Università degli Studi di Udine
Dipartimento di Matematica e Informatica
Corso di Laurea in Informatica

Bachelor Thesis

Implementation of an interpreter with graphical interface of a simple imperative language

Candidate:
Matteo Morena

Supervisor:
Prof. Marco Comini

Academic Year 2021–2022

Dipartimento di Matematica e Informatica
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine
Italia

Abstract

Programming languages are the essential tool to create software. They can be classified in various ways, such as object-oriented or functional, interpreted or compiled, statically or dynamically typed, etc. There are hundreds of such languages, each one having its own set of benefits and disadvantages. Given their central role in software development, I consider it important to understand how programming languages themselves are made.

For the purposes of this dissertation, I developed *Devin*, an imperative language complete with graphical user interface to edit and execute code. The goal of this thesis is to provide readers with the knowledge necessary to understand the fundamental concepts of programming language internals, while offering a practical example of implementation.

Building programming languages is not a trivial task. Multiple concepts, such as type theory, syntax and formal grammars, automata, deductive systems and semantics have to be understood. Nevertheless, implementing programming languages is a rewarding process; it is my objective to provide an introduction to anyone interested in this topic.

Lots of resources explain the theory behind programming languages; many forego implementation. I believe that true understanding comes only after practice as well: that is, by writing actual code. Devin is a small, minimal, language. It strikes a good balance in terms of explanatory power: it is simple enough to be described entirely, yet complex enough to hint at how more advanced features would need to be implemented.

Acknowledgments

I would like to thank my supervisor, professor Marco Comini, for the insightful conversations we had, for his assistance in developing parts of the software discussed in this dissertation, and for proofreading of both source code and the thesis itself.

I also extend my thanks to Matteo Poggi and Alberto Pastorutti for their time in reading parts of my thesis, providing valuable feedback, and offering emotional support.

Contents

1

Haskell programming

This chapter provides an overview of the Haskell programming language. I'll assume that the reader has familiarity with any imperative programming language like Java or C++.

Haskell [1] is a purely functional programming language. Programs are not encoded as a sequence of steps with procedures abstracting over them; notably, there are no statements. Instead, everything in Haskell is an expression; functions are the primary building blocks that abstract over expressions.

Functions operate on their arguments and compute some result. If a function gets called twice with the same arguments, it's guaranteed to yield the same result: this is called *referential transparency*. This property guarantees that every function application can be substituted with its corresponding value; thinking about execution as such a rewrite system can improve correctness while making it easier to reason about program behavior.

In Haskell, expressions are evaluated *lazily* by default: their value is not computed until it is actually needed. Among others, lazy evaluation allows the definition and manipulation of infinite data structures, such as the list of all squares. This evaluation model implies that the order of execution isn't defined by the programmer; rather, it is up to the runtime to decide when and what to evaluate. This aligns well with referential transparency, encouraging to think of programs as a series of transformations on data.

1.1 Expressions and function definitions

In Haskell, a program is a set of functions. For instance, the function for calculating the hypotenuse of a triangle with catheti `x` and `y` can be defined as follows:

```
pythagoras = \x y -> sqrt (x ^ 2  + y ^ 2)
```

This declaration binds the name `pythagoras` to a function of two arguments which applies Pythagoras' theorem. The symbol \ is pronounced *lambda*, reflecting the notation used in lambda calculus: $\lambda xy.\sqrt{x^2 + y^2}$.

In Haskell, functions are first-class citizens: they can be used as ordinary values. The syntax used earlier makes this explicit, as `pythagoras` is bound to a value that *is* a function. Alternatively, syntactic sugar allows for an equivalent definition:

```
pythagoras x y = sqrt (x ^ 2 + y ^ 2)
```

Haskell is *statically typed*. In the definition for `pythagoras`, `x` and `y` must support multiplication, addition and exponentiation; thus, the compiler infers that they must be floating-point numbers. Explicit type annotations can be provided: in fact, it is conventional to include them in top-level definitions for clarity.

```
pythagoras :: Floating a => a -> a -> a
pythagoras x y = sqrt (x ^ 2 + y ^ 2)
```

The type annotation specifies that, given any type `a` that supports floating-point arithmetic, `pythagoras` is a function `a -> a -> a`. The parameter types are separated by `(->)`, and there appears to be no special distinction between parameter and return types.

Data in Haskell is immutable; this includes arguments and local bindings. Consider implementing a function to compute the factorial of a number n in an imperative language: you'd likely have an accumulator initially set to 1, and a loop that repeatedly multiplies said accumulator with the numbers from 1 to n. Since there is no concept of mutable variables in Haskell, other mechanisms have to be employed instead; the simplest purely functional implementation might use linear recursion:

```
factorial :: Integral a => a -> a
factorial 0 = 1
factorial n | n > 0 = n * factorial (n - 1)
```

The definition above may be read as follows: the factorial of `n` is `1` if `n` equals `0`, and is `n * factorial (n - 1)` if `n > 0`. The boolean expression `n > 0` is called a *guard*, and the result of `factorial` is undefined if `n < 0`. An equivalent definition could have been given using conditionals:

```
factorial :: Integral a => a -> a
factorial n =
  if n == 0 then 1
  else if n > 0 then n * factorial (n - 1)
  else undefined
```

Yet another alternative to define the factorial function is to employ tail recursion:

```
factorial' :: Integral a => a -> a
factorial' n = go 1 n
  where
    go acc 0 = acc
    go acc i | i > 0 = go (acc * i) (i - 1)
```

The function `factorial'` (the single quote `'` is part of its name) employs a local helper function `go` which performs the actual computation. Notice that `go` resembles the imperative loop from `1` to `n` which accumulates the product into `acc`. Since Haskell is lazily evaluated, the difference in performance between `factorial` and `factorial'` is negligible.

Haskell supports the definition of arbitrary infix operators. For instance, an integer exponentiation operator could be defined as follows:

```
(^) :: (Num a, Integral b) => a -> b -> a
b ^ 0 = 1
b ^ n | n > 0 = b * (b ^ (n - 1))
```

Binary infix operators are just functions with a name made up of one or more special characters like `!`, `#`, `$`, `%`, `&`, `*`, `+`, `.`, `/`, `<`, `=`, `>`, `?`, `@`, `\`, `^`, `|`, `-`, `~`, `:`. In fact, given `b` and `n`, the expression `b ^ n` has an equivalent prefix form, `(^) b n`; a non-symbolic alias can be defined as `pow = (^)`, making `b ^ n` equivalent to `pow b n`. Symmetrically, any binary function can be used as an infix operator: given the function `div` for integer division, `div n d` has an equivalent infix form, `n `div` d`.

1.2 Fundamental data types

Without importing other modules, Haskell supports the following numeric types:

- `Int`: fixed-precision integers, at least in the range $\left[-2^{29}, 2^{29}-1\right]$ [2];
- `Word`: unsigned fixed-precision integers of the same size as `Int`;
- `Integer`: arbitrary precision integers;
- `Float`: single-precision floating-point numbers;
- `Double`: double-precision floating-point numbers;
- `Rational`: rational numbers, represented as a ratio of two `Integer` values.

All numeric types support addition, subtraction, multiplication, and comparison. The sign of a number `x` can be extracted with `signum x`; its absolute value can be computed with `abs x`. The integral data types `Int`, `Word` and `Integer` support integer division via `div` and remainder via `mod`. Two numbers `x` and `y` of the same fractional type can be divided with `x / y`. Further, many trigonometric, hyperbolic and related functions like `sin`, `cos`,

`sqt` and `log` are supported by the floating-point data types `Float` and `Double`. More functions can be found in Haskell's standard library, as described in the documentation.

In addition to numeric types, Haskell provides `Bool` and `Char` as basic data types. These represent the boolean values `True`/`False` and Unicode scalar values, respectively.

1.3 Tuples

Multiple values can be grouped into *tuples*. For example, two-dimensional vectors (x, y) can be stored as a 2-tuple of type `(Double, Double)`; addition, dot product and magnitude functions can thus be defined as follows:

```
sum2D :: (Double, Double) -> (Double, Double) -> (Double, Double)
sum2D (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)

dot2D :: (Double, Double) -> (Double, Double) -> Double
dot2D (x1, y1) (x2, y2) = x1 * x2 + y1 * y2

magnitude2D :: Vector2D -> Double
magnitude2D v = sqrt (dot2D v v)
```

The functions `sum2D` and `dot2D` both deconstruct the arguments via *pattern matching*: the individual components of the tuples are bound to names for use within the function definitions. As we'll see, pattern matching is extensively used in Haskell.

For 2-tuples, Haskell provides the `fst` and `snd` functions to extract the first and second item, respectively. Using these, `sum2D` and `dot2D` could have been defined as:

```
sum2D :: (Double, Double) -> (Double, Double) -> (Double, Double)
sum2D v1 v2 = (fst v1 + fst v2, snd v1 + snd v2)

dot2D :: (Double, Double) -> (Double, Double) -> Double
dot2D v1 v2 = fst v1 * fst v2 + snd v1 * snd v2
```

In general, tuples can have any number of components, each of which can be of any type. For example, `(1, '2', "3")` denotes a 3-tuple of type `(Int, Char, String)`.

1.4 Lists

Singly linked lists are essential in Haskell. In fact, the standard library represents strings as lists of characters. Lists consist of zero or more elements of the same type; for instance, a list of the numbers `4`, `5` and `6` can be defined as follows:

```
list456 :: [Int]
list456 = [4, 5, 6]
```

The *cons operator* `(:)` can be used to efficiently prepend an element to a list. The `(++)` operator can be used to concatenate two lists.

```
list3456 :: [Int]
list3456 = 3 : list456

list123456 :: [Int]
list123456 = [1, 2] ++ list3456
```

The complexity of `(++)` is $O(n)$, where n is the length of the first operand. This can be easily proven by analyzing the function's implementation:

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

Again, pattern matching is used to ease the implementation. The pattern `[]` matches an empty list, while `x : xs` matches a non-empty list with `x` as its head and `xs` as its tail. The signature `(++) :: [a] -> [a] -> [a]` indicates that the types of the items of the two lists being concatenated don't matter, as long as both lists contain items of the same type `a`.

Haskell provides many functions for operating on lists. Some commonly used ones are:

- `head :: [a] -> a`: returns the first element of a list;
- `tail :: [a] -> [a]`: returns the elements after the head of a list;
- `null :: [a] -> Bool`: tests whether a list is empty;
- `length :: [a] -> Int`: returns the length of a list;
- `reverse :: [a] -> [a]`: returns the elements of a list in reverse order;
- `take :: Int -> [a] -> [a]`: given n, returns the prefix of length n of a list;
- `drop :: Int -> [a] -> [a]`: given n, returns the elements after the first n;
- `elem :: Eq a => a -> [a] -> Bool`: tests whether an element occurs in a list;
- `(!!) :: [a] -> Int -> a`: list index operator, starting from 0;
- `delete :: Eq a => a -> [a] -> [a]`: removes the first occurrence of an element;
- `sort :: Ord a => [a] -> [a]`: sorts elements from lowest to highest.

1.5 Strings

As mentioned, Haskell strings are just lists of characters. For example, the word `Hello` is represented by the literal `"Hello"`, which desugars to the list `['H', 'e', 'l', 'l', 'o']`.

The standard library exports the following type alias:

```
type String = [Char]
```

Since strings are lists, they support all operations provided for lists:

```
olleh :: String
olleh = reverse "Hello"
```

In addition to all the list operations, the following functions are specifically designed to work with strings:

- `lines :: String -> [String]`: splits the input string into a list of strings at line break characters.
- `words :: String -> [String]`: splits the input string into a list of words which were delimited by white space.
- `unlines :: [String] -> String`: the inverse of `lines`. It joins a list of lines, appending a line break after each one.
- `unwords :: [String] -> String`: the inverse of `words`. It joins a list of words with separating spaces.

1.6 User-defined types, `Maybe` and `Either`

While lists and tuples are undeniably useful, Haskell also allows the definition of new data types. This form of abstraction improves upon the use of simple tuples, as structurally similar objects like 2D vectors and complex numbers, which are both pairs of numbers, can be associated with different types.

For example, a data type for complex numbers $a + ib$ can be defined as follows:

```
data Complex = Complex Double Double
  deriving (Eq, Show)

complexRe :: Complex -> Double
complexRe (Complex a _) = a

complexIm :: Complex -> Double
complexIm (Complex _ b) = b
```

The data declaration on the first line introduces a new type, `Complex`; a value of such a type can be constructed with the constructor `Complex :: Double -> Double -> Complex`. In other words, if `a` and `b` are of type `Double`, the expression `Complex a b` constructs a new value of type `Complex`.

The deriving declaration `deriving (Eq, Show)` instructs the compiler to generate three functions: `(==), (/=) :: Complex -> Complex -> Bool` to test for equality between values, and `show :: Complex -> String` to obtain a string representation of a `Complex`.

The functions `complexRe` and `complexIm` defined on the successive lines are referred to as *accessors*. Both definitions use pattern matching to extract the required component; the wildcard pattern `_` is used to mark an ignored value.

Since accessors are commonly required, an alternative way of defining the same data type is with *record syntax*. This way, accessors are automatically generated for each field:

```
data Complex = Complex {
  complexRe :: Double,
  complexIm :: Double
} deriving (Eq, Show)
```

Data types can be defined with any number of constructors; moreover, constructors can have any number of arguments, including zero. For instance, academic grades as used in the US can be defined as follows:

```
data Grade = F | D | C | B | A
  deriving (Eq, Ord, Show)
```

We say that `Grade` is an *algebraic data type* of constructors `F`, `D`, `C`, `B`, `A`. Deriving `Ord` generates implementations for the comparison operators `(<)` and `(>)` such that the expressions `F < D`, `D < C`, `C < B` and `B < A` evaluate to `True`.

Two very useful predefined algebraic data types provided by Haskell are listed below; for simplicity, deriving clauses are omitted.

```
data Maybe a
  = Nothing
  | Just a

data Either a b
  = Left a
  | Right b
```

Both `Maybe` and `Either` are called *type constructors*, as they are parameterized by type variables: `Maybe` takes a single parameter `a`, while `Either` takes two parameters `a` and `b`. Syntactically, type variables are distinguished from type constructors by casing: variables are lowercase, while constructors are uppercase.

