Modeling Languages: A new Paradigm of Programming *

Tony Hürlimann [a]

[a] *Institute of Informatics, University of Fribourg, Switzerland*
E-mail: tony.huerlimann@unifr.ch , http://www-iiuf.unifr.ch/tcs/lpl

This paper presents a new type of programming language (also called modeling language) which allows a modeler to combine declarative and algorithmic knowledge, that is, mathematical-logical constraints on the one hand and an instruction sequence defining an algorithm on the other hand. The approach is new in the sense that it strictly separates syntactically and sematically the declarative from the algorithmic part. Advantages of doing so are presented. Examples illustrate the flavour of such a language.

Keywords: Modeling Language, Index-set, Hierarchical Structure

1. Introduction

We use programming languages to formulate problems in order to deal with them by computers. At the same time, we want these languages to be readable for human beings. In fact, the emergence of computers has created a systematic way in which problems are formulated as *computations*. We now frequently make use of this mould even if computers are not involved. It is true that problem formulation as computation has a long tradition. The ancient Babylonians (19th–6th century B.C.), who had the most advanced mathematics in the Middle East, formulated all their mathematical problems as computations. They easily handled quadratic and even some cubic equations. They knew many primitive Pythagorean triples. They mastered the calculation of the volume of the truncated pyramid and many more impressive computations [7,28]. Arithmetics was highly developed to ensure their calculations for architectural and astronomi-

* This research is supported by the Federal National Fond of Switzerland and financed by the project no. 1217-45922.95.

cal purposes. But we cannot find a single instance of what we nowadays call a demonstration using symbolic variables and constraints. Only the process of the calculation – today we may say the *procedure* or the *algorithm* – was described. In this sense, the Babylonian mathematicians were in fact computer scientists. To illustrate how they did mathematics, one can consult the examples in [21].

Traditionally, programming languages are classified into three paradigms: imperative, functional, and logic programming [23,27].

The *imperative* – sometimes called *procedural* – *programming paradigm* is closely related to the physical way of how (the von Neumann) computer works: Given a set of memory locations, a program is a sequence of well defined instructions on retrieving, storing and transforming the content of these locations. At each moment, the state of the computation can be represented by the momentary content of the memory locations (its memory-variables), instruction pointer and other register contents. At each step of the computation this state changes until the desired result (the goal state) is obtained, where the computation stops. The explicit notation of a sequential execution, the use of memory-variables (symbolic names) representing memory locations, and the use of assignment to change the values of variables are the characteristic building block of each imperative language. *Object-oriented programming*, where computation proceeds by changing the local state of objects, can be viewed as an extension – or the ultimate consequence – of imperative programming. Object-oriented programming is in a sense opposite to functional programming, where no such states exist.

The *functional paradigm* of computation is based on the evaluation of functions. Every program (i.e. computation) can be viewed as a function which translates an input into a unique output. Hence, a program can be represented by the function $X \xrightarrow{f} Y$, X being the domain of input, Y being the output, and f being the computation. A distinction is made between function definition, which describes how a function is to be computed, and function application, which is a call to a defined function. Since every value can be expressed as a function, there is no need to use explicitly variables, i.e. symbolic names for memory locations, nor do we need the assignment operation. Loops in the imperative paradigm are replaced by recursion. Furthermore, functions are first-class values, that is, they must be viewed as values themselves, which can be computed by other functions [23]. The computational model of functional programming is based on the λ–calculus invented by Church A. (1936) as a mathematical formalism for

expressing the concept of a computation; this was at the same time as Turing invented the Turing machine for the same purpose. One can show that the two formalism are equivalent in the strict sense that both can compute the same set of functions – called the computable functions. We call a programming language *Turing complete*, if it can compute all computable functions. With respect to functional programming we have the important result that: "A programming language is Turing complete if it has integer values, arithmetic functions on these values, and if it has a mechanism for defining new functions using existing functions, selection, and recursion." [23, p. 12].

The *paradigm of logic programming* emerged in the 1960s, when programs have been written that construct proofs of mathematical theorems using the axioms of logic. This led to the insight that a proof can be seen as a kind of computation. But it was realised soon that also the reverse is true: computation can also be viewed as a kind of proof. This inspired the development of the programming language Prolog. In Prolog, a program consists of a set of (Horn)-rules together with at least one goal. Using the backtracking algorithm which consists of resolution and unification, the goal is derived from the rules. A logic programming language can be defined as a "notational system for writing logical statements together with specified algorithms for implementing inference rules." [23, p. 426]. The unification-resolution algorithm is quite limited and has been replaced by various constraint solving mechanisms in the *constraint logic programming*, a modern extension of the logic programming paradigm [20].

All three programming paradigms concentrate on problem representation as a *computation*. The computation on how to solve the problem *is* the representation. We may call a notational system, that represent a problem by writing down its computation to find a solution, an *algorithmic language*. In this sense, all presented programming paradigms consist of algorithmic languages.

> **Definition:** An *algorithmic language* describes (explicitly or implicitly) the process (the computation) of solving a problem, that is, *how* a problem can be processed using a machine. The computation consists of a sequence of instructions which can be executed in a finite time by a Turing machine. The information of a problem which is captured by an algorithmic language is called *algorithmic knowledge* of the problem.

Algorithmic knowledge to describe a problem is so common in our everyday life

that one may ask whether the human brain is "predisposed" to preferably look at a problem in describing its solution recipe (computation). Are we in fact predominantly using "algorithmic thinking"? (see also [22]).

2. Algorithmic Versus Declarative Knowledge

Describing a problem by writing down its computation for solving the problem is only one way of representing it. It is (only) the *algorithmic knowledge* of the problem solution that is described, it is the Babylonian way of problem representation. There exists at least one other way to capture knowledge about a problem; it is the way to define *what* the problem is, rather than saying *how* to solve it. How can we say what the problem is? By defining the properties of the problem! Mathematically, the properties can be described as a feasible space which is a subset of a well-defined state space. Let X be a continuous or discrete (or mixed) state space ($\mathbb{R}^m \cup \mathbb{N}^n$) and let x be an arbitrary element within X ($x \in X$), then any relation $R : X \rightarrow \{false, true\}$ defines a subset Y of X. This is often written as $Y = \{x \in X \mid R(x)\}$. Mathematical and logical formula can be used to specify the relation R. These formula are called *constraints*. The unknown quantities x in $\{x \mid R(x)\}$ are called *variables*. The expression $Y = \{x \in X \mid R(x)\}$ is also called a *mathematical model* for the problem at hand. A notational system that represents a problem by writing down its properties using a state space is called a *declarative language*.

