First-Order Logic

Part 1: Syntax, Semantics, Pragmatics

JKU Course "Logic"

Wolfgang Schreiner Wolfgang Windsteiger

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at
Wolfgang.Windsteiger@risc.jku.at

July 16, 2020

Abstract

These lecture notes discuss the syntax, semantics, and pragmatics of the language of first-order logic, also called predicate logic. The presented material covers Part 1 of the module "First-Order Logic" of the course "Logic"; this course is given in the bachelor program "Computer Science" of the Johannes Kepler University Linz. Part 2 will discuss formal reasoning in first-order logic and more advanced topics.

Contents

1 Motivation

So far we have discussed the language of *propositional logic* where a logic formula F is constructed according to the following grammar:

$$F ::= p \mid \top \mid \bot \mid (\neg F) \mid (F_1 \wedge F_2) \mid (F_1 \vee F_2) \mid (F_1 \to F_2) \mid (F_1 \leftrightarrow F_2)$$

Each formula describes a "sentence" that is "true" or "false". Such sentences are constructed, starting with *propositional variables* (any element p of some given collection $\mathcal{P}$ of propositional variables) and the *propositional constants* "true" ($\top$) and "false" ($\bot$), by application of the *(logical) connectives* "not" ($\neg$), "and" ($\wedge$), "or" ($\vee$), "implies" ($\to$), and "equivalent" ($\leftrightarrow$).

However, the expressiveness of propositional logic is very limited. For instance, let us try to describe the logical content of the following sentence:

> For all numbers x and y it is the case that, if x is greater equal zero and y is greater equal zero, then x times y is zero or not less than x.

In propositional logic, the best we can do is to write the formula

$$\big((a \wedge b) \to (c \vee (\neg d))\big)$$

respectively (dropping the parentheses which the usual precedence rules make redundant)

$$a \wedge b \to c \vee \neg d.$$

Here we completely ignore the sentence's prefix "for all numbers x and y" and use the propositional variables a, b, c, and d as abstractions of the sentences "x is greater equal zero", "y is greater equal zero", "x times y is zero", and "x times y is less than x", respectively. While the formula thus captures somehow the "shape" of the sentence, it does not at all describe its "content"; in particular, while the original sentence is true for arbitrary numbers x and y, the later formula can be true or false, depending on the truth values of the propositional variables.

This inadequacy of the formalization of the sentence is a consequence of the fact that propositional logic is not expressive enough to talk about concrete objects (e.g., numbers), their relationships, and the fact whether a sentence is true for all or just for just some objects of a domain. *First-order predicate logic* (also called just *first-order logic* or *predicate logic*) is a much more expressive logic that extends propositional logic in such a way that is able to adequately formalize such sentences.

2 Syntax

In this section, we discuss the *syntax* (form) of formulas in first-order logic.

Terms and Formulas First-order logic has two different kinds of syntactic phrases ("expressions"), *terms* and *formulas*. A term is a phrase that denotes an "object" (a value) while a formula is a phrase that denotes a "property" of objects (a truth value "true" or "false"):

1. A term t is constructed according to the following grammar:

   $$t ::= v \mid c \mid f(t_1, \ldots, t_n)$$

   Thus a term can be one of the following:

   - A *variable* v (any element of some given collection $\mathcal{V}$ of variables); to a variable we may assign varying objects.
   - A *constant* c (any element of some given collection $\mathcal{C}$ of constants); in contrast to a variable, a constant denotes a fixed object.
   - a *function application*, i.e., the application of a *function symbol* f (any element of some given collection $\mathcal{F}$ of function symbols) with *arity* $n \geq 1$ to a sequence of n terms $t_1, \ldots, t_n$; such a function symbol f denotes an n-ary function that, when applied to n objects, returns another such object.

2. A formula F is constructed according to the following grammar (the underlined alternatives describe the extensions of first-order logic compared to propositional logic):

   $$F ::= \underline{p(t_1, \ldots, t_n)} \mid \top \mid \bot \mid (\neg F) \mid (F_1 \wedge F_2) \mid (F_1 \vee F_2) \mid (F_1 \rightarrow F_2) \mid (F_1 \leftrightarrow F_2)$$
   $$\mid \underline{(\forall v : F)} \mid \underline{(\exists v : F)}$$

   Thus, apart from the constructions already present in propositional logic, a formula can be one of the following entities:

   - An *atomic predicate* $p(t_1, \ldots, t_n)$, i.e., the application of a *predicate symbol* p (any element of some given collection of $\mathcal{P}$ of predicate symbols) with *arity* $n \geq 1$ to a sequence of n terms $t_1, \ldots, t_n$; such a predicate symbol p denotes an n-ary predicate that, when applied to n objects, returns "true" or "false".
   - A *universally quantified formula* $(\forall v : F)$, read as "for all (possible objects assigned to) v, F is true", with the *universal quantifier* "for all" ($\forall$) applied to a variable v and a formula F.
   - An *existentially quantified formula* $(\exists v : F)$, read as "there exists some (possible object assigned to) v, for which F is true", with the *existential quantifier* "exists" ($\exists$) applied to a variable v and a formula F.

The difference between a function and a predicate is that the application of a function returns an object, while the application of a predicate returns a truth value.

Writing Formulas According to the grammar above, we may express the informal sentence

> Tanja is female and every female is the daughter of her father.

by the following formula in first-order logic:

$$\big(\text{isFemale}(\text{Tanja}) \wedge \big(\forall x\colon (\text{isFemale}(x) \rightarrow \text{isDaughterOf}(x, \text{fatherOf}(x)))\big)\big).$$

Here Tanja is a constant, x is a variable, isFemale is a predicate symbol of arity 1 (a *unary* predicate symbol), fatherOf is a function symbol of arity 1, and isDaughterOf is a predicate symbol of arity 2 (a *binary* predicate symbol). Thus above formula contains the terms

- Tanja,
- x,
- $\text{fatherOf}(x)$

and the (sub)formulas

- $\text{isFemale}(\text{Tanja})$,
- $\text{isFemale}(x)$,
- $\text{isDaughterOf}(x, \text{fatherOf}(x))$,
- $(\text{isFemale}(x) \rightarrow \text{isDaughterOf}(x, \text{fatherOf}(x)))$,
- $\big(\forall x\colon (\text{isFemale}(x) \rightarrow \text{isDaughterOf}(x, \text{fatherOf}(x)))\big)$.

To reduce the number of parentheses, we agree on the following "binding powers":

$$(\neg) \gg (\wedge) \gg (\vee) \gg (\rightarrow) \gg (\leftrightarrow) \gg (\forall, \exists)$$

Here $(x) \gg (y)$ means "operator x binds stronger than operator y", i.e., $(F_1\ x\ F_2\ y\ F_3)$ is to be interpreted as $((F_1\ x\ F_2)\ y\ F_3)$, not as $(F_1\ x\ (F_2\ y\ F_3))$. Consequently, without parentheses the the scope of a quantified formula $\forall v\colon F$ or $\exists v\colon F$ reaches to the end of the enclosing formula. Therefore we may write the formula

$$\big(\text{isFemale}(\text{Tanja}) \wedge \big(\forall x\colon (\text{isFemale}(x) \rightarrow \text{isDaughterOf}(x, \text{fatherOf}(x)))\big)\big)$$

simply as

$$\text{isFemale}(\text{Tanja}) \wedge \forall x\colon \text{isFemale}(x) \rightarrow \text{isDaughterOf}(x, \text{fatherOf}(x)).$$

Now we are also in the position to write the sentence introduced in the previous section

> For all numbers x and y it is the case that, if x is greater equal zero and y is greater equal zero, then x times y is zero or not less than x.

as the formula

$$\forall x\colon \forall y\colon \text{greaterEqual}(x, \text{zero}) \wedge \text{greaterEqual}(y, \text{zero}) \rightarrow \text{equal}(\text{times}(x, y), \text{zero}) \vee \neg\text{lessThan}(\text{times}(x, y), x)$$

with the variables x and y, the constant 'zero', the binary function 'times', and the binary predicates 'greaterEqual', 'equal', and 'lessThan'.

Some more examples of informal sentences written as first-order formulas are given below:

- "Alex is Tom's sister":

  $$\text{isSisterOf}(\text{Alex}, \text{Tom})$$

  Here "Alex" and "Tom" are two constants while "is sister of" becomes a binary predicate.

- "Tom has a sister in Linz":

  $$\exists x\colon \text{isSisterOf}(x, \text{Tom}) \wedge \text{livesIn}(x, \text{Linz})$$

  The sentence thus can be read as "there exists some person such that this person is the sister of Tom and this person lives in Linz".

- "Tom has two sisters":

  $$\exists x, y\colon \neg\text{equal}(x, y) \wedge \text{isSisterOf}(x, \text{Tom}) \wedge \text{isSisterOf}(y, \text{Tom})$$

  The sentence can be read as "there exist some persons that are not identical and that are both the sister of Tom". The fact that the persons are not identical has to be explicitly stated since otherwise the variables x and y could refer to the same person.

- "Tom has no brother":

  $$\text{either: } \neg\exists x\colon \text{isBrotherOf}(x, \text{Tom})$$
  $$\text{or: } \forall x\colon \neg\text{isBrotherOf}(x, \text{Tom})$$

  Thus the sentence can be either read as "there does not exist a brother of Tom" or (equivalently) as "everybody is not the brother of Tom".

Abstract Syntax versus Concrete Syntax So far, we have written formulas and terms in a form where in every atomic formula $p(t_1, \ldots, t_n)$ and in every function application $f(t_1, \ldots, t_n)$ the predicate symbol p respectively the function symbol f appears before the terms to which they were applied ("prefix notation"). This *abstract syntax* of first-order logic allows to uniquely identify the "types" of "expressions" (the predicate and function symbols that are applied) and their "subexpressions" (the terms to which the symbols are applied). In practice, however, we usually encounter first-order logic formulas in various forms of *concrete syntax* where predicate and function symbols are written among their arguments ("infix notation") or

after their arguments ("postfix notation"); also many other forms (e.g., subscript notation) are possible. It is therefore important that we are able to determine how expressions in concrete syntax can be written in abstract syntax.

For instance, here are some examples of common (mathematical) operations, given first in the usual concrete syntax and then in the corresponding abstract syntax (once with the symbol given in its usual notation, once with the symbol replaced by a textual identifier):

Concrete Syntax	Abstract Syntax	
a/b	$/(a,b)$	$\mathrm{quotient}(a,b)$
$\frac{a}{b}$	$/(a,b)$	$\mathrm{quotient}(a,b)$
$a\|b$	$\|(a,b)$	$\mathrm{divides}(a,b)$
$a = b$	$=(a,b)$	$\mathrm{equals}(a,b)$
$a < b$	$<(a,b)$	$\mathrm{less}(a,b)$
$\sqrt{a}$	$\sqrt{}(a)$	$\mathrm{sqrt}(a)$
$a[i]$	$[\,](a,i)$	$\mathrm{index}(a,i)$
a_i	$[\,](a,i)$	$\mathrm{index}(a,i)$
$[a,b]$	$[\,](a,b)$	$\mathrm{interval}(a,b)$
f'	$'(f)$	$\mathrm{derivative}(f)$
$\int f$	$\int(f)$	$\mathrm{integral}(f)$
$f \to a$	$\to(f,a)$	$\mathrm{converges}(f,a)$

Consequently, a formula written in concrete syntax as $\frac{a}{a+b} < 1$ becomes in abstract syntax $<(/(a,+(a,b)),1)$ respectively $\mathrm{less}(\mathrm{quotient}(a,\mathrm{sum}(a,b)),\mathrm{one})$.

