize to the first person who could prove or disprove the conjecture before the end of 1970. The prize was won in November of the same year by a team from the Massachusetts Institute of Technology, led by Bill Gosper; the "Gosper gun" shown in Figure 6.24 produces a glider every 30 generation. This first glider gun is still the smallest one known.

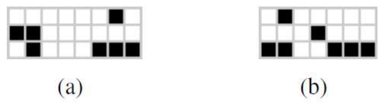


Figure 6.23: Simple patterns that grow. (a) "Diehard" is a pattern that eventually disappears after 130 generations, or steps. (b) "Acorn" takes 5206 generations to generate at least 25 gliders and stabilise as many oscillators.

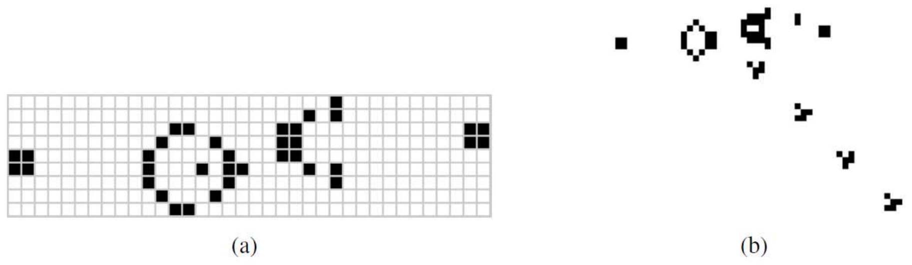


Figure 6.24: The "Gosper" glider gun. (a) Initial configuration. (b) The glider gun in action.

Universal Computation: Looking at the Game of Life from a computational point of view, gliders can be seen as bits of information being transmitted, glider guns as input to the system, and other static objects as providing the structure for the computation. For instance, Figure 6.25 illustrates how any logical primitive can be implemented in the Game of Life.
More generally, it has been shown that the Game of Life is Turing complete, i.e. that it can compute anything that a universal Turing machine can compute. Furthermore, a pattern can contain a collection of guns that combine to construct new objects, including copies of the original pattern. A "universal constructor" can be built which contains a Turing complete computer, and which can build many types of complex objects – including more copies of itself!

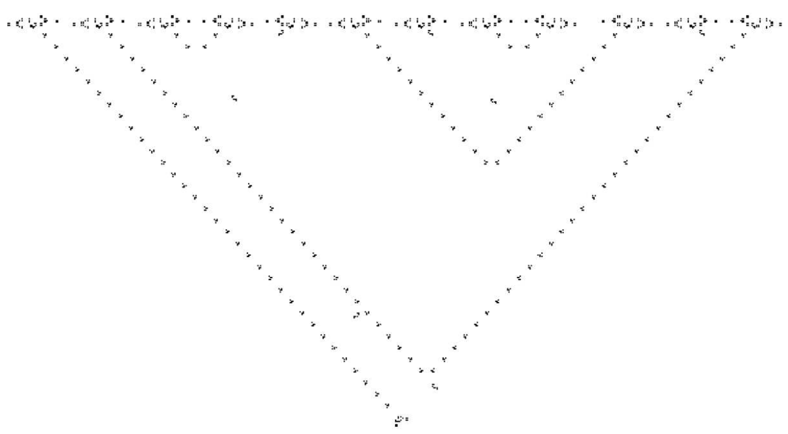
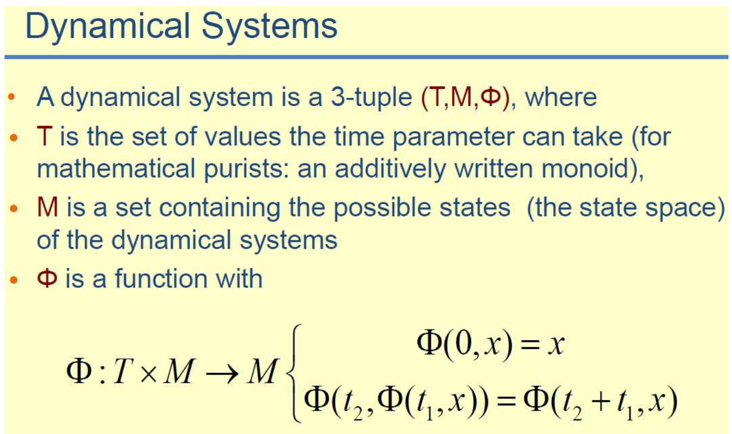


Figure 6.25: Logical operations implemented in the Game of Life. The streams of gliders represent the input of the system, and the presence or absence of glider at the very bottom the binary output of the logical operation.

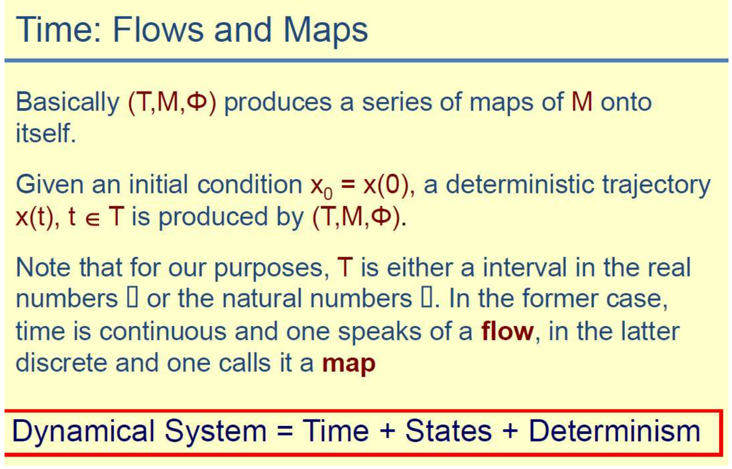
<u>Chapter 7: Dynamical Systems (nur slides kopiert, Skript war zu schwierig)</u>



Dynamical Systems

- A dynamical system is a 3-tuple (T,M,Φ), where
- T is the set of values the time parameter can take (for mathematical purists: an additively written monoid),
- M is a set containing the possible states (the state space) of the dynamical systems
- Φ is a function with

$$\Phi : T \times M \to M \begin{cases} \Phi(0,x) = x \\ \Phi(t_2, \Phi(t_1, x)) = \Phi(t_2 + t_1, x) \end{cases}$$



Time: Flows and Maps

Basically (T,M,Φ) produces a series of maps of M onto itself.

