Being Declarative

Searching for the Essence of Declarativeness

Martin Bravenboer
martin@cs.uu.nl

Center for Software Technology,
Institute of Information and Computing Sciences,
Utrecht University,
P.O. Box 80089, 3508 TB,
Utrecht, The Netherlands.

February 4, 2004

1 Programming Machines using Languages

Programming languages distinguish themselves in how they reduce the complexity of computer programming

Programming in the most general sense is about defining a configuration of a machine to control its operation. If the term computer programming is used, this often refers to any kind of programming related to a computer. A computer program is a specification of a computation that is to be performed by a computer. A programming language is a language in which such programs are written. A programming language where the constructs of the language directly correspond to the instructions supported by a processor is also known as assembly. Otherwise the programming language *abstracts* from the language accepted by the processor. Programs written in such languages are to be translated into an executable program by a compiler, or can be executed by *interpreting* the program using yet another program.

The subject of programming is often only indirectly the computer itself. Instead the subject machine is software on this computer. This software is configured to perform a certain operation. This subject software is a configuration of the machine by itself, which might lead to a large chain of configurations of machines.

The challenge of computer programming is controlling its complexity. Despite the deceiving simplicity of the concept of a modern computer, the source of computer programs tends to be far to complex to be comprehended by a single human mind. This situation leads to unintended behavior (aka bugs) and unmaintainable code. One of the ways of controlling this complexity is by using a more abstract programming language. A computer programmer needs to *express* his thoughts of the computation he is programming in the language he is using. From the very beginning of computer programming there has been a demand for programming languages that allow a programmer to express his thoughts in a natural, clear and abstract way. The programmer does not want to be bothered with irrelevant details: the details he has not got in his mind when he is thinking about the computation he wants to express.

Abstraction from the exact procedure, algorithm, implementation is important if you want to reduce complexity. But what is abstraction? What is a procedure? For example: recent object oriented languages like Java and C# all support overloading of methods based on argument types. Nearly every language supports overloading of basic arithmetic operators like + over the primitive data types of the languages. Some languages even support user-defined operator overloading, where you can define what a certain operator should do for custom data types. Clearly this is an abstraction for the programmer: thanks to overloading and the resolving of these 'details' in a compiler you do not have to specify exactly what method should be invoked.

It is about time to explain where this overview of terminology and concepts is leading to. In some overviews of programming paradigms 'declarative programming' is listed as a possible programming paradigm. Also actual programming languages often claim to be declarative, without actually calling it a paradigm. Such a statement claims a certain fitness of the programming language for the job of computer programming, often in a particular application domain. In order to understand the value of such claims it is important to investigate what being declarative actually involves.

Unfortunately declaritiveness has now become one of the most ill-defined terms in computer science. The term is widely used to emphasize properties of programming languages or programs, but far too often it is not discussed why the speaker actually uses the term declarative and where it is based upon. Therefore it has become one of the worst labels for a programming language (the worst label is probably 'powerful'). In this essay I will try to find out what it means to be declarative by uncovering the essence of declarativeness. I will suggest how and why to avoid the term declarative and state the real essence of the programming language instead.

2 Computation, Models, and Machines

Every program will be used for configuration of a machine.
Every piece of data is program.
Every program is a piece of data.

Programming languages provide the programmer a certain view on the machine that a programmer wants to configure. A programming language might provide a little or a lot of abstraction from the way a computation is performed on a computer, or it might even seem completely unrelated to the original way of computation in a computer. This view on computation, known as a computation model, is a set of concepts on which a programming language is based. A computation model is implemented by a machine in software (interpreter), hardware (processor), or by translation into a different computation model (compiler).

Programming languages are classified by separating them in a set of computation models, or programming paradigms. The term programming paradigm is used in practice to denote a class of programming languages that have some core idea in common. The word paradigm originates from the Greek word paradeigma, which means example or pattern. Programming paradigms, such as structured programming, imperative programming, functional programming and so on, group programming languages by their concepts. A programming language might be based on several paradigms and a programming paradigm might be a subset of an other paradigm. Most programming languages support more then one paradigm. In programming languages like Java and Haskell one paradigm is very prominent (respectively object oriented and functional programming), but object oriented programming in Lisp or Haskell is feasible as well. Because of this it would be better to emphasize that a certain language *supports* programming in a certain paradigm by various language constructs. An objective classification of programming languages is therefore more like a weighted graph, where programming languages and paradigms are nodes and edges express the relations between the languages and the paradigms.

