Functional programming and declarative programming

Sergei Winitzki

Academy by the Bay 2024

2024-04-14

Outline

Functional programming is interesting, but in a weird way

- FP is (almost) engineering
- Other programming paradigms are (almost) artisanship

A definition of declarative programming:

- A language for problem requirements, understandable to people
- The same language is mechanically translated into code
- The only known "silver bullet" of programming

Declarative programming must be domain-specific (DSL)

- We get a productivity boost as long as we avoid the pitfalls

Implementing DSLs is a "killer app" for functional programming

- FP is declarative for working with recursive data structures, such as labeled trees
- Requires learning some programming language theory

Is the Functional Programming community weird?

The FP community is unlike other programmers' communities

- Others are focused on a chosen programming language (Java, Python, JavaScript, etc.), and on designing and using libraries and frameworks
    - *"setup this YAML config, override this method, use this annotation"*
- People in the FP community talk in a very different way
    - *"referential transparency, algebraic data types, monoid laws, parametric polymorphism, free applicative functors, monad transformers, Yoneda lemma, Curry-Howard isomorphism, profunctor lenses, catamorphisms"*
        - ★ A glossary of FP terminology (more than 100 terms)
    - From SBTB 2018: *The Functor, Applicative, Monad talk*
        - ★ By 2018, everyone *expects* to hear these concepts mentioned

As a former theoretical physicist, I recognize that sort of jargon

- The FP jargon is used similarly to an engineer's jargon
- It is based on math but heavily adapted to the engineering domain
    - Rigor is on the need-to-know basis, mathematical abstraction is limited

Programming as engineering vs. artisanship

- FP is similar to engineering in how it uses math-based tools
  - Mechanical, electrical, chemical engineering use calculus, complex variables, classical and quantum mechanics, electrodynamics, thermodynamics, physical chemistry
  - FP uses category theory, type theory, logic proof theory, λ-calculus
- Engineers use *a lot* of special terminology
  - Examples from mechanical, electrical, chemical engineering: rank-4 tensors, Lagrangians with non-holonomic constraints, Fourier transform of the delta function, inverse Z-transform, Gibbs free energy 1, 2
  - Examples from FP: rank-N types, continuation-passing transformation, polymorphic lambda functions, free monads, hylomorphisms
- As in engineering, the special terminology in FP is *not* self-explanatory
  - "Gibbs free energy" is not energy that J. W. Gibbs provides for free

All that stuff is mathematics-based knowledge that needs learning

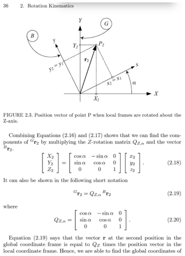
- Programmers today neither study as engineers, nor work as engineers

Books on engineering vs. artisanship

Functional programming looks like engineering

Today's "software engineering" resembles artisanship

Books on mechanical, electrical, chemical engineering design:



36 2. Rotation Kinematics

FIGURE 2.3. Position vector of point P when local frames are rotated about the Z-axis.

Combining Equations (2.16) and (2.17) shows that we can find the components of ${}^G\mathbf{r}_2$ by multiplying the Z-rotation matrix $Q_{Z,\alpha}$ and the vector ${}^B\mathbf{r}_2$.

$$\begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix}. \qquad (2.18)$$

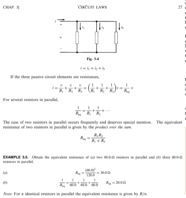
It can also be shown in the following short notation

$${}^G\mathbf{r}_2 = Q_{Z,\alpha}\, {}^B\mathbf{r}_2 \qquad (2.19)$$

where

$$Q_{Z,\alpha} = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}. \qquad (2.20)$$

Equation (2.19) says that the vector $\mathbf{r}$ at the second position in the global coordinate frame is equal to Q_Z times the position vector in the local coordinate frame. Hence, we are able to find the global coordinates of



