

Functional Programming for Domain-Specific Languages

Jeremy Gibbons

Department of Computer Science, University of Oxford
<http://www.cs.ox.ac.uk/jeremy.gibbons/>

Abstract. Domain-specific languages become effective only in the presence of convenient lightweight tools for defining, implementing, and optimizing new languages. Functional programming provides a promising framework for such tasks; FP and DSLs are natural partners. In these lectures we will discuss FP techniques for DSLs—especially standalone versus embedded DSLs, and shallow versus deep embeddings.

1 Introduction

In his book [1], Fowler defines a domain-specific language (DSL) as

a computer programming language of limited expressiveness focussed on a particular domain

A DSL is targetted at a specific class of programming tasks; it may indeed not be Turing-complete. By restricting scope to a particular domain, one can tailor the language specifically for that domain. Common concepts or idioms in the domain can be made more easily and directly expressible—even at the cost of making things outside the intended domain more difficult to write. The assumptions common to the domain may be encoded within the language itself, so that they need not be repeated over and over for each program within the domain—and again, those assumptions may be inconsistent with applications outside the domain.

The term ‘DSL’ is rather more recent than its meaning; DSLs have been prevalent throughout the history of computing. As Mernik et al. [2] observe, DSLs have in the past been called ‘application-oriented’, ‘special-purpose’, ‘specialized’, ‘task-specific’, and ‘application’ languages, and perhaps many other things too. The ‘fourth-generation languages’ (4GLs) popular in the 1980s were essentially DSLs for database-oriented applications, and were expected at the time to supercede general-purpose 3GLs such as Pascal and C. One might even say that Fortran and Cobol were domain-specific languages, focussed on scientific and business applications respectively, although they are both Turing-complete. Bentley [3] wrote influentially in his Programming Pearls column about the ‘little languages’ constituting the philosophy and much of the functionality of the Unix operating system: tools for programmers such as the shell, regular expressions, lex and yacc, and tools for non-programmers such as the Pic language for line drawings and a language for specifying surveys.

There are two main approaches to implementing DSLs. The historically prevalent approach has been to build *standalone* languages, with their own custom syntax. Standard compilation techniques are used to translate or interpret programs written in the DSL into a general-purpose language (GPL) for execution. This has the advantage that the syntax of the DSL can be designed specifically for the intended users, and need bear no relation to that of the host language—indeed, there may be many different host languages, as there are for SQL, or the DSL ‘syntax’ may be diagrammatic rather than textual.

Conversely, implementing a new standalone DSL is a significant undertaking, involving a separate parser and compiler, and perhaps an interactive editor too. Moreover, the more the DSL is a kind of ‘programming’ language, the more likely it is that it shares some features with most GPLs—variables, definitions, conditionals, etc—which will have to be designed and integrated with the DSL. In the process of reducing repetition and raising the level of abstraction for the DSL programmer, we have introduced repetition and lowered the level of abstraction for the DSL implementer. That may well be a rational compromise. But is there a way of getting the best of both worlds?

The second approach to implementing DSLs attempts exactly that: to retain as much as possible of the convenient syntax and raised level of abstraction that a DSL provides, without having to go to the trouble of defining a separate language. Instead, the DSL is *embedded* within a host language, essentially as a library of definitions written in the host GPL (although it is debatable to what extent ‘library’ and ‘language’ coincide: we return to this point in Section 2.1 below). All the existing facilities and infrastructure of the host environment can continue to be used, and familiarity with the syntactic conventions of the host can be carried over to the DSL.

However, there are some downsides. DSL programs have to be written in the syntax of the host language; this may be clumsy if the the host syntax is rigid, and daunting to non-programmer domain specialists if the host syntax is obscure. It can be difficult to preserve the abstract boundary between the DSL its host: naive users may unwittingly invoke sophisticated language features, and error messages may be reported unhelpfully in terms of the host language rather than the DSL. Needless to say, these issues are hot research topics among those working on embedded DSLs.

Fowler [1] calls the standalone and embedded approaches ‘external’ and ‘internal’ respectively. He does this not least because ‘embedded’ suggests misleadingly that specialized code written in a DSL is quoted verbatim within a host program written in a GPL, with the whole being expressed in a hybrid language that is neither the DSL nor the GPL; for example, JavaServer Pages ‘programs’ are hybrids, consisting of HTML markup interspersed with fragments of Java. (In fact, Fowler calls such hybrids ‘fragmentary’, and uses the term ‘standalone’ for pure-bred DSLs, in which any program is written in just one language, whether internal or external.) That objection notwithstanding, we will stick in this article to the terms ‘standalone’ and ‘embedded’.

We will concentrate in this article on embedded DSLs, only briefly making the connection back to standalone DSLs. Again, there are two main approaches to embedded DSLs, which are conventionally called *deep* and *shallow* embedding. With a deep embedding, terms in the DSL are implemented simply to construct an abstract syntax tree; this tree is subsequently transformed for optimization and traversed for evaluation. With a shallow embedding, terms in the DSL are implemented directly as the values to which they evaluate, bypassing the intermediate AST and its traversal. We explore this distinction in Section 2.4.

It turns out that functional programming languages are particularly well suited for hosting embedded DSLs. Language features such as algebraic datatypes, higher-order functions, lazy evaluation, and rich type systems supporting type inference all contribute. We discuss these factors in more detail in Section 3.

The syntax of the host language is another factor, albeit a relatively minor one: functional languages often have lightweight syntax, for example favouring the use of whitespace and layout rather than punctuation for expressing program structure, and strongly supporting orthogonality of naming in the sense that both symbolic as well as alphabetic identifiers may be used in definitions. Both of these features improve flexibility, so that an embedded DSL can have a syntax close to what one might provide for a corresponding standalone DSL. Of course, there are functional languages with noisy syntactic conventions, and non-functional languages with quiet ones, so this factor doesn't map precisely onto the language paradigm. We make no more of it in this article, simply using Haskell syntax for convenience.

We use a number of little examples of embedded DSLs throughout the first part of the article. We conclude in Section 4, with a more detailed study of one particular embedded DSL, namely Yorgey's Diagrams package [4].

2 Exploring the design space

In the interests of focussing on the essence of DSLs, we start with a very simple example: a DSL for finite sets of integers. This consists of a representation of sets, and a number of operations manipulating that representation:

```
type IntegerSet = ...
empty  :: IntegerSet
insert :: Integer → IntegerSet → IntegerSet
delete :: Integer → IntegerSet → IntegerSet
member :: Integer → IntegerSet → Bool
```

For example, one might evaluate the expression

```
member 3 (insert 1 (delete 3 (insert 2 (insert 3 empty))))
```

and get the result *False* (assuming the usual semantics of these operations).

2.1 Libraries

The first approach one might take to implementing this characterization of integer sets might be as a *library*, that is, as a collection of related functions. For example, one might represent the set as a list, possibly with duplicates and treating order as insignificant:

```
type IntegerSet = [Integer] -- unsorted, duplicates allowed

empty :: IntegerSet
empty = []

insert :: Integer → IntegerSet → IntegerSet
insert x xs = x : xs

delete :: Integer → IntegerSet → IntegerSet
delete x xs = filter (≠ x) xs

member :: Integer → IntegerSet → Bool
member x xs = any (≡ x) xs
```

(Here, the standard library function $\text{any } p = \text{foldr } ((\vee) \circ p) \text{ False}$ determines whether any element of a list satisfies predicate p .)

