Script: Functional computations

Headline

Abstraction mechanisms for functional computations

Description

Applications of pure functions return the same result whenever provided with the same arguments; they do not have any side effects. This may be viewed as a limitation when we need to model more general computations in functional programming. For instance, it may be more difficult to define functions that essentially manipulate some state or process some input.

However, there is a functional programming abstraction, the monad, which comes to rescue. A monad is essentially an abstract data type to facilitate the composition of computations as opposed to functions. There are various monads to deal with all the computational effects that one may encounter, e.g., the State monad, the Maybe monad, the Reader monad, the Writer monad, and the IO monad.

The aforementioned monads all concern relatively general effects. However, monadic style (in the sense of abstract data types over computations) is also useful in more domain-specific contexts. This is illustrated for the domain of parsing, i.e., analysing text according to a given grammar (syntax definition) and mapping the text to an appropriate tree-like structure, i.e., a parse tree (or syntax tree).

In modern Haskell, monads "compete" with applicative functors. "In functional programming, an applicative functor, or an applicative for short, is an intermediate structure between functors and monads. Applicative functors allow for functorial computations to be sequenced (unlike plain functors), but don't allow using results from prior computations in the definition of subsequent ones (unlike monads)." Wikipedia (Applicative Functor), 2023-07-09 We will also introduce applicatives. In fact, applicatives are arguably easier to grasp than monads, especially if we assume an understanding of functors. So we might as well start the discussion with applicatives and then proceed to monads.

Concepts

- Applicative functor
- Monad
  - State monad
  - Maybe monad
  - Writer monad
- Parsing

Languages

- Language:Haskell

Technologies

- Technology:Parsec

Features

- Feature:Logging
- Feature:Parsing

Contributions

- Contribution:haskellLogging
- Contribution:haskellWriter
- Contribution:haskellAcceptor
- Contribution:haskellParsec

Concept: Parse tree

Headline

Another term for syntax tree

Illustration

See the illustration of syntax tree.

Concept: Syntax tree

Headline

A tree representing the grammatical structure of text

Description

We explain the concept by means of an illustrative example.

Illustration

See the illustrations of abstract and concrete syntax for simple intuitions.

A more profound illustration follows.

Consider the following context-free grammar; we label the productions for convenience:

```
[literal] expression ::= literal
[binary] expression ::= "(" expression op expression ")"
[plus] op ::= "+"
[times] op ::= "*"
```

We assume that literals are integers as in sequences of digits.

Now consider this input:

```
((4 * 10) + 2)
```

A parsing algorithm would basically accept the input string, as it is an element of the language generated by the grammar; see also the parsing problem. In addition, an actual parser would also represent the grammar-based structure of the input string by a parse tree. We represent the parse tree for the input string here as a prefix term such that we use production labels as function symbols:

```
binary(
  "(",
  binary(
    "(",
    literal("4"),
    times("*"),
    literal("10"),
    ")",
  ),
  plus("+"),
  literal("2"),
  ")",
)
```

That is, each non-leaf node in the tree corresponds to a production label and each leaf node is a terminal. Further, each non-leaf node has as many branches as the underlying production has symbols in the right-hand side sequence. We assume that the branches are ordered in the same way as the underlying right-hand side and the subtrees are parse trees for the symbols in the right-hand side. A parse tree for a given nonterminal is rooted by a production with the nonterminal on the left-hand side.

The parse tree shown above is a concrete syntax tree in that it captures even the terminals of the derivation. We may also remove those symbols to arrive at an abstract syntax tree. Thus:

```
binary(
  binary(
    literal("4"),
    times,
    literal("10")
  ),
  plus,
  literal("2")
)
```

Concept: Monad

Headline

A functional programming idiom for computing effects

Illustration

The term "monad" originates from category theory, but this illustration focuses on the functional programming view where "monad" refers to a programming idiom for composing computations, specifically computations that may involve side effects or I/O actions. Monads have been popularized by Language:Haskell.

In Haskell, monads are developed and used with the help of the type class *Monad* which is parametrized by a type constructor for the actual monad. Here is a sketch of the type class:

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  -- ... some details omitted
```

The *return* function serves the construction of trivial computations, i.e., computations that return values. The >>= (also knows as the bind function) compose a computation with a function that consumes the value of said computation to produce a composed computation. Here are some informal descriptions of popular monads:

- State monad
  - *return* v: return value *v* and pass on state
  - *bind* c f: apply computation *c* as state transformer and pass on transformed state to *f*
- Reader monad
  - *return* v: return value *v* and ignore environment
  - *bind* c f: pass environment to both *c* and *f*
- Writer monad
  - *return* v: return value *v* and empty output
  - *bind* c f: compose output from both *c* and *f*
- Maybe monad
  - *return* v: return "successful" value *v*
  - *bind* c f: fail if *c* fails, otherwise, pass on successful result to *f*

Concept: Abstract data type

Headline

A data type that does not reveal representation

Illustration

An abstract data type is usually just defined through the list of operations on the type, possibly enirched (formaly or informaly) by properties (invariants, pre- and postconditions).

Consider the following concrete data type for points:

```
data Point = Point { getX :: Int, getY :: Int }
  deriving (Eq, Show, Read)
```

Now suppose we want to hide the precise representation of points. In particular, we want to rule out that programmers can match and apply the constructor *Point*. The existing getters are sufficient to observe points without matching, but we need to provide some "public" means of constructing points.

