Declarative Programming Across the Undergraduate Curriculum

Amr Sabry
Department of Computer and Information Science
University of Oregon
Eugene, OR 97403

Abstract

In its ultimate form, declarative programming provides solutions to problems using only a statement of the problem as the program. Historically this level of abstraction has been associated with specialized programming languages, but it is now routinely used by both students and professional programmers in a variety of contexts, programming languages, and courses.

Despite the pervasiveness of declarative programming in a typical undergraduate curriculum, the exposition is based on isolated examples with no common principles. We argue that the principal foundation of declarative programming is *abstraction*. In many cases, providing the right data and control abstractions for the problem at hand requires the use or design of a domain-specific language, but in other cases, a simple library or datatype definition does the trick.

Thus we argue that the key to effective use of declarative programming in the classroom is to teach abstraction in programming at several levels: an introductory course should focus on defining abstraction in the computer science sense of separating concerns and representation independence, and then on finding the right abstractions to simple programming problems; capstone courses should relate the importance of abstraction in modern software engineering practice, and may even provide experience in designing and implementing sophisticated abstractions in the form of domain-specific languages. These courses at the two ends of the curriculum capitalize on the existing base of declarative programming examples in courses such as compilers, user interfaces, databases, distributed systems.

For concreteness, the paper provides several code examples and the outline for an existing introductory course that teaches the principles of declarative programming using machine language, assembly language, and Java.

1 Introduction

Declarative programming, in which one only states *what* needs to be computed without necessarily stating *how* it is to be computed is such an obviously good idea that it needs little motivation. Indeed most undergraduate courses that involve a programming component naturally use a declarative style whenever possible.

For example in a course on compilers students rarely write programs which explicitly state how to parse a language. Instead the common approach is to write regular expressions and context-free grammars which only specify what needs to be parsed. As another example in a course on user-interfaces students rarely write programs which explicitly position graphical user interface (GUI) elements on the screen. Instead the common approach is to write geometric constraints that specify the desired layout in declarative terms. Similarly in a course on distributed systems students rarely write programs which explicitly state how to establish a connection with a remote object using sockets, streams, and network protocols. Instead the common approach is to use a declarative notation in which one only specifies the interface of the remote object and the operation selected from that interface. Abstractions provided by CORBA [6] or Java's RMI [8] are then translated to the low-level code needed to establish a connection with the desired remote object. As a final example software engineering courses emphasize declarative programming notations (even if they are not executable) in the form of software architectures, object-oriented design patterns, and specification languages.

In summary most programming courses in a typical undergraduate curriculum actually teach declarative programming to a certain degree. Unfortunately this declarative programming theme is rarely emphasized as a separate topic and neither its foundations nor its applications are explicitly recognized. The wealth and scope of declarative programming examples suggests that the foundations of declarative programming encompass many programming languages and that it is motivated by industrial software engineering practice. Indeed we argue that the key to declarative programming is finding and developing the right abstractions. In complex cases this may require the use and design of a domain-specific language, but in other cases common programming constructs and standard libraries may be sufficient to do the job.

Thus an effective way to attract students to the declarative programming style without using esoteric programming languages and techniques is to simply capitalize on the strengths of existing curricula by supplementing them with a small number of strategic courses. A foundational introductory course would teach students how to find and develop the right abstractions and instill upon them the importance of doing so by relating different solutions at different levels

To appear in the workshop on Functional and Declarative Programming in Education, 1999

of abstractions. A capstone course would show how industrial software engineering practice dealing with the development of large complex systems led to the development of declarative solutions to many programming problems using libraries, little languages, and domain-specific languages.

To make the arguments in the paper concrete we use several code examples from courses on compilers, symbolic computation, user interfaces, and hardware specification and verification. We then present the outline of an introductory course that uses machine language, assembly language, and Java to teach among other things, the principles of declarative programming. We also briefly touch on the connections to software engineering and programming languages that might be incorporated in an advanced capstone course.

2 Examples in a Typical Curriculum

We begin with a number of examples of declarative programming inspired from typical undergraduate courses.

