Bird's-eye view of logic programming

Bird's-eye view of Logic Programming

Rik van Geldrop

Contents

1 Preface

This note gives a quick overview of logic programming. In general, no specific implementation is intended, but in concrete examples we use SWI-Prolog for which free software is available from http://www.swi-prolog.org .

The note is primarily aimed at readers who are familiar with 1^{st}-order predicate logic. Because a main part of the note is concerned with the design of logical programs some additional knowledge of functional programming and/or database query design will be helpful.

The note is organized as follows. In section 2 we introduce the logical paradigm by way of an example in predicate logic and we show how one can "program" with a specific form of predicates. In section 3 we give an overview of a logical system and discuss the several tasks of such a system in order to automate the programming style mentioned in section 2. The syntax and the semantics of logic programs is subject of section 4. In a subsection we will explain how any 1^{st}-order predicate formula can be converted into the logical syntax. Section 5 contains some details about the Prolog implementation of logical programs. In our opinion these details are sufficient to make a start with any Prolog-like system. Section 6 is concerned with the design of logical programs. Predicate calculus is an obvious construction means, but functional programming and database query design may be fruitfully exploited too. For each of these formal ways we give some examples. A larger and more generic example of program design is given in section 7. There it is shown how a problem that is modelled by an attributed context free grammar can be written as a logical program in a systematic way. We conclude this note with some words on unification, a process that is at the heart of each logical system.

Acknowledgements
I would like to thank Jaap van der Woude, Wim Feijen and Eugen Schindler for their constructive criticism on earlier versions of this note.

Rik van Geldrop

Eindhoven University of Technology
March 2004

2 The logical game

The primary goal of Logic Programming is to create a formal universe that enables us to answer certain questions about a "real" world. To that end this real world is modelled through a number of postulates stating facts of that world. The postulates take the form of logical expressions and may contain information about constants and about relations between variables. Thus the system of postulates forms a mathematical abstraction of certain facts in a real world, and it acts as a starting point for deducing new facts - "theorems"- of that world through logical reasoning.

A logical system -which has to perform the deduction in an automated way- puts its own demands on the shape of the postulates, the number of deduction rules and the shape of the theorems that can be deduced.

In the kind of logical systems that we are interested in, **postulates** take the shape of

$$[\langle \bigwedge i :: p_i \rangle \Rightarrow q] \tag{1}$$

where $\bigwedge$ stands for a -usually- finite conjunction and the square brackets are a shorthand for universal quantification over the anonymous variables.
A postulate of shape (1) is called "clause".

Only one **deduction rule** is allowed in deriving new theorems, i.e.

$$[P \Rightarrow Q] \wedge [Q \Rightarrow R] \Rightarrow [P \Rightarrow R] \tag{2}$$

called "resolution[1]".

A **theorem** is a clause too, as might be clear by the end of this section. A typical question -"query"- to be posed is whether or not the closed formula

G: $\quad \langle \exists x :: g.x \rangle$

is a theorem in this system of postulates. Here g may be one of the predicates p_i in one of the antecedents, or one of the consequents q in the system. Dummy x is a specified list of anonymous variables.

The logical game that will be played with this limited set of tools is a constructive one, i.e. an answer to the query (either "No" or "Yes, $x = ..$ is a witness") is constructed during the deduction process. Below we will illustrate the game by an example, but first we take a closer look at the resolution rule, the only means to draw conclusions w.r.t. the validity of G in the context of a given set of postulates. We see that the resolution rule may conclude to implications only. To enable a conclusion like G, G must be transformed into implicative form in, for instance, one of the following two ways

[1]The formulation of the resolution rule mentioned here is adapted to our forthcoming use. The proper formulation is $[P \vee Q] \wedge [\neg Q \vee R] \Rightarrow [P \vee R]$

(i) $true \Rightarrow G$

(ii) $(G \Rightarrow false) \Rightarrow false$

If we choose strategy (ii) we observe for its antecedent

$$\begin{array}{ll} & \langle \exists x :: g.x \rangle \Rightarrow false \\ \equiv & \{ \text{ predicate calculus } \} \\ & \langle \forall x :: g.x \Rightarrow false \rangle \\ \equiv & \{ \text{ shorthand } \} \\ & [g.x \Rightarrow false] \end{array}$$

i.e. ¬G is a clause just as the postulates. Whatever strategy is chosen, a query can be "resolved" constructively.

Example We consider the following system of postulates in which $a, b, ..$ are constants and $x, y, ..$ are variables:

(p0) $[q.(x, y) \wedge r.y \Rightarrow p.x]$

(p1) $[s.y \Rightarrow q.(c, y)]$

(p2) $[true \Rightarrow q.(a, b)]$

(p3) $[true \Rightarrow s.c]$

(p4) $[true \Rightarrow r.c]$

We are heading for resolving the query

$$\langle \exists x :: p.x \rangle$$

In order to use strategy (ii), we rewrite the query to the equivalent

$$[p.x \Rightarrow false] \Rightarrow false$$

We observe

for strategy (i)

$$\begin{array}{ll} & \langle \exists x :: p.x \rangle \\ \Leftarrow & \{ \text{ (p0) and (2) } \} \\ & \langle \exists x, y :: q.(x, y) \wedge r.y \rangle \\ \Leftarrow & \{ \text{ (p1), } \bullet x = c; \text{ (2) } \} \\ & \langle \exists y :: s.y \wedge r.y \rangle \\ \Leftarrow & \{ \text{ (p3), } \bullet y = c; \text{ (2) } \} \\ & true \wedge r.c \end{array}$$

for strategy (ii)

$$\begin{array}{ll} & [p.x \Rightarrow false] \\ \Rightarrow & \{ \text{ (p0) and (2) } \} \\ & [q.(x, y) \wedge r.y \Rightarrow false] \\ \Rightarrow & \{ \text{ (p1), } \bullet x = c; \text{ (2) } \} \\ & [s.y \wedge r.y \Rightarrow false] \\ \Rightarrow & \{ \text{ (p3), } \bullet y = c; \text{ (2) } \} \\ & true \wedge r.c \Rightarrow false \end{array}$$

$\Leftarrow$ { (p4); (2) }

$true \wedge true$

$\equiv$ { pred calc }

$true$

$\Rightarrow$ { (p4); (2) }

$true \Rightarrow false$

$\equiv$ { pred calc }

$false$

□

Remarks.

• In both derivations a witness to the query, viz. $x = c$, is constructed. Note that the intermediate $y = c$ is not part of the witness.

• The two derivations are mathematically the same. A human being may prefer strategy (i), our logical system takes the strategy (ii) approach because the starting point, i.e. $\neg G$, is a clause and so are all the derived expressions.

• In the second step of the above deductions we could have appealed to (p2) as well, with $x, y = a, b$, which would result in $r.b$ and $r.b \Rightarrow false$ respectively. Since our postulates contain no such information, the query cannot be resolved in this way.

• A similar derivation may be given for a query with "subqueries", viz. $\langle \exists x :: p.x \wedge s.x \rangle$

□

We conclude this section with some more terminology used by logical programmers:
The "goal" of a derivation is the clause $\neg G$. The "empty clause" -the clause *true* $\Rightarrow$ *false*- is the final clause in a successful derivation. A successful derivation is a "proof of the goal". (Note that there may exist several proofs for a given goal, even several proofs for a given goal with the same witness. See the first remark above.)
The terminology w.r.t. an application of the resolution rule will be introduced on the basis of an example. Consider the second step of the above derivation:
The resolution rule is applied to the (derived) goal $[q.(x, y) \wedge r.y \Rightarrow false]$ and (p1), i.e. $[s.y \Rightarrow q.(c, y)]$: "the two clauses are resolved against each other". In this resolution step predicate q is eliminated: "the clauses are resolved upon q". By the substitution $\theta = [x := c, y := y]$ the two occurrences of q are made textually equal: "the two occurrences of q are unified". The result -"resolvent"- of this resolution step is $[s.y \wedge r.y \Rightarrow false]$.

3 Overview of logical systems

Logical systems are designed to generate proofs like the above in a mechanical way.
Let us summarize the previous proof for a global view of a logical system and its behaviour.
We were given:

1. a set S of clauses each of which has exactly one predicate in its consequent, a "definite" clause.

2. a goal G of shape $\langle \exists x :: g.x \rangle$. Note that the negation of a goal is a clause.

The truth of G, being equivalent to the falsity of $\neg G$, is proven by

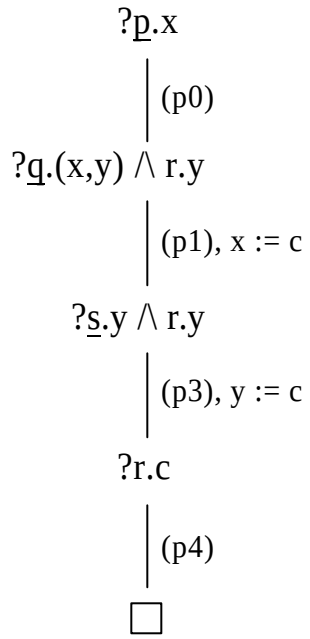
- adding the clause $\neg G$ to the "model"

- showing the inconsistency of the augmented "model"

In each derivation step the goal is resolved against one of the clauses of S selected by us, resolved upon a subgoal again selected by us, and replaced by the resolvent. By making appropriate choices we finally derived the empty clause while on the fly a counterexample to $\neg G$ was constructed.

This approach of solving queries is known as "SLD-resolution", a special kind of "resolution refutation". The term resolution refutation refers to the strategy to strive for falsification of the negated goal. The abbreviation SLD stands for:
- S : selection, viz. a selection rule determines the predicate upon which will be resolved.
- L : linear, viz. in each resolution step a subgoal of the former step is involved.
- D : definite, viz. only definite clauses participate in the resolution process.
SLD-resolution is implemented in the so-called "proof procedure" of many logical systems. The proof procedure acts as an automated theorem prover; in logic programming this is expressed as: the proof procedure "executes program S on input G".

Remark. The derivation that we gave in the previous section is usually depicted as an SLD-tree in logic programming:



To see that an SLD-tree shows a same derivation as the one in the previous section:
- read "?p.x" as $p.x \Rightarrow false$,
  where the prefix operator "?" has a lower priority than conjunction.
- an underlining indicates the predicate upon which will be resolved.
- the symbol "□" denotes the empty clause, i.e. a successful derivation.
□

The proof procedure takes care of:
1. Selection of the ingredients of a resolution step, viz. a program clause and a subgoal. (In SLD-resolution the goal and the program clause are resolved upon the subgoal.)
2. Execution of the resolution step, viz. unification of the two occurrences of the subgoal and computation of the resolvent.
3. Handling situations in which no resolution step is possible.

Below we will give some insight in each of these subtasks and in the consequences of the design decisions that are taken in Prolog-like implementations. Since the unification algorithm is at the heart of the proof procedure we start with subtask 2.

3.1 Subtask 2: Unification

Whether two given postulates can be resolved against each other and which substitution of the variables is appropriate is seen by humans in the blink of an eye. Unification formalizes this insight.