We have been writing code in this style—that is, collections of types and related functions—from the earliest days of computing. Indeed, compilers are so called because they ‘compile’ (collect and assemble the pieces for) an executable by linking together the programmer’s main program with the necessary functions from the library. The problem with this style is that there is no encapsulation of the data representation: it is evident to all clients of the abstraction that the representation uses lists, and client code may exploit this knowledge by using other list functions on the representation. The representation is public knowledge, and it becomes very difficult to change it.

2.2 Modules

The realisation that libraries expose data representations prompted the notion of *modular programming*, especially as advocated by Parnas [5]: code should be partitioned into modules, and in particular, the modules should be chosen so that each hides a design decision (such as, but not necessarily, a choice of data representation) from all the others, allowing that decision subsequently to be changed.

The modular style that Parnas espouses presupposes mutable state: the module hides a single data structure, and operations query and modify the value of that data structure. Because of this dependence on mutable state, it is a little awkward to write in the Parnas style in a pure functional language like Haskell. To capture this behaviour using only pure features, one adapts the operations so that each accepts the ‘current’ value of the data structure as an additional argument, and returns the ‘updated’ value as an additional result. Thus, an impure function of type $a \rightarrow b$ acting statefully on a data structure of type s can be represented as a pure function of type $(a, s) \rightarrow (b, s)$, or equivalently by

currying, $a \rightarrow (s \rightarrow (b, s))$. The return part $s \rightarrow (b, s)$ of this is an instance of the *state monad*, implemented in the Haskell standard library as a type *State s b*. Then the set module can be written as follows:

```
module SetModule (Set, runSet, insert, delete, member) where
  import Control.Monad.State
  type IntegerSet = [Integer]
  newtype Set a = S {runS :: State IntegerSet a}
  instance Monad Set where
    return a = S (return a)
    m >>= k = S (runS m >>= (runS o k))

  runSet :: Set a → a
  runSet x = evalState (runS x) []

  insert :: Integer → Set ()
  insert x = S $ do { modify (x:) }

  delete :: Integer → Set ()
  delete x = S $ do { modify (filter (≠ x)) }

  member :: Integer → Set Bool
  member x = S $ do { xs ← get; return (any (≡ x) xs) }
```

Here, the type *Set* of stateful operations on the set is abstract: the representation is not exported from the module, only the type and an observer function *runSet* are. The operations *insert*, *delete*, and *member* are also exported; they may be sequenced together to construct larger computations on the set. But the only way of observing this larger computation is via *runSet*, which initializes the set to empty before running the computation. Haskell's **do** notation conveniently hides the plumbing required to pass the set representation from operation to operation:

```
runSet $ do { insert 3; insert 2; delete 3; insert 1; member 3 }
```

(To be precise, this stateful programming style does not really use mutable state: all data is still immutable, and each operation that ‘modifies’ the set in fact constructs a fresh data structure, possibly sharing parts of the original. Haskell does support true mutable state, with imperative in-place modifications; but to do this with the same interface as above requires the use of unsafe features, in particular *unsafePerformIO*.)

2.3 Abstract datatypes

Parnas's approach to modular programming favours modules that hide a single data structure; the attentive reader will note that it is easy to add a *union* operation to the library, but difficult to add one to the module. A slightly different approach is needed to support data abstractions that encompass multiple data structures—*abstract datatypes*. In this case, the module exports an abstract

type, which specifies the hidden representation, together with operations to create, modify and observe elements of this type.

```
module SetADT (IntegerSet, empty, insert, delete, member) where

  newtype IntegerSet = IS [Integer]

  empty :: IntegerSet
  empty = IS []

  insert :: IntegerSet → Integer → IntegerSet
  insert (IS xs) x = IS (x : xs)

  delete :: IntegerSet → Integer → IntegerSet
  delete (IS xs) x = IS (filter ( $\neq$  x) xs)

  member :: IntegerSet → Integer → Bool
  member (IS xs) x = any ( $\equiv$  x) xs
```

Note that in addition to the three operations *insert*, *delete* and *member* exported by *SetModule*, we now export the operation *empty* to create a new set, and the abstract type *IntegerSet* so that we can store its result, but not the constructor *IS* that would allow us to deconstruct sets and to construct them by other means than the provided operations. Note also that we can revert to a purely functional style; there is no monad, and ‘modifiers’ manifestly construct new sets—this was not an option when there was only one set. Finally, note that we have rearranged the order of arguments of the three operations, so that the source set is the first argument; this gives the feeling of an object-oriented style, whereby one ‘sends the *insert* message to an *IntegerSet* object’:

```
((((empty ‘insert’ 3) ‘insert’ 2) ‘delete’ 3) ‘insert’ 1) ‘member’ 3
```

2.4 Languages

One might, in fact, think of the abstract datatype *SetADT* as a DSL for sets, and the set expression above as a term in this DSL—there is at best a fuzzy line between ADTs and embedded DSLs. If one were to make a formal distinction between ‘languages’ and ‘libraries’, it would presumably be that a ‘language’ privileges one particular datatype whose elements represent terms in the language, with constructors that compose terms, and observers that analyse terms; a ‘library’, on the other hand, is just a collection of related functions, and may have no such privileged datatype.

The *SetADT* implementation above can be seen as an intermediate point on the continuum between two extreme approaches to implementing embedded DSLs: *deep* and *shallow embedding*. In a deep embedding, the operations that construct elements of the abstraction do as little work as possible—they simply preserve their arguments in an abstract syntax tree.

```
module SetLanguageDeep (IntegerSet (Empty, Insert, Delete), member) where

  data IntegerSet :: * where
```

```

Empty :: IntegerSet
Insert :: IntegerSet → Integer → IntegerSet
Delete :: IntegerSet → Integer → IntegerSet

member :: IntegerSet → Integer → Bool
member Empty y = False
member (Insert xs x) y = (y ≡ x) ∨ member xs y
member (Delete xs x) y = (y ≠ x) ∧ member xs y

```

Now we declare and export an algebraic datatype *IntegerSet* as the implementation of the three operations that yield a set; we have used Haskell’s generalized algebraic datatype notation to emphasize their return types, even though we make no use of the extra expressive power of GADTs. The *member* operation, on the other hand, is implemented as a traversal over the terms of the language, and is not itself part of the language.

```

((((Empty ‘Insert‘ 3) ‘Insert‘ 2) ‘Delete‘ 3) ‘Insert‘ 1) ‘member‘ 3

```

Whereas in a deep embedding the constructors do nothing and the observers do all the work, in a shallow embedding it is the other way round: the observers are trivial, and all the computation is in the constructors. Given that the sole observer in our set abstraction is the membership function, the shallow embedding represents the set directly as this membership function:

```

module SetLanguageShallow (IntegerSet, empty, insert, delete, member) where

  newtype IntegerSet = IS (Integer → Bool)

  empty  :: IntegerSet
  empty  = IS (λy → False)

  insert :: IntegerSet → Integer → IntegerSet
  insert (IS f) x = IS (λy → (y ≡ x) ∨ f y)

  delete :: IntegerSet → Integer → IntegerSet
  delete (IS f) x = IS (λy → (y ≠ x) ∧ f y)

  member :: IntegerSet → Integer → Bool
  member (IS f) = f

```

It is used in exactly the same way as the *SetADT* definition:

```

((((empty ‘insert‘ 3) ‘insert‘ 2) ‘delete‘ 3) ‘insert‘ 1) ‘member‘ 3

```

We have only a single observer, so the shallow embedding is as precisely that observer, and the observer itself is essentially the identity function. More generally, there may be multiple observers; then the embedding would be as a tuple of values, and the observers would be projections.

