The Practice of Haskell Programming

Exercises

Andres Löh

Version: June 21, 2012

1 Developing Haskell Applications

The goal of this part is to become more familiar with developing Haskell programs and using Haskell tools. This part is a bit more detailed than the later parts, because there are many small steps involved. Nevertheless, sometimes you will need to look for additional info on the web. Also don't hesitate to ask the assistants if you have any questions.

1.1 Cabal, cabal-install, Hackage

The `Cabal` library provides an infrastructure for Haskell packages. The `cabal-install` packages provides – somewhat confusingly – the binary frontend called `cabal`.

Exercise 1.1. Figure out how to invoke the `cabal` binary on your machine and type `cabal help` to get some general help.

Exercise 1.2. Download the current package list from Hackage by saying `cabal update`. This can take a little while.

Exercise 1.3. Type `cabal list` to get the long, long list of packages that are available on Hackage.

Exercise 1.4. Go to `http://hackage.haskell.org` and then click on the link "Packages" to see the same list of packages in a (slightly) nicer format.

Exercise 1.5. Say `cabal list hlint`. Then find the `hlint` package also in your browser in the Hackage package list. Click on the link. Verify that the version listed by `cabal list hlint` is the latest version currently available.

Exercise 1.6. Click on the "package description" link near the bottom of the page. Look at the package description in detail. Look what kind of information is presented here, and see if you understand all of it.

Further below, you'll find sections labeled "library" and "executable". This package defines both a library for use in other programs, and a binary that can be executed on the command line by the user once the package is installed.

Note how the "build-depends" line in the "library" section lists other packages of particular versions as dependencies.

Note how the "exposed-modules" line lists a single Haskell module of the library that can be used from other packages, and how "other-modules" lists several modules that are private to the library.

Exercise 1.7. Say `cabal unpack hlint` on your shell. This will download the latest version of `hlint` and unpack it for inspection in a directory underneath your current working directory. Change to that directory, and find the `hlint.cabal` file in there again. Verify that it is the same as the one you've looked at before online.

Exercise 1.8. Try to find the Haskell sources in the package. The `.cabal` file specifies where they are in the lines labeled "hs-source-dirs". Figure out how module names are mapped to the directory hierarchy. Find the single exposed module and look at that. You'll see that this module imports another source module and re-exports that. Find this wrapped module and briefly look at that.

Exercise 1.9. In the module you're just looking at, there is a line at the top that is a so-called *pragma*. It looks a bit like a comment, but is a special instruction for the compiler. Browse to the GHC homepage at `http://www.haskell.org/ghc/`, then find your way from there to "The User's Guide". In the documentation, look for pragmas and in particular the kind of pragma you found in the source file.

Exercise 1.10. The GHC User's Guide mentions a ghc invocation in the section you have just read to display a certain set of capabilities of ghc. Type in that command on your shell and browse the long list.

Exercise 1.11. Type `cabal install --dry-run hlint`. (This will only work if you've issued the command `cabal update` before.) This will present you with a list of packages that `cabal` is about to install on your system. Note how there is a certain similarity between this list of packages and the dependencies listed in `hlint`'s `.cabal` file or on the Hackage package for `hlint`.

Exercise 1.12. Now actually install `cabal install hlint`. This will download and install the packages to your system, underneath your home directory – so no special permissions are required. Depending on your exact configuration, this installation may *fail*, and if you repeat the command, you might see an error message claiming that `happy` could not be found. If so (and *only if you got the error message*), type `cabal install happy`.

You can already see that while `cabal` is great at resolving dependencies between libraries, it is not great at resolving dependencies between build tools, and `happy` is a tool, not a library.

Exercise 1.13. The `cabal` command installs binaries into a non-standard location, namely into the directory `.cabal/bin` underneath your home directory (note the initial `.`, which makes the file "hidden" on a Unix system). Change your search path to include this directory. If you've had to install `happy` before, then you should now be able to run `happy` from the command line, and then run `cabal install hlint` to completion.