`Maybe` encapsulates an optional value; a value of type `Maybe a` is either empty (represented as `Nothing`), or a value of type `a` (represented as `Just a`). `Either` represents values of two possibilities; a value of type `Either a b` is either `Left a` or `Right b`.

Both algebraic data types are commonly used for error reporting. For simple cases, where functions may not yield any meaningful value, `Maybe` is used: this is similar to the possibility of a null return value in C or Java. `Either` is conventionally used to represent success or failure: in the first case, the resulting value is held by the `Right` constructor; in the second, a representation of the error is held by `Left`.

To illustrate how pattern matching can be used with algebraic data types, implementations of a few utility functions from Haskell's standard library are listed below.

```
isJust :: Maybe a -> Bool
isJust Nothing = False
isJust (Just _) = True

isNothing :: Maybe a -> Bool
isNothing Nothing = True
isNothing (Just _) = False

fromJust :: Maybe a -> a
fromJust (Just x) = x

fromMaybe :: a -> Maybe a -> a
fromMaybe x Nothing = x
fromMaybe _ (Just x) = x
```

The standard library exports a very useful function operating on association lists: `lookup`. Given a list of key-value pairs `(a, b)`, this function looks up the value corresponding to a given key, if any. If no value is found, `Nothing` is returned; this illustrates how the algebraic data type `Maybe` can be used to indicate nullability.

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
lookup _ [] = Nothing
lookup key ((k, v) : _) | key == k = Just v
lookup key (_ : xs) = lookup key xs
```

1.7 Higher-order functions

Arguably, one of the most important features of any functional programming language is the support for higher-order functions; that is, functions that take other functions as arguments or yield a function as a result. Since Haskell functions are first-class, implementing and using higher-order functions is very succinct syntactically.

Consider two functions: one calculating the sum of every element in a list, and the other calculating the product of every element in a list. These functions could be defined as:

```
listSum :: Num a => [a] -> a
listSum [] = 0
listSum (x : xs) = x + sum xs
```

```
listProduct :: Num a => [a] -> a
listProduct [] = 1
listProduct (x : xs) = x * product xs
```

Both functions share the same structure:

- If the given list is empty, the base case is reached and a constant value is returned.
- Otherwise, a binary operator is applied to the head of the list and the result of a recursive call on the tail of the list.

This idiom is called *folding* over a list. A higher-order function, `fold`, which encapsulates this behavior, can be used to implement both `listSum` and `listProduct`:

```
fold :: (a -> b -> b) -> b -> [a] -> b
fold _ z [] = z
fold f z (x : xs) = f x (fold f z xs)

listSum :: Num a => [a] -> a
listSum xs = fold (\x acc -> x + acc) 0 xs

listProduct :: Num a => [a] -> a
listProduct xs = fold (\x acc -> x * acc) 1 xs
```

Given that `(+)` and `(*)` already denote the binary operators of addition and multiplication respectively, the two functions can be rewritten more succinctly as:

```
listSum :: Num a => [a] -> a
listSum xs = fold (+) 0 xs

listProduct :: Num a => [a] -> a
listProduct xs = fold (*) 1 xs
```

Lists are not the only kind of objects that can be folded over. For example, `Maybe`s can be thought of as lists that are either empty or have exactly one element; as such, they can be folded over as well. In general, if `t` is a type constructor, `t a` is foldable if there is an instance of the function `foldr :: (a -> b -> b) -> b -> t a -> b`. Haskell's standard library provides `foldr` for all types that can be folded over; thus, `listSum` and `listProduct` can be defined more generically as:

```
sum :: (Foldable t, Num a) => t a -> a
sum xs = foldr (+) 0 xs

product :: (Foldable t, Num a) => t a -> a
product xs = foldr (*) 1 xs
```

Besides `foldr`, another commonly used higher-order function provided by the standard library is `map`. Given a list, `map` returns a new list obtained by applying an input function to all elements of the original list. One possible implementation of `map` can be defined in terms of `foldr`:

```
map :: (a -> b) -> [a] -> [b]
map f xs = foldr (\x ys -> f x : ys) [] xs
```

With `map`, one can, for instance, double all the numbers in a list:

```
doubleList :: Num a => [a] -> [a]
doubleList xs = map (\x -> 2 * x) xs
```

The section `(2 *)`, equivalent to the partial application `((*) 2)`, can be used for brevity:

```
doubleList :: Num a => [a] -> [a]
doubleList xs = map (2 *) xs
```

Finally, using Haskell's polymorphic `fmap`, a function for doubling everything that can be mapped over can be defined as follows:

```
double :: (Functor f, Num a) => f a -> f a
double xs = fmap (2 *) xs
```

Besides `foldr` and `fmap`, another commonly used higher-order function is the *application operator* `($)`. It is defined as follows:

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

While `($)` might seem redundant, it has a low right-associative binding precedence, allowing it to be used to omit parentheses. For example, the expression `f $ g $ h x` would evaluate to `f (g (h x))`. Haskell programmers frequently employ this operator; to avoid writing needlessly cryptic code, I won't use it unless strictly necessary.

Haskell's standard library exposes many other higher-order functions and operators; these are all documented at `https://hackage.haskell.org/package/base`.

1.8 Type classes

As explained in the previous section, `foldr` is an interface for a function that is implemented for a class of different types, such as `[a]` and `Maybe a`. These kinds of interfaces are encoded through Haskell's type classes: a mechanism to allow for ad-hoc polymorphism.

As a first example, the interface for objects that can be mapped over, called *functors* in category theory, can be described as follows:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The listing above is read as follows: declare a type class named `Functor` that takes a single argument `f` (a type constructor); instances of this type class must implement the function `fmap :: (a -> b) -> f a -> f b`.

One of the most trivial instances of the `Functor` type class can be defined for `Maybe`:

```
instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

By default, function signatures aren't allowed in instance declarations; most Haskell compilers support the `InstanceSigs` language extension which eliminates this restriction:

```
{-# LANGUAGE InstanceSigs #-}

instance Functor Maybe where
  fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

From the signature above, it is now apparent that `fmap` is a higher-order function mapping a value of type `Maybe a` to a value of type `Maybe b` using a given function `a -> b`.

An equivalent interpretation is that `fmap` *lifts* a unary function `a -> b` so that its argument and return types are wrapped by a `Maybe` type constructor: given some `g :: a -> b`, the partial application `fmap g` has type `Maybe a -> Maybe b`. In general, as already said, given any functor, `fmap g` is of type `Functor f => f b -> f b`.

Lists are instances of `Functor`, too. The following listing implements `fmap` equivalently to the definition of `map` from the previous section. For the sake of brevity, the `InstanceSigs` extension will be assumed to be enabled from now on.

```
instance Functor [] where
  fmap :: (a -> b) -> [a] -> [b]
  fmap _ [] = []
  fmap f (x : xs) = f x : fmap f xs
```

Another example of a functor is `Either`. Recall that this algebraic data type represents failure or success through the `Left` and `Right` constructors. The value wrapped by `Right` can be somewhat arbitrarily chosen to be the subject of mapping. Implementing a `Functor` instance requires the type constructor `Either` to be applied partially, as follows:

```
instance Functor (Either a) where
  fmap :: (b -> c) -> Either a b -> Either a c
  fmap _ (Left x) = Left x
  fmap f (Right x) = Right (f x)
```

Haskell's standard library defines many type classes. Another quite important one is `Eq`, which provides equality `(==)` and inequality `(/=)` methods:

```
class Eq a where
  (==) :: a -> a -> Bool
  x == y = not (x /= y)

  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

This declaration gives default method implementations for both `(/=)` and `(==)`. Instances of this class should implement at least one of the two operators; if neither `(==)` nor `(/=)` are provided, both will loop. Primitive data types are instances of this class.

Instances of the `Eq` are typically generated via `deriving` declarations, as explained in section 1.6. Of course, such implementations can also be handwritten:

```
instance Eq Complex where
  (==) :: Complex -> Complex -> Bool
  Complex a1 b1 == Complex a2 b2 = a1 == a2 && b1 == b2
```

Interestingly, most composite types like lists and types provided by Haskell's base library implement `Eq` as well. For example, `Maybe`'s instance can be defined as follows:

```
instance Eq a => Eq (Maybe a) where
  (==) :: Maybe a -> Maybe a -> Bool
  Nothing == Nothing = True
  Just x == Just y = x == y
  _ == _ = False

  (/=) :: Maybe a -> Maybe a -> Bool
  Nothing /= Nothing = False
  Just x /= Just y = x /= y
  _ /= _ = True
```

Note that an `Eq` instance for `Maybe a` is only defined if `a` implements `Eq`. The context `Eq a =>` is lexically scoped to the instance declaration; thus, the complete signature of `(==)` and `(/=)` is `Eq a => Maybe a -> Maybe a -> Bool`.

While Haskell's type classes can be considered to be similar to Java's and C# interfaces or abstract classes, this is indeed an example of how they are more powerful. In both object-oriented languages, all non-primitive data types inherit from `Object`; among others, all objects must implement the methods `boolean equals(Object obj)` and `int hashCode()` (`bool Equals(object? obj)` and `int GetHashCode()` in the case of C#). The requirement that every object must be hashable stems from the need to support generic hash table containers. While the rationale for this requirement is certainly understandable, it can be argued that there are objects for which hashing or even equality methods don't make much sense; alas, there's no way in Java and C# to implement methods with constraints. Haskell, on the other hand, allows this: sets of functions can be defined only if certain conditions are met. For example, considering lists, the cons operator `(:)` is defined for *all* lists, while the operators `(==)` and `(/=)` are defined *only if* the list items can be checked for equality as well.

Extending `Eq`, Haskell defines a class for ordered types, `Ord`. This type class defines the methods `compare`, `(<)`, `(<=)`, `(>)`, `(>=)`, `min` and `max`:

```
data Ordering = LT | EQ | GT

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  compare x y = if x == y then EQ else if x <= y then LT else GT

  (<) :: a -> a -> Bool
  x < y = case compare x y of LT -> True; _ -> False

  (<=) :: a -> a -> Bool
  x <= y = case compare x y of GT -> False; _ -> True

  (>) :: a -> a -> Bool
  x > y = case compare x y of GT -> True; _ -> False

  (>=) :: a -> a -> Bool
  x >= y = case compare x y of LT -> False; _ -> True

  min :: a -> a -> a
  min x y = if x <= y then x else y

  max :: a -> a -> a
  max x y = if x <= y then y else x
```

The primitive types `Int`, `Word`, `Integer`, `Float`, `Double`, `Rational`, `Bool` and `Char` are all instances of this class, as are `[a]`, `Maybe a` and `Either b a`, given an appropriate `a`.

Most type classes are associated with certain *laws*: contracts that methods must adhere to. Among others, `Ord` requires reflexivity; that is, for all `x`, `x <= x` must be `True`. Haskell compilers can't enforce these laws statically. In fact, `Float` and `Double` are two unlawful instances of `Ord`: in both cases, `let nan = 0.0 / 0.0 in nan <= nan` evaluates to `False`. Because some laws can be difficult to understand, and because even Haskell's own standard library sometimes ignores them with unlawful instances, I'll avoid mentioning them further. In most cases, implementing a type class instance satisfies its laws without thinking about them.

1.9 The `Functor`, `Applicative` and `Monad` type classes

Arguably, the most important type classes defined in Haskell's `base` library are `Functor`, `Applicative` and `Monad`. To recap, the `Functor` class is defined as:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The definition for the `Applicative` class is listed below. Note that the official Haskell language report doesn't define the `Applicative` class; moreover, it doesn't define any relationship between it and `Monad` [3]. In practice, many recent Haskell implementations

define `Applicative`, so its definition is given for completeness; the `Monad` type class extends `Applicative`'s interface. The operator `(<$>)` is used as an infix alias of `fmap`.

```
class Functor f => Applicative f where
  pure :: a -> f a

  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  liftA2 f mx my = f <$> mx <*> my

  (<*>) :: f (a -> b) -> f a -> f b
  mf <*> mx = liftA2 (\f -> f) mf mx

  (*>) :: f a -> f b -> f b
  mx *> my = liftA2 (\_ y -> y) mx my

  (<*) :: f a -> f b -> f a
  mx <* my = liftA2 (\x _ -> x) mx my
```

As explained, `Functor`'s `fmap` lifts a function `a -> b` into `f a -> f b`. The `Applicative` class extends this concept by providing the method `liftA2`, which lifts a binary function `a -> b -> c` into `f a -> f b -> f c`. Further, it provides `pure` to lift a value `a` into `f a`. The functions `(<*>)`, `(*>)` and `(<*)` can be derived in terms of `liftA2`.

Consider writing a function to sum two `Maybe`s. Without the `Applicative` class, the following verbose definition would be needed:

```
sum2Maybes :: Num a => Maybe a -> Maybe a -> Maybe a
sum2Maybes (Just x) (Just y) = Just (x + y)
sum2Maybes _ _ = Nothing
```

Complex programs working with many data types would need to employ lots of pattern matching to work with lifted values. Avoiding this is one of the use cases of the higher-order function `liftA2`:

```
sum2Maybes :: Num a => Maybe a -> Maybe a -> Maybe a
sum2Maybes mx my = liftA2 (+) mx my
```

As an alternative to `liftA2`, `(<$>)` and the *sequential application operator* `(<*>)` can be used to obtain an equivalent definition:

```
sum2Maybes' :: Num a => Maybe a -> Maybe a -> Maybe a
sum2Maybes' mx my = (+) <$> mx <*> my
```

Looking at it, the expression `(+) <$> mx <*> my` resembles a regular application of the operator `(+)` in the form of `(+) x y`. While the implementation for `sum2Maybes'` is somewhat more complex than `sum2Maybes`—using two higher-order functions instead of one—it is syntactically more general than the `liftA2` approach.