2.1 Parsing

Compilers provide one of the classical and most compelling examples of declarative programming in a typical curriculum. Most compiler courses involve writing a program to parse the input language. This program is almost always written declaratively: its main component is the grammar of the language to be parsed.

For example consider a simple grammar for arithmetic expressions:

$$e ::= n \mid (e) \mid e + e \mid e * e$$

with the usual precedence of operators. A parser and interpreter for the grammar can be written declaratively using the CUP LALR parser generator for Java [1]:

```
----------------------------------------------------
terminal Integer NUMBER;
terminal PLUS, MULT, LPAR, RPAR;

non terminal Integer prog, exp;

precedence left PLUS;
precedence left MULT;

start with prog;

prog ::= exp:e              {: RESULT = e; :}
      ;

exp ::= NUMBER:n            {: RESULT = n; :}
     |  LPAR exp:e RPAR     {: RESULT = e; :}
     |  exp:e1 PLUS exp:e2
          {:
             RESULT = new Integer(e1.intValue() +
                                  e2.intValue());
          :}
     |  exp:e1 MULT exp:e2
          {:
             RESULT = new Integer(e1.intValue() *
                                  e2.intValue());
          :}
     ;
----------------------------------------------------
```

The description consists of the terminals of the grammar, the non-terminals, the precedence rules, the start symbol, and the productions. The interpreter is generated from the actions attached to each grammar production. Although the parser and interpreter for this small language are easily written in other ways the advantages of using the declarative style of parser generators should be clear when one considers larger and more complicated cases.

2.2 Lexical Analysis

The parsing program above assumes that the terminals of the language have already been recognized. This problem known as lexical analysis also has a declarative solution which only requires one to write a set of regular expressions describing the terminals.

Using the JLex lexical analyzer generator for Java [1] we write:

```
----------------------------------------------------
DIGIT=[0-9]
WHITE_SPACE=[\ \t\b\n\r]
NUMBER={DIGIT}+

{WHITE_SPACE}+  {}
"+"             { return tok(sym.PLUS); }
"*"             { return tok(sym.MULT); }
"("             { return tok(sym.LPAR); }
")"             { return tok(sym.RPAR); }
{NUMBER}        {
                 return tok(sym.NUMBER,
                             new Integer(yytext()));
                }
.               {}
----------------------------------------------------
```

The program is automatically generated from the above declarative description and interfaces nicely with the CUP generated-parser to yield a complete front-end for the language of arithmetic expressions.

2.3 Symbolic Computation

Programming problems also appear in some mathematical courses, *e.g.*, computational algebra. A common and simple problem in this domain is the solution of a system of linear equations. Using conventional general-purpose programming languages this problem might be solved by representing the coefficients of the equations using matrices and then implementing Gaussian elimination. In contrast when using a language like Mathematica [12] the problem can be solved by simply stating it. For example the expression:

$$\text{Solve}[x == 2y - 3, y == x - 2, x, y]$$

can be directly evaluated in Mathematica to yield the solution:

$$x \rightarrow 7, y \rightarrow 5$$

One can informally conclude that as far as this particular task is concerned Mathematica provides a more declarative programming style than other languages. The key observation is that Mathematica includes a large number of built-in algorithms for manipulating algebraic equations which relieves programmers from having to state how to solve particular problems.

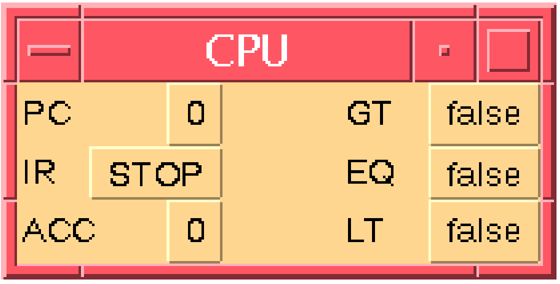

Figure 1: Layout of CPU Components

2.4 GUI Layout

Programming a GUI poses several programming problems, one of which is computing the layout of the GUI elements on the screen. At a low level of abstraction the programmer computes the pixel positions of each GUI element relative to another element, for example as provided in the initial version of the Fudgets library for Haskell [5]. This approach is clearly tedious, especially since the layout of the elements is usually required to be robust even when the containing frame is resized by the user or the window manager.