Sometimes the concrete syntax does not make the abstract syntax uniquely clear, for instance all of the following three translations are in principle legitimate:

Concrete Syntax	Abstract Syntax	
$a + b + c$	$+(a,b,c)$	$\mathrm{sum3}(a,b,c)$
	$+(a,+(b,c))$	$\mathrm{sum}(a,\mathrm{sum}(b,c))$
	$+(+(a,b),c)$	$\mathrm{sum}(\mathrm{sum}(a,b),c)$

Here either the concrete choice does not matter ($a+(b+c) = (a+b)+c$) or has to be determined from the context of the phrase.

Similarly, natural language phrases denoting formulas or terms have to be correspondingly translated into standard form:

Concrete Syntax	Abstract Syntax
the sum of all values from a to b	$\mathrm{summation}(a,b)$
the remainder of a divided by b	$\mathrm{remainder}(a,b)$
a is a divisor of b	$\mathrm{divides}(a,b)$
f converges to a	$\mathrm{converges}(f,a)$

Conditions and Quantifiers Frequently quantified statements contain "filter conditions" that constrain the domain of the quantification; for instance, the statements

- "every natural number is greater equal zero" and
- "there exists a natural number whose predecessor is zero"

constrain the domain of the quantification to those objects that satisfy the condition "is a natural number". Let us interpret the statement "x is a natural number" as the formula $x \in \mathbb{N}$ (with constant $\mathbb{N}$ denoting the set of all natural numbers and symbol $\in$ denoting the binary "is-element-of" predicate), the statement "x is greater equal zero" as the formula $x \geq 0$, and the statement "the predecessor of x is zero" as the formula $x - 1 = 0$ (with constants 0 and 1, binary function $-$, and binary predicates $\geq$ and $=$). Then we can formalize the quantified statements above as follows:

$$\forall x \colon x \in \mathbb{N} \to x \geq 0$$
$$\exists x \colon x \in \mathbb{N} \land x - 1 = 0$$

Here the first proposition states literally "for every value x, *if* x is a natural number, *then* x is greater equal zero"; the second proposition claims "there exists some x such that x is a natural number *and* x is greater than zero"; note that these formulas involve two different logical connectives $\to$ and $\land$ to express the filtering conditions. These considerations motivate the following two formula patterns for quantified formulas with filter condition C:

$$\forall x \colon C \to F$$
$$\exists x \colon C \land F$$

These patterns are often abbreviated as follows:

$$\forall C \colon F$$
$$\exists C \colon F$$

For example, above statements about natural numbers are often written as:

$$\forall x \in \mathbb{N} \colon x \geq 0$$
$$\exists x \in \mathbb{N} \colon x - 1 = 0$$

Here one must be clearly aware of the different interpretations of the logical connectives, once as $\to$ and once as $\land$.

From an abbreviated formula it is always necessary to deduce the quantified variable from the context. For instance, the formula

$$\forall x \in \mathbb{N} \colon \exists x < y \colon y < x + 2$$

must be expanded to

$$\forall x \colon x \in \mathbb{N} \to \exists y \colon x < y \land y < x + 2$$

where the universal quantifier binds variable x and the existential quantifier binds variable y. This formula thus expresses the actually intended statement "for every natural number x, there exists some number y between x and $x + 2$".

Free and Bound Variables The truth value of a formula

$$\text{equal}(x, \text{zero})$$

depends on the value we assign to the variable x: if we assign to x the value "zero", the formula is true; for any other value, the formula is false. We say that the variable x is *free* in this formula. On the contrary, the truth values of the formulas

$$\forall x : \text{equal}(x, \text{zero})$$
$$\exists x : \text{equal}(x, \text{zero})$$

do not depend on the values of x: the first formula is false (assuming that there exist multiple values, not every value is zero), the second formula is true (some value is zero). We say that x is *bound* in the formula. The quantifiers $\forall$ and $\exists$ are logical operators that bind variables and thus make the truth value of a formula independent of any assignments of values to these variables. A *closed* formula is a formula without free variables, i.e., all of its variables have been bound by quantifiers; the truth value of a closed formula does therefore not depend on values assigned to any of its variables.

The following example demonstrates, how the free variables of a formula can be determined "inside-out", i.e., starting with the innermost formulas and then proceeding outwards, by adding or removing free variables:

$$\underbrace{\forall x : \underbrace{\underbrace{p(x,w)}_{\text{free: } x,w} \rightarrow \underbrace{\exists y : \underbrace{q(x,y,z)}_{\text{free: } x,y,z}}_{\text{free: } x,z}}_{\text{free: } x,w,z}}_{\text{free: } w,z}$$

The computation of free variables is formalized by the following recursive function $\text{fv}(F)$ that returns the set of free variables of a given formula F:

$$\begin{aligned}
\text{fv}(p(t_1, \ldots, t_n)) &= \text{fv}(t_1) \cup \ldots \cup \text{fv}(t_n) \\
\text{fv}(\top) &= \emptyset \\
\text{fv}(\bot) &= \emptyset \\
\text{fv}(\neg F) &= \text{fv}(F) \\
\text{fv}(F_1 \wedge F_2) &= \text{fv}(F_1) \cup \text{fv}(F_2) \\
\text{fv}(F_1 \vee F_2) &= \text{fv}(F_1) \cup \text{fv}(F_2) \\
\text{fv}(F_1 \rightarrow F_2) &= \text{fv}(F_1) \cup \text{fv}(F_2) \\
\text{fv}(F_1 \leftrightarrow F_2) &= \text{fv}(F_1) \cup \text{fv}(F_2) \\
\text{fv}(\forall v : F) &= \underline{\text{fv}(F) \backslash \{v\}} \\
\text{fv}(\exists v : F) &= \underline{\text{fv}(F) \backslash \{v\}}
\end{aligned}$$

This computation depends on another recursive function $\mathrm{fv}(t)$ that returns the set of free variables of a given term t:

$$\begin{aligned}\mathrm{fv}(v) &= \{v\}\\ \mathrm{fv}(c) &= \emptyset\\ \mathrm{fv}(f(t_1,\ldots,t_n)) &= \mathrm{fv}(t_1)\cup\ldots\cup\mathrm{fv}(t_n)\end{aligned}$$

For instance, for $F = \forall x\colon p(x,w)\to\exists y\colon q(x,y,z)$ (see above example), we have:

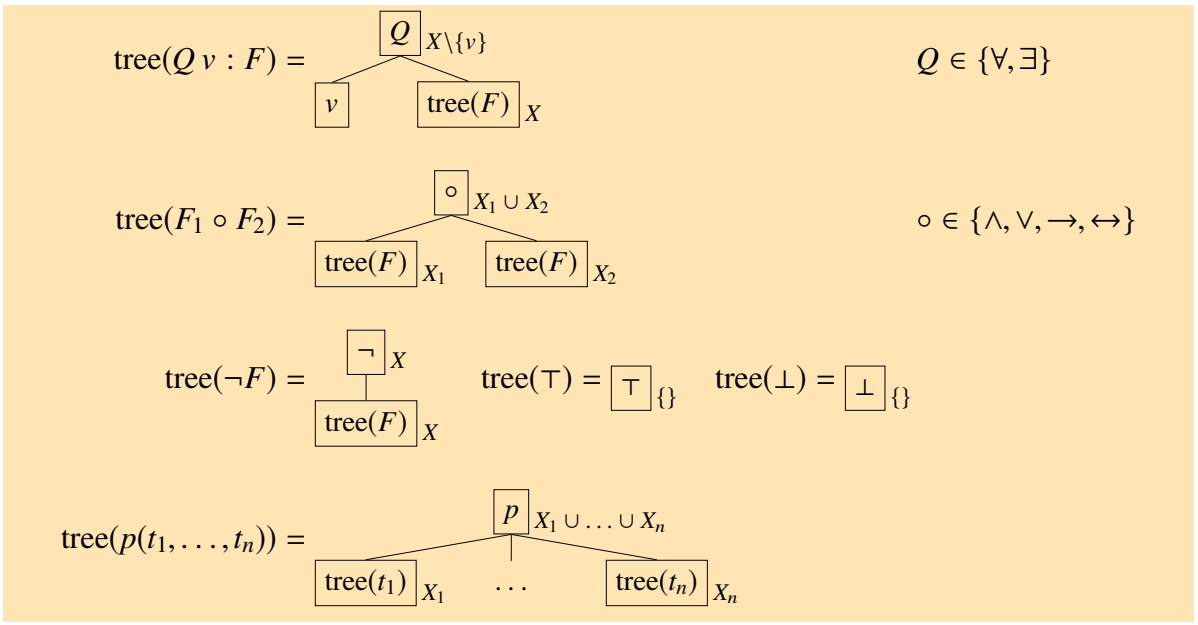
$$\begin{aligned}\mathrm{fv}(q(x,y,z)) &= \{x,y,z\}\\ \mathrm{fv}(\exists y\colon q(x,y,z)) &= \mathrm{fv}(q(x,y,z))\backslash\{y\}\\ &= \{x,y,z\}\backslash\{y\} = \{x,z\}\\ \mathrm{fv}(p(x,w)) &= \{x,w\}\\ \mathrm{fv}(p(x,w)\to\exists y\colon q(x,y,z)) &= \mathrm{fv}(p(x,w))\cup\mathrm{fv}(\exists y\colon q(x,y,z))\\ &= \{x,w\}\cup\{x,z\} = \{x,w,z\}\\ \mathrm{fv}(\forall x\colon p(x,w)\to\exists y\colon q(x,y,z)) &= \mathrm{fv}(p(x,w)\to\exists y\colon q(x,y,z))\backslash\{x\}\\ &= \{x,w,z\}\backslash\{x\} = \{w,z\}\end{aligned}$$

Syntax Analysis To understand in detail a first-order formula, we apply the technique of "syntax analysis" to generate from the concrete syntax of the formula (a linear text with possible multiple interpretations), a unique description of its abstract syntax, the *abstract syntax tree* (a data structure with only a single interpretation). Each node in this tree represents an expression (formula or term) within the given formula; this node is annotated with a tag that indicates the type of the expression (this tag may be a variable, a constant, a function symbol, a predicate symbol, a logical connective, or a quantifier); the children of the node represent the expression's subexpressions. Syntax analysis processes a formula "top-down" by analyzing

- its quantified formulas (constructed by quantifiers from variables and sub-formulas),
- its propositional formulas (constructed by logical connectives from sub-formulas),
- its atomic formulas (constructed by predicate symbols from terms),
- its terms (variables or constants or function applications that are constructed by function symbols from sub-terms).

