Declarative Languages in Education

Hugh Glaser, Pieter H. Hartel, Michael Leuschel and Andrew Martin
{hg,phh,mal}@ecs.soton.ac.uk, apm@comlab.ox.ac.uk

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-2000-1
January 1 2000

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom

Declarative Languages in Education

Hugh Glaser, Pieter H. Hartel, Michael Leuschel * and Andrew Martin †

January 1, 2000

1 Introduction

Declarative languages have been used since the beginning of automated computation, but have been a minority taste for general-purpose programming, while remaining important for a number of special-purpose languages. In addition, however, they have been widely used in the higher education sector, both for Computer Science students and other disciplines. In this article we will consider this particular area.

Following this introduction, we introduce declarative languages, discussing both the mathematics on which they are based (Section 2) and the languages (Section 3). The two subsequent sections are concerned with education. We have separated out two aspects of declarative languages, so that the objectives of each can be considered separately. The first of those is about teaching declarative languages themselves (Section 4), and using the process to learn programming skills and principles. The second aspect concerns using declarative languages in the curriculum to introduce and understand other subjects and ideas (Section 5). The last section concludes with a summary.

The article contains an extensive bibliography. Where possible we refer to books which might be used as teaching texts. We also include the more important and relevant papers in the field.

1.1 What is a Declarative Language?

The commonly-used programming languages, such as FORTRAN, Java, C++ and Cobol would not normally be considered declarative languages. Consequently it is tempting to describe a declarative language in terms of how it differs from such languages: what it is not, rather than what it is. Before making this comparison, however, we will discuss the characteristics that identify declarative languages.

The unifying theme behind the idea is that the programmer should be able to rise above the step-by-step underlying mechanism of the machine, and 'declare' the answer to a problem; the computer system should then be able to interpret that declaration to build the answer.

Declarative systems are common in everyday (engineering) life. Electronic engineers can draw circuit diagrams from which the systems are built, without being concerned with specifying the mechanics of construction. We are all happy to use maps to travel from place to place, and indeed many people prefer them and are uncomfortable with being given step-by-step instructions. Architects draw plans which show the floor layout without needing to specify details of construction, such as the fact the second floor must await the building of the first!

So how might we achieve this static, declarative, world?

Declarative languages have one thing in common in the way they approach the programming process: they appeal to mathematics. Of course, such appeal is not unique to declarative languages, but it is the case that the proponents look to find clean mathematical models and semantics to an extent that is rarely seen in other languages. When we consider the aims of declarative languages above, it is not really surprising that mathematics is a good place to start. Most mathematical systems are static and descriptive, since they are concerned with formulae defining (cf. declaring) values, rather than the process of using the formula definition. Recall that the first steps in programming languages were concerned with allowing programmers to write algebraic formulas (FORmula TRANslation) instead of the individual instructions.

It should now be possible to see the essential difference between declarative and non-declarative (usually called procedural) languages. Procedural languages request and require the programmer to specify a process or recipe for arriving at the solution to the problem, whereas declarative languages neither permit nor allow the process to be specified, but rely on a more abstract statement of the solution.

Declarative languages, are high-level languages in which one only has to state *what* is to be computed and not necessarily *how* it is to be computed. Logic programming, functional programming, and specification represent three prominent members of this class of languages. Functional programming is based on the λ-calculus, logic programming has its roots in first-order logic and automated theorem proving, and specification is based on logic

*Dept. of Electronics and Computer Science, Univ. of Southampton, Email: hg,phh,mal@ecs.soton.ac.uk

†Oxford University Software Engineering Centre, Email: apm@comlab.ox.ac.uk

and set theory. All three approaches share the view that a program is a theory and execution consists in performing deduction from that theory.

2 Mathematical bases

In this section we look at the major mathematical systems that are used to underpin declarative languages. We begin with the predicate calculus, which forms the foundation of logic programming. Section 2.2 introduces the λ-calculus, which underpins functional programming. Finally, Section 2.3 discusses set theory, which forms the basis of specification languages.

2.1 Predicate Calculus

Formal logic is an important branch within mathematics and originated from the attempt to formalise mathematical proof and truth. Formal logic also plays a central rôle within many areas of computer science. It is thus natural that logic is important within education of both mathematics and computer science [83, 110, 50, 115, 105].

Many different logics exist, but predicate calculus, also called first-order logic, is arguably the most wide-spread and widely used. The most basic logic is propositional logic. Formulas within the predicate calculus are constructed from the following connectives and quantifiers:

$\neg$ (negation), $\vee$ (or), $\wedge$ (and),
$\leftarrow$ (implication), $\leftrightarrow$ (equivalence),
$\forall$, $\exists$ (universal and existential quantifier).

The basic constituents are predicates whose arguments are constructed from logical variables and function symbols. For example the formula below states that disjunction ($\vee$) distributes over conjunction ($\wedge$).

$$\mathsf{a} \vee (\mathsf{b} \wedge \mathsf{c}) \leftrightarrow (\mathsf{a} \vee \mathsf{b}) \wedge (\mathsf{a} \vee \mathsf{c})$$

The meaning of the symbols a b, and c is unspecified, which brings us to the topic of the semantics or model theory of the predicate calculus.

2.1.1 Model theory

The semantics of the predicate calculus is based on interpretations and models. An interpretation over an arbitrarily chosen domain $\mathcal{D}$ assigns each function symbol with arity $\mathsf{n} \geq 0$ to a function $\mathcal{D}^{\mathsf{n}} \mapsto \mathcal{D}$ and each predicate symbol to a relation over $\mathcal{D}^{\mathsf{n}}$.

Based upon this interpretation we can assign every (closed) formula a unique truth value (true or false) based upon the classical truth tables (e.g., $\mathsf{true} \wedge \mathsf{false} = \mathsf{false}$). A model for a predicate calculus formula is then an interpretation which makes the formula true.

The important notion of logical consequence $\models$ is then defined as $\alpha \models \beta$ if and only if every model of α is also a model of β.

2.1.2 Proof Theory

Model theory provides a formal way of assigning meaning to predicate calculus formulas, independent of any manipulation. However, it is of no use to actually formally establish any logical consequence $\alpha \models \beta$ in a mechanical way, as the number of domains and interpretations is usually infinite.

To be able to reason about logical consequence, we turn to proof theory, which provides axioms and inference rules. Formally, we denote by $\alpha \vdash \beta$ that we can prove that β is true if we assume α to be true.

A particular inference system $\vdash$ is said to be *sound* if $\alpha \vdash \beta$ implies $\alpha \models \beta$, and *complete* if $\alpha \models \beta$ implies $\alpha \vdash \beta$.

Predicate calculus is semi-decidable, meaning that whenever $\alpha \models \beta$ we can actually prove $\alpha \vdash \beta$ in finite time [46], but if $\alpha \not\models \beta$ we might actually not be able to establish $\alpha \not\vdash \beta$. Furthermore, by Gödel's famous incompleteness result [45], predicate calculus is incomplete, in the sense that any theory containing arithmetic on the natural numbers cannot be captured. One cannot even restrain a theory to just have the natural numbers as model (see Corollary 4.10.1 in [31]).

2.2 The lambda-calculus

The λ-calculus is a logical theory that was developed by Alonzo Church in the 1930's to answer questions on computability. The theory has just two operations, function application and function abstraction. Yet the λ-calculus is powerful enough to express all computable functions, and to study all aspects of programming in their purest form.

The syntax of the untyped λ-calculus [11] is given by two syntactic categories: variables $\mathcal{V}$ (with v, w ranging over $\mathcal{V}$), and expressions $\mathcal{E}$ (with e, f ranging over $\mathcal{E}$).

$$\begin{aligned} \mathsf{v}, \mathsf{w} \in \mathcal{V} &= \mathsf{a} \mid \mathsf{b} \mid \mathsf{c} \mid \ldots \\ \mathsf{e}, \mathsf{f} \in \mathcal{E} &= \mathsf{v} \mid \lambda\ \mathsf{v}\ \cdot\ \mathsf{e} \mid \mathsf{f}\ \mathsf{e} \end{aligned}$$

Here $\lambda\ \mathsf{v}\ \cdot\ \mathsf{e}$ denotes the creation of a function with argument v and function 'body' e, and $\mathsf{f}\ \mathsf{e}$ denotes the application of a function f to the argument e.