Then you should be able to run `hlint` as well.

1.2 Code quality

We are going to run a few tools on code you and others have produced, to get a feel for how the tools behave, and to gain an understanding for stylistic issues.

Exercise 1.14. Go back to the top directory of the unpacked `hlint` *sources* and type `hlint src` on your shell. Note that even the author of `hlint` gets suggestions to improve his own code. Try to understand a few of the suggestions.

Exercise 1.15. Now run `hlint` on a couple of your own source files. Look at the suggestions closely. Try to fix a few of them. Then run `hlint` again. See if you get more.

Exercise 1.16. Call `ghc -Wall` or `ghci -Wall` on a few of your source files in order to see what warnings GHC will produce. Try to understand (and if possible fix) the warnings.

1.3 Building a Cabal package

The goal is now to create your own Cabal package out of one of your own programs.

Exercise 1.17. Try to create a small, but complete Haskell application, by taking a file with a couple of functions, making sure that there is no module header or alternative a line called

```
module Main where
```

on top. Then add a main function as follows:

```
main :: IO ()
main = print (...)
```

where you replace ... by an interesting function call. Verify that you can compile this file by calling `ghc` (not `ghci`) on the source file. Run the resulting executable and confirm it is producing the correct result.

Exercise 1.18. Create a new directory, named after the package you are going to create (choose a simple name). Place the source file in that directory. Then change to that directory and type `cabal init` on the shell. Follow the dialogue. Ideally, at the end, you will have a `.cabal` file in your directory. Look at that file.

Make sure the file has an "executable" section and a line "main-is" that points to the right file.

Exercise 1.19. Type `cabal configure` followed by `cabal build`. This should build your sources. The binary will not be installed, but will be located at `dist/build/...` underneath the package directory. To install, type `cabal install` (without further arguments). Run the installed command. Make some changes to the source file, then type `cabal install` again, then run the command again.

Exercise 1.20. Split your package into a library and an executable part, i.e., create a separate module with the function and have the main module import that module and just contain the `main` function. Choose a sufficiently unique module name for your library module.

Edit your `.cabal` file to have an extra `library` section. Use the `.cabal` file of `hlint` for inspiration.

Exercise 1.21. Use `cabal install` on your changed package to install both library and binary. Call `ghci` from a different directory and see if you can use `:m` on the `ghci` prompt to make your module available, and if you can run a function therein.

1.4 Haddock

Haddock is the most commonly used documentation tool for Haskell.

Exercise 1.22. Verify that `haddock` is properly installed on your machine by typing the command `haddock --version` on the command line.

Exercise 1.23. Go to Hackage in your browser and find the `fibonacci` package. In the "Modules" section, click on the link for the "Data.Numbers.Fibonacci" module. The page you're now seeing has been generated by Haddock.

Both `cabal install` and `cabal unpack` the `fibonacci` package. Find the source file for the module and figure out how the comments in the file have been annotated to produce the Haddock markup.

Look at the Haddock manual, to be found via the Haddock homepage at `http://haskell.org/haddock`, for further documentation on the Haddock markup format.

Exercise 1.24. Edit the library file in your own package and add Haddock comments. Then call `cabal haddock` to generate documentation for your package. Find it in the `dist` directory underneath your package directory, and view it in the browser.

2 Data structures

The goal here is to get familiar with a number of different data structures – in particular lists and sets or finite maps. Do not forget to run `hlint` and `ghc -Wall` on your code from time to time.

2.1 A spell checker

Exercise 2.1. A simple spell checking function has the following type:

```
spellCheck :: String → Dictionary → [String]
```

It takes an input string, a dictionary, and produces a list of words that occur in the string, but not in the dictionary.

Let us assume

```
type Dictionary = [String]
```

for now.

Use functions

```
words :: String → [String]
lines :: String → [String]
```

to split the input string into words. Use GHCi to test functions whenever necessary. Test the spellCheck function on very small hand-generated inputs.