```
mkPoint :: Int -> Int -> Point
mkPoint = Point
```

The idea is now to export *mkPoint*, but not the constructor, thereby making possible representation changes without changing any code that uses points. This is, of course, a trivial example, as the existing representation of points is probably quite appropriate, but see a more advanced illustration for an abstract data type Stack.

A complete module for an abstract data type for points may then look like this:

```
module Point(
  Point, -- constructor is NOT exported
  mkPoint,
  getX,
  getY
) where

data Point = Point { getX :: Int, getY :: Int }
  deriving (Eq, Show, Read)

mkPoint :: Int -> Int -> Point
mkPoint = Point
```

When defining an abstract data type, we take indeed the point of view that the representation and thus the implementation as such is not known or not to be looked at. Hence, ideally, the intended functionality should be described in some other way. For instance, we may describe the functionality by properties. For instance, in Haskell we may declare testable Technology:QuickCheck properties like this:

```
prop_getX :: Int -> Int -> Bool
prop_getX x y = getX (mkPoint x y) == x

prop_getY :: Int -> Int -> Bool
prop_getY x y = getY (mkPoint x y) == y
```

These properties describe the (trivial) correspondence between construction with *mkPoint* and observation with *getX* and *getY*. Logically, the first property says that for all given *x* and *y*, we can construct a point and we can retrieve *x* again from that point with *getX*.

Relationships

- An abstract data type is the opposite of a concrete data type.
- An abstract data type performs information hiding.

Concept: Writer monad

Headline

A monad for synthesizing results or output

Illustration

Let us put to work the writer monad in a simple interpreter.

A baseline interpreter

There are these expression forms:

```
-- Simple Boolean expressions
data Expr = Constant Bool | And Expr Expr | Or Expr Expr
  deriving (Eq, Show, Read)
```

For instance, the following expression should evaluate to true:

```
-- A sample term with two operations
sample = And (Constant True) (Or (Constant False) (Constant True))
```

Here is a simple interpreter, indeed:

```
-- A straightforward interpreter
eval :: Expr -> Bool
eval (Constant b) = b
eval (And e1 e2) = eval e1 && eval e2
eval (Or e1 e2) = eval e1 || eval e2
```

Adding counting to the interpreter

Now suppose that the interpreter should also return the number of operations applied. We count *And* and *Or*as operations. Thus, the sample term should count as 2. We may incorporate counting into the initial interpreter as follows:

```
-- Interpreter with counting operations
eval' :: Expr -> (Bool, Int)
eval' (Constant b) = (b, 0)
eval' (And e1 e2) =
  let
    (b1,i) = eval' e1
    (b2,i') = eval' e2
  in (b1 && b2, i+i'+1)
eval' (Or e1 e2) =
  let
    (b1,i) = eval' e1
    (b2,i') = eval' e2
  in (b1 || b2, i+i'+1)
```

Alas, the resulting interpreter is harder to understand. The collection of counts is entangled with the basic logic.

Monadic style

By conversion to monadic style, we can hide counting except when we need increment the counter. We use the Writer monad here so that we simply combine counts from subexpression (as also done in the non-monadic code above). We could also be using the state monad, if we wanted to really track the operations counter along evaluation; this would be useful if we were adding an expression form for retrieving the count.

```
evalM :: Expr -> Writer (Sum Int) Bool
evalM (Constant b) = return b
evalM (And e1 e2) =
  evalM e1 >>= \b1 ->
  evalM e2 >>= \b2 ->
  tell (Sum 1) >>
  return (b1 && b2)
```

```
evalM (Or e1 e2) =
  evalM e1 >>= \b1 ->
  evalM e2 >>= \b2 ->
  tell (Sum 1) >>
  return (b1 || b2)
```

We can also use do notation:

```
-- Monadic style interpreter in do notation
evalM :: Expr -> Writer (Sum Int) Bool
evalM (Constant b) = return b
evalM (And e1 e2) = do
  b1 <- evalM e1
  b2 <- evalM e2
  tell (Sum 1)
  return (b1 && b2)
evalM (Or e1 e2) = do
  b1 <- evalM e1
  b2 <- evalM e2
  tell (Sum 1)
  return (b1 || b2)
```

The Writer monad

The Writer monad is readily provided by the Haskell library (in Control.Monad.Trans.Writer.Lazy), but we may want to understand how it might have been implemented. The data type for the Writer monad could look like this:

```
-- Computations as pairs of value and "output"
newtype Writer w a = Writer { runWriter :: (a, w) }
```

Thus, a stateful computation is basically a function a value with some output. The output type is assumed to be monoid because an empty output and the combination of outputs is uniformly defined in this manner.

The corresponding instance of the type class *Monad* follows:

```
-- Monad instance for Writer
instance Monoid w => Monad (Writer w)
  where
    return a = Writer (a, mempty)
    (Writer (a, w)) >>= f =
      let (Writer (b, w')) = f a in
        (Writer (b, w `mappend` w'))
```

The definition of *return* conveys that a pure computation produces the empty output. The definition of *bind* conveys that outputs are to be combined (in a certain order) from the operands of *bind*. Finally, we need to define the writer-specific operation *tell* for producing ouput:

```
-- Produce output
tell :: w -> Writer w ()
tell w = Writer ((), w)
```

In modern Haskell, we also need to make *Writer* an instance of *Applicative* (for applicative functors and *Functor* (for functors). This code is omitted here, but see GitHub for this page.

See Contribution:haskellWriter for a contribution which uses the writer monad.

Concept: State monad

Headline

A monad for state

Illustration

Let us put to work the State monad in a simple interpreter.

A baseline interpreter

There are these expression forms:

```
-- Simple Boolean expressions
data Expr = Constant Bool | And Expr Expr | Or Expr Expr
  deriving (Eq, Show, Read)
```

For instance, the following expression should evaluate to true:

```
sample = And (Constant True) (Or (Constant False) (Constant True))
```

Here is a simple interpreter, indeed:

```
-- A straightforward interpreter
eval :: Expr -> Bool
eval (Constant b) = b
eval (And e1 e2) = eval e1 && eval e2
eval (Or e1 e2) = eval e1 || eval e2
```

Adding counting to the interpreter

Now suppose that the interpreter should keep track of the number of operations applied. We count *And* and *Or* as operations. Thus, the sample term should count as 2. We may incorporate counting into the initial interpreter by essentially maintaing an extra result component for the counter. (We could also synthesize the count as output; see the illustration of the writer monad.)

To make the example a bit more interesting, let's add a construct `Hide`. The expectation is that operations under Hide are not counted.

```
data Expr
  = Constant Bool
  | And Expr Expr
  | Or Expr Expr
  | Hide Expr
```

Because of the use of `Hide`, we would count 2 instead of 3 operations in the following term:

```
sample =
  And
    (Constant True)
    (Or
      (Constant True)
      (Hide (And
        (Constant False)
        (Constant False))))
```

Here is the interpreter which incorporates counting:

```
-- Interpreter with counting operations
eval' :: Expr -> Int -> (Bool, Int)
eval' (Constant b) i = (b, i)
eval' (And e1 e2) i =
  let
    (b1,i') = eval' e1 i
    (b2,i'') = eval' e2 i'
  in (b1 && b2, i''+1)
eval' (Or e1 e2) i =
  let
```

```
    (b1,i') = eval' e1 i
    (b2,i'') = eval' e2 i'
  in (b1 || b2, i''+1)
eval' (Hide e) i = (fst (eval' e i), i)
```

Alas, the resulting interpreter is harder to understand. The threading of counts is entangled with the basic logic.

Monadic style

By conversion to monadic style, we can hide counting except when we need increment the counter. We use the State monad here so that we really track the operations counter along evaluation; this would be useful if we were adding an expression form for retrieving the count. We could also be using the writer monad, if we were only interested in the final count.

```
-- Monadic style interpreter
evalM :: Expr -> State Int Bool
evalM (Constant b) = return b
evalM (And e1 e2) =
  evalM e1 >>= \b1 ->
  evalM e2 >>= \b2 ->
  modify (+1) >>
  return (b1 && b2)
evalM (Or e1 e2) =
  evalM e1 >>= \b1 ->
  evalM e2 >>= \b2 ->
  modify (+1) >>
  return (b1 || b2)
evalM (Hide e) =
  get >>= \i ->
  evalM e >>= \b ->
  put i >>= \() ->
  return b
```

We can also use do notation:

```
-- Monadic style interpreter in do notation
evalM' :: Expr -> State Int Bool
evalM' (Constant b) = return b
evalM' (And e1 e2) = do
  b1 <- evalM' e1
  b2 <- evalM' e2
  modify (+1)
  return (b1 && b2)
evalM' (Or e1 e2) = do
  b1 <- evalM' e1
  b2 <- evalM' e2
  modify (+1)
  return (b1 || b2)
evalM' (Hide e) = do
  i <- get
  b <- evalM e
  put i
  return b
```

The State monad

The state monad is readily provided by the Haskell library (in Control.Monad.State.Lazy), but we may want to understand how it might have been implemented. The data type for the State monad could look like this:

```
-- Data type for the State monad
newtype State s a = State { runState :: s -> (a,s) }
```

Thus, a stateful computation is basically a function on state which also returns a value.

The corresponding instance of the type class *Monad* follows:

```
-- Monad instance for State
instance Monad (State s)
  where
    return x = State (\s -> (x, s))
    c >>= f = State (\s -> let (x,s') = runState c s in runState (f x) s')
```

The definition of *return* conveys that a pure computation preserves the state. The definition of *bind* conveys that the state is to be threaded from the first argument to the second. Finally, we need to define state-

specific operations:

```
-- Important State operations: get/put state
get :: State s s
get = State (\s -> (s, s))

put :: s -> State s ()
put s = State (\_ -> ((), s))

-- Composition of get and put
modify :: (s -> s) -> State s ()
modify f = do { x <- get; put (f x) }
```

In modern Haskell, we also need to make *State* an instance of *Applicative* (for applicative functors and *Functor* (for functors). This code is omitted here, but see GitHub for this page.

Contribution: haskellLogging

Headline

Logging in Haskell with non-monadic code

Characteristics

Starting from a straightforward family of functions for cutting salaries, the concern of logging the salary changes is incorporated into the functions such that the function results are enriched by the log entries for salary changes. This code is relatively verbose and implies poor abstraction. In particular, functionality for composing logs is scattered all over the functions. Ultimately, such a problem must be addressed with monads.

Illustration

Salary changes can be tracked in logs as follows:

```
type Log = [LogEntry]

data LogEntry =
    LogEntry {
      name :: String,
      oldSalary :: Float,
      newSalary :: Float
    }
 deriving (Show)
```

Here are a few entries resulting from a salary cut for the sample company:

```
[LogEntry {name = "Craig", oldSalary = 123456.0, newSalary = 61728.0},
 LogEntry {name = "Erik", oldSalary = 12345.0, newSalary = 6172.5},
 LogEntry {name = "Ralf", oldSalary = 1234.0, newSalary = 617.0},
 LogEntry {name = "Ray", oldSalary = 234567.0, newSalary = 117283.5},
 LogEntry {name = "Klaus", oldSalary = 23456.0, newSalary = 11728.0},
 LogEntry {name = "Karl", oldSalary = 2345.0, newSalary = 1172.5},
 LogEntry {name = "Joe", oldSalary = 2344.0, newSalary = 1172.0}]
```

Given a log, the median of salary deltas can be computed as follows:

```
log2median :: Log -> Float
log2median = median . log2deltas

log2deltas :: Log -> [Float]
log2deltas = sort . map delta
  where
    delta entry = newSalary entry - oldSalary entry
```

The above log reduces to the following median:

```
-6172.5
```

Feature:Cut is implemented in logging-enabled fashion as follows:

```
cut :: Company -> (Company, Log)
cut (Company n ds) = (Company n ds', log)
  where
   (ds', logs) = unzip (map cutD ds)
   log = concat logs
   cutD :: Department -> (Department, Log)
   cutD (Department n m ds es)
     = (Department n m' ds' es', log)
     where
       (m',log1) = cutE m
       (ds', logs2) = unzip (map cutD ds)
       (es', logs3) = unzip (map cutE es)
       log = concat ([log1]++logs2++logs3)
       cutE :: Employee -> (Employee, Log)
       cutE (Employee n a s) = (e', log)
```

```
where
  e' = Employee n a s'
  s' = s/2
  log = [ LogEntry {
            name = n,
            oldSalary = s,
            newSalary = s'
          } ]
```

Thus, all functions return a regular data item (i.e., some part of the company) and a corresponding log. When lists of company parts are processed with map, then the lists of results must be unzipped (to go from a list of pairs to a pair of lists). In the function for departments, multiple logs arise for parts a department; these intermediate logs must be composed.

Relationships

- See Contribution:haskellComposition for the corresponding contribution that does not yet involve logging. The data model is preserved in the present contribution, but the functions for cutting salaries had to be rewritten since the logging concern crosscuts the function.
- See Contribution:haskellWriter for a variation on the present contribution, which leverages a writer monad, though, for conciseness and proper abstraction.

Architecture

There are these Haskell modules:

- Company.hs: the data model reused from Contribution:haskellComposition.
- Cut.hs: the combined implementation of Feature:Cut and Feature:Logging.
- Log.hs: types and functions for logs of salary changes needed for Feature:Logging.
- Main.hs: demonstration of all functions.

The contribution relies on the hackage package hstats.

Contribution: haskellWriter

Headline

Logging in Haskell with the Writer monad

Characteristics

Salary changes are logged in a Language:Haskell-based implementation with the help of a writer monad. Compared to a non-monadic implementation, the code is more concise. Details of logging are localized such that they only surface in the context of code that actually changes salaries.

Illustration

See Contribution:haskellLogging for a simpler, non-monadic implementation.

The present, monadic implementation differs only with regard to the cut function:

```
cut :: Company -> Writer Log Company
cut (Company n ds) =
  do
    ds' <- mapM cutD ds
    return (Company n ds')
  where
    cutD :: Department -> Writer Log Department
    cutD (Department n m ds es) =
      do
        m' <- cutE m
        ds' <- mapM cutD ds
        es' <- mapM cutE es
        return (Department n m' ds' es')
      where
        cutE :: Employee -> Writer Log Employee
        cutE (Employee n a s) =
          do
            let s' = s/2
            let log = [ LogEntry {
                          name = n,
                          oldSalary = s,
                          newSalary = s'
                        } ]
            tell log
            return (Employee n a s')
```

Thus, the family of functions uses a writer monad in the result types. The sub-traversals are all composed by monadic bind (possibly expressed in do-notation). The function for processing departments totally abstracts from the fact that logging is involved. In fact, that function could be defined to be parametrically polymorphic in the monad at hand.

Relationships

- See Contribution:haskellComposition for the corresponding contribution that does not yet involve logging. The data model is preserved in the present contribution, but the functions for cutting salaries had to be rewritten since the logging concern crosscuts the function.
- See Contribution:haskellLogging for a variation on the present contribution which does not yet use monadic style.

Architecture

See Contribution:haskellLogging.

Feature: Logging

Headline

Log and analyze salary changes

Description

Salaries of employees may change over time. For instance, a salary cut systematically decreases salaries. Of course, a pay raise could also happen; point-wise salary changes are conceivable as well. Salary changes are to be logged so that they can be analyzed within some window of interest. Specifically, a salary cut is to be logged with names of affected employees, salary before the change, and salary after the change. The log is to be analyzed in a statistical manner to determine the median and the mean of all salary deltas.

Motivation

The feature requires logging of updates to employee salaries. Depending on the programming language at hand, such logging may necessitate revision of the code that changes salaries. Specifically, logging of salary changes according to a salary cut may necessitate adaptation of the actual transformation for cutting salaries. Logging should be preferably added to a system while obeying separation of concerns. So logging is potentially a crosscutting concern, which may end being implemented in a scattered manner, unless some strong means of modularization can be adopted.

Illustration

The log for salary cut for the "standard" sample company would look as follows.

