Formal Foundations of Computer Science

Introduction to Formal Foundations of Computer Science

Number Systems

Number systems

- Positional notation or place-value notation systems (Stellenwertsysteme)
  - decimal system
    - „our natural number system“
    - base 10, digits 0..9
    - e.g.: $305 = 3 * 10^2 + 0 * 10^1 + 5 * 10^0$
  - binary/dual/base-2 system
    - base 2, digits 0 and 1
  - octal system
    - base 8, digits 0..7
  - hexadecimal (sededicmal) system
    - base 16
    - digits 0..9, A, B, C, D, E, F

Number Systems Overview

decimal	dual	octal	hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Number Systems Conversions

- decimal → dual
  - iterated division by 2
  - the number is divided by two, and the remainder is the least-significant bit
  - The (integer) result is again divided by two, its remainder is the next most significant bit
  - This process repeats until the result of further division becomes zero
  - e.g.: $Z = 43_{(10)}$

```
43 DIV 2 =    21      Remainder 1
21 DIV 2 =    10      Remainder 1
10 DIV 2 =    5       Remainder 0
5  DIV 2 =    2       Remainder 1
2  DIV 2 =    1       Remainder 0
1  DIV 2 =    0       Remainder 1
```

read bottom-up

$43_{(10)} = 101011_{(2)}$

Number Systems Conversions

- dual → decimal
  - reverse process, iterated multiplication
  - or simply add powers of 2 from all digits that are set to 1
  - e.g.:

dual	1 0 1 0 1 1
powers of 2	2^5 2^4 2^3 2^2 2^1 2^0
decimal	$2^5+2^3+2^1+2^0 = 32+8+2+1 = 43$

- in general: base b, m+1 positions

  decimal number: $(u_m..u_1u_0)_b = u_m*b^m+..+u_1*b^1+u_0*b^0$

  e.g.: $1A5_{16} = 1*16^2+10*16^1+5*16^0=256+160+5=421$

Number Systems
Conversion of Rational Numbers

- exact conversion from decimal to binary not always possible

Example: $0.11001_2 = ?_{10}$

$$
\begin{aligned}
0.11001 &= 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} \\
&= 1 \cdot 0.5 + 1 \cdot 0.25 + 0 \cdot 0.125 + 0 \cdot 0.0625 + 1 \cdot 0.03125 \\
&= 0.5 + 0.25 + 0.03125 \\
&= 0.78125_{10}
\end{aligned}
$$

Number Systems
Conversion of Rational Numbers

Example: $0.19_{10} = ?_2$ with $k = 9$ bit precision

Step i	N	Operation	R	$z_{(-i)}$
1	0.19	$0.19 \cdot 2 = 0.38$	0.38	0
2	0.38	$0.38 \cdot 2 = 0.76$	0.76	0
3	0.76	$0.76 \cdot 2 = 1.52$	0.52	1
4	0.52	$0.52 \cdot 2 = 1.04$	0.04	1
5	0.04	$0.04 \cdot 2 = 0.08$	0.08	0
6	0.08	$0.08 \cdot 2 = 0.16$	0.16	0
7	0.16	$0.16 \cdot 2 = 0.32$	0.32	0
8	0.32	$0.32 \cdot 2 = 0.64$	0.64	0
9	0.64	$0.64 \cdot 2 = 1.28$	0.28	1

$\Rightarrow 0.19_{10} = 0.001100001_2 + \varepsilon$

Multiplication!