Using `(<$>)` and `(<*>)`, functions of any arity can be lifted and applied:

```
sum3Maybes :: Num a => Maybe a -> Maybe a -> Maybe a -> Maybe a
sum3Maybes mx my mz = sum3 <$> mx <*> my <*> mz
  where
    sum3 x y z = x + y + z

sum4Maybes :: Num a => Maybe a -> Maybe a -> Maybe a -> Maybe a -> Maybe a
sum4Maybes mx my mz mw = sum4 <$> mx <*> my <*> mz <*> mw
  where
    sum4 x y z w = x + y + z + w
```

Given the definition of `Applicative`, one might wonder why it subclasses `Functor`. The reason is that, mathematically, every applicative is also a functor. This can be proven by providing an implementation for `fmap` using only `liftA2` and `pure`:

```
amap :: Applicative f => (a -> b) -> f a -> f b
amap f mx = liftA2 (\_ x -> f x) (pure ()) mx
```

Here, `()` is the empty tuple, also known as *unit*; its value is ignored by the function lifted by `liftA2`. Using `const x _ = x` as defined in Haskell's standard library, `amap` can be simplified to:

```
amap :: Applicative f => (a -> b) -> f a -> f b
amap f mx = liftA2 (const f) (pure ()) mx
```

Extending `Applicative`, there's the `Monad` type class. It defines the *bind operator* `(>>=)`, along with `(>>)` and `return`. These last two methods are maintained for backwards compatibility and can be ignored by readers.

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b

  (>>) :: m a -> m b -> m b
  mx >> my = mx >>= const my

  return :: a -> m a
  return x = pure x
```

Monads are perhaps Haskell's greatest innovation [4]; in the next section, I'll explain why the `Monad` type class turns out to be so useful.

Just as every applicative is a functor, every monad is an applicative; hence, all three classes are of great importance in the Haskell ecosystem. The listing below demonstrates how to derive `liftA2` using only `(>>=)` and `return`, thereby proving the relationship between applicatives and monads:

```
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
liftM2 f mx my = mx >>= \x -> my >>= \y -> return (f x y)
```

1.10 Behind the syntax of function types

As mentioned, the function definition `pythagoras x y = sqrt (x ^ 2 + y ^ 2)` is syntactic sugar for `pythagoras = \x y -> sqrt (x ^ 2 + y ^ 2)`. This definition can be further desugared into:

```
pythagoras :: Floating a => a -> a -> a
pythagoras = \x -> \y -> sqrt (x ^ 2 + y ^ 2)
```

Thus, if type `a` is an instance of the class `Floating`, `pythagoras` is a unary function of argument `x : a` that yields a closure of argument `y : a` which evaluates the expression `sqrt (x ^ 2 + y ^ 2)`. Encoding multiple arguments in this way is called *currying*, in honor of the mathematician Haskell Brooks Curry. This explains why in the signature `pythagoras :: Floating a => a -> a -> a` there's no apparent distinction between parameter and return types: because the syntax mirrors how function definitions are desugared. In Haskell, curried functions are generally preferred over other encodings such as tupled arguments (i.e. `pythagoras' :: Floating a => (a, a) -> a`).

Since `(->)` is right associative, the previous definition is also equivalent to:

```
pythagoras :: Floating a => a -> (a -> a)
pythagoras = \x -> (\y -> sqrt (x ^ 2 + y ^ 2))
```

Note that `(->)` doubles as a type constructor: given two types `a` and `b`, it constructs the function type `a -> b`. Thus, the following syntax can also be used:

```
pythagoras :: Floating a => (->) a ((->) a a)
pythagoras = \x -> (\y -> sqrt (x ^ 2 + y ^ 2))
```

It's worth mentioning that since `(->)` is a type constructor, given any type `r`, Haskell's standard library defines `Functor`, `Applicative` and `Monad` instances for `(->) r`.

1.11 Basic input and output

Let's implement a very simple program that performs the following task:

1. Prompt the user to insert a number by printing `n :=` to standard output;
2. Read a number n from standard input;
3. Compare the given number n to 0:
   - If $n < 0$, print `n < 0`;
   - If $n > 0$, print `n > 0`;
   - If $n = 0$, print `n = 0`.

In Java, such a program would be implemented as follows:

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Computation 1
        System.out.print("n := ");

        // Computation 2
        int n = scanner.nextInt();

        // Computation 3
        if (n < 0) {
            System.out.println("n < 0");
        } else if (n > 0) {
            System.out.println("n > 0");
        } else {
            System.out.println("n = 0");
        }
    }
}
```

Haskell, being purely functional, doesn't define procedures that perform side-effects like reading from standard input or writing to standard output. However, each computation can be represented as an I/O action that, *when run*, carries out the actual work. In this case, the three computations to perform are:

```
computation1 :: IO ()
computation1 = putStr "n := "

computation2 :: IO Integer
computation2 = readLn

computation3 :: Integer -> IO ()
computation3 n = case compare n 0 of
  LT -> putStrLn "n < 0"
  GT -> putStrLn "n > 0"
  EQ -> putStrLn "n = 0"
```

Since there's a `Monad` instance for `IO`, the computations can be sequenced with the operator `(>>=)`, binding the result of running one to a continuation determining the next:

```
main :: IO ()
main =
  putStr "n := " >>= \_ ->
    readLn >>= \n ->
      case compare n 0 of
        LT -> putStrLn "n < 0"
        EQ -> putStrLn "n = 0"
        GT -> putStrLn "n > 0"
```

As can be seen, the monadic interface allows for imperative-looking code to be written: the listing above already resembles the implementation in Java. Using syntactic sugar in the form of Haskell's *do expressions*, `main` can be expressed in a more readable way as:

```
main :: IO ()
main = do
  putStr "n := "
  n <- readLn

  case compare n 0 of
    LT -> putStrLn "n < 0"
    EQ -> putStrLn "n = 0"
    GT -> putStrLn "n > 0"
```

With this notation, programs of arbitrary complexity can be written. In fact, as long as imperative loops aren't required, programming in Haskell can be even more succinct than in languages that mainly focus on imperative-style programming:

```
main :: IO ()
main = do
  putStrLn "Terms of the quadratic equation:"

  putStr "a := "
  a <- readLn

  putStr "b := "
  b <- readLn

  putStr "c := "
  c <- readLn

  let d = b ^ 2 - 4 * a * c
  let x1 = (-b - sqrt d) / (2 * a)
  let x2 = (-b + sqrt d) / (2 * a)

  if isNaN x1 || isInfinite x1 || isNaN x2 || isInfinite x2 then do
    putStrLn ""
    putStrLn "No solutions!"
  else do
    putStrLn ""
    putStrLn "Solutions:"

    if x1 /= x2 then do
      putStrLn ("x1 = " ++ show x1)
      putStrLn ("x2 = " ++ show x2)
    else
      putStrLn ("x1 = x2 = " ++ show x1)
```

Note that `readLn :: Read a => IO a` is polymorphic and may read values of any type from standard input. While the preceding examples compile without problems, it may be necessary to add type annotations when the type of `readLn` can't be inferred:

```
{-# LANGUAGE TypeApplications #-}

main :: IO ()
main = do
  putStr "n := "
  n <- readLn @Integer
  putStrLn ("n + 1 = " ++ show (succ n))
```

1.12 Imperative-style programming

With the power of monads, Haskell allows for imperative-looking code to be expressed. Emphasis on *looking*: there's no actual imperative control flow; rather, referentially transparent representations of computations are chained together with `(>>=)`.

While imperative-style is always an option, it should be preferred to program in other ways, as Haskell doesn't have any syntax for constructs such as while- or for-loops. Even if higher-order functions can achieve equivalent functionality, it can be clumsy doing so.

As an example, consider, yet again, the function for computing the factorial of a number. With the `ST` monad from the standard library, it is possible to encode mutable references; these are encapsulated by the type `STRef s a`. The following operations are supported:

- `newSTRef :: a -> ST s (STRef s a)`: build a new `STRef` in the current `ST` monad;
- `readSTRef :: STRef s a -> ST s a`: read the value of an `STRef`;
- `writeSTRef :: STRef s a -> a -> ST s ()`: write a new value into an `STRef`;
- `modifySTRef :: STRef s a -> (a -> a) -> ST s ()`: mutate an `STRef`.

With these, `factorial :: Integral a => a -> a` can be implemented in terms of a helper function, say `factorialST :: Integral a => a -> ST s a`. The latter operates within the `ST` monad and uses the functions described above; the former is the one to be used by callers: it hides the `factorialST` implementation by computing the value produced by the `ST` monad through `runST :: (forall s. ST s a) -> a`.

The type variables `s` and `a` designate some internal state and the kind of value stored by mutable references, respectively. The exact meaning of `(forall s. ST s a) -> a` can be safely ignored; in short, the function `runST` returns the value computed inside the `ST` monad, ensuring that `s` remains inaccessible to the rest of the program.

```
import Control.Monad.ST
import Data.STRef

factorial :: Integral a => a -> a
factorial n = runST (factorialST n)
```

```
factorialST :: Integral a => a -> ST s a
factorialST n = do
  accRef <- newSTRef 1
  go accRef n
  readSTRef accRef

  where
    go accRef 0 = pure ()

    go accRef i | i > 0 = do
      modifySTRef accRef (* i)
      go accRef (i - 1)
```

The function `factorialST` can be read as follows: create a new mutable reference called `accRef` and initialize it to `1`, then call `go`, then return the updated value of `accRef`. The helper function `go` updates the accumulator, `accRef`, iterating from `n` downwards.

Mutable references as encoded by the `ST` monad are rarely used in Haskell programs; most of the time, a functional-style implementation suffices. There are real-world scenarios in which mutation is essential to achieve good performance, such as when implementing algorithms that make use of dynamic programming.

1.13 Importing and creating modules

In the previous section, I used import declarations to gain access to functions and types provided by the `Control.Monad.ST` and `Data.STRef` modules. Like most programming languages, Haskell provides a name spacing mechanism; while in Java there is `package` and `import`, in C# `namespace` and `using`, in Haskell there is `module` and `import`.

Haskell's standard library is called `base`; it provides many modules, each one exporting any number of functions, types, and type classes. Haskell's compiler implicitly imports the module named `Prelude`; most of the functions and types described in earlier sections come from `base`'s `Prelude` module.

Entities exported by modules can be brought into scope via qualified or unqualified import declarations. Let's consider the `Data.List` module from the `base` package: this module exports additional functions to be used on lists besides the ones provided by `Prelude`. One of these functions is `nub :: Eq a => [a] -> [a]`: it removes duplicate elements from a list, keeping only the first occurrence of each element. This function can be used with a qualified import as follows:

```
import qualified Data.List

main :: IO ()
main = do
  putStrLn "Please insert a sequence of items delimited by white space."
  input <- getLine
  putStrLn ""
  putStrLn "Without duplicates, the items are:"
  putStrLn (unwords (Data.List.nub (words input)))
```

The import declaration `import qualified Data.List` imports the entities exported by the `Data.List` module. Further, as explained in section 1.5, the function `words` splits a string on white space; its dual, `unwords`, joins a list of strings with separating spaces.

Imported modules may be assigned an alias using the `as` clause:

```
import qualified Data.List as L

main :: IO ()
main = do
  putStrLn "Please insert a sequence of items delimited by white space."
  input <- getLine
  putStrLn ""
  putStrLn "Without duplicates, the items are:"
  putStrLn (unwords (L.nub (words input)))
```

Entities from modules can also be imported in an unqualified manner, by omitting the `qualified` keyword and `as` clause. In the following, `nub` is used without qualification:

```
import Data.List

main :: IO ()
main = do
  putStrLn "Please insert a sequence of items delimited by white space."
  input <- getLine
  putStrLn ""
  putStrLn "Without duplicates, the items are:"
  putStrLn (unwords (nub (words input)))
```

Let's move on to defining modules. Consider the implementation of a module called `Triangle2D` which exports types and functions to operate on triangles situated on the Euclidean plane. Such a module would be defined in its own file, `Triangle2D.hs`; one possible implementation is given by the following listing:

```
module Triangle2D (Triangle (..), perimeter, area) where

import Data.List

data Triangle a = Triangle {x1, y1, x2, y2, x3, y3 :: a}
  deriving (Show, Read)
```

```
instance Eq a => Eq (Triangle a) where
  (==) :: Triangle a -> Triangle a -> Bool
  Triangle x11 y11 x12 y12 x13 y13 == Triangle x21 y21 x22 y22 x23 y23 =
    elem
      [(x11, y11), (x12, y12), (x13, y13)]
      (permutations [(x21, y21), (x22, y22), (x23, y23)])

perimeter :: Floating a => Triangle a -> a
perimeter (Triangle x1 y1 x2 y2 x3 y3) = a + b + c
  where
    a = segmentLength x1 y1 x2 y2
    b = segmentLength x2 y2 x3 y3
    c = segmentLength x3 y3 x1 y1

area :: Floating a => Triangle a -> a
area (Triangle x1 y1 x2 y2 x3 y3) = sqrt (s * (s - a) * (s - b) * (s - c))
  where
    s = (a + b + c) / 2
    a = segmentLength x1 y1 x2 y2
    b = segmentLength x2 y2 x3 y3
    c = segmentLength x3 y3 x1 y1

segmentLength :: Floating a => a -> a -> a -> a -> a
segmentLength x1 y1 x2 y2 = sqrt ((x2 - x1) ^ 2 + (y2 - y1) ^ 2)
```

The header `module Triangle2D (Triangle (..), perimeter, area) where` declares the name of the module being defined and the list of entities to be exported. Note that the function `segmentLength` is not exported from the `Triangle2D` module.

The module header is followed by the import declaration `import Data.List`, which, among others, brings the function `permutations` into scope.

The import declaration is followed by a set of top-level declarations: one data declaration, one instance declaration, and three function bindings. Implementations of `perimeter` and `segmentLength` are trivial and don't require further explanation; the `area` function computes its result using Heron's formula $A = \sqrt{s\,(s-a)\,(s-b)\,(s-c)}$, where $s = \frac{1}{2}\,(a+b+c)$ is the triangle's semi-perimeter. The `Eq` instance for `Triangle` verifies that the vertices of one triangle equal the vertices of the other, independently of their order; to do so, the functions `elem :: (Foldable t, Eq a) => a -> t a -> Bool` from `Prelude` and `permutations :: [a] -> [[a]]` from `Data.List` are used.

1.14 Aside: desugaring type classes

Before moving on to the next chapter, I want to mention an interesting aspect that many Haskell textbooks overlook: that is, how type classes may be implemented by the compiler.

Type classes are a fundamental abstraction provided by Haskell. This section discusses *one* possible implementation strategy; of course, the actual implementation of type classes may vary depending on the compiler. Nevertheless, the model described in this section aims to provide a deeper understanding of how type classes work and the issues they address—even if, just like the existence and distinction between the stack and the heap, it is an implementation detail.