```
[ LogEntry {name = "Craig", oldSalary = 123456.0, newSalary = 61728.0},
  LogEntry {name = "Erik", oldSalary = 12345.0, newSalary = 6172.5},
  LogEntry {name = "Ralf", oldSalary = 1234.0, newSalary = 617.0},
  LogEntry {name = "Ray", oldSalary = 234567.0, newSalary = 117283.5},
  LogEntry {name = "Klaus", oldSalary = 23456.0, newSalary = 11728.0},
  LogEntry {name = "Karl", oldSalary = 2345.0, newSalary = 1172.5},
  LogEntry {name = "Joe", oldSalary = 2344.0, newSalary = 1172.0}
]
```

For what it matters, the salary cut operates as a depth-first, left-to-right traversal of the company; thus the order of the entries in the log. Projection of changes to deltas and sorting them results in the following list of deltas:

```
[ -117283.5,
  -61728.0,
  -11728.0,
  -6172.5,
  -1172.5,
  -1172.0,
  -617.0
]
```

Clearly, the median is the element in the middle:

```
-6172.5
```

By contrast, the mean is much different because of the skewed distribution of salaries:

```
-28553.355
```

See Contribution:haskellLogging for a simple implementation of the feature in Language:Haskell.

Relationships

- The present feature builds on top of Feature:Cut, as it is required to demonstrate the analysis of logged deltas for the transformation of a salary cut.
- The present feature should be applicable to any data model of companies, specifically Feature:Flat company and Feature:Hierarchical company.

Guidelines

- The *name* of the type for logs should involve the term "log".
- A suitable *demonstration* of the feature's implementation should cut the sample company and compute the median of the salary deltas, as indeed stipulated above.

Feature: Parsing

Headline

Parse an external format for companies

Description

Users of the system:Company may need to exchange data with other systems or edit data independently of the system. To this end, some XML- or JSON-based format or a concrete textual or visual syntax may need to be supported. The corresponding representation format may actually be imposed on the system by external factors. Consequently, the system:Company may need to consume such an external representation through parsing, as covered by the present feature, or it may need to produce such data through unparsing, as covered by the extra Feature:Unparsing.

An implementation of parsing is to be demonstrated for a sample company as follows. In the most basic case, the implementation has to illustrate at least 'acceptance' of the input. Another option is that parsing populates a data model for companies. Yet another option is to perform the computation for totaling salaries according to Feature:Total along with parsing.

Relationships

- Feature:Parsing is complemented by Feature:Unparsing.
- Feature:Parsing and Feature:Unparsing are related to serialization, as covered by the designated Feature:Serialization, but we speak of parsing specifically, when the system:Company needs to actually process (parse) the external format, thus going beyond the uniform use of a serialization framework.

Contribution: haskellAcceptor

Headline

Parsing (acceptance only) in Haskell with Parsec

Motivation

The implementation demonstrates parsing (acceptance) in Haskell with the Parsec library of parser combinators. A concrete textual syntax for companies is assumed. Acceptance is considered only. Thus, no abstract syntax is constructued. We set up basic parsers for quoted strings and floating-point numbers. Further, we compose parsers for companies, departments, and employees using appropriate parser combinators for sequences, alternatives, and optionality. By design, the acceptor is kept simple in terms of leveraged programming technique; in particular, monadic style and applicative functors are avoided to the extent possible.

Illustration

We would like to process a textual representation of companies; "..." indicates an elision:

```
company "Acme Corporation" {
  department "Research" {
    manager "Craig" {
      address "Redmond"
      salary 123456.0
    }
    employee "Erik" {
      address "Utrecht"
      salary 12345.0
    }
    employee "Ralf" {
      address "Koblenz"
      salary 1234.0
    }
  }
  department "Development" {
    ...
  }
}
```

Let's assume that the textual representation is defined by the following context-free grammar:

```
company = "company" literal "{" department* "}"
department = "department" literal "{" manager subunit* "}"
subunit = nonmanager | department
manager = "manager" employee
nonmanager = "employee" employee
employee = literal "{" "address" literal "salary" float "}"
```

We can now apply a mapping from the grammar to a functional program in the following way:

- Each nonterminal becomes a function that is of Parsec's parser type.
- The function definition composes parsers following the production's structure.
- We may need to deal with lexical trivia such as spaces.
- We may want to check for the end-of-file to be sure to have looked at the complete input.

At this point, we are merely interested in the syntactic correctness of such inputs. Thus, the parser functions do not need to construct any proper parse trees. They merely return "()".

Here is the parser function for departments:

```
-- Accept a department
parseDepartment :: Acceptor
parseDepartment
  =  parseString "department"
  >> parseLiteral
  >> parseString "{"
  >> parseManager
```