In a suitable sense, the deep embedding can be seen as the most abstract implementation possible of the given interface, and the shallow embedding as the most concrete: there are transformations from the deep embedding to any intermediate implementation, such as *SetADT*—roughly,

```

elements :: SetLanguageDeep.IntegerSet → SetADT.IntegerSet
elements Empty      = []
elements (Insert xs x) = x : elements xs
elements (Delete xs x) = filter (≠ x) (elements xs)

```

and from this to the shallow embedding—roughly,

```

membership :: SetADT.IntegerSet → SetLanguageShallow.IntegerSet
membership xs = λx → any (≡ x) xs

```

Expressed categorically, there is a category of implementations and transformations between them, and in this category the deep embedding is the initial object and the shallow embedding the final object [6]. The shallow embedding arises by deforesting the abstract syntax tree that forms the basis of the deep embedding.

Kamin [7] calls deep and shallow embedding *operational* and *denotational domain modelling*, respectively, and advocates the latter in preference to the former. Erwig and Walkingshaw [8] call shallow embedding *semantics-driven design*, and also favour it over what they might call *syntax-driven design*.

Deep embedding makes it easier to extend the DSL with new observers, such as new analyses of programs in the language: just define a new function by induction over the abstract syntax. But it is more difficult to extend the syntax of the language with new operators, because each extension entails revisiting the definitions of all existing observers. Conversely, shallow embedding makes new operators easier to add than new observers. This dichotomy is reminiscent of that between OO programs structured around the VISITOR pattern [9] and those in the traditional OO style with methods attached to subclasses of an abstract *Node* class. The challenge of getting the best of both worlds—extensibility in both dimensions at once—has been called the *expression problem* [10].

2.5 Embedded and standalone

All the approaches described above have been for *embedded* DSLs, of one kind or another: ‘programs’ in the DSL are simply expressions in the host language. An alternative approach is given by *standalone* DSLs. As the name suggests, a standalone DSL is quite independent of its implementation language: it may have its own syntax, which need bear no relation to that of the implementation language—indeed, the same standalone DSL may have many implementations, in many different languages, which need have little in common with each other. Of course, being standalone, the DSL cannot depend on any of the features of any of its implementation languages; everything must be build independently, using more or less standard compiler technology: lexer, parser, optimizer, code generator, etc.

Fortunately, there is a shortcut. It turns out that a standalone DSL can share much of the engineering of the embedded DSL—especially if one is not so worried about absolute performance, and is more concerned about ease of implementation. The standalone DSL can be merely a frontend for the embedded DSL; one only needs to write a parser converting strings in the standalone DSL

to terms in the embedded DSL. (In fact, Parnas made a similar point over forty years ago [5]: he found that many of the design decisions—and hence the modules—could be shared between a compiler and an interpreter for the same language, in his case for Markov processes.)

For example, suppose that we are given a type *Parser a* of parsers reading values of type *a*

```
type Parser a = ...
```

and an observer that applies a parser to a string and returns either a value or an error message:

```
runParser :: Parser a → String → Either a String
```

Then one can write a parser *program :: Parser Bool* for little set programs in a special syntax; perhaps “{ }+3+2-3+1?3” should equate to the example expressions above, with “{ }” denoting the empty set, “+” and “-” the insertion and deletion operations, and “?” the membership test. Strings recognized by *program* are interpreted in terms of *insert*, *delete* etc, using one of the various implementations of sets described above. Then a simple wrapper program reads a string from the command line, tries to parse it, and writes out the boolean result or an error message:

```
main :: IO ()
main = do
  ss ← getArgs
  case ss of
    [s] → case runParser program s of
      Left b  → putStrLn ("OK: " ++ show b) -- parsed
      Right s → putStrLn ("Failed: " ++ s)   -- not parsed
    _    → do
      n ← getProgName
      putStrLn ("Usage: " ++ n ++ " <set-expr>")
```

Thus, from the command line:

```
> ./sets "{ }+3+2-3+1?3"
OK: False
```

We will return to this example in Section 3.3. Of course, parsers can be expressed as another DSL...

3 Functional programming for embedded DSLs

Having looked around the design space a little, we now step back to consider what it is about functional programming that makes it particularly convenient for implementing embedded DSLs. After all, a good proportion of the work on DSLs expressed in the functional paradigm focusses on embedded DSL; and conversely, most work on DSLs in other (such as OO) paradigms focusses on standalone

DSLs. Why is that? We contend that there are three main aspects of modern functional programming that play a part: they are both useful for implementing embedded DSLs, and absent from most other programming paradigms. These are *algebraic datatypes*, *higher-order functions*, and (perhaps to a lesser extent) *lazy evaluation*. We discuss each in turn, and illustrate with more simple examples of embedded DSLs. Some parts are left as exercises.

3.1 Algebraic datatypes

The deep embedding approach depends crucially on algebraic datatypes, which are used to represent abstract syntax trees for programs in the language. Without a lightweight mechanism for defining and manipulating new tree datatypes, this approach becomes unworkably tedious.

Algebraic datatypes are extremely convenient for representing abstract syntax trees within the implementation of a DSL. Operations and observers typically have simple recursive definitions, inductively defined over the structure of the tree; optimizations and transformations are often simple rearrangements of the tree—for example, rotations of tree nodes to enforce right-nesting of associative operators.

In addition to this, algebraic datatypes are also extremely convenient for making connections outside the DSL implementation. Often the DSL is one inhabitant of a much larger software ecosystem; while an embedded implementation within a functional programming language may be the locally optimal choice for this DSL, it may have to interface with other inhabitants of the ecosystem, which for legacy reasons or because of other constraints require completely different implementation paradigms. (For example, one might have a DSL for financial contracts, interfacing with Microsoft Excel at the front end for ease of use by domain specialists, and with monolithic C++ pricing engines at the back end for performance.) Algebraic datatypes form a very useful marshalling format for integration, parsed from strings as input and pretty-printed back to strings as output.

Consider a very simple language of arithmetic expressions, involving integer constants and addition. As a deeply embedded DSL, this can be captured by the following algebraic datatype:

```
data Expr = Val Integer
          | Add Expr Expr
```

Some people call this datatype *Hutton's Razor*, because Graham Hutton has been using it for years as a minimal vehicle for exploring many aspects of compilation [11].