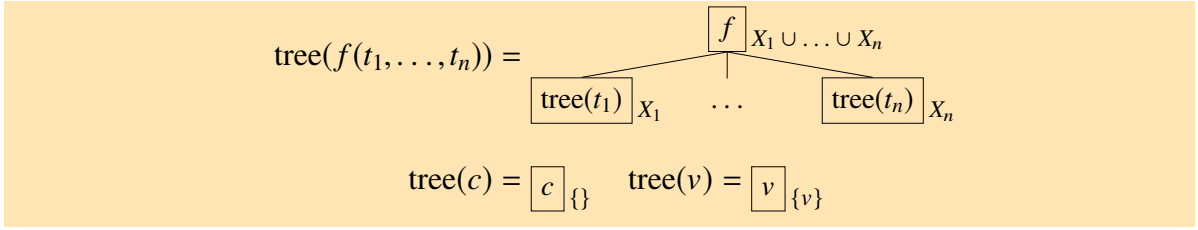
This process determines the role of each name as a variable, constant, function symbol, or predicate symbol and determines the free variables of every formula (often names like $x, y, z, \ldots$ are used for variables $a, b, c, \ldots$ for constants, $f, g, h, \ldots$ for function symbols $p, q, r, \ldots$ for predicate symbols, but this need not always be the case).

Formally, the analysis can be described by a function $\mathrm{tree}(F)$ that returns the abstract syntax tree of formula F; this function is defined as follows for every kind of formula F:



By application of tree(F), the abstract syntax tree of F is constructed top-down (from the root towards the leaves). As a side-effect, the root of this tree is annotated with the set of free variables of F whose computation proceeds bottom-up (from the leaves towards the root).

Likewise, tree(t) is defined for every kind of term t:



As an example, take the following formula in concrete syntax:

$$\forall x \in \mathbb{N}\colon\ x > 0 \to \exists y \in \mathbb{N}\colon\ y + 1 = x.$$

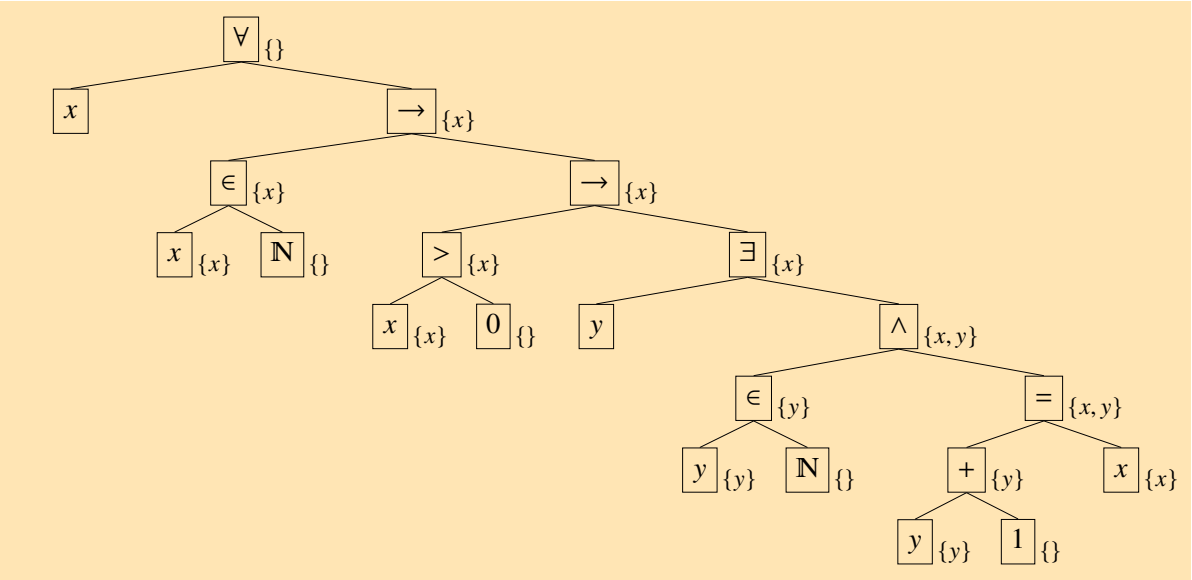
To analyze this formula, we first expand the conditions in the quantifiers:

$$\forall x\colon\ x \in \mathbb{N} \to (x > 0 \to \exists y\colon\ y \in \mathbb{N} \wedge y + 1 = x).$$

To clarify the structure of this formula, we write parentheses around every subformula:

$$(\forall x\colon\ ((x \in \mathbb{N}) \to ((x > 0) \to (\exists y\colon\ ((y \in \mathbb{N}) \wedge (y + 1 = x)))))).$$

The syntax analysis of this formula yields the abstract syntax tree shown below:



The top-down syntax analysis leading to this tree is explained in detail below (the abbreviations 'F' respectively 'Fs' denote "formula(s)", 'T' respectively 'Ts' denote "terms(s)", 'Q' denotes "quantifier", 'LC' denotes "logical connective", 'PS' denotes "predicate symbol", 'FS' denotes "function symbol", 'V' denotes "variable", 'C' denotes "constant"):

$$\forall x\colon \underbrace{\underbrace{\underbrace{x}_{\text{V}} \in \underbrace{\mathbb{N}}_{\text{C}}}_{\text{PS } \in,\ \text{Ts } (x, \mathbb{N})} \to \underbrace{\underbrace{x}_{\text{V}} > \underbrace{0}_{\text{C}}}_{\text{PS } >,\ \text{Ts } (x, 0)} \to \exists y\colon \underbrace{\underbrace{\underbrace{y}_{\text{V}} \in \underbrace{\mathbb{N}}_{\text{C}}}_{\text{PS } \in,\ \text{Ts } (y, \mathbb{N})} \wedge \underbrace{\underbrace{y}_{\text{V}} + \underbrace{1}_{\text{C}}}_{\text{FS } +,\ \text{Ts } (y, 1)} = \underbrace{x}_{\text{V}}}_{\text{PS } =,\ \text{Ts } (y+1, x)}}_{\text{LC } \wedge,\ \text{Fs } (y \in \mathbb{N},\ y+1 = x)}}_{\text{Q } \exists,\ \text{V } y,\ \text{F } (y \in \mathbb{N} \wedge \ldots)}}_{\text{LC } \to,\ \text{Fs } (x > 0,\ \exists y : \ldots)}}_{\text{LC } \to,\ \text{Fs } (x \in \mathbb{N},\ x > 0 \to \ldots)}}_{\text{Q } \forall,\ \text{V } x,\ \text{F } (x \in \mathbb{N} \to \ldots)} .$$

Reading the analysis from the last line to the first, the first step has determined the global structure of the formula as a quantified formula with quantifier $\forall$, variable x, and subformula $(x \in \mathbb{N} \to \ldots)$. The second step has determined the structure of the subformula as an implication with the logical connective $\to$ as the outermost operator and two subformulas. The first subformula $(x \in \mathbb{N})$ has been analyzed as an atomic formula with predicate symbol $\in$ and two terms x and $\mathbb{N}$; the first term is a variable, the second one a constant.

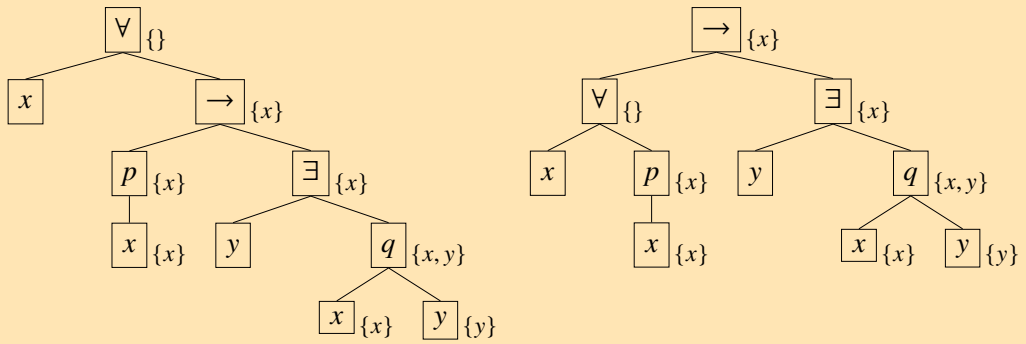
Similarly, the analysis of the second subformula $(x > 0 \to \ldots)$ has been analyzed as an implication. Its left subformula $(x > 0)$ is an atomic formula with predicate symbol $>$ and terms x and 0, a variable and a constant, respectively. The right subformula is a quantified formula with quantifier $\exists$, variable y, and subformula $(y \in \mathbb{N} \wedge y + 1 = x)$. This formula is a conjunction with $\wedge$ as the outermost logical connective. Its left subformula $(y \in \mathbb{N})$ is

an atomic predicate with predicate symbol $\in$ and terms y and $\mathbb{N}$, a variable and a constant, respectively. Also the second subformula $(y + 1 = x)$ is an atomic predicate with terms $(y + 1)$ and x. The term $(y + 1)$ is an application of function symbol $+$ to terms y and 1, a variable and a constant, respectively. The term x is a variable.

It should be noted, that (if we ignore the convention for the precedence rules of logical operators) that the concrete syntax of a formula does not necessarily determine the abstract syntax tree in a unique way. For instance, the formula

$$\forall x\colon p(x) \to \exists y\colon q(x, y)$$

may be analyzed as either of the following two syntax trees:



The left tree describes the "intended" structure of the formula which arises from the binding rules for the logical operators; it corresponds to the following parenthesization:

$$\forall x\colon (p(x) \to \exists y\colon q(x, y))$$

The right tree describes another structure which arises from ignoring the binding rules; it corresponds to this parenthesization:

$$(\forall x\colon p(x)) \to (\exists y\colon q(x, y))$$

Generally, abstract syntax trees corresponding to closed formulas are intended (the left syntax tree has no free variables, the right one has free variable x); if the syntactic structure might be doubtful, additional parenthesis should be added.

Further Constructs While logically not absolutely necessary, it is convenient to extend the syntax of formulas and terms to include also the following phrases:

$$(\textbf{let } v = t \textbf{ in } E)$$
$$(\textbf{if } F \textbf{ then } E_1 \textbf{ else } E_2)$$

In the phrase $(\textbf{let } v = t \textbf{ in } E)$, sometimes also written as $(E \textbf{ where } v = t)$ or $(E|_{v=t})$ (the parentheses are usually dropped), expression E can be a formula or a term; consequently, the whole phrase is then a formula or a term. The meaning of the phrase is identical to $E[t/x]$, which means that in E every free occurrence of variable v is replaced by term t. Thus the

construct represents a *quantifier* that binds variable v. The formula (**let** $v = t$ **in** F) is actually equivalent to $(\exists v : v = t \wedge F)$, but the first formulation makes the meaning more transparent.

In the phrase (**if** F **then** E_1 **else** E_2), F is a formula and E_1 and E_2 can be either both formulas or both terms; consequently, the whole phrase is then a formula or a term. The meaning of the phrase is that of E_1, if F is true, otherwise it is that of E_2. The formula (**if** F **then** F_1 **else** F_2) is actually equivalent to $(F \rightarrow F_1) \wedge (\neg F \rightarrow F_2)$ but the first formulation makes the meaning more transparent.