CHAP. 3] CIRCUIT LAWS 27

Fig. 3-4

$$i = i_1 + i_2 + i_3$$

If the three passive circuit elements are resistances,

$$i = \frac{v}{R_1} + \frac{v}{R_2} + \frac{v}{R_3} = \left(\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}\right) v = \frac{1}{R_{eq}} v$$

For several resistors in parallel,

$$\frac{1}{R_{eq}} = \frac{1}{R_1} + \frac{1}{R_2} + \cdots$$

The case of two resistors in parallel occurs frequently and deserves special mention. The equivalent resistance of two resistors in parallel is given by the *product over the sum.*

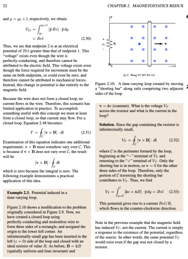
$$R_{eq} = \frac{R_1 R_2}{R_1 + R_2}$$

EXAMPLE 3.5. Obtain the equivalent resistance of (*a*) two 60.0-Ω resistors in parallel and (*b*) three 60.0-Ω resistors in parallel.

(*a*) $$R_{eq} = \frac{(60.0)^2}{120.0} = 30.0\,\Omega$$

(*b*) $$\frac{1}{R_{eq}} = \frac{1}{60.0} + \frac{1}{60.0} + \frac{1}{60.0} \qquad R_{eq} = 20.0\,\Omega$$

Note: For *n* identical resistors in parallel the equivalent resistance is given by R/n.



22 CHAPTER 2. MAGNETOSTATICS REDUX

and $y = y_0 + l$, respectively, we obtain

$$V_{21} = \int_{y_0}^{y_0+l} [\hat{y} B v] \cdot \hat{y}\, dy = Bvl \qquad (2.50)$$

Thus, we see that endpoint 2 is at an electrical potential of Bvl greater than that of endpoint 1. This "voltage" exists even though the wire is perfectly-conducting, and therefore cannot be attributed to the electric field. This voltage exists even though the force required for movement must be the same on both endpoints, or could even be zero, and therefore cannot be attributed to mechanical forces. Instead, this change in potential is due entirely to the magnetic field.

Because the wire does not form a closed loop, no current flows in the wire. Therefore, this scenario has limited application in practice. To accomplish something useful with this concept we must at least form a closed loop, so that current may flow. For a closed loop, Equation 2.48 becomes:

$$V = \oint_C [\mathbf{v} \times \mathbf{B}] \cdot d\mathbf{l} \qquad (2.51)$$

Examination of this equation indicates one additional requirement: $\mathbf{v} \times \mathbf{B}$ must somehow vary over $\mathcal{C}$. This is because if $\mathbf{v} \times \mathbf{B}$ does not vary over $\mathcal{C}$, the result will be

$$[\mathbf{v} \times \mathbf{B}] \cdot \oint_C d\mathbf{l}$$

which is zero because the integral is zero. The following example demonstrates a practical application of this idea.

Example 2.3. Potential induced in a time-varying loop.

Figure 2.10 shows a modification to the problem originally considered in Figure 2.9. Now, we have created a closed loop using perfectly-conducting and motionless wire to form three sides of a rectangle, and assigned the origin to the lower left corner. An infinitesimally-small gap has been inserted in the left ($z = 0$) side of the loop and closed with an ideal resistor of value R. As before, $\mathbf{B} = \hat{\mathbf{x}}B$ (spatially uniform and time invariant) and

© C. Wang CC BY-SA 4.0

Figure 2.10: A time-varying loop created by moving a "shorting bar" along rails comprising two adjacent sides of the loop.

$\mathbf{v} = \hat{\mathbf{z}}v$ (constant). What is the voltage V_T across the resistor and what is the current in the loop?