Exercises

1. Write an observer for the expression language, evaluating expressions as integers.

$$eval :: Expr \rightarrow Integer$$

2. Write another observer, printing expressions as strings.

print :: *Expr* → *String*

3. Reimplement the expression language using a shallow embedding, such that the interpretation is that of evaluations.

```
type Expr = Integer
val  :: Integer      → Expr
add  :: Expr → Expr → Expr
```

4. Reimplement the expression language via a shallow embedding again, but this time such that the interpretation is that of printing.

```
type Expr = String
val  :: Integer      → Expr
add  :: Expr → Expr → Expr
```

5. Reimplement via a shallow embedding again, such that the interpretation provides *both* evaluation and printing.
6. What interpretation of the shallow embedding provides the deep embedding? Conversely, given the deep embedding, what additional computation is needed to obtain the various interpretations we have used as shallow embeddings?
7. What if you wanted a third interpretation, say computing the *size* of an expression? What if you wanted to allow ten different interpretations? What about allowing for unforeseen future interpretations?

3.2 Generalized algebraic datatypes

The *Expr* DSL above is untyped, or rather “untyped”: there is only a single type involved, namely integer expressions. Suppose that we want to represent both integer- and boolean-valued expressions:

```
data Expr = ValI Integer
          | Add Expr Expr
          | ValB Boolean
          | And Expr Expr
          | EqZero Expr
          | If Expr Expr Expr
```

The idea is that *EqZero* yields a boolean value (whether its argument evaluates to zero), and *If* should take a boolean-valued expression as its first argument. But what can we do for the evaluation function? Sometimes it should return an integer, sometimes a boolean. One simple solution is to make it return an *Either* type:

```
eval :: Expr → Either Integer Bool
eval (ValI n)    = Left n
eval (Add x y)   = case (eval x, eval y) of (Left m, Left n) → Left (m + n)
```

```

eval (ValB b)    = Right b
eval (And x y)   = case (eval x, eval y) of (Right a, Right b) → Right (a ∧ b)
eval (EqZero x)  = case eval x of Left n → Right (n ≡ 0)
eval (If x y z)  = case eval x of Right b → if b then eval y else eval z

```

This is rather clumsy. For one thing, *eval* has become a partial function; there are improper values of type *Expr* such as *EqZero (ValB True)* on which *eval* is undefined. For a second, all the tagging and untagging of return types is a source of inefficiency. Both of these problems are familiar symptoms of dynamic type checking; if we could statically check the types instead, then we could rule out ill-typed programs, and also abolish the runtime tags—a compile-time proof of well-typedness prevents the former and eliminates the need for the latter.

A more sophisticated solution, and arguably The Right Way, is to use dependent types, as discussed by Edwin Brady elsewhere in this Summer School. There are various techniques one might use; for example, one might tuple values with value-level codes for types, provide an interpretation function from codes to the types they stand for, and carry around “proofs” that values do indeed inhabit the type corresponding to their type code.

Haskell provides an intermediate, lightweight solution in the form of *type indexing*, through so-called *generalized algebraic datatypes* or GADTs. Let us rewrite the *Expr* datatype in an equivalent but slightly more repetitive form:

```

data Expr :: * where
  ValI  :: Integer          → Expr
  Add   :: Expr → Expr      → Expr
  ValB  :: Bool             → Expr
  And   :: Expr → Expr      → Expr
  EqZero :: Expr            → Expr
  If     :: Expr → Expr → Expr → Expr

```

This form lists the signatures of each of the constructors; of course, they are all constructors for the datatype *Expr*, so they all repeat the same return type *Expr*. But this redundancy allows us some flexibility: we might allow the constructors to have different return types. Specifically, GADTs allow the constructors of a polymorphic datatype to have return types that are instances of the type being returned, rather than the full polymorphic type.

In this case, we make *Expr* a polymorphic type, but only provide constructors for values of type *Expr Integer* and *Expr Bool*, and not for other instances of the polymorphic type *Expr a*.

```

data Expr :: * → * where
  ValI  :: Integer          → Expr Integer
  Add   :: Expr Integer → Expr Integer → Expr Integer
  ValB  :: Bool             → Expr Bool
  And   :: Expr Bool → Expr Bool → Expr Bool
  EqZero :: Expr Integer    → Expr Bool
  If     :: Expr Bool → Expr a → Expr a → Expr a

```

We use the type parameter as an index: a term of type $Expr\ a$ is an expression that evaluates to a value of type a . Evaluation becomes much simpler:

```
eval :: Expr a → a
eval (ValI n)    = n
eval (Add x y)   = eval x + eval y
eval (ValB b)    = b
eval (And x y)   = eval x ∧ eval y
eval (EqZero x)  = eval x ≡ 0
eval (If x y z)  = if eval x then eval y else eval z
```

As well as being simpler, it is also safer (there is no possibility of ill-typed expressions, and *eval* is a total function again) and swifter (there are no runtime *Left* and *Right* tags to manipulate any more).

Exercises

8. The type parameter a in $Expr\ a$ is called a *phantom type*: it doesn't represent contents, as the type parameter in a container datatype such as $List\ a$ does, but some other property of the type. Indeed, there need be no *Boolean* inside an expression of type $Expr\ Bool$; give an expression of type $Expr\ Bool$ that contains no *Bools*. Is there always an *Integer* inside an expression of type $Expr\ Integer$?
9. How do Exercises 2 to 7 work out in terms of GADTs?

3.3 Higher-order functions

Deep embeddings lean rather heavily on algebraic datatypes. Conversely, shallow embeddings depend on *higher-order functions*—functions that accept functions as arguments or return them as results—and more generally on *functions as first-class citizens* of the host language. A simple example where this arises is if we were to extend the *Expr* DSL to allow for **let** bindings and variable references:

```
val :: Integer          → Expr
add :: Expr → Expr      → Expr
var :: String           → Expr
bnd :: (String, Expr) → Expr → Expr
```

The idea is that *bnd* represents **let**-bindings and *var* represents variable references, so that

```
bnd ("x", val 3) (add (var "x") (var "x"))
```

corresponds to the Haskell expression **let** $x = 3$ **in** $x + x$. The standard structure of an evaluator for languages with such bindings is to pass in and manipulate an *environment* of bindings from variables to values (not to expressions):

```
type Env = [(String, Integer)]
eval :: Expr → Env → Integer
```

The environment is initially empty, but is augmented when evaluating the body of a **let** expression. With a shallow embedding, the interpretation *is* the evaluation function:

type *Expr* = *Env* → *Integer*

That is, expressions are represented not as integers, or strings, or pairs, but as functions (from environments to values).

Exercises

10. Complete the definition of the *Expr* DSL with **let** bindings, via a shallow embedding whose interpretation provides evaluation in an environment.
11. Look again at Exercise 7: can we define a shallow embedding that allows for unforeseen future interpretations? Hint: think about a ‘generic’ or ‘parametrized’ interpretation, as a higher-order function, which can be instantiated to yield evaluation, or printing, or any of a number of other concrete interpretations. What is common to the evaluation and printing interpretations above, and what is specific? What kinds of function is it sensible to consider as ‘interpretations’, and what should be ruled out?

A larger and very popular example of shallow embeddings with functional interpretations is given by *parser combinators*. Recall the type *Parser a* of parsers recognizing values of type *a* from Section 2.5; such a parser is roughly a function of type *String* → *a*. But we will want to combine parsers sequentially, so it is important that a parser also returns the remainder of the string after recognizing a chunk, so it would be better to use functions of type *String* → (*a*, *String*). But we will also want to allow parsers that fail to match (so that we can try a series of alternatives until one matches), and more generally parsers that match in multiple ways, so it is better still to return a list of results:

type *Parser a* = *String* → [(*a*, *String*)]

(Technically, these are more than just parsers, because they combine semantic actions with recognizing and extracting structure from strings. But the terminology is well established.)