Consider again one application of the `Eq` type class: looking up a key in an association list. Recall that the `Eq` class and `lookup` function are defined as follows:

```
class Eq a where
  (==) :: a -> a -> Bool
  x == y = not (x /= y)

  (/=) :: a -> a -> Bool
  x /= y = not (x == y)

lookup :: Eq a => a -> [(a, b)] -> Maybe b
lookup _ [] = Nothing
lookup key ((k, v) : _) | key == k = Just v
lookup key (_ : xs) = lookup key xs
```

Lookup requires an `Eq` instance to be implemented for the key type `a` of the association list `[(a, b)]`. It is a compile time error to use `lookup` if this constraint isn't met.

How would such a function operate without built-in type classes? The key insight is to consider constraints as implicitly passed parameters; with this model, the `Eq` type class desugars into a record of functions:

```
data Eq a = Eq { (==), (/=) :: a -> a -> Bool }

lookup :: Eq a -> a -> [(a, b)] -> Maybe b
lookup _ _ [] = Nothing
lookup (Eq (==) _) key ((k, v) : _) | key == k = Just v
lookup eq key (_ : xs) = lookup eq key xs
```

If type classes become record types, it follows that type class instances become record values. An `Eq` instance for the algebraic data type `Grade` of section 1.6 can be defined as:

```
gradeEqInstance :: Eq Grade
gradeEqInstance = Eq f g
  where
    f F F = True
    f D D = True
    f C C = True
    f B B = True
    f A A = True
    f _ _ = False

    g x y = not (f x y)
```

Finally, instances may be generated in terms of others:

```
maybeEqInstanceGivenEq :: Eq a -> Eq (Maybe a)
maybeEqInstanceGivenEq (Eq (==) (/=)) = Eq f g
  where
    f Nothing Nothing = True
    f (Just x) (Just y) = x == y
    f _ _ = False

    g Nothing Nothing = False
    g (Just x) (Just y) = x /= y
    g _ _ = True
```

This, of course, is the desugared version of:

```
instance Eq a => Eq (Maybe a) where
  Nothing == Nothing = True
  Just x == Just y = x == y
  _ == _ = False

  Nothing /= Nothing = False
  Just x /= Just y = x /= y
  _ /= _ = True
```

Much can be said about this model of type classes. As this is not the subject of this thesis, I'll just mention that this implementation technique is called *dictionary passing style*.

2

Parsing and processing languages

Implementing programming languages is a complex task; traditionally, this task is divided into a series of steps, or *phases*. The first phase of any compiler or interpreter is known as *parsing*. Given an input string, the parser determines if it conforms to a set of rules; if it does, a logical representation of the input is constructed.

2.1 Languages and grammars

Consider the process of parsing arithmetic expressions. Given a sequence of characters as input, syntactically invalid strings like `1+` and `3//4` must be rejected, while valid strings like `2+3*(4+5)` must be accepted.

The set of all possible inputs that a parser may accept is called the *language* accepted by that parser. The set of rules dictating which inputs are to be accepted is called the *grammar* of the language.

For example, the grammar for arithmetic expressions can be defined recursively, asserting that an expression is any of the following items:

- Any sequence of characters representing a number is a *literal expression*;
- `(`x`)` is a *parenthesized expression* if x is an expression;
- `-`x are *unary expressions* if x is a literal/parenthesized expression;
- If x is a literal/unary/parenthesized expression, x is a *term*;
- x`*`y and x`/`y are *multiplicative expressions* if y is a term and x is either a term or a multiplicative expression;
- x`+`y and x`-`y are *additive expressions* if y is either a multiplicative expression or a term, and x is either a multiplicative expression or a term or an additive expression.

According to these rules, `2`, `3`, `4` and `5` are literal expressions, `4+5` is an additive expression, `(4+5)` is a parenthesized expression, `3*(4+5)` is a multiplicative expression, and `2+3*(4+5)` is an additive expression.

Once a grammar is specified, a parser for the language in question can be defined. Crucially, the parser's output must be a structure that can be easily manipulated in subsequent phases. For most implementations, this output data structure takes the form of a tree.

2.2 Syntax trees

The syntactic structure of a string accepted by a parser is represented by a *syntax tree*. In literature, two categories are typically defined: *concrete* and *abstract* syntax trees.

Consider the expression `2+3*(4+5)` in the language of arithmetic expressions as defined in the previous section. A concrete syntax tree typically indicates which grammatical rules were used during parsing to match a given substring, as shown in Figure 2.1.

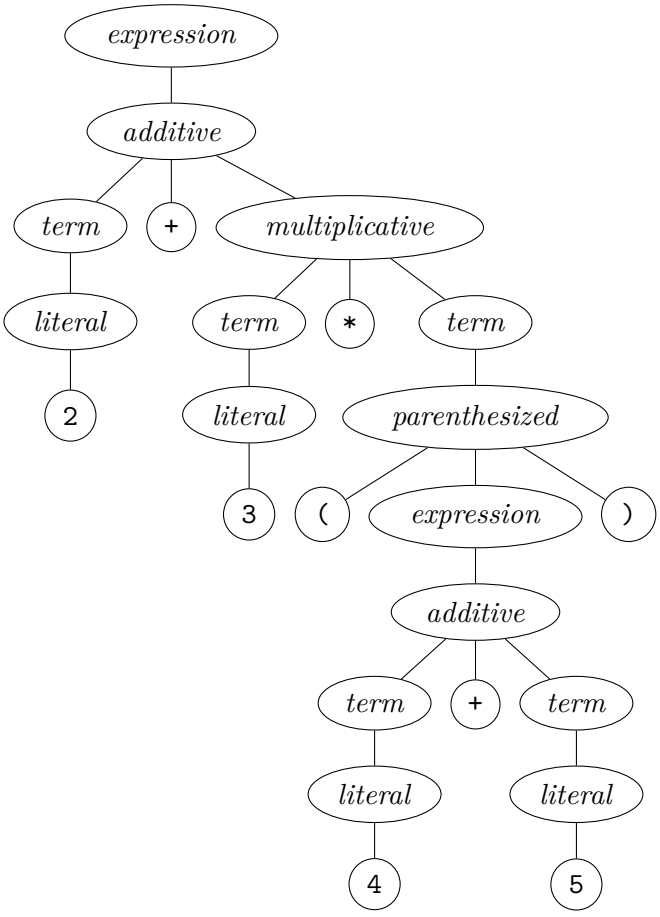

Figure 2.1: Concrete syntax tree for `2+3*(4+5)`

To ease the workflow, concrete syntax trees are often manipulated by removing redundant or unnecessary information. Figure 2.2 illustrates two possible abstract syntax trees for the string `2+3*(4+5)`, depending on how much information is to be retained. In both cases, the semantic meaning is the same: the tree still represents the same order of operations.

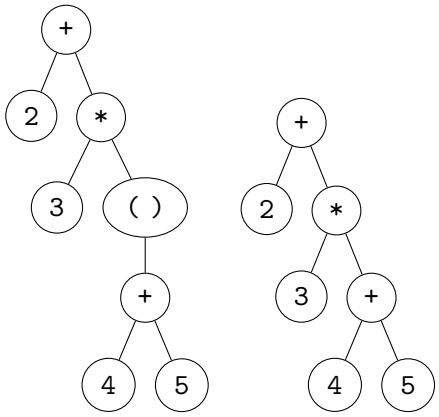

Figure 2.2: Two semantically equivalent abstract syntax trees for `2+3*(4+5)`

For our purposes, the distinction between abstract and concrete syntax trees isn't necessary: for this reason, from now on I'll just use the term *syntax tree*.

In general, syntax trees represent the structure of not only programming languages, but also markup and other kinds of languages. In this thesis, I'll focus on the processing of programming languages only.

2.3 Parser combinators

There are many ways to recognize some input and generate an associated syntax tree. Much academic research has been conducted, and there are many implementations of so-called *parser generators* readily available. These tools, which automatically generate code for parsing a language given its grammar, build upon well-known algorithms, such as those available for LL or LR [5] grammar classes, just to throw a few names around.

In this section, we'll see how there's no fundamental need for any generators. While such tools are convenient, it isn't too difficult to write a parser from scratch, using a technique called *recursive descent parsing*. Recursive descent parsers are used by the GCC compiler and the V8 JavaScript engine [6], among others.

Recursive descent parsing is a method to construct parsers using a collection of recursive functions. The simplest functions parse the atoms, or *tokens*, of the target language; examples of tokens include numbers and identifiers. By combining simpler functions,

more complex parsers can be created: for example, a parser for 2D coordinates (`x,y`) can be built by using the parsers for open-parenthesis, number, comma, number, and close-parenthesis, in sequence. Parsers can be combined recursively: a parser P could depend on a parser Q, which in turn depends on P. Consider while statements: the body of a loop is yet another statement, which could be a while statement.

The functional approach is perfectly suited for implementing recursive descent parsers. In particular, higher-order functions abstract the boilerplate that is otherwise necessary to glue different parsing functions together. These higher-order functions are called *parser combinators*, as they combine simpler parsers into more complex ones.

The general interface employed by parser combinators is that of a function that takes some input string and returns either an error or some result along with the remaining input. For instance, applying the function to parse an integer on the input string `20*2+2` yields the pair $(20, \texttt{*2+2})$.

Many libraries provide a framework for working with parser combinators. There is usually some overlap between different combinator libraries, as they all provide some shared functionality. Let P and Q be parsers; some common parser combinators include:

- The *sequence combinator*. This combinator runs two parsers in succession and combines their results. If P succeeds, then Q is run on the remaining input. If Q also succeeds, the combinator succeeds with the combined results of P and Q. If either P or Q fails, then the combinator also fails.
- The *alternative combinator*. This combinator implements choice. It non-deterministically runs P and Q on the same input. If either P or Q succeeds, the combinator succeeds with the same result as that parser. If both P and Q fail, the combinator also fails.
- The *option combinator*. This combinator runs a parser zero or one time. It tries to run P. If P succeeds, the combinator succeeds with the same result as P. If P fails, the combinator succeeds with an empty result.
- The *repetition combinator*. This combinator runs a parser zero or more times. It tries to run P as many times as possible until it fails. Each time P succeeds, the remaining input is fed back into the next application of P. This combinator succeeds with the accumulated results of all successful runs.

2.4 Tree traversal

The grammar of a language only describes its *syntax*: on its own, it only defines which input strings are valid and which are not; similarly, a parser only maps syntactically valid input strings into syntax trees. It is the role of other components to assign some *semantic meaning* to such trees.

Through tree traversal, the syntax tree constructed by a parser can be processed in a multitude of ways, depending on the desired functionality:

- An *evaluator* would associate the nodes of the syntax tree with actions to execute. Examples of such actions include performing arithmetic calculations, conditionally executing subtrees, and assigning or retrieving values from variables.
- A *compiler* would translate the syntax tree into another form, typically a lower-level target language. For example, the GCC compiler translates valid C syntax into object files: a form of machine-executable instructions and metadata. A linker would parse the metadata of one or more object files to produce an executable.
- For languages with static types, a *type checker* would verify that no type errors would occur during execution. In such systems, it is crucial that both the type checker and the evaluator/compiler adhere to the same semantics.
- A *linter* would analyze the syntax tree, suggesting improvements to the input source code. Unlike type checking, linting may not depend on static types (if any), and successful compilation/evaluation doesn't depend on the linter. As with the type checker, all components must adhere to the same semantics.
- A *syntax highlighter* would use the information stored in the syntax tree to perform syntax highlighting: displaying program text in different colors depending on semantic meaning. The syntax tree employed by a highlighter needs to be rather concrete, given that each node has to be associated with a position in the input string.

3

The Devin language and UI

In this thesis, I'll discuss the implementation of a simple imperative programming language that I have decided to call Devin. The name Devin is an homage to Duino, the city where I currently reside; its Slovenian name is, in fact, Devin.

Alongside Devin, I developed a graphical text editor for the language. The editor features syntax highlighting, error reporting, and a simple visual debugger.

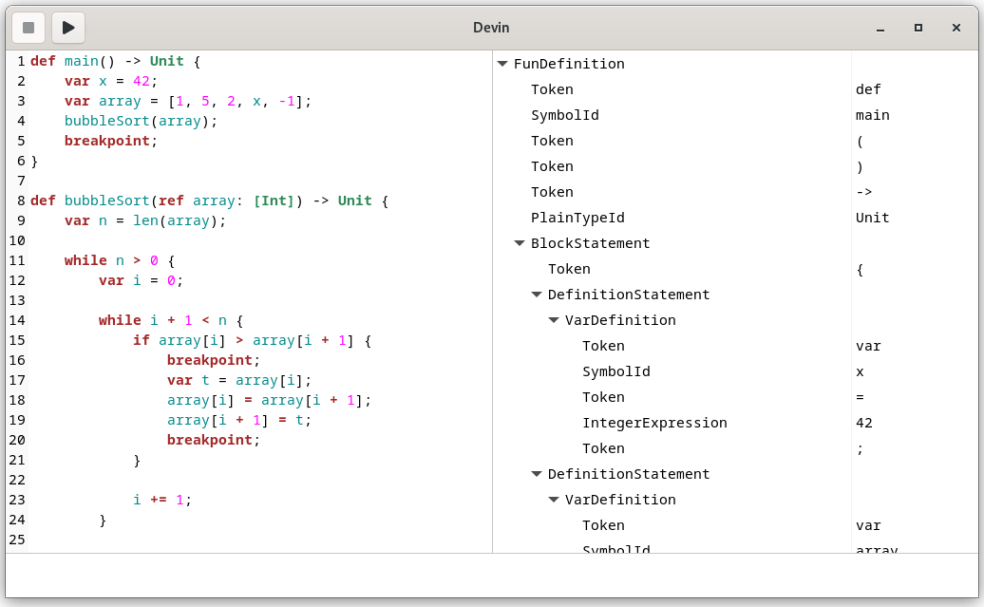

Figure 3.1: The graphical editor associated with Devin

3.1 Language features

Devin is an imperative programming language with lexical variable scoping and support for recursive function definitions. Syntactically, it's similar to any C descendant; among others, it supports variable definition and assignment, while loops, do-while loops, and branching via if and if-else statements.