Given an initial condition $x_0 = x(0)$, a deterministic trajectory $x(t)$, $t \in T$ is produced by (T,M,Φ).

Note that for our purposes, T is either a interval in the real numbers ▯ or the natural numbers ▯. In the former case, time is continuous and one speaks of a **flow**, in the latter discrete and one calls it a **map**

Dynamical System = Time + States + Determinism

How Is the Dynamics Described?

Very many dynamical systems can be expressed as sets of ordinary **differential equations (ODE):**

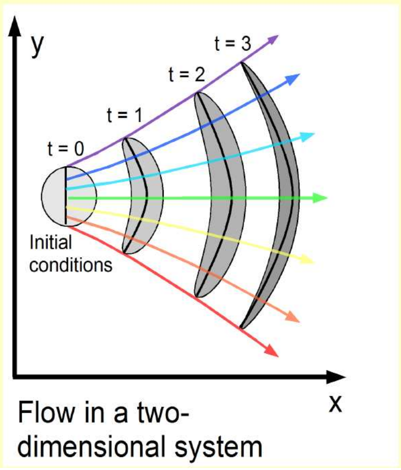
$$\begin{aligned}\dot{x}_1 &= f_1(x_1,\ldots,x_n;\lambda_1,\ldots,\lambda_m)\\ \vdots &= \vdots\\ \dot{x}_n &= f_n(x_1,\ldots,x_n;\lambda_1,\ldots,\lambda_m)\end{aligned}$$

Discrete systems are given by **iterations:**

$$x_n = f(x_{n-1}, x_{n-2}, \ldots, x_{n-m})$$

The flows and maps are global, but here, their description is local. The big technical question is how to „integrate" a local description in order to get a global flow.

A Flow in Two Dimensions



Flow in a two-dimensional system

A set of initial conditions, here a circle is transported through M by a flow.

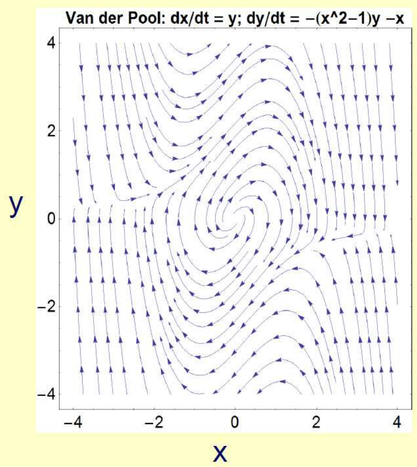
The flow deforms the initial region, but for each and every initial condition we have a trajectory.

If the system dissipates energy, these areas will shrink.

ODE, Iterations, Development

- Iterations and systems of ODE doesn't describe Φ in the sense of a blue print.
- They are rather developmental descriptions for the construction of Φ.

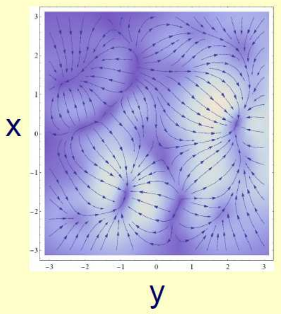
Visualization of a Two-Dimensional Flow

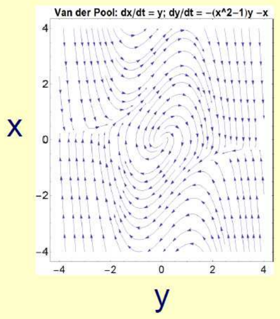


$$\dot{x} = y$$
$$\dot{y} = -(x^2-1)y - x$$

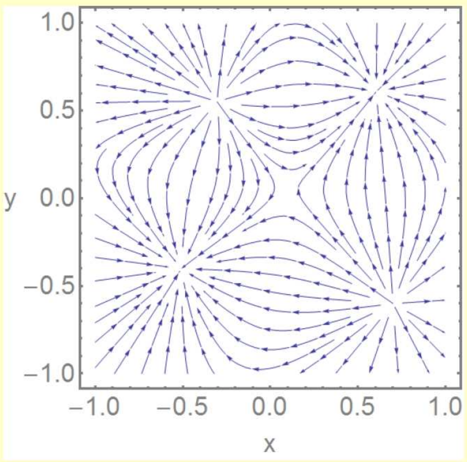
Attractors: Informal Definitions

After sufficiently long time, many systems tend to attain only a very small subset of their potential states. This small subset **attracts** all other points.



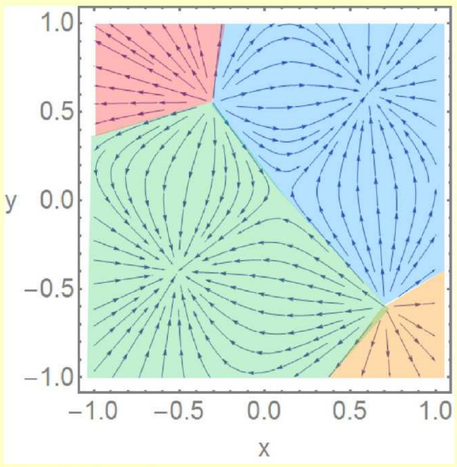


Basin of Attraction in Two Dimensions



Zeichnen Sie die Basins of Attraction!