Consider, for instance, the following postulates with constants a, b and variables x, v, w

$[p.x \Rightarrow q.(x, b)]$
$[q.(a, v) \wedge s.(v, w) \Rightarrow t.(a, w)]$

the two occurrences of q are made textually equal by a suitable substitution/unifier θ, where $\theta = [x := a, v := b]$, and the resolvent is $[(p.x)\theta \wedge (s.(v,w))\theta \Rightarrow (t.(a,w))\theta]$, viz. $[p.a \wedge s.(b,w) \Rightarrow t.(a,w)]$.

When we were given the postulates

$[p.x \Rightarrow q.(b,x)]$
$[q.(a,v) \wedge s.(v,w) \Rightarrow t.(a,w)]$

the two occurrences of q could not be made textually equal because of the different constants a and b. Consequently these postulates cannot be resolved against each other.

Applying a substitution to an expression can be done systematically, an implementation is straightforward. For finding a suitable substitution the literature gives several systematic ways. In the appendix we outline Robinson's approach and define his version of a unification algorithm.

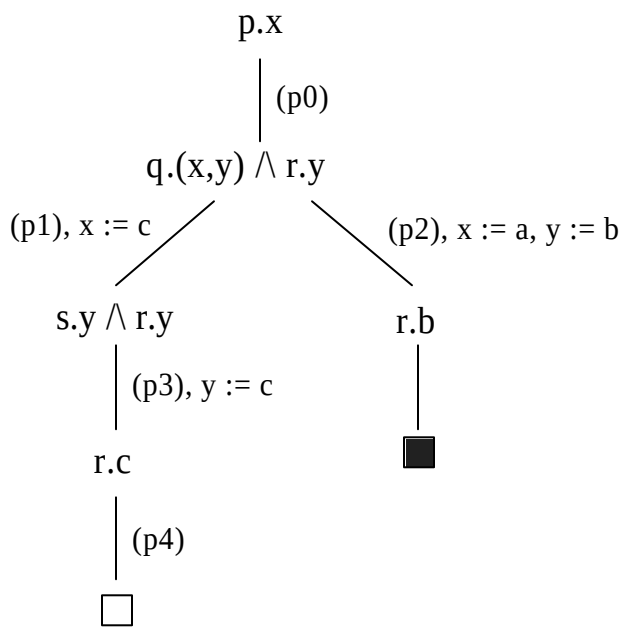
For further use we draw attention to the fact that, in a resolution step, a unifier is substituted in both clauses which are resolved against each other. As we will see in section [6.2.4] it is this "non-directionality" (or "bi-directionality") of substitutions that gives a logic program its expressive power.

Warning Because unification is a prominent operation in logic programming, many logical systems claim the "=" sign for it. "Equal" in logic programming means: textually equal, which may be very confusing. E.g. the expression `2 * 2 = 4` evaluates to *true* in non-logical systems while in logical systems it evaluates to *false* (because the expressions in the LHS and the RHS cannot be unified). Of course logical systems support other equalities than unification be it in sugared equality signs.
□

3.2 Subtask 1: Selection

In order to prepare a resolution step two choices have to be made:
1. which of the subgoals will be replaced in the next resolution step
2. which of the clauses of S will be used in the next resolution step
The first choice is irrelevant: in Lloyd [9] it is proven that independent of the selected subgoal a successful proof of G will be found when this is possible in the context of S. The second choice is important: an arbitrary selection of an S-clause may lead to an unprovable resolvent while another selection may lead to a provable one. To circumvent this problem logical systems construct all possible ways to find proofs for a goal.

It is common to represent this construction by means of search trees. As an illustration we give a search tree for the example from the previous section



where the black box denotes failure.
□

In Prolog-like systems the above choices 1 and 2 are filled in as follows:

for choice 1: always take the leftmost subgoal.
for choice 2: in order to enable the construction of all possible proofs:
- (i) the program clauses are sequentially searched (in textual order) for their ability to participate in the resolution step.
- (ii) a "backtracking" mechanism is added, i.e. the construction falls back to its previous point and will be continued via a sequential search in textual lower clauses.

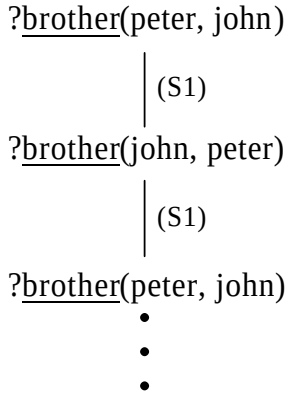
The back-part of backtracking is activated when
- no program clause is found which can participate in the resolution step
- the empty clause has been derived

As a consequence, the proof procedure terminates when all possible paths in the search tree are explored.

At this point a Prolog programmer has to be warned: In Prolog the ordering of clauses is important. Consider for instance the following program

(S1) $brother(x, y) \Rightarrow brother(y, x)$
(S2) $true \Rightarrow brother(john, peter).$

The query $brother(peter, john)$ will never be answered by the system. To see this we draw the SLD-tree



which means that Prolog will not explore all possible ways to find a proof.
Change the order of (S1) and (S2) and the system confirms the query in two steps.
For reasons of efficiency and termination, in Prolog programs facts -like (S2)- are textually higher than rules about them -like (S1).

3.3 Subtask 3: No resolution step possible

Now that we have seen how resolution can be done in a mechanical way it is time to envisage the situation in which no resolution step is possible. If all possible ways to find a proof are explored (e.g. via backtracking) we may conclude that alle possible solutions to a query are constructed. It might be the case that the set of solutions is empty. Clearly, in such case, program S contains no evidence about the query's validity. This raises the question whether this means that in the real world which is formalized by S, the query holds or not. There are two approaches to this question

the Open World Assumption This says that the query is neither *true* nor $false$. In this approach a query might have three possible truth values: *true*, $false$ and *unknown*. If a query evaluates to *unknown* the system might take special actions, e.g. consulting an alternative source of information.

the Closed World Assumption This says that the query is $false$. In this approach the real world is restricted to the information that is contained in S. Everything outside this information does not hold.

Prolog-like systems take the Closed World Assumption approach. Consequently they adopt an implicit implementation of negation: "negation by failure", viz. If query G fails to hold in the context of S, then ¬G holds.

Warning Prolog-like systems support a standard predicate `not` which is defined in terms of failure: `not(`$p.x$`)` holds whenever goal $p.x$ fails, i.e. `not(`$p.x$`)` means $\neg\langle\exists x :: p.x\rangle$ which is equivalent to $\langle\forall x :: \neg p.x\rangle$. Although the meaning of `not` differs from the negation operation in predicate calculus it still is a useful means in achieving negations, see section 5.

□

Citing [10], another consequence of taking the Closed World Assumption is that a predicate p is determined as the least solution to the equation $p :: p \Leftarrow q$ for some suitable q, which means that $p \equiv q$. That is the reason why in logical programming predicates are defined by one half of an equivalence only.

4 Syntax and semantics of logic programs

In Logic Programming the knowledge about a problem is expressed by a set of predicates of a particular, restricted shape -called Horn clauses. But this restriction on the shape does not impair its expressive power: quoting Kowalski [8]

> "any problem which can be expressed in logic can be re-expressed by means of Horn clauses."

Horn clauses are defined by the following syntax in a BNF-like notation.
Suppose we have mutually disjoint sets of constants, variables, function symbols and predicate symbols . Each function symbol and each predicate symbol has its own arity. Then clauses are generated according to the following syntax.

term	:=	*variable* | *constant* | $f(t_1, ..., t_k)$ where f is a k-ary function symbol t_i, $1 \leq i \leq k$, is a term
atom, or *atomic formula*	:=	$p(t_1, ..., t_k)$ where p is a k-ary predicate symbol, and t_i, $1 \leq i \leq k$ is a term
clause	:=	$\psi \Leftarrow \varphi$ where ψ is a finite disjunction of atoms φ is a finite conjunction of atoms
Horn clause	:=	clause with at most one atom in its consequent

In order to formulate a problem in "clausal form" we have to understand the meaning of a set of clauses.

The meaning of a set of clauses is the conjunction of the meaning of the individual clauses. Hence such a set corresponds to a logical formula in conjunctive normal form, i.e. a conjunction of a collection of formulae each of which is the disjunction of positive and negative atoms (when eliminating the follows-from symbols).

The meaning of

- a constant	is a named individual in the problem domain
- a variable	is an arbitrary individual
- a clause	is a universally valid relationship between predicates

Example. Consider the set of clause (p0)..(p4) with constants a, b, c, variables x, y and predicate symbols p, q, r, s:

$$\begin{array}{lllll}
(\text{p0}) & p(x) & \Leftarrow & q(x,y) \wedge r(y) \\
(\text{p1}) & q(c,y) & \Leftarrow & s(y) \\
(\text{p2}) & q(a,b) & \Leftarrow & \\
(\text{p3}) & s(c) & \Leftarrow & \\
(\text{p4}) & r(c) & \Leftarrow &
\end{array}$$

Its meaning is the following. The problem is concerned with three given individuals, named a, b, c, four given properties named p, q, r and s, and five relationships that are relevant to the problem.
The real world grants the validity of the expression

$$\begin{array}{l}
\langle \exists a, b, c, p, q, r, s :: \\
\quad \langle \forall x, y :: \\
\qquad (q.(x,y) \wedge r.y \Rightarrow p.x) \\
\quad \wedge \ (s.y \Rightarrow q.(c,y)) \\
\quad \wedge \ (q.(a,b) \wedge s.c \wedge r.c) \\
\quad \rangle \\
\rangle
\end{array}$$

□

In the above we see that the real world problem is modelled by a formula in the ∃∀-shape. But, what if the model requires properties of the ∀∃-shape ?

In such case the model hides some functional relationship between individuals, as we shall see shortly. By introducing additional function symbols this functional relationship can be made explicit and the problem will be modelled by a ∃∀-formula. This explains the presence of the syntactical construct $f(t_1, ..., t_k)$ as a term.
Below we will be more concrete about the transformation of a ∀∃-formula into a ∃∀-formula, a process which is known as "Skolemization". Using Skolemization each first order predicate formula can be brought in clausal form.

Remark. Besides their use in a clausal formulation of 1st order predicates, function symbols can als be exploited to model elements of inductive datatypes as terms. Consequently, relations over inductive datatypes may be described by logic programs. For examples see section 6.2.
□

4.1 Skolemization

Its is well-known that -in general- the quantifiers ∀ and ∃ may not be interchanged, so the question is how and when the interchange of those quantifiers is permissible. The answer is "Skolemization", a process based on the axiom of choice:

Skolem rule

$$\begin{array}{ll} & \langle \forall x :: \langle \exists y :: \varphi.(x,y) \rangle \rangle \\ \equiv_S & \{ \Rightarrow \text{axiom of choice; } \Leftarrow y := f.x \ \} \\ & \langle \exists f :: \langle \forall x :: \varphi.(x, f.x) \rangle \rangle \end{array}$$

□

The axiom of choice relies on a constructive interpretation of ∀∃-formulae. To say $\langle \forall x :: \langle \exists y :: \varphi.(x,y) \rangle \rangle$ means that for each x there is a way of constructing y related to x by φ. The Skolemization of the formula gives the construction a name -f here- guaranteed to exist by adopting the axiom of choice. If this f potentially depends on certain variables then the naming has to reflect this by taking those variables as parameters.