In our search for the essence of declarativeness we have to consider the existing computation models and machines. What kind of computation models and machines are we using on a daily basis in computer programming?

As mentioned in the introduction programming is about the configuration of a machine. The purpose of such a machine is not always directly related to computation. Computer programming is more about programming then about computation in the mathematical sense. Computing is usually related to computation, algorithms and results. Many modern programming languages are however not strictly related to a single computation. Programmable declarativeness and definitional declarativeness. For example: SQL, XPath, XQuery are typically related to a certain computation model. A language for storing information about books and their authors is typically not: it is just data, which is not bound to a certain computational model: there is no definite machine in mind for execution.

It is for example quite odd the investigate the Turing completeness of the HTML language. Yet, a web browser is an HTML rendering machine and an HTML documentation is a program, a configuration, of this machine. This new configuration turns your web browser into a new machine; A machine that accepts user input by clicking at hyper links. Loading an HTML file into a web browser is related to partial evaluation. The HTML machine is partially applied to one of its inputs: an HTML file. This produces a new machine, which you are using on a daily basis.

In this sense even a file name might be a program. The configuration of the Unix `rm` command for example includes the files that have to be removed. By allowing wildcards such as `?` and `*` in the configuration of the `rm` command, for example like `rm *.java`, the configuration of the 'machine' rm becomes a more expressive programming language: it allows the programmer to *abstract* over the reoccuring task of removing files with similar names. The computation model of this language is very abstract: there is no need to iterate through the characters by hand using more executional language constructs like for loops. This language might look extremely declarative, yet it is quite bound to a certain computation model. A 'program' in this language can only be used to filter filenames. The programming language of tokens and wildcards such as `?` and `*` is however absolutely not bound to a certain *application*: the language can be used to configure a wide range of file system tools on your system.

2.1 Machines and Data

In the introduction of this section I have sketched the scope of machines and how they are configured by software to perform a certain operation. To get more feeling of the scope of machines and the way they are configured with programs, it is useful to discuss some actual machines and there configuration we are all using in theory and practice.

Turing machine Some computation models and machines are more related to the foundations of computer science then to practical computer programming by humans. The Turing machine is the ultimate example of such a computation model. Alan Turing introduced the Turing machine in 1936 as a computational model for procedures and algorithms. Turing provided theoretical computer science with a foundation for reasoning about computability. The Church-Turing thesis claims that the Turing machine is capable of executing any computation we can imagine. This is more a theoretical then a practical tool when reasoning about computer programming, especially when reasoning about declarativeness. Turing completeness focuses on possible computations, not the way these computations are expressed.

The Turing machine is an interesting machine in this essay on declarativeness, especially the *Universal Turing Machine*. The Universal Turing Machine can be configured with a Turing machine and the input of this machine. This Turing machine and its input is encoded as data on a tape, which is the input, output, and the 'memory' of Universal Turing Machine. The configuration language of the Universal Turing Machine is thus a language for Turing Machines and tapes. The configured Universal Turing Machine executes the tape's Turing machine and input.

Why is this interesting in this discussion of machines? The input of a Turing machine is in general just a tape of pure data. Just data. Yet, the data can be in fact be a full program with the completeness of a Turing machine, which is quite complete.

Implementing Universal Turing Machines in programming language is an operational proof that a programming language is Turing complete. An example of this is the Universal Turing Machine in XSLT [3]. XSLT is a a stateless transformation language for XML. The input of an XSLT program is just data: XML. This Universal Turing Machine in XSLT configures the XSLT interpreter into an Universal Turing Machine. The Turing machine and input (in the Turing Machine Markup Language) to be executed is passed as an XML document to the Universal Turing Machine in XSLT. It is interesting to observe how a machine that is to be configured with a stateless language, which some people would call declarative, is turned into a machine accepting any Turing machine and input.