Exercise 2.2. Look at the Platform library documentation for lots of Haddock documentation about the basic libraries:

```
http://lambda.haskell.org/platform/doc/current/index.html
```

Also look at Hoogle

```
http://haskell.org/hoogle
```

to find functions by name *and by type*. And there's Hayoo

```
http://holumbus.fh-wedel.de/hayoo/hayoo.html
```

a search engine with yet again slightly different capabilities.

Exercise 2.3. Here is a wrapper that allows you to read the file to be spell-checked and the dictionary from disk, where we assume the dictionary contains simply one word per line:

```
spellCheckFiles :: FilePath → FilePath → IO ()
spellCheckFiles input dict =
  do
    inputTxt ← readFile input
    dictTxt  ← readFile dict
    let incorrectWords = spellCheck inputTxt (lines dictTxt)
    print incorrectWords
```

Note that

```
type FilePath = String
```

Try to run the wrapped spellCheck function on a text and dictionary of your choice. Note that most Unix machines contain some suitable dictionaries in `/usr/share/dict` – otherwise you'll certainly find suitable dictionaries on the internet.

Exercise 2.4 (bonus)**.** Replace

```
print incorrectWords
```

above with

```
mapM_ putStrLn incorrectWords
```

and see how that changes the output. Look at the types of the functions involved and see if you can figure out what's going on.

Exercise 2.5 (bonus)**.** Figure out how to access command line parameters of a program (hint: look at the documentation for module System.Environment). Write a main program for your spell checker that reads the two file names from the command line when the executable is invoked.

2.2 Sets

We are now going to represent the dictionary using a *set* instead of a list. A set is internally represented using a balanced search tree, so finding a word in a set is somewhat easier than finding it in a list.

Exercise 2.6. Write a version of the spell checker that uses

```
type Dictionary = Set String
```

Does this improve performance?

Exercise 2.7. Adapt the spell checker to do the comparison in a case-insensitive manner, but to still print the word in the way it was capitalized in the input file if it is found to be incorrect.

Exercise 2.8. Adapt the spell checker to print a line and column number for every word that is found to be incorrect.

2.3 Tries

A *trie* is a data structure enabling efficient lookup of sequences, by storing elements with a common prefix in a common subtree.

We are going to define a trie datatype with the following implementation:

```
data Trie k v = Node (Maybe v) (M.Map k (Trie k v))
```

Here, M.Map is the type of finite maps from Data.Map. We assume

```
import qualified Data.Map as M
```

A trie is a tree. Each node contains an optional value (corresponding to the current prefix), and a number of of subtrees indexed by the next element of the key. There can be many subtrees, so we store these in a finite map.

Let us look at how looking up an key works in a trie:

```
lookup :: Ord k ⇒ [k] → Trie k v → Maybe v
lookup []       (Node v _ ) = v
lookup (k : ks) (Node _ cs) = case M.lookup k cs of
                                Nothing → Nothing
                                Just t  → lookup ks t
```

Exercise 2.9. Define the empty trie

```
empty :: Trie k v
```

and a function

```
insert :: Ord k ⇒ [k] → v → Trie k v → Trie k v
```

that adds a new element to a trie. Then, define a function

```
fromList :: Ord k ⇒ [([k], v)] → Trie k v
```

that turns a list of key-value pairs into a trie.

Exercise 2.10. If we're not interested in the values, we can use

```
type TrieSet k = Trie k ()
```

Define a function

```
fromList′ :: [[k]] → TrieSet k
```

that creates such a trie from a list of keys.

Exercise 2.11. Rewrite the spell checker to use

```
type Dictionary = TrieSet Char
```

2.4 Further ideas

Exercise 2.12. Reimplement the finite map code from the lecture.

Exercise 2.13. Write a visualizer for binary search trees (as text). Use the visualizer to display what happens during a rotation.