3.1.1 Data types

Devin features the following data types:

- `Bool`, for the boolean values `true`, `false`.
- `Int`, for 64-bit integers denoted by sequences of digits like `42`.
- `Float`, for double-precision floating-point numbers like `0.0` and `3.14`. The presence of the fractional part distinguishes integers from floats. Contrary to C, exponential notation like `1E100` is not supported.
- `[`T`]`, for arrays like `[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]`.

In addition to the items above, Devin has a unit type named `Unit`; it is inhabited by one value only: `unit`. When a function has side-effects only, it is marked as returning `Unit`.

With unit types, there's no need to distinguish between functions that return a value and procedures that do not: all callables always return a value. One of the advantages of unit types is the uniform treatment of assignments. While unit types are traditionally used in functional programming languages, some recent imperative languages like Kotlin and Rust incorporate this feature as well.

3.1.2 Built-in operators

Booleans support the usual operations of negation, logical conjunction and disjunction. An operator for exclusive disjunction is provided as well.

Boolean binary operations are evaluated lazily: the right operand is only evaluated if the left operand doesn't already determine the result of the whole expression.

Description	Syntax	C equivalent	Semantic meaning
Negation	`not(`p`)`	`!`p	$\neg p$
Conjunction	p `and` q	p `&&` q	$p \wedge q$
Disjunction	p `or` q	p `||` q	$p \vee q$
Exclusive disjunction	p `xor` q	p `!=` q	$(p \wedge \neg q) \vee (\neg p \wedge q)$

Table 3.1: Boolean operators

Integers and floating-point numbers support addition, subtraction, multiplication and division. For integers (but not floats), there's also a modulo operator.

Binary arithmetic operations require both operands to be of the same numeric type: type conversions have to be performed explicitly via `intToFloat(`n`)` and `floatToInt(`x`)`.

Division (operator `/`) behaves differently depending on the types of the operands. In any case, the result of arithmetic operations has the same type as their operands.

Description	Syntax	Semantic meaning
Negation	`-`x	$-x$
Identity	`+`x	$+x$
Addition	x `+` y	$x + y$
Subtraction	x `-` y	$x - y$
Multiplication	x `*` y	$x \times y$
Integer division	n `/` m	$\lfloor n \div m \rfloor$
Floating-point division	x `/` y	$x \div y$
Modulo	n `%` m	$n - m \times \lfloor n \div m \rfloor$

Table 3.2: Arithmetic operators

Values can be compared with the expressions x `<` y, x `<=` y, x `>` y, and x `>=` y. Equality and inequality between any two values can be tested with x `==` y and x `!=` y, respectively.

Description	Syntax	Semantic meaning
Comparison	x `>` y	$x < y$
	x `<=` y	$x \leq y$
	x `>` y	$x > y$
	x `>=` y	$x \geq y$
Equality	x `==` y	$x = y$
	x `!=` y	$x \neq y$

Table 3.3: Relational operators

Variables and array elements can be assigned to using the `=` operator. Like many languages, Devin features a few shorthands as well.

Description	Syntax	Semantic meaning
Plain assignment	x `=` y	$x \leftarrow y$
Assignment shorthand	x `+=` y	x `=` x `+` y
	x `-=` y	x `=` x `-` y
	x `*=` y	x `=` x `*` y
	x `/=` y	x `=` x `/` y
	x `%=` y	x `=` x `%` y

Table 3.4: Assignment operators

3.1.3 Working with arrays

Arrays support the following operations:

- *Element access*: `a[n]`, where a denotes an array and n an integer index. The expression retrieves the n-th element of the array. As in most programming languages, the first element of the array has index 0.
- *Length information*: `len(a)`. This expression returns the number of elements contained in the array a.
- *Concatenation*: a `+` b. This expression yields a new array containing the elements of a concatenated with the elements of b.
- *Repetition*: a `*` n or n `*` a. Equivalent to concatenating a to itself n times. Zero-initialized arrays of size n can be generated with the expression `[0]` `*` n.

3.1.4 Variable definitions and scoping

Variables can be defined using statements of the form `var` x `=` y, where x is an identifier and y is an expression; the type of the new variable x is inferred from y. Variables can be defined at any point, whether globally or as function locals; their visibility extends from the point of their definition onward.

Devin is *block-structured*: it allows for the creation of blocks, including blocks nested within other blocks. Variables are lexically scoped: they can't be accessed from outside the block they are defined in. As in C, a block consists of a sequence of statements wrapped between a pair of curly braces (`{`, `}`). Statements are always terminated with semicolons.

```
def rotateRight(ref array) -> Unit {
    if len(array) > 0 {
        var i = len(array) - 1;
        var t = array[i];

        while i > 0 {
            array[i] = array[i - 1];
            i -= 1;
        }

        array[0] = t;
    }  // 'i', 't' go out of scope
}  // 'array' goes out of scope
```

Listing 3.1: Devin's scoping rules visualized

3.1.5 Branching and looping mechanisms

The following two standard looping constructs are supported:

- *While loops* in the form `while` p s. This construct evaluates p; if it is `true`, s is run. This repeats until p evaluates to `false`.
- *Do-while loops* in the form `do` s `while` p. This construct runs s; this repeats unless a subsequent evaluation of p yields `false`.

Conditional code execution is supported through statements of the form `if` p s_1 and `if` p s_1 `else` s_2, where p is a boolean expression and s_1 and s_2 are statements.

3.1.6 Assertions

Devin provides a mechanism for asserting that predicates evaluate to `true` at runtime; if they do not, an exception is raised, leading to program termination. Assertions can be used to check internal assumptions or to guard against violation of function contracts, among other things.

For example, the assertion `assert n >= 0` can be used as the first statement of a function calculating the factorial of `n`, as the result is not defined if $\texttt{n} < 0$.

3.1.7 Function definition and application

Functions can be defined either globally or within other functions. They adhere to the same scoping rules as variables. The entry point for Devin programs is a global function named `main`; it must take no arguments.

The syntax for calling functions is the same as in C: f`()` calls f with zero arguments, g`(`x`)` calls g with a single argument, h`(`x`, `y`)` calls h with two arguments, and so on.

The evaluation strategy of Devin is eager; arguments are evaluated from left to right. By default, function arguments are passed by value; with the `ref` keyword, they can be passed by reference instead (see Listing 3.2).

3.1.8 Optional types

A peculiar feature of Devin is that of *optional types*. With optional types, the semantics of the language doesn't depend on the static type system [7]. A notable example of an optionally typed language is Python. With this feature, type annotations can be omitted from Devin programs. Type checking is not performed on terms where type annotations are missing.

```
def swap1(ref array: [Int], i: Int, j: Int) -> Unit {
    var t = array[i];
    array[i] = array[j];
    array[j] = t;
}

// Note: 'swap2' is more general than 'swap1', as it can be used with any array
def swap2(ref array, i, j) {
    var t = array[i];
    array[i] = array[j];
    array[j] = t;
}
```

Listing 3.2: Optional types exemplified

3.2 Editor features

Devin comes with a code editing UI built on GTK+, a popular library for creating graphical user interfaces. GTK+ is compatible with Windows, macOS and many UNIX-like platforms [8]. Code editing is facilitated by GtkSourceView, a library that extends the GTK+ framework to support text editing and configurable syntax highlighting [9].

In addition to syntax highlighting, Devin's editor features error reporting: both syntactic and semantic errors are displayed according to their position in the source code.

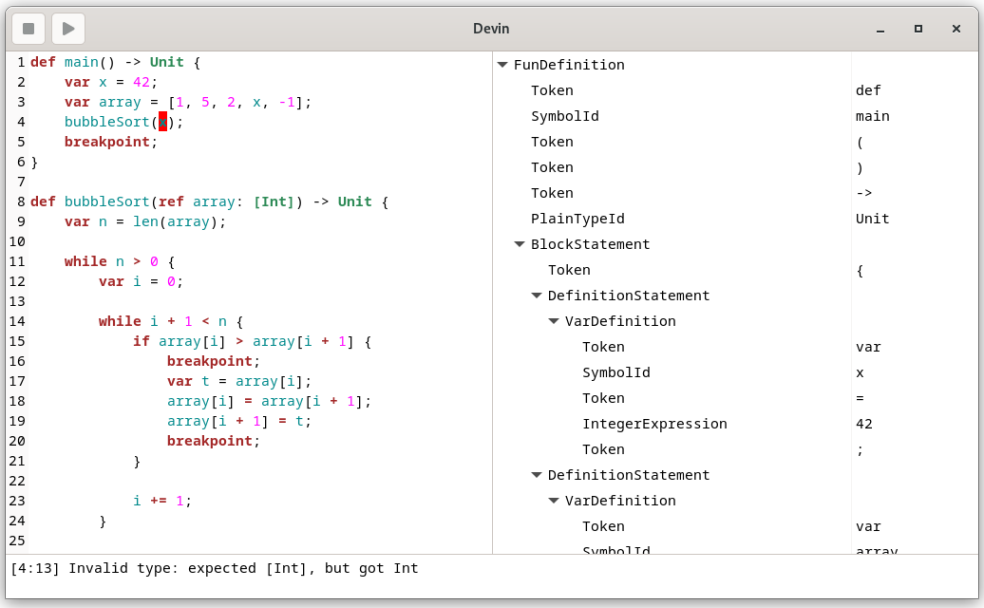

Figure 3.2: Semantic error reporting and highlighting

Once a valid program is written, it can be executed by clicking the ▶ button. The execution state can be observed with breakpoint statements; these instruct the evaluator to pause execution and display the current runtime stack, as seen in Figure 3.3. The stack, which maps variables and function arguments to their values, can be viewed from the user interface. Each function call pushes a new frame is onto the stack, containing associations between parameter names and their passed values. Variable definitions within the called function add new associations to the newly pushed frame. When the called function returns, the topmost frame is popped off the stack and execution continues normally.

Devin has been designed with simplicity of implementation in mind. In particular, Devin's call stack exhibits two properties:

- Each block pushes a new frame onto the stack: this allows treating nested scopes with the same mechanism as function calls. The debugger hides this detail by displaying what appears to be a single frame for each call.
- Program execution always starts with a non-empty stack containing a single frame. This frame associates the name `true` to the truth value, `false` to `not true`, and `unit` to the unit value.

  In practice, this means that `true`, `false` and `unit` are not special keywords or constants; instead, they are ordinary variables. This is a deliberate design decision.

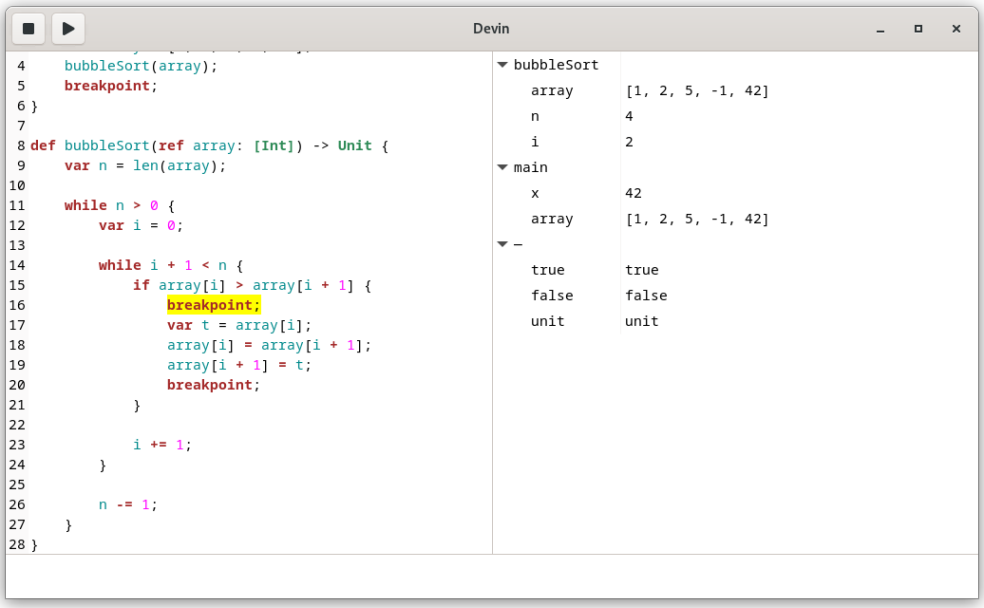

Figure 3.3: The debugger displaying the current state of the stack

Runtime errors are handled as well. When an error occurs, the offending code fragment is highlighted and a descriptive message dialog is shown to the user. Runtime errors include, but are not limited to, index out of bounds errors, division by zero, assertion violations.

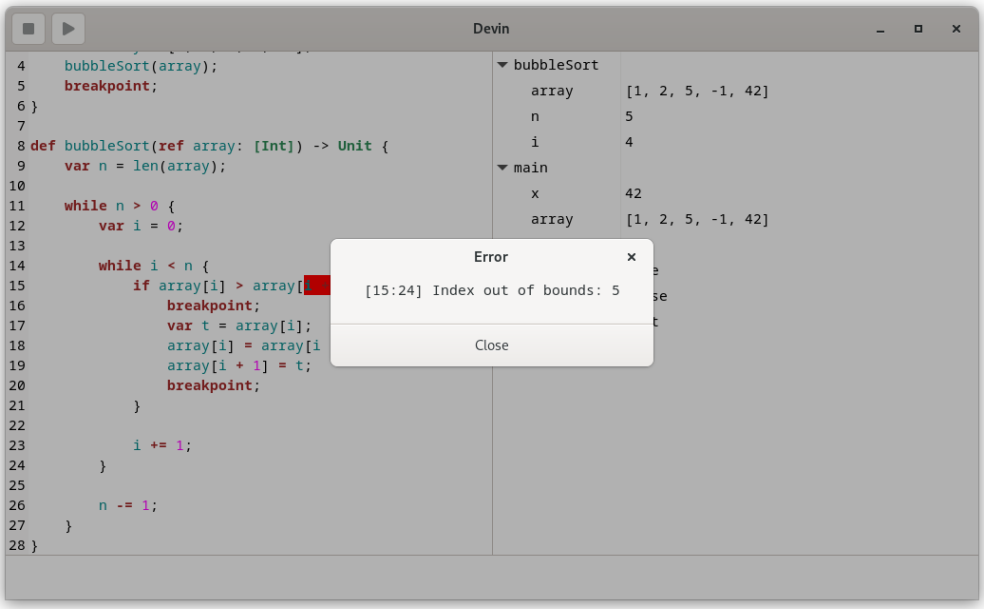

Figure 3.4: Runtime error reporting and highlighting

3.3 Devin's design choices

3.3.1 Operations between numeric types

Addition, subtraction, multiplication, division and modulo are supported only if both operands are of the same numeric type. For instance, it isn't legal to add a floating-point number to an integer. While this restriction might seem unconventional at first, there is a very specific reason for it.