The *runParser* function introduced in Section 2.5 takes such a parser and an input string, and returns either a successful result or an error message:

```
runParser :: Parser a → String → Either a String
runParser p s = case p s of
  [(a, s)] → if all isSpace s then Left a else Right ("Leftover input: " ++ s)
  []       → Right "No parse"
  x        → Right ("Ambiguous, with leftovers " ++ show (map snd x))
```

If the parser yields a single match, and any leftover input is all whitespace, we return that value; if there is a nontrivial remainder, no match, or multiple matches, we return an error message.

Such parsers can be assembled from the following small set of combinators:

```

success :: a                → Parser a
failure ::                  Parser a
(⟨*) :: Parser (a → b) → Parser a → Parser b
(⟨|) :: Parser a → Parser a → Parser a
match :: (Char → Bool)    → Parser Char

```

In other words, these are the operators of a small DSL for parsers. The intention is that: parser *success* *x* always succeeds, consumes no input, and returns *x*; *failure* always fails; *p*⟨*⟩*q* is a kind of sequential composition, matching according to *p* (yielding a function) and then on the remaining input to *q* (yielding an argument), and applying the function to the argument; *p*⟨|⟩*q* is a kind of choice, matching according to *p* or to *q*; and *match* *b* matches the single character at the head of the input, if this satisfies *b*, and fails if the input is empty or the head doesn't satisfy.

We can implement the DSL via a shallow embedding, such that the interpretation is the type *Parser a*. Each operator has a one- or two-line implementation:

```

success :: a → Parser a
success x s = [(x, s)]

failure :: Parser a
failure s = []

(⟨*) :: Parser (a → b) → Parser a → Parser b
(p ⟨*⟩ q) s = [(f a, s'') | (f, s') ← p s, (a, s'') ← q s']

(⟨|) :: Parser a → Parser a → Parser a
(p ⟨|⟩ q) s = p s ++ q s

match :: (Char → Bool) → Parser Char
match q [] = []
match q (c : s) = if q c then [(c, s)] else []

```

From the basic operators above, we can derive many more, without depending any further on the representation of parsers as functions. In each of the following exercises, the answer is another one- or two-liner.

Exercises

- Implement two variations of sequential composition, in which the first (respectively, the second) recognized value is discarded. These are useful when one of the recognized values is mere punctuation.

```

(*) :: Parser a → Parser b → Parser b
(⟨*⟩ :: Parser a → Parser b → Parser a

```

- Implement iteration of parsers, so-called Kleene plus (*some*) and Kleene star (*many*), which recognize one or more (respectively, zero or more) occurrences of what their argument recognizes.

```

some, many :: Parser a → Parser [a]

```

14. Implement a *whitespace* parser, which recognizes a nonempty section of whitespace characters (you might find the Haskell standard library function *Data.Char.isSpace* helpful). Implement a variation *ows* for which the whitespace is optional. For both of these, we suppose that the actual nature of the whitespace is irrelevant, and should be discarded.

whitespace, ows :: *Parser* ()

15. Implement a parser *token*, which takes a string and recognizes exactly and only that string at the start of the input. Again, we assume that the string so matched is irrelevant, since we know precisely what it will be.

token :: *String* → *Parser* ()

16. Now implement the parser *program* from Section 2.5, which recognizes a “set program”. A set program starts with the empty set {}, has zero or more insert (+) and delete (−) operations, and a mandatory final member (?) operation. Each operation is followed by an integer argument. Optional whitespace is allowed in all sensible places.

program :: *Parser* *Bool*

3.4 Lazy evaluation

A third aspect of modern functional programming that lends itself to embedded DSLs—albeit, perhaps, less important than algebraic datatypes and higher-order functions—is *lazy evaluation*. Evaluation is demand-driven, and function arguments are not evaluated until their value is needed to determine the next step (for example, to determine which of multiple clauses of a definition to apply); and moreover, once an argument is evaluated, that value is preserved and reused rather than being discarded and recomputed for subsequent uses.

One nice consequence of lazy evaluation is that infinite data structures work just as well as finite ones: as long as finite parts of the result of a function can be constructed from just finite parts of the input, the complete infinite data structure may not need ever to be constructed. This is sometimes convenient for a shallow embedding, allowing one to use a datatype of infinite data structures for the domain of interpretation. This can lead to simpler programs than would be the case if one were restricted to finite data structures—in the latter case, some terminating behaviour has to be interwoven with the generator, whereas in the former, the two can be quite separate. But we will not study infinite data structures further in this article.

A second consequence of lazy evaluation manifests itself even in finite data: if one component of a result is not used anywhere, it is not evaluated. This is very convenient for shallow embeddings of DSLs with multiple observers. The interpretation is then as a tuple containing all the observations of a term; but if some of those observations are not used, they need not be computed.

Exercises

17. Review Exercise 5, which was to define a shallow embedding interpreted as a pair, providing both evaluation and printing. Convince yourself that if only one component of the pair is demanded, only that component gets computed.
18. Here is an alternative technique for allowing for multiple observers with a shallow embedding. It is presented here using Haskell type classes; but the general idea is about having a data abstraction with an interface and a choice of implementations, and doing abstract interpretation in one of those implementations. For simplicity, let us return to Hutton's Razor

```
type Expr = ...
val  :: Integer      → Expr
add  :: Expr → Expr → Expr
```

with two desired observers

```
eval :: Expr → Integer
print :: Expr → String
```

The trick is to define *Expr* as a type class, the class of those types suitable as representations of expressions according to this little DSL. What operations must a type support, if it is to be suitable for representing expressions? It needs to have at least the *val* and *add* operations:

```
class Expr a where
  val  :: Integer → a
  add  :: a → a → a
```

Of course, it is easy to define these two operations on integers:

```
instance Expr Integer where
  val n    = n
  add x y  = x + y
```

It is also easy to define them on strings:

```
instance Expr String where
  val n    = show n
  add x y  = "(" ++ x ++ "+" ++ y ++ ")"
```

Now, a term in the expression DSL has a polymorphic type: it can be interpreted in *any* type in the type class *Expr*.

```
expr :: Expr a ⇒ a
expr = add (val 3) (val 4)
```

Then evaluating and printing expressions amounts to interpreting the polymorphic type at the appropriate instance:

```
eval_Expr :: Integer
eval_Expr = expr
```

```
print_Expr :: String
print_Expr = expr
```

Try this approach out. (You’ll find that you need some language extensions for the *String* instance, but the Haskell type checker should guide you in the right direction.) It’s a bit of an idiosyncratic way of implementing data abstraction: the implementation is chosen implicitly by fixing a type, rather than explicitly by passing a parameter. This is a slight problem, if you want two different interpretations on the same type, such as compact and verbose printings. What can you do to work around that?

