4a. Programming languages (continued)

Programming languages differ not only in their underlying paradigm, but also in many other aspects

- e.g., in the way how functions and operators are written

a function:

mathematically a mapping

f: a_1 a_2 a_3 a_4 $\longrightarrow$ r (= result)

in mathematics we write normally:

$f(a_1, a_2, a_3, a_4) = r$

Different numbers of arguments ("arity"):

$f()$	0-ary function (no argument)
$f(a_1)$	unary function
$f(a_1, a_2)$	binary function
$f(a_1, a_2, a_3)$	ternary function

....

in this notation, first comes the function symbol and then the list of arguments (enclosed in parentheses)

= "*prefix notation*"

Disadvantage: in case of nested functions, evaluation proceeds *from right to left* (contrary to usual direction of reading in everyday life)

g(*f*(x)) : apply first *f*, then *g*

An operation like "addition" can also be seen as the application of a function "+"

in this case we write usually $a_1 + a_2$

= "*infix notation*"

The function symbol stands *between* the two operands

- in prefix notation it would be $+(a_1, a_2)$

Disadvantages of infix notation:

- only possible for 2 arguments
- danger of ambiguities: a1 + a2 * a3
  must be resolved by priority rule

Both prefix and infix notation are used in many programming languages, e.g. C and Java

3rd possibility: "*postfix notation*"

used in the language *PostScript* for *all* functions (including addition, multiplication...)

- function and operator symbols stand *always behind* their operands
- if consequently applied, no parentheses necessary!

$a_1 f$ stands for $f(a_1)$
$a_1 a_2 f$ stands for $f(a_1, a_2)$ etc.

also $a_1 a_2$ add for $a_1 + a_2$

x*fg* : apply first *f*, then *g* (order of evaluation from left to right)

example: what is the result of the PostScript expression

```
3   5   7   add   1   sub   mul   ?

3    12           1   sub   mul

3            11                 mul

                33
```

In other languages, too, postfix notation is sometimes used
e.g. in Java and C: x++ means "increment x by 1"
"++" as an operator in postfix notation

5. Boolean algebra and propositional logic

Logic: The art of thinking, especially: of drawing correct conclusions

Formal Logic: The search for correct **forms** of conclusions

Aristotle, stoics, medieval scholars

Relevant for computers!

Propositional logic: Analyzing sentences up to whole constituent sentences. Further analysis in predicate logic.

What is a "proposition" ?

Propositions: Utterances which are **true or false**.

Examples:

The “Lausitzer Rundschau” is a newspaper.

It is raining.

$3 < 2$.

Counterexamples:

Is it raining?

Shut the door, please!

Truth values of propositions: *true*, abbreviated: *T* or *t*
false, abbr. *F* or *f*

Propositions can be *combined* using *junctors*

(operators controlling the truth of the combined proposition)

Examples:
The "Lausitzer Rundschau" is a newspaper **and** the Venus is a planet.
3 < 2 **or** 1+1=2 .

Symbolic notation for propositional junctors:

$\neg$ "not" (Negation; has only one operand)

$\wedge$ "and" (Conjunction) – reminder: And $\approx \Lambda$

$\vee$ "or" (Disjunction) – reminder: latin "vel"

$\rightarrow$ "implies" (Implication)

$\leftrightarrow$ "are equivalent" (Equivalence)

Convention for dropping braces: $\neg$ binds tighter than $\wedge$, $\wedge$ binds tighter than $\vee$, $\vee$ binds tighter than $\rightarrow$ and $\leftrightarrow$.