The semantics of the untyped λ-calculus is given by a reduction relation over expressions. This reduction relation is usually specified in a natural deduction style, by a set of rules and axioms. The most important of these is the β-reduction rule, which specifies how a function f is applied to its argument e:

$$[\beta]\ \frac{\mathsf{f} \rightarrow \lambda\ \mathsf{v}\ \cdot\ \mathsf{f}' \qquad \mathsf{e} \rightarrow \mathsf{e}'}{\mathsf{f}\ \mathsf{e} \rightarrow \mathsf{f}'\ [\ \mathsf{v}\ :=\ \mathsf{e}'\ ]}$$

This specification shows that:

- The evaluation of the function (expression f) must yield a λ-abstraction (i.e. of the form $\lambda\ \mathsf{v}\ \cdot\ \mathsf{f}'$).
- The evaluation of the argument expression e yields an expression e'.
- The result is an expression obtained by substitution of e' for every free occurrence of v in the expression f' (denoted by $[\ \mathsf{v}\ :=\ Ve'\ ]$).

2.2.1 Normal forms

The purpose of evaluating a λ-term is to arrive at a normal form, which is an expression that cannot be usefully evaluated further. A normal form is characterised by the fact that it does not contain occurrences of the form $(\lambda\ \mathsf{v}\cdot\mathsf{f})\ \mathsf{e}$.

To evaluate a λ-term one would repeatedly apply the β-reduction rule, until it can be applied no more. The result is then a normal form. It can be shown that normal forms are unique.

It is perfectly possible for a λ-term not to have a normal form. For example the term ω below has no normal form, because it reduces to ω itself:

$$\omega\ =\ (\lambda\ \mathsf{a}\ \cdot\ \mathsf{a}\ \mathsf{a})\ (\lambda\ \mathsf{a}\ \cdot\ \mathsf{a}\ \mathsf{a})$$

2.2.2 Reduction strategy

The rule β specifies that the argument e must be evaluated before the substitution. This is called *applicative order reduction*. It is also possible to substitute e unevaluated. This is called *normal order reduction*:

$$[\beta'']\ \frac{\mathsf{f} \to \lambda\ \mathsf{v}\ \cdot\ \mathsf{f}'}{\mathsf{f}\ \mathsf{e} \to \mathsf{f}'\ [\ \mathsf{v}\ :=\ \mathsf{e}\ ]}$$

Whether to use applicative order reduction or normal order reduction is important when considering termination. It can be shown that if there is a normal form, then it will be found with normal order reduction. However, applicative order reduction can usually be implemented more efficiently. The reader is referred to Peyton Jones for further details [92].

2.3 Set theory

Set theory forms a basis for a great deal of modern mathematics (some would say *all* of modern mathematics). Despite its power and breadth as a means of discussing mathematics, it is in essence simple, and is widely taught to children in elementary mathematics lessons. The central notions are that of set (a collection of things) and membership (the property of being one of the things in a given set).

The study of sets was begun by Georg Cantor (1845–1918). Early 20th century mathematicians discovered a number of problems with a naive consideration of set theory. Perhaps the most famous is Russell's paradox—if B is the set of all those sets which are not members of themselves, then is B a member of itself? As a result, modern set theory is usually presented axiomatically [36]. The most popular axiomatic system derives from work of Zermelo and Fraenkel, hence it is called 'ZF set theory'.

The axioms use the central ideas of sets and members to define other familiar features of sets, such as union and intersection, Cartesian product (ordered pairs) and the empty set, as well as more challenging concepts such as infinity.

For example when given two sets A and B the union $\mathsf{A} \cup \mathsf{B}$, intersection $\mathsf{A} \cap \mathsf{B}$, and the set of ordered pairs or Cartesian product $\mathsf{A} \times \mathsf{B}$ are defined as:

$$\mathsf{x} \in \mathsf{A} \wedge \mathsf{x} \in \mathsf{B} \leftrightarrow \mathsf{x} \in \mathsf{A} \cap \mathsf{B}$$
$$\mathsf{x} \in \mathsf{A} \vee \mathsf{x} \in \mathsf{B} \leftrightarrow \mathsf{x} \in \mathsf{A} \cup \mathsf{B}$$
$$\mathsf{x} \in \mathsf{A} \wedge \mathsf{y} \in \mathsf{B} \leftrightarrow (\mathsf{x}, \mathsf{y}) \in \mathsf{A} \times \mathsf{B}$$

Whilst a full axiomatic treatment of set theory is outside the scope of this article, it is by no means irrelevant to computing, and declarative languages in particular. Type systems, in particular, will most obviously be underpinned by set theory.

First-order logic usually contains a notion of *functions*, but in set theory it is common to describe functions differently, via their *graphs*. The graph of a function is a set of ordered pairs, the first element of each denoting a value in the function's domain, and the second element denoting that element of the range which the first is mapped to by the function. This shows that functions can be described within set theory, and without the need for additional notions.

3 Declarative languages

In this section we first present logic programming in relation to its mathematical basis the predicate calculus. Then Section 3.2 introduces functional programming in relation to the λ-calculus. Finally Section 3.3, discusses specification languages with respect to set theory.

3.1 Logic programming languages

Logic programming grew out of the insight that a subset of first-order logic, based on *Horn clauses*, has an efficient operational reading and can thus be used as the basis of a programming language.

3.1.1 Horn clause logic

A Horn clause is a logical formula of the form $\mathsf{A} \leftarrow \mathsf{B}_1 \wedge \mathsf{B}_2 \wedge \ldots \mathsf{B}_n$, where $\mathsf{A}, \mathsf{B}_1, \ldots \mathsf{B}_n$ are terms consisting of

symbols but no logical connectives. A *logic program* then is just a set of Horn clauses.

An important property of logic programs (and of clausal form formulas in general) is that they have a model if and only if they have a Herbrand model. These are restricted, syntactic models which map every term to itself. This insight [58] has lead to an efficient proof theory, which is sound, complete, and can be automated, based upon unification and resolution [97]. This in turn has lead in the beginning of the 70's to the development of linear SLD-resolution [72] and the logic programming language Prolog.

Unification matches up two terms a and b by finding the most general substitution θ instantiating all variables in a and b so that $\mathsf{a}\theta$ is identical to $\mathsf{b}\theta$. For example the unification of $\mathsf{deriv}(\mathsf{deriv}(\mathsf{X}+\mathsf{Y})\times\mathsf{Z})$ and $\mathsf{deriv}(\mathsf{A}\times\mathsf{B})$ would produce the substitution $\{\mathsf{A}/\mathsf{deriv}(\mathsf{X}+\mathsf{Y}), \mathsf{B}/\mathsf{Z}\}$.

Consider as an example the following Horn clause, which would be part of a logic program computing the derivative of a function:

$$\mathsf{deriv}(\mathsf{F}+\mathsf{G}, \mathsf{F}'+\mathsf{G}') \leftarrow \mathsf{deriv}(\mathsf{F}, \mathsf{F}') \wedge \mathsf{deriv}(\mathsf{G}, \mathsf{G}')$$

The Horn clause has a logical semantics, whose validity can be checked independently of the rest of the logic program. There is also a natural language translation of this clause:

> If F' is the derivative of F and G' is the derivative of G then $\mathsf{F}'+\mathsf{G}'$ is the derivative of $\mathsf{F}+\mathsf{G}$.