> **Definition:** An *declarative language* describes the problem as a set of properties that can be expressed as a subset of a given state space. This space can be finite or infinite, countable or non-countable. The information of a problem which is captured by a declarative language is called *declarative knowledge* of the problem.

The algorithmic way is a precise sequence of instructions (a recipe) of finding the solution to a given problem. We can be sure that the sequence will terminate and we can even – for most algorithms– calculate a upper bound of its execution time – which is called its *complexity*. The proof is another computation which shows that the first computation is correct.

The declarative way, on the other hand, does not give any indication on how the solution can be found. It only states what the problem is. (That is not much in contrast to an algorithmic statement of the problem, which can even be proven to be correct.) Of course, there exists a trivial algorithm to solve a declaratively

stated problem, which is to enumerate the state space and to check whether a given $x \in X$ violates the constraint R. The algorithm breaks down, however, whenever the state space is infinite. But even if the state space is finite, it is – for most nontrivial problems – so large that a full enumeration is practically impossible.

In practice, however, we are not so badly off. There exists problem classes, even with an infinite state space, which can be solved using general methods. An example is linear programming, another is recursive definitions (see below).

Algorithmic and *declarative* representations are two fundamentally different kinds of modeling and representing knowledge. Declarative knowledge answers the question "what is?", whereas algorithmic knowledge asks "how to?" [8] [63, p. 20]. An algorithm gives an exact recipe of how to solve a problem. A mathematical model, i.e. its declarative representation, on the other hand, defines the problem as a subspace of the state space, as explained above. No algorithm is given to find all or a single element of the feasible subspace.

The functional and logic programming paradigm are sometimes called declarative. The following example will make the difference clear between truly declarative, in the defined sense above, and functional, respectively logic, declaration. In mathematics, the square root can be defined (declaratively) as:

$$\{\sqrt{x} = y \mid y \geq 0, y^2 = x\}$$

One can translate this definition directly into a Prolog program as follows:

```
Sqr(X,Y) :- X = Y*Y.
```

Now the query

```
? :- Sqr(X,7).
```

will eventually return the answer $\{X = 49\}$, since it is easy to multiply $Y * Y$ $(7 * 7)$ and to bind X to 49. If however the query

```
? :- Sqr(49,Y).
```

is made then the interpreter may not return a result, because it does not know what the inverse operation of squaring is. In a constraint logic language, the interpreter may or may not return a result, depending on whether the square root is implemented as an inverse of the square operation. This is but a simple

example, but it shows an essential point: the logic program is declarative only in one direction, but not in the other, except when implementing *explicitly* the inverse into the system. This situation is similar in functional programming. The function (in Scheme)

```
(define (Sqr y)
   (* y y))
```

solves the problem: "given `y` find `(Sqr y)`". The inverse problem, however, which is also part of the mathematical definition, would be: "given the body `(* y y)`, which evaluates to a single value, find the value of `y`". This second problem cannot be solved using the functional definition. To find the square root, one needs an entirely different algorithm (e.g. Newton's approximation formula: $x_n = \frac{x_{n-1}+\frac{a}{x_{n-1}}}{2}$). In a truly declarative language (see below), such as LPL [16], one could write:

```
VARIABLE x; y; CONSTRAINT R: y*y = x; y=7;
```

or

```
VARIABLE x; y; CONSTRAINT R: y*y = x; x=49;
```

The problem specification is valid in both directions and does not change, regardless of what are known and what are unknown quantities. By the way, a clever modeling language (which is not the case for LPL) would reduce the first model immediately to $\{y = 7, x = 49\}$ using only local propagation; the second model would be fed to a non-linear solver which eventually applies a gradient search method, for which Newton's approximation is a special case. One could also imagine that the language has built-in facilities for *symbolically transform* the model. In our case, the transformation could be to extract the root from the first constraint, giving the model:

```
VARIABLE x; y CONSTRAINT R: y = sqrt(x); x=49;
```

Again, the model could be reduced immediately to $\{y = 7, x = 49\}$. That such ideas are not pure speculations has been shown in [18], where a complex procedure is described, that transforms *symbolically* logical constraints into equivalent 0–1 linear constraints.

3. Historical Notes

Declarative and demonstrative mathematics originated in Greece during the Hellenic Age (as late as ca. 800–336 B.C.). The three centuries from about 600 (Thales) to 300 B.C. (Elements of Euclid) constitute a period of extraordinary achievement. In an economical context where free farmers and merchants traded and met in the most important marketplace of the time – the *Agora* in Athens – to exchange their products and ideas, it became increasingly natural to ask *why* some reasoning must hold and not just *how* to process knowledge. The reasons for this are simple enough: When free people interact and disagree on some subjects, then one asks the question of how to decide who is right. The answers must be justified on some grounds. This laid the foundations for a declarative knowledge based on axioms and deductions. Despite this amazing triumph of Greek mathematics, it is not often realised that much of the symbolism of our elementary algebra is less than 400 years old [7, p. 179]. The concepts of the mathematical model as a declarative representation of a problem, are historically even more recent. According to Tarski, "the invention of variables constitutes a turning point in the history of mathematics." Greek mathematicians used them rarely and in an very ad hoc way. Vieta (1540– 1603) was the first who made use of variables on a systematic way. He introduced vowels to represent variables and consonants to represent known quantities (parameters) [7, p. 278]. "Only at the end of the 19th century, however, when the notion of a quantifier had become firmly established, was the role of variables in scientific language and especially in the formulation of mathematical theorems fully recognized" [26, p. 12].