```
  >> many parseSubUnit
  >> parseString "}"
```

The composition uses ">>" for sequential composition in the same way as the original production for departments uses juxtaposition for the sequential composition of various terminals and nonterminals. The type *Acceptor* is defined as a parser type where the type of parse trees is "()":

```
-- The parser type for simple acceptors
type Acceptor = Parsec String () ()
```

We also need parsers for the basic units of input: literals (strings) and floats. Here is the parser for floats:

```
-- Accept a float
parseFloat :: Acceptor
parseFloat
  =  many1 digit
  >> char '.'
  >> many1 digit
  >> spaces
  >> return ()
```

That is, a float is defined to start with a non-empty sequence of digits, followed by ".", followed by another non-empty sequence of digits. In addition, any pending spaces are consumed as well. Finally, "()" is returned as the trivial parse tree of such an acceptor.

Relationships

- Contribution:haskellParsec advances this acceptor into a proper parser.
- Contribution:antlrAcceptor and others use the same textual representation.

Architecture

There are these modules:

- Main: acceptance test
- Company/Parser: the actual parser (acceptor)

The input is parsed from a file "sampleCompany.txt".

Usage

See https://github.com/101companies/101haskell/blob/master/README.md.

Concept: Maybe monad

Headline

A monad for dealing with partiality or error handling

Illustration

Let us put to work the Maybe monad in a simple interpreter.

A baseline interpreter

There are these expression forms for floats, addition, and square roots:

```
-- Simple arithmetic expressions
data Expr = Constant Float | Add Expr Expr | Sqrt Expr
  deriving (Eq, Show, Read)
```

Consider these samples:

```
-- Sample terms
sample = Sqrt (Constant 4)
sample' = Sqrt (Constant (-1))
```

The first expression should evaluate to 2.0. Evaluation should somehow fail for the second one. The most straightforward interpreter may be this one:

```
-- A straightforward interpreter
eval :: Expr -> Float
eval (Constant f) = f
eval (Add e1 e2) = eval e1 + eval e2
eval (Sqrt e) = sqrt (eval e)
```

Adding error handling to the interpreter

This interpreter would return *NaN* (not a number) for the second sample. This is suboptimal if we want to represent the error situation explicitly as an error value so that we cannot possibly miss the problem and it is propagated properly. To this end, we may use a Maybe type in the interpreter as follows:

```
-- An interpreter using a Maybe type for partiality
eval' :: Expr -> Maybe Float
eval' (Constant f) = Just f
eval' (Add e1 e2) =
  case eval' e1 of
    Nothing -> Nothing
    Just f1 ->
      case eval' e2 of
        Nothing -> Nothing
        Just f2 -> Just (f1 + f2)
eval' (Sqrt e) =
  case eval' e of
    Nothing -> Nothing
    Just f -> if f < 0.0
              then Nothing
              else Just (sqrt f)
```

Alas, the resulting interpreter is harder to understand. Maybes need to be handled for all subexpressions and the intention of propagating *Nothing* is expressed time and again.

Monadic style

By conversion to monadic style, we can hide error handling:

```
-- A monadic style interpreter
evalM :: Expr -> Maybe Float
evalM (Constant f) = return f
evalM (Add e1 e2) =
  evalM e1 >>= \f1 ->
  evalM e2 >>= \f2 ->
```

```
  return (f1 + f2)
evalM (Sqrt e) =
  evalM e >>= \f ->
  guard (f >= 0.0) >>
  return (sqrt f)
```

We can also use do notation:

```
-- A monadic style interpreter in do notation
evalM' :: Expr -> Maybe Float
evalM' (Constant f) = return f
evalM' (Add e1 e2) = do
  f1 <- evalM' e1
  f2 <- evalM' e2
  return (f1 + f2)
evalM' (Sqrt e) = do
  f <- evalM' e
  guard (f >= 0.0)
  return (sqrt f)
```

The Maybe monad

The Maybe monad is readily provided by the Haskell library, but we may want to understand how it might have been implemented. The corresponding instance of the type class *Monad* follows:

```
-- Monad instance for Maybe
instance Monad Maybe
  where
    return = Just
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
```

The definition of *return* conveys that a pure computation is successful. The definition of *bind* conveys that *Nothing* for the first argument is to be propagated. The Maybe monad actually is a more special monad, i.e., a monad with + and *0*:

```
-- Type class MonadPlus (see Control.Monad)
class Monad m => MonadPlus m
  where
    mzero :: m a
    mplus :: m a -> m a -> m a

-- MonadPlus instance for Maybe
instance MonadPlus Maybe
  where
    mzero = Nothing
    Nothing `mplus` y = y
    x `mplus` _ = x
```

The Haskell library provides the *guard* function, which we used in the interpreter:

```
-- Succeed or fail
guard :: MonadPlus m => Bool -> m ()
guard b = if b then return () else mzero
```

In modern Haskell, we also need to make *Maybe* an instance of *Applicative* (for applicative functors and *Functor* (for functors). This code is omitted here, but see GitHub for this page.

Technology: Parsec

Headline

A parser combinator library in Haskell

Illustration

Parsec-based parsers are built from parser combinators. For instance, the following trivial expressions denote parser for a digit or a letter, respectively. Such parsers for character classes are readily provided by Parsec:

```
digit
```

```
letter
```

The following expression denotes a parser for a non-empty sequence of digits; the *many1* combinator corresponds essentially to "+" in EBNF notation for context-free grammars:

```
many1 digit
```

We will look at other combinators shortly, but let us first run the composed parsers. Parsec provides a *runP* function. For instance, we can attempt to parse a digit:

```
> runP digit () "" "1"
Right '1'
```

The input string for the parser *digit* is "1". The remaining arguments resolve some parameterization of Parsec which we skip here. The run returns the successfully parsed character in the right summand of an either type; the left operand is reserved for error handling. We see an unsuccessful parse, indeed, in the next example:

```
> runP digit () "" "x"
Left (line 1, column 1):
unexpected "x"
expecting digit
```

That is, we receive an error message with line and column information about the discrepancy between actual and expected input. Clearly, the input "x" cannot be parsed as a digit. Let us also run the parser for non-empty sequences on a few inputs:

```
> runP (many1 digit) () "" "42"
Right "42"
> runP (many1 digit) () "" "42x"
Right "42"
> runP (many1 digit) () "" "x42"
Left (line 1, column 1):
unexpected "x"
expecting digit
```

The first is successful because the input string "42" is exactly a sequence of digits. The second parse is also successful because the input string "42x" does at least have a sequence of digits as a prefix. The third parse fails because it does not start with a non-empty sequence of digits.

We can compose parser sequentially and by choice:

```
> runP (letter >> digit) () "" "a1"
Right "1"
> runP (many1 (letter <|> digit)) () "" "a1"
Right "a1"
```

The first parser parses a sequence of a letter and a digit. The second parser parses any non-empty sequence of letters or digits (""). Consider the parse tree returned for the first parsers. It is evident that the first component of the sequence does not contribute to the resulting parse tree. This is because the simple form of sequential composition (">>") indeed ignores the result of the first operand. We would need to leverage a more complex form of sequential composition (">>=") to explicitly capture the intermediate results for both operands and return their composition. Thus:

```
> runP (letter >>= \l -> digit >>= \d -> return [l,d]) () "" "a1"
```

```
Right "a1"
```

This form of sequential composition passes the result from the first operand to the second so that the latter can capture the result with a lambda. We also see how the sequential composition is finished off with a trivial parser with simply *returns* a value. We can also use the value-passing form of sequential composition to improve the earlier example of a parser for a digit sequence such that the parser returns an actual int rather than a list of characters:

```
> runP (many1 digit >>= \s -> return (read s :: Int)) () "" "42"
Right 42
```

That is, we compose *many1 digit* with a function which converts the parsed string to an int and returns it as the final result. The function *return* is also a parser combinator, which is used when a given value should be returned as opposed to invoking an actual parser on the input. (If you are familiar with monads, then you realize that Parsec leverages a monad with its operations *return*, ">>", and ">>=" for parsing, but if you are not aware of monads, then this should not be any problem.)

In the most general case, parsers are of this polymorphic type:

```
data ParsecT s u m a
```

The type parameters serve these roles:

- *s*: the stream type for the input
- *u*: a type for user state, e.g., for a symbol table
- *m*: an extra monad to add effects to parsing
- *a*: the type of the parse tree

When actual parsing does not involve any underlying monad, then the identity monad is used:

```
type Parsec s u = ParsecT s u Identity
```

In simple applications of Parsec, the stream type is *String]] and no user state is used. This results in the following simplification; we also provide a simplified variation on* runP*:*

```
type Parsec' = Parsec String ()
```

Here is another sample parser. It models names as they are similarly defined in many language syntax. That is, names should start with a letter and proceed with any number of letters or digits:

```
name :: Parsec' String
name = letter
     >>= \l -> many (letter <|> digit)
     >>= return . (l:)
```

The interesting bit is how we (need to) compose the initial letter with the remaining sequence. That is, we need to "cons up" the first letter with the remaining sequence. For instance:

```
> runP' name "a42 b88"
Right "a42"
```

See Contribution:haskellParsec for an illustration of using Parsec.

Language: Haskell

Headline

The functional programming language Haskell

Details

101wiki hosts plenty of Haskell-based contributions. This is evident from corresponding back-links. More selective sets of Haskell-based contributions are organized in themes: Theme:Haskell data, Theme:Haskell potpourri, and Theme:Haskell genericity. Haskell is also the language of choice for a course supported by 101wiki: Course:Lambdas_in_Koblenz.

Illustration

The following expression takes the first 42 elements of the infinite list of natural numbers:

```
> take 42 [0..]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41]
```

In this example, we leverage Haskell's lazy evaluation.

Concept: Grammar

Headline

A set of formation rules for strings, trees, graphs, or other artifacts

Illustration

See the concept of context-free grammars for a more specific form of grammars and an associated illustration.

Concept: Reader monad

Headline

A monad for environment passing

Contribution: haskellParsec

Headline

Parsing in Haskell with Parsec

Motivation

The implementation demonstrates parsing in Haskell with the Parsec library of parser combinators. A concrete textual syntax for companies is assumed. Parse trees are constructed in accordance to an abstract syntax defined in terms of algebraic data types. We set up basic parsers for quoted strings and floating-point numbers. Further, we compose parsers for companies, departments, and employees using appropriate parser combinators for sequences, alternatives, and optionality. By design, the parser is kept simple in terms of leveraged programming technique; in particular, monadic style and applicative functors are avoided to the extent possible.

Illustration

See Contribution:haskellAcceptor for a basic illustrationof Parsec-based parsing. The present contribution is more complex in that it constructs proper parse trees.