Basin of Attraction in Two Dimensions

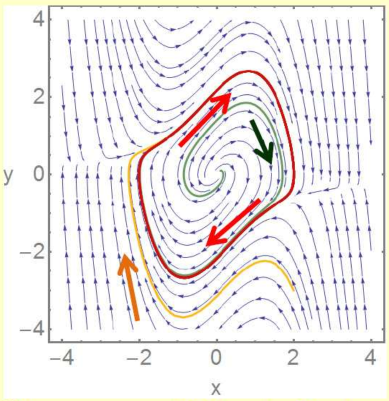


Wir können vier Basins of Attraction identifizieren!

Limit Cycles

- Systems with two or morê dimensions need not to end up in an point attractor.
- They may eventually enter an orbit that goes on forever.
- Such an orbit is called a limit cycle.

Grenzzyklen



$$\dot{x} = y$$
$$\dot{y} = -y(x^2 - 1) - x$$

Van der Pol - Oszillator

Grenzzyklus
Transiente von Aussen
Transiente von Innen

Strong and Weak Determinism

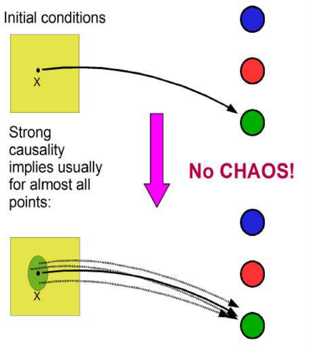
- **Weak determinism**: Identical initial conditions lead to identical results.
- **Strong determinism**: Weak determinism holds and additionally, similar conditions lead to similar results.
- Systems subject to strong determinism are easy to handle, e.g. bicycles, tin openers, computers.
- Systems subject to weak determinism are deterministic, but if one doesn't know exactly the initial conditions, the final outcome might not be predictable.

Chaos – Informal Definition

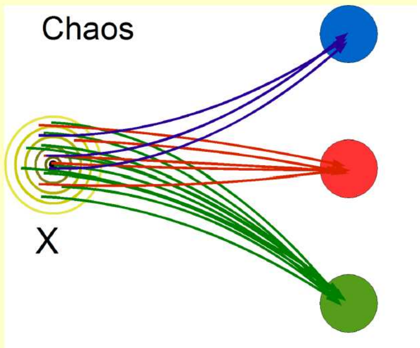
- Roughly said, systems that are subject to weak but not to strong determinism are called **chaotic**.
- Note that whether or not a system behaves in a chaotic manner may depend on some parameters. It may also be the case that the system is only chaotic for some portions of the space of potential initial conditions.

Chaos – Geometrical Perspective

System with three fixed points.



Means, if X goes green, so do his immediate buddies. This is true for almost all X's, except some borderliners. If X lifes in a n-dim space, the borderliners life on n-1 dim subsets.



Despite the fact that the fate of X is determined and leads to the „green“ fixed point, one finds in all, arbitrary small, neighborhoods of X points going to another fixed point.

Case Study: Logistic Map

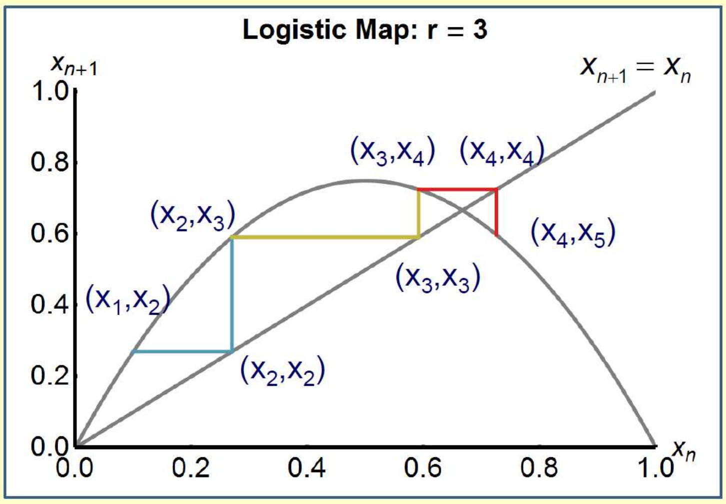
- The logistic map is a very simple ecological model.
- Two assumption:
  - Sexual reproduction ➔ $\sim$ population size
  - Competition for food ➔$\sim$ prob. of meeting a potential competitor ➔ $\sim$ (population size)^2.

$$x_{n+1} = rx_n(1-x_n)$$

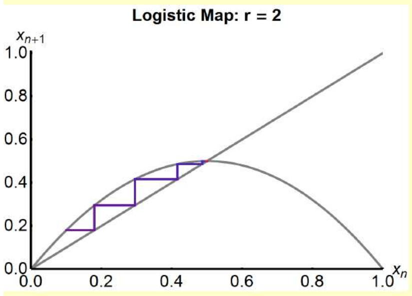
x_n: Population size at discrete time n

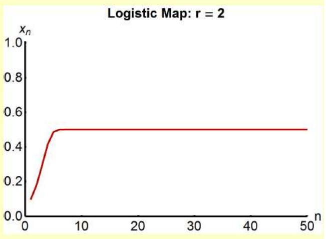
r : Parameter (after normalization)

Logistic Map Visualized



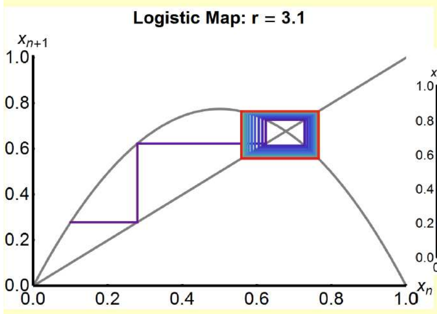
Behavior of the Logistic Map

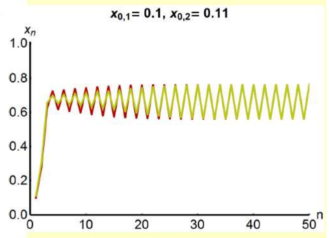




For small r: convergence to a single point.