3 Testing

The goal of this part is to make you familiar with the QuickCheck and HUnit libraries, as well as Haskell Program Coverage.

3.1 Lecture

Exercise 3.1. Reimplement the sorting function and the tests from the lecture.

Exercise 3.2. Define and test a property that sorting is *idempotent*, i.e., that sorting a list twice produces the same result as sorting it once.

Exercise 3.3. Define and test a property that your sorting function is the same as Data.List.sort.

Exercise 3.4. Figure out what happens if you test a polymorphic or overloaded property in GHCi. What test values are being generated? I.e., if you have a property parameterized by a polymorphic list, what element types are chosen? What lessons do you draw from this behaviour?

For example, try to test the following property:

```
everyListIsSorted :: Ord a ⇒ [a] → Bool
everyListIsSorted xs = sorted xs
```

(generalize the type of sorted if necessary).

Exercise 3.5. Can you adjust the dropTwice property from the lecture so that it actually passes?

3.2 Permutations

Exercise 3.6. Define a function

```
permutations :: [a] → [[a]]
```

that generates all permutations of a list.

Exercise 3.7. Define a function

```
sameElems :: [a] → [a] → Bool
```

that more efficiently checks if one list is a permutation of the other. Avoid using sort here. Use sameElems to make the test in the lecture more efficient.

3.3 Stability

These are bonus exercises.

Exercise 3.8. Generalize sort to sortBy:

```
sortBy :: (a → a → Ordering) → [a] → [a]
```

The standard comparison function is compare. So

```
sortBy compare
```

should be the same as the original sort.

Exercise 3.9. Can you test whether a given comparison function is a proper ordering?

Exercise 3.10. Try to define a property that tests if sortBy is stable, i.e., if "equal" elements appear in the same order in both the original and the sorted list.

3.4 Merge sort

Exercise 3.11. Implement a function

```
mergeSort :: Ord a ⇒ [a] → [a]
```

that sorts a list by applying the "merge sort" (rather than "insertion sort") algorithm. Expose your function to the same tests as the original sort function.

3.5 Generators

Exercise 3.12. Use sample on a number of generators to test them. See what arbitrary does for different types. See what vector does. Note that you have to use type annotations in GHCi in order to get the right "instance" of sample.

Exercise 3.13. Look at the Haddock documentation for Test.QuickCheck.Modifiers (on Hackage). See if you can in the same style generate a modifier

```
newtype EvenList a = ...
```

that encapsulates a list of even numbers.

Exercise 3.14. Try to define an instance of Arbitrary for Set, by generating a list and transforming the list into a set.

Exercise 3.15. Let us try to define an instance of Arbitrary for binary trees:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
  deriving (Eq, Show)
```

One first attempt is to use the oneof function provided by QuickCheck:

```
oneof :: [Gen a] → Gen a
```

Now we can define:

```
instance Arbitrary a ⇒ Arbitrary (Tree a) where
  arbitrary = oneof [pure Leaf <∗> arbitrary,
                     pure Node <∗> arbitrary <∗> arbitrary]
```

Note that you have to import the Control.Applicative module in order to be able to use pure and (<∗>).

Try sample on the resulting generator a couple of times. Do you notice that some of the trees are very small and others are *very* large? There is even a good chance that sample will generate infinitely large trees. Can you see why?

In order to fix this problem, we have to keep track of the size of generated trees. We can make use of the fact that generators already have hidden information about the rough size of the terms they want generated. We can access this hidden information using

```
sized :: (Int → Gen a) → Gen a
```

Now complete the following definition:

```
instance Arbitrary a ⇒ Arbitrary (Tree a) where
  arbitrary = sized gen
    where
      gen :: Arbitrary a ⇒ Int → Gen (Tree a)
      gen 0 = pure Leaf <∗> arbitrary          -- zero-sized trees are always leaves
      gen n = let subtree = gen (n 'div' 2)    -- generate smaller subtrees
              in ...
```

Make sure you not only produce balanced trees! Try the resulting generator using sample once again, and verify that the trees stay much smaller.

If you like, you can tweak the generated trees somewhat more by using