Solution. Since the gap containing the resistor is infinitesimally small,

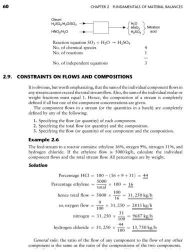
$$V_T = \oint_C [\mathbf{v} \times \mathbf{B}] \cdot d\mathbf{l} \qquad (2.52)$$

where $\mathcal{C}$ is the perimeter formed by the loop, beginning at the "−" terminal of V_T and returning to the "+" terminal of V_T. Only the shorting bar is in motion, so $\mathbf{v} = 0$ for the other three sides of the loop. Therefore, only the portion of $\mathcal{C}$ traversing the shorting bar contributes to V_T. Thus, we find

$$V_T = \int_{y=0}^{l} [\hat{\mathbf{z}}v \times \hat{\mathbf{x}}B] \cdot \hat{\mathbf{y}}\, dy = Bvl \qquad (2.53)$$

This potential gives rise to a current Bvl/R, which flows in the counter-clockwise direction.

Note in the previous example that the magnetic field has induced V_T, not the current. The current is simply a response to the existence of the potential, regardless of the source. In other words, the same potential V_T would exist even if the gap was not closed by a resistor.



60 CHAPTER 2 FUNDAMENTALS OF MATERIAL BALANCES

Reaction equation $SO_3 + H_2O \rightarrow H_2SO_4$	
No. of chemical species	4
No. of reactions	1
No. of independent equations	3

2.9. CONSTRAINTS ON FLOWS AND COMPOSITIONS

It is obvious, but worth emphasizing, that the sum of the individual component flows in any stream cannot exceed the total stream flow. Also, the sum of the individual molar or weight fractions must equal 1. Hence, the composition of a stream is completely defined if all but one of the component concentrations are given.

The component flows in a stream (or the quantities in a batch) are completely defined by any of the following:

1. Specifying the flow (or quantity) of each component.
2. Specifying the total flow (or quantity) and the composition.
3. Specifying the flow (or quantity) of one component and the composition.

Example 2.6

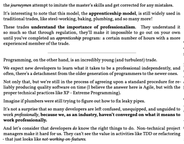
The feed stream to a reactor contains: ethylene 16%, oxygen 9%, nitrogen 31%, and hydrogen chloride. If the ethylene flow is 5000 kg/h, calculate the individual component flows and the total stream flow. All percentages are by weight.

Solution

$$\text{Percentage HCl} = 100 - (16 + 9 + 31) = \underline{\underline{44}}$$
$$\text{Percentage ethylene} = \frac{5000}{\text{total}} \times 100 = \underline{\underline{16}}$$
$$\text{hence total flow} = 5000 \times \frac{100}{16} = \underline{\underline{31,250\ \text{kg/h}}}$$
$$\text{so, oxygen flow} = \frac{9}{100} \times 31,250 = \underline{\underline{2813\ \text{kg/h}}}$$
$$\text{nitrogen} = 31,250 \times \frac{31}{100} = \underline{\underline{9687\ \text{kg/h}}}$$
$$\text{hydrogen chloride} = 31,250 \times \frac{44}{100} = \underline{\underline{13,750\ \text{kg/h}}}$$

General rule: the ratio of the flow of any component to the flow of any other component is the same as the ratio of the compositions of the two components.

Books on software design and architecture:



the *journeymen* attempt to imitate the master's skills and get corrected for any mistakes.

It's interesting to note that this model, the **apprenticeship model**, is still widely used in traditional trades, like steel-working, baking, plumbing, and so many more!

These trades **understand the importance of professionalism**. They understand it so much so that through regulation, they'll make it impossible to go out on your own until you've completed an *apprenticeship* program: a certain number of hours with a more experienced member of the trade.

Programming, on the other hand, is an incredibly young (and turbulent) trade.

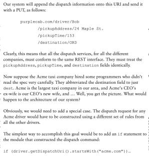
We expect new developers to learn what it takes to be a professional independently, and often, there's a detachment from the older generation of programmers to the newer ones.

Not only that, but we're still in the process of agreeing upon a standard procedure for reliably producing quality software on time (I believe the answer here is Agile, but with the proper technical practices like XP - Extreme Programming).

Imagine if plumbers were still trying to figure out how to fix leaky pipes.