4 An extended example: diagrams

We now turn to a larger example of an embedded DSL, inspired by Brent Yorgey’s **diagrams** project [4] for two-dimensional diagrams. That project implements a very powerful language which Yorgey doesn’t name, but we’ll call it *Diagrams*. But it’s also rather a large language, so we won’t attempt to cover the whole thing; instead, we build a much simpler language in the same style. The **diagrams** project does, however, provide a useful backend to output Scalable Vector Graphics (SVG) files, which we will borrow to save ourselves from having to reinvent one.

4.1 Shapes, styles, and pictures

The basics of our diagram DSL can be expressed in three simpler sublanguages, for shapes, styles, and pictures. We express them first via deep embedding. First, there are primitive shapes—as a language, these aren’t very interesting, because they aren’t recursive.

```
data Shape
  = Rectangle Double Double
  | Ellipse Double Double
  | Triangle Double
```

The parameters of a *Rectangle* specify its width and height; those of an *Ellipse* its x- and y-radii. A *Triangle* is equilateral, with its lowest edge parallel to the x-axis; the parameter is the length of the side.

Then there are drawing styles. A *StyleSheet* is a (possibly empty) sequence of stylings, each of which specifies fill colour, stroke colour, and stroke width. (The defaults are for no fill, and very thin black strokes.)

```
type StyleSheet = [Styling]
data Styling
  = FillColour Col
  | StrokeColour Col
  | StrokeWidth Double
```

Here, colours are defined in an external library, which among other things provides a large number of colour constants named according to the W3C SVG Recommendation [12, §4.4].

```
type Col = ...
  red, blue, green, yellow, brown, black ... :: Col
```

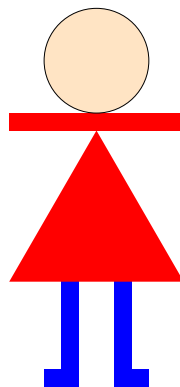
Finally, pictures are arrangements of shapes: individual shapes, with styling; or one picture above another, or one beside another. For simplicity, we specify that horizontal and vertical alignment of pictures is by their centres.

```
data Picture
  = Place StyleSheet Shape
  | Above Picture Picture
  | Beside Picture Picture
```

For example, here is a little stick figure of a woman in a red dress and blue stockings.

```
figure :: Picture
figure = Place [StrokeWidth 0.1, FillColour bisque] (Ellipse 3 3) 'Above'
  Place [FillColour red, StrokeWidth 0] (Rectangle 10 1) 'Above'
  Place [FillColour red, StrokeWidth 0] (Triangle 10) 'Above'
  (Place [FillColour blue, StrokeWidth 0] (Rectangle 1 5) 'Beside'
    Place [StrokeWidth 0] (Rectangle 2 5) 'Beside'
    Place [FillColour blue, StrokeWidth 0] (Rectangle 1 5)) 'Above'
  (Place [FillColour blue, StrokeWidth 0] (Rectangle 2 1) 'Beside'
    Place [StrokeWidth 0] (Rectangle 2 1) 'Beside'
    Place [FillColour blue, StrokeWidth 0] (Rectangle 2 1))
```

The intention is that it should be drawn like this:



(Note that blank spaces can be obtained by rectangles with zero stroke width.)

4.2 Transformations

In order to arrange pictures, we will need to be able to translate them. Later on, we will introduce some other transformations too; with that foresight in mind, we introduce a simple language of transformations—the identity transformation, translations, and compositions of these.

```
type Pos = Complex Double

data Transform
  = Identity
  | Translate Pos
  | Compose Transform Transform
```

For simplicity, we borrow the *Complex* type from the Haskell libraries to represent points in the plane; the point with coordinates (x, y) is represented by the complex number $x + iy$. *Complex* is an instance of the *Num* type class, so we get arithmetic operations on points too. For example, we can apply a *Transform* to a point:

```
transformPos :: Transform → Pos → Pos
transformPos Identity      = id
transformPos (Translate p) = (p+)
transformPos (Compose t u) = transformPos t ∘ transformPos u
```

Exercises

19. *Transform* is represented above via a deep embedding, with a separate observer function *transformPos*. Reimplement *Transform* via a shallow embedding, with this sole observer.

4.3 Simplified pictures

As it happens, we could easily translate the *Picture* language directly into *Diagrams*: it has equivalents of *Above* and *Beside*, for example. But if we were “executing” our pictures in a less sophisticated setting—for example, if we had to implement the SVG backend from first principles—we would eventually have to simplify recursively structured pictures into a flatter form.

Here, we flatten the hierarchy into a non-empty sequence of transformed styled shapes:

```
type Drawing = [(Transform, StyleSheet, Shape)]
```

In order to simplify alignment by centres, we will arrange that each simplified *Drawing* is itself centred: that is, the combined extent of all translated shapes will be centred on the origin. Extents are represented as pairs of points, for the lower left and upper right corners of the orthogonal bounding box.

```
type Extent = (Pos, Pos)
```

The crucial operation on extents is to compute their union:

$$\begin{aligned} \text{unionExtent} &:: \text{Extent} \rightarrow \text{Extent} \rightarrow \text{Extent} \\ \text{unionExtent } (llx_1 :+ lly_1, urx_1 :+ ury_1) (llx_2 :+ lly_2, urx_2 :+ ury_2) \\ &= (\min llx_1 llx_2 :+ \min lly_1 lly_2, \max urx_1 urx_2 :+ \max ury_1 ury_2) \end{aligned}$$

Now, the extent of a drawing is the union of the extents of each of its translated shapes, where the extent of a translated shape is the translation of the two corners of the extent of the untranslated shape:

$$\begin{aligned} \text{drawingExtent} &:: \text{Drawing} \rightarrow \text{Extent} \\ \text{drawingExtent} &= \text{foldr1 } \text{unionExtent} \circ \text{map } \text{getExtent} \textbf{ where} \\ \text{getExtent } (t, -, s) &= \textbf{let } (ll, ur) = \text{shapeExtent } s \\ &\quad \textbf{in } (\text{transformPos } t ll, \text{transformPos } t ur) \end{aligned}$$

(You might have thought initially that if all *Drawings* are kept centred, one point rather than two serves to define the extent. But this doesn't work: in computing the extent of a whole *Picture*, of course we have to translate its constituent *Drawings* off-centre.) The extents of individual shapes can be computed using a little geometry:

$$\begin{aligned} \text{shapeExtent} &:: \text{Shape} \rightarrow \text{Extent} \\ \text{shapeExtent } (\text{Ellipse } xr \ yr) &= (-(xr :+ yr), xr :+ yr) \\ \text{shapeExtent } (\text{Rectangle } w \ h) &= (-(w/2 :+ h/2), w/2 :+ h/2) \\ \text{shapeExtent } (\text{Triangle } s) &= (-(s/2 :+ \sqrt{3} \times s/4), s/2 :+ \sqrt{3} \times s/4) \end{aligned}$$

Now to simplify *Pictures* into *Drawings*, via a straightforward traversal over the structure of the *Picture*.

$$\begin{aligned} \text{drawPicture} &:: \text{Picture} \rightarrow \text{Drawing} \\ \text{drawPicture } (\text{Place } u \ s) &= \text{drawShape } u \ s \\ \text{drawPicture } (\text{Above } p \ q) &= \text{drawPicture } p \text{ 'aboveD' } \text{drawPicture } q \\ \text{drawPicture } (\text{Beside } p \ q) &= \text{drawPicture } p \text{ 'besideD' } \text{drawPicture } q \end{aligned}$$