It is interesting to note that this long history of mathematics with respect to algorithmic and declarative representation of knowledge is mirrored in the short history of computer science and the development of programming languages. The very first attempt to devise an algorithmic language was undertaken in 1948 by K. Zuse [25]. Soon, FORTRAN became standard, and many algorithmic languages have been implemented since then.

The first step towards a declarative language was done by LISP. But LISP is not a fully-fletched declarative language, since a function $X \xrightarrow{f} Y$ can only be executed in one direction of calculation, from X to Y. However, an important characteristic of a declarative language is the symmetry of input and output. PROLOG was one of the first attempts to make use of a declarative language, since a PROLOG program is a set of rules. Unfortunately, besides of other limi-

tations, the order in which the rules are written is essential for their processing, showing that PROLOG is not truly declarative either. But Prolog had an invaluable influence on the recent development of *constraint logic programming* (CLP), the only programming paradigm today that merges algorithmic and declarative knowledge. Unfortunately, the declarative part of a problem and the solution process are so closely intermingled that it is difficult to "plug in" different solvers into a CLP language. However, this is exactly what is needed. Since a general and efficient algorithm for all (computable) mathematical problems is unlikely, it is extremely important of being able to link a given declaratively stated problem to *different* solver procedure. The solution and formulation process should be strictly separated.

In logic programming the underlying search mechanism is behind the scene and the computation is intrinsically coupled with the language. This could be a strength because the programmer does not have to be aware of how the computation is taking place, he or she only writes the rules in a descriptive way and triggers the computation by a request. In reality, however, it is an important drawback, because, for most nontrivial problem, the programmer *must* be aware on how the computation is taking place. To guide the computation, the program is interspersed with additional rules which have nothing to do with the description of the original problem. This leads to programs that are difficult to survey and hard to debug and to maintain. This problem is somewhat relieved by the constraint logic programming paradigm, where specialized algorithms for different problem classes are directly implemented in the underlying search mechanism. However, this makes the language dependent of the implemented algorithms and problems that are not solvable by one of these algorithm cannot easily be stated in the language.

In the previous decade a new kind of purely declarative languages emerged in the community of operations research. The insight, that a single method can solve almost all linear programs, led to the development of algebraic languages ([3,4,6,10,16] and others), which are becoming increasingly popular in the community of operations research. Algebraic modeling languages represent the problem in a purely declarative way, although the different languages include also some computational facilities to manipulate the data. One of their strength is the complete separation of the problem formulation as a declarative model from the solution process. This allows the modelers not only to separate the two main

tasks of model formulation and model solution, but also to switch easily between several solvers. This is an invaluable benefit for many difficult problems, since it is not uncommon that a model instance can be solved using one method, and another instance is solvable only using another method. Another advantage of such languages is to separate clearly between model structure, which only contains parameters (place-holder for data) but no data, and model instance, in which the parameters are replaced by a specific data set. This leads to a natural separation between model formulation and data gathering stored in databases. Hence, the main features of these algebraic languages are:

1. Purely declarative representation of the problem,
2. Clear separation between formulation and solution,
3. Clear separation between model structure and model data.

Algebraic languages are very limited in the sense that by definition no algorithm can be written using their notation. However, it is naive to think that, in general, it will sufficient to state a problem in a purely declarative way in order to solve it. Therefore, while these languages can be used successfully for linear and some non-linear classes of models, they failed to be of much help in hard discrete problems.

4. Is Declarative Knowledge Useful?

Why would we like to represent a problem declaratively, since we must solve it anyway and, hence, represent it as an algorithm? The reasons are, besides others, *conciseness*, *insight*, and *documentation*. Many problems can be represented declaratively in a very concise way, while the representation of their computation is long and complex. Concise writings favour also the insight to a problem: we can look at the formulation and grasp the essential "at a glance". The statement $\frac{z}{e^z-1} = \sum_{n\geq 0} B_n \frac{z^n}{n!}$, for example, gives us a concise definition of what the Bernoulli number B_n are [12, p. 285], namely the coefficients of this power series, in a way that we almost can "see" them. However, it does not give us an easy way of finding them.

Another reason why we would like to represent a problem declaratively is documentation. In many scientific papers which deal with a mathematical problem and its solution, the problem is stated in a declarative way for documentational purposes only. However, documentation is by no means limited to human

beings. We can imagine *declarative languages* implemented on a computer like algorithmic languages, which are parsed and interpreted by a compiler. In this way, an interpretative system can *analyse the structure* of a declarative program, can *pretty-print* it on a printer or a screen, can *classify* it, or *symbolically transform* it in order to view it as a diagram or in another textual form.

Are these reasons sufficient to create declarative languages. Certainly! These reasons *alone* justify the design and implementation of declarative languages! There are a whole bunch of tasks we can execute using the computer for declarative knowledge, besides solving the problem (see [17] for more details).

Of course, the burning question is, of whether the declarative way of representing a problem could be of any help in *solving* the problem. The answer is *yes* and *no*. Clearly, the declarative representation $\{x \in X \mid R(x)\}$, in its general form, does not give the slightest hint on how to find such a x. Problems represented in this way can be arbitrary complex to solve.

A well known example is Fermat's last conjecture, which can be formulated as following:

$$\{a, b, c, n \in \mathbb{N}^+ \mid a^n + b^n = c^n, a, b, c \geq 1, n > 2\} \stackrel{?}{=} \emptyset$$

It took the mathematical community 350 years to solve this problem as it seems by now. Only a handful of highly specialized mathematicians understand the proof. (This raises the question of whether the problem can be settled or how many mathematicians are necessary to make the proof acceptable. But that is another story.) Even seemingly harmless problem can be arbitrarily difficult to solve. The problem of finding the smallest x and y such that $\{x, y \in \mathbb{N} \mid x^2 = 991y^2 + 1\}$, is a hard problem, because the smallest numbers contain 30 and 29 digits and apparently no other procedure than enumeration is known to find them.