Regular Expressions and Context-Free Grammars A regular expression over an alphabet of symbols defines a set of sentences (language) over this set of symbols. Usually this set of symbols is some set of characters, in which case the regular expression defines a set of character strings. Interpreters of regular expressions are used a lot in practice: almost every development platform (especially those targeted at web-scripting) comes with a library for evaluating regular expressions over character strings. Although regular expressions are sometimes related to computability in formal reasoning, regular expressions are not regarded as programs in practice.

A regular expression defines a language, but the language of regular expressions is a language as well. This language is supported by the regular expression interpreter, the machine. This machine is configured with a program: the regular expression. It then becomes a machine that recognizes sentences of the language defined by the regular expression.

Regular expression are a very successful abstraction for recognizing sentences of some language. Implementing such a recognizer by hand in some more operational language such as an imperative or functional programming language is tiresome and error-prone. Regular expressions thus provide a very useful abstraction, yet there is quite a fixed operational semantics for recognition of a sentence using a language of regular expression. The implementations of the interpreter can be very different, but the application of a regular expressions will always be related to the recognition of sentences. The language of regular expressions thus clearly provides a useful abstraction, is suitable for formal reasoning, but cannot be applied for many different purposes.

Regular expressions can only be used to define regular languages. Regular grammars are an alternative formalism for defining regular languages. The language of regular grammars is a restricted form of a formal grammar. Context-free grammars are more a less restrictive form of formal grammars and can be used to define a wider range of languages. Context-free grammars are widely used to define the syntax of programming languages and thus many languages and tools have been developed for them. The 'declarativeness' of these tools and languages is interesting to observe, since context-free grammars provide a lot of abstraction and syntax definitions are used for different purposes.

Due the algorithmic complexity of parser for grammars in the full class of context-free grammars many parser generators limit the syntax-definition to a subclass of the context-free grammars like LR, LL or LALR. These subclasses are not less expressive, but they make the syntax definitions less natural. When using these subclasses the syntax definition is restricted in some way to fit a certain algorithm applied in the parser. This influences the 'declarativeness' of the syntax definition. This practice has quite some consequences. First of all, the full class of context-free grammars is the only subclass of the context-free grammars that is closed under composition. All other subclasses, such as LL, LR and LALR are not. This disallows abstraction through the use of modules. Since in LL, LR and LALR the syntax definition is restructured to meet the requirements of the subclass, the syntax definition is changed in way that does not contribute to the clearness of the structure of the language. Tools that want to operate on this structure, such as generators of parsers, data types for abstract syntax trees, and pretty-printers, cannot distinguish the constructs that have been restructured to meet the requirements of the subclass, from relevant structure. The syntax definition formalism SDF2 [2, 4] supports the full class of context-free grammars. This has some consequences for the complexity of parsing, but it allows syntax definitions to be more concise and precise. This language is thus *declarative* then the syntax definitions used in conventional parsing technology. But how can SDF2 distinguish itself from other syntax definition languages if all these language claim to be declarative?

Von Neumann Architecture The Von Neumann architecture is named after John von Neumann, who proposed this architecture for computers in 1945. The architecture is very general: it is does not specify usual ingredients of a computation model, such as operations and data types. The Von Neumann architecture has become de facto architecture for computers. Different architectures of the hardware of a computer are almost impossible to imagine these days, although recently the Harvard Architecture is getting more attention because of security and digital rights issues.

In the Von Neumann architecture data and instructions for the computer are in the same area. Instructions are pure data, which allows them to be modified. In fact this is just like in the Universal Turing Machine. where the Turing machine to be executed (the instructions) and the data are on the same tape. The Von Neumann machine is thus re programmable.

Until now we have seen several machines and languages. Most programming languages feel more declarative if they feel more like just data that is used to configure a machine. In this point of view it is quite ironic that instructions for a machine of the Von Neumann architecture are usually not considered to be declarative. The main contribution o the Von Neumann architecture is that instructions and data are treated in the same way: instructions are just data.

Recently security issues have influenced the view of instructions as modifiable data. To prevent typical exploits such as the buffer overflow, the modifiable parts of the memory of a process are no longer executable. The instructions on, for example, the stack cannot be executed in this case. This approach, although not reflected in the hardware of a computer, is related to the Harvard Architecture, where data and instructions are separated from each other.