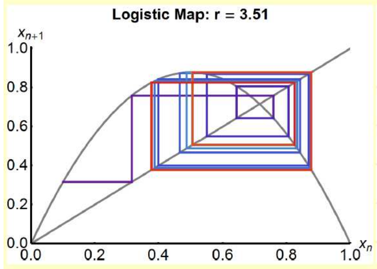
Behavior of the Logistic Map

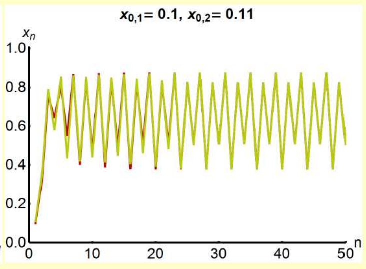




For larger r: convergence to oscillation between two points.
No dependence on initial conditions.

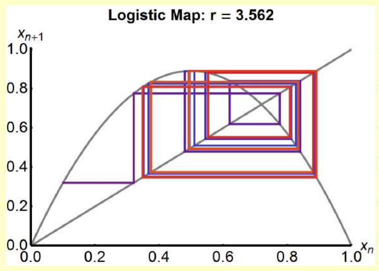
Behavior of the Logistic Map

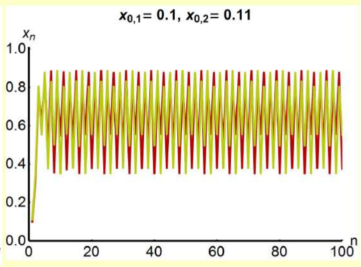




For even larger r: convergence to oscillation between four points.
No dependence on initial conditions.

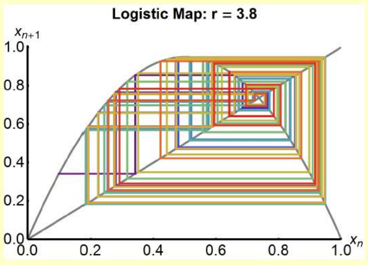
Behavior of the Logistic Map

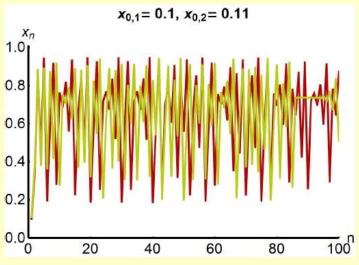




For even larger r: convergence to oscillation between eight points.
No important dependence on initial conditions.
Actually, we observe a phase shift.

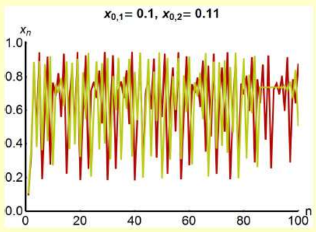
Behavior of the Logistic Map

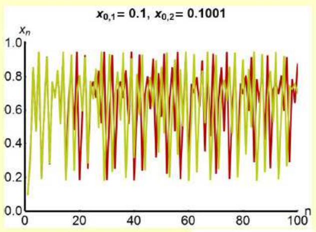


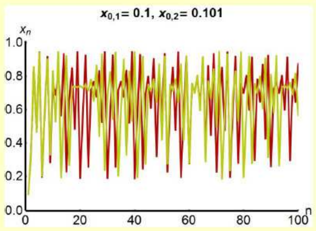


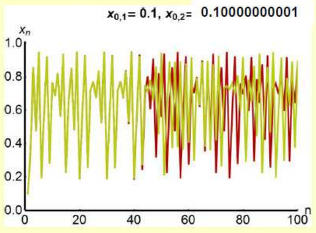
Above critical value of r: Transition to chaotic behavior.

Behavior of the Logistic Map



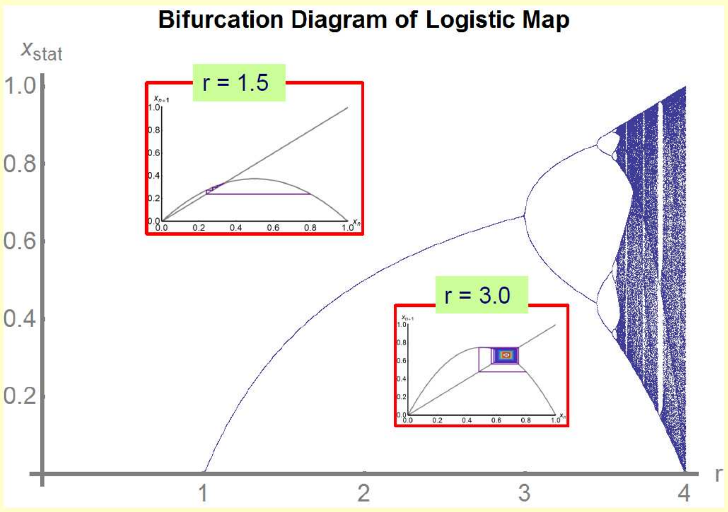




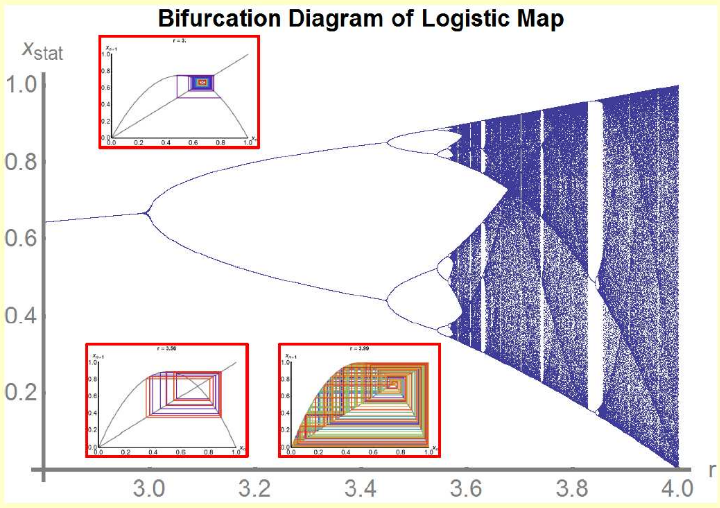


Initial conditions are very critical: Arbitrary small deviations eventually take large effect.