So, if the problem modelling results in a ∀∃-shaped formula, it can be rephrased in a ∃∀-shaped formula containing an external "f". The external "f" represents a function from individuals to individuals and is specified by

$y = f.x$ such that $\varphi.(x,y)$ holds for all x.

* * *

For practical reasons we add two more Skolem-like rules to our repertoire of logical laws, namely for the particular cases where the modelling results in

(i) $\langle \forall x :: \langle \forall y :: p.(x,y) \rangle \Rightarrow q.x \rangle$
(ii) $\langle \forall x :: p.x \Rightarrow \langle \exists y :: q.(x,y) \rangle \rangle$

Re (i)
In this case the formula $\langle \forall x :: \langle \forall y :: p.(x,y) \rangle \Rightarrow q.x \rangle$ can be rewritten as follows:

$$\begin{array}{l} \langle \forall x :: \langle \forall y :: p.(x,y) \rangle \Rightarrow q.x \rangle \\ \equiv \\ \langle \forall x :: \neg \langle \forall y :: p.(x,y) \rangle \vee q.x \rangle \\ \equiv \\ \langle \forall x :: \langle \exists y :: \neg p.(x,y) \rangle \vee q.x \rangle \\ \equiv \\ \langle \forall x :: \langle \exists y :: \neg p.(x,y) \vee q.x \rangle \rangle \\ \equiv \\ \langle \forall x :: \langle \exists y :: p.(x,y) \Rightarrow q.x \rangle \rangle \end{array}$$

Skolemization of the last formula yields $\langle \exists f :: \langle \forall x :: p.(x, f.x) \Rightarrow q.x \rangle \rangle$ and by the above calculation we have established SK0.

SK0: $\langle \forall x :: \langle \forall y :: p.(x,y)\rangle \Rightarrow q.x\rangle$
$\equiv_S$
$\langle \exists f :: \langle \forall x :: p.(x, f.x) \Rightarrow q.x\rangle\rangle$
□

The Skolem function that is introduced here is specified by

(3) $y = f.x$ such that $p.(x,y) \Rightarrow q.x$ holds for all x.

□

Re(ii)
In this case the formule $\langle \forall x :: p.x \Rightarrow \langle \exists y :: q.(x,y)\rangle\rangle$ can be rewritten as follows:

$\langle \forall x :: p.x \Rightarrow \langle \exists y :: q.(x,y)\rangle\rangle$
$\equiv$
$\langle \forall x :: \langle \exists y :: p.x \Rightarrow q.(x,y)\rangle\rangle$

Skolemization of the last formula yields $\langle \exists f :: \langle \forall x :: p.x \Rightarrow q.(x, f.x)\rangle\rangle$ and by the above calculation we have established SK1

SK1: $\langle \forall x :: p.x \Rightarrow \langle \exists y :: q.(x,y)\rangle\rangle$
$\equiv_S$
$\langle \exists f :: \langle \forall x :: p.x \Rightarrow q.(x, f.x)\rangle\rangle$
□

The Skolem function that is introduced here is specified by

$y = f.x$ such that $p.x \Rightarrow q.(x,y)$ holds for all x.
□

Equipped with our extended repertoire of logical laws, a conversion from logical formulae to clausal form has become straightforward: a machine could do the job. Indeed, algorithms exist that convert any first order logical formula to clausal form, but an experienced programmer can usually obtain the result much more effectively by his manipulative agility in logic.

Below we give an example of how the conversion would be achieved by a machine - supplying the steps in the hints- and how it would be achieved by a programmer who masters the predicate calculus.

Example. Convert the following formula to clausal form

$\langle \forall x :: empty.x \equiv \neg\langle \exists y :: y \in x\rangle\rangle$

Solution 1 { using a general algorithm, [7] }

$\langle \forall x :: empty.x \equiv \neg\langle \exists y :: y \in x\rangle\rangle$

$\equiv$ { step 1: eliminate $\equiv$ }

$\langle \forall x :: (empty.x \Rightarrow \neg\langle \exists y :: y \in x\rangle) \ \wedge \ (\neg\langle \exists y :: y \in x\rangle \Rightarrow empty.x)\rangle$

$\equiv$ { step 2: eliminate $\Rightarrow$ }

$\langle \forall x :: (\neg empty.x \vee \neg\langle \exists y :: y \in x\rangle) \ \wedge \ (\langle \exists y :: y \in x\rangle \vee empty.x)\rangle$

$\equiv$ { step 3: distribute $\neg$ towards atoms }

$\langle \forall x :: (\neg empty.x \vee \langle \forall y :: y \notin x\rangle) \ \wedge \ (\langle \exists y :: y \in x\rangle \vee empty.x)\rangle$

$\equiv$ { step 4: distribute $\vee$ towards atoms }

$\langle \forall x :: \langle \forall y :: \neg empty.x \vee y \notin x\rangle \ \wedge \ \langle \exists y :: y \in x \vee empty.x\rangle$

$\equiv$ { step 5: distribute $\forall$ over conjunctions }

$\langle \forall x :: \langle \forall y :: \neg empty.x \vee y \notin x\rangle\rangle \ \wedge \ \langle \forall x :: \langle \exists y :: y \in x \vee empty.x\rangle\rangle$

$\equiv_S$ { step 6: Skolemize, intro f such that $f.x = y$ }

$\langle \forall x, y :: \neg empty.x \vee y \notin x\rangle\rangle \ \wedge \ \langle \exists f :: \langle \forall x :: f.x \in x \vee empty.x\rangle\rangle$

$\equiv$ { $\wedge$ over $\exists$ }

$\langle \exists f :: \langle \forall x, y :: \neg empty.x \vee y \notin x\rangle \ \wedge \ \langle \forall x :: f.x \notin x \Rightarrow empty.x\rangle\rangle$

Adding f to the function symbols we obtain the clauses

$y \notin x \quad \Leftarrow empty(x)$
$empty(x) \Leftarrow f(x) \notin x$

□

Remark. The general algorithm terminates when all $\forall\exists$-formulae are eliminated. In other examples this will require some more steps 5 & 6, because conjuncts may start with a mixed string of $\forall$ and $\exists$ quantifiers.
□

Solution 2 { using the extended repertoire of logical laws }

$\langle \forall x :: empty.x \equiv \neg\langle \exists y :: y \in x\rangle\rangle$

$\equiv$ { distribute $\neg$ }

$\langle \forall x :: empty.x \equiv \langle \forall y :: y \notin x\rangle\rangle$

$\equiv$ { eliminate $\equiv$ }

$\langle \forall x :: (empty.x \Rightarrow \langle \forall y :: y \notin x\rangle) \ \wedge \ (\langle \forall y :: y \notin x\rangle \Rightarrow empty.x)\rangle$

$\equiv$ { distribute $\forall$ }

$\langle \forall x :: empty.x \Rightarrow \langle \forall y :: y \notin x\rangle\rangle \ \wedge \ \langle \forall x :: \langle \forall y :: y \notin x\rangle \Rightarrow empty.x\rangle$

$\equiv_S$ { pred. calc on 1^{st} conjunct
SK0 on 2^{nd} conjunct: intro f such that $f.x = y$ }

$$\langle \forall x, y :: empty.x \Rightarrow y \notin x \rangle \ \wedge \ \langle \exists f :: \langle \forall x :: f.x \notin x \Rightarrow empty.x \rangle \rangle$$

$$\equiv \quad \{ \ \wedge \text{ over } \exists \ \}$$

$$\langle \exists f :: \langle \forall x, y :: empty.x \Rightarrow y \notin x \rangle \ \wedge \ \langle \forall x :: f.x \notin x \Rightarrow empty.x \rangle \rangle$$

Adding f to the function symbols we obtain the clauses

$$y \notin x \quad \Leftarrow empty(x)$$
$$empty(x) \Leftarrow f(x) \notin x$$

□

The Skolem function f that is introduced in this example can be given according to the following precept, see (3)

$$\begin{array}{lll} f.x= & \textbf{if } x = \emptyset \rightarrow & \text{“}choose\ arbitrary\ y\text{”} \\ & [] \ x \neq \emptyset \rightarrow & \text{“}choose\ some\ y \in x\text{”} \\ & \textbf{fi} & \end{array}$$

5 Horn clauses in Prolog

In this section we will show:
(i) how clauses are to be expressed in a Prolog implementation. It is only a conversion from the syntax given in section 4. (For those who want to become familiar with Prolog as an expression means we recommend the text-books of [2] and [10].)
(ii) Although we think that our conversion is sufficient for a first introduction to Prolog we will summarize some items which deserve attention because of a typical Prolog implementation.

5.1 From clausal syntax to Prolog

In order to convert the clausal syntax of section 4 into Prolog clauses note that

- constants, predicate names and function names are rendered by lower-case letters.
- variables are rendered by capitals
- conjunction is written by ","
- "$\Leftarrow$" is written by "`:-`"
- a clause is terminated by "`.`"
- a line contains only one clause

In a clause "`p :- q.`" , `p` is called the "head", `q` is called the "body".
A clause with an empty body is called a "fact" and it is rendered by `p.` .
A program clause has a nonempty head.
The set of clauses with the same predicate name `p` in their head is called the "definition" of `p`.

5.2 Some typical Prolog issues

A conversion of a (well-designed) logical program to Prolog is not guaranteed to behave as expected because of the design decisions in Prolog implementations. Below we will summarize some items. This list is not exhaustive, additions are welcome!

- Facts about a predicate have to textually precede its further rules. Recall the brothers-example in section 3.
- "=" means textual (syntactical) equality, not an arithmetic (semantic) one. See the warning in section 3.

- The built-in predicate `not` takes a query as an argument, evaluates it and then reverses its truth value. The witness information is lost. As an example consider the following Prolog program

```
p(1).
p(3).
r(X):-not(p(X)).
```

On input `r(1).` the answer is `No`.
On input `r(2).` the answer is `Yes`.
On input `r(X).` the answer is `No`.

The `not`-predicate should be used carefully.
Firstly, the standard interpretation of a query as a witnessed existential quantification cannot be given anymore: where a query like `?p(X).` meant "give the truth value of $\langle \exists x :: p.x \rangle$ and provide a witness if it exists", a query like `?r(X).` asks for the value of $\langle \forall x :: \neg p.x \rangle$, no witness is provided even if the answer is "No".
In a system with "`not`'s" universal quantifications are lurking and witnesses are not guaranteed since the answer to a "negative" query is either "Yes" or "No", and a possible witness will not be given because the binding to the variables is lost

Secondly, the meaning of `not` varies with its position in the body of a clause, the reason being the execution mechanism of Prolog. When we have to assign a meaning to a query we must consider its complete search tree. In the above we saw a `not` occurring on the first (and unique) position in the body, below we will explore some other cases.