Java Virtual Machine The Java language, byte-code and machine, are the ingredients of a much more complex computation model that is quickly becoming the de facto platform for application development, together with the comparable .NET platform. Java byte-code is to be executed in Java Virtual Machine. The term 'virtual' in the name of this machine is short sighted. It suggest that the machine on which Java byte-code is executed is implemented in software, which is not necessarily required. Java Virtual Machines have been implemented in hardware for application in micro devices. Even if the machine would be implemented in software, the question remains why a machine implemented in software is more virtual then a hardware machine.

An interesting aspect of the Java Virtual Machine and its computational model is that the byte-code is much less atomic then the lambda calculus, instructions for typical machines of the Von Neumann machine, and a Turing machine and its input encoded on a tape. Java byte-code is very close to the Java language itself as it is used by programmers, unlike the lambda calculus and assembly instructions. Java byte-code is more or less just type-checked Java with a somewhat different notation for instructions. A Java Virtual Machine implementation must therefore have quite a detailed knowledge of classes, objects, inheritance, overriding, packages, and so on.

2.2 It's Just Data, It's Just a Program

As a first, careful, conclusion it has become clear that the difference between data and programs is vague. Data as well as programs are after all both used to configure machines to perform a certain computation. At which point of expressiveness this data becomes a program is unclear. If a certain piece of data, a program if you prefer this term, can be used in many different computations, then this might be an indication that the language used for this data is not strongly bound to a certain way of computation. These are all extremely imprecise notations. Even data that we would absolutely speak of as a program (like a piece of C source code) can be used in different ways. A program can for example be the input to a documentation generator, line count utility, a refactoring tool, and so on. But, in this case there is of course a main way of execution.

3 Definition Efforts

Declarativeness is ill-defined

In the introduction we claimed that declarativeness is an ill-defined term. It might be ill-defined indeed, but this does not imply that no effort has been undertaken to define what declarativeness is. In this section we will present some opinions and definitions of people who have at least been thinking about declarativeness.

3.1 The New Oxford Dictionary of English

Of course the first thing to do is to take a proper dictionary. The New Oxford Dictionary of English, which is quite a proper one, defines declarative in the context of computer programming as:

> *denoting high-level programming languages which can be used to solve problems without requiring the programmer to specify an exact procedure to be followed.*
>
> *The New Oxford Dictionary of English: Declarative*

Some questions arise while reading this definition: when is a procedure specified by a programmer 'exact'? What is a 'procedure'? The expression `1 + 2` is for example quite exact in what has to be done: it states that the numbers 1 and 2 should be added. There is some abstraction involved here: the programmer does not need to specify that the integers 1 and 2 must be put on the stack, followed by the execution of some addition instruction on integers of the processor of the computer. Yet, $1 + 2$ leaves not a lot of choice to the computer: it has to add the two numbers and why this is required to perform some computation is entirely unclear. Therefore the programmer still has encoded some procedure to be followed. Almost every high-level programming language (such as SQL, XQuery, XPath, and Haskell) provide an addition operator. The question is: what abstraction must a language provide to be classified as declarative?

3.2 Wikipedia

The Wikipedia [5] usually gives an interesting overview of what people currently relate to term since everyone is able to contribute to an entry of this extraordinary encyclopedia. The Wikipedia defines declarative programming as:

> *An approach to computer programming that takes a different approach from traditional imperative programming in Fortran, C++ or Java. Whereas imperative programming gives the computer a list of instructions to execute in a particular order, declarative programming describes to the computer a set of conditions and lets the computer figure out how to satisfy them.*
>
> *http://en.wikipedia.org/wiki/Declarative_programming*

The Wikipedia entry on declarative programming also mentions that functional programming and logic programming are forms of declarative programming. Miranda, Prolog, and SQL are mentioned as examples of declarative programming languages. Interesting to observe in this definition that the authors define declarative programming as every kind of programming that is not imperative. This does not allow a not imperative language (declarative) to distinguish itself from other not imperative languages. This definition of declarativeness in a striking way shows how the term declarative programming is used in practice and how it has become an almost meaningless term.