```
frequency :: [(Int, Gen a)] → Gen a
```

rather than oneof that picks the different elements of the list with potentially unequal probability.

The technique we have described here can be used to define good Arbitrary instances for other tree-like data structures.

Exercise 3.16. Write a function

```
size :: Tree → Int
```

that computes the number of leaves in a tree (use the standard design pattern for trees). Also write a function

```
flatten :: Tree a → [a]
```

that traverses a tree from left to right and produces its elements in a list.

Exercise 3.17. Test that flattening a tree and measuring the length of the resulting list is the same as computing the size of the tree. Write a function

```
rev :: Tree a → Tree a
```

and test that reversing a tree twice yields the original tree.

3.6 Haskell Program Coverage

Exercise 3.18. Compile your current test program for Haskell Program Coverage. Run a few tests. Then generate a report. See if you can identify uncovered code. Try to obtain complete coverage.

Exercise 3.19. Do a similar thing for the spell checker you wrote: compile it for Haskell Program Coverage, then run it on an example file and dictionary. Do you get full coverage?

3.7 HUnit

Exercise 3.20. Look at the Hackage documentation of `HUnit`. Define a number of simple test cases for your spell checker using `HUnit`. Try to improve coverage by doing so.

3.8 Further ideas

Exercise 3.21. Have a look at the `test-framework-hunit` and `test-framework-quickcheck2` packages.

Exercise 3.22. Figure out how test suites can be integrated into a `.cabal` file. There are examples on HackageDB, for instance the `math-functions` package.

There is also some documentation at

```
http://www.haskell.org/cabal/users-guide/developing-packages.html#test-suites
```

However, the documentation *incorrectly* tells you that you should use the `detailed` test-suite type. That's wrong, as `detailed` is unfortunately not quite ready for prime time yet. But using the `exitcode-stdio` type is entirely fine.

4 Evaluation

4.1 Heap profiles

Exercise 4.1. Generate heap profiles for the following functions:

```
rev  = foldl (flip (:)) []
rev' = foldr (\x r → r ++ [x]) []
```

by using them as function f in a main program as follows

```
main = print $ f [1..1000000]
```

(adapt the size of 1000000 according to the speed of your machine to get good results). Interpret and try to explain the results!

Exercise 4.2. Do the same for

```
conc xs ys = foldr (:) ys xs
conc'      = foldl (\k x → k ∘ (x:)) id
```

with

```
main = print $ f [1..1000000] [1..1000000]
```

(where f is conc or conc').

Exercise 4.3. Now have a look at

```
f1 = let xs = [1..1000000] in if length xs > 0 then head xs else 0
f2 = if length [1..1000000] > 0 then head [1..1000000] else 0
```

with

```
main = print f
```

(where f is f_1 or f_2).

4.2 Selective strictness

Exercise 4.4. Try to figure out how many different values of type

```
(Bool, Bool)
```

there are if you consider $\perp$ to be a separate value. Define them all. Now try to write programs that *distinguish* these values. How many fully defined values are there?

Exercise 4.5. For concrete datatypes, it is possible to force evaluation by pattern matching rather than seq. So can you solve the previous exercise without using seq or $! (if you haven't already)?

Exercise 4.6. Write a function

```
spineList :: [a] → b → b
```

that forces the spine (but not the elements) of the complete list before returning its second argument. Then write

```
forceList :: [a] → b → b
```

that also forces the elements.

Exercise 4.7. Make a heap profile of your spell checker. If you still have them, do it for the different versions using different data structures.

Does it change anything in memory and runtime if you force the dictionary before you start checking?

Exercise 4.8. Why would a function

```
force :: a → a
force = seq a a
```

be useless?

4.3 Sharing and memoization

Let us try to compute the edit distance between two lists. An edit operation is supposed to be