1. For `r` defined by

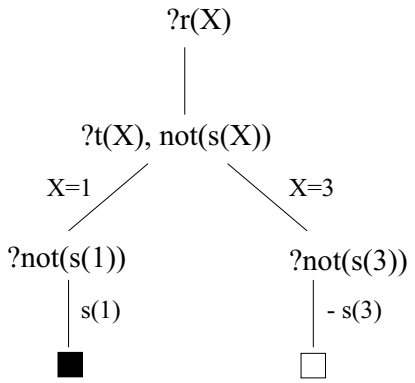
```
r(X):-not(s(X)),t(X).
```

   the meaning of `?r(X).` is $\langle \forall z :: \neg s.z \rangle \wedge \langle \exists x : t.x \rangle$, or, moving the first conjunct into the domain of the existential quantification, $\langle \exists x : \langle \forall z :: \neg s.z \rangle : t.x \rangle$.
   Note that the arguments of s and t are treated independently. The argument of s is "removed" by universal quantification, while the argument of t is still open for witnesses, but only after the first conjunct is found valid. Here, as before, `not(s(X))` refers to negative information about all s-instantiations.

2. For `r` defined by

```
r(X):-t(X),not(s(X)).
```

   the meaning of `?r(X).` is $\langle \exists x : t.x : \neg s.x \rangle$ plus a possible witness. The interpretation of the query still has an existential form and `not(s(X))` refers to negative information about those s-instantiations that satisfy t. In Prolog this is implemented by first finding a witness, say x_0, for t and then exploring whether $\neg s.x_0$ holds. As an illustration we give the search tree for `?r(X).` assuming that the following facts about `t` and `s` are known: `t(1).`, `t(3).`, `s(1).` en `s(2).`



3. For `r` defined by

   ```
   r(X,Y):-t(X),not(s(X,Y)),u(X,Y).
   ```

   the meaning of `?r(X,Y).` is $\langle \exists x : t.x : \langle \forall z :: \neg s.x.z \rangle \wedge \langle \exists y :: u.x.y \rangle \rangle$.
   In Prolog this is implemented by :
   1. find a witness, say x_0, for t,
   2. explore whether $\langle \forall z :: \neg s.x_0.z \rangle$ holds,
   3. if so, then find a witness for $u.x_0.y$

   In each failing (sub)goal backtracking is activated.

4. For `r` defined by

   ```
   r(X,Y):-t(X),u(X,Y),not(s(X,Y)).
   ```

   the meaning of `?r(X,Y).` is $\langle \exists x : t.x : \langle \exists y : u.x.y : \neg s.x.y \rangle \rangle$.
   In Prolog this is implemented by:
   1. find a witness, say x_0 for t
   2. find a witness for $u.x_0.y$, say $y = y_0$
   3. explore whether $\neg s.x_0.y_0$ holds.

From the above it might be clear that each (call to a) subgoal introduces a domain restriction. When a subgoal of shape `not(p(--))` is called in a context where all variables of `p` are restricted then `not` acts like logical negation on the instantiation. Since all restrictions to the left form the restricted context it is advisable to maximize the restriction by placing a "negative" predicate in the rightmost position (if it is necessary at all).

- Given a Prolog program and a goal the interpreter constructs all solutions via a traversal of the search tree. Sometimes **we** know that certain branches are doomed to fail and that efficiency would be improved when those branches were cut off. Other circumstances are that we are interested in some solutions not in all. Prolog enables **us** to prune branches via the built-in predicate "cut" (written as !). The "cut" has no clear declarative meaning (i.e. it is always *true*) but it instructs the interpreter not to try other alternatives beyond the point at which it occurs. This pruning of the search tree is not only determined by the clause in which it occurs but also by

the other clauses.

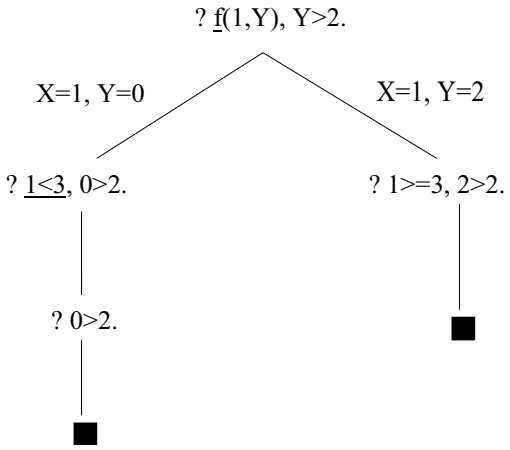
As an example consider the step function

$f.x = 0 \qquad \text{if } x < 3$
$f.x = 2 \qquad \text{if } x \geq 3$

In Prolog this can be written as

```
f(X,0):-X<3.
f(X,2):-X>=3.
```

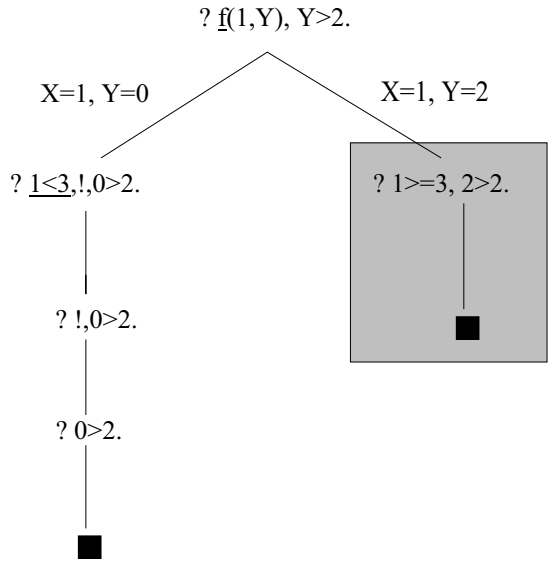
The search tree for query `?f(1,Y),Y>2.` is



In the left preferent unification the `f`-term is reduced first. There are only two alternatives. It is useless to explore the second alternative because the first alternative succeeds and the two alternatives are mutually exclusive. We prune the right branch as follows:

```
f(X,0):-X<3,!.
f(X,2):-X>=3.
```

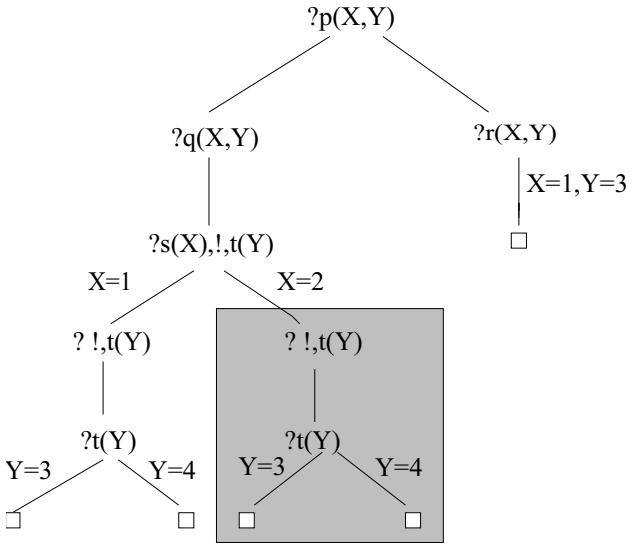
The search tree for query `?f(1,Y),Y>2.` is given below, the shaded rectangle indicates the pruned branch. Once the first alternative for `f` has succeeded no other `f`-alternatives are explored anymore.



As another example consider the program

```
p(X,Y):-q(X,Y).
p(X,Y):-r(X,Y).
q(X,Y):-s(X),!,t(Y).
r(1,3).
s(1).
s(2).
t(3).
t(4).
```

The search tree for query `?p(X,Y).` is given below, the shaded rectangle indicates the pruned branch.



Be aware “cut” should be used with care because it can disrupt the execution of a program in unexpected ways. For detailed information about cut, see [2], [10] and [4].

6 Design of logic programs

A logic program is a set of Horn clauses (with exactly one atomic formula in its antecedent) which forms a mathematical abstraction of a real world problem.

In order to achieve such an abstraction we may be guided by (at least) three formal methods:

1. Predicate logic
2. Function design
3. Query design

Below we will explain how these formal methods can be employed to obtain a logic program.

6.1 Predicate logic as guidance

Predicate logic is widely accepted as a universally applicable tool to formalize a problem. How a logical formalization may lead to a logic program is described in section 4.

6.2 Function design

Functions are a special kind of relations. Since clauses define relations and atomic formulae represent (a special form of) k-ary relations, it may be expected that it is not too difficult to implement a function by a logic program. Designing a logic program via a function definition has two advantages

1. For its correctness we can rely on the correctness of the function definition.

2. All programming techniques known from imperative and functional programming can be exploited to achieve a function definition.

How do we obtain a logic program for a given function f?
We start with a general observation.
A function $f :: \alpha \to \beta$ -where $\beta \neq \mathbb{B}$- defines a relation, say rf, on $\alpha \times \beta$, i.e. a unary function f translates into a binary predicate rf. The additional argument in rf is needed to denote the result values of f and it is common usage in logic programming that the function argument textually precedes the result value, i.e. we will specify rf by

$$rf(x, y) \equiv (f.x = y)$$

Next we have to give a clausal definition of rf which, among others, means that arguments to rf are terms. In a functional environment, arguments can be given by patterns (which are supported by the system) and results by suitable expressions (which can be evaluated by the system). In a logical environment however, this is not always the case. Thus a clausal definition for rf may require more effort than is expected on first sight. As an illustration we give some examples.

6.2.1 Functions over the naturals

Derive a clausal definition for the factorial function.
The factorial function fac is defined by

$$\begin{aligned} &fac :: \mathbb{N} \to \mathbb{N} \\ &fac.0 \;=\; 1 \\ &fac.(n+1) \;=\; (n+1) * fac.n \end{aligned}$$

Its relational version $rfac$ is specified by

$$rfac(x, y) \;\equiv\; (fac.x = y)$$

An inductive definition for $rfac :: \mathbb{N} \times \mathbb{N} \to \mathbb{B}$ may be constructed in the usual way:

case(0) $\quad rfac(0, 1)$

case(n+1)
Construction hypothesis (CH): $rfac(n, y) \;\equiv\; (fac.n = y)$

$$\begin{aligned} & rfac(n+1, m) \\ \equiv \quad & \{ \text{ spec } rfac \ \} \\ & fac.(n+1) = m \\ \equiv \quad & \{ \text{ def } fac \ \} \\ & (n+1) * fac.n = m \\ \Leftarrow \quad & \{ \text{ intro } y \ \} \\ & (n+1) * y = m \;\wedge\; y = fac.n \\ \equiv \quad & \{ \text{ CH } \} \\ & (n+1) * y = m \;\wedge\; rfac(n, y) \end{aligned}$$

Thus we have derived

$$\begin{aligned} &rfac(0, 1) \\ &rfac(n+1, m) \;\Leftarrow\; (n+1) * y = m \;\wedge\; rfac(n, y) \end{aligned}$$

Now we have to bring $rfac$ into clausal form.
The base case is a fact (and thus already in clausal form) and the overall shape of the composite case agrees with the clausal syntax too. The argument $n + 1$ of $rfac$ and the conjunct $(n + 1) * y = m$ need further investigation.