"Truth tables":

Truth table representation of semantics of propositional junctors:

ϕ	ψ	$\neg\phi$	$\phi \wedge \psi$	$\phi \vee \psi$	$\phi \rightarrow \psi$	$\phi \leftrightarrow \psi$
f	f	t	f	f	t	t
f	t	t	f	t	t	f
t	f	f	f	t	f	f
t	t	f	t	t	t	t

further junctors: NAND (not and) and NOR (not or)

Definition of NAND and NOR

$(\phi \text{NAND} \psi) \leftrightarrow \neg(\phi \wedge \psi)$

$(\phi \text{NOR} \psi) \leftrightarrow \neg(\phi \vee \psi)$

ϕ	ψ	NAND	NOR
f	f	t	t
f	t	t	f
t	f	t	f
t	t	f	f

Boolean functions

George Boole, Scotland (1815-1864, developed Boolean algebra)

Switching functions: Functions from $\{0,1\}^m \to \{0,1\}^n$

Boolean functions: Switching function with $n = 1$.

Theory of Boolean functions: Connected to propositional logic (by using the code $F \mapsto 0, T \mapsto 1$). Propositional junctors are used to describe Boolean functions.

An *arbitrary* switching function can be represented by a vector of n Boolean functions of the form $\{0,1\}^m \to \{0,1\}$.

How many Boolean functions are there with *n* arguments?

Nullary Boolean functions: can be identified with 0 and 1.

Unary Boolean functions: There are four. Inputs are 0 and 1. Results can be:

$(0 \mapsto 0, 1 \mapsto 0)$: The constant 0.
$(0 \mapsto 0, 1 \mapsto 1)$: Identity.
$(0 \mapsto 1, 1 \mapsto 0)$: Negation.
$(0 \mapsto 1, 1 \mapsto 1)$: The constant 1.

Binary Boolean functions: There are 16.

n-ary Boolean functions: There are 2^n inputs, and for each input there are 2 alternatives, thus: There are 2^{2^n} n-ary Boolean functions.

Boolean operators: $\neg, \vee, \wedge$

Binary Boolean functions:

All binary Boolean functions

Arguments (xy):	(00)	(01)	(10)	(11)	a Boolean expression for the function
	0	0	0	0	$x \wedge \neg x$
	0	0	0	1	$x \wedge y$
	0	0	1	0	$x \wedge \neg y$
	0	0	1	1	x
	0	1	0	0	$\neg x \wedge y$
	0	1	0	1	y
	0	1	1	0	$(\neg x \wedge y) \vee (x \wedge \neg y)$
	0	1	1	1	$x \vee y$
	1	0	0	0	$\neg(x \vee y)$
	1	0	0	1	$(\neg x \wedge \neg y) \vee (x \wedge y)$
	1	0	1	0	$\neg y$
	1	0	1	1	$\neg y \vee x$
	1	1	0	0	$\neg x$
	1	1	0	1	$\neg x \vee y$
	1	1	1	0	$\neg(x \wedge y)$
	1	1	1	1	$x \vee \neg x$

Each n-ary Boolean function can be built from binary Boolean functions.

Applications of Boolean functions:

- design of switching circuits from simple elements (e.g., NAND- or NOR-gates)
- combination of conditions which must be fulfilled in order to execute some parts of programmes (most programming languages provide Boolean data type and functions)
- part of network models, e.g., in molecular genetics
- Proofs of equivalence of logical expressions
- they are part of logic-based programming languages like PROLOG
- applications in "knowledge engineering"

How can knowledge be represented in computers?

- not only by simple listing of numbers
- not by text only (text must be read and interpreted by humans, the computer cannot understand its meaning)

we need a representation of "knowledge items" (statements, facts...) which can be processed by the computer, i.e., sensefully transformed in a purely formal way (without interpretation by human beings)

A first step:
Propositional formulas

= basically: propositions where variables are allowed

We build propositional formulas recursively from:

- variables
- constants (*T* and *F*)
- junctors
- auxiliary symbols (parentheses)

Express propositions in a strictly defined language

Recursive definition of propositional formulas (p.f.):

1) Propositional variables p_i ($i \in \mathbb{N}_0$) are p.f.

2a), 2b) T (true) and F (false) are p.f.

3) For a p.f. ϕ, the negation $(\neg\phi)$ is a p.f.