There are even problems that can be represented in a declarative way, but cannot be computed. The problem of enumerate all even integers $\{x \in \mathbb{N} \mid 2x \in \mathbb{N}\}$, for example, cannot be computed, because the computation would take an infinite time. The problem of printing all real number within a unit circle $\{x, y \in \mathbb{R} \mid x^2 + y^2 \leq 1\}$ is even not enumerable. It is well-known, to give a nontrivial example, that the set of theorems in first-order logic, a restricted subset of classical logic, is undecidable (not computable). This can be shown by reducing the halting problem to the problem of deciding which logical statements are theorems [Floyd al. p. 520]. These examples simply have the shattering

consequence: Given an arbitrary declarative formulation of a problem, we can even not say in the general case whether the problem has a solution or not.

> Hence, it seems that the question, of whether the declarative way of representing a problem could be of any help in solving the problem, can be answered negatively.

However, there are problem classes for which a declarative representation *is* the computation. A famous example is the algorithm of Euclid to find the greatest common divisor (gcd) of two integers. One can easily show that

$$\gcd(a,b) = \begin{cases} \gcd(b, a \bmod b), & b > 0 \\ a, & b = 0 \end{cases}$$

This is clearly a declarative way of representing the problem of finding the gcd , since one can transform it into

$$\{x \in \mathbb{N} \mid x = \gcd(a,b), \gcd(a,b) = \text{if}(b = 0, a, \gcd(b, a \ mod \ b)), a, b \in \mathbb{N}\}$$

This formulation can directly be used to implement a functional (recursive) program.

Another example is the 0–1 Knapsack problem, for which a Knapsack of capacity K should be filled with a subset of n items of volume $w_i, i \in \{1 \ldots n\}$. The problem can be expressed by a predicate P_n^K which is true if a Knapsack of capacity K using a subset of the n items exists and false otherwise:

$$P_n^K = \begin{cases} \text{true}, & \text{if } n = 0, K = 0 \\ \text{false}, & \text{if } n = 0, K > 0 \\ P_{n-1}^K \vee P_{n-1}^{K-w_n}, & \text{otherwise} \end{cases}$$

It is, again, straightforward to write a program (in a functional programming style, here in Pascal) that computes this recursive definition:

```
function Knapsack (n:integer; K:integer; w:IntArray): boolean;
begin
   if (n = 0) then
      if (K = 0) then Knapsack := true else Knapsack := false
   else  Knapsack := Knapsack(n-1, K) or Knapsack(n-1, K-w[n]);
end;
```

While this program has an exponential complexity in the number of items, one can easily translate it into a *memoized recursion* (recursion using a global table which stores for each (n, K) pair whether the Knapsack is "possible", "impossible" or "yet unknown"). The memoized recursion has complexity $O(nK)$, which is the best one can achieve in general for the 0–1 Knapsack problem.

All problems, for which the instances can be reduced to smaller instances until finding a trivial (non-reducible) instance, can be treated in this way. This class of problem is surprisingly large and sometimes even more efficient algorithms can be discovered using this paradigm. There are textbooks in Algorithmics entirely based on it, an example is [24].

A class of problems of a very different kind are linear programs, which can be represented declaratively in the following way:

$$\{\min cx \mid Ax \geq b\}$$

.

From this formulation – in contrast to the recursive definitions above – nothing can be deduced that would be useful in solving the problem. However, there exists well-known methods, e.g the Simplex method, which solves almost all instances in a very efficient way. Hence, to make the declarative formulation of a linear program useful for solving it, one only needs to translate it into a form the Simplex algorithm accepts as input. A commercial standard formulation is the MPSX format [19]. The translation from the declarative formulation $\{\min\ cx \mid Ax \geq b\}$ to the MPSX-form can be automated and is a essential task of the mentioned algebraic languages.

The paradigm of linear programs can be extended to non-linear problems and to some discrete problems too. Most algebraic languages accept also non-linear models. The communication with a (non-linear) solver now becomes more involving, but not impossible. For many solvers, the declarative language only needs to supply the constraints and their derivatives of the problem in a form of executable expressions in order to be able to solve it. This can be done in various ways: (1) generating source code in a algorithmic language (like C or FORTRAN), compile it and link it with the solver, (2) implement callback functions into the modeling language, or (3) generating and compile the code "on–the–fly", a compiler technique that gains more and more acceptance now [11]. Certainly,

the achievements in the domain of non-linear and specific discrete models are much less spectacular than in linear programming. The reasons are that (1) the interface of most solvers to other programs is much less standardised; (2) most algebraic languages work on a interpretative basis (compiling "on-the-fly" has not yet widely gained adepts), which make the adopted solution relatively slow compared to ad hoc methods; (3) the question of how to communicate the smallest necessary amount of information to a solver is still a topic of research; this question is especially disturbing for problems that must call the solver repeatedly. Despite these shortcomings – which will be partially eliminated as we gain more experiences – we can conclude that,

> yes, for a wide range of problems the declarative way of representing them can be of use for solving them.

5. Declarative and Algorithmic Knowledge Combined

Nevertheless, it seems doubtful that the paradigm of *purely* declarative languages will ever be very useful for the large and – from the practical point of view – important class of *discrete* models. Using declarative formulation for many discrete problems in order to solve them really appears to "triple" the effort: (1) stating the problem in a declarative way, (2) translate it for a solver, (3) writing the solver – the last task would have been enough.

Therefore, languages that combine declarative and algorithmic knowledge would be of great help. We could then formulate the solver eventually within the same language as the declarative part of the model. The difference to the constraint logic paradigm is that the declarative and algorithmic knowledge are clearly separated into a declarative and executable part. This is reflected syntactically by the main structure of such a language, which is:

```
MODEL
   <declarative part of the model>
BEGIN
   <algorithmic (executable) part of the model>
END
```

One of the two parts can be empty, indicating that the model is purely declarative or purely algorithmic. The declarative part consists of the basis building blocks of declarative knowledge: variables, parameters, constraints, model check-

ing facilities, and sets (that is a way to "multiply" basis building blocks). This part may also contain "ordinary declarations" of a programming language (e.g., type and function declarations). The executable part consists of all control structures which make the language Turing complete. We may imagine our favourite programming language being implemented in the executable part. We shall call a language that combines declarative and algorithmic knowledge, *modeling language*.

> **Definition:** A *modeling language* is a notational system which allows us to combine (not to merge) declarative and algorithmic knowledge in the same framework. The content captured by such a notation is called *model*.