Logistic Map: Bifurcation Diagram



Logistic Map: Bifurcation Diagram



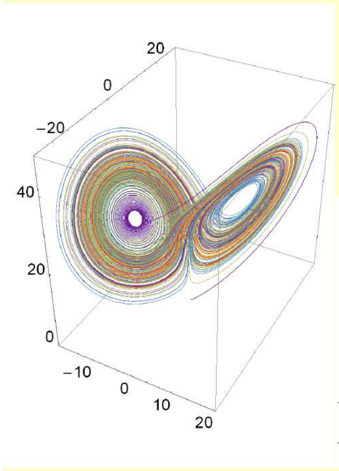
Case Study: Lorenz Attractor

- E. Lorenz aspired a model for some meteorological phenomena (convection).
- He observed critical dependence on initial condititions.
- First „practical“ observation of weak determinism.

σ: Prandtl number
ρ: Rayleigh number

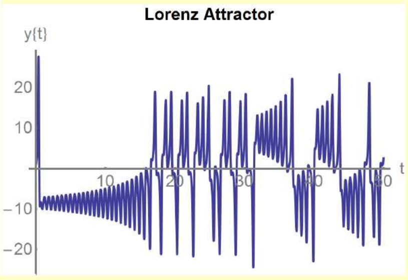
$$\dot{x} = \sigma(y - x)$$
$$\dot{y} = x(\rho - z) - y$$
$$\dot{z} = xy - \beta z$$

Lorenz Attractor

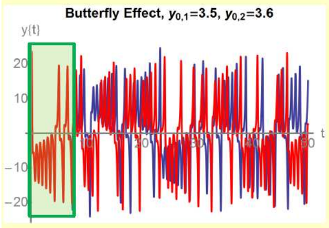


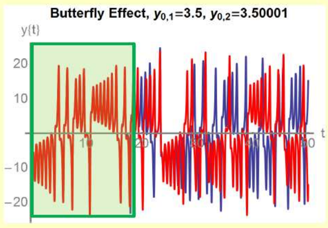
σ: 10, **β:** 8/3, **ρ:** 28
Fractal dimension: 2.06

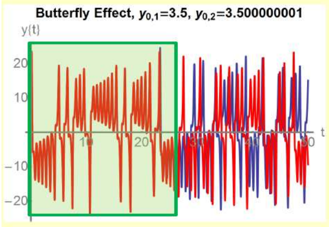
$$\dot{x} = \sigma(y - x)$$
$$\dot{y} = x(\rho - z) - y$$
$$\dot{z} = xy - \beta z$$



Lorenz Attractor: Butterfly Effect







Tiny variation have large effects: Orders of magnitude in precision lead only to linear increase of correctly predicted time!
The Lorenz system is strictly deterministic, but in various senses unpredictable.

Determinism and Non-Determinism

- Abstractly, the trajectories of the Lorenz – system are given in a unique way for each possible set of initial conditions ➔ the system is deterministic.
- The system may be chaotic: Given that we can't measure initial conditions to any degree of precision and given that numerical computations are subject to random errors, we can't calculate the long term behavior of the system. **But this is a problem of our computation, not of the system itself.** There is no sudden emergence of non-determinism.

Dynamical Systems – State of the Art

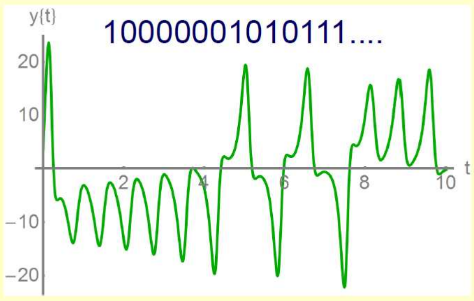
	Number of variables				
	n = 1 Growth, decay, equilibrium	n = 2 Oscillations	n = 3 Chaos	n >> 1 Collective phenomena	Continuum Waves and patterns
Linear	Populations RC – circuits Radioactive decay	Mass and spring RCL- circuit 2-body problem	Civil and electrical engineering	Solid state physics Molecular dynamics	Wave equation Elasticity Hydrodynamics
Non-linear	Fixed points Bifurcations Overdamped systems Logistic map	Pendulum Anharmonic oscillator Limit cycles Biological osc. Predator-prey Non-linear electronics	Strange attractors 3-body problem Chemical kinetics Fractals Iterated maps Practical uses of chaos	Non-equilibrium statistical mech. Lasers Heart cells Neural networks Immune system Eco-systems	QFT Earthquakes Reaction-diffusion systems. Fibrillation Epilepsy Turbulent flow Climate

Relation to other IT Systems

	Fixed points	Limit cycles, Closed orbits, regular behavior	Chaos, strange attractors, complex behavior
Systems of differential equations			
Dim 1	Yes	No	No
Dim 2	Yes	Yes	No
Dim ≥ 3	Yes	Yes	Yes
Cellular Automata			
Class 1	Yes	No	No
Class 2	No	Yes	No
Class 3	No	No	Chaos
Class 4	Yes (?)	Yes (?)	Complex beh.
Automata			
FSA	Yes	Yes	No
TM	Yes (Halting)	Yes	Yes

Dynamical Systems and Information

Dynamical systems can switch between different states.
A dynamical system together with an initial condition can be understood as a (maybe compressed) representation of a sequence.



Chapter 8: Fractals

- *Fractals* refer to structures displaying self-similarity on different scales.

Measuring the Length of Coastlines

=> the border of a country does not necessarily have a “true” length, but that the measured length of a border depends on the unit of measurement.
Richardson demonstrated that the measured length of coastlines and other natural features appears to increase without limit as the unit of measurement is made smaller. This is known as the *Richardson effect*.