Number Systems Conversions

- dual ↔ hexadecimal
  - divide number into groups of 4 digits (add leading zeroes)
  - replace every 4 binary digits by 1 hexadecimal digit
  - e.g.: 101011 = 0010 1011 = $2B_{16}$
  - reverse process: replace every hexadecimal digit by 4 binary digits

- dual ↔ octal
  - similar process with groups of 3 digits
  - replace every 3 binary digits by 1 octal digit
  - e.g.: 101011 = 101 011 = 53_{8}
  - reverse process: replace every octal digit by 3 binary digits

Number Systems Binary Arithmetic

	Operation	Result	Carry
Addition	$0 + 0$	0	0
	$0 + 1$	1	0
	$1 + 0$	1	0
	$1 + 1$	0	1
Subtraction	$0 - 0$	0	0
	$0 - 1$	1	1
	$1 - 0$	1	0
	$1 - 1$	0	0
Multiplication	$0 \cdot 0$	0	0
	$0 \cdot 1$	0	0
	$1 \cdot 0$	0	0
	$1 \cdot 1$	1	0

Number Systems - Binary Addition

- same rules as in decimal system
- even easier since fewer possibilities ☺
- but 1+1 is not 2 but 0 (and 1 carry)

$$
\begin{array}{r}
101010 \\
+1101111 \\
\hline
10011001
\end{array}
$$

Negative Numbers
Integer Representation

- one posssible solution:
  - coding of sign in 1st bit
  - 0 = „+“ positive number
  - 1 = „-“ negative number
  - e.g.: 4 bits = range from -7 to +7

0000 = +0	1000 = -0
0001 = +1	1001 = -1
0010 = +2	1010 = -2
0011 = +3	1011 = -3
0100 = +4	1100 = -4
0101 = +5	1101 = -5
0110 = +6	1110 = -6
0111 = +7	1111 = -7

Negative Numbers
Integer Representation

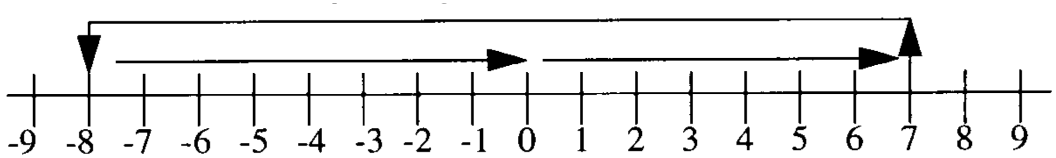
- Drawbacks of this solution:
  - 2 possibilities of coding zero +0,-0?
  - problems with binary arithmetic, no simple addition and substraction possible

-3	1011
+ +5	+ 0101
+2	????

Negative Numbers
2-Complement (B-Complement)

- commonly used representation of integers
  - positive numbers as usual
  - special rule for negative numbers

- e.g. 4 bit coding:
  - $2^4 = 16$ numbers can be coded
  - binary mapping of decimal numbers –8 to +7
  - s



1000 = -8	1100 = -4	0000 = 0	0100 = +4
1001 = -7	1101 = -3	0001 = +1	0101 = +5
1010 = -6	1110 = -2	0010 = +2	0110 = +6
1011 = -5	1111 = -1	0011 = +3	0111 = +7