It's not a surprise that so many developers are left confused, unequipped, and unguided to work *professionally*, **because we, as an industry, haven't converged on what it means to work professionally.**

And let's consider that developers *do* know the right things to do. Non-technical project managers make it hard for us. They can't see the value in activities like TDD or refactoring - that just looks like *not-working-on-features.*



Our system will append the dispatch information onto this URI and send it with a PUT, as follows:

```
purplecab.com/driver/Bob
      /pickupAddress/24 Maple St.
      /pickupTime/153
      /destination/ORD
```

Clearly, this means that all the dispatch services, for all the different companies, must conform to the same REST interface. They must treat the `pickupAddress`, `pickupTime`, and `destination` fields identically.

Now suppose the Acme taxi company hired some programmers who didn't read the spec very carefully. They abbreviated the destination field to just `dest`. Acme is the largest taxi company in our area, and Acme's CEO's ex-wife is our CEO's new wife, and … Well, you get the picture. What would happen to the architecture of our system?

Obviously, we would need to add a special case. The dispatch request for any Acme driver would have to be constructed using a different set of rules from all the other drivers.

The simplest way to accomplish this goal would be to add an `if` statement to the module that constructed the dispatch command:

```
if (driver.getDispatchUri().startsWith("acme.com"))…
```

6.2 Good Class Interfaces 133

multiple instances. For complex applications, this systemwide dependence on state means that you must keep track of the current font instance throughout code that uses the Font functions. Complexity tends to proliferate, and for applications of any size, better alternatives exist.

Inside the abstract data type, you'll have a wealth of options for handling multiple instances, but outside, this sums up the choices if you're working in a non-object-oriented language.

ADTs and Classes

Abstract data types form the foundation for the concept of classes. In languages that support classes, you can implement each abstract data type as its own class. Classes usually involve the additional concepts of inheritance and polymorphism. One way of thinking of a class is as an abstract data type plus inheritance and polymorphism.

6.2 Good Class Interfaces

The first and probably most important step in creating a high-quality class is creating a good interface. This consists of creating a good abstraction for the interface to represent and ensuring that the details remain hidden behind the abstraction.

Good Abstraction

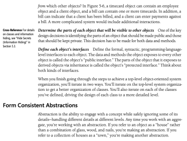
As "Form Consistent Abstractions" in Section 5.3 described, abstraction is the ability to view a complex operation in a simplified form. A class interface provides an abstraction of the implementation that's hidden behind the interface. The class's interface should offer a group of routines that clearly belong together.

You might have a class that implements an employee. It would contain data describing the employee's name, address, phone number, and so on. It would offer services to initialize and use an employee. Here's how that might look.

Cross-Reference Code samples in this book are formatted using a coding convention that emphasizes similarity of styles across multiple languages. For details on the convention (and discussions about multiple coding styles), see "Mixed-Language Programming Considerations" in Section 31.4.

C++ Example of a Class Interface That Presents a Good Abstraction

```
class Employee {
public:
   // public constructors and destructors
   Employee();
   Employee(
      FullName name,
      String address,
      String workPhone,
      String homePhone,
      TaxId taxIdNumber,
      JobClassification jobClass
   );
   virtual ~Employee();
```



from which other objects? In Figure 5-6, a timecard object can contain an employee object and a client object, and a bill can contain one or more timecards. In addition, a bill can indicate that a client has been billed, and a client can enter payments against a bill. A more complicated system would include additional interactions.

Cross-Reference For details on classes and information hiding, see "Hide Secrets (Information Hiding)" in Section 5.3.

Determine the parts of each object that will be visible to other objects One of the key design decisions is identifying the parts of an object that should be made public and those that should be kept private. This decision has to be made for both data and methods.

Define each object's interfaces Define the formal, syntactic, programming-language-level interfaces to each object. The data and methods the object exposes to every other object is called the object's "public interface." The parts of the object that it exposes to derived objects via inheritance is called the object's "protected interface." Think about both kinds of interfaces.

When you finish going through the steps to achieve a top-level object-oriented system organization, you'll iterate in two ways. You'll iterate on the top-level system organization to get a better organization of classes. You'll also iterate on each of the classes you've defined, driving the design of each class to a more detailed level.