4a), 4b), 4c), 4d) For two p.f. ϕ and ψ, the conjunction $(\phi \wedge \psi)$, the disjunction $(\phi \vee \psi)$, the implication $(\phi \rightarrow \psi)$ and the equivalence $(\phi \leftrightarrow \psi)$ are p.f.

Propositional junctors: $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$

This defines a **syntax**, i.e. a set of correctly formed finite sequences over the set of symbols.

Examples for correctly formed sentences: p_0, T, F, $\neg p_3$, $p_3 \wedge p_{17}$

What are their meanings, i.e. their **semantics**?

Assumption:
We can "know" if a proposition (without variables!) is true in a given situation.
(This means, we use only such "secure" propositions in our knowledge base – or we restrict the situations accordingly.)

Model relation:

$$A \models \phi$$

means: Proposition ϕ is true in situation A.

Semantic implication:

$$\phi_1 \dots \phi_n \models \phi$$

means: In all situations in which $\phi_1 \dots \phi_n$ are true, also ϕ is true.

Example: $p_1, p_2 \models p_1 \wedge p_2$

A p.f. is interpreted as either true or false. But how does its structure determine its meaning, i.e. its **semantics**?

The propositional variables p_i represent **concrete propositions**, which can be true or false in any given situation. Example: p_0 can mean: “It is raining.”

In a given situation, each p_i is assigned a truth value. We represent this by a function $\rho : \{p_i | i \in \mathbb{N}_0\} \to \{\text{true}, \text{false}\}$, **assigning a truth value to each propositional variable**.

The meaning of a formula ϕ depends **only** on the interpretation of the propositional variables ρ; thus, **situation A (in $A \models \phi$) can be represented by ρ alone.**

Meaning of ϕ in a situation characterized by ρ is written as $[\![\phi]\!]_\rho$.

We define the meaning of a p.f. inductively; the junctors represent abstractions of their equivalents in natural language:

1) $[\![p_i]\!]_\rho$ equals $\rho(p_i)$

2a) $[\![T]\!]\rho$ equals true, 2b) $[\![F]\!]\rho$ equals false, i.e. both meanings are independent of ρ.

3) $[\![\neg\phi]\!]_\rho$ equals the negation of $[\![\phi]\!]_\rho$.

4a) $[\![\phi \wedge \psi]\!]_\rho$ is true if and only if (iff) both $[\![\phi]\!]_\rho$ and $[\![\psi]\!]_\rho$ are true.

4b) $[\![\phi \vee \psi]\!]_\rho$ is true iff at least one of$[\![\phi]\!]_\rho$ and $[\![\psi]\!]_\rho$ is true.

4c) $[\![\phi \rightarrow \psi]\!]_\rho$ is true iff $[\![\phi]\!]_\rho$ is false or $[\![\psi]\!]_\rho$ is true.

4d) $[\![\phi \leftrightarrow \psi]\!]_\rho$ is true iff $[\![\phi]\!]_\rho$ and $[\![\psi]\!]_\rho$ have the same value.

3. and 4. are the inductive parts of the definition in which we show how the meaning of a complicated expression depends on the meanings of its parts.

Example: Computing the semantics of $((p_0 \rightarrow p_1) \rightarrow p_0) \rightarrow p_0$:

p_0	p_1	$H = p_0 \rightarrow p_1$	$G = H \rightarrow p_0$	$G \rightarrow p_0$
f	f	t	f	t
f	t	t	f	t
t	f	f	t	t
t	t	t	t	t

Properties of propositional formulas:

Meaning does **not** depend on interpretation of prop. variables which do **not** occur in the formula.

Tautology: A formula which is true in all situations. Examples: $p_0 \vee \neg p_0$,$p_0 \leftrightarrow p_0$, $(p_0 \rightarrow p_1) \leftrightarrow (\neg p_0 \vee p_1)$, $(p_0 \leftrightarrow p_1) \leftrightarrow ((p_0 \wedge p_1) \vee (\neg p_0 \wedge \neg p_1))$, T

Contradiction: A formula which is false in all situations. Example: $p_0 \wedge \neg p_0$, F

Equivalence. Two formulas ϕ and ψ are **equivalent** if $(\phi) \leftrightarrow (\psi)$ is a tautology.