The Horn clause also has an operational reading, allowing for an efficient execution mechanism:

> To calculate $\mathsf{deriv}(\mathsf{F}+\mathsf{G}, \mathsf{F}'+\mathsf{G}')$ one should first calculate $\mathsf{deriv}(\mathsf{F}, \mathsf{F}')$ and then calculate $\mathsf{deriv}(\mathsf{G}, \mathsf{G}')$.

Using logic as the basis of a programming language also means that a uniform language can be used to express and reason about programs, specifications, databases, queries and integrity constraints. Also, because of their clear (and often simple) semantical foundations, declarative languages offer significant advantages for the design of semantics based program analysers, transformers and optimisers [53].

Logic programming languages allow non-determinism, making them especially well-suited for applications like parsing. They also provide for automatic memory management, thus avoiding a major source of errors in other programming languages. Another advantage of logic programming languages is that they can compute with partially specified data and that the input/output relation is not fixed beforehand. For instance, the above mentioned program can not only be used to compute the derivative of e.g. $\mathsf{F}\times\mathsf{G}$ via the query $\leftarrow \mathsf{deriv}(\mathsf{F}\times\mathsf{G}, \mathsf{H})$ it can (in theory) also be used to calculate its integral via the query $\leftarrow \mathsf{deriv}(\mathsf{E}, \mathsf{F}\times\mathsf{G})$.

Finally, although early logic programming languages were renowned for their lack of efficiency, the implementations have become more efficient, recent efforts reaching or even surpassing the speeds of imperative languages for some applications.

Good introductions to logic programming can be found in [6, 74, 87, 33, 7, 39].

3.1.2 Semantics, Negation as Failure, Non-monotonic Reasoning

Given that a program P is just a set of formulas, which happen to be clauses, the logical meaning of P might be seen as all the formulas F for which $\mathsf{P} \models \mathsf{F}$. However, in logic programming one (usually) assumes that the program gives a *complete* description of the intended interpretation, i.e., anything which cannot be inferred from the program is assumed to be false. This is the so-called *closed world assumption*. When given a formula F, the closed world assumption states that $\neg\mathsf{F}$ is a logical consequence of a program P if F is not a logical consequence of P. This means that, from a logic programming perspective, the program below captures exactly the natural numbers:

$$\begin{array}{ll} \mathsf{nat}(0) & \leftarrow \\ \mathsf{nat}(\mathsf{succ}(\mathsf{X})) & \leftarrow \mathsf{nat}(\mathsf{X}) \end{array}$$

This is impossible to accomplish within predicate calculus alone.

Logic programs have been extended to also allow negations in the body of clauses and a multitude of semantics have been developed [8]. All of this makes logic programming an ideal setting for certain applications in artificial intelligence, such as non-monotonic reasoning or abduction. It also provides an elegant solution to the so-called frame problem.

3.1.3 Meta-programming

In essence, meta-programming is the art of treating *programs as objects*, which can be modified and manipulated mechanically by another program, the meta-program. Usually the object and meta-program are supposed to be written in (almost) the same language. It turns out that logic programming is especially suited for Meta-programming [9] and a great deal of research has been done on that issue. Meta-programming has many applications (extending the programming language, debugging, program analysis, program transformation,...) and is also relevant in areas such as multi-agent systems, so as to reason about other agents and their (possible) behaviour.

3.1.4 Prominent logic programming languages

The five main families of logic programming languages are:

- pure first-order languages. These languages try to remain within the logical basis as much as possible. The programming language Gödel [59] is one prominent member of this family. Mercury [102] is a pure logic programming language which achieves a remarkable execution speed by exploiting mode and type information of the program.

- There has recently been considerable interest in developing integrated functional/logic languages leading to the development of such languages as Escher, Curry, λ-Prolog, ALF, Babel. These languages keep (most of) the advantages of logic programming languages while adding support for (higher-order) functions.

- (impure) Prolog and its descendants. Many variants and implementations of Prolog exists, all with slight variation in functionality and syntax. This has prompted the development of an international ISO Standard for Prolog [32].

- Constraint Logic Programming (CLP) languages. An important discovery was that the resolution and unification process of logic programming could be extended to handle constraints over arbitrary domains. This has lead to many successful applications in industry and the development of CLP languages such as CHIP, Prolog III and IV, and Eclipse.

- Concurrent logic languages. Another important discovery was that logic programming languages provide an elegant way of describing and developing concurrent or distributed systems: logical variables can be seen as communication channels and communication and synchronisation occurs via instantiation of such variables. This insight was the basis of the Japanese "Fifth-Generation Computer Systems" research initiative. Some languages of this type are Parlog, Concurrent Prolog, FGHC, AKL (Andorra Kernel Language), Janus, KL1, and KLIC.

3.2 Functional programming languages

A functional program is a set of equations. To 'execute' such a program means to solve these equations. The implementation automatically derives the solution via the β-reduction processes. Functional programmers do not write equations in the λ-calculus, because this would be cumbersome. Instead one uses high level notations, which can be translated into the λ-calculus.

Below is a function take, which when given a natural number n, and a list xs, returns the first n elements of the list. The function is defined by three equations, numbered $\{1\}, \{2\}, \{3\}$. Only one of these equations applies at any one time. The empty list is denoted by []. The notation x : xs indicates a list which differs from the list xs in that the former has one more element in front. The first equation matches only when $n = 0$. The second and third match when $n > 0$.

```
take 0       xs      = []                {1}
take (n + 1) []      = []                {2}
take (n + 1) (x : xs) = x : take n xs    {3}
```

3.2.1 Referential transparency

The hallmark of functional programming is that variables are treated as in mathematics. Once a variable is associated with a value, it remains associated with that value; it never changes. This is called *referential transparency*. Languages which have this property are also called *pure*. Referential transparency makes it impossible to mutate data structures. This can be a problem because many algorithms rely on mutable data structures for efficiency. However, in the implementation, data structures may be mutated as long as the compiler is able to guarantee that the data structure is not shared.

3.2.2 Equational reasoning

The execution of a functional program can be represented as a sequence of steps, where an expression denoting a value may be replaced by another expression denoting the same value ("equals are always replaced by equals"). Here we show the calculation of an initial segment from a 5-element list. The steps are annotated on the right with the appropriate equation for take:

```
take 3 (1 : 2 : 3 : 4 : 5 : [])                {3}
     = 1 : take 2 (2 : 3 : 4 : 5 : [])         {3}
     = 1 : 2 : take 1 (3 : 4 : 5 : [])         {3}
     = 1 : 2 : 3 : take 0 (4 : 5 : [])         {3}
     = 1 : 2 : 3 : []                          {1}
```

The similarity between the above process and the usual mathematical practice of replacing equals by equals makes functional programming attractive for mathematically oriented applications.

3.2.3 Polymorphism

In the above example we have applied the function take to a list of numbers. However, take would work just as well when applied to, say, a list of strings.

```
take 3 ("one" : "two" : "three" : "four" : [])
     = "one" : "two" : "three" : []
```

We say that the function take is *polymorphic* in the element type of the list. The type of the function could be written as follows:

$$\mathsf{take} :: \mathbb{N} \to \alpha\ \mathbf{list} \to \alpha\ \mathbf{list}$$

This type judgement says that the function takes a natural number as its first argument, a list of any element type α as second argument, then produces a list with elements of the *same* type α.

3.2.4 Complexity

The function qsort below embodies the Quick sort algorithm. It uses list comprehensions of the form [F x | x ← xs, P x], which denote that when x ranges over the elements of the list xs, only those elements of the list xs that satisfy the predicate P will be offered to the function F for inclusion in the result. The operator ++ concatenates two lists.

```
qsort []       = []                                        {1}
qsort (x : xs) = qsort [y | y ← xs, y < x]                 {2}
                 ++ [x] ++
                 qsort [y | y ← xs, y ≥ x]
```

Sometimes normal order evaluation gives a different complexity to the more commonly used applicative order evaluation. For example, one of the remarkable properties of functional programming based on normal order evaluation is the fact that taking only the first element after sorting a list has linear complexity:

Calculation	Complexity
take 1 (qsort xs)	$\mathcal{O}(n)$
take n (qsort xs)	$\mathcal{O}(n \log n)$

3.2.5 Prominent functional languages

The three main families of functional programming languages are:

- Pure, lazy languages (based on normal order reduction) with the main representative Haskell [63].
- Pure, eager languages (based on applicative order reduction), represented by Standard ML [79]. Although Standard ML has an impure extension, the bulk of the language is pure.
- Lisp and its descendants such as Scheme [96] are eager languages that permit the use of mutable data structures.