In Devin, both integers and floating-point numbers occupy 64 bits. Adding two numbers of the same type T yields yet another number of type T: nothing surprising here. But what should be the result of adding an integer to a floating-point number, or vice versa? Many languages take the approach of silently converting the integer operand to a floating-point value, and then performing the actual addition. Double-precision floating-point numbers as specified in IEEE 754 can't safely represent integers less than $-\left(2^{53}-1\right)$ or greater than $2^{53}-1$ without loss of precision [10]; as such, I argue that a programming language should make conversions between integers and floats explicit.

This design decision is largely inspired by how Haskell deals with numbers. Examples of other languages which lack implicit conversions are Go [11] and Rust [12].

3.3.2 Optional types

As discussed previously, type annotations may be omitted at will. As Devin's type checker and evaluator are independent, optional types *emerged* as a possible additional feature.

Much can be said regarding the benefits and drawbacks of optional types. Devin was born due to my own curiosity regarding programming languages and their implementation; recreating the next Pascal or C clone is not an objective. As optional types are not a common feature, and as languages shape the way we think about solving problems, I decided to add this feature to be played around with.

4

Implementing Devin

Devin's implementation spans almost 3700 lines of code and is entirely written in Haskell. Given the size of its implementation, some details are omitted for brevity. In particular, import declarations and `LANGUAGE` pragmas are left out for simplicity. Note that some of the functionality of the package `extra` is used: the library provides extra functions for the standard Haskell libraries, filling out missing functionality [13]. Devin's full implementation can be found in the appendix.

In terms of implementation complexity, both the type checker and the evaluator are somewhat simpler than the parser. The parser makes much greater use of higher-order functions and uses multi-parameter type classes, causing some syntactical noise on function signatures; for this reason, I'll describe parsing last.

4.1 The syntax tree

Devin's syntax tree stores enough information to perform syntax highlighting. Given a node in the tree, it is always possible to determine its position within the parsed source code. Only leaf nodes store their positions directly; the positions of non-leaf nodes can be computed by considering their first and last children.

Two leaf nodes commonly used across Devin's syntax tree are `SymbolId` and `Token`. They store information about identifiers (e.g., `x`, `Int`) and tokens (e.g., `{`, `}`, `->`), respectively. They are defined as follows:

```
data SymbolId = SymbolId { name :: String, interval :: (Int, Int) }

data Token = Token { interval :: (Int, Int) }
```

4.1.1 Representing expressions

Literals, numbers and variables are the simplest kinds of expressions, as they are leaves in the syntax tree. More complex expressions can be built from simpler ones: for example, a binary expression is constructed by two sub-expressions and a binary operator.

The pair of unary operators `+` and `-`, along with the 20 binary operators `+`, `-`, `*`, `/`, `%`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `and`, `or`, `xor`, `=`, `+=`, `-=`, `*=`, `/=`, `%=` are represented by the algebraic data types `UnaryOperator` and `BinaryOperator`, respectively:

```
data UnaryOperator
  = PlusOperator { interval :: (Int, Int) }
  | MinusOperator { interval :: (Int, Int) }

data BinaryOperator
  = AddOperator { interval :: (Int, Int) }
  | SubtractOperator { interval :: (Int, Int) }
  | MultiplyOperator { interval :: (Int, Int) }
  | DivideOperator { interval :: (Int, Int) }
  | ModuloOperator { interval :: (Int, Int) }
  | EqualOperator { interval :: (Int, Int) }
  | NotEqualOperator { interval :: (Int, Int) }
  | LessOperator { interval :: (Int, Int) }
  | LessOrEqualOperator { interval :: (Int, Int) }
  | GreaterOperator { interval :: (Int, Int) }
  | GreaterOrEqualOperator { interval :: (Int, Int) }
  | AndOperator { interval :: (Int, Int) }
  | OrOperator { interval :: (Int, Int) }
  | XorOperator { interval :: (Int, Int) }
  | PlainAssignOperator { interval :: (Int, Int) }
  | AddAssignOperator { interval :: (Int, Int) }
  | SubtractAssignOperator { interval :: (Int, Int) }
  | MultiplyAssignOperator { interval :: (Int, Int) }
  | DivideAssignOperator { interval :: (Int, Int) }
  | ModuloAssignOperator { interval :: (Int, Int) }
```

Expressions are represented by a single algebraic data type: `Expression`. Opening and closing parentheses are stored in the syntax tree. As a convention, parentheses are always identified by the `open` and `close` fields, regardless of the kind: round, square, or curly.

```
data Expression where
  VarExpression :: {
    varName :: String,
    interval :: (Int, Int)
  } -> Expression

  IntegerExpression :: {
    integer :: Integer,
    interval :: (Int, Int)
  } -> Expression
```

```
RationalExpression :: {
  rational :: Rational,
  interval :: (Int, Int)
} -> Expression

ArrayExpression :: {
  open :: Token,
  elems :: [Expression],
  commas :: [Token],
  close :: Token
} -> Expression

AccessExpression :: {
  array :: Expression,
  open :: Token,
  index :: Expression,
  close :: Token
} -> Expression

CallExpression :: {
  funId :: SymbolId,
  open :: Token,
  args :: [Expression],
  commas :: [Token],
  close :: Token
} -> Expression

UnaryExpression :: {
  unary :: UnaryOperator,
  operand :: Expression
} -> Expression

BinaryExpression :: {
  left :: Expression,
  binary :: BinaryOperator,
  right :: Expression
} -> Expression

ParenthesizedExpression :: {
  open :: Token,
  inner :: Expression,
  close :: Token
} -> Expression
```

4.1.2 Representing statements

Any expression followed by a semicolon is a statement. These expression statements are represented by the algebraic data type `Statement`, along with if, if-else, while, do-while, return, assert, breakpoint, and block statements.

At any point in which a statement can occur, a variable or function definition may be placed instead. The data constructor `DefinitionStatement` represents this idea.

```
data Statement where
  DefinitionStatement :: {
    definition :: Definition
  } -> Statement

  ExpressionStatement :: {
    effect :: Expression,
    semicolon :: Token
  } -> Statement

  IfStatement :: {
    ifKeyword :: Token,
    predicate :: Expression,
    trueBranch :: Statement
  } -> Statement

  IfElseStatement :: {
    ifKeyword :: Token,
    predicate :: Expression,
    trueBranch :: Statement,
    elseKeyword :: Token,
    falseBranch :: Statement
  } -> Statement

  WhileStatement :: {
    whileKeyword :: Token,
    predicate :: Expression,
    body :: Statement
  } -> Statement

  DoWhileStatement :: {
    doKeyword :: Token,
    body :: Statement,
    whileKeyword :: Token,
    predicate :: Expression,
    semicolon :: Token
  } -> Statement

  ReturnStatement :: {
    returnKeyword :: Token,
    result :: Maybe Expression,
    semicolon :: Token
  } -> Statement

  AssertStatement :: {
    assertKeyword :: Token,
    predicate :: Expression,
    semicolon :: Token
  } -> Statement
```

```
  BreakpointStatement :: {
    breakpointKeyword :: Token,
    semicolon :: Token
  } -> Statement

  BlockStatement :: {
    open :: Token,
    statements :: [Statement],
    close :: Token
  } -> Statement
```

4.1.3 Representing variable and function definitions

Devin's function definitions may or may not include signatures with explicit types. The syntax for array types involves a type enclosed in square brackets, as in `[Int]` or `[[Float]]`; the arbitrary syntactical nesting of paired square brackets around a type is permitted by the algebraic data type `TypeId`:

```
data TypeId
  = PlainTypeId { name :: String, interval :: (Int, Int) }
  | ArrayTypeId { open :: Token, innerTypeId :: TypeId, close :: Token }
```

Variables and function definitions are represented by `VarDefinition` and `FunDefinition`:

```
data Definition where
  VarDefinition :: {
    varKeyword :: Token,
    varId :: SymbolId,
    equalSign :: Token,
    value :: Expression,
    semicolon :: Token
  } -> Definition

  FunDefinition :: {
    defKeyword :: Token,
    funId :: SymbolId,
    open :: Token,
    params :: [(Maybe Token, SymbolId, Maybe (Token, TypeId))],
    commas :: [Token],
    close :: Token,
    returnInfo :: Maybe (Token, TypeId),
    body :: Statement
  } -> Definition
```

4.2 Representing semantic errors

Both the evaluator and the type checker benefit from having a data type to represent errors. Runtime errors, such as division by zero, are raised by the evaluator. Static

analysis errors, like missing return statements, are reported by the type checker. Since some errors are common to both phases, a single algebraic data type is used:

```
data Error where
  UnknownVar :: {
    varName :: String,
    interval :: (Int, Int)
  } -> Error

  UnknownFun :: {
    funName :: String,
    interval :: (Int, Int)
  } -> Error

  UnknownType :: {
    typeName :: String,
    interval :: (Int, Int)
  } -> Error

  InvalidUnary :: {
    unary :: UnaryOperator,
    operandT :: Type
  } -> Error

  InvalidBinary :: {
    binary :: BinaryOperator,
    leftT :: Type,
    rightT :: Type
  } -> Error

  InvalidType :: {
    expression :: Expression,
    expectedT :: Type,
    actualT :: Type
  } -> Error

  -- Static errors:

  MissingReturnValue :: {
    statement :: Statement,
    expectedT :: Type
  } -> Error

  MissingReturnStatement :: {
    funId :: SymbolId
  } -> Error

  -- Runtime errors:

  IntegerOverflow :: {
    expression :: Expression
  } -> Error
```

```
  DivisionByZero :: {
    expression :: Expression
  } -> Error

  IndexOutOfBounds :: {
    expression :: Expression,
    value :: Int64
  } -> Error

  InvalidArgCount :: {
    expression :: Expression,
    expected :: Int,
    actual :: Int
  } -> Error

  AssertionFailed :: {
    statement :: Statement
  } -> Error
```

4.3 The type checker

Type checking is performed by a set of functions that operate on an *environment*: a data structure associating identifiers with types. The environment is used to verify if function calls and variable usages are valid within their context.

The module `Devin.Type` provides the necessary data types to represent types themselves. The data constructors `Unit`, `Bool`, `Int`, `Float` and `Array` correspond to Devin's supported types. To aid the type checker, two other data constructors are provided, both indicating some sort of error: `Unknown` and `Placeholder`. While `Unknown` stands for an unknown type, `Placeholder` represents a specific unresolved type. Differentiating between `Placeholder` and `Unknown` allows for fine-grained error messages and diagnostics. For instance, if a misspelled type occurs the same way n times within some scope, it corresponds to *one*, instead of n, type errors.

```
data Type
  = Unknown
  | Unit
  | Bool
  | Int
  | Float
  | Array Type
  | Placeholder String
```

To compare types, the relation operator `(<:)` is provided:

```
(<:) :: Type -> Type -> Bool
t1 <: t2 = isJust (merge t1 t2)
```

```
merge :: Type -> Type -> Maybe Type
merge Unknown _ = Just Unknown
merge _ Unknown = Just Unknown
merge Unit Unit = Just Unit
merge Bool Bool = Just Bool
merge Int Int = Just Int
merge Float Float = Just Float
merge (Array t1) (Array t2) = Array <$> merge t1 t2
merge (Placeholder n1) (Placeholder n2) | n1 == n2 = Just (Placeholder n1)
merge _ _ = Nothing
```

Given two types, the helper function `merge` yields a type that is compatible with both, if possible. All types except for `Unknown` are only compatible with themselves; `Unknown`, on the other hand, is compatible with all other types.

The module `Devin.Typer` implements the type checker interface. To account for Devin's lexical scoping rules, the environment is represented as a list of association lists.

```
data Scope = Scope {
  types :: [(String, Type)],
  funs :: [(String, ([Type], Type))],
  vars :: [(String, Type)]
}

type Environment = [Scope]

data Typer a = Typer { runTyper :: Environment -> (a, Environment, [Error]) }
  deriving Functor

predefinedEnv :: Environment
predefinedEnv =
  let t1 = ("Unit", Unit)
      t2 = ("Bool", Bool)
      t3 = ("Int", Int)
      t4 = ("Float", Float)

      f1 = ("not", ([Bool], Bool))
      f2 = ("len", ([Array Unknown], Int))
      f3 = ("intToFloat", ([Int], Float))
      f4 = ("floatToInt", ([Float], Int))

      v1 = ("true", Bool)
      v2 = ("false", Bool)
      v3 = ("unit", Unit)

   in [Scope [t4, t3, t2, t1] [f4, f3, f2, f1] [v3, v2, v1]]
```

In the listing above, `Typer` wraps a function that takes an `Environment` and returns a new environment and a potentially empty list of `Error`s, along with a result of some kind. `deriving Functor` instructs GHC to automatically generate an implementation for the function `fmap :: (a -> b) -> Typer a -> Typer b`.

Below, implementations for `pure`, `liftA2`, `(>>=)` are provided. These functions, in conjunction with `fmap`, render `Typer` a monad.

```
instance Applicative Typer where
  pure :: a -> Typer a
  pure x = Typer (\env -> (x, env, []))

  liftA2 :: (a -> b -> c) -> Typer a -> Typer b -> Typer c
  liftA2 f mx my = Typer $ \env ->
    let (x, env', errors1) = runTyper mx env
        (y, env'', errors2) = runTyper my env'
     in (f x y, env'', errors1 ++ errors2)

instance Monad Typer where
  (>>=) :: Typer a -> (a -> Typer b) -> Typer b
  mx >>= f = Typer $ \env ->
    let (x, env', errors1) = runTyper mx env
        (y, env'', errors2) = runTyper (f x) env'
     in (y, env'', errors1 ++ errors2)
```

Some utility functions are defined. `defineType` binds a type name (such as `"Unit"`) to a `Type` (such as `Unit`); the binding is added to the current environment. Its counterpart, `lookupType`, looks up the type to which a name is bound. `lookupType` searches the current environment; if no binding is found, the parent environment is scanned next.

```
defineType :: String -> Type -> Typer Type
defineType name t = Typer $ \case
  [] -> (t, [Scope [(name, t)] [] []], [])

  scope : scopes ->
    let types' = (name, t) : types scope
     in (t, scope{types = types'} : scopes, [])

lookupType :: String -> Typer (Maybe (Type, Int))
lookupType name = Typer (\env -> (go 0 env, env, []))
  where
    go _ [] = Nothing

    go depth (Scope{types} : scopes) = case lookup name types of
      Just t -> Just (t, depth)
      Nothing -> go (depth + 1) scopes
```

An expression of the form `\case { ` p_1 `-> ` e_1`; ...; ` p_n `-> ` e_n ` }` is called a *lambda-case*; it is equivalent to `\`x `-> case ` x ` of { ` p_1 `-> ` e_1`; ...; ` p_n `-> ` e_n ` }`, where x is a fresh variable. Lambda-case syntax is enabled by the `LambdaCase` language extension.