```
data Edit a = Cp a | Ins a | Del a
  deriving (Eq, Show)
```

We are aiming to write

```
diff :: Eq a ⇒ [a] → [a] → [Edit a]
```

and

```
cost :: [Edit a] → Int
```

such that we can then define

```
dist :: Eq a ⇒ [a] → [a] → Int
dist xs ys = cost (diff xs ys)
```

Exercise 4.9. Define the function

```
cost :: [Edit a] → Int
```

such that each Cp operation is free, and each other operation has a cost of 1.

Exercise 4.10. Define a function

```
patch :: Eq a ⇒ [a] → [Edit a] → Maybe [a]
```

that tries to apply a sequence of edit operations to a given string.

Exercise 4.11. Define a function

```
revert :: [Edit a] → [Edit a]
```

that changes all inserts into deletions and all deletions into insertions.

Exercise 4.12. Define a function

```
pick :: Int → [Edit a] → [Edit a] → [Edit a]
```

that looks at two sequences of edit operations and picks the better one by looking at a prefix of the given length each and picking the one where the prefix has lower cost.

Exercise 4.13. Define the function

```
diff :: Eq a ⇒ [a] → [a] → [Edit a]
```

that traverses the two lists in parallel. If one of the lists is empty, it is clear that we have to either insert or delete. If both lists are non-empty, we look at the first element. If the two elements are the same, then copying is clearly the best option. Otherwise, we have a choice between

- deleting an element from the first list (Del), or
- deleting an element from the second list (Ins).

We simply call pick with a suitable bound to choose the better one.

Exercise 4.14. Try to test a few properties such as that diffing a string with itself yields the empty edit sequence, or that patching the first of two strings with their diff yields the second string. Is the dist function commutative? Does diffing two strings both ways yield a patch and its reverted patch? Does the value we pass to pick make a difference for correctness and/or efficiency?

Exercise 4.15 (difficult)**.** Try to fix the efficiency problems of diff by writing a version that makes use of tabulation.

4.4 Further ideas

Exercise 4.16. Extend the spell checker to make suggestions based on edit distance.

Exercise 4.17. Implement the game "Boggle", and an automatic solver for it.

5 EDSLs

5.1 Parsers

Exercise 5.1. Reimplement the simple parsers from the lecture. You have two options: Use a type synonym:

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

Or use a **newtype**/**data**:

```
newtype Parser a = P (String → [(a, String)])
```

If you go for the first version, you should not try to reuse the standard classes Functor and Applicative, but rather define the operators in unoverloaded form yourself. If you use the **newtype**, you can try to make the parsers an instance of Functor, Applicative and even Monad.

Exercise 5.2. Define a parser

```
eof :: Parser ()
```

that succeeds only if the end of input has been reached

Exercise 5.3. Define a parser

```
≪|> ::Parser a → Parser a → Parser a
```

that defines a "greedy" version of <|> – the second option should only be tried if the first yields no successful result.

Exercise 5.4. Use the greedy choice to implement a greedy version of many:

```
greedy :: Parser a → Parser [a]
```

Also implement many1 and greedy1 that expect *at least one occurrence* of the argument.

Exercise 5.5. Write a parser

```
num :: Parser Int
```

that parsers a non-empty sequence of digits followed by spaces in a greedy way, and interprets the result as an integer.

Exercise 5.6. Modify the parser

```
ident :: Parser String
```

given in the lecture to also be greedy and consume spaces at the end.

Exercise 5.7. Define a parser

```
key :: String → Parser String
```

that greedily parses a given string followed by spaces. In particular

```
plus :: Parser String
plus = const <$> key "+" <*> spaces
```

should parse a + followed by spaces.

Exercise 5.8. Consider this expression language:

```
data Expr = Lit Int | Add Expr Expr
  deriving (Eq, Show)
```

Write a parser for this language, allowing only natural number literals and a right-associative infix +, to avoid left-recursion.

Exercise 5.9. Given this parser combinator:

```
chainl1 :: Parser a → Parser (a → a → a) → Parser a
chainl1 p op  =  (\x fs → foldl (flip ($)) x fs)
             <$> p <*> many (flip <$> op <*> p)
```

and the definition

```
expr :: Parser Expr
expr = chainl1 (Lit <$> num) ((\_ x y → Add x y) <$> plus)
```

try to understand what chainl does. Draw a picture if necessary of how the input is split into pieces and consumed.

5.2 Parsec

Exercise 5.10. Try to use the same parser using the parser combinator library parsec. Use the documentation to figure out where to start. Many combinators will hopefully be familiar.

5.3 Extended expressions

Let us add variables and let-binding to the expression language:

```
data Expr = Lit Int                -- natural number literal
          | Add Expr Expr          -- addition
          | Var String             -- identifier
          | Let String Expr Expr   -- let x = e1 in e2
  deriving (Eq, Show)
```

Exercise 5.11. Define a function

```
free :: Expr → Set String
```

that determines the free variables in an expression, i.e., the variables not bound by any Let. Assume that the Let is non-recursive (in deviation from Haskell semantics), so that the bound variable only scopes over the second expression.

Exercise 5.12. Define a function

```
eval :: Expr → Env → Maybe Int
```

that evaluates an expression in a given environment. An environment Env should map identifier names to values. Choose a suitable implementation of Env.

Exercise 5.13. Now let us define

```
newtype Eval a = E (Env → Maybe a)
```

Then make Eval an instance of the classes Monad, Functor and Applicative (hint: as explained in the lecture, you only have to think about Monad, and can define the other two in a straightforward way). Then reimplement

```
eval :: Expr → Eval Int
```

Exercise 5.14. Find the documentation for the MonadReader class and try to implement

```
instance MonadReader Env Eval
```

Then reimplement eval without ever doing pattern matching on E.

Exercise 5.15. Extend the parser to cover the extended expression language.

5.4 Postfix

This is rather tricky. See it as a challenge of what can be done within Haskell's type system.

Exercise 5.16. Find Haskell definitions for the functions start, stop, store, add and mul such that you can embed a stack-based language into Haskell:

```
p1, p2, p3 :: Int
p1 = start store 3 store 5 add stop
p2 = start store 3 store 6 store 2 mul add stop
p3 = start store 2 add stop
```

Here, p_1 should evaluate to 8 and p_2 should evaluate to 15. The program p_3 is allowed to fail at runtime.

Hint: Type classes are *not* required to solve this assignment. Try to first think about the types that the operations should have, then about the implementation.

Exercise 5.17. Once you have that, try to find a solution that rejects programs that require non-existing stack elements during type checking.

5.5 Further ideas

Exercise 5.18. Look at the `diagrams` package.

Exercise 5.19. Look at a library for *lenses*.

Exercise 5.20. Look at Text.PrettyPrint.

6 Data-parallel arrays

6.1 Repa

Exercise 6.1. In order to work with `repa`, you need to install the `repa` package using `cabal`.

Exercise 6.2. Create an unboxed array one-dimensional array with integer elements 1 up to 12 using the function fromListUnboxed. Note that you may have to use type annotations, because so much overloading is being used in `repa`. You want a result of type

```
Array U (Z :. Int) Int
```

Try what happens if the length of the list does not match the shape you are passing.

Exercise 6.3. Try to create a similar one-dimensional array of Double and Bool. What happens if you try to create an array with elements of type Maybe Int?

Exercise 6.4. Try to create an array of the first 25 square numbers using the function fromFunction. GHCi will complain that it cannot show the result. Why do you think that is? What is the type of the array you constructed?

Try to apply computeSUnboxed to the array. How does that change the type? Can it now be shown?

Exercise 6.5. What is the extent of the shape Z :. 3 :. 4 :: DIM2? And of Z :. 2 :. 2 :. 2 :: DIM3? Try to use fromListUnboxed to create arrays of these shapes. Use (!) to access an element from the array.

Exercise 6.6. Write a function