Functional programming has many other interesting properties that we glossed over here. The reader is referred to one of the many text books [15, 119, 91, 61, 29, 114, 112, 24].

3.3 Specification languages

Specification languages are an area where programming meets mathematics. Just as engineers of many disciplines have long found value in constructing mathematical models of the systems they are to build, software engineers have begun to find it useful to construct precise descriptions of software before it is implemented. In the traditional engineering domains, these descriptions usually involve continuous mathematics and the differential and integral calculi; in software engineering, discrete mathematics, set theory and predicate calculus (or other logical systems) are more appropriate.

In both cases, the descriptions are declarative in that their focus is on an abstract statement of a solution, not on a process by which it is to be achieved. Software is unusual, however, in that declarative specifications may be executable. Moreover, even if the specifications are not directly executable, it is possible to transform them mathematically into programs. Whereas bridges are emphatically not bridge specifications, the line between software specifications and software products is blurred.

There are two main approaches to writing specifications of programs. The first is the algebraic approach, the second is the model based approach. Each will be described in subsequent sections.

3.3.1 Algebraic Specification

Algebraic specifications assume a simple vocabulary of named sets and total functions upon those sets. Specifications are written as axioms (usually equations) describing properties which those functions must have. Such equations may be regarded as a superset of those written in functional programming—by removing constraints (over what may appear at the left-hand side of an equation), executability may be lost, but a great freedom of expression is gained.

Algebraic specification gained popularity though the 1980s, giving rise to the languages Clear and OBJ3. A text on the subject is [47].

Process algebras are notations for describing concurrent computation and communication, at a high level. As with algebraic specification, the description is written using a series of equations which describe the operation of a process. Algebraic laws allow the effects of parallel composition, communication hiding, etc., to be calculated. The two leading process algebras are CSP [60] and CCS [78].

Below is a small CCS specification inspired by an example from Milners book [78]. It represents a vending machine selling chocolates to two customers. A full stop represents sequencing of actions, a plus separates two alternatives and a vertical bar introduces two concurrent processes. Agents may engage in named input and output actions. An output action is indicated by a bar, an

input action is without a bar. When one agent is willing to engage in an input action while another is willing to engage in an output action of the same name, communication takes place.

$$\begin{aligned}
\mathsf{Vending} &= \mathsf{insert2p}.\overline{\mathsf{collectbig}}.\mathsf{Vending} + \\
&\quad\ \mathsf{insert1p}.\overline{\mathsf{collectsmall}}.\mathsf{Vending} \\
\mathsf{Cust}_1 &= \overline{\mathsf{insert2p}}.\mathsf{collectbig}.\mathsf{Cust}_1 \\
\mathsf{Cust}_2 &= \overline{\mathsf{insert1p}}.\mathsf{collectsmall}.\mathsf{Cust}_2 \\
\mathsf{System} &= \mathsf{Cust}_1 \mid \mathsf{Cust}_2 \mid \mathsf{Vending}
\end{aligned}$$

The vending machine accepts either a 1p coin for a small chocolate (input action insert1p) or a 2p coin for a big chocolate. Once a coin has been accepted, the machine insists that the appropriate chocolate be dispensed before it accepts another coin. The first customer keeps purchasing big chocolates by engaging in the output action $\overline{\mathsf{instert2p}}$. Similarly, the second customer keeps purchasing small chocolates.

The π-calculus, a variant of CCS, has greater flexibility, but a greater learning overhead. CSP includes a notion of refinement, which makes it appropriate for the specification and development of large systems. Recently, model-checking tools for CSP have been used to great effect [98]. By permitting exhaustive exploration of the state space defined by a CSP process, the declarative description of a protocol, say, can be checked for properties such as deadlock-freedom which would be hard to establish using an implementation. The same tool checks refinement, so a process specification can be automatically tested against a desirable (or undesirable) property.

3.3.2 Model-based specification

The contrast between model-based specification and algebraic specification may at first sight be rather subtle, but in fact is quite profound. In a model-based specification, one constructs a description of the artifact (program, data type, etc.) by using simpler mathematical objects (sets, relations, functions). A model is constructed in that there is an abstract entity which denotes the artifact.

In practice, a model-based specification may contain many similar equations/axioms to an algebraic specification, but the emphasis is different: the model-based specification typically describes some abstract 'machine', whereas the algebraic specification is more usually concerned with properties of abstract data types.

The following section gives an example of a model based specification in the widely used Z notation.

The Z Notation The Z notation [104] began in the work of Abrial and others in Oxford, and has evolved over the last twenty years to cover a wide international community. To a large degree, it is simply a style for using first-order predicates and ZF set theory. Z's main additions to simple set theory are a strong type system reminiscent of certain functional languages, and a notion of schemas, through which specifications are structured. A programming-language style grammar ensures that various aspects of Z specifications (such as type-correctness) can be machine-checked. To facilitate convergence in such tools, an international standard for Z is under construction [85].

Z schemas serve many purposes. Usually, a schema is presented as a collection of variable declarations, together with some predicates describing fixed relationships between those variables. For example, the following schema describes a data file, as a sequence of bytes (modelled as natural numbers), together with a measure of the file size.

File

$$\begin{aligned}
&contents : \operatorname{seq} \mathbb{Z} \\
&size : \mathbb{N} \\
&\hline
&size = \#contents
\end{aligned}$$

The *size* component is redundant in that it is derived from the value of *contents*; however, it is often found to be useful to record redundant information in this way, for the succinct expression of specifications. The predicate part provides a state invariant which ensures that *size* always holds the correct value.

Depending on the specification being constructed, this $File$ might represent a data type (like a structure), or an item of system state, or indeed an object. The most common use of schemas in Z is in the specification of state-based systems. Having defined a file as above, we might proceed by declaring operations which append input data ($extra?$) to the file and read the contents of the file as an output ($data!$).

Append

$$\begin{aligned}
&\Delta File \\
&extra? : \operatorname{seq} \mathbb{Z} \\
&\hline
&contents' = contents \frown extra?
\end{aligned}$$

Read

$$\begin{aligned}
&\Xi File \\
&data! : \operatorname{seq} \mathbb{Z} \\
&\hline
&contents' = contents = data!
\end{aligned}$$

Append is a schema which denotes not the state of some system, but an operation upon that system. The notation $\Delta File$ indicates that the operation relates a 'before' state (with undecorated variables) and an 'after' state (in which variables carry a prime $'$) of the components of $File$. Similarly, $\Xi File$ indicates a similar relationship, but with the

added caveat that the components of $File$ do not change their values. $extra?$ is an input to the *Append* operation, and $data!$ is an output.

In a state-based system it is also important to define how the system is to be initialised. This may be described using a degenerate operation, which has an 'after' state, but no 'before' state:

InitFile

$File'$

$contents' = \langle\rangle$

Specifications in this style can be formally *refined* towards code by stepwise transformation. Such refinements usually end with imperative code, though other declarative languages are also a possibility. The usual expectation is that the state variables will become global variables in some program (or module, or class). The initialisation schema will be refined either by initialisation of the variables by the compiler, or by some explicit initialisation procedure (or method). The operation schemas will be refined by procedures (or functions, if they do not change the state, or methods) with appropriate inputs and outputs.

3.3.3 Executability

The similarity between certain specification notations and certain programming languages raises the possibility of executing the specifications (models) of systems. In almost every case, this is no substitute for system development, but might form part of a prototyping exercise.