Modern languages like Java provide libraries in which the user can express a layout declaratively using geometric constraints. For example the layout of the GUI in Figure 1 can be programmed in a straightforward manner as follows:

```
---------------------------------------------------
class CPU_panel extends Panel {
    CPU_panel () {
        setLayout(new BorderLayout());
        add(new ALU_panel(),
            BorderLayout.EAST);
        add(new Registers_panel(),
            BorderLayout.WEST);
    }
}

class Registers_panel extends Panel {
    Registers_panel () {
        setLayout(new BorderLayout());
        add(new PC_panel(),
            BorderLayout.NORTH);
        add(new ACC_panel(),
            BorderLayout.SOUTH);
        add(new IR_panel(),
            BorderLayout.CENTER);
    }
}

class ALU_panel extends Panel {
    ALU_panel () {
        setLayout(new BorderLayout());
        add(new GT_panel(),
            BorderLayout.NORTH);
        add(new EQ_panel(),
            BorderLayout.CENTER);
        add(new LT_panel(),
            BorderLayout.SOUTH);
    }
}

abstract class REG_panel extends Panel {
    REG_panel (String name, String init) {
        setLayout(new BorderLayout());
        add(new Label(name),
            BorderLayout.WEST);
        add(new Button(init),
            BorderLayout.EAST);
    }
}

class PC_panel extends REG_panel {
    PC_panel () { super("PC", "0"); }
}

class GT_panel extends REG_panel {
    GT_panel () { super("GT", "false"); }
}

class ACC_panel extends REG_panel {
    ACC_panel () { super("ACC", "0"); }
}

class IR_panel extends REG_panel {
    IR_panel () { super("IR", "STOP"); }
}

class EQ_panel extends REG_panel {
    EQ_panel () { super("EQ", "false"); }
}

class LT_panel extends REG_panel {
    LT_panel () { super("LT", "false"); }
}
---------------------------------------------------
```

The Java abstract window toolkit and more recently the Swing library offer many more declarative ways for specifying the layout of GUI components. The most sophisticated `GridBagLayout` manager requires explicit geometric constraints that define the relative size, weight, and position of each component. The constraints are expressed as an instance of the class `GridBagConstraints` in what is essentially a separate little language.

2.5 Hardware Design

Modern hardware description languages allow engineers to express the *behavior* of a hardware circuit at a high-level of abstraction. Once the design is deemed correct a *structural* description of the circuit that uses low-level gates and transistors is synthesized.

In our terms the behavioral description of a hardware circuit is a declarative program that precisely states what the circuit does but not how to construct it. As a simple example consider the problem of designing a 4-bit synchronous counter. The counter has one input: the clock, and one output: the current count. Initially the output should be 0, and then it should be incremented at every negative edge of the clock. In a hardware description language like Verilog it is possible to write a simple behavioral (declarative) description of the counter that can be used for simulation and verification [9]:

```
---------------------------------------------------
module counter (ck, count);
   input ck;
```

```
   output count;
   reg [3:0] count;

   initial count = 0;
   always @(negedge ck) #10 count = count + 1;
endmodule
---------------------------------------------------
```

The counter module should be almost self-explanatory even for readers not familiar with Verilog. (See [3] for a precise operational semantics of behavioral Verilog.) The key thing to note about the the behavioral description of the circuit is that the increment operation of the counter is performed using the mathematical addition function +.

In contrast a structural description of the counter would use low-level gates to realize the increment operation. Again in Verilog we would write [9]:

```
---------------------------------------------------
module counter (ck, count);
   input ck;

   output [3:0] count;

   dEdgeFF
     d1(count[0], ck, ~count[0]),
     d2(count[1], ck, count[1] ^ count[0]),
     d3(count[2], ck, count[2] ^ & count[1:0]),
     d4(count[3], ck, count[3] ^ & count[2:0]);
endmodule

module dEdgeFF (q, ck, data);
    input ck, data;

    output q;
    reg q;

    initial q = 0;
    always @(negedge ck) #10 q = data;
endmodule
---------------------------------------------------
```

The structural description uses four flip-flops and logic gates to realize the increment operation which is much less apparent than the mathematical addition function but closer to a real hardware circuit.