Form Consistent Abstractions

Abstraction is the ability to engage with a concept while safely ignoring some of its details—handling different details at different levels. Any time you work with an aggregate, you're working with an abstraction. If you refer to an object as a "house" rather than a combination of glass, wood, and nails, you're making an abstraction. If you refer to a collection of houses as a "town," you're making another abstraction.

Questions that have rigorous answers

In engineering, certain questions about device design have rigorous answers
In FP, certain questions about code design have rigorous answers

- The answers are *not* a matter of opinion or experience
- The answers are found via mathematical derivations and reasoning
- The answers guide the design

Examples of reasoning tasks:

- Can we implement these APIs?

```
f :: (r -> Either z a) -> Either z (r -> a) -- Haskell
g :: Either z (r -> a) -> r -> Either z a
def f[Z, R, A](r: R => Either[Z, A]): Either[Z, R => A] // Scala
def g[Z, R, A](e: Either[Z, R => A]): R => Either[Z, A]
```

- Can use the data structure `F` as a monad in our code?

```
type F a = Maybe (a, a, a) -- Haskell
bind :: F a -> (a -> F b) -> F b
type F[A] = Option[(A, A, A)] // Scala
def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
```

A definition of “declarative”

Programming is “declarative” when *“specifications are programs”*

- “Being declarative” is not a property of a programming language alone

A language L is **declarative for an application domain** D if:

- The domain D has a good specification formalism F
  - “good” = visual, pragmatically convenient, complete for the domain D
- There is a syntactic translation from F to L
- The resulting program correctly implements the specification

Less formally:

- A declarative language is a “readable DSL” for the given domain

Example: declarative FORTRAN 77

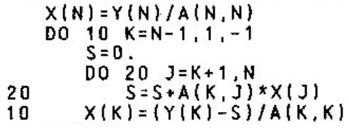
- Application domain: numerical mathematical expressions
- Specification: a mathematical formula involving *numbers* and *functions*
- Example specification:

$$f(x,p,q) = \frac{\sin px}{x^2} - \frac{(\sin qx)^2}{x^3}$$

  - Implementation: `F(X,P,Q) = SIN(P*X)/X**2-(SIN(Q*X))**2/X**3`

- For more complicated tasks, FORTRAN is not declarative

$$\tilde{X}_k = Y_k - \sum_{j=k+1}^{n} A_{kj} X_j, \quad \forall k \in [1..n]$$

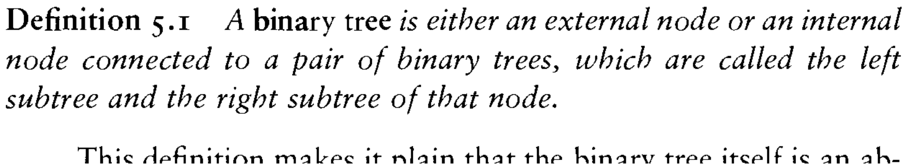


```
      X(N)=Y(N)/A(N,N)
      DO 10 K=N-1,1,-1
         S=0.
         DO 20 J=K+1,N
20          S=S+A(K,J)*X(J)
10       X(K)=(Y(K)-S)/A(K,K)
```

(example code, 1987)

Example: declarative Haskell 98

- Application domain: recursively defined, algebraic data structures
- Specifications: inductive definitions of functions on ADTs
- Example (from R. Sedgewick, *Algorithms in C*, 1998)



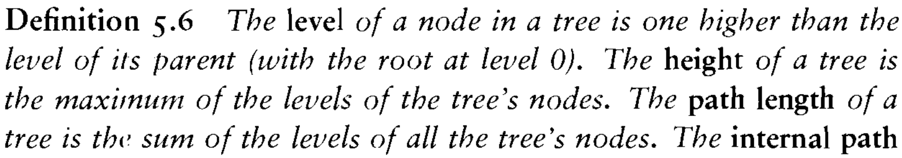
Definition 5.1 *A* **binary tree** *is either an external node or an internal node connected to a pair of binary trees, which are called the left subtree and the right subtree of that node.*