```
parseDepartment :: Parser Department
parseDepartment = Department
  <$> (parseString "department"
  >>  parseLiteral)
  <*  parseString "{"
  <*> parseEmployee "manager"
  <*> many parseSubUnit
  <*  parseString "}"
```

To this end, we use a parser type that is still parametric in the type of parse trees. Thus:

```
-- Shorthand for the parser type
type Parser = Parsec String ()
```

Relationships

- Contribution:haskellAcceptor is merely an acceptor as opposed to the proper parser at hand.
- Contribution:haskellVariation sponsored the data model used in the present contribution.
- Contribution:antlrAcceptor and others use the same textual representation.

Architecture

There are these modules:

- Main: parser test
- Company/Parser: the actual parser
- Company/Data: the abstract syntax definition
- Company/Sample: a baseline for testing at the level of abstract syntax

The input is parsed from a file "sampleCompany.txt".

Usage

See https://github.com/101companies/101haskell/blob/master/README.md.

Concept: Syntax definition

Headline

The process of defining syntax or the status of an artifact to serve as a definition

---

Concept: Applicative functor

Headline

A functor with function application within the functor

Description

Applicative functors are described here briefly in Haskell's sense.

The corresponding type class (modulo some simplifications) looks as follows.

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

The expectation is that *pure* promotes a value to a functorial value whereas "" can be seen as a variation of fmap such that a function within the functor (as opposed to just a plain function) is applied to a functorial value.

The following laws are assumed.

```
pure f <*> x = fmap f x
pure id <*> v = v
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
```

Illustration

Simple examples

We make *Maybe* and lists applicative functors:

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  (Just f) <*> x = fmap f x

instance Applicative [] where
  pure x = [x]
  fs <*> xs = [ f x | f <- fs, x <- xs ]
```

Thus, in the *Maybe* case, a *Nothing* as a function makes us return a *Nothing* as result, but if the function is available then it is fmapped over the argument. In the list case, we use a list comprehension to apply all available functions too all available values.

The instances can be exercised at the Haskell prompt as follows:

```
> Just odd <*> Just 2
Just False
> [odd, even] <*> [1,2,3,4]
[True,False,True,False,False,True,False,True]
```

To see that applicative functors facilitate function application for functorial values pretty well, consider the following functorial variation on plain function application.

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

Consider the following application.

```
> (+) <$> [1,2] <*> [3,4]
[4,5,5,6]
```

Thus, the applicative operator "" is used to line up (any number of) functorial arguments and *fmap* is used for the "rest" of the application.

A more advanced example

We will use now an applicative functor to support environment passing within a recursive computation.

Consider the following interpreter for simple expressions:

```
data Exp
  = Var String
  | Val Int
  | Add Exp Exp

-- Environments with a fetch (lookup) function
type Env = [(String, Int)]
fetch x ((y,v):n) = if x==y then v else fetch x n

-- Straightforward interpreter; we take care of environment passing
eval :: Exp -> Env -> Int
eval (Var x) n = fetch x n
eval (Val v) _ = v
eval (Add e1 e2) n = eval e1 n + eval e2 n
```

We can evaluate expressions like this:

```
> eval (Add (Var "x") (Val 22)) [("x", 20)]
42
```

Let's try to switch to a more combinatorial style such that we abstract from explicit environment passing. To this end, we leverage the so-called SKI combinators:

```
-- More point-free, combinatorial interpreter hiding some environment passing
eval' :: Exp -> Env -> Int
eval' (Var x) = fetch x
eval' (Val v) = k v
eval' (Add e1 e2) = k (+) `s` eval' e1 `s` eval' e2

-- https://en.wikipedia.org/wiki/SKI_combinator_calculus
i :: a -> a
i x = x -- aka id
k :: a -> b -> a
k x y = x -- aka const
s :: (a -> b -> c) -> (a -> b) -> a -> c
s x y z = x z (y z) -- aka <*> of applicative
```

The applicative functor for the instance "(->) a" provides exactly the necessary abstraction:

```
-- Switch to applicative functor style, thereby demonstrating a general pattern
eval'' :: Exp -> Env -> Int
eval'' (Var x) = fetch x
eval'' (Val v) = pure v
eval'' (Add e1 e2) = pure (+) <*> eval'' e1 <*> eval'' e2
```

Concept: Parsing

Headline

Analysis of text and construction of parse trees

Relationships

- Parsing is performed by a parser.
- Parsing is the opposite of unparsing.
- Parsing solves the parsing problem.
- In the context of textual input syntax, a parser consumes text according to (a grammar for) concrete syntax and (typically) construct parse trees according to an abstract syntax.

Illustration

See Contribution:haskellParsec for an extensive illustration.

Here is a simple illustration based on the assumed I/O behavior of parsing Java code.

Input of parsing: text

Consider the following textual representation of a Java statement, subject to Java's concrete syntax.

```
x = 42
```

Output of parsing: tree

Here is the corresponding parse tree, assuming some XML-based representation for abstract syntax:

```
<assign>
 <lhs>
   <id>x</id>
 </lhs>
 <rhs>
   <constant>42</constant>
 </rhs>
</assign>
```