There has been a long debate about the desirability of executing specifications. The original paper is [55]; the most recent contribution is [48]. On the one hand, it is argued that enabling the execution/animation of specifications is a useful tool in understanding a specification and communicating its contents to a non-mathematical reader. The counter-argument is that the purpose of formal specification is to achieve a clear description of *what* is to be done, without committing on *how* this will work. If the specifier has even half an eye to the possible execution of their specification, the result will tend to be a compromise on the clarity.

3.3.4 Prominent specification languages

The Z notation as introduced above is the most widely used specification language to date.

The Vienna Development Method [67] was initially the work of a group concerned with formal semantics and compiler design, at IBM laboratories in Vienna. As the name implies, VDM embraces a means of program development, with rules of proof for program design steps. Our interest, however, is in VDM-SL, the specification language of VDM. VDM-SL has much in common with the Z notation [56], and like Z it has been the subject of an international standardisation effort [64].

VDM-SL differs from Z in concrete syntax (it lacks the boxes, and uses keywords instead). Semantically it differs in using a three-valued logic of partial functions and a domain-theoretic semantics. Methodologically, it differs in the status of state invariants, and in having explicit imperative programming constructs in the specification language.

Promela is the specification language used by the widely used Spin model checker [62]. The language supports dynamic creation of concurrent processes and both synchronous and asynchronous message passing.

B is a model-oriented specification notation [3] whereby a system is specified in terms of an explicit abstract model of the state along with operations on the state. It is based on set theory and the weakest pre-condition calculus. The B-toolkit provides a comprehensive package of tools for animation, proof, refinement and implementation of descriptions [84].

3.4 Other Languages

Many languages and notations exists which have a large declarative subset and/or which are used in a way similar to the functional, logical and specification languages introduced above. Of particular interested from the point of view of their use in education are the following.

ISETL is a programming language specifically created to support courses in Discrete Mathematics. ISETL has a large declarative core, offering all the necessary ingredients for experimentation with concepts from Discrete Mathematics, such as finite sets, sequences, relations, first and higher order functions, existential and universal quantifiers, and ZF-expressions.

MATLAB [111], Maple [44], and Mathematica [121] are systems/languages for computer based mathematical calculations in Science and Engineering. Each contains a large declarative subset used to express the mathematical operations for which these systems have been designed.

4 Teaching Declarative Languages

Having introduced the mathematical bases and the major declarative languages, we now consider where declarative languages are used in the curriculum. In this section we briefly look at the ways in which declarative languages are taught as topics of interest in their own right. The next section looks at the role of declarative languages in other parts of the curriculum.

4.1 Logic Programming

Logic programming languages have been used as the first language taught in universities or even schools [37]. While Prolog used to be the only choice for this, recent years have seen increased popularity of purer logic programming languages, such as Gödel [59] and Mercury [102].

Many books exist that provide good introductions to logic programming and Prolog. Classical texts [23, 22] mainly concentrated on the programming aspects. More recent books more and more cover the logical side of Prolog [109, 108, 90, 25, 87, 7, 33, 74].

A recent development is ToonTalk [69] which, in the spirit of Logo, tries to teach children as young as 4 years to program by associating graphics, animation, and sound with logic programs.

4.2 Functional Programming

A functional language is close to discrete mathematics, thus making it an ideal vehicle to teach the principles of programming in a first programming course. With a minimum of syntactic burden, it is possible to teach the fundamental tools of recursion and induction. The student should be capable of writing abstract data types for lists, queues and trees in just a few weeks. Many course have been developed on the basis of this idea. Some are targeted at a sophisticated audience [15], others are targeted at a more youthful audience [41]. There are many other good texts [1, 119, 91, 61, 29, 114, 112, 24].

4.3 Specification

In most UK universities formal methods has become a standard subject in the computer science undergraduate syllabus. In particular, use of Z is widely taught, and many text books are available [3, 65, 94, 103, 124]. Often, this is a vehicle for the teaching of discrete mathematics, so students come to regard the subject as 'hard' and more theoretical than practical.

Students with the benefit of greater perspective — on graduate courses, or returning to study after time in industry — have been found to be able to use Z to better effect, seeing it as a modelling tool with great practical benefit. One of the texts [122] comes from the experience of many years teaching Z to industrial practitioners.

5 Using Declarative Languages in Teaching

In this section we visit each of the nine subject areas defined by the 1991 ACM/IEEE Computer Science curriculum [113] to discuss whether and how declarative languages are being used to support teaching. Before this we consider Discrete Mathematics and Logic.

In considering each of these subject areas, we report briefly on the use of each of the three declarative languages that we have identified: logic programming, functional programming and specification.

5.1 Discrete Mathematics

Wainwright [117] describes laboratory assignments using a functional language to support a discrete mathematics course. The author reports that even without prior exposure to programming, all students found the laboratory assignments useful and recommend that they be kept as part of the course. In a later experiment Schoenefeld and Wainwright [99] teach discrete mathematics using the book by Skiena [101], which offers extensive laboratory assignments based on Mathematica. Again the participants were "pleased that Mathematica was integrated into the course". The use of Mathematica as an exploratory tool makes operational important mathematical concepts such as abstraction and generalisation.

Cousineau and Mauny [24] use ML to create aesthetically pleasing drawings in order to explain various geometric concepts, such as tiling. The images generated resemble M.C. Escher's engravings. The same book also uses ML to support the presentation of exact arithmetic over natural, integer and rational numbers.

One of the text books on ISETL [38] encourages the students to experiment with small ISETL programs, to try and discover the underlying mathematical concepts. Virtually all ISETL examples given in the book are purely declarative. In some cases the students are asked to create declarative solutions to problems for which an imperative solution is given. The 'ISETL method' is reported to be used at over 75 colleges throughout the world [35].

5.2 Logic

Declarative languages are directly based on a logic, and declarative programs can be viewed as executable logics. This makes declarative languages suitable to support a course on logic.

Because of its logical basis, Prolog is an ideal support language for mathematical logic and also many fields in discrete mathematics. Automated theorem provers can be built easily in Prolog and many concepts in logic programming are important in mathematics as well, so that a joint study is highly beneficial [39, 12].

Hein [57] describes tools to teach discrete mathematics and logic using a large variety of short and relevant laboratory assignments. Simple assignments are described to experiment with laws of logic (using Prolog). Other assignments support learning about functions and function composition using functional languages.

Since Z incorporates classical logic and set theory, its notation provides a natural way to teach these topics to computing students and software engineers [123].

In his book, Downward [34] explores the connection between logic and declarative programming in four different ways:

- Logic programming is related to first order logic.
- Many sorted logic is the foundation of abstract data types. The OBJ2 [42] term rewrite system supports this logic directly.
- Domain theory is used to establish models for the λ-calculus, which in turn is the basis of functional programming.
- Intuitionistic logic is shown to be the basis of the theory of types underlying all modern functional languages.

Throughout the book, Downward encourages the reader to write programs in the different programming languages to experiment with the concepts being studied.

5.3 Algorithms and Data structures

Teaching algorithms and data structures using a purely functional languages is a challenge. The main problem is that many data structures and algorithms rely on mutable data structures for efficiency. Some courses have been taught using a functional language (Standard ML), relying on language extensions that support mutable data structures [51]. This is less than satisfactory because the advantage of a functional programming language is lost, i.e. to be able to use equational reasoning.

The work of Okasaki represents an important step in that it develops a wealth of data structures in Standard ML [89], without using the impure extensions. The topics discussed include single and double ended queues, binary search trees, heaps, tries and random access lists, all in a number of important varieties. The key idea is that good amortised complexity is at least as important as worst case complexity. The book is neither a straight replacement for classical texts such as Sedgewick's Algorithms and Data Structures, nor does it claim to be such a replacement.

Another text for advanced courses on data structures and algorithms is Bird & de Moor [14]. This uses a purely declarative style—its approach is compatible with functional programming, although the methods used also require a little category theory.

