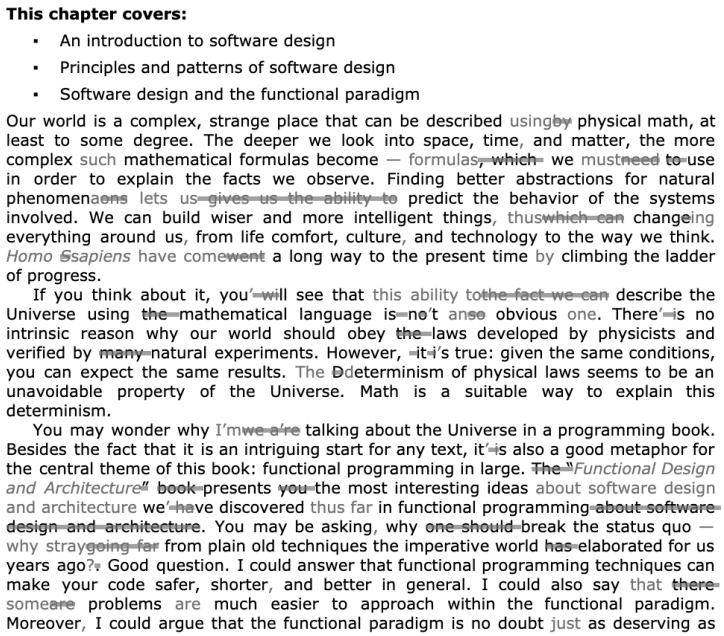
Preamble

Hello dear reader!

This is a draft of my book available for a free reading. This version of the book is incomplete, unedited, not properly styled. It won't be updated. Consider buying the book, and you'll get a complete text revised by a professional editor. You'll also get some additional materials such as educational videos.

https://leanpub.com/functional-design-and-architecture

Here is the example of how deeply the text was edited:



This chapter covers:

- An introduction to software design
- Principles and patterns of software design
- Software design and the functional paradigm

Our world is a complex, strange place that can be described usingby physical math, at least to some degree. The deeper we look into space, time, and matter, the more complex such mathematical formulas become — formulas, which we mustneed to use in order to explain the facts we observe. Finding better abstractions for natural phenomenaons lets us gives us the ability to predict the behavior of the systems involved. We can build wiser and more intelligent things, thuswhich can changeing everything around us, from life comfort, culture, and technology to the way we think. *Homo Ssapiens* have comewent a long way to the present time by climbing the ladder of progress.

If you think about it, you'will see that this ability tothe fact we can describe the Universe using the mathematical language isno't anso obvious one. There'is no intrinsic reason why our world should obey the laws developed by physicists and verified by many natural experiments. However, it'is true: given the same conditions, you can expect the same results. The Ddeterminism of physical laws seems to be an unavoidable property of the Universe. Math is a suitable way to explain this determinism.

You may wonder why I'mwe a're talking about the Universe in a programming book. Besides the fact that it is an intriguing start for any text, it'is also a good metaphor for the central theme of this book: functional programming in large. The "*Functional Design and Architecture*" book presents you the most interesting ideas about software design and architecture we'have discovered thus far in functional programming about software design and architecture. You may be asking, why one should break the status quo — why straygoing far from plain old techniques the imperative world has elaborated for us years ago?. Good question. I could answer that functional programming techniques can make your code safer, shorter, and better in general. I could also say that there someare problems are much easier to approach within the functional paradigm. Moreover, I could argue that the functional paradigm is no doubt just as deserving as

4 Domain model design

This chapter covers:

- How to analyze a domain mnemonically and find its general properties
- How to model the domain in different embedded DSLs
- Combinatorial languages and domain modeling
- External language parsing and translation

When Sir Isaac Newton invented his theory of gravitation, he probably never imagined the basis of that theory would be significantly reworked in the future, involving an entirely different set of mathematical abstractions. The theory, while being inapplicable to very massive or very fast objects, still works fine for a narrower scale of conditions. The Newton's apple will fall down accurately following the law of universal gravitation if it has a low speed or falls to the relatively low mass object. What abstraction we should choose for our calculations — Newton's classical mechanics or Einstein's more modern theory of relativity — depends on the problem we want to solve. It's important to consider whether we are landing our ship on the Moon or traveling near the black hole in the center of our galaxy, because the conditions are very different, and we must be sure our formulas are appropriate to give us an adequate result. After all, our lives depend on the right decision of what abstraction to choose so we must not be wrong.

Software developers do make wrong decisions, and it happens more often than it actually should. Fortunately, software development is not typically so fatalistic. People's lives don't usually depend on the code written — usually, but not always. That's definitely not true if we talk about sensitive domains like SCADA (Supervisory Control and Data Acquisition) and nuclear bomb calculations. Every developer must be responsible and professional to decrease the risks of critical software errors, but the probability of such will never become zero (unless we shift to formal methods to prove the correctness of the program, which is really hard). So why then do we ignore the approaches that are intended to help developers write valid, less buggy code? Why do we commit to technologies that often don't prove to be good for solving our regular tasks? The history of development has many black pages where using the wrong abstractions ruined great projects. Did you know there was a satellite called the Mars Climate Orbiter that burned up in that planet's atmosphere because of a software bug? The problem was a measurement unit mismatch, where the program returned a pound-seconds measurement but it should have been newton-seconds. This bug could have been eliminated

by testing or by static type-level logic. It seems the developers had missed something important when they were programming this behavior.

Abstractions can save us from bugs in two ways.

- By making it simple to reason about the domain; in other words, decreasing accidental complexity (which is good), so that we can easily see the bugs.
- By making it impossible to push the program into an invalid state; in other words, encoding a domain so that only correct operations are allowed.

We call a domain the knowledge of how a certain object or process works in the real world. Domain model is a representation of a domain. Domain model includes more or less formal definition of data and transformations of that domain. We usually deal with domain models expressed by code, but also it can be a set of diagrams or a specific domain language. In this chapter we'll study how to design a domain model, what are the tools do we have to make it correct and simple. While it's certainly fine to want correctness in software but it's not so obvious why unnecessary complex abstractions may lead you to bugs. The main reason here is that we lose the focus of the domain we are implementing and start treating the abstraction as an universal hammer that we think may solve all our problems at once. You probably know the result when the project suddenly falls into abstraction hell and no one piece of domain becomes visible through it. How can you be sure all of the requirements are handled? In this situation it's more likely you'll find many bugs you could avoid by having a fine-readable and simple but still adequate abstraction. The abstractions shouldn't be too abstract, otherwise they tend to leak and obfuscate your code. We'll discuss domain modeling as the main approach to follow and see what abstractions we have for this. We'll also continue developing the ideas we touched on in the previous chapters.

Roll up your sleeves; the hard work on the Logic Control subsystem is ahead. So far, the Logic Control eDSL we wrote as a demonstration of domain-driven modeling seems to be naive and incomplete as it doesn't cover all of the corresponding domains. The scope of work in this chapter includes:

1. Define the domain of Logic Control.
2. Describe the Logic Control subsystem's functional requirements and responsibilities.
3. Implement the domain model for Logic Control in the form of embedded DSLs.
4. Elaborate the combinatorial interface to the Logic Control subsystem.
5. Create an external representation of the Logic Control DSL.
6. Test the subsystem.

The first goal of this chapter is to create the code that implements most of the functional requirements of Logic Control. The second goal is to learn domain model design and new functional concepts that are applicable at this scale of software development. Keep in mind that we are descending from the architecture to the design of subsystems and layers in a top–bottom development process. We discussed the approaches to modularizing the application in chapter 2, and we even tried out some ideas of domain modeling on the Logic Control and Hardware subsystems, but not all questions were clarified. Let's clarify it, then.

4.1 Defining the domain model and requirements

What is a domain? What is a domain model? We all have an intuitive understanding of these terms just because every time we code we implement a tool for solving problems in some domain. By doing this, we want to ease tasks, and maybe introduce automation into them. Our tool should simplify solving of hard tasks and make it possible to solve tasks one cannot solve without computers at all. As we plunged into the SCADA field, a good example of this will be manufacturing automation – a SCADA domain in which a quality and an amount of production play the leading role. But this is what our developer's activity looks like from the outside, to the customer. In contrast, when we discuss a domain, we are interested in the

details of its internal structure and behavior, what notions we should take into consideration and what things we can ignore. Then we start thinking about the domain model; namely, what data structures, values, and interactions could represent these notions and behaviors in order to reflect them properly. This is what we have to learn to do every day. In fact, we already got familiar with domain modeling – a process we participate in to design domain models. We just need more tools to do that right.

As usual, we'll start from requirements, building on what we described for Logic Control earlier. This step is very important to better understand what we should do, and how. Remember how we converted requirements into a domain-specific language for hardware definition in chapter 3? We can't be successful in functional domain modeling if we haven't sorted out the real nature of the domain and its notions.

The Logic Control subsystem encapsulates code that covers a relatively big part of the spaceship domain. The subsystem provides a set of tools (functional services, eDSLs) for the following functionality (partially described in the mind map in figure 2.7):

1. Reading actual data from sensors.
2. Accessing archive data and parameters of the hardware under control.
3. Handling events from other subsystems and from hardware.
4. Running control scripts and programs. Scripts can be run by different conditions:
   - By Boolean condition
   - By exact time or by hitting a time interval
   - By event
   - By demand from other subsystems
   - Periodically with a time interval
5. Accessing the hardware schema of a spaceship.
6. Monitoring of spaceship state and properties.
7. Autonomously correcting spaceship parameters according to predefined behavioral programs.
8. Sending commands to control units.
9. Evaluating mathematical calculations.
10. Handling hardware errors and failures.
11. Testing scenarios before pushing them into a real environment.
12. Abstracting different hardware protocols and devices.
13. Logging and managing security logs.

Characterizing this list of requirements as a comprehensive domain description would be a mistake, but it's what we have for now. You probably noticed that we discover requirements in increments while descending from the big scale of application design to the details of particular subsystems and even modules. This gives us a scope of the functionality that we know we can implement immediately. In real development you'll probably want to focus on one concrete subsystem until you get it working properly. But it's always probable you'll end up with a malformed and inconsistent design the rest of the subsystems don't match with, and then you'll be forced to redesign it when the problems come out.

4.2 Simple embedded DSLs

It would be perfect to unify all this domain stuff with a limited yet powerful domain-specific language. You may ask, why a DSL? Why not just implement it “as is” using functional style?

In talking about design decisions like this one, we should take into consideration the factors discussed in chapter 1. Will it help to achieve goals? Can a DSL give the desired level of simplicity? Does the solution cover all the requirements? Do we have human resources and money to support this solution in the later stages of the software lifecycle?

The DSL approach addresses one main problem: the smooth translation of the domain's essence into the code without increasing accidental complexity. Done right, a DSL can replace tons of messy code, but of course, there is a risk that it will restrict and annoy the user too much if it's designed badly. Unfortunately, there are no guarantees that introducing a DSL will be a good decision. Fortunately, we don't have any choice: the SCADA software should be operated by a scripting language. We just follow the requirements.

In this section we'll take several approaches and see if they are good to model embedded languages. The first one is rather primitive and straightforward: "a pyramid" of plain functions. It's fine, if you may fit all the domains in one page. Otherwise, you'd better try algebraic data types. In this case, your algebraic structures encode every notion of a domain, from objects to processes and users. Depending on your taste, you may design it more or less granular. It's fine if you just keep data in these structures, but you still need "a pyramid" of plain functions to do related stuff. Otherwise, you'll better try a combinatorial approach that is described in the next section.

4.2.1 Domain model eDSL using functions and primitive types

In functional programming, everything that has some predefined behavior can be implemented by a composition of pure and impure functions that operate on the primitive types. Following is the example produces a Morse encoded FizzBuzzes – a simple problem we all know very well:

```
import Data.Char (toLower)

fizzBuzz :: Int -> String
fizzBuzz x | (x `mod` 15) == 0 = "FizzBuzz"
           | (x `mod` 5)  == 0 = "Buzz"
           | (x `mod` 3)  == 0 = "Fizz"
           | otherwise = show x

morseCode :: [(Char, String)]
morseCode =
  [ ('b', "-..."), ('f', "..-."), ('i', ".."), ('u', "..-")
  , ('z', "--.."), ('0', "-----"), ('1', ".----"), ('2', "..---")
  , ('3', "...--"), ('4', "....-"), ('5', "....."), ('6', "-....")
  , ('7', "--..."), ('8', "---.."), ('9', "----.") ]

toMorse :: Char -> String
toMorse char = case lookup char morseCode of
    Just code -> code
    Nothing -> "???"

morseBelt :: Int -> [String]
morseBelt = map (' ' :) . map toMorse . map toLower . fizzBuzz

morseFizzBuzzes :: String
morseFizzBuzzes = (concat . concatMap morseBelt) [1..100]

main = putStrLn morseFizzBuzzes
```

You see here a long functional conveyor of transformations over primitive data types – the `morseBelt` function that takes an integer and returns a list of strings. Four separate functions are composed together by composition operator `(.)`: each of them does a small piece of work and passes the result to the next workshop, from right to left. The transformation process starts from a fizzBuzz function that converts a number to FizzBuzz string. Then the function `map toLower` takes the baton and lowercases every letter by mapping over the string. Going further, the lowercased string becomes a list of Morse encoded strings, and the last function `(' ' :)` pads it by spaces. We strongly feel composition of functions like this one as functional code.