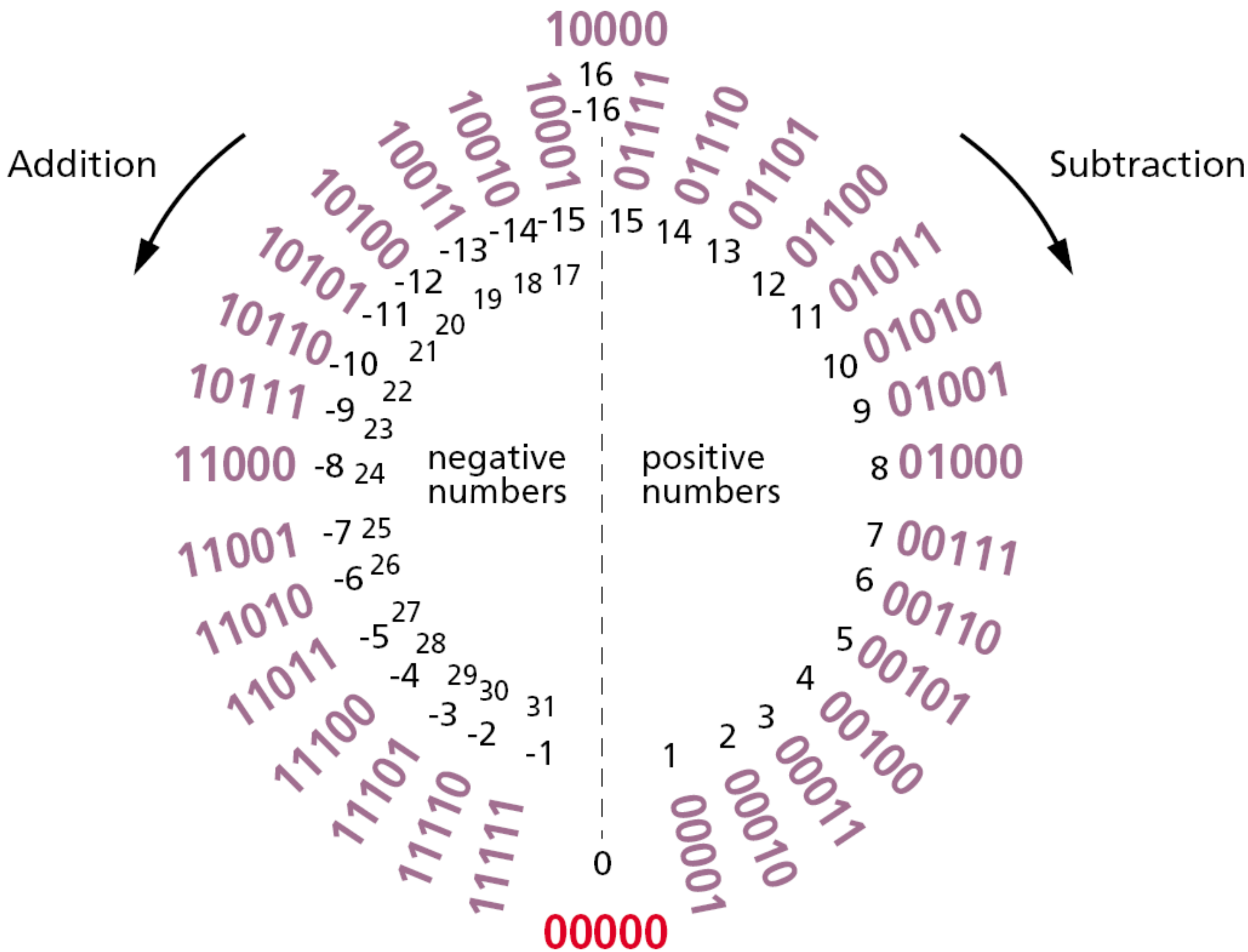
Negative Numbers 2-Complement

- Advantages:
  - first bit is sign again
  - only one representation of zero
  - binary arithmetic as usual

- general rule:
  - with N bits it is possible to represent the number range from -2^{N-1} to $+2^{N-1}-1$
  - the bitstring $b_n b_{n-1} \ldots b_1 b_0$ represents the decimal number z, where
- $z = b_n * (-2^n) + b_{n-1} * 2^{n-1} + \ldots + b_1 * 2^1 + b_0$

2-Complement Number Ring

Number ring for 5-digit 2-complement



2 Complement Calculation

- Negative numbers are created by:
  - bitwise negation (complement) of the positive value (swap every 1 with zero and vice versa)
  - add 1 to avoid the negative zero
- e.g.: -5 in 4 bit representation

value (+5)	0101
negation	1010
add 1	+ 0001
2 complement	1011

2 Complement Substraction

- substraction can be mapped to addition with the 2-complement
- e.g.: 8 bit coding
  - carry is ignored

$$\begin{array}{r} 12 \\ - \quad 4 \\ \hline 8 \end{array}$$

4	00000100
neg	11111011
add 1	00000001
-4	11111100

$$\begin{array}{r} 00001100 \\ +\ 11111100 \\ \hline 00001000 \end{array}$$

Floating Point Numbers

- how to code real numbers like ?
  - e.g. 4.53, 0.5665
- how to code large numbers (10^{14}) or very small numbers (10^{-29}) or numbers like ($103.4*10^{17}$, $1.45*10^{-29}$) ?