Next, the functions `defineFunSignature` and `lookupFunSignature` define and lookup associations between names and function signatures, represented by a `([Type], Type)`. The first element of the tuple is the list of parameter types, the second is the return type.

```
defineFunSignature :: String -> ([Type], Type) -> Typer ()
defineFunSignature name signature = Typer $ \case
  [] -> ((), [Scope [] [(name, signature)] []], [])

  scope : scopes ->
    let funs' = (name, signature) : funs scope
     in ((), scope{funs = funs'} : scopes, [])

lookupFunSignature :: String -> Typer (Maybe (([Type], Type), Int))
lookupFunSignature name = Typer (\env -> (go 0 env, env, []))
  where
    go _ [] = Nothing

    go depth (Scope{funs} : scopes) = case lookup name funs of
      Just signature -> Just (signature, depth)
      Nothing -> go (depth + 1) scopes
```

Finally, variables can be defined and looked up with `defineVarType` and `lookupVarType`.

```
defineVarType :: String -> Type -> Typer ()
defineVarType name t = Typer $ \case
  [] -> ((), [Scope [] [] [(name, t)]], [])

  scope : scopes ->
    let vars' = (name, t) : vars scope
     in ((), scope{vars = vars'} : scopes, [])

lookupVarType :: String -> Typer (Maybe (Type, Int))
lookupVarType name = Typer (\env -> (go 0 env, env, []))
  where
    go _ [] = Nothing

    go depth (Scope{vars} : scopes) = case lookup name vars of
      Just t -> Just (t, depth)
      Nothing -> go (depth + 1) scopes
```

Functions for removing previously defined associations could be implemented; however, this is not necessary. Instead, the function `withNewScope` is provided to run another `Typer` in a modified environment where an empty `Scope` is added; when the function returns, the previous environment is restored.

```
withNewScope :: Typer a -> Typer a
withNewScope mx = Typer $ \env ->
  let (x, env', errors) = runTyper mx (Scope [] [] [] : env)
   in (x, tail env', errors)
```

One last function is provided for utility: `report`. As its name suggests, this function reports some `Error`; errors are accumulated by the `Typer` monad and can be used in successive phases for error reporting and highlighting.

```
report :: Error -> Typer ()
report error = Typer (\env -> ((), env, [error]))
```

4.3.1 Checking expressions

Expression-related type checking is implemented in the `Devin.Typers` module with the `checkExpression :: Expression -> Typer Type` function.

Devin has a rudimentary mechanism for type inference. The type of literal expressions is trivially determined by the kind of literal. For variables, a lookup in the current environment is performed; if the variable is not bound, an error is reported.

```
checkExpression IntegerExpression{} = pure Int

checkExpression RationalExpression{} = pure Float

checkExpression VarExpression{varName, interval} =
  lookupVarType varName >>= \case
    Just (t, _) -> pure t

    Nothing -> do
      report (UnknownVar varName interval)
      pure Unknown
```

For simplicity, the function `report'` is provided for future use. It is similar to `report`, but its result wraps `Unknown` instead of the unit type `()`.

```
report' :: Error -> Typer Type
report' error = do
  report error
  pure Unknown
```

Inferring the type of arrays requires special care. In general, the type of an array is represented by `Array` T. If all elements of the array have the same type T_0, then T can be determined to be T_0. If there are at least two elements of the array with different types, then T can't be inferred.

```
checkExpression ArrayExpression{elems = elem : elems} = do
  t <- checkExpression elem

  flip loopM elems $ \case
    [] -> pure (Right (Array t))

    elem : elems -> checkExpression elem >>= \case
      Unknown -> do
        for_ elems checkExpression
        pure (Right (Array Unknown))

      t' | t' <: t -> pure (Left elems)

      t' -> do
        report (InvalidType elem t t')
        pure (Left elems)
```

The higher-order function `loopM :: Monad m => (a -> m (Either a b)) -> a -> m b` expresses a monadic loop operation where the predicate returns `Left` as a seed for the next loop, or `Right` to abort the loop. To apply `loopM` with flipped argument order, `flip :: (a -> b -> c) -> b -> a -> c` is used.

As a special case, the type of an array can't be inferred when it is empty. To allow empty arrays to be used, no error is reported in such instance.

```
checkExpression ArrayExpression{elems = []} = pure (Array Unknown)
```

Access to array elements is checked as follows:

```
checkExpression AccessExpression{array, index} = do
  arrayT <- checkExpression array
  indexT <- checkExpression index

  case (arrayT, indexT) of
    (Unknown, Unknown) -> pure Unknown
    (Unknown, Int) -> pure Unknown
    (Unknown, _) -> report' (InvalidType index Int indexT)
    (Array t, Unknown) -> pure t
    (Array t, Int) -> pure t
    (Array _, _) -> report' (InvalidType index Int indexT)
    (_, _) -> report' (InvalidType array (Array Unknown) arrayT)
```

Unary arithmetic expressions with the operators `+` and `-` are type checked by ensuring that the operand is of a numeric type. If it is, the result of evaluating the whole expression has the same type as the operand.

```
checkExpression UnaryExpression{unary, operand}
  | PlusOperator{} <- unary = do
    operandT <- checkExpression operand

    case operandT of
      Unknown -> pure Unknown
      Int -> pure Int
      Float -> pure Float
      _ -> report' (InvalidUnary unary operandT)

checkExpression UnaryExpression{unary, operand}
  | MinusOperator{} <- unary = do
    operandT <- checkExpression operand

    case operandT of
      Unknown -> pure Unknown
      Int -> pure Int
      Float -> pure Float
      _ -> report' (InvalidUnary unary operandT)
```

There is a certain degree of code repetition within Devin's type checker. While *DRY* (Don't Repeat Yourself) is generally a good software development practice, it does be-

come impractical when edge cases have to be considered. For instance, while both binary operators `+` and `-` operate on arithmetic types, `+` also works with arrays. Worse, in the evaluator, both `+` and `and` compute some result based on their operands, but the evaluation strategy of `and` differs from that of `+`. Source code could certainly be rearranged to avoid some amount of repetition; however, it would become less readable to some extent.

What follows is the implementation to type check expressions with the binary operator `/`. Both operands have to be of the same type; if they are, the result of evaluating the whole expression has the same type as the operands.

```
checkExpression BinaryExpression{left, binary, right}
  | DivideOperator{} <- binary = do
    leftT <- checkExpression left
    rightT <- checkExpression right

    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
      (Int, Int) -> pure Int
      (Float, Float) -> pure Float
      (_, _) -> report' (InvalidBinary binary leftT rightT)
```

The binary operators `-` and `%` are type checked similarly to `/`, except for the fact that `%` only accepts operands of type `Int`.

The semantic meaning of the binary operator `+` depends on the type of the operands. If both are numeric, the operator denotes arithmetic addition; if the operands are arrays, `+` stands for array concatenation. Similarly, `*` could indicate both numeric multiplication and array repetition. As these operators are overloaded, all possibilities have to be considered during type checking.

```
checkExpression BinaryExpression{left, binary, right}
  | AddOperator{} <- binary = do
    leftT <- checkExpression left
    rightT <- checkExpression right

    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
      (Int, Int) -> pure Int
      (Float, Float) -> pure Float
      (Array t1, Array t2) | Just t <- merge t1 t2 -> pure (Array t)
      (_, _) -> report' (InvalidBinary binary leftT rightT)

checkExpression BinaryExpression{left, binary, right}
  | MultiplyOperator{} <- binary = do
    leftT <- checkExpression left
    rightT <- checkExpression right
```

```
    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
      (Int, Int) -> pure Int
      (Float, Float) -> pure Float
      (Array t, Int) -> pure (Array t)
      (Int, Array t) -> pure (Array t)
      (_, _) -> report' (InvalidBinary binary leftT rightT)
```

Values can be compared with the binary operators `==`, `!=`, `<`, `<=`, `>` and `>=`. Type checking ensures that both operands are of the same type; if they are, expressions with such operators yield a `Bool`; as we'll see, the semantic meaning of comparing non-numeric values is lexicographical ordering. Type checking is implemented in the same way for all six comparison operators:

```
checkExpression BinaryExpression{left, binary, right}
  | LessOperator{} <- binary = do
    leftT <- checkExpression left
    rightT <- checkExpression right

    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
      (_, _) | Just _ <- merge leftT rightT -> pure Bool
      (_, _) -> report' (InvalidBinary binary leftT rightT)
```

The operators `and`, `or` and `xor` are all type checked in the same way:

```
checkExpression BinaryExpression{left, binary, right}
  | AndOperator{} <- binary = do
    leftT <- checkExpression left
    rightT <- checkExpression right

    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
      (Bool, Bool) -> pure Bool
      (_, _) -> report' (InvalidBinary binary leftT rightT)
```

Plain assignment expressions with `=` are checked by verifying that both the left and right operands are of the same type:

```
checkExpression BinaryExpression{left, binary, right}
  | PlainAssignOperator{} <- binary = do
    leftT <- checkExpression left
    rightT <- checkExpression right

    if rightT <: leftT then
      pure rightT
    else
      report' (InvalidBinary binary leftT rightT)
```

Expressions with the binary operators `+=`, `-=`, `*=`, `/=` and `%=` are type checked considering the shorthands detailed in Table 3.4.

The type resulting from function invocations is determined by looking up the callee's signature. If a call refers to an unbound identifier, or if the wrong number of arguments is provided, or if an argument has the wrong type, an error is reported.

```
checkExpression expression@CallExpression{funId, args} = do
  let SymbolId{name, interval} = funId

  lookupFunSignature name >>= \case
    Nothing -> report' (UnknownFun name interval)

    Just ((paramTs, returnT), _) -> go 0 args paramTs
      where
        go _ [] [] = pure returnT

        go n (arg : args) (paramT : paramTs) = do
          argT <- checkExpression arg
          unless (argT <: paramT) (report (InvalidType arg paramT argT))
          go (n + 1) args paramTs

        go n args paramTs = do
          let expected = n + length paramTs
          let actual = n + length args
          report' (InvalidArgCount expression expected actual)
```

4.3.2 Checking statements

The function `checkStatement :: Type -> Statement -> Typer Bool` type checks statements. A type must be provided: on return statements, `checkStatement` verifies that the returned value is of the given type.

Inside functions, return statements may signal control to be `return`ed to the callee. The function `checkStatement` wraps a `Bool` which is `True` if control is guaranteed to be returned to the callee, or `False` otherwise.

Expression and definition statements are trivial to check. In both cases, type checking is delegated to the relevant function. `checkDefinitions` is defined in the next section.

```
checkStatement _ ExpressionStatement{effect} = do
  checkExpression effect
  pure False

checkStatement _ DefinitionStatement{definition} = do
  checkDefinitions [definition]
  pure False
```

For breakpoint statements, nothing needs to be done:

```
checkStatement _ BreakpointStatement{} = pure False
```

If-else statements are checked by ensuring that the predicate is a boolean expression and by recursively type checking both potential execution paths. We adopt a conservative approach, verifying if both branches return control; if so, the whole if-else statement surely must return control as well.

```
checkStatement expectedT IfElseStatement{predicate, trueBranch, falseBranch} = do
  t <- checkExpression predicate
  unless (t <: Bool) (report (InvalidType predicate Bool t))
  trueBranchDoesReturn <- withNewScope (checkStatement expectedT trueBranch)
  falseBranchDoesReturn <- withNewScope (checkStatement expectedT falseBranch)
  pure (trueBranchDoesReturn && falseBranchDoesReturn)
```

For if statements without an `else` clause, we follow the same conservative approach: since control isn't guaranteed to be returned to the callee, `False` is returned.

```
checkStatement expectedT IfStatement{predicate, trueBranch} = do
  t <- checkExpression predicate
  unless (t <: Bool) (report (InvalidType predicate Bool t))
  withNewScope (checkStatement expectedT trueBranch)
  pure False
```

Both while and do-while statements are type checked with considerations similar to those for if and if-else statements, respectively.

```
checkStatement expectedT WhileStatement{predicate, body} = do
  t <- checkExpression predicate
  unless (t <: Bool) (report (InvalidType predicate Bool t))
  withNewScope (checkStatement expectedT body)
  pure False

checkStatement expectedT DoWhileStatement{body, predicate} = do
  doesReturn <- withNewScope (checkStatement expectedT body)
  t <- checkExpression predicate
  unless (t <: Bool) (report (InvalidType predicate Bool t))
  pure doesReturn
```

Return statements are checked by ensuring that the returned value is of the correct type. If the return value is omitted, it is assumed that `unit` is returned instead.

```
checkStatement expectedT ReturnStatement{result = Just result} = do
  t <- checkExpression result
  unless (t <: expectedT) (report (InvalidType result expectedT t))
  pure True

checkStatement expectedT statement@ReturnStatement{result = Nothing} = do
  unless (Unit <: expectedT) (report (MissingReturnValue statement expectedT))
  pure True
```

Assertions are type checked by verifying whether the predicate is of type `Bool`.

```
checkStatement _ AssertStatement{predicate} = do
  t <- checkExpression predicate
  unless (t <: Bool) (report (InvalidType predicate Bool t))
  pure False
```

Finally, blocks are handled by recursively type checking all statements they contain. It is deduced that a block returns control if at least one of its statements does so.