> **DEFINITION** A *functional eDS*L is an embedded domain-specific language that uses functional idioms and patterns for expressing domain notions and behavior. A functional eDSL should contain a concise set of precise combinators that do one small thing and are able to be composed in a functional manner.

On the basic level of "functional programming Jediism" you don't even need to know any advanced concepts coming from mathematical theories, because these concepts serve the purpose of unifying code patterns, to abstract behavior and to make the code much more safe, expressive, and powerful. But the possibility of just writing functions over functions is still there. By going down this path, you'll probably end up with verbose code that will look like a functional pyramid (see figure 1.10 in chapter 1). It will work fine, though. Moreover, the code can be made less clumsy if you group functions together by the criterion that they relate to the same part of the domain, regardless of whether this domain is really the domain for what the software is, or the auxiliary domain of programming concepts (like the message-passing system, for example).

Let's compose an impure script for obtaining `n` values from a sensor once a second. It gets current time, compares it with the old time stamp, and if the difference is bigger than the delay desired, it reads the sensor and decrements a counter. When the counter hits zero, all n values are read. Here is a Haskell-like pseudocode:

```
-- Function from system library:
getTime :: IO Time
-- Function from hardware-related library:
readDevice :: Controller -> Parameter -> IO Value

-- "eDSL" (not really):
readNEverySecond n controller parameter = do
    t <- getTime
    out <- reading' n ([], t) controller  parameter
    return (reverse out)

-- Auxiliary recursive function:
reading' 0 (out, _) _ _ = return out
reading' n (out, t1) controller parameter = do
    t2 <- getTime
    if (t2 - t1 >= 1)
    then do
        val <- readDevice controller parameter
        reading' (n-1) (val:out, t2) controller  parameter
    else reading' n (out, t1) controller name
```

This whole code looks ugly. It wastes CPU time by looping indefinitely and asks time from the system very often. It can't be normally configured by a custom time interval, because the `readNEverySecond` function is too specific. Imagine how many functions will be in our eDSL for different purposes!

```
readNEverySecond
readTwice
readEveryTwoSecondsFor10Seconds
...
```

And the biggest problem with this eDSL is that our functions aren't that handy for use in higher-level scenarios. Namely, these functions violate the Single Responsibility Principle: there are mixed responsibilities of reading of a sensor, counting the values and of asking the time. This DSL isn't combinatorial, because it doesn't provide any functions with single behavior you might use as a constructor to solve your big task. The code above is an example of spagetti-like functional code.

Let's turn the preceding code into a combinatorial language. The simplest way to get composable combinators is to convert the small actions into the higher-order functions, partially applied functions, and curried functions:

```
-- Functions from system libraries:
threadDelay :: DiffTime -> IO ()
replicate :: Int -> a -> [a]
sequence :: Monad m => [m a] -> m [a]
(>>=) :: Monad m => m a -> (a -> m b) -> m b

-- Function from hardware-related library:
readDevice :: Controller -> Parameter -> IO Value

-- eDSL:
delay :: DiffTime -> Value → IO Value
delay dt value = do
    threadDelay dt
    return value

times :: Int -> IO a -> IO [a]
times n f = sequence (replicate n f)

-- Script:
readValues :: DiffTime -> Int -> Controller -> Parameter -> IO [Value]
readValues dt n controller param = times n (reader >>= delayer)
    where
        reader = readDevice controller param
        delayer = delay dt
```

Let's characterize this code:

- *Impure*. It makes unit testing hard or even impossible.
- *Instantly executable*. We stated earlier in the previous chapter that interpretable languages give us another level of abstraction. You write a script but it really can't be executed immediately but rather it should be translated into executable form first and then executed. This means your script is declarative. The closest analogy here is string of code the `eval` function will evaluate in such languages as PHP and JS. So you don't

describe your domain in a specific language, you program your domain in the host language.[1]

- *Vulnerable*. Some decisions (like `threadDelay`, which blocks the current thread) can't be considered acceptable; there should be a better way. Indeed, we'll see many ways better than the eDSL implemented as shown here.

Interestingly, every part of functional code that unites a set of functions can be called an internal DSL for that small piece of the domain. From this point of view, the only measure or a function to be a part of a DSL is to reflect its notion with the appropriate naming.

4.2.2 Domain model eDSL using ADTs

It's hard to imagine functional programming without algebraic data types. ADTs cover all your needs when you want to design a complex data structure — tree, graph, dictionary, and so forth — but they are also suitable for modeling domain notions. Scientifically speaking, an ADT can be thought of as "a sum type of product types," which simply means "a union of variants" or "a union of named tuples." Algebraic type shouldn't necessarily be an exclusive feature of a functional language, but including ADTs into any language is a nice idea due to good underlying theory. Pattern matching, which we already dealt with in the previous chapter, makes the code concise and clean, and the compiler will guard you from missing variants to be matched.

Designing the domain model using algebraic data types can be done in different ways. The `Procedure` data type we developed in chapter 2 represents a kind of straightforward, naive approach to sequential scenarios. The following listing shows this type.

Listing 4.1 Naive eDSL using algebraic data type

```
-- These types are defined by a separate library
data Value = FloatValue Float
           | IntValue Int
           | StringValue String
           | BoolValue Bool
           | ListValue [Value]

-- Dummy types, should be designed later
data Status = Online | Offline
data Controller = Controller String
data Power = Float
type Duration = Float
type Temperature = Kelvin Float

data Procedure
    = Report Value
    | Store Value
    | InitBoosters (Controller -> Script)
    | ReadTemperature Controller (Temperature -> Script)
    | AskStatus Controller (Status -> Script)
    | HeatUpBoosters Power Duration
```

[1] Well, the line between programming of a domain and describing it using an eDSL is not that clear. We can always say that the I/O action isn't really a procedure, but a definition of a procedure that will be evaluated later.

```
type Script = [Procedure]
```

The “procedures” this type declares are related to some effect: storing a value in a database, working with boosters after the controller is initialized, reporting a value. But they are just declarations. We actually don't know what real types will be used in runtime as a device instance, as a database handler and controller object. We abstract from the real impure actions: communicating with databases, calling library functions to connect to a controller, sending report messages to remote log, and so on. This language is a declaration of logic because when we create a value of the `Script` type, nothing actually happens. Something real will happen when we'll bind these actions to real functions that do the real work. Notice also the sequencing of procedures is encoded as a list, namely the `Script` type. This is how a script may look:

```
storeSomeValues :: Script
storeSomeValues =
    [ StoreValue (FloatValue 1.0)
    , StoreValue (FloatValue 2.0)
    , StoreValue (FloatValue 3.0) ]
```

This script may be transferred to the subsystem that works with the database. There should be an interpreting function that will translate the script into calls to real database, something like that:

```
-- imaginary bindings to database interface:
import Control.Database as DB

-- interpreter:
interpretScript :: DB.SQLiteConnection -> Script -> IO ()
interpretScript conn [] = return ()
interpretScript conn (StoreValue v : procs) = do
    DB.store conn "my_table" (DB.format v)
    interpretScript procs
interpretScript conn (unknownProc : procs) = interpretScript procs
```

It should be clear why three of value constructors (`Report`, `Store`, `HeatUpBoosters`) have arguments of a regular type. We pass some useful information the procedure should have to function properly. We don’t expect the evaluation of these procedures will return something useful. However, the other three procedures should produce a particular value when they are evaluated. For example, the procedure of boosters initialization should initialize the device and then return a sort of handle to its controller. Or one more example: being asked for temperature, that controller should give you a measured value back. To reflect this fact, we declare additional fields with function types in the “returning” value constructors: `(Controller -> Script)`, `(Status -> Script)` and (`Temperature -> Script`). We also declare the abstracted types for values that should be returned (`Controller`, `Status`, `Temperature`). This doesn't mean the particular subsystem will use exactly these types as runtime types. It's more likely the real interpreter when it meets a value of `Controller` type will transform it to its own runtime type, let's say, `ControllerInstance`. This `ControllerInstance` value may have many useful fields such as time of creation, manufacturer, GUID, and we don't have to know about that stuff. The only thing we should know is that successful interpretation of `InitBoosters` should return some handle to the controller. But why do we use function type `(Controller -> Script)` to declare this behavior? The reason is we want to do something with the value

returned. We may want to read temperature using the controller instance. This means we want to combine several actions, make them chained, dependent. This is easily achievable if we adopt the same ideas we discussed in the previous chapter: we use recursive nesting of continuations for this purpose. The field `(Controller -> Script)` of the `InitBoosters` will hold a lambda that declares what to do with the controller handle we just obtained. By the design of the language, all we can do now is to read the temperature or to ask the status. The following demonstrates a complex script:

```
doNothing :: Temperature -> Script
doNothing _ = []

readTemperature :: Controller -> Script
readTemperature controller = [ ReadTemperature controller doNothing ]

script :: Script
script = [ InitBoosters readTemperature ]
```

Now all three scripts are combined together. For simplicity we may say that the `InitBoosters` procedure "returns" `Controller`, `ReadTemperature` "returns" `Temperature`, and `AskStatus` "returns" the controller's status

Listing 4.2 gives you one more example that contains a script for the following scenario:

1. Initialize boosters and get a working controller as result.
2. Read current temperature using the controller. Process this value: report it and store it in the database.
3. Heat up boosters for 10 seconds with power 1.
4. Read current temperature using the controller. Process this value: report it and store it in the database.

Listing 4.2 Script for heating the boosters and reporting the temperature

```
-- Step 1 of the scenario.
initAndHeatUpScript :: Script
initAndHeatUpScript = [ InitBoosters heatingUpScript ]

-- Steps 2, 3 and 4 of the scenario.
heatingUpScript :: Controller -> Script
heatingUpScript controller =
    [ ReadTemperature controller processTemp
    , HeatUpBoosters 1.0 (seconds 10)
    , ReadTemperature controller processTemp ]

-- Scripts for steps 2 and 4.
reportAndStore :: Value -> Script
reportAndStore val = [ Report val, Store val ]

processTemp :: Temperature -> Script
processTemp t = reportAndStore (temperatureToValue t)
```

An impure testing interpreter we wrote earlier (see listing 2.5) traces every step of the script to the console. The following listing shows the interpretation for the script `initAndHeatUpScript`:

```
"Init boosters"
"Read temperature"
"Report: FloatValue 0.0"
"Store: FloatValue 0.0"
"Heat up boosters"
"Read temperature"
"Report: FloatValue 0.0"
"Store: FloatValue 0.0"
```

Unfortunately, the approach of modeling a domain 1:1 in algebraic data types has many significant problems:

- *It's too object-oriented*. Types you design by copying the domain notions, will tend to look like "classes" (the `Controller` type) and "methods" (the `ReadTemperature` value constructor). A desire to cover all of the domain notions will lead you to the notions-specific code because it's less likely you'll see abstract properties of your data by exploiting which you could probably join several notions into one generic. For example, the procedures `Store` and `Report` may be generalized by just one `(SendTo Receiver Value)` procedure that is configured by the specific receiver: either a database or reporter or something else you need. The `Receiver` type can be a lambda that knows what to do: `(type Receiver :: Value -> IO ())`, however your domain doesn't have this exact object, and you should invent it yourself.
- *Different scopes are mixed in just one God-type* `Procedure`. It's easy to dump everything into a single pile. In our case, we have two scopes that seem to be separate: the procedures for working with the controller and the reporting/storing procedures.
- *It's inconvenient*. Verbose lists as sequence of actions, value constructors of the `Procedure` type you have to place in your code, limits of the actions you may do with your list items, - all these issues restrict you too much.
- *It encodes a domain "as is."* The wider a domain is, the fatter the DSL will be. It's like when you create classes `DeadCell` and `AliveCell` inheriting them from the interface `IGameOfLifeCell`, and your class `Board` holds a `FieldGraph` of these objects which are connected by `GraphEdge` objects... And this whole complexity can be removed by just one good old two-dimensional array of short integers. If there is a lesser set of meaningful abstractions your domain can be described by, why to avoid them?
- *It's primitive*. The language doesn't provide any useful abstractions.

The issues listed can be summarized as the main weakness of this modeling approach: we don't see the underlying properties of a domain. Despite this, there are some good points here:

- *Straightforward modeling is fast*. It may be useful for rapid prototype development or when the domain is not so big. It also helps to understand and clarify the requirements.
- *It's simple*. There are only two patterns all the "procedures" should match: specifically, a procedure with a return type and a procedure without one. This also means the approach has low accidental complexity.