Furthermore, mathematics knows many more quantifiers than just the universal and the existential quantifier. For instance:

- $\sum_{i=a}^{b} t$ binds variable i; its meaning is the sum $t[a/i] + \cdots + t[b/i]$.
- $\prod_{i=a}^{b} t$ binds variable i; its meaning is the product $t[a/i] * \cdots * t[b/i]$.
- $\{x \in S \mid F\}$ binds variable x; its meaning is the set of all elements x of set S for which formula F is true.
- $\{t \mid x \in S \wedge F\}$: this quantifier binds variable x; its meaning is set of all values of term t where x is some element of set S for which formula F is true.
- $\lim_{x \rightarrow v} t$ binds variable x; its meaning is the limit of term t when x goes to value v.
- $\max_{x \in S} t$ binds variable x; its meaning is the maximum of all values of term t where x is some element of set S.
- $\min_{x \in S} t$ binds variable x; its meaning is the minimum of all values of term t where x is some element of set S.

Generally, whenever a language construct introduces a local variable, it is from the logical point of view a quantifier.

3 Semantics

We now turn our attention to the *semantics* (meaning) of formulas in first-order logic.

Structures and assignments In first-order logic, the semantics of of a phrase (formula or term) depends on two entities, a *structure* and an *assignment*:

Structure A *structure* (D, I) is a pair of a domain D and an interpretation I on D.

Domain A domain D is a non-empty collection of objects that represents the "universe" about which the formula talks; for example, D may be any non-empty set.

Interpretation An interpretation I on D maps

- every constant $c \in \mathcal{C}$ to a value $I(c)$ in D; in set theory we write this as:

$$I(c) \in D;$$

- every function symbol $f \in \mathcal{F}$ with arity n to an n-ary function $I(f)$ on D; in set theory we write this as:

$$I(f) : \underbrace{D \times \ldots \times D}_{n \text{ times}} \to D;$$

- every predicate symbol $p \in \mathcal{P}$ with arity n to an n-ary predicate $I(p)$ on D ; in set theory we write this as:

$$I(p) \subseteq \underbrace{D \times \ldots \times D}_{n \text{ times}}.$$

Assignment An assignment a maps every variable $v \in \mathcal{V}$ to a value $a(v)$ in D; in set theory we write this as:

$$a(v) \in D.$$

As an example, we may have the structure (D, I) defined as

$$\begin{aligned} D &= \mathbb{N} \\ I &= [0 \mapsto \text{zero}, + \mapsto \text{add}, < \mapsto \text{less-than}, \ldots] \end{aligned}$$

where $f = [x \mapsto y, \ldots]$ means $f(x) = y$. Here the domain D is the set $\mathbb{N}$ of natural numbers, $I(0)$ is the natural number zero, $I(+)$ denotes the addition function on $\mathbb{N}$, and $I(<)$ denotes the "less-than" predicate on $\mathbb{N}$. The assignment

$$a = [x \mapsto \text{one}, y \mapsto \text{zero}, z \mapsto \text{three}, \ldots]$$

maps the variables x, y, and z to the natural numbers one, zero, and three, respectively.

Informal Semantics Given a structure (D, I) and an assignment a, the meaning of terms and formulas can be informally described as follows:

Terms The meaning of a term is an object in D.

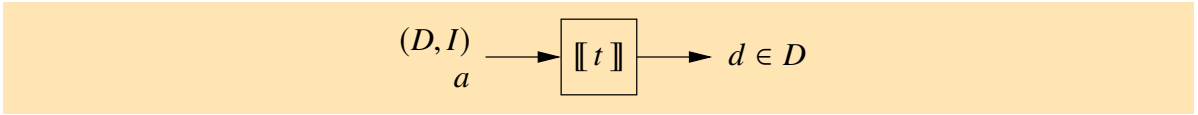
- The meaning of a variable v is the value assigned to it by a, i.e., $a(v)$.
- The meaning of a constant c is its interpretation in I, i.e., $I(c)$.
- The meaning of a function application $f(t_1, \ldots, t_n)$ is the result of applying its interpretation in I, i.e., the function $I(f)$, to the meanings of the terms $t_1, \ldots, t_n$.

Formulas The meaning of a formula is "true" or "false".

- The meaning of an atomic formula $p(t_1, \ldots, t_n)$ is the result of applying its interpretation in I, i.e., the predicate $I(p)$, to the meanings of the terms $t_1, \ldots, t_n$.

  Equality As a special case with a fixed interpretation, the meaning of an equality $t_1 = t_2$ is "true", if and only if t_1 has the same meaning as t_2.
- The meaning of the logical constant $\top$ is "true", the meaning of $\bot$ is "false". The meaning of the compound formulas $\neg F$, $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \rightarrow F_2$, and $F_1 \leftrightarrow F_2$ is determined from the meanings of the subformulas (F, F_1, F_2) by applying the truth table of propositional logic for the corresponding logical connective.
- The meaning of the universally quantified formula $(\forall x\colon F)$ is true if and only if the meaning of formula F is true *for all* possible values that we give to the variable x in assignment a.
- The meaning of the existentially quantified formula $(\exists x\colon F)$ is true if and only if the meaning of formula F is true *for some* possible value that we give to the variable x in assignment a.

Term Semantics We now formally define the semantics $[\![t]\!]_a^{D,I}$ of a term t in structure (D, I) and assignment a, which is a value in D. The computation of this semantics can be visualized as follows:



The semantics itself is defined by the following equations:

$$
\begin{aligned}
[\![v]\!]_a^{D,I} &:= a(v) \\
[\![c]\!]_a^{D,I} &:= I(c) \\
[\![f(t_1, \ldots, t_n)]\!]_a^{D,I} &:= I(f)\big([\![t_1]\!]_a^{D,I}, \ldots, [\![t_n]\!]_a^{D,I}\big)
\end{aligned}
$$

As an example, consider the semantics of the following term:

$$x + (y + 0)$$

First we consider the semantics of this term with structure (D, I) defined as

$$D = \mathbb{N} = \{\text{zero}, \text{one}, \text{two}, \text{three}, \ldots\}$$
$$I = [0 \mapsto \text{zero}, + \mapsto \text{add}, \ldots]$$

where the elements of D are natural numbers and the operation '+' is interpreted as the addition of two such numbers. For the assignment

$$a = [x \mapsto \text{one}, y \mapsto \text{two}, \ldots]$$

the term has the value three as determined by the following evaluation:

$$\begin{aligned}
[\![\, x + (y + 0) \,]\!]_a^{D,I} &= \text{add}([\![\, x \,]\!]_a^{D,I}, [\![\, y + 0 \,]\!]_a^{D,I}) \\
&= \text{add}(a(x), [\![\, y + 0 \,]\!]_a^{D,I}) \\
&= \text{add}(\text{one}, [\![\, y + 0 \,]\!]_a^{D,I}) \\
&= \text{add}(\text{one}, \text{add}([\![\, y \,]\!]_a^{D,I}, [\![\, 0 \,]\!]_a^{D,I})) \\
&= \text{add}(\text{one}, \text{add}(a(y), I(0)) \\
&= \text{add}(\text{one}, \text{add}(\text{two}, \text{zero})) \\
&= \text{add}(\text{one}, \text{two}) \\
&= \text{three}.
\end{aligned}$$

Now we consider the structure (D, I) defined as

$$D = \mathcal{P}(\mathbb{N}) = \{\emptyset, \{\text{zero}\}, \{\text{one}\}, \{\text{two}\}, \ldots, \{\text{zero}, \text{one}\}, \ldots\}$$
$$I = [0 \mapsto \emptyset, + \mapsto \text{union}, \ldots]$$

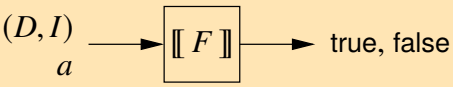
where the values of D are sets of natural numbers and operation '+' is interpreted as the union of two such sets. For the assignment

$$a = [x \mapsto \{\text{one}\}, y \mapsto \{\text{two}\}, \ldots]$$

the value of the term is $\{\text{one}, \text{two}\}$ as determined by the following evaluation:

$$\begin{aligned}
[\![\, x + (y + 0) \,]\!]_a^{D,I} &= \text{union}([\![\, x \,]\!]_a^{D,I}, [\![\, y + 0 \,]\!]_a^{D,I}) \\
&= \text{union}(a(x), [\![\, y + 0 \,]\!]_a^{D,I}) \\
&= \text{union}(\{\text{one}\}, [\![\, y + 0 \,]\!]_a^{D,I}) \\
&= \text{union}(\{\text{one}\}, \text{union}([\![\, y \,]\!]_a^{D,I}, [\![\, 0 \,]\!]_a^{D,I})) \\
&= \text{union}(\{\text{one}\}, \text{union}(a(y), I(0)) \\
&= \text{union}(\{\text{one}\}, \text{union}(\{\text{two}\}, \emptyset)) \\
&= \text{union}(\{\text{one}\}, \{\text{two}\}) \\
&= \{\text{one}, \text{two}\}
\end{aligned}$$

Formula Semantics Now we formally define the semantics $[\![F]\!]^{D,I}_a \in D$ of a formula F. Its computation can be visualized by the following figure:



The semantics itself is defined by the following equations:

$$
\begin{aligned}
[\![p(t_1,\ldots,t_n)]\!]^{D,I}_a &:= I(p)([\![t_1]\!]^{D,I}_a,\ldots,[\![t_n]\!]^{D,I}_a) \\
[\![t_1 = t_2]\!]^{D,I}_a &:= \begin{cases} \text{true} & \text{if } [\![t_1]\!]^{D,I}_a = [\![t_2]\!]^{D,I}_a \\ \text{false} & \text{else} \end{cases} \\
[\![\top]\!]^{D,I}_a &:= \text{true} \\
[\![\bot]\!]^{D,I}_a &:= \text{false} \\
[\![\neg F]\!]^{D,I}_a &:= \begin{cases} \text{true} & \text{if } [\![F]\!]^{D,I}_a = \text{false} \\ \text{false} & \text{else} \end{cases} \\
[\![F_1 \wedge F_2]\!]^{D,I}_a &:= \begin{cases} \text{true} & \text{if } [\![F_1]\!]^{D,I}_a = [\![F_2]\!]^{D,I}_a = \text{true} \\ \text{false} & \text{else} \end{cases} \\
[\![F_1 \vee F_2]\!]^{D,I}_a &:= \begin{cases} \text{false} & \text{if } [\![F_1]\!]^{D,I}_a = [\![F_2]\!]^{D,I}_a = \text{false} \\ \text{true} & \text{else} \end{cases} \\
[\![F_1 \rightarrow F_2]\!]^{D,I}_a &:= \begin{cases} \text{false} & \text{if } [\![F_1]\!]^{D,I}_a = \text{true and } [\![F_2]\!]^{D,I}_a = \text{false} \\ \text{true} & \text{else} \end{cases} \\
[\![F_1 \leftrightarrow F_2]\!]^{D,I}_a &:= \begin{cases} \text{true} & \text{if } [\![F_1]\!]^{D,I}_a = [\![F_2]\!]^{D,I}_a \\ \text{false} & \text{else} \end{cases} \\
[\![\forall x\colon F]\!]^{D,I}_a &:= \begin{cases} \text{true} & \text{if } [\![F]\!]^{D,I}_{a[x\mapsto d]} = \text{true for all } d \text{ in } D \\ \text{false} & \text{else} \end{cases} \\
[\![\exists x\colon F]\!]^{D,I}_a &:= \begin{cases} \text{true} & \text{if } [\![F]\!]^{D,I}_{a[x\mapsto d]} = \text{true for some } d \text{ in } D \\ \text{false} & \text{else} \end{cases}
\end{aligned}
$$