All the work is in the individual operations. *drawShape* constructs an atomic styled *Drawing*, centred on the origin.

$$\begin{aligned} \text{drawShape} &:: \text{StyleSheet} \rightarrow \text{Shape} \rightarrow \text{Drawing} \\ \text{drawShape } u \ s &= [(\text{Identity}, u, s)] \end{aligned}$$

aboveD and *besideD* both work by forming the “union” of the two child *Drawings*, but first translating each child by the appropriate amount—an amount calculated so as to ensure that the resulting *Drawing* is again centred on the origin.

$$\begin{aligned} \text{aboveD}, \text{besideD} &:: \text{Drawing} \rightarrow \text{Drawing} \rightarrow \text{Drawing} \\ \text{pd 'aboveD' qd} &= \text{transformDrawing } (\text{Translate } (0 :+ qury)) \text{ pd} \text{ ++} \\ &\quad \text{transformDrawing } (\text{Translate } (0 :+ plly)) \text{ qd} \textbf{ where} \\ &\quad (pllx :+ plly, pur) = \text{drawingExtent } \text{pd} \\ &\quad (qll, qurx :+ qury) = \text{drawingExtent } \text{qd} \\ \text{pd 'besideD' qd} &= \text{transformDrawing } (\text{Translate } (qllx :+ 0)) \text{ pd} \text{ ++} \\ &\quad \text{transformDrawing } (\text{Translate } (purx :+ 0)) \text{ qd} \textbf{ where} \end{aligned}$$

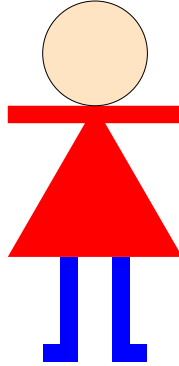
$$\begin{aligned} (pll, purx :+ pury) &= \text{drawingExtent } pd \\ (qlx :+ qlly, qur) &= \text{drawingExtent } qd \end{aligned}$$

This involves transforming the child *Drawings*; but that's easy, given our representation.

$$\begin{aligned} \text{transformDrawing} &:: \text{Transform} \rightarrow \text{Drawing} \rightarrow \text{Drawing} \\ \text{transformDrawing } t &= \text{map } (\lambda(t', u, s) \rightarrow (\text{Compose } t \, t', u, s)) \end{aligned}$$

Exercises

20. Add *Square* and *Circle* to the available *Shapes*; for simplicity, you can implement these using *rect* and *ellipseXY*.
21. Add *Blank* to the available shapes; implement this as a rectangle with zero stroke width.
22. Centring and alignment, as described above, are only approximations, because we don't take stroke width into account. How would you do so?
23. Add *InFrontOf* :: *Picture* → *Picture* → *Picture* as an operator to the *Picture* language, for placing one *Picture* in front of (that is, on top of) another. Using this, you can draw a slightly less childish-looking stick figure, with the “arms” overlaid on the “body”:



24. Add *FlipV* :: *Picture* → *Picture* as an operator to the *Picture* language, for flipping a *Picture* vertically (that is, from top to bottom, about a horizontal axis). Then you can draw this chicken:



You'll need to add a corresponding operator *ReflectY* to the *Transform* language; you might note that the *conjugate* function on complex numbers takes $x \div y$ to $x \div (-y)$. Be careful in computing the extent of a flipped picture!

25. *Picture* is represented above via a deep embedding, with a separate observer function *drawPicture*. Reimplement *Picture* via a shallow embedding, with this sole observer.

4.4 Generating SVG

The final step is to assemble our simplified *Drawing* into some expression in the *Diagrams* language. What we need are the following:

- A type for representing diagrams.

type *DiagramSVG* = ...

(This is actually a synonym for a specialization of a more flexible *Diagrams* type from Yorgey's library.)

- Primitives of type *DiagramSVG*:

rect :: *Double* → *Double* → *DiagramSVG*
ellipseXY :: *Double* → *Double* → *DiagramSVG*
eqTriangle :: *Double* → *DiagramSVG*

- An operator for superimposing diagrams:

atop :: *DiagramSVG* → *DiagramSVG* → *DiagramSVG*

- Transformations on diagrams:

translate ∘ *r2* :: (*Double*, *Double*) → *DiagramSVG* → *DiagramSVG*
reflectY :: *DiagramSVG* → *DiagramSVG*

(The latter is needed for Exercise 24.)

- Functions for setting fill colour, stroke colour, and stroke width attributes:

fc :: *Col* → *DiagramSVG* → *DiagramSVG*
lc :: *Col* → *DiagramSVG* → *DiagramSVG*
lw :: *Double* → *DiagramSVG* → *DiagramSVG*

- A wrapper function that writes a diagram out in SVG format to a specified file:

writeSVG :: *FilePath* → *DiagramSVG* → *IO* ()

Then a *Drawing* can be assembled into a *DiagramSVG* by laying one translated styled shape on top of another:

assemble :: *Drawing* → *DiagramSVG*
assemble = *foldr1* *atop* ∘ *map draw* **where**
draw (*t*, *u*, *s*) = *transformDiagram* *t* (*diagramShape* *u* *s*)

Note that *Shapes* earlier in the list appear “in front” of those later; you’ll need to use this fact in solving Exercise 23.

A *StyleSheet* represents a sequence of functions, which are composed into one styling function:

```

applyStyleSheet :: StyleSheet → (DiagramSVG → DiagramSVG)
applyStyleSheet = foldr (◦) id ◦ map applyStyling

applyStyling :: Styling → DiagramSVG → DiagramSVG
applyStyling (FillColor c)    = fc c
applyStyling (StrokeColour c) = lc c
applyStyling (StrokeWidth w) = lw w

```

A single styled shape is drawn by applying the styling function to the corresponding atomic diagram:

```

diagramShape :: StyleSheet → Shape → DiagramSVG
diagramShape u s = shape (applyStyleSheet u) s where
  shape f (Ellipse xr yr) = f (ellipseXY xr yr)
  shape f (Rectangle w h) = f (rect w h)
  shape f (Triangle s)    = f (translate (r2 (0, -y)) (eqTriangle s))
                        where y = s × √3/12

```

(The odd translation of the triangle is because we place triangles by their centre, but *Diagrams* places them by their centroid.)

A transformed shape is drawn by transforming the diagram of the underlying shape.

```

transformDiagram :: Transform → DiagramSVG → DiagramSVG
transformDiagram Identity      = id
transformDiagram (Translate (x :+ y)) = translate (r2 (x, y))
transformDiagram (Compose t u)  = transformDiagram t ◦
                                transformDiagram u

```

And that’s it! (You can look at the source file **Shapes.lhs** for the definition of *writeSVG*, and some other details.)

Exercises

26. In Exercise 19, we reimplemented *Transform* as a shallow embedding, with the sole observer being to transform a point. This doesn’t allow us to apply the same transformations to *DiagramSVG* objects, as required by the function *transformDiagram* above. Extend the shallow embedding of *Transform* so that it has two observers, for transforming both points and diagrams.
27. A better solution to Exercise 26 would be to represent *Transform* via a shallow embedding with a single parametrized observer, which can be instantiated at least to the two uses we require. What are the requirements on such instantiations?