- *The eDSL is safe*. If the language says you can't combine two incompatible procedures, then you can't, really.
- *The eDSL is interpretable*. This property allows you to mock subsystems or process an eDSL in different ways.
- *The eDSL can be converted to a Free monad eDSL*. When it's done, the domain is effectively hidden behind the monadic interface so the client code won't be broken if you change the internal structure of your eDSL.

We will soon see how to decompose our domain into much smaller, consistent and high-cohesive parts than this eDSL has. We'll then investigate the properties these parts have. This should enlighten us by hinting at how to design a better, combinatorial eDSL.

4.3 Combinatorial eDSLs

Functional programming is finally entering the mainstream, its emergence stemming from three major directions: functional languages are becoming more popular, mainstream languages are extending their syntax with functional elements, and functional design patterns are being adopted by cutting-edge developers. While enthusiasts are pushing this wave, they are continuing to hear questions about what functional programming is and why people should care. The common use case of lambdas we see in the mainstream is passing simple operations into library functions that work with collections generically. For example, the operations over an abstract container in the C++ Standard Template Library may receive lambdas for comparison operators, for accumulation algorithms, and so on. But be careful about saying this kind of lambda usage is functional programming. It's not; it's just elements of functional programming but not a functional design. The essence of functional programming is composition of combinators and the functional idioms which make this composition possible. For example, the function `map :: (a -> b) -> [a] -> [b]` is a combinator that takes a function, a list and returns a new list with every element modified by that function. The function `map` is a combinator because you can combine several of them:

```
morseBelt :: Int -> [String]
morseBelt = map (' ' :) . map toMorse . map toLower . fizzBuzz
```

And you even may improve this code according to the rewriting rule:

```
map f . map g == map (f . g)

morseBelt' :: Int -> [String]
morseBelt' = map ((' ' :) . toMorse . toLower) . fizzBuzz
```

It would be wrong to say that a procedure that simply takes a lambda function (for instance, `std::sort()` in C++) is functional programming. The procedure isn't a combinator because it's not a function and therefore you can't combine it with something else. In fairness, the C++ Standard Template Library is the implementation of generic style programming that is close to functional programming, but many of the functions this library has imply both mutability and uncontrolled side effects. Immutability and side effects may ruin your functional design.

Functional programming abounds with embedded combinatorial languages. The accidental complexity of a language you design in combinatorial style is small due to the uniform way of reasoning about combinatorial code, regardless of the size of the combinators. Have you heard of the Parsec library, perhaps the best example of a combinatorial language? Parsec is a library of monadic parsing combinators. Every monadic parser it provides parses one small

piece of text. Being monadic functions in the `Parser` monad, parsers can be combined monadically into a bigger monadic parser that is in no way different but works with a bigger piece of text. Monadic parsers give a look to a code like it's a Backus–Naur Form (BNF) of the structure you are trying to extract from text. Reading of such code becomes simple even for nonprogrammers after they have had a little introduction to BNF and monadic parsing concepts. Consider the following example of parsing constant statements. The text we want to parse looks so:

```
const thisIsIdentifier = 1
```

The following code shows parser combinators and the `Statement` structure in which we'll keep the result parsed:

```
import Text.Parsec as P
data Expr = …     -- some type to hold expression tree.
data Statement = ConstantStmt String Expr

constantStatement :: P.Parser Statement
constantStatement = do
    P.string "const"           -- parses string "const"
    P.spaces                   -- parses one or many spaces
    constId <- identifier      -- parses identifier
    P.spaces                   -- parses spaces again
    P.char '='                 -- parses char '='
    P.spaces                   -- parses spaces, too
    e <- expr                  -- parses expression
    return (ConstantStmt constId e)

str = "const thisIsIdentifier     =    1"
parseString = P.parse constantStatement "" str
```

Here, `identifier` and `expr` are the parser combinators having the same `Parser a` type:

```
identifier :: Parser String
expr :: Parser Expr
```

We just put useful stuff into variables and wrapped it in the `Statement` algebraic data type. The corresponding BNF notation looks very similar:

```
constantStatement ::= "const" spaces identifier spaces "=" spaces exprAnd
```

If we line every token of BNF it becomes even closer to the parser:

```
constantStatement ::=
    "const"
    spaces
    identifier
    spaces "=" spaces
    expr
```

We'll return to this theme in section 4.4. There we will build an external DSL with its own syntax. Parsec will help us to parse external scripts into abstract syntax trees that we then

translate to our combinatorial eDSL (going ahead, it will be a compound Free eDSL with many Free DSLs inside). Before that we should create an eDSL. Let's do this.

4.3.1 Mnemonic domain analysis

The `Procedure` data type was modeled to support the scenario of heating boosters, but we haven't yet analyzed the domain deeply because the requirements were incomplete. We just projected a single scenario into ADT structure one-to-one and got what we got: an inconsistent DSL with dissimilar notions mixed together. In this section we'll redesign this eDSL and wrap the result into several combinatorial interfaces — but we have to revisit the domain of Logic Control first.

We'll now try a method of analysis that states a scenario to be written in mnemonic form using various pseudolanguages. Determining the internal properties of domain notions can't be done effectively without some juggling of the user scenarios being written in pseudolanguages. The juggling can also lead you to surprising ideas about how to compose different parts uniformly or how to remove unnecessary details by adopting a general solution instead.

The next scenario we'll be working with collects several needs of the Logic Control subsystem (conditional evaluation, mathematical calculations, and handling of devices):

```
Scenario: monitor outside thermometer temperature
Given: outside thermometer @therm
Run: once a second

scenario:
    Read temperature from @therm, result: @reading(@time, @temp, @therm)
    If @temp < -10C Then
        register @reading
        log problem @reading
        raise alarm "Outside temperature lower than bound"
    Else If @temp > 50C Then
        register @reading
        log problem @reading
        raise alarm "Outside temperature higher than bound"
    Else register @reading

register (@time, @tempCelsius, @device):
    @tempCelsius + 273.15, result: @tempKelvin
    Store in database (@time, @tempKelvin, @device)
```

It should be clear that the scenario reads a thermometer and then runs one of the possible subroutines: registering the value in a database if the temperature doesn't cross the bounds; logging the problem and raising an alarm otherwise. The `register` subroutine is defined too. It converts the value from Celsius to Kelvin and stores it in the database along with additional information: the timestamp of the measurement and the device from which the value was read.

According to the Logic Control elements diagram (see figure 2.13), the instructions this scenario has can be generalized and distributed to small, separate domain-specific languages: Calculations DSL, Data Access DSL, Fault Handling DSL, and so on. Within one DSL, any instruction should be generalized to support a class of real actions rather than one concrete action. A language constructed this way will resist domain changes better than a language that reflects the domain directly. For example, there is no real sense in holding many different measurements by supporting a separate action for each of them, as we did

earlier, because we can make one general parameterized action to code a whole class of measurements:

```
data Parameter = Pressure | Temperature
data Procedure = Read Controller Parameter (Measurement -> Script)
```

where the `Parameter` type will say what we want to read.

Looking ahead...

In the Andromeda project the type `Procedure` looks different. The `Read` value constructor has one more field with type `ComponentIndex` that is just `ByteString`. It holds an index of a sensor inside a device to point to one of them that is plugged inside. The types `Parameter` and `Measurement` are different too. They have one extra type variable `tag`: (`Parameter tag`) and (`Measurement tag`) that is really a *phantom type* to keep the parameter and the measurement consistent. This phantom type ensures these two fields work with the same measurement units. For example, if we read temperature then we should write (`Parameter Kelvin`) and we'll then get the value of (`Measurement Kelvin`) by no exception. This is the theme of the chapter about type logic calculations. If you are interested, the `Procedure` type is presented below. Don't be scared by `forall` keyword, you may freely ignore it now.

```
data Procedure a
    = Get Controller Property (Value -> a)
    | Set Controller Property Value a
    | forall tag. Read
        Controller ComponentIndex (Parameter tag) (Measurement tag -> a)
    | Run Controller Command (CommandResult -> a)
```

A different conclusion we may draw from the scenario is that it's completely imperative. All the parts have some instructions that are clinging to each other. This property of the scenario forces us to create a sequential embedded domain language, and the best way to do this is to wrap it in a monad. We could use the `IO` monad here, but we know how dangerous it can be if the user of our eDSL has too much freedom. So we'll adopt a better solution—namely, the `Free` monad pattern—and see how it can be even better than we discussed in chapter 3.

However, being sequential is not a must for domain languages. In fact, we started thinking our scenario was imperative because we didn't try any other forms of mnemonic analysis. Let's continue juggling and see what happens:

```
Scenario: monitor outside thermometer temperature
Given: outside thermometer @therm

// Stream of measurements of the thermometer
stream therm_readings <once a second>:
    run script therm_temperature(), result: @reading
    return @reading

// Stream of results of the thermometer
stream therm_monitor <for @reading in therm_readings>:
    Store in database @reading
    run script validate_temperature(@reading), result: @result
```

```
    If @result == (Failure, @message) Then
        log problem @reading
        raise alarm @message
    return @result

// Script that reads value from the thermometer
script therm_temperature:
    Read temperature from @therm, result: @reading(@time, @tempCelsius,
@therm)
    @tempCelsius + 273.15, result: @tempKelvin
    return (@time, @tempKelvin, @therm)

// Script that validates temperature
script validate_temperature (@time, @temp, @therm):
    If @temp < 263.15K Then
        return (Failure, "Outside temperature lower than bound for " +
@therm)
    Else If @temp > 323.15K Then
        return (Failure, "Outside temperature higher than bound for " +
@therm)
    Else return Success
```

Oh, wow! This scenario is more wordy than the previous one, but you see a new object of interest here — a stream. Actually, you see two of them: the `therm_readings` stream that returns an infinite set of measurements and the `therm_monitor` stream that processes these values and does other stuff. Every stream has the evaluation condition: once a second or whenever the other stream is producing a value. This makes the notion of a stream different from a script: the former works periodically and infinitely, whereas the latter should be called as a single instruction.

This form of mnemonic scenario opens a door to many functional idioms. The first, which is perhaps obvious, is functional reactive streams. These streams run constantly and produce values you can catch and react to. "Functionality" of streams means you can compose and transform them in a functional way. Reactive streams are a good abstraction for interoperability code, but here we're talking about the design of a domain model rather than the architecture of the application. In our case, it's possible to wrap value reading and transforming processes into the streams and then construct a reactive model of the domain. The scenario gives a rough view of how it will look in code.

Functional reactive streams could probably be a beneficial solution to our task, but we'll try something more functional (and perhaps more mind-blowing): arrows and arrowized languages. The scenario doesn't reveal any evidence of this concept, but in fact, every function is an arrow. Moreover, it's possible to implement an arrowized interface to reactive streams to make them even more composable and declarative. Consequently, using this concept you may express everything you see in the scenario, like scripts, mathematical calculations, or streams of values, and the code will be highly mnemonic. So what is this mysterious concept of arrows? Keep reading; the truth is out there. First, though, we'll return to the simple and important way of creating composable embedded languages, using the `Free` monad pattern, and see how it can be improved.

4.3.2 Monadic Free eDSL

Any monad abstracts a chain of computations and makes them composable in a monadic way. Rolling your own monad over the computations you have can be really hard because not all sequential computations can be monads in a mathematical sense. Fortunately, there is a

shortcut that is called "the `Free` monad pattern." We discussed this pattern already in Chapter 3, and now we'll create another `Free` language that will be abstract, with the implementation details hidden. Let's revise the "free monadizing" algorithm. See table 4.1

Table 4.1: The algorithm of making a Free language

Algorithm step	Example

Create parameterized ADT for domain logic where type variable `a` is needed to make the type a Functor.	`data FizzBuzz a`
	`  = GetFizz Int Int (String -> a)`
	`  | GetBuzz Int Int (String -> a)`
	`  | GetNum  Int Int (String -> a)`

Make it an instance of `Functor` by implementing the `fmap` function. Every Functor type should be parameterized to carry any other type inside.	`instance Functor FizzBuzz where`
	`  fmap f (GetFizz n m next)`
	`  = GetFizz n m (fmap f next)`
	`  fmap f (GetBuzz n m next)`
	`  = GetBuzz n m (fmap f next)`
	`  fmap f (GetNum n m next)`
	`  = GetNum  n m (fmap f next)`

Create a Free monad type based on your Functor.	`type FizzBuzzer a = Free FizzBuzz a`

Create smart constructors that wrap (lift) your ADT into your Free monad type. You may either use the `liftF` function or define it by hand:	`-- Automatically wrapped (lifted):`
	`getFizz, getBuzz, getNum`
	`  :: Int -> Int -> FizzBuzzer String`
	`getFizz n m = liftF (GetFizz n m id)`
	`getBuzz n m = liftF (GetBuzz n m id)`
	`getNum z n  = liftF (GetNum  z n id)`
	
	`-- Manually wrapped:`
	`getFizz', getBuzz', getNum'`
	`  :: Int -> Int -> FizzBuzzer String`
	`getFizz' n m = Free (GetFizz n m Pure)`
	`getBuzz' n m = Free (GetBuzz n m Pure)`
	`getNum' z n  = Free (GetNum  z n Pure)`

Your language is ready. Create interpreters and scripts on your taste.	`getFizzBuzz :: Int -> FizzBuzzer String`
	`getFizzBuzz n = do`
	`  fizz <- getFizz n 5`
	`  buzz <- getBuzz n 3`
	`  let fb = fizz ++ buzz`
	`  s <- getNum (length fb) n`
	`  return $ s ++ fizz ++ buzz`

First, we'll generalize working with remote devices. In reality, all things we do with sensors and devices we do by operating the intellectual controller that is embedded into any manageable device. So reading measurements from a sensor is equivalent to asking a controller to read measurements from that particular sensor, because one device can have

many sensors. In turn, measurements vary for different kinds of sensors. Also, the controller has an internal state with many properties that depend on the type of the controller, for example, its local time, connectivity status, errors The scripting language should allow us to get and set these properties (in a limited way, possibly). Finally, the device may be intended to do some operations: open and close valves, turn lights on and off, start and stop something, and more. To operate the device, we send a command to the controller. Knowing that, we are able to redesign our `Procedure` data type as shown in the following listing.