More accessible but less rigorous is the book by Rabhi and Lapalme [95], who cover many important algorithms, such as sorting, searching, graph algorithms, divide-and-conquer, dynamic programming, and the traveling salesperson. Most algorithms are accompanied by a worst case space and time complexity bound, unfortunately mostly without a derivation. The emphasis of Rabhi and Lapalme is on the high level specification of the algorithms, often with a fold/unfold style program transformation to improve the efficiency. The book makes good use of the facilities provided by a modern functional programming language (Haskell) in showing how classes of algorithms can be represented in the language as higher order functions. The drawback of the high-level approach is that the space complexity of the resulting algorithms is often worse than that obtained with imperative methods. In addition, to understand the space complexity considerable knowledge is required of the implementation of functional languages. The proposed complexity analysis method does not apply to higher order functions nor to essentially lazy algorithms.

5.4 Architecture

Some authors teach abstract machines in terms of declarative programs. Prolog [26] even enables 'reverse execution' (i.e. given a finite state machine and its output, the Prolog system will reconstruct the possible inputs). Piotrowski [93] observes that Miranda is less versatile, but offers more elegant and robust (typed) specifications of abstract machines. Both of these declarative approaches to experimenting with abstract machines offer readily executable specifications of abstract machines.

The Marktoberdorf summer schools often consider topics related to declarative languages [18]. In the cited volume, chapters by M. P. Fourman and A. J. Martin describe digital circuits in this way, at considerable levels of detail—in Martin's case, as the basis for synthesis of VLSI circuits. Similar material has been included in undergraduate curricula [88].

5.5 Artificial Intelligence and robotics

Together with Lisp, Prolog is arguably the favourite language for teaching artificial intelligence. Many books on artificial intelligence are based on Prolog [17, 100, 70, 75, 40]. Prolog is especially well suited to implement expert systems [16], planners and problem solving [73].

Logic programming has been used with success as a sound basis for Machine Learning, leading to a whole new field of *Inductive Logic Programming* [82, 30, 13]

Within *Natural Language Processing*, logic programming also plays a major role. For example, the Definite Clause Grammars (DCG) provided by Prolog have applications in parsing and natural language processing [19, 77].

5.6 Data Base and information retrieval

There is a tight link between relational/deductive databases and logic programming. Indeed, from a theoretical point of view, a relational or deductive database can be viewed as a special kind of logic program. This is reflected in the common semantical foundation and a whole series of books [43, 49, 80, 27, 20, 21].

Furthermore, certain versions of Prolog (e.g., XSB-Prolog) can be used as efficient database engines, allowing the programmer to get the best of both worlds.

Queries of databases, such as those of the widely used language SQL [28] often have a large declarative core. This is because query languages provide an abstraction away from the manner in which the database system computes the results to a query. However, updates to data bases cannot be regarded in the same declarative way.

5.7 Human Computer Communication

Descriptions of the computer's role in Human Computer Communication are amenable to a declarative treatment. For example the construction of a graphical user interface in a modern windowing toolkit takes place at a level of abstraction where one declares components to be in a particular relationship (e.g. adjacent, or laid out on a grid). The windowing system then works out the details (e.g. order and placement) of instantiating the components.

Recent developments around XML [118] indicate that here again the basic paradigm is declarative. One describes structures and their relationships, without actually worrying about the details of rendering and placement. SMIL [116] is an XML based declarative language for hypermedia presentations on the Web. It describes relationships in time as well as in space.

5.8 Numerical and symbolic computation

Since the early 1960's dozens of symbolic mathematics packages have been developed. Because of the close ties with mathematics most of these packages have a large declarative core. At the time of writing three packages dominate the market: Maple [44], which is particularly strong in symbolic calculation, MATLAB [111], which is primarily devoted to numerical computation and Mathematica [121], which aims to combine both of these. For each of these packages many hundreds of books are available on topics ranging from financial mathematics to scientific computation. Many of these books have been written specifically to support teaching in specialist areas in science, and engineering.

5.9 Operating systems

Many of the early published case studies in using Z [54] cover operating systems topics—perhaps unsurprisingly as the transaction processing domain was the first major application of Z. These allow the Z-literate student to gain a detailed understanding of what, for example the UNIX filing system does, without needing to be concerned with any implementation details. The alternative is to teach using a large piece of production code, which means it is difficult to abstract away from low-level detail, or to teach using a specially-crafted operating system (which may entail more work than writing a specification of a real one).

Such an approach also lends itself to easy prototyping. A Z description of virtual memory page replacement policies is readily turned into a declarative program, with which one can run simulations to discover the effects of various patterns of memory usage.

5.9.1 Distributed systems

Hoare's book on CSP [60] includes implementation ideas using Lisp. It provides a thorough course on the more theoretical aspects of distributed and concurrent computation, including processes, synchronisation, communication, deadlock, and live lock. A more practical approach is taken by [76], which is also presented as a course textbook. It uses CSP-style models to provide a high-level description of processes, and presents them alongside Java programs which implement them, using Java threads to simulate concurrency.

5.10 Programming Languages

Programming languages is a large area, ranging from the activity of programming itself, to the implementation and theory of programming languages.

5.10.1 Imperative programming

Declarative specifications written in Z or some similar notation may be refined into imperative programs—that is, transformed using mathematical laws so that the implemented program provably achieves the specified outcome. A detailed methodology for doing this is explained in [81], which has been used as an introductory programming text at Oxford University for several years. Refinement within Z is covered in [122], which also arose from extensive teaching experience. The B method [3] (and associated tools) includes similar material, though industrially-oriented, and also features in software engineering education.

It is also possible to use an introduction of a few weeks with a purely functional language to give the students

enough confidence and knowledge of the principles of programming to be able to start programming in an imperative language. Hartel & Muller have developed a method whereby the main tool is transformation of tried and tested functional programs (using Standard ML) into C programs [52].

5.10.2 Compilers

Appel [5] shows that a declarative approach to compiler design is possible. The book encourages the student to write a complete compiler in Standard ML, using the ML versions of lex and yacc. The book discusses all the major topics from a compilers course, including lexical analysis, parsing, semantic analysis, code generation, runtime systems, register allocation and code optimisation. In addition the text covers topics like garbage collection in more detail than a standard text, such as Aho, Sethi and Ullman [4].

Recursive descent parsing is easy to implement in Prolog by using definite clause grammars [106] and Prolog is a good language to develop compiler prototypes. In fact, the first compiler for the successful telecommunications language Erlang was written in Prolog.

Finally, as declarative languages are amenable to refined static analysis [2] and partial evaluation, one can achieve the automatic generation of (prototype) compilers from interpreters [68].

5.10.3 Semantics

The study of the semantics of programming languages is heavily based on branches of discrete mathematics, such as λ-calculus, predicate logic, set theory, domain theory and category theory. A functional language is thus appropriate to experiment with the static and dynamic semantics of programming languages. Kirkerud [71] contains a chapter explaining how the denotational semantics of a simple imperative language can be represented in Standard ML. Nielson and Nielson [86] provide a number of appendices showing how both operational semantics and denotational semantics can be represented in Miranda. Winksel [120] recommends using Prolog, Standard ML or Miranda to "enliven the treatment of operational semantics".

Stepney [106] uses Prolog to implement the semantics of a programming language first specified in Z. Prolog is an ideal language for rapid prototyping and unification together with non-determinism allow the easy implementation of varying semantics for many different programming languages or formalisms.

5.11 Software Methodology and Engineering

The absence of types and modules means that classical Prolog is not well suited for software engineering purposes. Nonetheless, more recent logic programming languages such as Mercury and Gödel support these features and are being used to teach software engineering principles. Also, the logical nature coupled with the simple semantical model, make logic programming languages especially well suited for applying techniques such as formal verification, synthesis of provably correct programs [66] or declarative debugging.