3.3 Concepts, Techniques, and Models of Computer Programming

In Concepts, Techniques, and Models of Computer Programming (CTM) [1] the authors, Peter Van Roy and Seif Haridi, provide much more detailed insight in declarativeness. In this book declarative programming is defined as

> *For now, we define this [declarative programming] as evaluating functions over partial data structures. This is sometimes called stateless programming, as opposed to stateful programming (also called imperative programming)*
>
> *CTM, Introduction of Chapter 2: "Declarative Computation Model"*

This definition is quite general and vague, but fortunately the author discuss declarativeness in much more detail later in the book. In chapter ... an operation is called declarative if,

> *whenever called with the same arguments, it returns the same results independent of any other computation state. A declarative operation is independent (does not depend on any execution state outside itself), stateless (has no internal execution state that is remembered between calls), and deterministic (always gives the same results when given the same arguments.*

CTM, Section 3.1: "What is Declarativeness?"

In CTM declarative programming is classified by expressiveness into descriptive declarativeness and programmable declarativeness. Descriptive declarative languages 'just' allow the definition of data. Programmable declarative languages are 'real' programming languages: they are as expressive as Turing machine. Functional and logic programming are in this category. The most interesting thing about this definition in CTM is that it considers declarative programming to be stateless programming. This is comparable to the definition of the Wikipedia if we consider stateful programming to be imperative programming? Most people will indeed describe stateful programming as imperative programming, but is this indeed a meaningful term? Let us get back to this point later.

The authors of CTM return the point of stateful, stateless, imperative and declarative programming in chapter 6. They also touch exactly the sore point of the term declarativeness.

> *Stateless and stateful programming are often called declarative and imperative programming, respectively. The latter terms are not quite right, but tradition has kept their use. Declarative programming, taken literally, means programming with declarations, i.e., saying what is required and letting the system determine how to achieve it. Imperative programming, taken literally, means to give commands, i.e., to say how to do something. In this sense, the declarative model of Chapter 2 is imperative too, because it defines sequences of commands. The real problem is that declarative is not an absolute property, but a matter of degree.*
>
> *CTM, Subsection "A problem of terminology"*
> *of Chapter 6: "Explicit State"*

4 Relatively declarative

Every programming language is declarative
if compared to some other programming language

The most important problem with the term declarative is that declarativeness is relative. Abstraction, details, and exact procedure all assume a different machine model then the model of a programming language that provides abstraction, hides exact procedures and does not bother you with details. What are those details? They are the details of the machine. The details have to be 'added' when the computer program gets executed on a certain machine. Labeling a language as being declarative therefore implicitly labels the language with a default execution machine model.

What it means to be declarative is sometimes described with the *how* versus the *what*. If you must exactly specify in programming language how some computation should be performed, it bothers you with the irrelevant details of how a computation should be performed and is therefore labeled as not being declarative. If a language allows you to express what should happen, then it abstracts over the exact procedure to follow. It hides some irrelevant details of the way the computation is performed on the target machine.

From this point of view imperative languages are not declarative. In imperative languages you must almost exactly specify how some computation should be performed. Nobody will argue that an imperative languages is not declarative: in fact the hive mind definition of being declarative is very close to not being imperative.

But now we have moved the problem: where does *what* stop and where does *how* start? If you implement a sorting algorithm in a functional programming language like Haskell, you have to specify exactly how the list should sorted: you cannot just say "sort this list!". Haskell itself does not even know what being sorted means. Still Haskell is usually labeled as a *what* language, not a *how* language. Why? We are so indoctrinated by execution on a Von Neumann machine, that *how* is being reserved for languages in which you have to express yourself in terms of an execution on this Von Neumann machine using an imperative language.

Programming in terms of *what* should happen is therefore a *relative* notion: leaving out the how details assumes that there are how details: it assumes a certain machine model of the machine where the programming will be executed on. If a functional programming is executed on a lambda calculus machine, there are no how details: the programming is a complete specification of how a machine should perform the computation.

Because of this *what* programming is only a senseful term if you have a programming language that gives you the feeling that you can just express what should happen, and if you assume a certain model for the machine the program will be executed.