Listing 4.3 Improved Procedure eDSL for working with remote devices

```
-- These types are defined in a separate library
data Value = FloatValue Float
           | IntValue Int
           | StringValue String
           | BoolValue Bool
           | ListValue [Value]
data Measurement = Measurement Value
data Parameter = Temperature | Pressure

-- Dummy types, should be designed later
data Property = Version | Status | SensorsList
data Controller = Controller String
data Command = Command String
type CommandResult = Either String String
type SensorIndex = String

-- Parametrized type for a Free eDSL
data Procedure a
    = Set Controller Property Value a      #A
    | Get Controller Property (Value -> a)           #B
    | Run Controller Command (CommandResult -> a)    #B
    | Read Controller SensorIndex Parameter (Measurement -> a)
```

#A "Non-returning" definition
#B "Returning" definitions

> **NOTE** The `Measurement` type knows nothing about measurement units. This is a problem. What if you requested a `Temperature` parameter but accidentally got pressure units? How would your system behave then? In the Andromeda project, this type is improved by a phantom type tag: (`Measurement tag`), so you really should use it with units like so: (`Measurement Kelvin`). The `Parameter` type also has this tag: (`Parameter tag`). These two types while used at once require units to be consistent, that means the values should have types with identical tags. For more information you may look into the Andromeda project or read the chapter about type-level logic.

This new language is composed of instructions to work with remote devices through a controller. This type is also parameterized by a type variable because it will be a `Free` monad language, and we need to make it a `Functor`. To illustrate this better, we'll need to complete the rest of the "free monadizing" algorithm: namely, make this type an instance of `Functor` and provide convenient smart constructors. Listing 4.4 shows the instance:

Listing 4.4: The instance of Functor for the Procedure type

```
instance Functor Procedure where
    fmap g (Set  c p v next) = Set  c p v  (g next)
    fmap g (Get  c p nextF)      = Get  c p    (g . nextF)
    fmap g (Read c si p nextF)   = Read c si p (g . nextF)
    fmap g (Run  c cmd nextF)    = Run  c cmd  (g . nextF)
```

Let's figure out how this works and why there are two sorts of application of the `g` function passed to `fmap`:

```
(g next)
(g . nextF)
```

From the previous chapter we know that a type is a `Functor` if we can apply some function `g` to it's contents without changing the whole structure. The `fmap` function will do the application of `g` for us, so to make a type to be a `Functor`, we should define how the `fmap` function behaves. The `Free` monad uses the `fmap` function to nest actions in continuation fields we provide in our domain algebra. This is the main way to combine monadic operations (actions) in the `Free` monad. So every value constructor of our algebra should have a continuation field.

We have four value constructors encoding four domain operations (actions) in the `Procedure` type. The `Set` value constructor is rather simple:

```
data Procedure a
    = Set Controller Property Value a
```

It has four fields of type `Controller`, `Property`, `Value` and a continuation field with a generic type `a`. This field should be mappable in the sense of `Functor`. This means, the `fmap` function should apply a generic `g` function to this continuation field:

```
fmap g (Set c p v next) = Set c p v (g next)
```

We call this field `next` because it should be interpreted next to the `Set` procedure.
The last fields every value constructor has denoted the continuation. In other words, this is the action that should be evaluated next. Also, the action encoded by the `Set` value constructor, returns nothing useful. However the actions encoded by the `Get`, `Read`, and `Run`, value constructors, do return something useful, namely the `Value`, `Measurement`, and `CommandResult` values, respectively. That is why the continuation fields differ. It's now not just of type `a` but of function type (`someReturnType -> a`):

```
data Procedure a
    = Get Controller Property (Value -> a)
    | Run Controller Command (CommandResult -> a)
    | Read Controller SensorIndex Parameter (Measurement -> a)
```

Such continuation fields hold actions that know what to do with the value returned. When the `Free` monad combines two actions it ensures the value the first action returns is what the second action is awaiting as the argument. For example, when the `Get` value constructor is interpreted, it will return a value of type `Value`. The nested action should be of type (`Value -> something`) to be combined.

The `fmap` function counts that. It receives the `g` function and applies to the mappable contents of this concrete value constructor:

```
fmap g (Get c p nextF) = Get c p (g . nextF)
```

The application of function `g` to a regular value `next` is just (`g next`) as it's shown above. The application of function `g` to a function `nextF` is composition of them: (`g . nextF`). We map function `g` over the single field and leave all other fields unchanged.

The trick of nesting of continuations is exactly the same one we used in the previous version of the `Procedure` type but now we are dealing with a better abstraction - the `Free` monad pattern. Strictly speaking, the `Free` monad pattern is able to handle returning values by keeping a continuation in the field with a function type, and a continuation is nothing more than a function in the same monad that accepts a value of the input type and processes it.

The next step of the "free monadizing" algorithm is presented in listing 4.5. We define a synonym for the `Free` type and declare smart constructors:

Listing 4.5: Type for monadic eDSL and smart constructors

```
type ControllerScript a = Free Procedure a

-- Smart constructors:
set :: Controller -> Property -> Value -> ControllerScript ()
set c p v = Free (Set c p v (Pure ()))

get :: Controller -> Property -> ControllerScript Value
get c p = Free (Get c p Pure)

read :: Controller -> SensorIndex -> Parameter
     -> ControllerScript Measurement
read c si p = Free (Read c si p Pure)

run :: Controller -> Command -> ControllerScript CommandResult
run c cmd = Free (Run c cmd Pure)
```

These smart constructors wrap procedures into the monadic `ControllerScript a` type (the same as `Free Procedure a`). To be precise, they construct a monadic value in the `Free` monad parametrized by the `Procedure` functor. We can't directly compose value constructors `Get`, `Set`, `Read and Run` in the monadic scripts. The `Procedure a` type is not a monad, just a functor. But the `set` function and others make a composable combinator in the `ControllerScript` monad instead. This all may be looking monstrously, but it's actually not that hard, just meditate over the code and try to transform types one into other starting from the definition of the `Free` type (we discussed it in the previous chapter):

```
data Free f a = Pure a
              | Free (f (Free f a))
```

You'll discover the types `Free` and `Procedure` are now mutually nested in a smart recursive way.

Notice the `Pure` value constructor in the smart constructors. It denotes the end of the monadic chain. You can put `Pure ()` into the `Set` value constructor, but you can't put it into the `Get`, `Read` and `Run` value constructors. Why? You may infer the type of the `Get`'s continuation field as we did it in the previous chapter. It will be (`Value -> ControllerScript a`) while `Pure ()` has type `ControllerScript a`. You just need a

function instead of regular value to place it into a continuation field of this sort. The partially applied value `Pure :: a -> Free f a` is what you need. Compare this carefully:

```
set c p v = Free (Set c p v (Pure ()))
get c p = Free (Get c p Pure)

fmap g (Set c p v next) = Set  c p v (g next)
fmap g (Get c p nextF) = Get c p (g . nextF)
```

Whenever you write `Pure`, you may write `return` instead, they do the same thing.

```
set c p v = Free (Set c p v (return ()))
get c p = Free (Get c p return)
```

In the monad definition for the `Free` type, the `return` function is defined to be a partially applied `Pure` value constructor:

```
instance Functor f => Monad (Free f) where
    return = Pure

-- Monadic composition
    bind (Pure a) f = f a
    bind (Free m) f = Free ((`bind` f) <$> m)
```

Don't care about the definition of the monadic bind. We don't need it in this book.

The sample script is presented in listing 4.6. Notice it's composed from the `get` action and the `process` action. The `process` function works in the same monad `ControllerScript`, so it may be composed with other functions of the same type monadically.

Listing 4.6: Sample script in the eDSL

```
controller = Controller "test"
sensor = "thermometer 00:01"
version = Version
temperature = Temperature

-- Subroutine:
process :: Value -> ControllerScript String
process (StringValue "1.0") = do
    temp <- read controller sensor temperature
    return (show temp)
process (StringValue v) = return ("Not supported: " ++ v)
process _ = error "Value type mismatch."

-- Sample script:
script :: ControllerScript String
script = do
    v <- get controller version
    process v
```

Let's now develop an sample interpreter. After that, we'll consider how to hide the details of the language from the client code. Whether the value constructors of the `Procedure` type should be public? It seems, this isn't a must While they are public, the user can interpret a language by pattern matching. The listing 4.7 shows how we roll out the structure of the `Free` type recursively and interpret the procedures nested one inside another. The deeper a procedure lies, the later it will be processed, so the invariant of sequencing of the monadic actions is preserving.

Listing 4.7 Possible interpreter in the IO monad

```
{- Impure interpreter in the IO monad that prints every instruction
   with parameters. It also returns some dummy values
   for Get, Read, and Run instructions. -}

interpret :: ControllerScript a -> IO a
interpret (Pure a) = return a
interpret (Free (Set c p v next)) = do      #A
    print ("Set", c, v, p)
    interpret next                          #B
interpret (Free (Get c p nextF)) = do       #C
    print ("Get", c, p)
    interpret (nextF (StringValue "1.0"))   #D
interpret (Free (Read c si p nextF)) = do
    print ("Read", c, si, p)
    interpret (nextF (toKelvin 1.1))
interpret (Free (Run c cmd nextF)) = do
    print ("Run", c, cmd)
    interpret (nextF (Right "OK."))
```

#A next keeps the action to be interpreted next.
#B Continue interpreting
#C nextF keeps the function that is awaiting a value as argument
#D Continue interpreting after the nextF action is received the value

It's not a bad thing in this case, but does the interpreter provider want to know about the `Free` type and how to decompose it with pattern matching? Do they want to do recursive calls? Can we facilitate their life here? Yes, we can. This is the theme of the next section.

4.3.3 The abstract interpreter pattern

To conceal the `Free` monad nature of our language, to hide explicit recursion, and to make interpreters clean and robust, we need to abstract the whole interpretation process behind an interface — but this interface shouldn't restrict you in writing interpreters. It's more likely you'll wrap the interpreters into some monad. For instance, you can store operational data in the local state (the `State` monad) or immediately print values to the console during the process (the `IO` monad). Consequently, our interface should have the same expressiveness. The pattern we will adopt here has the name "the abstract interpreter."

The pattern has two parts: the functional interface to the abstract interpreter you should implement, and the base `interpret` function that calls the methods of the interface while a Free type is recursively decomposed. Let's start with the former. It will be a specific type class, see Listing 4.8.

Listing 4.8 Interface to abstract interpreter

```
class Monad m => Interpreter m where
```

```
    onSet  :: Controller -> Property -> Value -> m ()
    onGet  :: Controller -> Property -> m Value
    onRead :: Controller -> SensorIndex -> Parameter -> m Measurement
    onRun  :: Controller -> Command -> m CommandResult
```

The constraint `Monad` for type variable `m` you can see in the class definition says that every method of the type class should operate in some monad `m`. This obligates the instance of the interface to be monadic; we allow the client code to engage the power of monads, but we don't dictate any concrete monad. What monad to choose is up to you, depending on your current tasks. The type class doesn't have any references to our language; no any value constructor of the `Procedure` type is present there. How will it work, then? Patience, we need one more part: the template interpreting function. It's very similar to the interpreting function in listing 4.7, except it calls the methods the type class `Interpreter` yields. The following listing demonstrates this code.