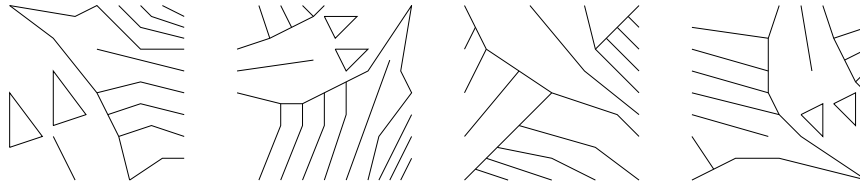
28. Simplifying a *Picture* into a *Drawing* is a bit inefficient, because we have to continually recompute extents. A more efficient approach would be to extend the *Drawing* type so that it caches the extent, as well as storing the list of shapes. Try this.
29. It can be a bit painful to specify a complicated *Picture* with lots of *Shapes* all drawn in a common style—for example, all blue, with a thick black stroke—because those style settings have to be repeated for every single *Shape*. Extend the *Picture* language so that *Pictures* too may have *StyleSheets*; styles should be inherited by children, unless they are overridden.
30. Add an operator *Tile* to the *Shape* language, for square tiles with markings on. It should take a *Double* parameter for the length of the side, and a list of lists of points for the markings; each list of points has length at least two, and denotes a path of straight-line segments between those points. For example, here is one such pattern of markings:

```

markingsP :: [[Pos]]
markingsP = [(4 :+ 4), (6 :+ 0)],
              [(0 :+ 3), (3 :+ 4), (0 :+ 8), (0 :+ 3)],
              [(4 :+ 5), (7 :+ 6), (4 :+ 10), (4 :+ 5)],
              [(11 :+ 0), (10 :+ 4), (8 :+ 8), (4 :+ 13), (0 :+ 16)],
              [(11 :+ 0), (14 :+ 2), (16 :+ 2)],
              [(10 :+ 4), (13 :+ 5), (16 :+ 4)],
              [(9 :+ 6), (12 :+ 7), (16 :+ 6)],
              [(8 :+ 8), (12 :+ 9), (16 :+ 8)],
              [(8 :+ 12), (16 :+ 10)],
              [(0 :+ 16), (6 :+ 15), (8 :+ 16), (12 :+ 12), (16 :+ 12)],
              [(10 :+ 16), (12 :+ 14), (16 :+ 13)],
              [(12 :+ 16), (13 :+ 15), (16 :+ 14)],
              [(14 :+ 16), (16 :+ 15)]
            ]

```

In `Shapes.lhs`, you'll find this definition plus three others like it. They yield tile markings looking like this:



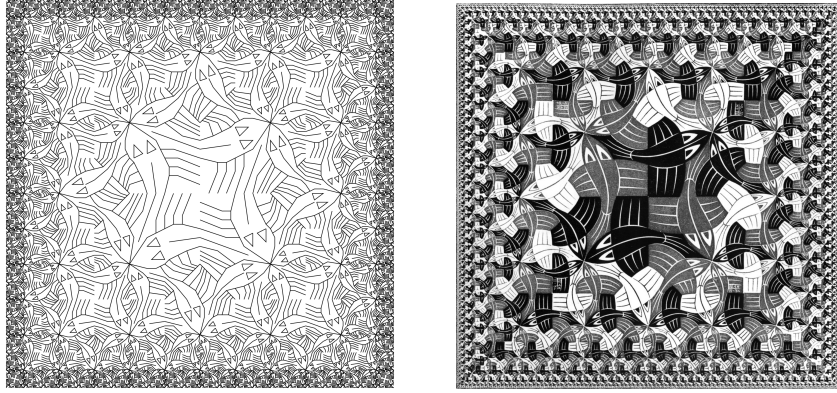
You can draw such tiles via the function

```
tile :: [[Pos]] → DiagramSVG
```

provided for you. Also add operators to the *Picture* and *Transform* languages to support scaling by a constant factor and rotation by a quarter-turn anticlockwise, both centred on the origin. You can implement these on the *DiagramSVG* type using two *Diagrams* operators:

$$\begin{aligned} \text{scale} &:: \text{Double} \rightarrow \text{DiagramSVG} \rightarrow \text{DiagramSVG} \\ \text{rotateBy } (1/4) &:: \text{DiagramSVG} \rightarrow \text{DiagramSVG} \end{aligned}$$

Then suitable placements, rotations, and scalings of the four marked tiles will produce a rough version of Escher’s “Square Limit” print, as shown in the left-hand image below:



This construction was explored by Peter Henderson in a famous early paper on functional geometry [13, 14]; I have taken the data for the markings from a note by Frank Buß [15]. The image on the right is taken from WikiPaintings [16].

31. Morally, “Square Limit” is a fractal image: the recursive decomposition can be taken ad infinitum. Because Haskell uses lazy evaluation, that’s not an insurmountable obstacle. The datatype *Picture* includes also infinite terms; and because *Diagrams* is an embedded DSL, you can use a recursive Haskell definition to define an infinite *Picture*. You can’t render it directly to SVG, though; that would at best yield an infinite SVG file. But still, you can prune the infinite picture to a finite depth, and then render the result. Construct the infinite *Picture*. (You’ll probably need to refactor some code. Note that you can’t compute the extent of an infinite *Picture* either.)

References

1. Fowler, M.: Domain-Specific Languages. Addison-Wesley (2011)
2. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. *ACM Computing Surveys* **37**(4) (2005) 316–344
3. Bentley, J.: Little languages. *Communications of the ACM* **29**(8) (1986) 711–721 Also in ‘More Programming Pearls’ (Addison-Wesley, 1988).
4. Yorgey, B.: Diagrams 0.6. <http://projects.haskell.org/diagrams/> (2012)
5. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. *Communications of the ACM* **15**(12) (1972) 1053–1058
6. Wand, M.: Final algebra semantics and data type extensions. *Journal of Computer and System Sciences* **19** (1979) 27–44

7. Kamin, S.: An implementation-oriented semantics of Wadler's pretty-printing combinators. Oregon Graduate Institute, <http://www-sal.cs.uiuc.edu/~kamin/pubs/pprint.ps> (1998)
8. Erwig, M., Walkingshaw, E.: Semantics-driven DSL design. In Mernik, M., ed.: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments. IGI-Global (2012) 56–80
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
10. Wadler, P.L.: The expression problem. Posting to java-genericity mailing list (1998)
11. Hutton, G.: Fold and unfold for program semantics. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, Baltimore, Maryland (1998) 280–288
12. W3C: Scalable vector graphics (SVG) 1.1: Recognized color keyword names. <http://www.w3.org/TR/SVG11/types.html#ColorKeywords> (2011)
13. Henderson, P.: Functional geometry. In: Lisp and Functional Programming. (1982) 179–187 <http://users.ecs.soton.ac.uk/ph/funcgeo.pdf>.
14. Henderson, P.: Functional geometry. Higher Order and Symbolic Computing **15**(4) (2002) 349–365 Revision of [13].
15. Buß, F.: Functional geometry. <http://www.frank-buss.de/lisp/functional.html> (2005)
16. Escher, M.C.: Square limit. <http://www.wikipaintings.org/en/m-c-escher/square-limit> (1964)