- Not all infinite real numbers can be represented!
  - limited precision
  - rounding errors

- Real numbers in C
  - Single Precision (32 bit) = float
  - Double Precision (64 bit) = double

Floating Point Numbers Binary Representation

- sign bit: S
  - positive 0
  - negative 1

- exponent: E
  - The exponent represents a value raised to the power of 2

- mantissa: M
  - The mantissa represents a fractional value between 0 and 1
  - $m_1 \ldots m_n$
  - Interpretation: $m_1 * 2^{-1} + m_2 * 2^{-2} + \ldots + m_n * 2^{-n}$

Floating Point Numbers
Normalized Numbers

- normalized floating point number
  - $\pm\ 1.m_1m_2...m_n\ *\ 2^E$
  - most significant bit (1) is not stored
  - optimal storage of mantissa bits

- every floating point number can be converted to a normalized representation
  - just shift the mantissa, and update (±1) the exponent

Floating Point Numbers Examples

decimal	dual
0.5	0.1
5.75	101.11
0.25	0.01
0.1	0.00011001100110011....

- conversion example: 1 bit sign, 3 bit E, 8 bit M
  - wanted: representation of 5.75
  - binary conversion = 101.11
  - normalization: $1.0111 * 2^2$
  - sign bit: 0
  - exponent: 2 = 010
  - mantissa: 01110000

Floating Point Numbers Standards

- IEEE 754 Standard for Binary Floating-Point Arithmetic
  - Institute of Electrical and Electronics Engineers

	S	e	M
32 bit	1	8	23
64 bit	1	11	52

  - no sign bit for exponent
  - shifting (addition of a bias) is used instead
    - exponent E = e - bias
    - 32bit: bias 127, E from -127 to 128
    - 64bit: bias 1023, E from -1023 to 1024
  - this leads to easier comparison of the exponents

Floating Point Numbers
Limited Precision

- Not all values can be represented

Example:

- mantissa: 2 decimal digits
- exponent: 1 decimal digit

- Sample number: $74 \cdot 10^2 = 7400$

What is the next higher value?

$75 \cdot 10^2 = 7500$

What about values $7400 < x < 7500$?

$\Rightarrow$ **They cannot be represented!!!**

Floating Point Numbers
Limited Accuracy

- Floating point arithmetic is not associative and distributive

- e.g 7-digit decimal arithmetic:
  - $1234.567 + 45.67844 = 1280.245$
  - $1280.245 + 0.0004 = 1280.245$
  - but $45.67844 + 0.0004 = 45.67884$
  - $45.67884 + 1234.567 = 1280.246$

  - $1234.567 \times 3.333333 = 4115.223$
  - $1.234567 \times 3.333333 = 4.115223$
  - $4115.223 + 4.115223 = 4119.338$
  - but $1234.567 + 1.234567 = 1235.802$
  - $1235.802 \times 3.333333 = 4119.340$

Floating Point Numbers
Limited Accuracy

- Cancellation (Auslöschung)
  - subtraction of nearly equal operands may cause extreme loss of accuracy

- Truncation/rounding problems
  - e.g. converting (63.0/9.0) to integer yields 7, but converting (0.63/0.09) may yield 6

- Limited exponent range
  - results might overflow yielding infinity, or underflow yielding a denormal value or zero

- Testing for safe division is problematical
  - Checking that the divisor is not zero does not guarantee that a division will not overflow and yield infinity

- Equality is problematical!
  - instead of if (result == expectedResult) use
    if (fabs(result - expectedResult) < 0.00001)

Use double instead of float for accuracy!!!

Final Words

- Humor of computer scientists:

  "There are only 10 types of people in the world: Those who understand binary and those who don't."