Listing 4.9: Abstract interpreting function for the free eDSL

```
module Andromeda.LogicControl.Language (
    interpret,
    Interpreter(..),
    ControllerScript,
    get,
    set,
    read,
    run
  ) where

{- here the content of listings 4.3, 4.4, 4.5 goes -}

-- The base interpret function
interpret :: (Monad m, Interpreter m) => ControllerScript a -> m a
interpret (Pure a) = return a
interpret (Free (Get c p next)) = do
    v <- onGet c p
    interpret (next v)
interpret (Free (Set c p v next)) = do
    onSet c p v
    interpret next
interpret (Free (Read c si p next)) = do
    v <- onRead c si p
    interpret (next v)
interpret (Free (Run c cmd next)) = do
    v <- onRun c cmd
    interpret (next v)
```

We hide the `Procedure` type, but we export the function `interpret`, the type class `Interpreter`, the type `ControllerScript`, and the smart constructors. What does this mean? Imagine you take this someone's weird library. You want to construct a script and interpret it too. The first task is easily achievable. Let's say you have written the script like in listing 4.6. Now you try to write an interpreter like in listing 4.7, but you can't because no value constructors are available outside the library. But you notice there is the `interpret` function that requires the type class `Interpreter` to be instantiated. This is the only way to

interpret your `Free` script into something real. You should have a parameterized type to make this type an instance of the `Interpreter` type class. The type should be an instance of a monad, also. Suppose, you are building an exact copy of the interpreter in listing 4.7. You should adopt the `IO` monad then. The code you'll probably write may look so:

```
import Andromeda.LogicControl.Language

instance Interpreter IO where
    onSet c prop v = print ("Set", c, v, prop)
    onGet c prop = do
        print ("Get", c, prop)
        return (StringValue "1.0")
    onRead c si par = do
        print ("Read", c, si, par)
        return (toKelvin 1.1)
    onRun c cmd = do
        print ("Run", c, cmd)
        return (Right "OK.")
```

After this, you interpret the script in listing 4.6:

```
interpret script

-- The result
-- ("Get",Controller "test",Version)
-- ("Read",Controller "test","thermometer 00:01",Temperature)
-- "Measurement (FloatValue 1.1)
```

Hiding the implementation details will force the developer to implement the type class. This is a functional interface to our subsystem. The functional interface to the language itself is now accompanied by the functional interface to the interpreter.[2]

Other interpreters could be implemented inside other monads; for example, `State` or `State + IO`. A consequence of this design is that it keeps our interpreters consistent automatically, because when the language gets another procedure the `Interpreter` type class will be updated, and we'll get a compilation error until we update our instances. So we can't forget to implement a new method, in contrast to the previous design, where the Free language was visible for prying eyes.

It's perfect, wouldn't you agree?

4.3.4 Free language of Free languages

Scripts we can write with the `ControllerScript` language cover only a small part of the domain, while the requirements say we need to operate with a database, raise alarms if needed, run calculations, and do other things to control the ship. These "subdomains" should be somewhat independent from each other because we don't want a mess like we saw in the "naive" language. Later we will probably add some capabilities for user input/output communication — this will be another declarative language that we can embed into our focused domain languages without too much trouble. There is a chance that you may find the approach I suggest in this section too heavy, but it gives us some freedom to implement domain logic partially, prove it correct with concise tests, and move on. When the skeleton of

[2] The idea of this pattern is very close to the idea of the object-oriented Visitor pattern. But the functional pattern is better because it can restrict all impure and mutable operations by prohibiting the `IO` monad.

the application is designed well, we can return and complete the logic, providing the missing functionality. That is, such design allows us to stay on the top level, which is good in the early stages of development.

Let's now discuss this design approach — a pointer of pointers to arrays of pointers to foo! Oh, sorry, wrong book... I meant a `Free` language of `Free` languages!

The idea is to have several small languages to cover separate parts of the domain. Each language is intended to communicate with some subsystem, but not directly, because every language here is a `Free` eDSL. Remember, we make this "letter of intent" acting by the interpretation process. We can even say the compilation stage is our functional analogue of "late binding" from OOP. Our eDSLs are highly declarative, easily constructible, and interpretable. That's why we have another big advantage: the ease of translation of our scripts to an external language and back again (you'll see this in the corresponding section of this chapter). This would be very difficult to do otherwise.

Check out the elements diagram for Logic Control (figure 2.13) and the architecture diagram (figure 2.15): this approach was born there, but it was a little cryptic and had inaccurate naming. We'll adopt these names for the languages:

- `ControllerScript` — The `Free` eDSL to communicate with the Hardware subsystem
- `InfrastructureScript` — The `Free` eDSL for logging, authorization, filesystem access, operating system calls, raising events, and so on (or maybe we should partition these responsibilities?)
- `ComputationScript` — The eDSL for mathematical computations (not a `Free` language, possibly)
- `DataAccessScript` — The `Free` eDSL for operating with the database
- `ControlProgram` — The `Free` eDSL that allows us to run any of the scripts and also provides reactive capabilities (we'll return to this in the next chapters)

Figure 4.1 illustrates the whole design.

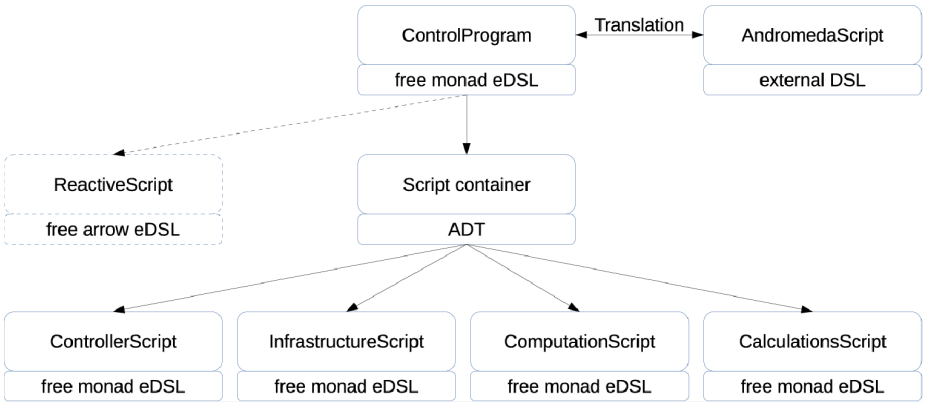


Figure 4.1 Design of domain model.

Pictures are good; code is better. Let's get familiar with the `InfrastructureScript` DSL shortly, see listing 4.10.

Listing 4.10: The InfrastructureScritp free DSL.

```
{-# LANGUAGE DeriveFunctor #-}     #1
module Andromeda.LogicControl.Language.Infrastructure

-- Dummy types, should be designed later.
type ValueSource = String
type Receiver = Value -> IO ()

data Action a = StoreReading Reading a
              | SendTo Receiver Value a
              | GetCurrentTime (Time -> a)
    deriving (Functor)                          #2

type InfrastructureScript a = Free Action a

storeReading :: Reading -> InfrastructureScript ()
sendTo :: Receiver -> Value -> InfrastructureScript ()
logMsg :: String -> InfrastructureScript ()
alarm :: String -> InfrastructureScript ()
getCurrentTime :: InfrastructureScript Time
```

#1 Useful Haskell language extension
#2 The automatic instantiation of a Functor type class in Haskell.

Notice we went by a short path exist in Haskell: we automatically derived a `Functor` instance for the `Action` type (#2). No more annoying `fmap` definitions! We are too lazy to do all this boilerplate by our hands. This only works with the `DeriveFunctor` extension enabled (#1).

Listing 4.11 displays the `Script` algebraic data type that ties many languages together.

Listing 4.11 The Script container

```
-- Script container, wraps all Free languages.
data Script b = ControllerScriptDef (ControllerScript b)
              | ComputationScriptDef (ComputationScript b)
              | InfrastructureScriptDef (InfrastructureScript b)
              | DataAccessScriptDef (DataAccessScript b)
```

The `b` type denotes something a script returns. There are no any restrictions to what `b` should be. We may want to return value of any type. The following two scripts return different types:

```
startBoosters :: ControllerScript CommandResult
startBoosters = run (Controller "boosters") (Command "start")

startBoostersScript :: Script CommandResult
startBoostersScript = ControllerScriptDef startBoosters

getTomorrowTime :: InfrastructureScript Time
getTomorrowTime = do
    time <- getCurrentTime
    return (time + 60*60*24)
```

```
getTomorrowTimeScript :: Script Time
getTomorrowTimeScript = InfrastructureScriptDef getTomorrowTime
```

The problem here is that the type of the two "wrapped" functions `startBoostersScript` and `getTomorrowTimeScript` don't match. `Script Time` is not equal to `Script CommandResult`. This means, we have two different containers; what can we do with them? Suppose we want to unify all these scripts in one top-level composable free eDSL as it was intended in figure 4.1. Consider the first try:

```
-- Top-level Free language.
data Control b a = EvalScript(Script b) a
```

This is the algebra that should store a script container. The container's type is parametrized, so we provide a type argument for `Script b` in the type constructor `Control b`. We know this type will be a functor. There should be a field of another type `a` the `fmap` function will be mapped over. Therefore we add this type parameter to the type constructor: `Control b a`. However we forgot a value of the `b` type a script will return. According to the `Free` monad pattern, the value must be passed to nested actions. The continuation field should be of function type:

```
data Control b a = EvalScript(Script b) (b -> a)
```

So far, so good. Let's define a `Functor` with the `b` type variable frozen, because it should be a Functor of the single type variable `a`:

```
instance Functor (Control b) where
    fmap f (EvalScript scr g) = EvalScript scr (f . g)
```

And finally a `Free` type with a smart constructor:

```
type ControlProgram b a = Free (Control b) a

evalScript :: Script b -> ControlProgram b a
evalScript scr = Free (EvalScript scr Pure)
```

It's pretty good except it won't compile. Why? Is that fair? We did all what we usually do. We did it right. But the two type variables `b` the `evalScript` has the compiler can't match. It can't be sure they are equal for some weird reason. However if you'll try the following definition, it will compile:

```
evalScript :: Script a -> ControlProgram a a
evalScript scr = Free (EvalScript scr Pure)
```

But it's all wrong because the type variable of the Script can't be specialized more than once. Consider the following script that should be of "quantum" type:

```
unifiedScript :: ControlProgram ??? (CommandResult, Time)
unifiedScript = do
    time <- evalScript getTomorrowTimeScript
    result <- evalScript startBoostersScript
    return (result, time)
```

Why quantum? Because two scripts in this monad have return types `CommandResult` and `Time` but how to say it in the type definition instead of three question marks? Definitely, the `b` type variable takes two possible types in quantum superposition. I believe you may do so in some imaginary Universe, but here, quantum types are prohibited. The type variable `b` must take either `CommandResult` or `Time` type. But this completely ruins the idea of a `Free` language over `Free` languages. In dynamically typed languages this situation is gently avoided. Dynamic languages do have quantum types! Does that mean statically typed languages are defective?

Fortunately, no, it doesn't. We just need to summon the type-level tricks and explain to the compiler what we actually want. The right decision here is to hide the `b` type variable behind the scenes. Look at the `unifiedScript` and the `ControlProgram` type again: do you want to carry `b` everywhere? I don't think so. This type variable denotes the return type from script. When you call a script, you get a value. Then you pass that value to the continuation. Consequently the only place this type exists is localized between the script itself and the continuation. The following code describes this situation:

```
{-# LANGUAGE ExistentialQuantification #-}
data Control a = forall b. EvalScript (Script b) (b -> a)
```

As you can see, the `b` type variable isn't presented in the `Control` type constructor. No one who uses this type will ever know there is an internal type `b`. To declare that it's internal, we write the `forall` quantificator. Doing so we defined the scope for the `b` type. It's bounded by the `EvalScript` value constructor (because the `forall` keyword stays right before it). We may use the `b` type variable inside the value constructor, but it's completely invisible from the outside. All it does inside is showing the `b` type from a script is the same type the continuation is awaiting. It doesn't matter what the `b` type actually is. Anything. All you want. It says to the compiler: just put a script and an action of the same type into the `EvalScript` value constructor and don't accept two artifacts of different types. What to do with the value the script returns, the action will decide by itself. One more note: this all is possible due to the Existential Quantification extension of the Haskell language.

The complete design of the `ControlProgram` free language is shown in listing 4.12:

Listing 4.12: The ControlProgram free eDSL.

```
-- Script container, wraps all Free languages
data Script b = ControllerScript (ControllerScript b)
              | ComputationScript (ComputationScript b)
              | InfrastructureScript (InfrastructureScript b)
              | DataAccessScript (DataAccessScript b)

-- Smart constructors:
infrastructureScript :: InfrastructureScript b -> Script b
infrastructureScript = InfrastructureScriptDef

controllerScript :: ControllerScript b -> Script b
controllerScript = ControllerScriptDef

computationScript :: ComputationScript b -> Script b
computationScript = ComputationScriptDef

dataAccessScript :: DataAccessScript b -> Script b
```

```
dataAccessScript = DataAccessScriptDef

-- Top-level eDSL. It should be a Functor
data Control a = forall b. EvalScript (Script b) (b -> a)

instance Functor Control where
    fmap f (EvalScript scr g) = EvalScript scr (f . g)

-- Top-level Free language.
type ControlProgram a = Free Control a

-- Smart constructor
evalScript :: Script a -> ControlProgram a
evalScript scr = Free (EvalScript scr Pure)

-- sample script
unifiedScript :: ControlProgram (CommandResult, Time)
unifiedScript = do
    time <- evalScript getTomorrowTimeScript
    result <- evalScript startBoostersScript
    return (result, time)
```

Notice that the smart constructors are added for the `Script` type (`infrastructureScript` and others). Smart constructors make our life much easier.