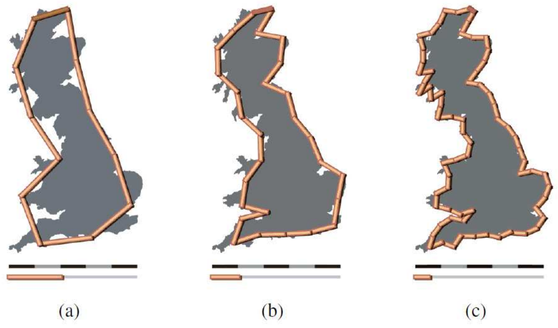


Figure 8.2: Measuring the length of the coast of Britain with different units of measurement. Notice that the smaller the ruler, the bigger the result.

Fractional Dimension

=> how to measure "normal objects"

one-dimensional object:

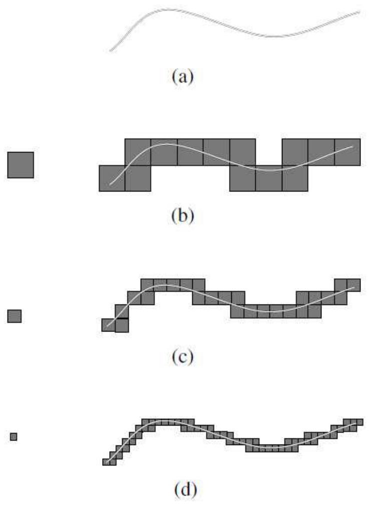


Figure 8.4: Measuring the size of a one-dimensional object, the line shown in (a), using measuring units of different sizes. Reducing the size of the measuring unit by a (linear) factor of 2 requires approximately twice as many units to cover the line.

If a represents the size of the measuring unit (i.e. the side length of the square) and N represents the number of measuring units needed to cover the line, we have the relation:

$$N \approx c_1 \cdot \frac{1}{a} \tag{8.1}$$

with a particular constant c_1.

The plot of this relation in log-log scales in shown in Figure 8.5(a). In particular, the measured length L of the line, defined as $L = N \cdot a$ converges to a fixed value as a gets smaller, since we have:

$$L = N \cdot a \approx \left(c_1 \cdot \frac{1}{a}\right) \cdot a = c_1$$

two-dimensional object:

Let us now measure the size of a two-dimensional object, the surface shown in the top left corner of Figure 8.6, using again the same square measuring unit. This time, reducing the size of the measuring unit by a linear factor of 2 requires approximately four times as many units to cover the surface. We thus have now the relation:

$$N \approx c_2 \cdot \left(\frac{1}{a}\right)^2 \tag{8.2}$$

with a particular constant c_2.

The plot of this relation in log-log scales in shown in Figure 8.7(a). The measured area A of the surface, defined as $A = N \cdot a^2$ converges again to a fixed value as a gets smaller:

$$A = N \cdot a^2 \approx \left[c_2 \cdot \left(\frac{1}{a}\right)^2\right] \cdot a^2 = c_2$$

D-dimensional object:

Equations 8.1 and 8.2 can be summarized, in the general case of a D-dimensional object, as:

$$N \approx c \cdot \left(\frac{1}{a}\right)^D \tag{8.3}$$

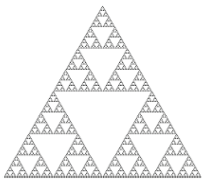
Applying the $\log$ to both sides and taking the limit where a get infinitesimally small, we can rewrite this equation as:

$$D = \lim_{a \to 0} \frac{\log(N)}{\log(\frac{1}{a})} \tag{8.4}$$

Another way of looking at this equation is that the slope of N as function of a in a log-log plot gives us $-D$. This relation is illustrated in Figure 8.8, which combines together the two plots of Figures 8.5(a) and 8.7(a).

fractal:

Let us consider a fractal figure that we've already encountered in the previous chapter – the Sierpinsky gasket:



Let us now measure its surface, using again squares as measuring units. Figure 8.9 shows how the number of measuring units required to cover the fractal pattern increases as the size of the measuring unit decreases.

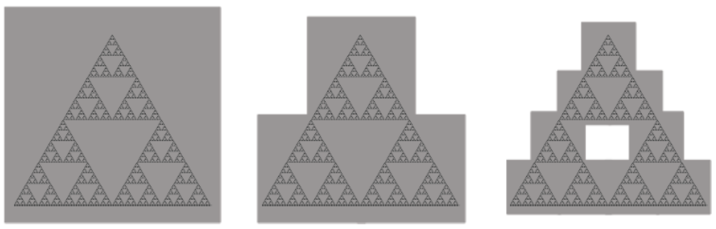


Figure 8.9: Number of measuring units needed to cover objects as function of the size of the measuring unit. The slope (with the reverse sign) gives us the dimension of the object being measured.

Let us consider a serie of measuring units whose size are each time divided by two: $a_n = \left(\frac{1}{2}\right)^n$. From Figure 8.9, we see that the number of measuring units needed to cover the pattern will each time increase by three: $N = 3^n$.

The dimension of this pattern is thus:

$$\begin{aligned} D &= \lim_{a\to 0} \frac{\log(N)}{\log(\frac{1}{a})} = \lim_{n\to\infty} \frac{\log(3^n)}{\log(2^n)} \\ &= \lim_{n\to\infty} \frac{n\log(3)}{n\log(2)} = \lim_{n\to\infty} \frac{\log(3)}{\log(2)} \\ &= \frac{\log(3)}{\log(2)} \approx 1.585 \end{aligned}$$

The dimension is not an integer anymore, but is *fractional*. In a sense, the Sierpinsky gasket is neither one-dimensional (as a line) nor two-dimensional (as a surface), but somewhere in between.