Instead of going through the lengthy description of the syntax and semantics of such a combined language (details can be found in [17]), several examples are given which illustrates its advantages.

6. Examples

The following six examples are chosen to show that some problems are best stated in an purely algorithmic way (examples 1 and 2), others in a purely declarative way (examples 3 and 4), still others can only be processed efficiently using both paradigms at the same time (examples 5 and 6).

6.1. Sorting

Sorting is a problem which is preferably expressed in an algorithmic way. Declaratively stated, the problem is the following:

Given an array $A[1 \ldots n]$ of integer. Find a permutation such that

$$\forall (i \in \{1 \ldots n-1\}) A[i] \leq A[i+1]$$

.

This is clearly a declarative problem formulation and it could be implemented using the Prolog language as follows:

```
sort(S,T) :- permutation(S,T), sorted(T).
sorted([]).
sorted[X]).
```

```
sorted([X,Y|Z]) :- X=<Y, sorted([Y|Z]).
permutation([],[]).
permutation(X,[Y|Z]) :- append(U,[Y|V],X), append(U,V,W),
                        permutation(W,Z).
```

This will likely be the lowest sort algorithm one could imagine: enumerate all permutation, until one happens to be sorted! (Of course, in Prolog one can also implement an reasonably efficient sort algorithm.) The point is that the declarative form is useless for efficiently sorting an array. A simple algorithm of complexity $O(n^2)$ is the insertion sort or the bubble sort.

The reason why the sorting problem is best formulated as an algorithm is probably that the state space is exponential in the number of items, however, the best algorithm only has complexity $O(n \log n)$.

6.2. Euclid's Algorithm

Euclid's algorithm represent a large class of recursive problems (as mentioned already). Mathematically, the gcd (greatest common divisor) is *defined* as:

$\gcd(n, 0) \equiv n$ (trivial case) , $\gcd(n, m) \equiv \gcd(m, n \ mod \ m)$, $(0 < m < n)$

Note that the symbol $\equiv$ means "is equivalent". Hence, the first part of the definition means that the gcd of a number n and zero is n, but also that every number n can be expressed as $\gcd(n, 0)$; the second definition says that $\gcd(n, m)$ for all positive number can be substituted by $\gcd(m, n \ mod \ m)$, but also the other way round. However, the interesting case is, in fact, only the "forward" direction, that is to interpret the definition as a recurrence:

$\gcd(n, 0) \mapsto n$ (trivial case) , $\gcd(n, m) \mapsto \gcd(m, n \ mod \ m)$, $(0 < m < n)$

where the symbol $\mapsto$ means "can be substituted by". If interpreted as a recurrence, one can immediately derive an algorithm to find the gcd of two numbers: "Substitute recursively the left hand side by the right hand side until the trivial case shows up." This paradigm of recursion is extremely powerful for designing algorithms. Hoare [13] wrote that he had no easy way to explain his Quichsort before he was aware of the recursion in Algol60.

6.3. Assembling a Radio

This example (and the next one) show a problem which is most easily expressed in a purely declarative way. It also illustrates how logical and mathe-

matical constraints can be mixed in the same model formulation. This is worth mentioning, because LPL is the only algebraic language to my knowledge which can mix logical and mathematical constraints in the same model. The example is from [1, p. 28], where also a constraint logic programming formulation is given. The problem can be stated informally as follows:

To assemble a radio any of three types ($T1, T2, T3$) of tubes can be used. The box may be either of wood W or of plastic material P. When using P, dimensionality requirements impose the choice of $T2$ and a special power supply F (since there is no space for a transformer). $T1$ needs F. When $T2$ or $T3$ is chosen then we need S and not F. The price of the components are, as indicated, in the objective function. Which variants of radios are to be built in order to maximize profit?

A formal specification of the model is given in Figure 1.

The constraints are Boolean expressions and the objective function is a mathematical (linear) statement. The variables can be shared because the convention is adopted in LPL to interpret the values (TRUE, FALSE) of a Boolean variable numerically as ONE and ZERO.

The Boolean (0–1) variables are:	
$T1, T2, T3$	use one of the three types of tubes
W, P	use wood or plastic box
F, S	use transformer or a special power supply
The constraints are:	
$\mathrm{XOR}(T1, T2, T3)$	Any one of the tubes can be used,
$W \dot{\vee} P$	The box may be either of wood or plastic,
$P \rightarrow T2 \wedge S$	When using P, dimensionality requirements impose the choice of $T2$ and a special power supply,
$T1 \rightarrow F$	$T1$ needs F,
$T1 \vee T2 \rightarrow S$	When choosing $T2$ or $T3$ we need S,
$F \dot{\vee} S$	Either F or S must be chosen.
The objective is to maximze the revenu:	
$110W + 105P - (28T1 + 30T2 + 31T3 + 25F + 23S + 9W$	
$+6P + 27T1 + 28T2 + 25T3 + 10)$	

Figure 1: Assembling a Radio

In LPL (version 4.30), the model is formulated as follows:

```
MODEL Radio "how to assemble a radio";
 BINARY VARIABLE  T1, T2, T3, W, P, F, S;
 CONSTRAINT
   R1: XOR(T1,T2,T3);     R2: W XOR P;
```

```
  R3: P -> T2 AND S;     R4: T1 -> F;
  R5: T1 OR T2 -> S;     R6: F XOR S;
MAXIMIZE Revenue : 110*W + 105*P
  -(28*T1+30*T2+31*T3+25*F+23*S+9*W+6*P+27*T1+28*T2+25*T3+10);
END Radio.
```

LPL translates the Boolean expressions into a conjunctive normal form and then generates linear constraints, giving a simple 0–1 IP problem which can now be solved be a standard MIP-solver.

It is difficult to imagine how this problem could be formulated in an easy way using only algorithmic knowledge!