As usual, in the final stage of the Free language development, you create an abstract interpreter for the `ControlProgram` language. Conceptually, the abstract interpreter has the same structure: the `Interpreter` type class and the base function `interpret`.

```
class Monad m => Interpreter m where
    onEvalScript :: Script b -> m b

interpret :: (Monad m, Interpreter m) => ControlProgram a -> m a
interpret (Pure a) = return a
interpret (Free (EvalScript s nextF)) = do
    v <- onEvalScript s
    interpret (nextF v)
```

When someone implements the `Interpreter` class type, he should call `interpret` functions for nested languages. The implementation may look so:

```
module InterpreterInstance where

import qualified ControllerDSL as C
import qualified InfrastructureDSL as I
import qualified DataAccessDSL as DA
import qualified ComputationDSL as Comp

interpretScript (ControllerScriptDef scr) = C.interpret scr
interpretScript (InfrastructureScriptDef scr) = I.interpret scr
interpretScript (ComputationScriptDef scr) = DA.interpret scr
interpretScript (DataAccessScriptDef scr) = Comp.interpret scr
instance Interpreter IO where
```

```
onEvalScript scr = interpretScript scr
```

The `Control` type now has only one field that is a declaration to evaluate one of the scripts available, but in the future we can extend it to support, for example, declaration of a reactive model for FRP. It's an interesting possibility, however not that simple. Stay in touch, we go further!

4.3.5 Arrows for eDSLs

Are you tired of learning about complex concepts? Take a break and grab some coffee. Now let's look more closely at the concept of arrows.

The arrow is just a generalization of the monad, which is just a monoid in the category of... oh, forget it. If you have never met functional arrows before, I'll try to give you a little background on them, but this will be a light touch, because our goal differs: we would do better to form an intuition of when and why the arrowized language is an appropriate domain representation than to learn how to grok arrows. For more information, consider consulting some external resources; there are many of them for Haskell and Scala. You should be motivated to choose an arrowized language when you have:

- Computations that are like electrical circuits: there are many transforming functions ("radio elements") connected by logical links ("wires") into one big graph ("circuit")
- Time-continuing and time-varying processes, transformations, and calculations that depend on the results of one another
- Computations that should be run by time condition: periodically, once at the given time, many times during the given period, and so forth
- Parallel and distributed computations
- Computation flow (data flow) with some context or effect. The need for a combinatorial language in addition to or instead of a monadic language

It's not by chance that, being a concept that abstracts a flow of computations, an arrowized script is representable as a flow diagram. Like monads, the arrowized computations can depend on context that is hidden in the background of an arrow's mechanism and thus completely invisible to the developer. It's easy to convert a monadic function `f :: a -> m b` into the arrow `arr1 :: MyArrow a b`, preserving all the goodness of a monad `m` during `arr1` evaluation. This is how the concept of arrows generalizes the concept of monads. And even easier to create an arrow `arr2 :: MyArrow b c` from just a non-monadic function `g :: b -> c`. This is how the concept of arrows generalizes the function type.

Finally, when you have two arrows, it's not a big deal to chain them together:

```
arr1 :: MyArrow a b
arr1 = makeMonadicArrow f

arr2 :: MyArrow b c
arr2 = makePureArrow g

arr3 :: MyArrow a c
arr3 = arr1 >>> arr2
```

All we should know to compose arrows is their type: the first arrow converts values of type `a` to values of type `b`, and the second arrow converts values of type `b` to values of type `c`. That is, these two arrows both convert values by the scheme `(a -> b) -> c`. Applied to a value of type `a`, the arrow `arr3` will first do `f a` with monadic effect resulting in a value of type `m b`, and then will evaluate `g b` resulting in value of type `c` (or `m c`. - it depends on the

concrete monad you use inside the arrow). In short, if `runArrow` is application of your arrow to an argument, then `runArrow arr3 a` may be equivalent to this:

```
apply :: (a -> m b) -> (b -> c) -> m c    -- not escaped from monad
apply f g a = do
    b <- f a
    let c = g b
    return c
```

or to this:

```
apply :: (a -> m b) -> (b -> c) -> c    -- escaped from monad
apply f g a =
    let b = runMonad f a
        c =g b
    in c
```

That's how the `(>>>)` combinator works: it applies the left arrow and then the right one. And it's aware of arrow's internals so it may run monad for monadically composed arrow. This operation is associative:

```
arr1 >>> arr2 >>> arr3
```

To apply an arrow to a value you call a "run" function from a library:

```
toStringA :: MyArrow Int String
toStringA = arr show

evaluateScenario = do
result <- runArrow toStringA 10
print result
```

The `arr` function should be defined for every arrow because it present in the `Arrow` type class (ignore `Category` type class for now):

```
class Category a => Arrow a where
    arr :: (b -> c) -> a b c
    first :: a b c -> a (b,d) (c,d)
```

You might have noticed that when looking at arrow types, you often can't conclude whether there is a monadic effect or not. For example: what monad is hided under the imaginary arrow `WillItHangArrow Program Bool`? Is there a kind of `IO` monad? Or maybe the `State` monad is embedded there? You'll never know unless you'll open its source code. Is that bad or good? Hard to say. We went to the next level of abstraction, and we can even cipher different effects in one computation flow by switching between different arrows. But the purity rule works anyway: if a particular arrow is made with IO inside, you'll be forced to run that arrow in the `IO` monad.

```
ioActionArrow :: MyIOArrow () ()
ioActionArrow = makeMonadicArrow (\_ -> putStrLn "Hello, World!")

-- Fine:
```

```
main :: IO ()
main = runMyIOArrow ioActionArrow ()

-- Won't compile:
pureFunction :: Int -> Int
pureFunction n = runMyIOArrow  ioActionArrow ()
```

The arrows composed only with the sequential combinator `(>>>)` look quite boring in the flow diagram (see figure 4.2).

Figure 4.2 Sequential flow diagram.

Certainly, we aren't limited to sequential composition only. As usual, if we realize our domain model can fit into an arrowized language, we can take advantage of all the combinators the arrow library provides. There is a wide range of arrow combinators we may use to make our computational networks much more interesting: parallel execution of arrows, splitting the flow into several flows and merging several flows into one, looping the computation, and conditional evaluation are all supported.

We'll construct an arrowized interface over the `Free` languages in the Logic Control subsystem so you can add the flow graph to your toolbox for domain modeling. But before we do that, consider the mnemonic arrow notation. The arrow `arrowA` that accepts the `input` value and returns the `output` value is written as:

```
output <- arrowA -< input
```

Because every arrow is a generalization of a function, it should have input and output, but we can always pass the unit value `()` if the arrow doesn't actually need this:

```
-- no input, no output:
() <- setA ("PATH", "/home/user") -< ()
```

If two arrows depend on the same input, they can be run in parallel. Will it be a real parallelization or just logical possibility depends on the arrow's mechanism. You may construct an arrow type that will run these two expressions concurrently:

```
factorial <- factorialA -< n
fibonacci <- fibonacciA -< n
```

Arrows can take and return compound results. The most important structure for arrows is a pair. It is used in the arrow machinery to split a pair and feed two arrows by own parts of a pair (see the split `(***)` operator below). You may write an arrow that will change only the first or the second item of a pair. For example, the following two arrows take either `n` or `m` to calculate a factorial but leave another value unchanged:

```
(factorialN, m) <- first  factorialA -< (n, m)
```

```
(n, factorialM) <- second factorialA -< (n, m)
```

The combinators `first` and `second` should be defined for every arrow, as well as the `(>>>)` combinator and others. The fanout `(&&&)` combinator makes an arrow from two of them, running them in parallel with the input argument cloned. The output will be a pair of results from the first and second arrows:

```
(factorial, fibonacci) <- factorialA &&& fibonacciA -< n
```

The split `(***)` combinator behaves like the `(&&&)` combinator, but takes a pair of inputs for each of two arrows it combines:

```
(factorialN, fibonacciM) <- factorialA *** fibonacciA -< (n, m)
```

Figure 4.3 illustrates these combinators as input/output boxes.

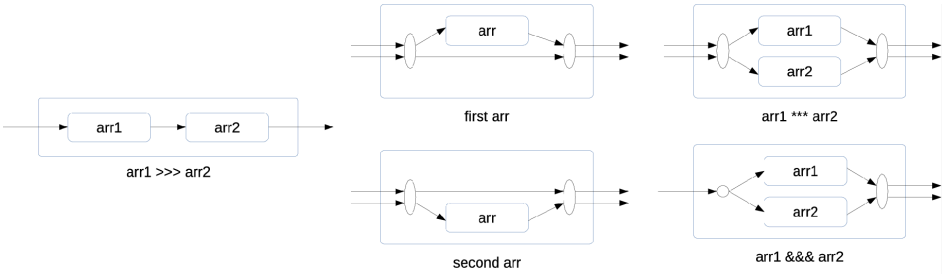


Figure 4.3 Arrow combinators.

> **TIP** Some self-descriptiveness can be achieved with a "conveyor belt diagram," where arrows associate with machines and the belt supplies them with values to be processed. The tutorial Haskell: /Understanding arrows"[3] uses this metaphor (see visualization) and gives a broad introduction into arrows.

4.3.6 Arrowized eDSL over Free eDSLs

Let's take the last mnemonic monitoring scenario and reformulate it in the arrowized way. Meet the arrow that monitors readings from the thermometer:

```
Scenario: monitor outside thermometer temperature
Given: outside thermometer @therm
Evaluation: once a second, run arrow thermMonitorA(@therm)

arrow thermMonitorA [In: @therm, Out: (@time, @therm, @tempK)]
    @tempC <- thermTemperatureA -< @therm
    @tempK <- toKelvinA         -< @tempC
    @time  <- getTimeA          -< ()
    ()     <- processReadingA   -< (@time, @therm, @tempK)
    return (@time, @therm, @tempK)
```

[3] https://en.wikibooks.org/wiki/Haskell/Understanding_arrows

It calls other arrows to make transformations and to call scripts from the `Free` domain languages. The `thermTemperatureA` arrow reads the temperature:

```
arrow thermTemperatureA [In: @therm, Out: @tempC]
    @tempC <- runScriptA -< thermTemperatureS(@therm)
    return @tempC
```

Arrows that store readings, validate temperatures, and raise alarms when problems are detected are combined in the `processReadingA` arrow:

```
arrow processReadingA [In: (@time, @therm, @tempK),  Out: ()]
    ()       <- storeReadingA    -< @reading
    @result <- validateReadingA -< @reading
    ()       <- alarmOnFailA     -< @result
    return ()
```

We could define other arrows, but I think it's now obvious how they describe scenarios mnemonically. The full computation is better shown by a flow diagram (see figure 4.4).

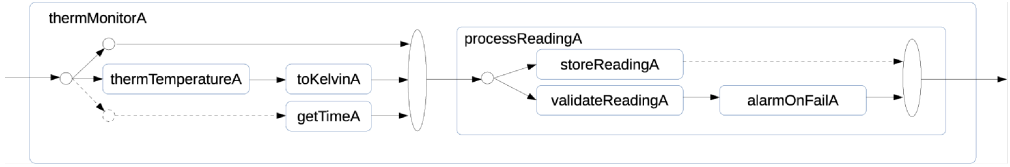


Figure 4.4 Flow diagram for thermometer monitoring arrow.

If the mnemonic arrow notation and the computational graph have scared you a little, you haven't seen the combinatorial code yet! There are several ways to compose an arrow for the diagram, and one of them - to make the calculation process completely sequential. In the following code, many arrows are combined together to transform results from each other sequentially:

```
thermMonitorA = (arr $ \b -> (b, b))
                >>> second (thermTemperatureA >>> toKelvinA)
                >>> (arr $ \x -> ((), x))
                >>> first getTimeA
                >>> (arr $ \(t, (inst, m)) -> (t, inst, m))
                >>> (arr $ \b -> (b, b))
                >>> second (storeReadingA &&& validateReadingA)
                >>> second (second alarmOnFailA)
                >>> (arr $ \(b, _) -> b)
```

What is the code here? It looks very cryptic, like Perl, or a paragraph from a math paper. This is actually valid Haskell code that you may freely skip. It's here for those who want a full set of examples, but you may be impressed that Haskell has a nicer `proc` notation for arrows that is very close to the mnemonic notation we have introduced. Before it is introduced, let's prepare the arrow type. Suppose we have constructed the `FlowArr b c` arrow type somehow that is able to describe the diagram in figure 4.4. This arrow is specially designed

to wrap our free languages and scenarios. It doesn't have any own logic, it only provides an arrowized interface to the languages. You may or you may not use it depending on your taste.