It can be hard to motivate Software Engineering teaching in the classroom, because the size of the systems which can be analysed tend not to show up the problems that Software Engineering techniques are designed to solve. This is especially true in the teaching of notations like Z. Nevertheless, the benefit of using Z to discuss such issues is that it does represent best industrial practice — as witnessed by texts like [10, 65, 124, 107], all of which come from industrial authors.

6 Summary

It is clear from the discussion above that declarative languages can be, and indeed are, widely used in education. Some areas are in fact completely dominated by declarative languages, such as formal methods in software engineering, and artificial intelligence. Other areas, such as operating systems rely less on specific declarative ideas. However as understanding of Computer Science increases, the formal basis for such subjects is growing, and with it the need to teach in a declarative fashion.

We conclude the article with an extensive bibliography, from which we hope the interested reader can find support for teaching with declarative languages and ideas.

Acknowledgements

We thank Simon Cox and Dave DeRoure for their help in preparing the article.

References

[1] H. Abelson and G. J. Sussman. *Structure and interpretation of computer programs.* MIT Press, Cambridge, Massachusetts, 1985.

[2] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Ltd, 1987.

[3] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge Univ. Press, UK, 1996.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, techniques, and tools*. Addison Wesley, Reading, Massachusetts, 1986.

[5] A. W. Appel. *Modern compiler implementation in ML – basic techniques*. Cambridge Univ. Press, UK, 1997.

[6] K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 10, pages 495–574. North-Holland Amsterdam, 1990.

[7] K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.

[8] K. R. Apt and Roland N. Bol. Logic programming and negation: A survey. *The Journal of Logic Programming*, 19 & 20:9–72, May 1994.

[9] K. R. Apt and F. Turini. *Meta-logics and Logic Programming*. MIT Press, 1995.

[10] R. Barden, S. Stepney, and D. Cooper. *Z in Practice*. BCS Practitioner Series. Prentice Hall, 1994.

[11] H. P. Barendregt. *The lambda calculus, its syntax and semantics*. North Holland, Amsterdam, Amsterdam, The Netherlands, 1984.

[12] M. Ben-Ari. *Mathematical Logic for Computer Science*. Prentice Hall, 1993.

[13] F. Bergadano and D. Gunettie. *Inductive Logic Programming: From Machine Learning to Software Engineering*. MIT Press, 1995.

[14] R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice Hall International, Hemel Hempstead, 1997.

[15] R. S. Bird and P. L. Wadler. *Introduction to functional programming*. Prentice Hall, New York, 1988.

[16] K. A. Bowen. *Prolog and Expert Systems*. McGraw Hill, 1991.

[17] I. Bratko. *Prolog Programming for Artificial Intelligence (2nd Edition)*. Addison Wesley, 1990.

[18] M. Broy, editor. *Deductive Program Design*, volume 152 of *NATO ASI Series F*. Springer-Verlag, 1996. Proceedings of the NATO Advanced Study Institute on Deductive Program Design, held in Marktoberdorf, Germany, July 26 – August 7, 1994.

[19] M. A. Cavington. *Natural Language Processing for Prolog Programmers*. Prentice Hall, 1993.

[20] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, June 1990.

[21] J. Chomicki and Gunter Saake, editors. *Logics for Databases and Information Systems*. Kluwer Academic Publishers, March 1998.

[22] W. Clocksin. *Clause and Effect*. Springer-Verlag, Berlin, Sep. 1997.

[23] W. Clocksin and C. Mellish. *Programming in Prolog (Third Edition)*. Springer-Verlag, 1987.

[24] G. Cousineau and M. Mauny. *The Functional approach to programming*. Cambridge Univ. Press, UK, 1998.

[25] M. A. Covington, D. Nute, and A. Vellino. *Prolog Programming in Depth*. Prentice Hall, 1996.

[26] D. Crookes. Using Prolog to present abstract machines. *ACM SIGCSE bulletin*, 20(3):8–12, Sep 1988.

[27] Subrata Kumar Das. *Deductive Databases and Logic Programming*. Addison-Wesley, Wokingham, 1992.

[28] C. J. Date. *An introduction ro data base systems*. Addison Wesley, Reading, Massachusetts, sixth edition, 1995.

[29] A. J. T. Davie. *An introduction to functional programming systems*. Cambridge Univ. Press, UK, 1992.

[30] L. De Raedt. *Interactive Theory Revision: An Inductive Logic Programming Approach*. Academic Press, 1992.

[31] M. Denecker. *Knowledge Representation and Reasoning in Incomplete Logic Programming*. PhD thesis, Department of Computer Science, K.U.Leuven, 1993.

[32] P. Derensart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard, Reference Manual*. Springer-Verlag, 1996.

[33] K. Doets. *From Logic to Logic Programming*. MIT Press, 1994.

[34] M. Downward. *Logic And Declarative Language*. Taylor and Francis Inc, London, UK, Jul 1998.

[35] E. Dubinsky. ISETL: A programming language for learning mathematics. *Communications in Pure and Applied Mathematics*, 48:1–25, 1995.

[36] H. B. Enderton. *Elements of Set Theory*. Academic Press, 1977.

[37] R. Ennals. *Teaching logic as a computer language in schools*. Department of Computing, Imperial College, April 1983. First International Logic Programming Conference, Marseilles 1982.

[38] W. E. Fenton and E. Dubinsky. *Introduction to Discrete Mathematics with ISETL*. Springer-Verlag, Berlin, 1996.

[39] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 1990.

[40] P. A. Flach. *Simply Logical: Intelligent Reasoning by Example*. J. Wiley & Sons, 1994.

[41] D. P. Friedman, M. Felleisen, and D. Bibby. *The Little Schemer*. MIT Press, Cambridge, Massachusetts, Jan 1996.

[42] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer:. Principles of OBJ2. In *12th Int. Conf. Principles of programming languages (POPL)*, pages 52–66, New Orleans, Louisiana, Jan 1985. ACM, New York.

[43] H. Gallaire and J. Minker, editors. *Logic and Databases*. Plenum, New York, 1978.

[44] W. Gander and J. Hrebicek. *Solving Problems in Scientific Computing Using Maple and MATLAB*. Springer-Verlag, Berlin, third edition, 1997.

[45] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[46] K. Gödel. The completeness of the axioms of the functional calculus of logic. In J. van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, pages 582–591. Harvard University Press, 1967.

[47] J. Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.

[48] A. M. Gravell and P. Henderson. Executing formal specifications need not be harmful. *IEE/BCS Software Engineering Journal*, 11(2):104–110, March 1996.

[49] P. Gray. *Logic, Algebra and Databases*. Ellis Horwood, Chichester, 1984.

[50] D. Gries. Education: The need for education in useful formal logic. *Computer*, 29(4):29–30, April 1996.

[51] R. Harrison. *Abstract data types in Standard ML*. John Wiley & Sons, Chichester, UK, 1993.

[52] P. H. Hartel and H. L. Muller. *Functional C*. Addison Wesley Longman, Harlow, UK, 1997. `www.awl-he.com/ computing/ titles/ 0-201-41950-5.html`.

[53] J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors. *Partial Evaluation: Practice and Theory*, LNCS 1706. Springer-Verlag, 1999.

[54] I. J. Hayes, editor. *Specification Case Studies*. Prentice Hall International Series in Computer Science, 2nd edition, 1993.

[55] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *IEE/BCS Software Engineering Journal*, 4(6):330–338, November 1989.

[56] I. J. Hayes, C. B. Jones, and J. E. Nicholls. Understanding the differences between VDM and Z. *FACS Europe*, Series I, 1(1):7–30, Autumn 1993.

[57] J. L. Hein. A declarative laboratory approach for discrete structures, logic and computability. *ACM SIGCSE bulletin*, 25(3):19–24, Sep 1993.

[58] J. Herbrand. Investigations in proof theory. In J. van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, pages 525–581. Harvard University Press, 1967.

[59] P. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[60] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall Intenational, 1985.