Let us start with the argument $n+1$. It is required to be a term, but, unfortunately, it is not (and most logical systems do not support any means to use this pattern as a term). The problem can be remedied by a global substitution $n := n - 1$, i.e. the composite case can be written as

$$rfac(n, m) \;\Leftarrow\; n * y = m \;\wedge\; rfac(n-1, y) \qquad \text{for } n > 0$$

Now the argument $n-1$ appears in the RHS. Again not a term, but RHS's may contain an arbitrary number of conjuncts so we add one

$$rfac(n,m) \quad \Leftarrow \quad n*y=m \ \wedge \ n1=n-1 \ \wedge \ rfac(n1,y) \qquad \text{for } n>0$$

Next we investigate the conjuncts $n*y=m$ and $n1=n-1$. They have to be expressed in atomic formulae. One would hope that this can be done with standard atoms like "`*(n,y,m)`", but alas!. However most logical systems admit a (limited) form of arithmetic and in Prolog-like systems a kind of assignment is implemented via the reserved word "is", e.g.

$$m \ is \ (n*y)$$

A disadvantage of this implementation is that it only allows uni-directional use, i.e. n and y have to be known before m can be given a value. An impure logical aspect!

Summarizing the results, we obtain the following Prolog definition

```
rfac(0,1).
rfac(N,M) :- N1 is N-1, rfac(N1,Y), M is N*Y.
```

Note that we rearranged the conjuncts in order of their sequentially need in the "is"-construction.

6.2.2 Functions over Lists

Derive a clausal definition for list-concatenation.
The functional definition is

$$\begin{aligned}
&{+\!\!+} :: [\alpha] \times [\alpha] \to [\alpha] \\
&[]{+\!\!+}ys \ = \ ys \\
&(x:xs){+\!\!+}ys \ = \ x:(xs{+\!\!+}ys)
\end{aligned}$$

Its relational version $rconc$ is specified by

$$rconc(us,vs,ws) \ \equiv \ (us{+\!\!+}vs = ws)$$

An inductive definition for $rconc$ is constructed in the usual way:

$\underline{\text{case}([])}$ $\quad rconc([],vs,vs)$

$\underline{\text{case}(u:us)}$
CH: $rconc(us,vs,rs) \ \equiv \ (us{+\!\!+}vs = rs)$

$$\begin{aligned}
& rconc(u:us,vs,ws) \\
\equiv \quad & \{ \text{ spec } rconc \ \} \\
& (u:us){+\!\!+}vs = ws \\
\equiv \quad & \{ \text{ def } ({+\!\!+}) \ \} \\
& u:(us{+\!\!+}vs) = ws
\end{aligned}$$

$$
\begin{array}{ll}
\Leftarrow & \{ \text{ intro } rs \ \} \\
& u : rs = ws \ \wedge \ rs = (us \sharp vs) \\
\equiv & \{ \text{ CH } \} \\
& (u : rs) = ws \ \wedge \ rconc(us, vs, rs)
\end{array}
$$

Thus we have derived

$$
\begin{array}{l}
rconc([], vs, vs) \\
rconc(u : us, vs, ws) \quad \Leftarrow \quad (u : rs) = ws \ \wedge \ rconc(us, vs, rs)
\end{array}
$$

Now we have to bring *rconc* into clausal form.
The base case is okay and the overall shape of the composite case agrees with the clausal syntax too. It is the argument $u : us$ of *rconc* and the conjunct $(u : rs) = ws$ which need a further investigation.

Let us start with the argument $u : us$. It is required to be a term, and, fortunately, it is. Prolog-like systems support a means to use list-patterns as a term: the pattern (a:b:c) may be expressed by the term `[a,b|c]`.

Next we investigate the conjunct $(u : rs) = ws$ and see that it can be eliminated by the 1-point rule. We obtain the following Prolog definition

```
rconc([],Vs,Vs).
rconc([U|Us],Vs,[U|Rs]) :- rconc(Us,Vs,Rs).
```

`rconc` is a pure logical program and we will use it in section 6.2.4 where we illustrate some of the differences between a logic and a functional program.

6.2.3 Functions over trees

While in other programming styles the system support to booleans, integers and lists is taken for granted, the support of a logical system to such subject is negligible; the user is responsible for the type and the operations he requires. Despite this lack of support it is not too difficult to give a "term implementation" of inductively defined sets. The idea is to introduce function symbols for its constructors.
As an example, consider the set $\mathcal{T}$ defined by

Definition The set $\mathcal{T}$ is the least set X such that

$$
\begin{array}{lll}
 & \langle\rangle \in \mathrm{X} & \text{empty tree} \\
\mathrm{lt} \in \mathrm{X},\ \mathrm{rt} \in \mathrm{X},\ \mathrm{n} \in \mathbb{Z} & \Rightarrow \ \langle \mathrm{lt,n,rt} \rangle \in \mathrm{X} &
\end{array}
$$

□

The set $\mathcal{T}$ has two constructors, $\langle\rangle \in \mathbf{1} \to \mathcal{T}$ and $\langle -, -, - \rangle \in \mathcal{T} \times \mathbb{Z} \times \mathcal{T} \to \mathcal{T}$. Introduce the constant `nil` for the first constructor and the function symbol `t` for the latter one. Then

every element of $\mathcal{T}$ has been given a term implementation in Prolog. E.g. `t(nil,1,nil)` is a one-element tree.

Having terms which represent trees we are able to derive clausal definitions for functions on $\mathcal{T}$. As before we will do so by starting from a functional definition.

Derive a clausal definition for the maximum of a non-empty tree.
The functional definition is

$$\begin{array}{lcl} maxT :: \mathcal{T} \to \mathbb{Z} & & \\ maxT.\langle\langle\rangle, n, \langle\rangle\rangle & = & n \\ maxT.\langle\langle\rangle, n, rt\rangle & = & n \uparrow (maxT.rt) \\ maxT.\langle lt, n, \langle\rangle\rangle & = & n \uparrow (maxT.lt) \\ maxT.\langle lt, n, rt\rangle & = & n \uparrow (maxT.lt) \uparrow (maxT.rt) \end{array}$$

Its relational version $rmaxT$ is specified by

$$rmaxT(t, m) \;\equiv\; (maxT.t = m)$$

An inductive definition for $rmaxT$ is constructed as follows:

$\underline{\text{case}(\langle\langle\rangle, n, \langle\rangle\rangle)}$

$$rmaxT(\langle\langle\rangle, n, \langle\rangle\rangle, n)$$

$\underline{\text{case}(\langle\langle\rangle, n, rt\rangle)}$, with $rt \neq \langle\rangle$
CH: $rmaxT(rt, p) \;\equiv\; (maxT.rt = p)$

$$\begin{array}{ll} & rmaxT(\langle\langle\rangle, n, rt\rangle, m) \\ \equiv & \quad \{ \text{ spec } rmaxT \text{ } \} \\ & maxT.\langle\langle\rangle, n, rt\rangle = m \\ \equiv & \quad \{ \text{ def } maxT \text{ } \} \\ & (n \uparrow (maxT.rt)) = m \\ \Leftarrow & \quad \{ \text{ intro } p \text{ } \} \\ & (n \uparrow p) = m \;\wedge\; maxT.rt = p \\ \equiv & \quad \{ \text{ CH } \} \\ & (n \uparrow p) = m \;\wedge\; rmaxT(rt, p) \end{array}$$

$\underline{\text{case}(\langle lt, n, \langle\rangle\rangle)}$, with $lt \neq \langle\rangle$
CH: $rmaxT(lt, p) \;\equiv\; (maxT.lt = p)$

$$\begin{array}{ll} & rmaxT(\langle lt, n, \langle\rangle\rangle, m) \\ \Leftarrow & \quad \{ \text{ similarly } \} \\ & (n \uparrow p) = m \;\wedge\; rmaxT(lt, p) \end{array}$$

$\underline{\text{case}(\langle lt, n, rt\rangle)}$, with $lt, rt \neq \langle\rangle$

CH: $rmaxT(lt, p) \equiv (maxT.lt = p)$
$rmaxT(rt, q) \equiv (maxT.rt = q)$

$$\begin{array}{ll} & rmaxT(\langle lt, n, rt\rangle, m) \\ \Leftarrow & \quad \{ \text{ similarly } \} \\ & (n \uparrow p \uparrow q) = m \ \wedge \ rmaxT(lt, p) \ \wedge \ rmaxT(rt, q) \end{array}$$

Thus we have derived

$$\begin{array}{lll} rmaxT(\langle\langle\rangle, n, \langle\rangle\rangle, n) & & \\ rmaxT(\langle\langle\rangle, n, rt\rangle) & \Leftarrow \ (n \uparrow p) = m \ \wedge \ rmaxT(rt, p) & \text{for } rt \neq \langle\rangle \\ rmaxT(\langle lt, n, \langle\rangle\rangle, m) & \Leftarrow \ (n \uparrow p) = m \ \wedge \ rmaxT(lt, p) & \text{for } lt \neq \langle\rangle \\ rmaxT(\langle lt, n, rt\rangle, m) & \Leftarrow \ (n \uparrow p \uparrow q) = m \ \wedge \ rmaxT(lt, p) \ \wedge \ rmaxT(rt, q) & \text{for } lt, rt \neq \langle\rangle \end{array}$$

Now we have to bring $rmaxT$ into clausal form.
It is only the conjunct $(n \uparrow p \uparrow q) = m$ which has to be investigated. Unfortunately there is no standard predicate for the maximum operator $\uparrow$, but the Prolog arithmetic allows atoms like

$$m \ is \ max(n, p)$$

Using the "`is`"-construction with a binary max only the maximum of two numbers can be calculated. For a more general use we define an additional predicate `max` that computes the maximum of a non-empty list of numbers:

```
max([X],X).
max([X|Xs],Y):-max(Xs,Z),Y is max(X,Z).
```

Summarizing the results, we obtain the following Prolog definition for $rmaxT$

```
rmaxT(t(nil,N,nil),N).
rmaxT(t(nil,N,R),M) :- R\=nil,rmaxT(R,P),max([N,P],M).
rmaxT(t(L,N,nil),M) :- L\=nil,rmaxT(L,P),max([N,P],M).
rmaxT(t(L,N,R),M)   :- L\=nil,R\=nil,rmaxT(L,P),rmaxT(R,Q),max([N,P,Q],M).
```

6.2.4 Logical versus functional programs

Not every functional program has a clausal counterpart. Think of the fact that functions may be arguments as well as results in a functional program while atomic formulae have a limited expressive power.

Furthermore, (most) functional languages are typed while (most) logical languages are untyped. The use of a type system is to detect program errors at compile time thereby enhancing the reliability of execution results. Since logical languages are not equipped with a type system the execution results may be incomprehensible in case of a syntactical and/or semantical incorrect program.