6.4. A Two-Persons Zero-Sum Game

Another problem class are two-persons zero-sum games. Such problems are best formulated as linear programs, as shows the coding. The following game was described in [14, p. 770ff]:

Let us play the following two-persons game: Both players chose at random a positive number and note it on a piece of paper. They then compare them. If both numbers are equal, then neither player gets a payoff. If the difference between the two numbers is one, then the player who has chosen the higher number obtains the sum of both; otherwise the player who has chosen the smaller number obtains the sum of both. What is the optimal strategy for a player, i.e. which numbers should be chosen with what frequencies to get the maximal payoff?

The game is a two-persons zero-sum game and can be formulated as shown in Figure 2.

The sets are:	
I	a set of strategies for player one
J	a set of strategies for player two
The parameters with $i \in I, j \in J$ are:	
$p_{i,j}$	the payoff matrix
The variables are:	
x_i	optimal frequency of the i-th strategy (player one)
The constraint is:	
$\sum_i x_i = 1$	the probabilities must sum to one
The objective is to maximize the minimal gain is:	
MAXIMIZE:	$\min_j(\sum_i p_{i,j}x_i)$

Figure 2: A 2-Persons-Zero-Sum Game

This game has an interesting solution as will be seen shortly. Let us just play the game for a while to get a feeling for it. Suppose a player chooses a very big number, say a million, then chances are high that the second player chooses a smaller number and wins the round with a high payoff. If, on the other hand, the first player chooses a very small numbers chances are high (but not so high) that the second player chooses a bigger number. Hence the game is not symmetric with respect to big and small numbers. At least by choosing a small number, a player cannot loose a large amount of money. But choosing one all the time will also be a loosing strategy, since the other player then always chooses two.

The number of strategies is infinite (any positive number can be chosen). Let us restrict ourselves to the range of numbers, say [1,50], otherwise we have an infinite large problem! (The result will justify this choice). The payoff matrix is calculated as follows:

$$p_{i,j} = \begin{cases} -j, & \text{if } i > j+1 \\ i+j, & \text{if } i = j+1 \\ 0, & \text{if } i = j \\ -i-j, & \text{if } i = j-1 \\ j, & \text{if } i < j-1 \end{cases}$$

In LPL, the complete model is coded as follows:

```
MODEL Game "A finite two-person zero-sum game";
   SET i ALIAS j := /1:50/;
   PARAMETER p{i,j}:= IF(j>i,IF(j=i+1,-i-j,MIN(i,j)),
                          IF(j<i,-p[j,i],0));
   VARIABLE   x{i};
   CONSTRAINT  R: SUM{i} x[i] = 1;
   MAXIMIZE  gain: MIN{j} (SUM{i} p[i,j]*x[i]);
   WRITE x;
END Game.
```

Note that this formulation is not a linear program. LPL, however, will automatically translate this model symbolically into a linear one: the maximizing function is replaced by the following part of the model:

```
VARIABLE X11  -- introduce a new (continuous) variable
CONSTRAINT X12{j}:=X11<= SUM{i} p[i,j]*x[i]
MAXIMIZE gain:=X11
```

(The surprising result of this model is, that a player should never choose numbers that are larger than 5. One should be chosen with frequency 24.75%, two with 18.81%, three with 26.73%, four with 15.84%, and five with 13.86%.)

Again, it would be hard to find a formulation of this problem class which is more economic. Furthermore, only the definition of the parameter p must be changed to get a different two-persons zero-sum game!

6.5. The Cutting Stock Problem

Let us now present two examples, for which it is probably best to mix algorithmic and declarative knowledge in a well-defined and controlled way. The problem is the following:

Paper is manufactured in rolls of 100 inches width. The following orders have to be fulfilled:

97 rolls of 45 inches width
610 rolls of 36 inches width
395 rolls of 31 inches width
211 rolls of 14 inches width

How should the initial rolls of 100 inches width be cut into slices such that paper waste is minimized?

Using the common mathematical notation, the problem can be formulated in a purely declarative way as shown in Figure 3.

Two sets are declared as:	
W	the different widths that have been ordered,
P	all possible cutting patterns
With $w \in W, p \in P$, the parameters are:	
$a_{w,p}$	number of cut rolls of width w in pattern p
d_w	demand for the cut rolls
With $p \in P$, the variables are:	
X_p	number of the initial rolls cut according to pattern p,
The constraints are as following:	
$\sum_p a_{w,p} X_p \geq d_w$	orders are to be met forall $w \in W$
Minimize the number of rolls:	
MINIMIZE: $\sum_p X_p$	

Figure 3: An LP of the Cutting Stock Problem

Unfortunately, this formulation has, even for moderate size problem instances, a very large number of variables which exceeds any memory capacity on a computer. While compactly formulated in a purely declarative way, this

problem cannot be solved directly.

A well-known method in operations research to solve this problem is to use a column generation method (see [5, Chap. 13] for details), that is, a small instance with only a few patterns (variables) is solved and a rewarding column is added repeatedly to the problem. The new problem is then solved again. This process is repeated until no pattern can be added. The problem consists of two declaratively formulated problems (a linear program and a Knapsack problem), which are repeatedly solved. Hence, part of the problem can be formulated declaratively (Figure 4 combines both declarative models), and part of the problem is stated in an algorithmic form (Figure 5).

Two sets are declared as:	
W	the different widths that have been ordered,
P	all possible cutting patterns
With $w \in W, p \in P$, the parameters are:	
$a_{w,p}$	number of cut rolls of width w in pattern p
d_w	demand for the cut rolls
b_w	width of the cut (ordered) rolls
B	width of the initial rolls
With $p \in P$, the variables are:	
X_p	number of the initial rolls cut according to pattern p,
y_w	number of rolls of size w in a newly generated pattern,
z	total number of initial rolls used,
v	contribution that a newly generated pattern can make.
The constraints are as following:	
(1) $\sum_p a_{w,p} X_p \geq d_w$	orders are to be met forall $w \in W$
(2) $z = \sum_p X_p$	the total number of initial rolls is given
(3) $\sum_w b_w y_w \leq B$	the initial width cannot be exceeded by a pattern
(4) $v = \sum_w C_w^* y_w$	the contribution (C_w^* are the marginals of (1))
Minimize the number of rolls:	
MINIMIZE: $\sum_p X_p$	

Figure 4: The Cutting Stock Problem (declarative part)

There are clear advantages of stating part of the problem declaratively and part of it algorithmically, without commingle them with each other: (1) the code becomes more readable and compact, (2) the two parts can be maintained separately, giving more flexibility, (3) many problems have a knowledge part that is more suitable to be formulated in a declarative way, and another part that is more apt to be stated algorithmically, an optimal mix can be chosen.