Information Theory

Information theory
Introduction

- founded by Shannon in 1948 in his paper "A Mathematical Theory of Communication"

- goal: quantification of information to find fundamental limits on compressing and reliably communicating data

- key measure is information entropy or Shannon entropy (Maß für Informationsgehalt)
  - average number of bits needed for storage or communication
  - entropy quantifies the uncertainty involved in a random variable
  - e.g. a fair coin flip will have less entropy than a roll of a die

Information Theory
Shannon Entropy

- Shannon information content of character x measured in bits
  - h = ld(1/p) = -ld p (p…probability of x, ld=logarithmus dualis)

  - ASCII characters, if chosen uniformly at random, have an entropy of exactly 7 bits per character
  - but some characters are chosen more frequently in English, then our uncertainty is lower
  - therefore the Shannon entropy is lower
  - A long string of repeating characters has an entropy of 0 (predictable)
  - The entropy of English text is between 1.0 and 1.5 bits per letter
  - The entropy of a n-digit decimal number?
    - p=1/10
    - h=n*ld 10

Information Theory
Shannon Entropy

- Entropy of a discrete random variable X with alphabet $Z=\{z_1,z_2,z_3,\ldots\}$ and $P(X \in Z)=1$
  - is the weighted sum, across all symbols with non-zero probability of the information content of each symbol

$$H(X) = -\sum_{i=1}^{|Z|} p_i \cdot \log_2(p_i)$$

where $p_i = p(z_i) = P(X = z_i)$

Information Theory
Shannon Entropy

- Example
  - Alphabet Z={x,y,z}
  - probabilities: p(x)=0.5, p(y)=0.25, p(z)=0.25

  - therefore h(x)=ld 2=1, h(y)=ld 4=2, h(z)=2
  - and finally H(X)=0.5*1+0.25*2+0.25*2 = 1.5 bit

  - Consequences for coding?

    - optimal code for this example: variable-length binary code where x=1, y=01, z=00
    - e.g. yxxzyx=011100011

Information Theory
Huffman Coding

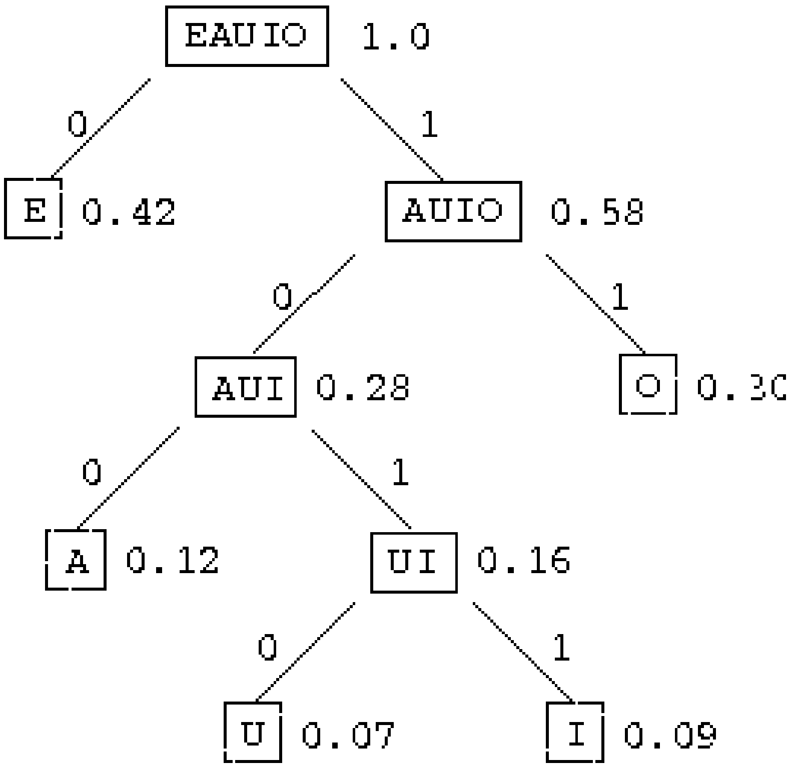
- entropy encoding algorithm used for lossless data compression
- Example
  - encode the letters *A (0.12), E (0.42), I (0.09), O (0.30), U (0.07)*, listed with their respective probabilities