As the mnemonic scenario says, the `thermMonitorA` arrow takes an instance of thermometer (let it be of type `SensorInstance`) and returns a single reading of type `Reading` from it:

```
thermMonitorA :: FlowArr SensorInstance Reading
thermMonitorA = proc sensorInst -> do
    tempK <- toKelvinA <<< thermTemperatureA -< sensorInst
    time  <- getTimeA -< ()

    let reading = (time, sensorInst, tempK)

    ()      <- storeReadingA    -< reading
    result  <- validateReadingA -< reading
    ()      <- alarmOnFailA     -< result
    returnA -< reading
```

The `proc` keyword opens a special syntax for arrow definition. The variable `sensorInst` is the input argument. The arrowized `do` block, which is extended compared to the monadic `do` block, defines the arrow's body. At the end, the `returnA` function should be called to pass the result out.

> **TIP** To enable the `proc` notation in a Haskell module, you should set the compiler pragma `Arrows` at the top of the source file: `{-# LANGUAGE Arrows #-}`. It's disabled by default due to nonstandard syntax.

Here's the definition of the `FlowArr` arrow type:

```
type FlowArr b c = ArrEffFree Control b c
```

This type denotes an arrow that receives `b` and returns `c`. The `ArrEffFree` type, which we specialize by our top-level eDSL type `Control`, came from the special library I designed for the demonstration of the Free Arrow concept. This library has a kind of stream transformer arrow wrapping the `Free` monad. Sounds menacing to our calm, but we won't discuss the details here. If you are interested, the little intro in Appendix A is for you. All you need from that library now is the `runFreeArr`. Remember we were speaking about whether you should interpret a Free language if you want to run it in a real environment? This is the same for the arrowized interface over a Free language. To run arrow, you pass exactly the same interpreter to it:

```
sensorInst = (Controller "boosters", "00:01")
test = runFreeArr interpret thermMonitorA sensorInst
```

Here, `interpretControlProgram` is an interpreting function for the `ControlProgram` language, `thermMonitorA` is the arrow to run, and `sensorInst` is the value the arrow is awaiting as the input. Running the arrow calls the interpreter for the top-level language, and the internal language interpreters will be called from it. We'll omit this code. What we'll see is the implementation of combinators.

"Run script X" arrows are simple — we just wrap every monadic action with the library arrow creator `mArr` for effective (monadic) functions:

```
thermTemperatureA :: FlowArr SensorInstance Measurement
thermTemperatureA = mArr f
  where
    f inst :: SensorInstance -> ControlProgram Measurement
    f inst = evalScript (readSensor Temperature inst)
```

Thus, `f` is the monadic function in the `ControlProgram` monad. It's a composition of two functions, the `evalScript` function and `readSensor`, the custom function defined like so:

```
readSensor :: Parameter -> SensorInstance -> Script Measurement
readSensor parameter (cont, idx) = controllerScript readSensor'
  where
      -- The script itself.
      -- "read" is a smart constructor.
      readSensor' :: ControllerScript Measurement
      readSensor' = read cont idx parameter
```

Figure 4.5 shows the structure of nested scripts. Top blocks are functions, bottom blocks are types.

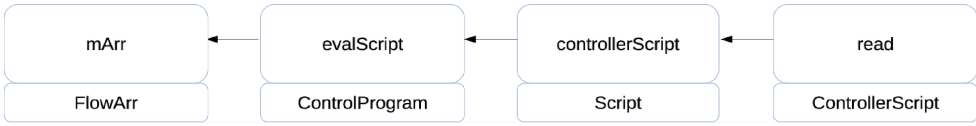


Figure 4.5: Scripts nesting.

The `readSensor` function puts the script in the `ControllerScript` monad into an intermediate `Script` container, the form the `evalScript` function is awaiting as input (see listing 4.12):

```
data Control a = forall b. EvalScript (Script b) (b -> a)

evalScript :: Script a -> ControlProgram a
evalScript scr = Free (EvalScript scr Pure)
```

We do the same with the infrastructure script and others:

```
getTimeA :: FlowArr b Time
getTimeA = mArr f
  where
    f :: b -> ControlProgram Time
    f _ = evalScript (infrastructureScript getCurrentTime)

storeReadingA :: FlowArr Reading ()
storeReadingA = mArr (evalScript . infrastructureScript . storeReading)
```

And also we convert pure functions to arrows with the library wrapper `arr`:

```
validateReadingA :: FlowArr Reading ValidationResult
validateReadingA = arr validateReading

validateReading :: Reading -> ValidationResult
validateReading (_, si, Measurement (FloatValue tempK)) = ...
```

Finally, when the arrowized language is filled with different arrows, we are able to write comprehensive scenarios in a combinatorial way — not just with monads! Let's weigh the pros and cons. The arrowized eDSL is good for a few reasons:

- *It's useful for flow diagrams*. This is a natural way to express flow scenarios to control the ship.
- *It's highly combinatorial and abstract*. As a result, you write less code. You don't even need to know what monad is running under the hood.

However, arrowized eDSLs have disadvantages too:

- *Arrows don't get the consideration they deserve and Free Arrows are not investigated properly*. This is a cutting-edge field of computer science, and there are not so many industrial applications of arrows.
- *They're harder to create*. We made a simple arrowized language over `Free` languages, but the library this language is built upon is complex.
- *Combinators for arrows such as `(&&&)` or `(>>>)` may blow your mind.* Besides combinators, how many languages have a special arrowized syntax? Only Haskell, unfortunately.

Arrows are quite interesting. There are combinators for choice evaluation of arrows for looped and recursive flows, which makes arrows an excellent choice for complex calculation graphs; for example, mathematical iterative formulas or electrical circuits. Some useful applications of arrows include effective arrowized parsers, XML processing tools, and functional reactive programming libraries. It's also important to note that the arrow concept, while being a functional idiom, has laws every arrow should obey. We won't enumerate them here, so as not to fall into details and to stay on the design level.

4.4 External DSLs

Every SCADA application has to support external scripting. This is one of the main functional requirements because SCADA is an environment that should be carefully configured and programmed for the particular industrial process. Scripting functionality may include:

- *Compilable scripts in programming languages such as C*. You write program code in a real language and then load it into a SCADA application directly or compile it into pluggable binaries.
- *Pluggable binaries with scripts*. After the binaries are plugged into the application, you have additional scripts in your toolbox.
- *Interpretable external DSLs*. You write scripts using an external scripting language provided by the application. You may save your scripts in text files and then load them into the application.

Perhaps, this is the most difficult part of the domain. All industrial SCADA systems are supplied with their own scripting language and integrated development environment (IDE) to write control code. They can also be powered by such tools as graphical editors, project designers, and code analyzers. All of these things are about programming language

compilation. Compilers, interpreters, translators, various grammars, parsing, code optimization, graph and tree algorithms, type theory, paradigms, memory management, code generation, and so on and so forth... This part of computer science is really big and hard.

We certainly don't want to be roped into compilation theory and practice: this book is not long enough to discuss even a little part of it! We'll try to stay on top of the design discussion and investigate the place and structure of an external eDSL in our application.

4.4.1 External DSL structure

First, we'll fix the requirement: Andromeda software should have an external representation of the Logic Control eDSLs with additional programming language possibilities. The internal representation, namely, ControlProgram Free eDSL, can be simply used in unit and functional tests of subsystems, and the external representation is for real scripts a spaceship will be controlled by. We call this external DSL *AndromedaScript*. The engineer should be able to load, save, and run AndromedaScript files. The mapping of eDSLs to AndromedaScript is not symmetric:

- Embedded Logic Control DSLs can be fully translated to AndromedaScript.
- AndromedaScript can be partially translated into the Logic Control eDSLs.

AndromedaScript should contain possibilities of common programming languages. This code can't be translated to eDSLs because we don't have such notions there: neither Free eDSL of Logic Control contains `if-then-else` blocks, variables, constants, and so on. This part of AndromedaScript will be transformed into the intermediate structures and then interpreted as desired.

Table 4.2 describes all the main characteristics of AndromedaScript.

Table 4.2 Main characteristics of AndromedaScript.

Characteristic	Description
Semantics	AndromedaScript is strict and imperative. Every instruction should be defined in a separate line. All variables are immutable. Delimiters aren't provided.
Code blocks	Code blocks should be organized by syntactic indentation with 4 spaces.
Type system	Implicit dynamic type system. Type correctness will be partially checked in the translation phase.
Supported constructions	`if-then-else`, range `for` loop, procedures and functions, immutable variables, custom lightweight algebraic data types.
Base library	Predefined data types, algebraic data types, procedures, mapping to the Logic Control eDSLs, mapping to the Hardware eDSLs.
Versions	Irrelevant at the start of the Andromeda project; only a single version is supported. Version policy will be reworked in the future when the syntax is stabilized.

The next big deal is to create a syntax for the language and write code examples. The examples can be meaningless, but they should show all the possibilities in pieces. We'll use them to test the translator. See the sample code in listing 4.13.

Listing 4.13 Example of AndromedaScript

```
val boosters = Controller ("boosters")
val start = Command ("start")
val stop = Command ("stop")
val success = Right ("OK")

// This procedure uses the ControllerScript possibilities.
[ControllerScript] BoostersOnOffProgram:
    val result1 = Run (boosters, start)
    if (result1 == success) then
        LogInfo ("boosters start success.")
        val result2 = Run (boosters, Command ("stop"))
        if (result2 == success) then
            LogInfo ("boosters stop success.")
        else
            LogError ("boosters stop failed.")
    else
        LogError ("boosters start failed.")

// Script entry point.
// May be absent if it's just a library of scripts.
Main:
    LogInfo ("script is started.")
    BoostersOnOffProgram
    LogInfo ("script is finished.")
```

You may notice there is no distinction between predefined value constructors such as `Controller` or `Command` and procedure calls such as `Run`. In Haskell, every value constructor of algebraic data type is a function that creates a value of this type — it's no different from a regular function. Knowing this, we simplify the language syntax by making every procedure and function a kind of value constructor. Unlike the Haskell syntax, arguments are comma-separated and bracketed, because it's easier to parse.

What parts should a typical compiler contain? Let's enumerate them:

- *Grammar description*. Usually a Backus–Naur Form (BNF) for simple grammars, but it can be a syntax diagram or a set of grammar rules over the alphabet.
- *Parser*. Code that translates the text of a program into the internal representation, usually an abstract syntax tree (AST). Parsing can consist of lexical and syntax analysis before the AST creation. The approaches for how to parse a certain language highly depend on the properties of its syntax and requirements of the compiler.
- *Translator, compiler, or interpreter*. Code that translates one representation of the program into another. Translators and compilers use translation rules to manipulate abstract syntax trees and intermediate structures.

The grammar of the AndromedaScript language is context-free, so it can be described by the BNF notation. We won't have it as a separate artifact because we'll be using the monadic parsing library and thus the BNF will take the form of parser combinators. There is no need

to verify the correctness of the BNF description: if the parser works right, then the syntax is correct. In our implementation, the parser will read text and translate the code into the AST, skipping any intermediate representation. There should be a translator that is able to translate a relevant part of the AST into the `ControlProgram` eDSL and an interpreter translator that does the opposite transformation. Why? Because it's our interface to the Logic Control subsystem and we assume we have all the machinery that connects the `ControlProgram` eDSL with real hardware and other subsystems. The rest of the AndromedaScript code will be evaluated by the AST interpreter.

The project structure is updated with a new top-level subsystem, `Andromeda.Language`:

```
Andromeda\
    Language                #A
    Language\
        External\           #B
            AST             #B
            Parser          #B
            Translator      #B
            Interpreter     #B
```

#A Top-level module that reexports modules of the compiler
#B AST, operational data structures, parsers, and translators of the AndromedaScript language

In the future, the external language will be complemented by the foreign programming language C, and we'll place that stuff into the folder *Andromeda\Language\Foreign*, near *Andromeda\Language\External*.

4.4.2 Parsing to the abstract syntax tree

The abstract syntax tree is a form of grammar in hierarchical data structures that is convenient for transformations and analysis. We can build the AST from top to bottom by taking the entire program code and descending to separate tokens, but it's likely we'll come to a dead end where it's not clear what element should be inside. For example, the main data type should contain a list of... what? Procedures? Statements? Declarations?

```
data Program = Program [???]
```

The better way to construct the AST is related to the BNF creation, or in our case, the parser creation, starting from small parsers and going up to the big ones. The Parsec library already has many important combinators. We also need combinators for parsing integer constants, string constants, identifiers, end-of-lines, and lists of comma-separated things between brackets. Here are some of them:

```
-- Integer constant parser. Returns Int.
integerConstant :: Parser Int
integerConstant = do
    res <- many1 digit  #A
    return (read res)   #B

-- Identifier parser: first character is lowercase,
-- others may be letters, digits, or underscores.
-- Returns parsed string.
identifier :: Parser String
identifier = do
    c <- lower <|> char '_'   #C
```

```
    rest <- many (alphaNum <|> char '_')
    return (c : rest)
```

#A Parse one or more digits and put to res
#B The variable res is a string; the read function converts it to an integer
#C Parse lowercase letter; if this fails, parse the underscore symbol

---

A note about monadic parsing combinators and the Parsec library

Parsec is great. Suppose you have a log file and you want to parse it:

```
10 [err] "Syntax error: unexpected '(' (line 4, column 23)"
```

It contains an integer number, severity, and message string, each delimited by one space (exactly). The corresponding parser will be:

```
logEntry :: LogParser (Int, Severity, String)
logEntry = do
    n <- integerConstant
    space
    s <- severity
    space
    msg <- stringConstant
    return (n, s, msg)
```

This parser calls smaller parsers for constants and for severity. It also consumes and throws out the spaces in between. The result is a triple that describes the log entry. The `severity` parser will look like so:

```
data Severity = Err | Inf

severity' s = between (char '[') (char ']') (string s)
errSeverity = severity' "err" >> return Err
infSeverity = severity' "inf" >> return Inf

severity = errSeverity <|> infSeverity
```

Here, the (`p1 <|> p2`) expression means if `p1` fails to parse, the `p2` will try. This reads as "or". Let's run the parser against some input string:

```
str = "10 [err] \"Syntax error: unexpected '(' (line 4, column 23)\""
test = case Parsec.parse severity str of
    Left e -> print "FAILED"
    Right (i, s, msg) -> print ("Parsed", i, s, msg)
```

The functions `between`, `char`, `string`, `(<|>)`, `space`, `alphaNum`, `digit`, `many`, `many1`, `lower`, and `upper` are standard parsing combinators with the obvious meanings. A shapely set of bricks for your mason's imagination!