Sometimes the lack of typing gives a programmer more freedom, as we will show in the following examples

Goal: `rconc([1,a],[2,b],L).` Answer: `L = [1,a,2,b]`
Goal: `rconc([1,[2,3]],[a,[]],L).` Answer: `L = [1,[2,3],a,[]]`

The declarative content of `rconc` is the same as in the functional program for (⫴) -the input lists given in the first two arguments are appended and the result is "assigned" to the third argument- but in a functional program a new sum-type has to be defined to comprise the element type of a list.

The most prominent difference between a logic and a functional program is in their input-output behaviour: functional programs only allow uni-directional use while logic programs also allow non-directional use.
Functional programs are executed using reduction or term-rewriting. That is, given a function f and a domain-value v, the arguments in the LHS of the (equational) definition of f are matched against v. This produces bindings of the variables in the equation for f and $f.v$ is rewritten by the RHS bounded to these bindings. Thus, there is a transfer of information from the given v to a use of the equation. Stated differently, functional programs make an explicit commitment as to the inputs and the outputs of a function (input-output directionality).

Logical programs are executed using resolution. That is, given a goal that contains both values and variables, resolution produces all possible bindings which make the goal true. During resolution the LHS of a clause is unified with a goal, potentially producing bindings for the variables in both, the clause and the goal. Thus, the transfer of information is in both directions. Logic programs make no commitment to inputs and outputs of a relation (non-directionality).

To see that this non-directionality has a great impact on the computational content we will give some examples

- Goal: `rconc(L1,L2,[a,b,c]).` Answer: `L1 = []`, `L2 = [a,b,c]`
  `L1 = [a]`, `L2 = [b,c]`
  `L1 = [a,b]`, `L2 = [b]`
  `L1 = [a,b,c]`, `L2 = []`
  "all splittings of `[a,b,c]` are generated"
- Goal: `rconc(L1,[3|L2],[1,2,3,4,5]).` Answer: `L1 = [1,2]`, `L2 = [4,5]`
  "splitting at a specific value"
- Goal: `rconc(_,[X,3,Y|_],[1,2,3,4,5]).` Answer: `X = 2`, `Y = 4`
  "searching for neighbours"
- Furthermore, other relations may be defined as an instance of `rconc`, e.g.

```
member(X,L) :- rconc(L1,[X|L2],L).
```

We see that a single logic program corresponds to several functional programs which contain the same declarative information but have different input-output directionality.
The declarative content of the logic program `rconc` and the functional program (♯) is the same, but the computational content is different.

In particular, a logic program for a function is a logic program for the function's inverse too.

For an extensive discussion about the differences between a logic and a functional program see [8], [2] and [3].

6.3 Query design

In many practical situations databases (database schemes) are a natural means to model a real world problem. A database scheme is a (finite) set of relational schemes together with a (finite) set of (database) constraints. A database instance consists of a set of consistent relational instances of their schemes. A consistent database -or database for short- is a database instance that satisfies all (database) constraints.
As a current example we take the following beer database scheme [2]

Example Scheme for the beer database:

$likes(drinker, beer)$
$visits(drinker, pub)$
$serves(pub, beer)$

The entity and the attribute names are meaningful - their intended meaning is suggested. The (database) constraints are

- Each drinker likes at least one beer
- Each drinker visits at least one pub
- Each pub serves at least one beer

□

From this modelling we may infer that in an actual beer database:
All actual drinkers are mentioned in the *likes* instance as well as in the *visits* instance, all actual pubs are mentioned in the *serves* instance and the actual beers contain at least the beers that are mentioned in the *likes* or in the *serves* instance.

In the sequel we will show that databases and their operations smoothly fit into the logical approach.

6.3.1 Databases, represented logically

In a logical system, a database may be represented in two different ways:

[2] This database scheme is suggested to us by Paul de Bra.

non-deductively: all information is expressed explicitly, i.e.
the database is a collection of facts. Each fact represents a tuple of the database, e.g. likes(rik,triple). It is assumed that this collection satisfies the constraints that are part of the database scheme. Constraints are represented by clauses -"rules"- and their only role is to maintain the database's consistency during update operations.

deductively: the information is partly explicit and partly implicit, i.e.
the database is a collection of facts and rules. There are tuples (or tables) whose values are partly mentioned while their remaining values may be inferred from the rules, e.g. because of inclusion/functional dependencies. Again it is assumed that the collection satisfies the constraints (represented by rules), but in this case the constraints do not solely maintain the database's consistency they also may serve to generate (part of) tuple or table values. Deductive databases form a basis for knowledge based system, rule based systems and expert systems. More detailed information on deductive databases can be found in [5].
A standard example of a deductive database is a Prolog program: it states facts and relations between data in a real world problem. With its Closed World Assumption this means that Prolog's view of the world is limited to a collection of basic and derived facts (outside this collection nothing does hold). Because of this view, a Prolog program is often referred to as a Prolog database.

Besides the choice for a deductive or non-deductive database, we have to decide upon the representation of a tuple. There are two possibilities, each of which with their own application type:

1. a tuple as an atomic formula, i.e.
   the tuple constructor is represented by a predicate symbol while its attribute values are terms. E.g. the entity "rik likes triple" is represented by likes(rik,triple).
   This representation is in accordance with the relational database approach in a very natural way and is used in practical applications.

2. a tuple as a term, i.e.
   the tuple constructor is represented by a function symbol and the (complete) tuple as well as its attribute values are terms. E.g. the entity "rik likes triple" may be represented by item(likes(rik,triple)).
   In this representation properties and relations between two or more tuples can be phrased in an atom or clause. Typically this representation is used when one likes to prove properties of a logical program.

In the remaining part of this section we assume that our example database is represented non-deductively and that tuples are represented by atomic formulae.

6.3.2 Database retrieval

Once a database is implemented (deductively or not) it can be retrieved by queries posed as a goal. Queries are usually phrased in natural language and their formalization might be

surprisingly complicated, even for simple databases like the one above. Whatever database implementation is considered, a safe start of the formalization process is to express the query by a predicate. Afterwards this predicate can be transformed into the required formalism: a goal here, an expression in (e.g.) relational algebra, tuple calculus or SQL in other implementations. A goal -a conjunct of atoms- seems easier to establish than e.g. a relational algebra expression, but the complexity -mainly due to occurrences of negation and/or sets- remains. Below we will give some example queries together with their formalizations in SQL and Prolog. We choose to introduce new predicates to describe answers to queries. In order to keep the SQL formulation compact we will use the following abbreviations

L(d,b) for *likes(drinker,beer)*
V(d,p) for *visits(drinker,pub)*
S(p,b) for *serves(pub,beer)*

Query 0 { Projection }
Which beers are served ?

SQL: Select s.b
From S as s

Prolog:
```
q0(B) :- serves(P,B).
```

Recall that in each proof of a goal, Prolog constructs (at most) one solution. By triggering the backtracking process the remaining solutions are constructed (again one by one).

From the database constraints we infer that all drinkers and all pubs considered in the database scheme can be obtained from a projection of the *likes* table and the *serves* table respectively. I.e. the *drinker* table and the *pub* table can be defined "deductively" by adding the following clauses to the database

```
drinker(D) :- likes(D,B).
pub(P)     :- serves(P,B).
```

□

Query 1 { Selection }
Which pubs serve "Heineken" ?

SQL: Select s.p
From S as s
Where s.b = "Heineken"

Prolog:
```
q1(P) :- serves(P,B),B=heineken.
```

or using the 1-point rule

```
q1(P) :- serves(P,heineken).
```

□

Query 2 { Join }
A potential visitor of a pub is a drinker who may go to that pub because they serve a beer he likes. Extend the serves table with potential visitors.

SQL: Select s*, l.d
From S as s, L as l
Where s.b = l.b

Prolog:
```
q2(P,B,D) :- serves(P,B),likes(D,C),B=C.
```

or 1-point rule

```
q2(P,B,D) :- serves(P,B),likes(D,B).
```

□

Intermezzo. In each proof of a goal, Prolog constructs (at most) one solution. Sometimes we want all solutions to a goal in one proof. For these cases Prolog has three 2nd order standard predicates

- `findall`
- `bagof`
- `setof`

As an example of their use, consider the query `q1L(Ps).` where `q1L` is defined by

```
q1L(Ps) :- findall(P,serves(P,heineken),Ps).
```

Here `findall(P,serves(P,heineken),Ps)` may be considered as a quantified expression: `findall` acts like a quantifier, with `P` as a dummy and `serves(P,heineken)` as the domain. `Ps` is a name coupled to the result which is of type list, the value of `Ps` is a list with all pubs which serve "Heineken".
A similar statement can be made for `bagof` and `setof`.

Note that the solutions collected in `Ps` all have different proofs but it is not guaranteed that `Ps` is a set! There may be more than one proof for a solution. If we are interested in all different solutions we may call

```
q1S(Ps) :- newsetof(P,serves(P,heineken),Ps).
```

Here `newsetof` is a redefined version (by us) of the built-in predicate `setof`, see below. This redefinition was necessary because the predicate `setof` fails when an empty set is encountered while we consider such case as successful.

```
newsetof(Var,Goal,Vs) :- findall(Var,Goal,Vl),list_to_set(Vl,Vs).
```

where `list_to_set` is a Prolog predicate which converts `Vl` of type list to `Vs` of type set.
Also, `newsetof(Var,Goal,Vs)` may be considered as a quantified expression:
`newsetof` acts like a quantifier, with `Var` as a dummy and `Goal` as the domain. `Vs` is a name coupled to the result which is of type set, and the value of `Vs` is a set containing all elements which satisfy the `Goal`.
□

Query 3 { Group by }
Give per drinker the different kinds of beer he likes.

SQL:
```
Select    l.d, distinct l.b
From      L as l
Group by  l.d
```

Prolog:
```
q3(D,Bs) :- drinker(D),newsetof(B,likes(D,B),Bs).
```
□

Query 4 { Group by with aggregation }
Give per drinker the number of different kinds of beer he likes.

SQL:
```
Select    l.d, count distinct l.b
From      L as l
Group by  l.d
```

Prolog:
```
q4(D,N) :- drinker(D),newsetof(B,likes(D,B),Bs),length(Bs,N).
```

`length` is a standard predicate in Prolog.
□

Query 5 { Subqueries }
Which drinker likes the largest number of beers.

SQL:
```
With   ( Select    l.d, count distinct l.b
         From      L as l
         Group by  l.d
         ) as      tmp(d,n)
Select   t.d
From     tmp as t
Where    t.n =     (Select max u.n
                    From   tmp as u)
```

Prolog:
```
q5(D) :- q4(D,N),maxnr(N).
maxnr(M) :- findall(N,q4(D,N),Ns),max(Ns,M).
```

The predicate `max` which computes the maximum of a non-empty list of numbers is defined in section 6.2.
□

Query 6 { Subsets }
Verify whether constraint "Each drinker likes at least one beer" holds.
(In this query we assume that a drinkers table D is available.)
We have to verify whether the drinkers in D are a subset of the drinkers in L.
We choose to answer the validity of this subset-requirement via negation. I.e. we will

construct a table with drinkers who don't like any beer at all. If this table is empty (hence the corresponding Prolog query fails) then the constraint is valid. Otherwise not.