4.1 Program verification

There is an interesting link between this point of view on *what* versus *how* and the research area of proving program correctness. Some people claim that a program written in a functional program does not need a proof. This statement is a logical result of the statement that functional programming is about what (not how) should happen. A correctness proof usually compares a description of what should happen to how this actually happens in an implementation. However, if there is no how, there is nothing left to proof (except the correctness of the compiler or interpreter). It will be clear that this "your code is your proof" is a vast exaggeration of the declarative nature of a functional language. It is not difficult to make a mistake in a Haskell program: just take the head element of an empty list or sort a list in the wrong order in the implementation of a sorting algorithm like quick-sort.

The crux of the matter is that two languages play a role in implementing a specification: a *specification* language and an *implementation* language. If the specification language is a functional language and the implementation language is imperative, the relative nature of 'what' comes up: the functional specification feels like 'what' should happen and the imperative implementation feels like 'how' this should happen.

If the implementation language is however a functional language, just as the specification language, there is no computation model, paradigm (way of expression) border to be crossed: the specification *is* the implementation. The programs stay in the same way of expression.

If you the specification language is an imperative language and the specification language is imperative as well, there is again no way of expression border that has to be crossed by a formal proof that shows that the implementation is a correct way of executing the computation described in the specification.

5 More Precise Terminology

Focus on real language features,
not on the comparison to other languages

In this essay I have discussed some typical machines, opinions on declarativeness, and a fresh point of view this. Criticism on current practices should always be accompanied by constructive advice. How should programming languages be described if the term declarativeness is as unclear as I have argued?

Instead of the ill-defined term 'declarative' it is more useful to mention the actual features of a language that provide the programmer with the feeling of specifying a program in a declarative way. The term declarative itself has become a meaningless term since every programming language that is not imperative is called declarative. As more and more not imperative languages are designed, the term declarative is becoming more and more meaningless: there is no differentiation possible in this area if there is just one term to describe them all. Programming languages are said to be declarative for different reasons and these underlying reasons should replace the term declarative in classifications and descriptions of programming languages. Reasons for calling a programming language declarative are:

1. the language is stateless, or to be more precise: there is not implicit state.
2. the language provides a reasonable amount of abstraction over the exact procedure to be followed.
3. an application of the language typically results in concise and clear programs.
4. the language is suitable for formal reasoning

More accurate descriptions are much more useful when trying to explain vision of a programming language to the programmer. The whole purpose of explaining the features of a programming language is to make clear why the programming language is fit for expressing the thoughts about a certain computation.

Imperative programming plays a central role in this discussion on what it means to be declarative. The imperative programming style expresses a computation in terms of statements that change the state of the machine on which the computation is performed. This style of programming naturally emerged out of the design of the machines on which computations are performed. In fact imperative programming was not really designed as a programming style, but evolved from programming machines by providing it native machine code instructions. By introducing more and more high level abstractions imperative programming became, structured (iteration), procedural (procedures) and object oriented (data and procedures).

Not imperative programming languages are in most cases better described as stateless languages, since even the amount of imperativeness in stateless languages can differ quite a lot. In fact imperative programming languages are even better described as stateful 'language, languages with an implicit state, since imperativeness is not a feature exclusive to stateful languages. The term stateless needs more explaining as well. Such languages are more precisely described as languages without *implicit state*. Having no implicit state might improve the clearness and predictability of source code. However, care must be taken not to bash implicit state without clear arguments. Implicit under clear control of the programmer is not necessarily a very bad thing. The main problem of stateful languages is that the encourage an overuse of implicit state. Restricted use of implicit state might however even improve the conciseness of a program. In the Stratego program transformation language scoped dynamic rewrite rules play for example an important role in implementing context dependent transformations, such as function inlining, constant propagation and bound variable renaming. These transformations would be hard to implement without the implicit state of dynamic rewrite rules. By limiting the scope of dynamic rules the implicit state in the programs is easy to control.

In some cases the term declarative could be replaced by the term 'specification' or 'definition'. These terms are somewhat ambiguous as well, but they might make more clear that a program is not strongly bound to execution at a certain machine. Programs that feel more like being just data are usually more descriptive and abstract from exact procedures and algorithms. Context-free grammars as syntax definitions are good examples of this.

References

[1] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[2] SDF2 website. www.syntax-definition.org.

[3] Unidex. Turing machine markup language (TMML). www.unidex.com/turing/index.htm, 2001.

[4] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[5] Wikipedia, the free encyclopedia. www.wikipedia.org.