In summary engineers rely on behavioral (declarative) solutions for reasoning about hardware designs. Not surprisingly typical courses on hardware specification also use behavioral descriptions for verification purposes. See:

http://www.cs.uoregon.edu/~sabry/teaching/hardware

and

http://www.cl.cam.ac.uk/users/mjcg/Teaching/SpecVer2

for examples and links to other courses.

2.6 Other Examples

The above examples are not meant to be inclusive. A typical curriculum is likely to have many more courses with programming components that rely on a declarative style. For example in a course on software engineering students might use a declarative specification language for stating the requirements. In a course on artificial intelligence students might solve planning problems by simply stating the rules defining how plans could be constructed and letting a Prolog engine find the plan. In a course on distributed systems students might use declarative notations for specifying the interfaces of remote objects and for accessing remote operations. Courses on simulation, databases, shell programming, editing, and web page construction are also likely to use declarative solutions to programming problems. The important observation is that the declarative programming style is prevalent in a typical curriculum.

3 Declarative Programming Principles

Some reflection on the above examples of declarative programming reveals that the underlying common principle is the use of the right abstractions that match the problem domain. In some cases the abstractions are essentially abstract data types, *e.g.*, the datatype of linear equations in Mathematica. In other cases the abstractions are encapsulated in a library, *e.g.*, Java's abstract window toolkit. Finally in more complex cases the abstractions are provided by a domain-specific language, *e.g.*, JLex or CUP, and as Hudak argues [10] domain-specific languages are "the ultimate abstraction".

The Verilog examples are an interesting case. The Verilog language itself is considered a domain-specific language for describing hardware circuits. But it is clear that within that language some programs are considered declarative and some are not. In our example the problem of addition is given two solutions: a declarative solution using the mathematical function + and a non-declarative solution implemented using low-level primitives.

4 Teaching with Declarative Style

Having argued that a typical curriculum includes a substantial exposition to declarative programming examples and that the key to declarative programming is *abstraction* we turn our attention to how to best use a declarative style in the classroom. The strategy we propose exploits the existing base of declarative examples and does not depend on the choice of programming languages. The main point is to use abstraction as a unifying theme and principle. Early courses can simply teach students the importance of using abstractions and how to use existing abstractions effectively. More advanced courses would require the students to develop new abstractions which could even require the design and implementation of a complete domain-specific language. Finally the practical relevance of declarative programming to modern software engineering should be stressed; as should be the importance of using and developing the proper abstractions.

For concreteness we explain how our introductory course at the University of Oregon teaches the principles of declarative programming using machine language, assembly language, and Java. For more information see:

http://www.cs.uoregon.edu/~sabry/duckMachine

4.1 Introductory Course Outline

Not unlike many introductory computer science courses the goal of our course is to introduce the field of computer science, its main subject areas, its recurring concepts, and its mathematical foundations in the context of *programming* exercises and projects. Before exploring how the course emphasizes the principles of declarative programming we briefly review the course outline.

The two main themes in the course are:

1. Program Construction: Modern software systems are built by the composition of independent components.

One of the most important factors in the success of these systems is the ease with which their functionality can be extended and their components upgraded. The course therefore emphasizes program construction techniques (design patterns) that lead to flexible and extensible software systems.

2. Evaluation Model: It is impossible to become a good programmer without having an understanding of how programs evaluate. At the end of the day programs execute on hardware machines with physical limitations on the number of registers and the size of the memory; programmers must understand the basics of these machines and be aware of their limitations.

4.2 The Duck Machine

To meet both high-level objectives the course contains a laboratory component in which students build a Java model of a simple hardware machine [7] called the *Duck Machine*. The exercise of building the Duck Machine serves several purposes:

- First it explains the architecture and organization of a hardware machine and the evaluation of programs at that level.
- Second it serves as an example of a modern software system that must be built to be flexible and extensible. Indeed the project emphasizes the use of sophisticated design patterns during the implementation.
- Finally the organization of the course around the implementation of a hardware machine leads to a natural overview of some of the most important fields of computer science including computer organization, operating systems, software engineering, programming languages, data structures, algorithms, and user interfaces. More importantly the beauty of the course is that all these topics are presented in concert to provide a coherent whole. It also illustrates that many areas of computer science need to be mastered for even the simplest of projects.