Here $a[x \mapsto d]$ denotes the assignment that is identical to a except that it maps x to d:

$$
a[x \mapsto d](y) := \begin{cases} d & \text{if } x = y \\ a(y) & \text{else} \end{cases}
$$

As an example, we consider the semantics of the formula

$$\forall x\colon \exists y\colon x + y = z$$

in structure (D, I) defined as

$$D = \mathbb{N}_3 = \{\text{zero}, \text{one}, \text{two}\}$$
$$I = [0 \mapsto \text{zero}, + \mapsto \text{add}, \ldots]$$

For the assignment

$$a = [x \mapsto \text{one}, y \mapsto \text{two}, z \mapsto \text{two}, \ldots]$$

we can determine the truth value $[\![\forall x \colon \exists y \colon x + y = z]\!]_a^{D,I}$ of this universally quantified formula by considering the truth values of the existentially quantified formula for all possible values of x:

- $[\![\exists y \colon x + y = z]\!]^{D,I}_{a[x \mapsto \text{zero}]}$
- $[\![\exists y \colon x + y = z]\!]^{D,I}_{a[x \mapsto \text{one}]}$
- $[\![\exists y \colon x + y = z]\!]^{D,I}_{a[x \mapsto \text{two}]}$

For every value of x, we consider all possible values of y. If there exists some value of y that makes the equality true, the existentially quantified formula is true for the chosen value of x:

- $[\![\exists y \colon x + y = z]\!]^{D,I}_{a[x \mapsto \text{zero}]} = \text{true}$
  - $[\![x + y = z]\!]^{D,I}_{a[x \mapsto \text{zero}, y \mapsto \text{zero}]} = \text{false}$
  - $[\![x + y = z]\!]^{D,I}_{a[x \mapsto \text{zero}, y \mapsto \text{one}]} = \text{false}$
  - $[\![x + y = z]\!]^{D,I}_{a[x \mapsto \text{zero}, y \mapsto \text{two}]} = \underline{\text{true}}$
- $[\![\exists y \colon x + y = z]\!]^{D,I}_{a[x \mapsto \text{one}]} = \text{true}$
  - $[\![x + y = z]\!]^{D,I}_{a[x \mapsto \text{one}, y \mapsto \text{zero}]} = \text{false}$
  - $[\![x + y = z]\!]^{D,I}_{a[x \mapsto \text{one}, y \mapsto \text{one}]} = \underline{\text{true}}$
  - $[\![x + y = z]\!]^{D,I}_{a[x \mapsto \text{one}, y \mapsto \text{two}]} = \text{false}$
- $[\![\exists y \colon x + y = z]\!]^{D,I}_{a[x \mapsto \text{two}]} = \text{true}$
  - $[\![x + y = z]\!]^{D,I}_{a[x \mapsto \text{two}, y \mapsto \text{zero}]} = \underline{\text{true}}$
  - $[\![x + y = z]\!]^{D,I}_{a[x \mapsto \text{two}, y \mapsto \text{one}]} = \text{false}$
  - $[\![x + y = z]\!]^{D,I}_{a[x \mapsto \text{two}, y \mapsto \text{two}]} = \text{false}$

Since thus for all values of x the existentially quantified formula is true, the universally quantified formula is true:

$$[\![\forall x \colon \exists y \colon x + y = z]\!]^{D,I}_a = \text{true}$$

Clearly, the truth value of a formula depends on the considered interpretation of symbols. Take for example the following formula:

$$\forall x\colon R(x,x)$$

Choosing as the structure (D, I) the domain of natural numbers with the interpretation of R as the divisibility relation, above formula states "every natural number is divisible by itself", which is clearly true. However, if we interpret R as the "less than" relation, the formula states "every natural number is less than itself", which is clearly false.

Furthermore, the truth value of a non-closed formula depends on the considered assignment of variables to values. Take for example the following formula with free variables y and z:

$$\exists x\colon R(y,x) \wedge R(x,z)$$

We consider as (D, I) the domain of natural numbers with the interpretation of R as the "less than" relation. For the assignment $y = 2$ and $z = 4$, the formula is true, because there exists a natural number x with $2 < x$ and $x < 4$, namely $x := 3$. However, for the assignment $y = 2$ and $z = 3$, the formula is false, because there does not exist any natural number x with $2 < x$ and $x < 3$.

Please also note that the semantics of a formula depends on the order of nested quantifiers. Choosing as the structure (D, I) the domain of natural numbers with the usual interpretation of predicate symbol $<$, we have for every assignment a:

- The formula $(\forall x\colon \exists y\colon x < y)$ is true: for every natural number x, there exists a number y greater than x, namely $y := x + 1$.
- The formula $(\exists y\colon \forall x\colon x < y)$ is false. To show this, we assume that the formula is true and derive a contradiction. Because of the assumption, there exists some natural number y such that $(\forall x\colon x < y)$ is true. But then, since $x < y$ is true for every value of x, it is also true for $x := y$. Thus $y < y$ is true, which we know to be false.

Semantic Notions We are now going to define several fundamental notions of the semantics of first-order logic. In the following, let F denote a formula, $M = (D, I)$ a structure, and a an assignment.

Satisfiability Formula F is *satisfiable*, if there exists some structure M and assignment a such that $[\![F]\!]^M_a = \text{true}$.

- Example: $p(0, x)$ is satisfiable; $q(x) \wedge \neg q(x)$ is not.

Model Structure M is a *model* of formula F, written as $M \models F$, if for every assignment a, we have $[\![F]\!]^M_a = \text{true}$.

- Example: $(\mathbb{N}, [0 \mapsto \text{zero}, p \mapsto \text{less-equal}]) \models p(0, x)$

Validity Formula F is *valid*, written as $\models F$, if every structure M is a model of F, i.e., for every structure M we have $M \models F$.

- Example: $\models p(x) \wedge (p(x) \rightarrow q(x)) \rightarrow q(x)$

Thus a satisfiable formula is a "possibility" that may be true for some structure and assignment; a valid formula is a "certainty" that is true for all possible structures and assignments. Consequently, in analogy to propositional logic, also first-order logic has the following properties:

- F is satisfiable, if $\neg F$ is not valid.
- F is valid, if $\neg F$ is not satisfiable.

Furthermore, we introduce the following notions:

Logical Consequence Formula F_2 is a *logical consequence* of formula F_1, written as $F_1 \models F_2$, if for every structure M and assignment a, the following is true:

If $[\![F_1]\!]_a^M = \text{true}$, then also $[\![F_2]\!]_a^M = \text{true}$.

- Example: $p(x) \wedge (p(x) \rightarrow q(x)) \models q(x)$

Logical Consequence Generalized Formula F is a *logical consequence* of multiple formulas $F_1, \ldots, F_n$, written as $F_1, \ldots, F_n \models F$, if for every structure M and assignment a, the following is true:

If for every formula F_i we have $[\![F_i]\!]_a^M = \text{true}$, then $[\![F]\!]_a^M = \text{true}$.

- Example: $p(x), q(x) \models p(x) \wedge q(x)$

Logical Equivalence Formulas F_1 and F_2 are *logically equivalent*, written as $F_1 \Leftrightarrow F_2$, if and only if F_1 is a logical consequence of F_2 and F_2 is a logical consequence of F_1, i.e., $F_1 \models F_2$ and $F_2 \models F_1$.

- Example: $p(x) \rightarrow q(x) \Leftrightarrow \neg p(x) \vee q(x)$

We then have the following propositions:

- Formula F_2 is a logical consequence of formula F_1 (i.e., $F_1 \models F_2$) if and only if the formula $(F_1 \rightarrow F_2)$ is valid.
- Formula F is a logical consequence of formulas $F_1, \ldots, F_n$ (i.e., $F_1, \ldots, F_n \models F$) if and only if the formula $(F_1 \wedge \ldots \wedge F_n \rightarrow F)$ is valid.
- Formula F_1 and formula F_2 are logically equivalent (i.e., $F_1 \Leftrightarrow F_2$) if and only if the formula $(F_1 \leftrightarrow F_2)$ is valid.

Thus the logical consequence between formulas and the logical equivalence of formulas can be reduced to the validity of a single formula (respectively to the non-satisfiability of the negation of this formula), an implication, and an equivalence, respectively.

Equivalent formulas can be substituted in any context:

- If $F \Leftrightarrow F'$ and $G \Leftrightarrow G'$, then the following equivalences hold:

$$\begin{aligned}
\neg F &\Leftrightarrow \neg F' \\
F \wedge G &\Leftrightarrow F' \wedge G' \\
F \vee G &\Leftrightarrow F' \vee G' \\
F \rightarrow G &\Leftrightarrow F' \rightarrow G' \\
F \leftrightarrow G &\Leftrightarrow F' \leftrightarrow G' \\
\forall x\colon F &\Leftrightarrow \forall x\colon F' \\
\exists x\colon F &\Leftrightarrow \exists x\colon F'
\end{aligned}$$

All the equivalences of propositional logic transfer to first-order logic. In addition, however, we have many equivalences related to quantified formulas, for instance:

$$\begin{aligned}
\neg\forall x\colon F &\Leftrightarrow \exists x\colon \neg F && \text{(De Morgan's Law)} \\
\neg\exists x\colon F &\Leftrightarrow \forall x\colon \neg F && \text{(De Morgan's Law)} \\
\forall x\colon (F_1 \wedge F_2) &\Leftrightarrow (\forall x\colon F_1) \wedge (\forall x\colon F_2) && \\
\exists x\colon (F_1 \vee F_2) &\Leftrightarrow (\exists x\colon F_1) \vee (\exists x\colon F_2) && \\
\forall x\colon (F_1 \vee F_2) &\Leftrightarrow F_1 \vee (\forall x\colon F_2) && \text{if } x \text{ is not free in } F_1 \\
\exists x\colon (F_1 \wedge F_2) &\Leftrightarrow F_1 \wedge (\exists x\colon F_2) && \text{if } x \text{ is not free in } F_1
\end{aligned}$$

If a domain is finite, we may replace quantified formulas by sequences of conjunctions respectively disjunctions:

- For a finite domain $D = \{v_1, \ldots, v_n\}$ we have:

$$\begin{aligned}
\forall x\colon F &\Leftrightarrow F[v_1/x] \wedge \ldots \wedge F[v_n/x] \\
\exists x\colon F &\Leftrightarrow F[v_1/x] \vee \ldots \vee F[v_n/x]
\end{aligned}$$

Below we give some examples how to apply above equivalences:

- Push negations from the outside to the inside:

$$\begin{aligned}
&\neg(\forall x\colon p(x) \rightarrow \exists y\colon q(x,y)) \\
\Leftrightarrow\ &\exists x\colon \neg(p(x) \rightarrow \exists y\colon q(x,y)) \\
\Leftrightarrow\ &\exists x\colon \neg((\neg p(x)) \vee \exists y\colon q(x,y)) \\
\Leftrightarrow\ &\exists x\colon ((\neg\neg p(x)) \wedge \neg\exists y\colon q(x,y)) \\
\Leftrightarrow\ &\exists x\colon (p(x) \wedge \neg\exists y\colon q(x,y)) \\
\Leftrightarrow\ &\exists x\colon (p(x) \wedge \forall y\colon \neg q(x,y))
\end{aligned}$$

- Reduce the scope of quantifiers:

$$\forall x, y\colon (p(x) \rightarrow q(x,y))$$

$$
\begin{aligned}
&\Leftrightarrow \forall x, y : (\neg p(x) \vee q(x, y)) \\
&\Leftrightarrow \forall x : (\neg p(x) \vee \forall y : q(x, y)) \\
&\Leftrightarrow \forall x : (p(x) \rightarrow \forall y : q(x, y))
\end{aligned}
$$

- Replace quantification in a finite domain $D = \{0, 1, 2\}$:

$$
\begin{aligned}
&\forall x : p(x) \\
&\Leftrightarrow p(0) \wedge p(1) \wedge p(2)
\end{aligned}
$$

4 Pragmatics

In this section, we turn to the *pragmatics* (practical use) of first order logic. Here we focus on the application of first-order logic to the *formal specification* of computational problems: we describe by logic formulas the assumptions on the given inputs of a computation and the guarantees for the computed outputs.

In order to adequately specify problems, we may need to write *formal definitions* that, based on some mathematical "standard models" of domains (sets of values) and operations on these domains (constants, functions, predicates), introduce new models which can be used in the formalization of the specification; in computer science we call such models "data types".

Standard Models Our specifications make use of the usual number domains:

Natural Numbers The constant $\mathbb{N}$ represents the set of all natural numbers $0, 1, 2, \ldots$ while $\mathbb{N}_n$ represents the subset of the first n natural numbers $0, 1, \ldots, n-1$. The constant $\mathbb{N}_{>0}$ represents the set of natural numbers $1, 2, \ldots$ without 0.

Integer Numbers The constant $\mathbb{Z}$ represents the set of all integer numbers $\ldots, -2, -1, 0, 1, 2, \ldots$.

Real Numbers The constant $\mathbb{R}$ represents the set of all real numbers with the subsets $\mathbb{R}_{\geq 0}$ of all non-negative real numbers and $\mathbb{R}_{>0}$ of all positive real numbers.

For all these domains we assume the usual arithmetic operations (constants, functions, and predicates). For instance $\mathbb{N}_8$ is the set of the natural numbers $0, \ldots, 7$ and $1 + 2 \cdot 3 > 6$ is true.

We also use the following *domain constructors*:

Sets The "powerset" domain $\mathcal{P}(T)$ describes the set of all sets whose elements are from set T. We assume the membership predicate $e \in S$ ("e is an element of set S") and the set builder term $\{t \mid x \in S \wedge \ldots \wedge F\}$ ("the set of all values of term t that satisfy formula F where the variables $x, \ldots$ denote all possible values of sets $S, \ldots$").

For instance $x \in \mathbb{N}_8$ means that x is one of the natural numbers $0, \ldots, 7$ while $S \in \mathcal{P}(\mathbb{N}8)$ means that S is a set of such numbers. The term $\{2 \cdot x \mid x \in \mathbb{N} \wedge x > 0\}$ denotes the set of all positive even numbers.

Products The domain $T_1 \times \ldots \times T_n$ denotes the set of all tuples $(c_1, \ldots, c_n)$ with n components $c_1, \ldots, c_n$ that are elements of sets $T_1, \ldots, T_n$, respectively. For tuple $t = (c_1, \ldots, c_n)$ and index $i = 1, \ldots, n$, the tuple selector $t.i$ denotes the component c_i.

For example, $t \in \mathbb{N}_2 \times \mathbb{Z}$ means t is a tuple with two components; its first component $t.1$ is a bit (0 or 1) and its second component $t.2$ is an integer.

Sequences The domain T^* denotes the set of all finite sequences of values from set T while T^ω denotes the set of all infinite sequences of values from set T. For a finite sequence $s \in T^*$, the term length(s) denotes the length of s. For $s \in T^*$ or $s \in T^\omega$, the sequence selector $s(i)$ (or: $s[i]$) denotes the element at position i of sequence s, a value from set T. For an infinite sequence $s \in T^\omega$, every index $i \in \mathbb{N}$ is legal ($s(0)$ denotes the first element of the

sequence). For an finite sequence $s \in T^*$, only index $i \in \mathbb{N}_{\text{length}(s)}$ is legal, i.e., i must be less than length(s).

For example, $s \in \mathbb{Z}^*$ means that s is a finite sequence of integers; if length(s) = 4, s holds the values $s(0)$, $s(1)$, $s(2)$, $s(3)$.

In the following, we discuss how on the basis fo the standard models described above, we can introduce new models (domains, functions, and predicates) by formal *definitions*.

Domain Definitions A new domain T may be introduced by a definition

$$T := t$$

of a new constant T where term t must denote a set (constructed from previously introduced sets by the application of set builders and/or domain constructors.

For example, we may define the following domains:

$$\begin{aligned}
\text{Nat} &:= \mathbb{N}_{2^{32}} \\
\text{Int} &:= \{i \mid i \in \mathbb{Z} \wedge -2^{31} \leq i \wedge i < 2^{31}\} \\
\text{IntArray} &:= \text{Int}^* \\
\text{IntStream} &:= \text{Int}^\omega \\
\text{Primes} &:= \{x \mid x \in \mathbb{N} \wedge x \geq 2 \wedge (\forall y \in \mathbb{N} : 1 < y \wedge y < x \rightarrow \neg(y|x))\}
\end{aligned}$$

Explicit Function Definitions A new function f may be introduced by explicitly defining its result value for all possible argument values. Such an *explicit function definition* has the following form:

$$\begin{aligned}
f &: T_1 \times \ldots \times T_n \rightarrow T \\
f(x_1, \ldots, x_n) &:= t_x
\end{aligned}$$

This definition consists of the following components:

- a new n-ary *function symbol* f,
- a *type signature* $T_1 \times \ldots \times T_n \rightarrow T$ with sets $T_1, \ldots, T_n, T$,
- a list of variables $x_1, \ldots, x_n$ (the *parameters*), and
- a term t_x (the *body*) whose free variables occur in $x_1, \ldots, x_n$;
- case $n = 0$: the definition of a constant $f : T, f := t$.

The body t_x may only refer to *previously* defined functions (i.e., t must not contain an application of function f itself); recursive function definitions are thus prohibited. Furthermore, the definition must obey the constraint

$$\forall x_1 \in T_1, \ldots, x_n \in T_n : t_x \in T$$

i.e., for all values of the parameters from the denoted parameter types the result must be of the denoted result type.

If all the conditions above are met, then the definition is well-formed and we know for the newly introduced function f

$$\forall x_1 \in T_1, \ldots, x_n \in T_n : f(x_1, \ldots, x_n) = t_x$$

i.e., for all all values of the parameters from the denoted parameter types the result of the function is the value of the body.

Below we give some examples of informal explicit function definitions and their formalizations in first-order logic:

- *Definition:* Let x and y be natural numbers. Then the *square sum* of x and y is the sum of the squares of x and y.

$$\begin{aligned} &\text{squaresum} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ &\text{squaresum}(x, y) := x^2 + y^2 \end{aligned}$$

- *Definition:* Let x and y be natural numbers. Then the *squared sum* of x and y is the square of z where z is the sum of x and y.

$$\begin{aligned} &\text{sumsquared} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ &\text{sumsquared}(x, y) := \textbf{let } z = x + y \textbf{ in } z^2 \end{aligned}$$

- *Definition:* Let n be a natural number. Then the *square sum set* of n is the set of the square sums of all numbers x and y from 1 to n.

$$\begin{aligned} &\text{squaresumset} : \mathbb{N} \to \mathcal{P}(\mathbb{N}) \\ &\text{squaresumset}(n) := \{\text{squaresum}(x, y) \mid x, y \in \mathbb{N} \wedge 1 \leq x \leq n \wedge 1 \leq y \leq n\} \end{aligned}$$

Predicate Definitions Also a new predicate p may be introduced by explicitly defining its truth value for all possible argument values. Such a *predicate definition* has this form:

$$\begin{aligned} &p \subseteq T_1 \times \ldots \times T_n \\ &p(x_1, \ldots, x_n) :\Leftrightarrow F_x \end{aligned}$$

This definition consists of the following components:

- a new n-ary *predicate symbol* p,
- a *type signature* $T_1 \times \ldots \times T_n$ with sets $T_1, \ldots, T_n$,

- a list of variables $x_1, \ldots, x_n$ (the *parameters*), and
- a formula F (the *body*) whose free variables occur in $x_1, \ldots, x_n$;
- case $n = 0$: the definition of a truth value constant $p :\Leftrightarrow F_x$.

The body F_x may only refer to *previously* defined functions (i.e., F must not contain an application of predicate p itself); recursive predicate definitions are thus prohibited.

If all the conditions above are met, then the definition is well-formed and we know for the newly introduced predicate p

$$\forall x_1 \in T_1, \ldots, x_n \in T_n \colon p(x_1, \ldots, x_n) \leftrightarrow F_x$$

i.e., for all all values of the parameters from the denoted parameter types the truth value of the predicate is the truth value of the body.

Below we give some examples of informal predicate definitions and their formalizations in first-order logic:

- *Definition:* Let x, y be natural numbers. Then x *divides* y (written as $x|y$) if $x \cdot z = y$ for some natural number z.

  $$| \subseteq \mathbb{N} \times \mathbb{N}$$
  $$x|y :\Leftrightarrow \exists z \in \mathbb{N} : x \cdot z = y$$

- *Definition:* Let x be a natural number. Then x *is prime* if x is at least two and the only divisors of x are one and x itself.

  $$\text{isprime} \subseteq \mathbb{N}$$
  $$\text{isprime}(x) :\Leftrightarrow x \geq 2 \wedge \forall y \in \mathbb{N} : y|x \rightarrow y = 1 \vee y = x$$

- *Definition:* Let p, n be a natural numbers. Then p is a *prime factor* of n, if p is prime and divides n.

  $$\text{isprimefactor} \subseteq \mathbb{N} \times \mathbb{N}$$
  $$\text{isprimefactor}(p, n) :\Leftrightarrow \text{isprime}(p) \wedge p|n$$

Implicit Function Definitions A new function may be also introduced by giving a condition on its result value. Such an *implicit function definition* has the following form:

$$f \colon T_1 \times \ldots \times T_n \rightarrow T$$
$$f(x_1, \ldots, x_n) := \textbf{such } y \colon F_{x,y} \text{ (or: } \textbf{the } y \colon F_{x,y})$$

This definition consists of the following components:

- a new n-ary *function constant* f,

- a *type signature* $T_1 \times \ldots \times T_n \to T$ with sets $T_1, \ldots, T_n, T$,
- a list of variables $x_1, \ldots, x_n$ (the *parameters*),
- a variable y (the *result variable*),
- a formula $F_{x,y}$ (the *result condition*) whose free variables occur in $x_1, \ldots, x_n, y$.