Initialize P: Let P be a set of the same cardinality as W.
Initialize $a_{w,p}$: if $w = p$ then $a_{w,p} = \lfloor \frac{B}{b_w} \rfloor$ else $a_{w,p} = 0$
Solve the cutting-stock model defined by: $\{\min z$, subject to (1) and (2) $\}$
Solve the find-pattern model defined by: $\{\max v$, subject to (3) and (4) $\}$
while $v > 1$ do
  add a new element s to P ($P = P \cup \{s\}$)
  update the table a as follows: $a_{w,s} = y_w$
  Solve the cutting-stock model
  Solve the find-pattern model
endwhile

Figure 5: The Cutting Stock Problem (algorithmic part)

The overall structure of the cutting stock problem, coded in LPL version 4.30 is:

```
MODEL CuttingStock "The cutting stock problem" ;
   MODEL cutting_stock;  <...define the model...>
   MODEL find_pattern;   <...define the model...>
BEGIN  (* executable (procedural) part *)
   SOLVE cutting_stock;
   SOLVE find_pattern;
   WHILE <...condition...> DO
      <...add a new pattern, found in find_pattern...>
      SOLVE cutting_stock;
      SOLVE find_pattern;
   END;
END CuttingStock.
```

This is a compact and very readable way to formulate the process of column generation! The declarative part of the model consists of stating two (sub)- models, the smaller cutting-stock model (an lp) and the find-pattern model (a Knapsack problem). Each of them could contain its own declarative and algorithmic part. Since the first submodel is an lp, and the second one can easily be solved by a standard MIP solvers, if $\mid W \mid$ is not too large, neither model does need to contain their own executable part. Both can be formulated as purely declarative models.

The complete model (also coded in LPL 4.30) can now be written as follows:

```
MODEL CuttingStock "The (fractional) cutting stock problem" ;
  SET
    widths ALIAS w         "the different ordered smaller rolls";
```

```
    patterns ALIAS p       "possible cutting patterns";

    MODEL cutting_stock ALIAS cs;
      PARAMETER
        a*{w,p}             "number of w rolls in p";
        demand{w}           "the demand for small roll w";
     VARIABLE
        rolls_cut{p}    "number of rolls cut according to p";
        total_number    "the total number of rolls to be cut";
      CONSTRAINT
        cuts-{w}   ::= SUM{p} a * rolls_cut >= demand;
        objective  ::= total_number = SUM{p} rolls_cut;
      MINIMIZE obj ::= total_number;
      WRITE rolls_cut; total_number;
    END cutting_stock.

    MODEL find_pattern ALIAS fp(c);
      PARAMETER
        length{w}         "the length of small roll w";
        total_length      "the width of the (uncut) roll";
      VARIABLE
         y-{w} INTEGER    "the number of rolls of size w";
         contribute-      "the contribution of a new pattern";
      CONSTRAINT
        pattern    ::= SUM{w} length * y <= total_length;
        objective  ::= contribute = SUM{w}  c.dual * y;
      MAXIMIZE obj ::= contribute;
    END find_pattern.

  BEGIN  (* -------- executable part --------- *)
    SOLVE cutting_stock;
    SOLVE find_pattern(cs.cuts);
    WHILE (fp.contribute > 1) DO
        p := p + {'pattern_'+str(card(p))}; (* union operator *)
        cs.a{w,#p} := fp.y[w];
        SOLVE cutting_stock;
        SOLVE find_pattern(cs.cuts);
    END;
    WRITE cutting_stock;
```

```
END CuttingStock.
```

It should be noted that [2] has independently developed similar ideas, and the new version of their AIMMS modeling system implements them.

6.6. The Travelling Salesperson Problem (TSP)

This problem can be stated as follows: the shortest round-trip of n towns is to be found visiting each town exactly once. There exist many way to tackle the TSP problem. One is sketched here to illustrate again how useful a combination of declarative and algorithmic knowledge can be. The approach is also a well-known technique in operations research: row-cut generation. First the TSP is stated as an assignment problem, which is solved. As long as some subtours exist in the solution, constraints are added. Unfortunately, adding these constraints might introduce non-integer (extreme) solutions, and many other cuts are needed – but this is another story. Here in our model, integrality on the variables is imposed explicitly to illustrate our point. (We do not claim it to be an efficient solution approach.) The problem can be stated (in a future version of LPL ; LPL 4.30 does not have yet the feature FUNCTION) as follows:

```
MODEL Tsp;
   SET i ALIAS j "locations to visit";
   PARAMETER c{i,j} "distance matrix";
   VARIABLE x{i,j} BINARY "=1 if (i,j) is in the tour else 0";
   CONSTRAINT  R1{i} : SUM{j} x = 1;  R2{j} : SUM{i} x = 1;
   MINIMIZE tour: SUM{i,j} c*x;
   SET k; s{k,i};
   CONSTRAINT SubToursEli{k} :
      SUM{i IN s[k] circular} x[i,i+1] <= #s[k]-1;
   FUNCTION StrongComponent(x) BEGIN <...> END;
BEGIN
   SET temp :={};
   PARAMETER n := 0;
   WHILE true DO
       SOLVE TSP USING lp;
       temp := StrongComp(x);
       IF #temp[1]=#i then BREAK;
      WHILE temp[1] DO
          n:=n+1;
```

```
            Add(k,n);
            Add(s,temp);
        END;
      END;
  END Tsp.
```

7. Conclusion

The few examples show clearly that a modeling language, that is a language which combines algorithmic and declarative knowledge, can have clear benefits. Problems which cannot be formulated easily in either paradigm, can be expressed compactly and in a "natural" way. All these problems have been implements previously using algorithmic knowledge alone. This led to ad hoc, and lengthy code, where the process of solving and the task of formulating the declarative part are closely woven with each other. Such programs are harder to debug, to maintain, to document, and to understand.