4.3 Using Proper Abstractions

The organization of the course around building a hardware machine naturally exposes the students to programming at different levels of abstractions. In fact several of the examples in Section 2 were drawn from our introductory course.

Not unlike the Verilog examples in Section 2.5 consider the problem of multiplying two numbers. Java provides a declarative solution to the problem using the built-in operation of multiplication. Java also provides an operational solution using a loop that repeatedly adds one of the operands to itself. This loop is the natural way to express the solution in assembly using the ADD instruction of the machine.

At a yet lower level of abstraction the ADD instruction only states what needs to be done: add the contents of the accumulator to the contents of a memory location without stating how this operation is performed. Eventually students deal with the details of how to perform the operation: load the contents of the memory location in question to some register R, select the operation *add* of the arithmetic and logic unit passing the contents of the accumulator and register R, and finally copy the result into the accumulator.

The additional complexity at each level is obvious; and so is the meaning and power of abstraction.

The GUI layout problem for the CPU in Section 2.4 is also inspired by our introductory course. After being introduced to the built-in layout managers of the Java library students immediately see the appeal of specifying geometric constraints between GUI components instead of explicitly positioning these components on the screen.

In addition to layout managers the Java libraries offer an extensive choice of abstractions. For example the `io` package includes many high-level input/output streams in which one simply states what kind of data needs to be read or written without worrying about the details of the data representation and the parsing and pretty-printing algorithms involved in translating to and from the different representations. And indeed much of Java programming is about selecting appropriate data types and operations from the standard API. Following industrial practice and modern software engineering principles the course generally requires that students use the abstractions provided by the standard API as much as possible.

In summary we argue that most introductory programming courses have ample opportunity to motivate declarative programming by emphasizing the data and control abstractions available in the programming language of choice, teaching the effective use of such abstractions, and explaining the cost of not doing so in terms of complexity of code and development time. The lessons are more pronounced if the course uses more than one programming language. Indeed if one uses different programming languages that provide radically different levels of abstractions the contrast between the declarative and non-declarative styles of programming is even more pronounced and the advantages of the declarative style are even more obvious.

4.4 Develop Proper Abstractions

As argued above declarative programming builds on a foundation of data and control abstractions. When the programming language provides the proper abstractions for a problem domain the task of programming a solution to a particular problem in a declarative style is reduced to choosing the right primitives from the language which although a non-trivial task is simpler than programming from scratch.

But what happens when the programming language does not provide the proper abstractions for a particular problem? In this case the task of programming requires in addition to the selection of a combination of primitive language abstractions, the building of a proper higher-order abstraction that provides a *reusable* declarative solution to the problems in the same domain.

Declarative programming is therefore as much about using the proper abstractions as it is about producing the proper abstractions for future use. In the current introductory course this point is reinforced by having the students develop abstractions early in the course like the machine components and the assembler that are used to provide declarative solutions to problems later in the course. Some of the abstractions are developed using individual Java constructs like classes and interfaces; some require more sophisticated combinations of Java constructs organized in a design pattern (like the visitor pattern [4]).

The course also appeals to software engineering principles which deal largely with program design issues. Indeed the discipline of software engineering provides general guiding principles for program design like encapsulation which when used properly result in reusable abstractions. Software engineering also provides technologies like software architectures and object-oriented design patterns that naturally lead to a declarative style of programming by encouraging programmers to use abstractions whenever possible and more importantly to develop the right abstractions if these are not available. Finally software engineering deals extensively with program specifications which are declarative by nature and even sometimes executable.

In a more advanced course one would challenge students to develop more complex abstractions like control abstractions, *e.g.*, coroutines, or even the more challenging task of matching the semantics of a particular domain by developing a domain-specific language that provides abstractions tailored for the domain and thus providing declarative solutions to a family of problems. For developing such abstractions the host programming language must itself provide a rather rich set of abstractions suitable for defining new languages. As experience has shown functional languages with many higher-order constructs and rich type systems like SML and Haskell have proved invaluable in this regard [10, 11].