- Go through the following steps:

1. Consider each of the letters as a symbol with its respective probability.
2. Find the two symbols with the smallest probability (or frequency count) and combine them into a new symbol with both letters by adding the probabilities.
3. Repeat step 2 until there is only one symbol left with a probability of 1.
4. To see the code, redraw all the symbols in the form of a tree, where each symbol contains either a single letter or splits up into two smaller symbols. Label all the left branches of the tree with a 0 and all the right branches with a 1. The code for each of the letters is the sequence of 0's and 1's that lead to it on the tree, starting from the symbol with a probability of 1.

Information Theory
Huffman Coding

- generated huffman tree



- resulting codes:
  - A - 100
  - E - 0
  - I - 1011
  - O - 11
  - U - 1010

Introduction to Mathematical Logic

Propositional Logic Basics

- Boolean algebra (logic)
  - algebra of only 2 values:
    - TRUE (T) = 1
    - FALSE (F) = 0
    - represented by 1 bit
  - Basic operations
    - NOT, AND, OR, XOR, …
  - Axioms
    - commutativity, associativity, distributivity, …

Propositional Logic Basics

- Representation with truth tables
  - input, operation, output
  - notation:
    - complement (negation): NOT, ¬, ‾
    - conjunction: AND, ∧, ., &
    - disjunction: OR, ∨, +
    - exclusive-or (parity): XOR, ⊕
    - Implication: ⊃, →
    - equivalence: ≡, ↔

¬	
F	T
T	F

∧	F	T
F	F	F
T	F	T

∨	F	T
F	F	T
T	T	T

⊕	F	T
F	F	T
T	T	F

⊃	F	T
F	T	T
T	F	T

≡	F	T
F	T	F
T	F	T

Propositional Logic Basics

a	b	¬a	$a \wedge b$	$a \vee b$	$a \oplus b$	$a \supset b$	$a \equiv b$
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	0	0
1	1	0	1	1	0	1	1

- Note: Some connectives can be simulated by others, which is why only a subset of them needs to be implemented in hardware!

- For instance, one of the following is enough:
  - OR and NOT
  - AND and NOT
  - NAND (= combined NOT and AND)

Boolean Algebra Laws

- Let B be a set with at least two elements 0 and 1. Let two binary operations $\vee$ and $\cdot$, and a unary operation $\overline{}$ are defined on B. The algebraic system
  $\langle B, \vee, \cdot, \overline{}, 0,1\rangle$ is a **Boolean algebra**, if the following postulates are satisfied:

1. Idempotent laws: $a \vee a = a$, $a \cdot a = a$;
2. Commutative laws: $a \vee b = b \vee a$, $a \cdot b = b \cdot a$
3. Associative laws: $a \vee (b \vee c) = (a \vee b) \vee c$,
   $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
4. Absorption laws: $a \vee (a \cdot b) = a$, $a \cdot (a \vee b) = a$
5. Distributive laws: $a \vee (b \cdot c) = (a \vee b) \cdot (a \vee c)$,
   $a \cdot (b \vee c) = (a \cdot b) \vee (a \cdot c)$

Note:
Strictly speaking, there is not just one Boolean algebra, but *any* algebraic structure satisfying these postulates is a Boolean algebra. But in practice one can usually think of it as “the” (single) Boolean algebra.

Boolean Algebra Laws

6. Involution: $\overline{\overline{a}} = a$

7. Complements: $a \vee \overline{a} = 1$, $a \cdot \overline{a} = 0$;

8. Identities: $a \vee 0 = a$, $a \cdot 1 = a$;
   $a \vee 1 = 1$, $a \cdot 0 = 0$;

9. De Morgan's laws:

$$\overline{a \vee b} = \overline{a} \cdot \overline{b}$$

$$\overline{a \cdot b} = \overline{a} \vee \overline{b}$$

Propositional Logic Formalization

Syntax:

- We start with a non-empty **domain** (=set of propositional variables) **A.**
- Each of these variables can either be true or false.