Having plenty of small general parsers, we build bigger ones—for example, the parser for the value constructor entry. This gives us the corresponding algebraic data type:

```
data Constructor = Constructor String ArgDef

constructorName :: Parser String
```

```
constructorName = do
    bigChar <- upper
    smallChars <- many alphaNum
    return (bigChar : smallChars)

constructor :: Parser Constructor
constructor = do
    name <- constructorName
    spaces
    argDef <- argDef
    return (Constructor name argDef)
```

And then we have to define the parser and data type `argDef`. A small test of this concrete parser will show if we are doing things right or something is wrong. Parsec has the `parseTest` function for this (or you may write your own):

```
test :: IO ()
test = do
    let constructorStr = "Controller(\"boosters\")"
    parseTest constructor constructorStr
```

The AST we'll get this way can consist of dozens of algebraic data types with possibly recursive definitions. The AndromedaScript AST has more than 20 data types, and this is not the limit. Figure 4.6 shows the structure of it.

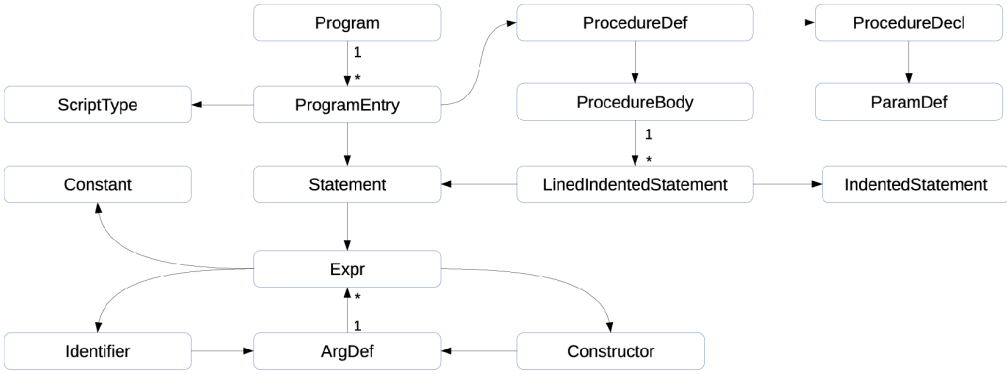


Figure 4.6 The AndromedaScript AST.

The more diverse your grammar, the deeper your syntax tree. And then you have to deal with it during the translation process. That's why Lisp is considered a language without syntax: it has only a few basic constructions that will fit into a tiny AST. Indeed, the s-expressions are the syntax trees themselves, which makes the transformation much simpler.

4.4.3 The translation subsystem

Although translation theory has more than a half-century history, the methods and tricks it suggests are still high science that can require significant adjacent skills and knowledge from the developer. We'll talk about the translator, focusing not on how to write one but on how it should be organized. Let's treat the translator as a subsystem that is responsible for interpreting the AST into evaluable code from one side and into embedded languages from the other side.

The translator is the code that pattern matches over the AST and does the necessary transformations while building a result representation of the script being processed. The translator has to work with different structures: some of them are predefined and immutable; others will be changed during the translation. Immutable data structures include symbol tables, number transition rules, and dictionaries. Tables may contain patterns and rules of optimization, transitions for state machines, and other useful structures. Mutable (operational) data structures include graphs, number generators, symbol tables, and flags controlling the process. The state of the translator highly depends on the tasks it's intended to solve, but it never is simple. Consequently, we shouldn't make it even more intricate by choosing the wrong abstractions.

There are several ways to make stateful computations in functional programming. From chapter 3 we know that if the subsystem has to work with state, the State monad and similar are the a good choices, unless we are worried by performance questions. It's very likely we'll want to print debug and log messages describing the translation process. The shortest (but not the best) path to do that is to make the translator impure. So this subsystem should have properties of two monads: `State` and `IO`, a frequent combination in Haskell applications. We'll study other options we have to carry the state in the next chapter.

We define the translation type as the state transformer with the `IO` monad inside, parameterized by the `Translator` type:

```
type TranslatorSt a = StateT Translator IO a
```

The `Translator` type is an algebraic data type that holds the operational state:

```
type Table = (String, Map String String)
type ScriptsTable = Map IdName (Script ())

data Tables = Tables {
      _constants :: Table
    , _values :: Table
    , _scriptDefs :: ScriptsDefsTable
    , _sysConstructors :: SysConstructorsTable
    , _scripts :: ScriptsTable
}

data Translator = Translator {
      _tables :: Tables
    , _controlProg :: ControlProgram ()
    , _scriptTranslation :: Maybe ScriptType
    , _indentation      :: Int
    , _printIndentation :: Int
    , _uniqueNumber     :: Int
}
```

The state has some management fields (`indentation`, `printIndentation`, `uniqueNumber`), tables, and other data. It is organized as nested ADTs. You may notice that nesting of data structures unavoidably complicates making changes when they are immutable: you need to unroll the structures from the outside in, modify an element, and roll them back up. You can mitigate this problem by providing mutation functions, but it becomes annoying to support all new data structures this way. Fortunately, there is a better approach, known as lenses. You may have heard about lenses or even be using them, but if not, I'll give you a very short overview.

Lenses are a way to generalize working with deep immutable data structures of any kind. Lenses are very similar to getters and setters in OOP, but they are combinatorial and do many things that getters and setters can't — for example, traversing a container and mutating every element inside the container or even deeper. You describe this operation with a few lens combinators and apply it to your container. The rest of the unrolling and wrapping of items will be done by the `lens` library. The following listing shows the idea.

Listing 4.14 Lenses simple example

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Lens (traverse, makeLenses, set)

data BottomItem = BottomItem { _str :: String }
data MiddleItem = MiddleItem { _bottomItem :: BottomItem }
data TopItem    = TopItem    { _middleItem :: MiddleItem }

-- Making lenses with TemplateHaskell:
makeLenses 'BottomItem
makeLenses 'MiddleItem
makeLenses 'TopItem

-- Now you have lenses for all underscored fields.
-- Lenses have the same names except underscores.
-- Lenses can be combined following the hierarchy of the types:
bottomItemLens = traverse.middleItem.bottomItem.str

container = [ TopItem (MiddleItem (BottomItem "ABC"))
            , TopItem (MiddleItem (BottomItem "CDE"))]

expected = [ TopItem (MiddleItem (BottomItem "XYZ"))
           , TopItem (MiddleItem (BottomItem "XYZ"))]

container' = set bottomItemLens "XYZ" container
test = print (expected == container')
```

Here the `set` combinator works over any lens irrespective the structure it points to. The structure may be a single value lying deeply or a range of values inside any traversable container (lists, arrays, trees are the example). The `test` function will print `True` because we have changed the internals of the container by applying to it the `set` combinator and the lens `bottomItemLens`, which pointed out what item to change. The definition of a lens looks like a chain of accessors to internal structures in the OOP manner, but it's a functional composition of smaller lenses that know how to address the particular part of the compound structure:

```
middleItem :: Lens TopItem MiddleItem
```

```
bottomItem :: Lens MiddleItem BottomItem
str        :: Lens BottomItem String

(middleItem . bottomItem)       :: Lens TopItem BottomItem
(middleItem . bottomItem . str) :: Lens TopItem String

traverse . middleItem . bottomItem . str :: Lens [TopItem] String
```

These small lenses are made by the `lens` library from named fields of algebraic data types prefixed by an underscore. It's the naming convention of Haskell's `lens` library that indicates what lenses we want to build for. Returning to the translator's state, we can see it has many fields with underscore prefixes — that is, we'll get a bunch of lenses for these underscored fields.

Both Haskell and Scala have lens libraries with tons of combinators. The common operations are: extracting values, mutating values, producing new structures, testing matches with predicates, transforming, traversing, and folding container-like structures. Almost any operation you can imagine can be replaced by a lens applied to the structure. What else is very important is that many of Haskell's lens combinators are designed to work inside the `State` monad: you don't have to store results in the variables, but you mutate your state directly (in the sense of the State monad mutation). For example, the translator tracks whether the syntactic indentation is correct. For this purpose it has the `_indentation` field, and there are two functions that increase and decrease the value:

```
incIndentation :: TranslatorSt ()
incIndentation = indentation += 1

decIndentation :: TranslatorSt ()
decIndentation = do
    assertIndentation (>0)
    indentation -= 1

assertIndentation :: (Int -> Bool) -> TranslatorSt ()
assertIndentation predicate = do
    i <- use indentation
    assert (predicate i) "wrong indentation:" I
```

Here, `indentation` is the lens pointing to the `_indentation` field inside the `Translator` data type. The `use` combinator reads the value of the lens from the context of the `State` monad. Note how the operators `(+=)` and `(-=)` make this code look imperative! Building a translator can be really hard. Why not make it less hard by plugging in lenses? I stop here studying the translation subsystem. If you want, you can keep going, digging into the code of the Andromeda software available on GitHub.

4.5 Summary

What are the reasons to develop domain-specific languages? We want to accomplish goals such as the following:

- Investigate the domain and define its properties, components, and laws.
- Based on the domain properties, design a set of domain languages in a form that is more suitable and natural for expressing user scenarios.
- Make the code testable.

- Follow the Single Responsibility Principle, keeping accidental complexity as low as possible.

In functional programming, it's more natural to design many domain-specific languages to model a domain. The myth that this is complicated has come from mainstream languages. The truth here is that traditional imperative and object-oriented languages weren't intended to be tools for creating DSLs. Neither the syntax nor the philosophy of imperative languages is adapted to supporting such a development approach. You can't deny the fact that when you program something, you are creating a sort of domain language to solve your problem, but when the host language is imperative, it's more likely that your domain language will be atomized and dissolved in unnecessary rituals. As a result, you have that domain language, but you can't see it; you have to dissipate your attention on many irrelevant things.

Imagine you wonder what a ball of water looks like, and you mold a lump of wet earth to find out. Well, it's spherical and has water inside, but you didn't get an answer to your question. You can only really see a ball of water in free fall or in zero gravity. Functional programming is like that water ball. Due to its abstractions, a domain can be finely mapped to code without any lumps of earth. When nothing obfuscates your domain language, the maintenance of it becomes simple and obvious, and the risk of bugs decreases.

The techniques and patterns for designing domain-specific languages we have discussed in this chapter are by no means comprehensive, and our work on the Logic Control subsystem is still not complete. The simple, "straightforward" eDSLs we developed first can be good, but it seems the monadic ones have more advantages. The `Free` monad pattern helps to build a scenario that we can interpret. In doing so, we separate the logic of a domain from the implementation. We also wrapped our `Free` languages into an arrowized interface. With it, we were able to illustrate our scenarios using flow diagrams.

You may ask why our domain model missed out the requirement to run scripts by time or event condition. We could probably model this in an event-driven manner: we run this functionality when we catch an event the special subsystem produces in a time interval. But this design often blurs the domain model because the time conditional logic lies too far from the domain logic, and changing of time conditions can be really hard. Also, we can't really interpret the "runnable" code. The second option to do that is to expand the `Control` eDSL with a special language, something like this:

```
data Control a = forall b. EvalScript (Script b) (b -> a)
               | forall b. EvalByTime Time (Script b) (b -> a)
               | forall b. EvalOnEvent Event (Script b) (b -> a)
```

But introducing such actions immediately makes our eDSL reactive (that is, the actions are reactions to events). To be honest, the domain of Logic Control has this property: it's really reactive, and we want to write reactive scenarios. But we are trying to invent functional reactive programming. Again, we have two options: use existing FRP libraries somehow, or continue developing our own with the functionality limited. The first option is inappropriate because our scenarios should be interpretable. Consequently, it's necessary to create a custom interpretable FRP library. However this will be a bit harder task than we can imagine. In further, we'll see some ways to create reactivity on the base of Software Transactional Memory. We'll leave the question about FRP for future books and materials, because it's really huge.