Note also that the actual measures (such as length or surface) of patterns with fractional dimenions do not converge to fixed (positive) value anymore. For instance, the Sierpinsky gasket has in fact no area, since:

$$\begin{aligned} A &= N \cdot a^2 \\ &= \lim_{n\to\infty} N_n \cdot a_n^2 \\ &= \lim_{n\to\infty} 3^n \cdot \left(\frac{1}{2^n}\right)^2 \\ &= \lim_{n\to\infty} \left(\frac{3}{4}\right)^n = 0 \end{aligned}$$

Furthermore, the measured length of the border of the Sierpinsky gasket diverges to infinity – similar to the lengths of the coastlines measured by Richardson!

$$\begin{aligned} L &= N \cdot a \\ &= \lim_{n\to\infty} N_n \cdot a_n \\ &= \lim_{n\to\infty} \left(\frac{3}{2}\right)^n \longrightarrow \infty \end{aligned}$$

Examples of Fractals

the Cantor Set

The so-called *Cantor set* is one of the simplest fractals. It can be constructed as follows (see Figure 8.10):

1. Start with the interval $[0, 1]$ and color it black.
2. Take out the middle third of it, yet leaving the endpoints. You end up with two intervals: $[0, \frac{1}{3}] \cup [\frac{2}{3}, 1]$.
3. From each remaining interval take out the (open) middle third and repeat this procedure for ever.

This pattern has a number of remarkable and deep properties.

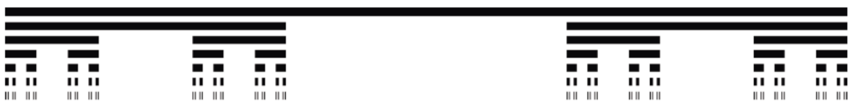


Figure 8.10: The first 7 iterations of the Cantor set.

properties:

What is the length of the Cantor set? After n steps, we have 2^n segments, each of which has a length of $\frac{1}{3^n}$. The length of the Cantor set is thus zero:

$$L = \lim_{n \to \infty} \frac{2^n}{3^n} = 0$$

Even though the Cantor set has a length of zero, it still contains some points. The "endpoints" of each interval are never removed, so the Cantor set contains an infinite number of points.

It may appear that only the endpoints are left, but that is not the case either. The number $\frac{1}{4}$, for example, is in the bottom third, so it is not removed at the first step. It is in the top third of the bottom third, and is in the bottom third of that, and in the top third of that, and in the bottom third of that, and so on ad infinitum – alternating between top third and bottom third. Since it is never in one of the middle thirds, it is never removed, and yet it is also not one of the endpoints of any middle third.

In fact, it can be shown that the Cantor set is *uncountable* – there are as many points in the Cantor set as there are in the original interval [0, 1]!

the Koch Curve

The Koch curve is one of the earliest fractal curves to have been described. Each stage is constructed in 3 steps by starting with a line segment and then recursively altering each line segment as follows (see Figure 8.11):

1. Divide the line segment into three segments of equal length.
2. Draw an equilateral triangle that has the middle segment from step one as its base.
3. Remove the line segment that is the base of the triangle from step 2.

The Koch snowflake, shown in Figure 8.12, is the same as the Koch curve, except that it starts with an equilateral triangle.

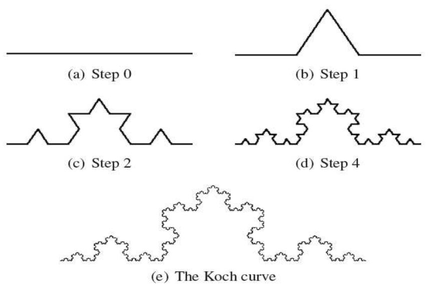


Figure 8.11: Recursive construction of the Koch curve.

L-Systems (Lindenmayer Systems)

=> essentially a formal grammar
With the advent of informatics, L-systems have not only become popular to model the growth processes of plant development, but also to graphically generate the morphology of complex organisms.

Figure 8.15: Fractal trees and plants created using a Lindenmayer system.

L-systems consist essentially of rewrite rules that are applied iteratively to some initial string of symbols. The *recursive* nature of the L-system rules leads to *self-similarity* and thereby to fractal-like forms, which are easy to describe with an L-system.
Note that the rules of the L-system grammar are applied iteratively starting from the initial state. During each iteration, as many rules as possible are applied *simultaneously*. This is the distinguishing feature between an L-system and the formal language generated by a grammar.

Example 8.1. The following grammar was Lindenmayer's original L-system for modelling the growth of algae:

- **Variables:** A, B
- **Constants:** (none)
- **Axiom:** A
- **Production rules:**

$$\begin{aligned} A &\rightarrow AB \\ B &\rightarrow A \end{aligned}$$

This produces:

Step 0: A

Step 1: AB

Step 2: ABA

Step 3: $ABAAB$

Step 4: $ABAABABA$

Step 5: $ABAABABAABAAB$

Step 6: $ABAABABAABAABABAABABA$

Step 7: $ABAABABAABAABABAABABAABAABABAABAAB$

Turtle Graphics

=> interprets L-Systems and draws it

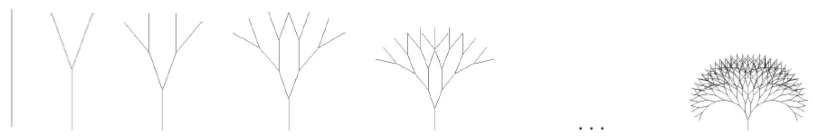
Command	Turtle Action
F	draw forward (for a fixed length)
|	draw forward (for a length computed from the execution depth)
+	turn right (by a fixed angle)
−	turn left (by a fixed angle)
[	save the turtle's current position and angle for later use
]	restore the turtle's position and angle saved during the corresponding] command

Table 8.1: Turtle graphics commands.