This definition makes it plain that the binary tree itself is an ab-

```
data BTree α = BTNode α | BTVertex α (BTree α) (BTree α)
```

```
enum BTree[+A]:     //Scala
  case BTNode[A](a:  A)
  case BTVertex[A](a:  A, left:  BTree[A], right:  BTree[A])
```

Example: declarative Haskell 98, continued



Definition 5.6 *The* **level** *of a node in a tree is one higher than the level of its parent (with the root at level 0). The* **height** *of a tree is the maximum of the levels of the tree's nodes. The* **path length** *of a tree is the sum of the levels of all the tree's nodes. The* **internal path**

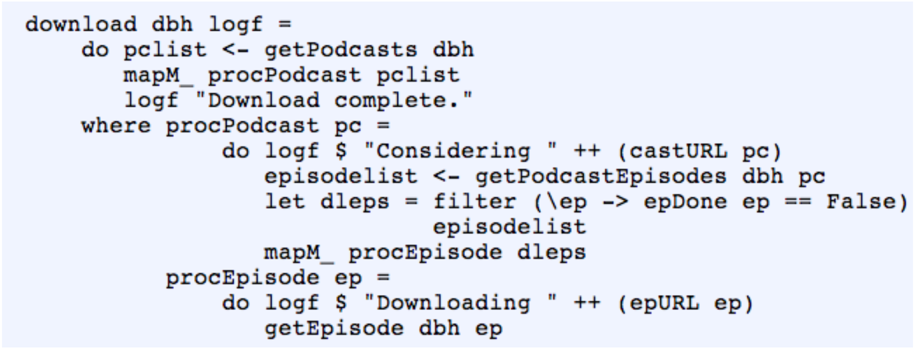
```
height ::  BTree α → Int     -- Haskell
height (BTNode _) = 0
height (BTVertex _ t1 t2) = 1 + max (height t1) (height t2)

def height[A]: BTree[A] => Int = {     //Scala
    case BTNode(_) => 0
    case BTVertex(_, t1, t2) =>
      1 + math.max(height(t1), height(t2))
}
```

Example: non-declarative Haskell

For a different application domain, Haskell is *not* declarative!

- Downloading data from server (from "*Real World Haskell*", 2008)



```
download dbh logf =
    do pclist <- getPodcasts dbh
       mapM_ procPodcast pclist
       logf "Download complete."
    where procPodcast pc =
              do logf $ "Considering " ++ (castURL pc)
                 episodelist <- getPodcastEpisodes dbh pc
                 let dleps = filter (\ep -> epDone ep == False)
                             episodelist
                 mapM_ procEpisode dleps
          procEpisode ep =
              do logf $ "Downloading " ++ (epURL ep)
                 getEpisode dbh ep
```

Declarative programming: Stories of success

Success stories (achieved the goals of declarative programming)

- Infix arithmetic: numerical math
- SQL: relational queries
- Autolayout: GUI layout on iOS and MacOS
- Haskell, Scala: Parsing, type-checking, evaluation of DSLs
- Prolog: logic puzzles (next slide)
  - for comparison, see: Matrix multplication with Prolog
- Chemical Abstract Machine: dining philosophers problem

Prolog as a DSL for logic puzzles

All jumping creatures are green. All small jumping creatures are martians. All green martians are intelligent. Ngtrks is small and green. Pgvdrk is a jumping martian. Who is intelligent? (inpired by *Invasion from Aldebaran*)

```
$ cat > martians.pl
green(X) :- jumping(X).
martian(X) :- small(X), jumping(X).
intelligent(X) :- green(X), martian(X).
small(ngtrks). green(ngtrks).
jumping(pgvdrk). martian(pgvdrk).
question :-
  intelligent(X), format('~w is intelligent.~n', X), halt.
^D
$ brew install swi-prolog
$ swipl -o martians -q -t question -c martians.pl
$ ./martians
pgvdrk is intelligent.
```