5 Challenges

Declarative programming has its known drawbacks: most notably declarative programs do not offer the user control over the parameters that were abstracted. For example by using *Solve* a Mathematica user gives up control over the chosen algorithm to solving the system of linear equations. Similarly declarative programs generally do not allow control over resources like memory allocation which might be crucial in some situations.

The use of declarative solutions in the classroom not only suffers from the same problems that occur when generally using declarative programming but also from the particularly acute problem of debugging. Students are likely to make mistakes but the declarative nature of the programs is now a liability as it hides much of the operational details necessary to track and correct the error.

For example consider the parsing program of Section 2.1 modified by removing the precedence declarations. Attempting to compile this specification gives four error messages of the form:

```
----------------------------------------------------
*** Shift/Reduce conflict found in state #11
  between exp ::= exp MULT exp (*)
  and     exp ::= exp (*) PLUS exp
  under symbol PLUS
  Resolved in favor of shifting.
----------------------------------------------------
```

Unfortunately without a precise operational understanding of LALR parsing such errors are hard to correct. Similar obscure errors occur in every one of the examples of Section 2.

6 Conclusion

Declarative programming is no longer an esoteric view of programming associated with specialized programming languages and problem domains. Rather, years of experience in writing large complex programs in industrial languages and settings have contributed to a general trend towards declarative programming. Indeed many courses in a typical undergraduate computer science curriculum use declarative programming whenever possible. To best use declarative programming in such an educational context we argue for an approach that builds upon and exploits the existing base by providing two courses:

1. An introductory course which emphasizes the principles of data and control abstraction.

2. A software engineering course which emphasizes the relevance of declarative programming principles to the development of software and challenges students to develop non-trivial abstractions.

These courses can be easily integrated in an existing curriculum as they do not require radical changes. For the introductory course given almost any language, some carefully chosen problems will have a mostly declarative solution in that language that can be contrasted with a solution at a lower level of abstraction. Also the software engineering principles of encapsulation and reuse and the design techniques based on software architectures and patterns all naturally lead to a declarative style of programming in which one develops reusable abstractions. These ideas can be partly explained in an introductory course or a more advanced software engineering course and can be reinforced with case studies from other courses like compilers, user interfaces, and distributed systems, that most likely use declarative programming.

More research is needed to address the problem of debugging declarative programs. Pedagogic programming environments like DrScheme [2] provide a model on how to integrate the research results in an educational context.

Acknowledgments

Thanks to Michal Young for a lively discussion about the nature, history, and practice of declarative programming.

The author was partially supported by the National Science Foundation under the title: "Career: Controlling Space Properties of Higher-Order Typed Programs", grant number CCR-9733088.

References

[1] Appel, A. *Modern Compiler Implementation.* Cambridge University Press, 1997.

[2] Findler, R., Flanagan, C., Flatt, M., Krishnamurthi, S., and Felleisen, M. DrScheme: A pedagogic programming environment for scheme. In *International Symposium on Programming Language Implementation and Logic Programming* (1997).

[3] Fiskio-Lasseter, J., and Sabry, A. Putting operatinal techniques to the test: A syntactic theory for behavioral Verilog. In the *Third International Workshop on Higher Order Operational Techniques in Semantics* (1999).

[4] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley, 1995.

[5] Hallgren, T., and Carlsson, M. Programming with Fudgets. In *Advanced Functional Programming* (1995), J. Jeuring and E. Meijer, Eds., Springer Verlag, LNCS 925, pp. 137–182.

[6] *CORBA: Architecture and Specification.* Object Management Group, Inc., 1995.

[7] Schneider, G. M., and Gersting, J. L. *An Invitation to Computer Science*. West Publishing Company, 1995.

[8] Sun Microsystems. Java RMI.

[9] Thomas, D. E., and Moorby, P. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Norwell, Mass., 1991.

[10] USENIX. *Proceedings of the Conference on Domain-Specific Languages* (Oct. 1997).

[11] USENIX. *Proceedings of the Conference on Domain-Specific Languages* (Oct. 1999).

[12] Wolfram, S. *Mathematica*. Addison-Wesley, Redwood City, CA, 1988.