Example 8.2. The following L-system:

- **Axiom:** F
- **Production rule:** F $\rightarrow$ F [−F][+F]
- **Angle:** 20

produces the following stages of the draw process:



Development Models

=> "extended Turtle-Graphics" on 3 Dimensions, additional information can be included into the production rules, including delay mechanisms, influence of environmental factors or stochastic elements – so that not all the plants look the same.

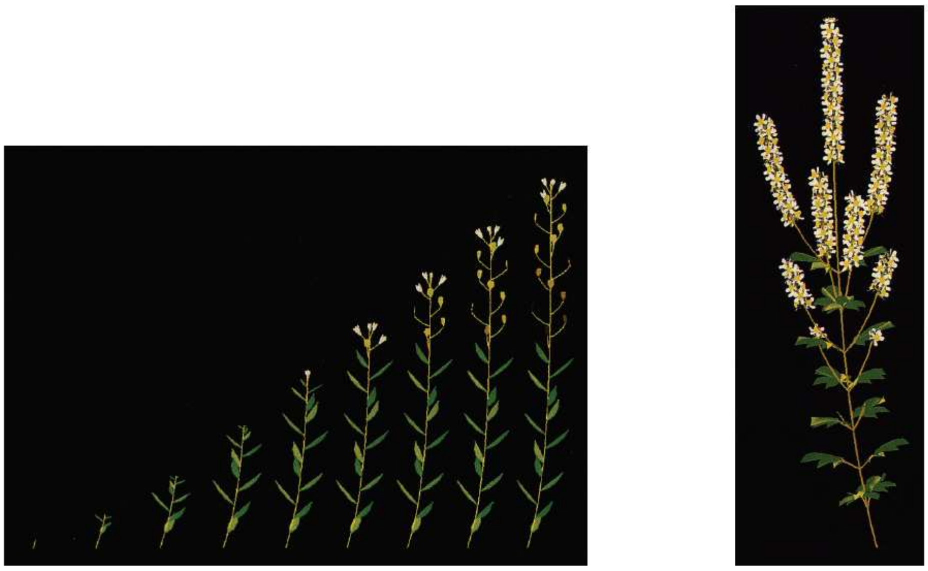


Figure 8.16: Simulated development of plants grown with more complex L-systems.

The Mandelbrot Set

In this section, wee will see once again how fractals are not necessarily related to recursive property of the system, but rather surprisingly closely related to decidability and chaos.

“Will it diverge?”

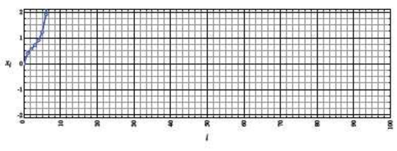
Let us consider the sequence of numbers defined by the following equation:

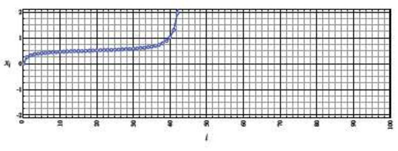
$$x_{i+1} = x_i^2 + c \tag{8.5}$$

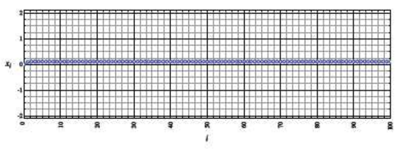
with $x_0 = 0$ and a given constant c.

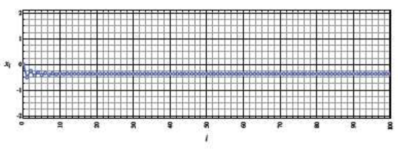
The question is: for what c will the serie diverge to infinity?

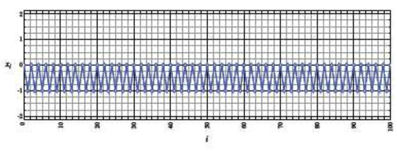
with real numbers:

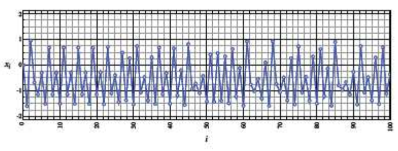

(a) $c = 0.4$

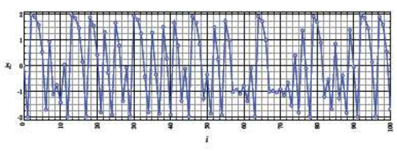

(b) $c = 0.255$


(c) $c = 0.1$


(d) $c = -0.5$


(e) $c = -1.0$


(f) $c = -1.6$


(g) $c = -1.99$

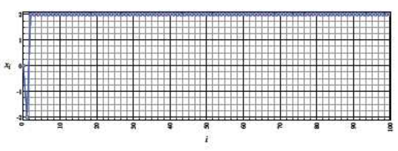

(h) $c = -2.0$

Figure 8.17: Evolution of the serie defined by Eq. 8.5 for different values of c.

with complex numbers:

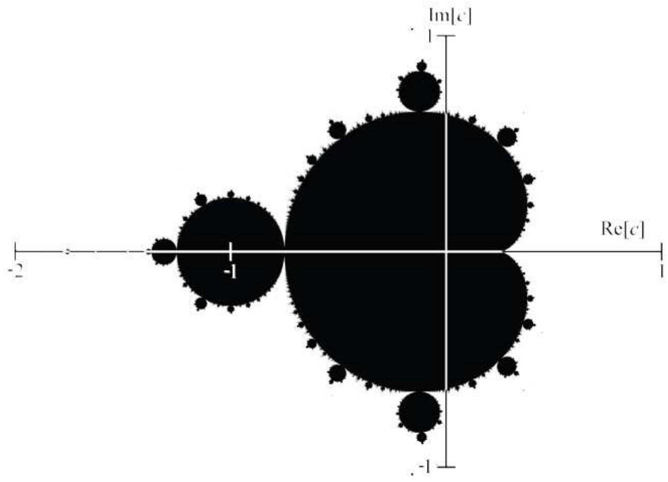


Figure 8.18: The Mandelbrot set.

Chapter 9: Graphs and Networks