Declarative programming: Pitfalls

When does declarative programming fail:

- Declarative language is used outside its domain
- Specifications become too large to understand
  - Make specifications modular and understandable in isolation
    - Example: domain-specific notations in mathematical sciences
  - Make DSL languages and programs small, and keep them small

Signs that programming becomes non-declarative:

- Lots of code has no clear mapping to the problem domain
  - Example: "dining philosophers" in most programming languages
- Code appears to say one thing but does another thing when run
  - "To shutdown the computer, press the **Start** button"
  - Leaky abstractions. What does `f()` return?

```
var x = 0;    // JavaScript
function f() { return x; } // Returns zero?
// But what if another module does this:
var y = f(); y = "abc";
```

Declarative programming: Stories of failure

Failure stories (did not turn out to be declarative)

- Programming languages resembling English (COBOL, AppleScript)
  - Programs become long and unreadable
- Languages where everything is a built-in feature (PL/I, ABAP)
  - Features interact in unforeseen ways, corner cases lead to bugs
- "Mark-up" languages (XML, HTML)
  - Used mostly outside their domain of declarativeness

Failures of XML and HTML

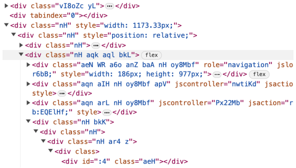
Failed due to predominant usage *outside* their designated domain

Intended usage (SGML, XML, HTML): plain text with occasional mark-up

```
<p> Hello! This is the home page of John Doe, a graduate
student at the Department of Electrical Engineering and
Computer Science, University of California, Los Angeles.
<p> You can click <a href="cv.html">here</a> to read
my CV. This page is under construction! Good bye!
```

In real life:

- XML: used as a data representation language (SOAP, config files)
- HTML: used as a GUI layout language for the Web



```
▶<div class="vI8oZc yL">⋯</div>
 <div tabindex="0"></div>
▼<div class="nH" style="width: 1173.33px;">
  ▼<div class="nH" style="position: relative;">
    ▶<div class="nH">⋯</div>
    ▼<div class="nH aqk aql bkL"> flex
      ▶<div class="aeN WR a6o anZ baA nH oy8Mbf" role="navigation" jslo
       r6bB;" style="width: 186px; height: 977px;">⋯</div> flex
      ▶<div class="aqn aIH nH oy8Mbf apV" jscontroller="nwtiKd" jsactio
       style>⋯</div>
      ▶<div class="aqn arL nH oy8Mbf" jscontroller="Px22Mb" jsaction="r
       b:EQElHf;" style>⋯</div>
      ▼<div class="nH bkK">
        ▼<div class="nH">
          ▼<div class="nH ar4 z">
            ▼<div class>
               <div id=":4" class="aeH"></div>
```

The joy of implementing DSLs in FP languages

Two choices for implementing a DSL:

- Embedded DSL
- External DSL

In both cases, FP works great

- For embedded DSLs:
  - Free monads, GADTs, non-leaky abstractions, strict typing
- For external DSLs:
  - Parser combinators, recursion schemes, GADTs, HOAS / PHOAS

Anecdotal evidence: one-person languages with compilers in Haskell

- Agda (Ulf Norell, 2007)
- Idris (Edwin Brady, 2007)
- Elm (Evan Czaplicki, 2012)
- PureScript (Phil Freeman, 2013)
- Dhall (Gabriella Gonzalez, 2016)
  - Time to re-implement Dhall in Scala: 2 months, 4K LOC

Conclusions

Functional programming has a steep learning curve

- Using FP techniques makes programmers' work closer to *engineering*
- Most artisans don't want to become engineers

Declarative programming means a symbiosis between human tradition and formal mathematics

- Best implemented by math-based programming paradigms
  - FP and logic programming

Implementing DSLs is one of FP's "killer apps"

- Even a small DSL is a productivity boost when it is declarative for the chosen domain
- FP languages are simple to implement once you get the theory