```
rev :: (Source r e) ⇒ Array r DIM1 e → Array D DIM1 e
```

that reverses a one-dimensional array by using the function backpermute.

Exercise 6.7. Write a function

```
cartesian :: (Source r1 a, Source r2 b) ⇒
             (a → b → c) → Array r1 DIM1 a → Array r2 DIM1 b →
             Array D DIM2 c
```

that takes a function and two one-dimensional arrays and computes a two-dimensional array containing the results of applying the function to all possible combinations of values from the two arrays.

Try to use the function unsafeTraverse2 to achieve this goal.

Exercise 6.8. Use cartesian with a somewhat expensive function such as $\backslash x\ y \rightarrow \log\ (x + 1) * \log\ (y + 1)$ to compute the cartesian product matrix of a vector with itself. Use fromFunction to create a suitable vector where the element at index i contains the Double that is i. Then use sumAllP to sum up the entire matrix.

Compile the program in various ways: compare the run-time for the unoptimized and optimized version. Compare the runtime for `-N1`, `-N2`, `-N4`, and `-N8`. Also look at the RTS statistics by passing the `-s` RTS flag. How many arrays are actually computed here?

Exercise 6.9. Modify the above example. What happens if you force the original vector using computeP? (Note that the monadic type of computeP is there so that you explcitly do that in sequence, because `repa` does not support nested parallel computations.) What happens if you force the result of cartesian before calling sumAllP?

Exercise 6.10. Look at the `repa-examples` package. Try some of the examples there, for example the matrix multiplication example.

6.2 Data and type families

This part is not needed for being able to use Repa. It demonstrates how to use one of Haskell's more advanced type system features in order to express complex relationships between types while maintaining full type safety. To enable the extension being used here, you will have to put `{-# LANGUAGE TypeFamilies #-}` at the top of your source file.

Internally, Repa (and `vector` and Accelerate) use an adaptive representation for arrays of tuples. Arrays of tuples are actually stored using several arrays, one for each component. This allows unboxed arrays to be used in such a situation, and helps efficiency.

In Repa, the Array data family is being used to select different representations based on the parameters. We will simulate a similar idea in a much simpler setting, using a type family. Data and type families are overloaded datatypes or type synonyms.

We are going to use a type family List that computes a "flat" representation of lists:

```
type family List a :: *
type instance List Int    = [Int]
type instance List Double = [Double]
type instance List (a, b) = (List a, List b)
```

Read this definition as if it was a type-level function. For some types a, we explain how to compute type List a.

Exercise 6.11. You can query GHCi for the result of expanding a type family by using the `:kind!` command (with the bang at the end). Use `:kind! List Int` and `:kind! List (Double, Int)` and observe the results. If you enter a type for which no definition is given, you won't get an error – instead, GHCi will simply not expand the type family (and that will make this combination unusable in practice).

Exercise 6.12. We now define an ordinary type class that converts between a list and a List:

```
class Flat a where
  flat   :: [a] → List a
  unflat :: List a → [a]
```

Provide instances for Int, Double and pairs. The first two are trivial, for the latter you will want to use the zip and unzip functions.

Then compute:

```
flat (replicate 10 (((1,2),(3,4)) :: ((Int, Double),(Double, Int))))
```

The type annotation is needed here. Can you unflat the result back to its original form?

6.3 Benchmarking

Exercise 6.13. Look at the `criterion` package. Follow the documentation on Hackage and in particular look at the Criterion.Main module. Implement the Fibonacci benchmark shown there as an example.

Exercise 6.14. Use `criterion` to benchmark your matrix multiplication code.

Exercise 6.15. Try `criterion` on some older programs of yours. Try, for example, to make a comparative benchmark of your insertion sort, your merge sort and the Data.List.sort implementations.