Modeling, that is, the skill to translate a problem into a declarative and algorithmic code, should no longer be an "ad hoc science" where "everything goes". It should be – like programming, the skill to write algorithms – an art not a black art. Furthermore, modeling should follow certain criteria of quality in the same way as software engineering teaches it for writing algorithmic programs.

The main criteria are: *reliability* and *transparency*. Reliability can be achieved by a unique notation to code models, and by various checking mechanisms (type checking, unit checking, data integrity checking and others). Transparency can be obtained by flexible decomposition techniques, like modular structure as well as access and protection mechanisms. This is probable the hottest topic in software engineering and language design today. Hence, what is true for programming languages is even more true for modeling languages [63].

References

[63] ABELSON H. & SUSSMAN G.J. & SUSSMAN J., 1985. Structure and Interpretation of Computer Programs, The MIT Press, Cambridge, Mass.

[1] BARTH P., 1996. Logic–Based 0–1 Constraint Programming, Kluwer Academic Publishers, Boston.

[2] BISSCHOP J., 1997. Personal Communication, Paragon Decision Technology B.V.

[3] BISSCHOP J. & ENTRIKEN R., 1993. AIMMS, The Modeling System, Paragon Decision Technology B.V.

[4] BROOKE A. & KENDRICK D. & MEERAUS A. 1988. GAMS, A User's Guide, The Scientific Press.

[5] CHVÁTAL V., 1973. Linear Programming, W.H. Freeman Company, New York.

[6] CUNNINGHAM K., & SCHRAGE L. 1989. The LINGO Modeling Language, University of Chicago, Preliminary, 27 February.

[7] EVES H., 1992. An Introduction to the History of Mathematics, sixth edition, The Saunders Series, Fort Worth.

[8] FEIGENBAUM E.A., 1996. How the What Becomes the How, *Communications of the ACM*, Vol. 39, No. 5, pp 97–104.

[9] FLOYD R. W. & BEIGEL R., 1994. The Language of Machines, An Introduction to Compu tability and Formal Languages, Computer Science Press.

[10] FOURER R. & GAY D.M. & KERNIGHAN B.W., 1993. AMPL, A Modeling Language For Mathematical Programming, The Scientific Press, San Francisco.

[11] FRANZ M., 1994. Code–Generation On–the–Fly: A Key to Portable Software, Informatik–Disseration, ETH Zrich, Nr. 47, Verlag der Fachvereine Zrich.

[12] GRAHAM R.L. & KNUTH D.E. & PATASHNIK O., 1994. Concrete Mathematics, (2nd edition), Addison–Wesley Publ. Comp., Reading, Massachusetts.

[13] HOARE C.A.R. & Jones C.B., 1989. Essays in Computing Science, Prentice Hall, New York.

[14] HOFSTADTER D.R., 1988. Metamagicum, Fragen nach der Essenz von Geist und Struktur, Klett–Cotta, Stuttgart.

[15] HOPL 1993: Cambridge, Massachusetts, USA, History of Programming Languages Conference (HOPL–II), Preprints, Cambridge, Massachusetts, USA, April 20–23, 1993. SIGPLAN Notices 28(3), March 1993.

[16] HÜRLIMANN T., 1997. Reference Manual for the LPL Modeling Language, Working Paper, Version 4.25, November 1997, Institute of Informatics, University of Fribourg, (newest version is always on: ftp://ftp–iiuf.unifr.ch/pub/lpl/doc, file Manual.ps).

[17] HÜRLIMANN T., 1997a. Computer–Based Mathematical Modeling, Habilitation Script, accepted by the Faculty of Economic and Social Sciences of the University of Fribourg, Switzerland, December 1997, Institute of Informatics, University of Fribourg.

[18] HÜRLIMANN T., 1998. An Efficient Logic-to-IP Translation Procedure, Working Paper, March 1998, Institute of Informatics, University of Fribourg, (a PostScript version is at the LPL site: ftp://ftp-iiuf.unifr.ch/pub/lpl/doc, file APMOD1.ps).

[19] IBM, 1988. Mathematical Programming System Extended/370 (MPSX/370), Version 2, Program Reference Manual.

[20] JAFFAR J. & MAHER M.J., 1996. Constraint Logic Programming: A Survey, (to appear), (a final draft can be downloaded from pop.cs.cmu.edu in the directory: /usr/joxan/public).

[21] KNUTH D.E., 1996. Selected Papers on Computer Science, Cambridge University Press. Chapter 11, Ancient Babylonian Algorithms, (first print 1972).

[22] KNUTH D.E., 1996a. Selected Papers on Computer Science, Cambridge University Press. Chapter 4, Algorithms in Modern Mathematics and Computer Science, (first print 1979).

[23] LOUDEN K.C., 1993. Programming Languages – Principles and Practice, PWS–KENT Publ. Comp., 1993.

[24] MANBER U., 1989. Introduction to Algorithmics, A Creative Approach, Addison–Wesley Publ. Comp., Reading, Massachusetts.

[25] NAUR P., 1981. The European Side of the Last Phase of the Development of ALGOL 60, in: History of Programming Languages, Wexelblat R.L. (ed.), (from the ACM SIGPLAN History of Programming Languages Conference, June 1–3, 1978), Academic Press.

[26] TARSKI A., 1994. Introduction to Logic and to the Methodology of the Deductive Sciences, 4rd edition, Oxford University Press.

[27] WILSON & CLARK, 1993. Comparative Programming Languages, Addison–Wesley, 1993.

[28] YAGLOM I.M., 1986. Mathematical Structures and Mathematical Modelling, Gordon and Breach Science Publ., New York, (transl. from the Russian by D. Hance, original ed. 1980).
@Bookpardal1-1, author=E.H.L. Aarts and J.H.M. Korst, title=Simulated annealing and Boltzmann machines, publisher=Wiley, year=1989
@Articlepardal1-2, author=R.E. Burkard and F. Rendl, title=A thermodynamically motivated simulation procedure for combinatorial optimization problems, journal=European J. Operations Research, volume=17, year=1984, pages=169-174