SQL:
```
Select  d.d
From    D as d
Where   not exists  (Select l*
                     From   L as l
                     Where l.d = d.d)
```

Prolog:
```
q6 :- drinker(D),not(likes(D,B)).
```

□

7 Application: Logical acceptors

Context-free grammars are often used to model problem domains. Attribute grammars are context-free grammars in which nonterminals have additional parameters. These parameters are used to compute additional information (about the parsing tree that is generated) during the derivation steps in the grammar.

In this note we will design some elementary components which enables the reader to construct a logical parser for any context-free grammar he likes. After a short introduction to attribute grammars we extend the logical parser for a context-free grammar to one that implements the attribute grammar as well.

7.1 Context-free grammars, logically implemented

Let $G = (N, \Sigma, P, S)$ be a context-free grammar, where N, Σ and P are the sets of nonterminals, terminals and production rules respectively, and where S is a special nonterminal called the start symbol. (We assume that all nonterminals are productive, i.e. for all $X \in N$ (its language) $\mathcal{L}$(X) is nonempty.)
As an example we take

$$\begin{array}{lcl} N & = & \{S, A, B\} \\ \Sigma & = & \{a, b\} \\ P & = & \{S \to AB, A \to AA, A \to a, B \to BB, B \to b\} \end{array}$$

When we have to design a (logical) acceptor for the language of G, the problem is specified by

$$accept(w) \equiv w \in \mathcal{L}(S)$$

Knowing that the language of S is constructed from the language of all nonterminals and terminals, we therefore introduce, for any $X \in N$, a predicate $parse$ defined by

$$parse(X, l, r) \equiv \langle \exists x :: x \in \mathcal{L}(X) \wedge l = x \Vdash r \rangle$$

Then the acceptor can be defined by

$$accept(w) \quad \equiv \quad parse(S, w, \varepsilon) \tag{4}$$

where ε denotes the empty string.
This requirement on *accept* becomes manifest in a logic program as

$$accept(w) \quad \Leftarrow \quad parse(S, w, \varepsilon) \tag{5}$$

(which is correct since an implementation will construct the strongest predicate *accept* satisfying (5)).

Next we will have to implement the $parse$-predicate, but we will deal with this in a more general setting, not just for the example grammar above.

* * *

The production rules of a context-free grammar are basically of the following shapes

$$\begin{array}{lll} X & \rightarrow & YZ \qquad\qquad \{\text{sequence}\} \\ X & \rightarrow & Y \mid Z \qquad\qquad \{\text{alternation}\} \end{array}$$

This leads us to extend the *parse*-predicate to expressions of nonterminals, i.e. we will explore

(i) $parse(YZ, l, r)$
(ii) $parse(Y \mid Z, l, r)$

as follows.

Re(i)

$$\begin{array}{ll} & parse(YZ, l, r) \\ \equiv & \qquad \{\ \bullet y \in \mathcal{L}(Y) \wedge z \in \mathcal{L}(Z)\ \} \\ & l = (y \sharp z) \sharp r \\ \equiv & \qquad \{\ \text{assoc}\ \sharp\ \} \\ & l = y \sharp (z \sharp r) \\ \equiv & \qquad \{\ \bullet r' : true\ \} \\ & l = y \sharp r' \wedge r' = z \sharp r \\ \equiv & \qquad \{\ \bullet y \in \mathcal{L}(Y),\ \bullet z \in \mathcal{L}(Z)\ \} \\ & parse(Y, l, r') \quad \wedge \quad parse(Z, r', r) \end{array}$$

i.e. we have proved, using $\mathcal{L}(Y) \neq \emptyset$ and $\mathcal{L}(Z) \neq \emptyset$

Lemma 1 If $X \rightarrow YZ$ then

$$parse(X, l, r) \equiv \langle \exists r' :: parse(Y, l, r') \ \wedge \ parse(Z, r', r) \rangle$$

□

In a logic program this requirement on $parse(X, l, r)$ becomes manifest as

$$parse(X, l, r) \Leftarrow parse(Y, l, r') \ \wedge \ parse(Z, r', r)$$

(with $\forall r'$ left implicit).

Since (UV)W = U(VW) lemma 1 gives a *parse*-predicate for each finite sequence of nonterminals. For further use we mention

Corollary 1 If $X \rightarrow YZW$ then

$$parse(X, l, r) \equiv \langle \exists r', r'' :: parse(Y, l, r') \quad \wedge \quad parse(Z, r', r'') \ \wedge \ parse(W, r'', r) \rangle$$

□

Re(ii)

$$
\begin{array}{ll}
& parse(Y \mid Z, l, r) \\
\equiv & \quad \{ \bullet l = u \sharp r \} \\
& u \in \mathcal{L}(Y \mid Z) \\
\equiv & \quad \{ \text{def } \mathcal{L} \} \\
& u \in \mathcal{L}(Y) \vee u \in \mathcal{L}(Z) \\
\equiv & \quad \{ \bullet l = u \sharp r \} \\
& parse(Y, l, r) \vee parse(Z, l, r)
\end{array}
$$

i.e. we have proved

Lemma 2 If $X \rightarrow Y \mid Z$ then

$$parse(X, l, r) \equiv parse(Y, l, r) \ \vee \ parse(Z, l, r)$$

□

In a logic program this requirement on $parse(X, l, r)$ becomes manifest as

$$(parse(X, l, r) \Leftarrow parse(Y, l, r)) \ \wedge \ (parse(X, l, r) \Leftarrow parse(Z, l, r))$$

(which follows from predicate calculus).

Since $(U \mid V) \mid W = U \mid (V \mid W)$ lemma 2 yields a *parse*-predicate for each finite alternation of nonterminals.

Having explored the *parse*-predicate for the composition of nonterminals we examine the *parse*-predicate for terminals and ε. This choice is motivated by the fact that RHS's of production rules are (possibly empty) sequences of terminals and nonterminals.

Let t be a terminal or ε, then

$$
\begin{array}{ll}
& parse(t, l, r) \\
\equiv & \quad \{ \text{def } parse \} \\
& \langle \exists u :: u \in \mathcal{L}(t) \wedge l = u \sharp r \rangle \\
\equiv & \quad \{ \mathcal{L}(t) = \{t\} \} \\
& l = t \sharp r
\end{array}
$$

i.e. we have proved

Lemma 3

If $X \rightarrow \varepsilon$ then	$parse(X, r, r)$
For each terminal t:	$parse(t, t : r, r)$

□

The results formulated in the lemmas 1, 2 and 3 are the components from which a logical acceptor may be built for any context-free grammar.
As an illustration we construct a logical acceptor for our example grammar

$$\begin{array}{lcl} accept(w) & \Leftarrow & parse(S, w, \varepsilon) \\ parse(S, l, r) & \Leftarrow & parse(A, l, r') \wedge parse(B, r', r) \\ parse(A, l, r) & \Leftarrow & parse(A, l, r') \wedge parse(A, r', r) \\ parse(A, l, r) & \Leftarrow & parse(a, l, r) \\ parse(B, l, r) & \Leftarrow & parse(B, l, r') \wedge parse(B, r', r) \\ parse(B, l, r) & \Leftarrow & parse(b, l, r) \\ parse(a, a : r, r) & \Leftarrow & true \\ parse(b, b : r, r) & \Leftarrow & true \end{array}$$

□

* * *

In a logical system the arguments of a predicate are subject to a unification process. Because all (non)terminals are constants we don't want to burden the system with unnecessarily unification steps. Therefore we decide to represent the ternary relation *parse* by a collection of binary predicates. As follows.

$$parse(X, l, r) \quad \underline{\text{is represented by}} \quad parseX(l, r) \qquad \text{for all (non)terminals } X$$

A Prolog program for our logical acceptor would then look like

```
accept(L)      :- parseS(L,[]).
parsea([a|R],R).
parseb([b|R],R).
parseS(L,R)    :- parseA(L,R1),parseB(R1,R).
parseA(L,R)    :- parsea(L,R).
parseA(L,R)    :- parseA(L,R1),parseA(R1,R).
parseB(L,R)    :- parseb(L,R).
parseB(L,R)    :- parseB(L,R1),parseB(R1,R).
```

□

7.2 Attribute grammars, logically implemented

Attribute grammars are an extension to context-free grammars in the sense that they add computations to the derivation steps that come with a context-free grammar. In this section we will show that attribute-grammars may serve as a means to generate (elements of the wider class of) context-sensitive languages. The example here is

$$L = \{a^n b^n c^n \mid n \geq 1\}$$

It is a context-sensitive language (in doubt, try out!) but it may be defined as a subset of an even regular language, as follows:

$$L = \{w \in GL \mid p_L.w\}$$

where

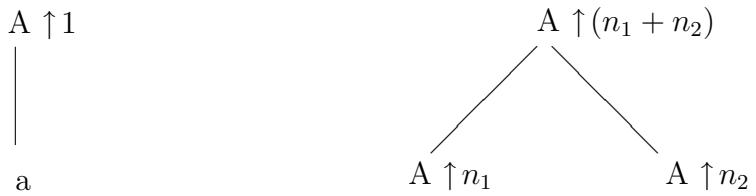
$$GL = \{a^k b^l c^m \mid k, l, m \geq 1\}$$
$$p_L.w \equiv \#(a, w) = \#(b, w) = \#(c, w)$$

The language GL is context-free and a grammar which generates GL is $G = (N, \Sigma, P, S)$ where

$$\begin{aligned} N &= \{S, A, B, C\} \\ \Sigma &= \{a, b, c\} \\ P &= \{S \to ABC, A \to AA, A \to a, B \to BB, B \to b, C \to CC, C \to c\} \end{aligned}$$

An acceptor for GL is easily constructed via the components developed in the previous section. The question now is: how can we compute $p_L.w$ during an acceptance of w ?
Well, an acceptor for GL constructs -for a given $w \in \{a, b, c\}^+$- a parse tree for w, if $w \in GL$. In this construction the production rules of G are used. The idea now is to define functions for each of the nonterminals in such a way that $p_L.w$ can be expressed in terms of these functions. As an example: expression $\#(a, w)$ occurs in $p_L.w$. Looking at the grammar it may be clear that those a's are generated by nonterminal A only. There are two production rules for A, viz. $A \to AA$ and $A \to a$, and together they have to take care of a correct computation of $\#(a, w)$.

The task for $A \to a$ is simple: when this rule is applied in a derivation, exactly one a is generated. For the recursive rule $A \to AA$, we rely on the induction/construction hypothesis that the number of a's are computed along with the derivation of the first and the second A in the RHS, say n_1 for the first A and n_2 for the second A. Then the task for $A \to AA$ is to add n_1 and n_2. Graphically the tasks can be depicted by::



Now that we know how $\#(a, w)$ can be defined by the grammar, we come to the following problem: in grammar land functions like $\#(a, w)$ are not explicitly named. Instead, parameters are added to the relevant nonterminals -in this case one output parameter of type $\mathbb{N}$is added to A- and along with each production rule a condition on these parameters is formulated, viz.

$$A\langle\uparrow n\rangle \to a \qquad\qquad n = 1$$

$$A\langle\uparrow n\rangle \;\rightarrow\; A\langle\uparrow n_1\rangle\ A\langle\uparrow n_2\rangle \qquad\qquad n = n_1 + n_2$$

The up-arrow indicates that the value of n is generated during the derivation. In grammar terminology this is phrased by "n is a synthesized attribute".