[61] I. Holyer. *Functional programming with Miranda*. Pitman publishing, London, UK, 1991.

[62] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[63] P. Hudak, S. L. Peyton Jones, and P. L. Wadler (eds.). Report on the programming language Haskell – a non-strict purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5):R1–R162, May 1992.

[64] ISO. *Information Technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language*. ISO/IEC JTC1/SC22/WG19-VDM, 1996. 13817-1.

[65] J. Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.

[66] J.-M. Jacquet. *Constructing Logic Programs.* Wiley, Chichester, 1993.

[67] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall Intenational, second edition, 1990.

[68] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[69] K. Kahn. From Prolog and Zelda to ToonTalk. In Danny De Schreye, editor, *Proceedings of the 1999 International Conference and on Logic Programming*, pages 67–76, Las Cruces, New Mexico, 1999. MIT Press.

[70] S. H. Kim. *Knowledge Systems Through Prolog.* Oxford University Press, 1992.

[71] B. Kirkerud. *Programming language semantics.* Int. Thomson computer press, London, UK, 1997.

[72] R. Kowalski. Predicate logic as a programming language. In *Proceedings IFIP Congress*, pages 569–574. IEEE, 1974.

[73] R. Kowalski. *Logic for Problem Solving.* North Holland, 1979.

[74] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[75] G. F. Luger and W. A. Stubblefield. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving (3rd Edition).* Addison-Wesley, 1997.

[76] J. Magee and J. Kramer. *Concurrency: state models & Java programs.* J. Wiley & Sons, Chichester, England, 1999.

[77] C. Matthews. *An Introduction to Natural Language Processing Through Prolog.* Addison Wesley, 1998.

[78] R. Milner. *Communication and concurrency.* Prentice Hall, Hemel Hempstead, UK, 1989.

[79] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[80] J. Minker, editor. *Foundations of Deductive Databases and Logic Programming.* Morgan-Kaufmanm, September 1987.

[81] C. C. Morgan. *Programming from Specifications.* Series in Computer Science. Prentice-Hall International, second edition, 1994.

[82] S. Muggleton. *Inductive Logic Programming.* Academic Press, 1992.

[83] J. P. Myers. The central role of mathematical logic in computer science. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 22(1):22–26, February 1990.

[84] D. S. Neilson and I. H. Sørensen. The B-technologies: a system for computer aided programming. In U. H. Engberg, K. G. Larsen, and P. D. Mosses, editors, *6th Nordic Workshop on Programming Theory*, pages 18–35. BRICS, Denmark, Oct 1994.

[85] J. Nicholls, editor. *Z Notation.* Z Standards Panel, ISO Panel JTC1/SC22/WG19 (Rapporteur Group for Z), 1995. Version 1.2, ISO Committee Draft; CD 13568.

[86] H. R. Nielson and F. Nielson. *Semantics with applications: A formal introduction.* John Wiley & Sons, Chichester, UK, 1991.

[87] U. Nilsson and J. Małuszyński. *Logic, Programming and Prolog.* Wiley, Chichester, 1990.

[88] J. O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in hydra. In P. H. Hartel and M. J. Plasmeijer, editors, *Functional programming languages in education (FPLE), LNCS 1022*, pages 195–214, Nijmegen, The Netherlands, Dec 1995. Springer-Verlag, Berlin.

[89] C. Okasaki. *Purely Functional Data structures.* Cambridge Univ. Press, UK, 1998.

[90] R. A. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.

[91] L. C. Paulson. *ML for the working programmer.* Cambridge Univ. Press, UK, 1991.

[92] S. L. Peyton Jones. *The implementation of functional programming languages.* Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[93] J. A. Piotrowski. Abstract machines in Miranda. *ACM SIGCSE bulletin*, 21(3):44–47, Sep 1989.

[94] B. F. Potter, J. E. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice Hall International Series in Computer Science, 2nd edition, 1996.

[95] F. Rabhi and G. Lapalme. *Algorithms, A functional programming approach.* Addison Wesley, Reading, Massachusetts, Apr 1999.

[96] J. A. Rees and W. Clinger. *Revised^4 Report on the Algorithmic Language Scheme*. MIT, Cambridge, Massachusetts, Nov 1991.

[97] J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, Jan 1965.

[98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Europe, 1998.

[99] W. Schreiner. Parallel functional programming — an annotated bibliography. Technical Report 93-24, RISC-Linz, Johannes Kepler Univ, Linz, Austria, May 1993.

[100] Y. Shoham. *Artificial Intelligence Techniques in Prolog*. Morgan Kaufmann, 1994.

[101] S. Skiena. *Implementing discrete mathematics – Combinatorics and Graph theory with Mathematica*. Addison Wesley, Reading, Massachusetts, 1990.

[102] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of mercury: an efficient purely declarative logic programming language. *J. Logic Programming*, 29(1-3):17–64, Oct -Dec. 1996.

[103] J. M. Spivey. *The Z notation*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[104] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[105] O. Štěpánková and P. Štěpánek. AI education and logic. In Z. Mařík, V.; Štěpánková, O.; Zdráhal, editor, *Proceedings of the CEPES-UNESCO International Symposium on Artificial Intelligence in Higher Education*, volume 451 of *LNAI*, pages 199–205, Prague, CSFR, October 1989. Springer Verlag.

[106] S. Stepney. *High integrity compilation: A case study*. Prentice Hall, Hemel Hempstead, UK, 1993.

[107] S. Stepney, R. Barden, and D. Cooper, editors. *Object Orientation in Z*. Workshops in Computing. Springer-Verlag, 1992.

[108] L. Sterling. *The Practice of Prolog*. MIT Press, 1990.

[109] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[110] The ASL Committee on Logic and Education. Guidelines for logic education. *The Bulletin of Symbolic Logic*, 1(1):4–7, March 1995.

[111] The MathWorks, Inc. *The Student Edition of MATLAB 5 User's Guide*. Prentice Hall, Hemel Hempstead, UK, 1997.

[112] S. Thompson. *Haskell : The Craft of Functional Programming*. Addison Wesley, Reading, Massachusetts, Aug 1996.

[113] A. J. Turner. A summary of the ACM/IEEE-CS joint curriculum task force report: Computing curricula 1991. *CACM*, 34(6):69–84, Jun 1991.

[114] J. D. Ullman. *Elements of ML programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[115] M. Y. Vardi, K. B. Bruce, Ph. G. Kolaitis, and D. M. Leivant. Logic in the computer science curriculum. In *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education (SIGCSE-98)*, volume 30(1) of *SIGCSE Bulletin*, pages 376–377, New York, February 25–March 1 1998. ACM Press.

[116] W3C. *Synchronized Multimedia Integration Language (SMIL) 1.0 Specification*. World Wide Web Consortium, MIT, Boston, Massachusetts, Jun 1998. `www.w3.org/ TR/ REC-smil/`.

[117] R. L. Wainwright. Introducing functional programming in discrete mathematics. In M. J. Mansfield, C. M. White, and J. Hartman, editors, *23rd Int. Conf. Computer science education*, pages 147–152, Kansas, Missouri, Mar 1992. ACM SIGCSE bulletin, 24(1).

[118] N. Walsh. *A Technical Introduction to XML*. Arbortext, Inc, Ann Arbor, Michogan, 1999. `www.arbortext.com`.

[119] A. Wikström. *Functional programming using Standard ML*. Prentice Hall, London, UK, 1987.

[120] G. Winksel. *The formal semantics of programming languages*. MIT Press, Cambridge, Massachusetts, 1993.

[121] S. Wolfram. *The Mathematica Book*. Cambridge Univ. Press, fourth edition, Apr 1999.

[122] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, Europe, 1996.

[123] J. C. P. Woodcock and M. Loomes. *Software Engineering Mathematics: Formal Methods Demystified*. Pitman, 1988.

[124] J. B. Wordsworth. *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering.* Addison-Wesley Publishing Company, 1993.