Examples:

- A_1 = { a, b }
- A_2 = { rainy, cloudy, sunny }

Propositional formulas over an alphabet A are inductively defined as follow:

- Each variable $v \in \mathbb{A}$ is a formula
- $\top$ and $\perp$ are formulas
- If f_1 and f_2 are formulas,
  then $\neg(f_1)$, $(f_1 \wedge f_2)$, $(f_1 \vee f_2)$, $(f_1 \oplus f_2)$, $(f_1 \supset f_2)$, $(f_1 \leftrightarrow f_2)$ are also formulas
- Formulas are *only* created by these rules

Examples:

- $f_1=(a \wedge b) \supset b$ is a formula over A_1, but „$(a \leftrightarrow b)\ c$" is not (missing connective)
- $f_2=rainy \equiv \neg sunny$ is a formula over A_2

Propositional Logic Formalization

Semantics:

- What is the *meaning* of a formula?
- It is either **true** or **false**!
- The truth value depends on the values of the propositional variables.

- Definition:
  An **interpretation** of a formula f over a domain A is a set $I \subseteq A$
  (the set of all propositions that are assumed to be true)

Examples:

- I_1={rainy} is an interpretation of above f_2
  (it rains, but it is not sunny and not cloudy)
- I_2={rainy,cloudy} is another interpretation of above f_2
  (it rains and is cloudy, but not sunny)
- I_3={rainy,sunny} is another interpretation of above f_2
  (it rains and is sunny, but not cloudy)
- From an interpretation (that defines the truth values of propositional variables), we want to get the truth value of the whole formula.

Propositional Logic Formalization

Semantics:

- Let A be an alphabet, f be a formula over A and $I \subseteq A$ be an interpretation of f.
- We say that "I satisfies f" or "I models f" (written: $I \vDash f$) to express that f is true
- under the truth values of propositional variables from I.
- Formally, the semantics is recursively defined:
  - $I \vDash a$ for an atom $a \in A$ if $a \in I$
  - $I \vDash \neg(f_1)$ if $I \vDash f_1$ does not hold
  - $I \vDash (f_1 \wedge f_2)$ if $I \vDash f_1$ and $I \vDash f_2$
  - $I \vDash (f_1 \vee f_2)$ if $I \vDash f_1$ or $I \vDash f_2$
  - $I \vDash (f_1 \oplus f_2)$ if either $I \vDash f_1$ or $I \vDash f_2$ but not both
  - $I \vDash (f_1 \supset f_2)$ if $I \nvDash f_1$ or $I \vDash f_2$
  - $I \vDash (f_1 \equiv f_2)$ if $I \vDash f_1$ if and only if $I \vDash f_2$

Propositional Logic Formalization

Examples:

Reconsider A_2 = { rainy, cloudy, sunny } and f_2=rainy ≡ ¬sunny is a formula over A_2

- For I_1={rainy} we have $I_1 \models f_2$
  (Under the assumption that it's rainy, but not cloudy or sunny, it is true that rainy and sunny have opposite truth values.)
- For I_2={rainy,cloudy} we have $I_2 \models f_2$
  (Under the assumption that it's rainy and cloudy but not sunny, it is true that rainy and sunny have opposite truth values.)
- For I_3={rainy,sunny} we have $I_3 \not\models f_2$
  (Under the assumption that it's rainy and sunny but not cloudy, it is **not** true that rainy and sunny have opposite truth values.)

An interpretation I that satisfies a formula f is called a **model of f**.

An interpretation that does not satisfy a formula f

is called a **countermodel** or **counterexample of f**.

Thus, I_1 and I_2 are models of f_2, while I_3 is a counterexample.