Again, the body $F_{x,y}$ may only refer to *previously* defined functions.

If all the conditions above are met, then the definition is well-formed and we know for the newly introduced function f

$$\begin{aligned}&\forall x_1 \in T_1, \ldots, x_n \in T_n : \\ &\quad (\exists y \in T : F_{x,y}) \to (\exists y \in T : F_{x,y} \wedge y = f(x_1, \ldots, x_n))\end{aligned}$$

i.e., if there is some value that satisfies the result condition, the function result is one such value (otherwise, it is arbitrary).

If we write in an implicit function definition the keyword **the** (instead of the keyword **such**), we claim that a value satisfying the body formula always exists and that it is uniquely determined by the formula (no two different values satisfy the formula).

Below we give some examples of informal implicit function definitions and their formalizations in first-order logic:

- *Definition:* Let x be a real number. *A root* of x is a real number y such that the square of y is x (if such a y exists).

  $$\begin{aligned}&\text{aRoot} : \mathbb{R} \to \mathbb{R} \\ &\text{aRoot}(x) := \textbf{such } y : y^2 = x\end{aligned}$$

  Note that for negative values of x no result with the stated property exists; if x is positive, the result is not uniquely determined (for $x = 4$ both $y = 2$ and $y = -2$ are legitimate).

- *Definition:* Let x be a non-negative real number. *The root* of x is that real number y such that the square of y is x and $y \geq 0$.

  $$\begin{aligned}&\text{theRoot} : \mathbb{R}_{\geq 0} \to \mathbb{R} \\ &\text{theRoot}(x) := \textbf{the } y : y^2 = x \wedge y \geq 0\end{aligned}$$

  For all non-negative values of x a result y with the stated property exists and is uniquely determined (for $x = 4$ only $y = 2$ is legitimate).

- *Definition:* Let $m, n \in \mathbb{N}$ with n positive. Then the *(truncated) quotient* $q \in \mathbb{N}$ of m and n is such that $m = n \cdot q + r$ for some $r \in \mathbb{N}$ with $r < n$.

  $$\begin{aligned}&\text{quotient} : \mathbb{N} \times \mathbb{N}_{>0} \to \mathbb{N} \\ &\text{quotient}(m, n) := \textbf{the } q : \exists r \in \mathbb{N} : m = n \cdot q + r \wedge r < n\end{aligned}$$

For all arguments m and n with $n > 0$, a result with the stated property exists and is uniquely determined.

- *Definition:* Let x, y be positive natural numbers. The *greatest common divisor* of x and y is the greatest such number that divides both x and y.

$$\begin{aligned}&\gcd\colon \mathbb{N}_{>0} \times \mathbb{N}_{>0} \to \mathbb{N}_{>0}\\&\gcd(x, y) := \textbf{the } z\colon z|x \land z|y \land \forall z' \in \mathbb{N}_{>0}\colon z'|x \land z'|y \to z' \leq z\end{aligned}$$

For all arguments $x > 0$ and $y > 0$, a result z with the stated property exists and is uniquely determined.

Often there is a choice whether a notion is formalized as a function or a predicate. For instance, the notion of a "prime factor" may be formalized in the following ways:

- A *predicate*:

$$\begin{aligned}&\text{isprimefactor} \subseteq \mathbb{N} \times \mathbb{N}\\&\text{isprimefactor}(p, n) :\Leftrightarrow \text{isprime}(p) \land p|n\end{aligned}$$

This predicate states that p is a prime factor of n.

- An *implicitly defined function*:

$$\begin{aligned}&\text{someprimefactor}\colon \mathbb{N} \to \mathbb{N}\\&\text{someprimefactor}(n) := \textbf{such } p\colon \text{isprime}(p) \land p|n\end{aligned}$$

This function returns some prime factor p of n.

- An *explicitly defined function* whose result is a set:

$$\begin{aligned}&\text{allprimefactors}\colon \mathbb{N} \to \mathcal{P}(\mathbb{N})\\&\text{allprimefactors}(n) := \{p \in \mathbb{N} \mid \text{isprime}(p) \land p|n\}\end{aligned}$$

This function returns the set of all prime factors of n.

Note that all definitions are based on the same formula ($\text{isprime}(p) \land p|n$); the preferred kind of definition is a matter of purpose and taste.

Since we are now able to define the domains of computational problems, we now turn to one of the main applications of first-order logic in computer science, the formal specification of such problems.

Specifying Problems The specification of a computational problem, short *problem specification*, has the following form:

Input: $x_1 \in T_1, \ldots, x_n \in T_n$ **where** I_x
Output: $y_1 \in U_1, \ldots, y_m \in U_m$ **where** $O_{x,y}$

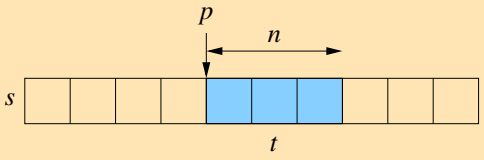
This problem specification consists of

- a list of *input variables* $x_1, \ldots, x_n$ with types $T_1, \ldots, T_n$,
- a formula I_x (the *input condition* or *precondition*) whose free variables occur in $x_1, \ldots, x_n$,
- a list of *output variables* $y_1, \ldots, y_m$ with types $U_1, \ldots, U_m$, and
- a formula $O_{x,y}$ (the *output condition* or *postcondition*) whose free variables occur in $x_1, \ldots, x_n, y_1, \ldots, y_m$.

The specification is expressed with the help of functions and predicates that have been previously defined to describe the problem domain.

As an example take the following problem specification.

Problem: extract from a finite sequence s of natural numbers a subsequence t of length n starting at position p.



Example: $s = [2,3,5,7,5,11], p = 2, n = 3 \rightsquigarrow t = [5,7,5]$

Input: $s \in \mathbb{N}^*, n \in \mathbb{N}, p \in \mathbb{N}$ **where**
$n + p \leq \text{length}(s)$ *(subsequence is in range of array)*
Output: $t \in \mathbb{N}^*$ **where**
$\text{length}(t) = n \wedge$ *(length of result sequence)*
$\forall i \in \mathbb{N}_n : t(i) = s(i + p)$ *(content of result sequence)*

In this specification, the input condition has as free variables only the input variables n, p, s; the output condition has as free variables n, p, s and the output variable t; the variable i is bound by the universal quantifier. The input condition states that the subsequence defined by n and p does not exceed the range of s; the output condition determines the appropriate length and content of the result sequence.

The Adequacy of Specifications Given a specification with input condition I_x and output condition $O_{x,y}$, we may validate the adequacy of the specification by asking the following questions:

- Is the precondition satisfiable?

$$\exists x : I_x$$

Otherwise no input is allowed.

- Is the precondition not trivial?

$$\exists x : \neg I_x$$

Otherwise every input is allowed; what is then the purpose of writing the precondition?

- Is the postcondition always satisfiable?

$$\forall x : (I_x \to \exists y : O_{x,y})$$

Otherwise no implementation is legal.

- Is the postcondition not always trivial?

$$\exists x, y : (I_x \wedge \neg O_{x,y})$$

Otherwise every implementation of the specification is legal.

- Is the result unique?

$$\forall x, y_1, y_2 : (I_x \wedge O_{x,y}[y_1/y] \wedge O_{x,y}[y_2/y] \to y_1 = y_2)$$

Whether this is required, depends on our expectations.

As an example, consider the following problem of "integer division" that we gradually approach by a sequence of more and more adequate problem specifications.

Problem: given natural numbers m and n, compute the truncated quotient q and remainder r of dividing m by n.

Input: $m \in \mathbb{N}, n \in \mathbb{N}$
Output: $q \in \mathbb{N}, r \in \mathbb{N}$ **where** $m = n \cdot q + r$

- The postcondition is always satisfiable but not trivial.
  - For $m = 13, n = 5$, e.g. $q = 2, r = 3$ is legal but $q = 2, r = 4$ is not.
- But the result is not unique.
  - For $m = 13, n = 5$, both $q = 2, r = 3$ and $q = 1, r = 8$ are legal.

Input: $m \in \mathbb{N}, n \in \mathbb{N}$
Output: $q \in \mathbb{N}, r \in \mathbb{N}$ **where** $m = n \cdot q + r \wedge r < n$

- Now the postcondition is not always satisfiable.
  - For $m = 13, n = 0$, no output is legal.

Input: $m \in \mathbb{N}, n \in \mathbb{N}$ **where** $n \neq 0$
Output: $q \in \mathbb{N}, r \in \mathbb{N}$ **where** $m = n \cdot q + r \wedge r < n$

- The precondition is not trivial but satisfiable.
  - $m = 13, n = 0$ is not legal but $m = 13, n = 5$ is.
- The postcondition is always satisfiable and result is unique.
  - For $m = 13, n = 5$, only $q = 2, r = 3$ is legal.

Example Specifications We continue with several examples of problem specifications.

Problem (Linear Search): given a finite integer sequence a and an integer x, determine the smallest position p at which x occurs in a ($p = -1$, if x does not occur in a).

Example: $a = [2, 3, 5, 7, 5, 11], x = 5 \rightsquigarrow p = 2$

Input: $a \in \mathbb{Z}^*, x \in \mathbb{Z}$
Output: $p \in \mathbb{N} \cup \{-1\}$ **where**
let $n = \text{length}(a)$ **in**
if $\exists p \in \mathbb{N}_n : a(p) = x$ *(x occurs in a)*
then $p \in \mathbb{N}_n \wedge a(p) = x \wedge$ *(p is the index of some occurrence of x)*
$(\forall q \in \mathbb{N}_n : a(q) = x \rightarrow p \leq q)$ *(p is the smallest such index)*
else $p = -1$

In this specification, all inputs are legal; the result always exists (either -1 or the index of an occurrence of x in a) and is uniquely determined (if multiple such indices exist, the result is the smallest such index).

Problem (Binary Search): given a finite integer sequence a that is sorted in ascending order and an integer x, determine some position p at which x occurs in a ($p = -1$, if x does not occur in a).

Example: $a = [2, 3, 5, 5, 5, 7, 11], x = 5 \rightsquigarrow p \in \{2, 3, 4\}$

Input: $a \in \mathbb{Z}^*, x \in \mathbb{Z}$ **where**
let $n = \text{length}(a)$ **in**
$\forall k \in \mathbb{N}_{n-1} : a(k) \leq a(k+1)$ *(a is sorted)*
Output: $p \in \mathbb{N} \cup \{-1\}$ **where**
let $n = \text{length}(a)$ **in**

if $\exists p \in \mathbb{N}_n : a(p) = x$ *(x occurs in a)*
then $p \in \mathbb{N}_n \wedge a(p) = x$ *(p is the index of some occurrence of x)*
else $p = -1$

In this specification, not all inputs are legal; for every legal input, the result exists but is not uniquely determined (the result may be the index of any occurrence of x). Since the input sequence is sorted, the problem may be solved by the "binary search" algorithm.