Disjunctive normal form (DNF)

A formula is a **literal** if it is a propositional variable or a negated propositional variable.

A formula is in **disjunctive normal form** if it is a finite disjunction of finite conjunctions of literals.

For each p.f., there is an equivalent one in disjunctive normal form. (Construct from truth table.)

The conjunctions describe different combinations of values for the propositional variables for which the formula becomes true.

Example: $(p_0 \leftrightarrow p_1)$ is equivalent to the DNF formula $(p_0 \wedge p_1) \vee (\neg p_0 \wedge \neg p_1)$ (seen at truth table).

DNFs can be written without braces.

Not all prop. junctors are strictly necessary. By expressing a formula as DNF, only $\vee$, $\wedge$ and $\neg$ are used.

Negation and conjunction suffice to express equivalent formulas:

F is equivalent to $\phi \wedge \neg\phi$.

$\phi \vee \psi$ is equivalent to $\neg(\neg\phi \wedge \neg\psi)$.

T is equivalent to $\phi \vee \neg\phi$.

$\phi \rightarrow \psi$ is equivalent to $\neg\phi \vee \psi$.

$\phi \leftrightarrow \psi$ is equivalent to $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$.

Also implication and F alone suffice to express the others:

$\neg\phi$ is equivalent to $\phi \rightarrow F$

A set of junctors which suffices to express the others is called **complete**. A minimal complete set of junctors is called a **basis**.

Boole's complete set: $\{\neg, \wedge, \vee\}$

DeMorgan's bases: $\{\neg, \wedge\}$ and $\{\neg, \vee\}$

Frege's basis: $\{\neg, \rightarrow\}$

NAND basis: {NAND}

NOR basis: {NOR}

Some important equivalences of propositional logic / Boolean algebra:

The formulas in the right-hand column are all tautologies, i.e., they give always the value *T* (true)

double negation	$\neg\neg\phi \leftrightarrow \phi$
commutativity of conjunction	$\phi \wedge \psi \leftrightarrow \psi \wedge \phi$
commutativity of disjunction	$\phi \vee \psi \leftrightarrow \psi \vee \phi$
associativity of conjunction	$(\phi \wedge \psi) \wedge \chi \leftrightarrow \phi \wedge (\psi \wedge \chi)$
associativity of disjunction	$(\phi \vee \psi) \vee \chi \leftrightarrow \phi \vee (\psi \vee \chi)$
distributivity of conjunction into disjunction	$\phi \wedge (\psi \vee \chi) \leftrightarrow \phi \wedge \psi \vee \phi \wedge \chi$
distributivity of disjunction into conjunction	$\phi \vee (\psi \wedge \chi) \leftrightarrow (\phi \vee \psi) \wedge (\phi \vee \chi)$
implication	$(\phi \rightarrow \psi) \leftrightarrow (\neg\phi \vee \psi)$
negation of implication operands	$(\phi \rightarrow \psi) \leftrightarrow (\neg\psi \rightarrow \neg\phi)$
de Morgan laws	$\neg(\phi \vee \psi) \leftrightarrow (\neg\phi \wedge \neg\psi)$ $\neg(\phi \wedge \psi) \leftrightarrow (\neg\phi \vee \neg\psi)$

Propositional formulas alone are normally not sufficient to represent knowledge.
Even in mathematical statements, there are often additional informations connected with variables which cannot be expressed by a propositional formula:
E.g., "*for all* x, x+1>x"

Next complexity level of logic: Predicate logic
- not covered here in detail, only the basic notations:

Formulas of the form $\forall x : P(x)$ mean: **All x in the underlying universe** have the property P.

Formulas of the form $\exists x : P(x)$ mean: **There is an x in the underlying universe** which has the property P.

Example in the universe of the integers: $\forall x : \exists y : y > x$