Remark. In many other applications nonterminals are "attributed" with input-parameters too such as, for instance, the list of declared variables. In such cases the value of the parameter/attribute must be known from the context ("inherited attributes"). In many a course on Compiler Construction attribute grammars are used to formalize the so-called context-conditions which are put on a context-free grammar.
□

Having seen how $\#(a, w)$ can be computed by the grammar, it is not difficult to see that the following attribute grammar generates our context-sensitive language L.

$$\begin{aligned}
S\langle\uparrow ac\rangle &\rightarrow\; A\langle\uparrow n\rangle\ B\langle\uparrow m\rangle\ C\langle\uparrow l\rangle && ac \equiv (n = m = l)\\
A\langle\uparrow n\rangle &\rightarrow\; a && n = 1\\
A\langle\uparrow n\rangle &\rightarrow\; A\langle\uparrow n_1\rangle\ A\langle\uparrow n_2\rangle && n = n_1 + n_2\\
B\langle\uparrow m\rangle &\rightarrow\; b && m = 1\\
B\langle\uparrow m\rangle &\rightarrow\; B\langle\uparrow m_1\rangle\ B\langle\uparrow m_2\rangle && m = m_1 + m_2\\
C\langle\uparrow l\rangle &\rightarrow\; c && l = 1\\
C\langle\uparrow l\rangle &\rightarrow\; C\langle\uparrow l_1\rangle\ C\langle\uparrow l_2\rangle && l = l_1 + l_2
\end{aligned}$$

□

After this compact introduction to attribute grammars we know that the shape of the context-free grammar is not really changed by an "attribuation": the only difference is that along with parsing the parameters are computed so as to satisfy the given conditions.

In our example, a word w belongs to L if it belongs to GL and if its derivation shows as many a's as there are b's and c's in w. A straightforward extension of the $\mathcal{L}(G)$ parser will do.

```
csaccept(L)      :- csparseS(L,[]).
csparsea([a|R],R).
csparseb([b|R],R).
csparsec([c|R],R).
csparseS(L,R)    :- csparseA(L,R1,N),csparseB(R1,R2,N),csparseC(R2,R,N).
csparseA(L,R,1)  :- csparsea(L,R).
csparseA(L,R,N)  :- csparseA(L,R1,N1),csparseA(R1,R,N2),N is N1+N2.
csparseB(L,R,1)  :- csparseb(L,R).
csparseB(L,R,N)  :- csparseB(L,R1,N1),csparseB(R1,R,N2),N is N1+N2.
csparseC(L,R,1)  :- csparsec(L,R).
csparseC(L,R,N)  :- csparseC(L,R1,N1),csparseC(R1,R,N2),N is N1+N2.
```

1. Note that attribute grammars only "work" for an attribuation in which a (previously known) finite number of parameters are involved. Fortunately, many real-world context-sensitive grammars are captured by this limitation.
2. Often it is difficult to infer the meaning of an attribute from its (mostly) recursive definition. Giving a formal specification of these attributes could promote the very powerful formalism of attribute grammars.

□

8 Appendix: A unification algorithm

A unification algorithm is at the heart of each polymorphic type system and also at the heart of logical systems. In unification algorithms the aim is

- to explore whether two expressions can be made textually identical by applying a suitable substitution for the variables. Such a substitution is called a "unifier".
- and, if such a unifier exists then a "most general unifier" (mgu) has to be constructed.

Our goal is to present a functional version of one of the existing unification algorithms: Robinson's algorithm, [11]. A derivation of this algorithm together with a formalisation of the concepts unifier and mgu is given in [1].

The shape of the expressions to be unified depends on the specific application, e.g. the type-expressions in Haskell and the "terms" in a logical system. In this note -which heavily depends on [1]- expressions are defined as follows.

Definition The set *EXPR* of expressions is generated according to the following BNF syntax

$$E := X \mid F \text{ "(" } E^* \text{ ")"}$$

The nonterminals X and F generate, in a not further specified way, variables and function symbols respectively.
□
Thus an expression is either a variable or a function symbol followed by "(", a (possibly empty) list of expressions and ")". An expression of the form $f()$ is just a constant and for brevity's sake we will omit the parentheses in our examples.

In the sequel we will use the following type conventions
- x, y, z, w denote variables
- a, f, g, h denote function symbols
- e, e' denote expressions
- es, es' denote lists of expressions

Now that we have fixed the kind of expressions to be unified, we will illustrate Robinson's approach in two examples.

8.1 Unification by example

Example 1. Unify the expressions e and e' where

$$e = h(a, w, x, f(f(x)))$$
$$e' = h(z, g(y), g(z), f(y))$$

The first question we have to face is "is unification of e and e' possible?", i.e. can we find

a substitution for the variables that makes e and e' textually identical? Well
- the function names are the same, viz. h, and
- the arities (i.e. length of the argument-lists) are the same, viz. 4

hence the answer depends on a successful unification of the arguments lists of e and e'. Robinson proved that the following left-to-right traversal of the lists yields an mgu, if an mgu exists:

$$\begin{cases} (a, w, x, f(f(x))) \\ (z, g(y), g(z), f(y)) \end{cases}$$

Unify heads, viz. $z := a$.
Continue with the substitution-adapted tails:

$$\begin{cases} (w, x, f(f(x))) \\ (g(y), g(a), f(y)) \end{cases}$$

Unify heads, viz. $w := g(y)$
Continue with the substitution-adapted tails:

$$\begin{cases} (x, f(f(x))) \\ (g(a), f(y)) \end{cases}$$

Unify heads, viz. $x := g(a)$
Continue with the substitution-adapted tails:

$$\begin{cases} (f(f(g(a)))) \\ (f(y)) \end{cases}$$

Unify heads, viz. $y := f(g(a))$
This ends the successive unification process because empty-lists are textually equal.
From the above we may conclude that a unification of e and e' exists and that the calculated mgu is

$$[z := a, w := g(y), x := g(a), y := f(g(a))]$$

□

In order to show that unification is not always as straightforward as it seems from the above, we give another example.

Example 2. Unify the expressions e and e', where

$$e = h(x, x)$$
$$e' = h(y, f(y))$$

The function names and the arities are the same, so we will try to find an mgu for

$$\begin{cases} (x, x) \\ (y, f(y)) \end{cases}$$

Unify heads, viz. $x := y$
Continue with the substitution-adapted tails

$$\begin{cases} (y) \\ (f(y)) \end{cases}$$

Now the substitution $y := f(y)$ doesn't lead to equal heads.
Unification of e and e' is not possible because y **occurs in** $f(y)$.
□

In each unification algorithm an "occur-check" has to be made to disallow self-referential bindings such as $y := f(y)$.

8.2 Definition of the unification algorithm

For a formalisation of the unification process exemplified above we introduce the following ingredients

- A function $unify$, where $unify.e.e'$ denotes the unifier of e and e'
- A way to denote a failing unification. We introduce nil for this purpose.
- A successful unification yields a substitution (unifier), i.e. a list of replacements each of which of the form $var := expr$. (Note that the order of the replacements is significant.) We may encounter the identical replacement $x := x$. In that case nothing has to be done and we don't insert it in the substitution (probably leading to an empty substitution).
- A function $applyL$ where $applyL.\theta.es$ distributes the substitution list θ over the expression list es. Its specification is

$$\begin{array}{lcl} applyL.\theta.es = es', \text{ with} & & \langle \forall i : 0 \leq i <| \ es \ | : es'_i = apply.\theta.es_i \rangle \\ apply.[].x & = & x \\ apply.((x := e) : \theta).x & = & e \\ apply.((x := e) : \theta).y & = & apply.\theta.y \\ apply.\theta.(f.es) & = & f.(applyL.\theta.es) \end{array}$$

- A predicate $occurs\_in$ which establishes the occur-check. Informally this predicate may be described by $occurs\_in.e.e' \equiv e \ is \ a \ proper \ subexpression \ of \ e'$. With induction on the second argument its definition is

$$\begin{array}{l} \neg occurs\_in.e.x \\ occurs\_in.e.f(es) \quad \equiv \quad e \in es \ \vee \ \langle \exists e' : e' \in es : occurs\_in.e.e' \rangle \end{array}$$

  To motivate that an mgu is constructed we mention the following lemma:

  If $\neg occurs\_in.x.e$ then $x := e$ is an mgu of x and e

Now the functional version of Robinson's algorithm is given by

Unification algorithm

$$
\begin{array}{lcll}
unify.x.y & = & [] & \text{if } x = y \\
 & = & [x := y] & \text{otherwise} \\
unify.x.f(es) & = & nil & \text{if } occurs\_in.x.f(es) \\
 & = & [x := f(es)] & \text{otherwise} \\
unify.f(es).x & = & unify.x.f(es) & \\
unify.f(es).g(es') & = & nil & \text{if } f \neq g \vee \#es \neq \#es' \\
 & = & unifyL.es.es' & \text{otherwise} \\
unifyL.().() & = & [] & \\
unifyL.(e : es).(e' : es') & = & nil & \text{if } unify.e.e' = nil \\
 & = & \text{let } \theta = unify.e.e' & \\
 & & \quad tes = applyL.\theta.es & \\
 & & \quad tes' = applyL.\theta.es' & \\
 & & \text{in} & \\
 & & nil & \text{if } unifyL.tes.tes' = nil \\
 & & \theta +\!\!+ unifyL.tes.tes' & \text{otherwise}
\end{array}
$$

References

[1] Backhouse, R.C. *A Unification Algorithm* Handout, Eindhoven University of Technology.

[2] Bratko,I. *Prolog Programming for Artificial Intelligence* Addison-Wesley, Great Britain.

[3] DeGroot, D. *Logic Programming, Functions, Relations, and Equations*, Prentice-Hall, New Jersey.

[4] Flach, P. *Simply Logical*, John Wiley & Sons, Great Britain.

[5] Frost, R.A. *Introduction to Knowledge Base Systems* Collins, London.

[6] Gray, P. *Logic, Algebra and Databases*, Ellis Horwood Limited, Great Britain.

[7] Hogger,C.J. *Essentials of Logic Programming* Oxford University Press, Oxford.

[8] Kowalski,R.A. *Logic for Problem Solving* Elsevier-North Holland, New York,1979a.

[9] Lloyd,J.W. *Foundations of Logic Programming* Springer-Verlag, Berlin.

[10] Malpas, J. *Prolog: A Relational Language and its Applications*, Prentice-Hall, New Jersey.

[11] Robinson,J.A. A Machine-Oriented Logic based on the Resolution Principle. *Journal of the ACM* , 12, 1(1965), 23-41