Problem (Sorting): given a finite integer sequence a, determine that permutation b of a that is sorted in ascending order.

Example: $a = [5,3,7,2,3] \rightsquigarrow b = [2,3,3,5,7]$

Input: $a \in \mathbb{Z}^*$
Output: $b \in \mathbb{Z}^*$ **where**
let $n = \text{length}(a)$ **in**
$\text{length}(b) = n \wedge$
$(\forall k \in \mathbb{N}_{n-1} : b(k) \leq b(k+1)) \wedge$ *(b is sorted)*
$\exists p \in \mathbb{N}_n{}^* :$ *(b is a permutation of a)*
$(\forall k1 \in \mathbb{N}_n, k2 \in \mathbb{N}_n : k1 \neq k2 \rightarrow p(k1) \neq p(k2)) \wedge$
$(\forall k \in \mathbb{N}_n : a(k) = b(p(k)))$

In this specification, every input sequence a is legal; the output sequence b, a sorted permutation of a, always exists and is uniquely determined. The fact that b is a permutation of a is established by the existence of a sequence p of n array indices (according to the type of p) which are all different (according to the first subcondition) such that the value at index k of p states where the value at index k in sequence a occurs in sequence b (according to the second subcondition). Since p maps every index k of a to at least one index of b, but not to more than one such index, the elements of a and b are in one-to-one correspondence; thus b is indeed a permutation of a.

Implementing Specifications The purpose of the specification of a computational problem is to describe the goal of an implementation that solves this problem; the specification describes *what* problem is to be solved while the implementation describes *how* to solve it.

From the logical perspective, the *implementation* of a specification with input condition I_x and output condition $O_{x,y}$ is the definition of a function

$$f : T_1 \times \ldots \times T_n \rightarrow U_1 \times \ldots \times U_m$$

with the following property:

$$\forall x_1 \in T_1, \ldots, x_n \in T_n : I_x \rightarrow \textbf{let } (y_1, \ldots, y_m) = f(x_1, \ldots, x_n) \textbf{ in } O_{x,y}$$

Therefore, for all arguments that satisfies the input condition, the function must return a result that satisfies the output condition. Actually the specification itself already defines such a function implicitly:

$$f(x_1, \ldots, x_n) := \textbf{such } y_1, \ldots, y_m : I_x \rightarrow O_{x,y}$$

However, the specification is adequately implemented only by an explicitly defined function:

$$f(x_1, \ldots, x_n) := t_x$$

In terms of computer science, this explicitly defined function is a computer program.

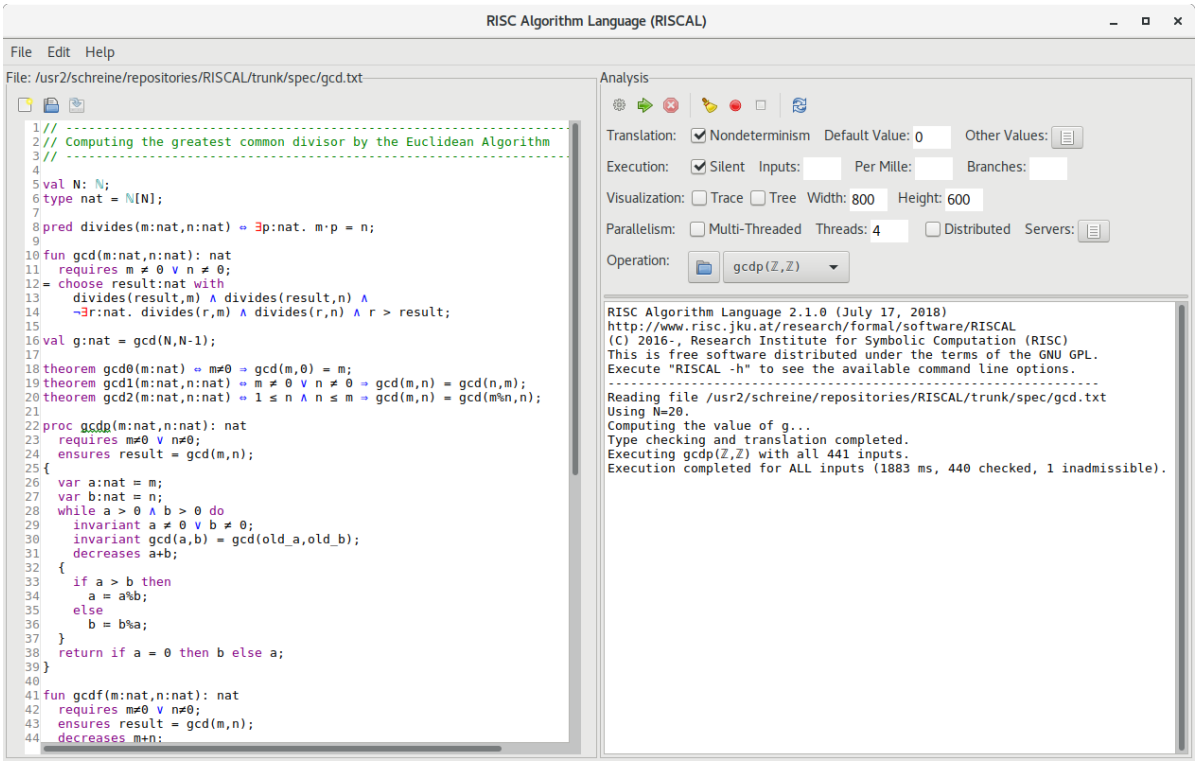
The goal of computer science is to adequately specify problems, to implement these specifications by computer programs, and ultimately to verify the correctness of the implementations. From formal specifications in first-order logic, the computer-supported verification of programs with respect to specifications (by interactive theorem provers or automatic model checkers) becomes possible; this is the topic of other courses in the computer science curriculum.

A The RISCAL Software

RISCAL (RISC Algorithm Language) is a language and associated software system for the formal specifications of mathematical theories and algorithms on the basis of first-order logic. The software is freely available from the following URL:

https://www.risc.jku.at/research/formal/software/RISCAL

A screenshot of the software's graphical user interface is given below:



The RISCAL language and software is extensively documented in the accompanying (online) manual; below we only give some introductory examples.

RISCAL is based on a type version of first-order logic. For instance, the definition

```
type nat = ℕ[20];
```

introduces a type `nat` of the first 21 natural numbers $0, \ldots, 20$. The definition

```
pred divides(m:nat,n:nat) ⇔ ∃p:nat. m·p = n;
```

introduces the predicate "m divides n" on that type; the quantified formula $(\exists v \in T\colon F)$ is written in RISCAL as `(∃v:T. F)` (note the dot '.'). Likewise the definition

```
pred isgcd(g:nat,m:nat,n:nat) ⇔
  divides(g,m) ∧ divides(g,n) ∧
  ∀g0:nat. divides(g0,m) ∧ divides(g0,n) ⇒ g0 ≤ g;
```

defines the predicate "g is the greatest common divisor of m and n". Implication and equivalence are written in RISCAL with double arrows as $F \Rightarrow G$ and $F \Leftrightarrow G$, respectively.

Based on this predicate, the implicit definition

```
fun gcd(m:nat,n:nat): nat
  requires m ≠ 0 ∨ n ≠ 0;
= choose g:nat with isgcd(g,m,n);
```

introduces a function $\gcd(m,n)$ that returns, if not both $m = 0$ and $n = 0$, a value g that satisfies this predicate.

While above definitions use Unicode symbols such as $\mathbb{N}$ or $\exists$, these symbols may be written also as ASCII strings according to the following table:

ASCII String	Unicode Character	ASCII String	Unicode Character
`Int`	$\mathbb{Z}$	`~=`	$\neq$
`Nat`	$\mathbb{N}$	`<=`	$\leq$
`:=`	$:=$	`>=`	$\geq$
`true`	$\top$	`*`	$\cdot$
`false`	$\bot$	`times`	$\times$
`~`	$\neg$	`{}`	$\emptyset$
`/\`	$\wedge$	`intersect`	$\cap$
`\/`	$\vee$	`union`	$\cup$
`=>`	$\Rightarrow$	`Intersect`	$\bigcap$
`<=>`	$\Leftrightarrow$	`Union`	$\bigcup$
`forall`	$\forall$	`isin`	$\in$
`exists`	$\exists$	`subseteq`	$\subseteq$
`sum`	$\sum$	`<<`	$\langle$
`product`	$\prod$	`>>`	$\rangle$

While these strings are also legal inputs, one may press after such a string the key combination `Ctrl+#` (respectively: `Strg+#`) to replace the string by the corresponding symbol.

RISCAL may execute definitions for all possible values of their parameters from the denoted domains (which are always finite). For instance, if we select in the menu "Operation" of the user interface the operation `gcd` and press the button "Start Execution", the system depicts the following output:

```
Executing gcd(ℤ,ℤ) with all 441 inputs.
Ignoring inadmissible inputs...
Run 1 of deterministic function gcd(1,0):
Result (1 ms): 1
...
Run 438 of deterministic function gcd(18,20):
Result (1 ms): 2
Run 439 of deterministic function gcd(19,20):
Result (0 ms): 1
Run 440 of deterministic function gcd(20,20):
Result (1 ms): 20
Execution completed for ALL inputs (2892 ms, 440 checked, 1 inadmissible).
```

For all legitimate values for m and n ($440 = 21 \cdot 21 - 1$), the system determines the result of the execution of gcd(m, n) by enumerating all possible values of g and choosing *some* for which the formula in the implicit function definition is true. If the option "Nondeterministic" is selected, the system actually determines *all* such values (in the case of the greatest common divisor, there is only one).

Furthermore, we may define theorems as predicates that are expected to be true for all possible values of their parameters. For instance, we may define the (correct) theorem

```
theorem gcd2(m:nat,n:nat) ⇔ 1 ≤ n ∧ n ≤ m ⇒ gcd(m,n) = gcd(m%n,n);
```

and execute it by selecting operation gcd2 in the menu and setting the "Execution" option "Silent" to suppress the printing of the truth values (which are presumably always "true"). The execution produces then only the following output:

```
Executing gcd2(ℤ,ℤ) with all 441 inputs.
Execution completed for ALL inputs (274 ms, 441 checked, 0 inadmissible).
```

Since no error was reported, the theorem indeed is true.

However, if we define the "wrong" theorem

```
theorem gcd3(m:nat,n:nat) ⇔ 1 ≤ n ∧ 1 ≤ m ⇒ gcd(m,n) = gcd(m/n,n);
```

the execution reports the following result:

```
Executing gcd3(ℤ,ℤ) with all 441 inputs.
ERROR in execution of gcd3(1,2): evaluation of
  gcd3
at line 42 in file gcd.txt:
  theorem is not true
ERROR encountered in execution.
```

This output reports that for $m = 1$ and $n = 2$ the supposed theorem is actually false.

With RISCAL, we are therefore able to check the validity of formulas in first-order logic over finite domains. Many other uses of the software are documented in the manual.