Devin's syntactical scoping rules have to be considered: declarations in a block have to be discarded at the end of its scope. Thus, statements and declarations within blocks must be type checked in a new temporary environment, using the `withNewScope` function.

```
checkStatement expectedT BlockStatement{statements} =
  withNewScope $ do
    for_ statements $ \case
      DefinitionStatement{definition} -> checkDefinition1 definition
      _ -> pure ()

    foldlM f False statements

  where
    f result DefinitionStatement{definition} = do
      checkDefinition2 definition
      pure result

    f result statement = do
      doesReturn <- checkStatement expectedT statement
      pure (result || doesReturn)
```

`foldlM :: (Foldable t, Monad m) => (b -> a -> m b) -> b -> t a -> m b` is an analogous to `foldl` (which is a left-associative version of `foldr` (section 1.7)), except that its result is encapsulated in a monad.

4.3.3 Checking variable and function definitions

Given a block, definitions are type checked using a two-pass strategy. The first pass searches for function definitions and stores the relevant signatures in the environment; the body of the functions is not checked at this stage. The second pass processes variable definitions and statements within function definitions. Without two-pass type checking, mutually recursive function definitions could not be supported, at least not without a mechanism like C's forward declarations.

```
checkDefinitions :: [Definition] -> Typer ()
checkDefinitions definitions = do
  for_ definitions checkDefinition1
  for_ definitions checkDefinition2
```

In the first pass, function signatures are processed by resolving the parameter and return types, if specified. Type identifiers are mapped to types with the `getType` helper function.

```
checkDefinition1 :: Definition -> Typer ()
checkDefinition1 = \case
  VarDefinition{} -> pure ()

  FunDefinition{funId = SymbolId{name}, params, returnInfo} -> do
    paramTs <- for params $ \(_, _, typeInfo) -> case typeInfo of
      Just (_, paramTypeId) -> getType paramTypeId
      Nothing -> pure Unknown

    returnT <- case returnInfo of
      Just (_, returnTypeId) -> getType returnTypeId
      Nothing -> pure Unknown

    defineFunSignature name (paramTs, returnT)

getType :: TypeId -> Typer Type
getType = \case
  PlainTypeId{name, interval} -> lookupType name >>= \case
    Just (t, _) -> pure t

    Nothing -> do
      report (UnknownType name interval)
      defineType name (Placeholder name)

  ArrayTypeId{innerTypeId} -> do
    t <- getType innerTypeId
    pure (Array t)
```

In the second pass, the function `checkDefinition2 :: Definition -> Typer ()` type checks variable and function definitions, adding new bindings to the environment.

For variable definitions, implementation is straightforward:

```
checkDefinition2 VarDefinition{varId = SymbolId{name}, value} = do
  t <- checkExpression value
  defineVarType name t
```

To type check function definitions, a new scope is introduced; this way, bindings for parameter names are local to the function which introduced them.

For each function definition, type checking is done in three steps:

1. Parameter names are bound to their type;
2. The return type is determined from the function signature. If no return type was provided, `Unknown` is assumed;
3. The body of the function is type checked. If the return type isn't `Unit` or `Unknown`, it is further verified that execution paths return a value.

```
checkDefinition2 FunDefinition{funId, params, returnInfo, body} =
  withNewScope $ do
    for_ params $ \(_, SymbolId{name}, typeInfo) -> case typeInfo of
      Just (_, paramTypeId) -> do
        paramT <- getType paramTypeId
        defineVarType name paramT

      Nothing -> defineVarType name Unknown

    returnT <- case returnInfo of
      Just (_, returnTypeId) -> getType returnTypeId
      Nothing -> pure Unknown

    case returnT of
      Unknown -> void (checkStatement Unknown body)
      Unit -> void (checkStatement Unit body)

      _ -> do
        doesReturn <- checkStatement returnT body
        unless doesReturn (report (MissingReturnStatement funId))
```

4.4 The evaluator

The data types and functions defined in the `Devin.Evaluator` module implement the evaluator interface. Evaluation operates on a linked list of *frames*; each frame holds bindings between function names and their definitions, and between variable names and their value stored in memory.

```
data Frame = Frame {
  poffset :: Int,  -- Static link
  funs :: [(String, Function)],
  vars :: [(String, Cell)]
}

type State = [Frame]
```

A `Function` can either be user-defined, or a Devin built-in with ad-hoc semantics.

```
data Function
  = BuiltinNot
  | BuiltinLen
  | BuiltinIntToFloat
  | BuiltinFloatToInt
  | UserDefined Definition
```

Values are not stored directly inside `State`; rather, each identifier is bound to a *cell* which holds the actual value. This way, multiple identifiers can refer to the same value. Devin's `Cell` data type wraps Haskell's `IORef`, which provides mutable references in the `IO` monad. Devin arrays are represented as `Vector Cell` instead of `[Cell]`, as vector access is $O(1)$.

```
data Value
  = Unit
  | Bool Bool
  | Int Int64
  | Float Double
  | Array (Vector Cell)

data Cell = Cell (IORef Value)
```

To operate on cells, the `newCell`, `readCell` and `writeCell` functions are provided. These create, read from, and modify cells, respectively.

```
newCell :: MonadIO m => Value -> m Cell
newCell val = liftIO $ do
  ref <- newIORef val
  pure (Cell ref)

readCell :: MonadIO m => Cell -> m Value
readCell (Cell ref) = liftIO (readIORef ref)

writeCell :: MonadIO m => Cell -> Value -> m Cell
writeCell (Cell ref) val = liftIO $ do
  writeIORef ref val
  pure (Cell ref)
```

Above, `MonadIO m` stands for the class of monads in which `IO` computations may be embedded; the function `liftIO :: MonadIO m => IO a -> m a` lifts a computation from the `IO` monad. This abstraction allows I/O operations to be run in many monadic stacks.

The functions `cloneCell` and `cloneVal` clone cells and values, respectively. As we'll see, these are needed to implement the pass by value evaluation strategy. To clone arrays, `Vector.forM :: Monad m => Vector a -> (a -> m b) -> m (Vector b)` is used; this function applies a monadic action to the elements of a vector, collecting the results.

```
cloneCell :: MonadIO m => Cell -> m Cell
cloneCell cell = do
  val <- readCell cell
  val' <- cloneVal val
  newCell val'

cloneVal :: MonadIO m => Value -> m Value
cloneVal = \case
  Unit -> pure Unit
  Bool x -> pure (Bool x)
  Int x -> pure (Int x)
  Float x -> pure (Float x)

  Array cells -> do
    cells' <- Vector.forM cells cloneCell
    pure (Array cells')
```

Finally, equality between pairs of cells or values can be tested with `compareCells` and `compareVals`. Array elements are accessed using `(!) :: Vector a -> Int -> a`. Note that non-numeric values are compared lexicographically.

```
compareCells :: MonadIO m => Cell -> Cell -> m (Either (Type, Type) Ordering)
compareCells cell1 cell2 = do
  val1 <- readCell cell1
  val2 <- readCell cell2
  compareVals val1 val2

compareVals :: MonadIO m => Value -> Value -> m (Either (Type, Type) Ordering)
compareVals val1 val2 = case (val1, val2) of
  (Unit, Unit) -> pure (Right EQ)
  (Bool x, Bool y) -> pure (Right (compare x y))
  (Int x, Int y) -> pure (Right (compare x y))
  (Float x, Float y) -> pure (Right (compare x y))

  (Array cells1, Array cells2) -> go (Vector.length cells1) (Vector.length cells2) 0
    where
      go n1 n2 i | i >= n1 || i >= n2 =
        pure (Right (compare n1 n2))

      go n1 n2 i = do
        val1 <- readCell (cells1 ! i)
        val2 <- readCell (cells2 ! i)

        compareVals val1 val2 >>= \case
          Right EQ -> go n1 n2 (i + 1)
          Right ordering -> pure (Right ordering)
          Left (t1, t2) -> pure (Left (t1, t2))

  (_, _) -> do
    t1 <- getType val1
    t2 <- getType val2
    pure (Left (t1, t2))
```

Devin's initial evaluation state can be created with `makePredefinedState`. The 2-tuple constructor `(,)` is used explicitly so that it can be applied partially.

```
makePredefinedState :: MonadIO m => m State
makePredefinedState = liftIO $ do
  let f1 = ("not", BuiltinNot)
  let f2 = ("len", BuiltinLen)
  let f3 = ("intToFloat", BuiltinIntToFloat)
  let f4 = ("floatToInt", BuiltinFloatToInt)

  v1 <- (,) "true" <$> newCell (Bool True)
  v2 <- (,) "false" <$> newCell (Bool False)
  v3 <- (,) "unit" <$> newCell Unit

  pure [Frame Nothing 0 [f4, f3, f2, f1] [v3, v2, v1]]
```

Execution is abstracted by the `Evaluator` data type; it represents a computation that updates a `State`, possibly producing some `IO` side-effect:

```
data Evaluator a = Evaluator (State -> IO (Result a, State))
  deriving Functor
```

To allow execution of Devin code to be performed in a step-by-step manner, the algebraic data type `Result` is provided. It has three constructors:

- `Done`, if the last statement has been executed. This data constructor stores the result of the computation.
- `Yield`, if a statement has been reached. The statement, along with the next step to execute, is stored by this constructor.
- `Error`, if an error occurred at runtime.

```
data Result a
  = Done a
  | Yield Statement (Evaluator a)
  | Error Error
  deriving Functor
```

As for the `Evaluator`, two functions are provided: `runEvaluatorStep` and `runEvaluator`. The former runs a single step of computation, while the latter runs all remaining steps.

```
runEvaluatorStep :: MonadIO m => Evaluator a -> State -> m (Result a, State)
runEvaluatorStep (Evaluator f) state = liftIO (f state)

runEvaluator :: MonadIO m => Evaluator a -> State -> m (Either Error a, State)
runEvaluator mx state = do
  (result, state') <- runEvaluatorStep mx state

  case result of
    Done x -> pure (Right x, state')
    Yield _ mx -> runEvaluator mx state'
    Error error -> pure (Left error, state')
```

To use `Evaluator`s monadically, instances of `Applicative` and `Monad` are provided:

```
instance Applicative Evaluator where
  pure :: a -> Evaluator a
  pure x = Evaluator (\state -> pure (Done x, state))

  liftA2 :: (a -> b -> c) -> Evaluator a -> Evaluator b -> Evaluator c
  liftA2 f mx my = Evaluator $ \state -> do
    (result, state') <- runEvaluatorStep mx state

    case result of
      Done x -> runEvaluatorStep (f x <$> my) state'
      Yield statement mx -> pure (Yield statement (liftA2 f mx my), state')
      Error error -> pure (Error error, state')
```

```
instance Monad Evaluator where
  (>>=) :: Evaluator a -> (a -> Evaluator b) -> Evaluator b
  mx >>= f = Evaluator $ \state -> do
    (result, state') <- runEvaluatorStep mx state

    case result of
      Done x -> runEvaluatorStep (f x) state'
      Yield statement mx -> pure (Yield statement (f =<< mx), state')
      Error error -> pure (Error error, state')
```

An instance for `MonadIO` is provided as well. This instance permits, among others, `newCell`, `readCell`, `writeCell`, `cloneCell`, `compareCells`, `cloneVals` and `compareVals` to be used within the `Evaluator` monad as well.

```
instance MonadIO Evaluator where
  liftIO :: IO a -> Evaluator a
  liftIO mx = Evaluator $ \state -> do
    x <- mx
    pure (Done x, state)
```

Functions and variables can be defined and looked up with the `defineFun`, `lookupFun`, `defineVar` and `lookupVar` functions. On lookup, the stack of frames is searched for the requested binding from top to bottom.

Contrary to the environment used in static type checking, there's not a one-to-one correspondence between the evaluator's runtime stack and lexical scoping. As such, each frame has to keep track of its lexical parent; this is done with the `poffset` field, which stands for *parent offset*. This value is used in `lookupFun` and `lookupVar`, as it indicates how many frames need to be skipped during search:

- When `poffset` $= 1$, the previous frame coincides with the lexical parent: only the current frame needs to be skipped to reach its parent.
- When `poffset` > 1, the previous frame doesn't coincide with the lexical parent: more than one frame needs to be skipped to reach the parent.

  Consider two functions, `f` and `g`, defined at the same nesting level; suppose that `f` calls `g`. It should be clear that `g`'s body doesn't have access to the variables defined inside `f`. At the same time, from the perspective of the runtime stack, `g`'s frame succeeds `f`'s frame. It is in cases like these that the `poffset` field needs to be used.

```
defineFun :: String -> Function -> Evaluator ()
defineFun name fun = Evaluator $ \case
  [] -> pure (Done (), [Frame 0 [(name, fun)] []])

  frame : frames -> do
    let funs' = (name, fun) : funs frame
    pure (Done (), frame{funs = funs'} : frames)
```

```
lookupFun :: String -> Evaluator (Maybe (Function, Int))
lookupFun name = Evaluator (\state -> pure (Done (go 0 state), state))
  where
    go _ [] = Nothing

    go depth (Frame{poffset, funs} : frames) = case lookup name funs of
      Just fun -> Just (fun, depth)
      Nothing -> go (depth + max 1 poffset) (drop (poffset - 1) frames)

defineVar :: String -> Cell -> Evaluator ()
defineVar name cell = Evaluator $ \case
  [] -> pure (Done (), [Frame 0 [] [(name, cell)]])

  frame : frames -> do
    let vars' = (name, cell) : vars frame
    pure (Done (), frame{vars = vars'} : frames)

lookupVar :: String -> Evaluator (Maybe (Cell, Int))
lookupVar name = Evaluator (\state -> pure (Done (go 0 state), state))
  where
    go _ [] = Nothing

    go depth (Frame{poffset, vars} : frames) = case lookup name vars of
      Just cell -> Just (cell, depth)
      Nothing -> go (depth + max 1 poffset) (drop (poffset - 1) frames)
```

The expressions `frame{funs = funs'}` and `frame{vars = vars'}` used in the functions `defineFun` and `defineVar` denote *record updates*: the record `frame` is copied, updating the fields `funs` and `vars`, respectively.

To run another `Evaluator` on a modified state where a new `Frame` is added, `withNewFrame` is provided; when the function returns, the previous state is restored.

```
withNewFrame :: Int -> Evaluator a -> Evaluator a
withNewFrame poffset mx = do
  pushFrame
  x <- mx
  popFrame
  pure x

  where
    pushFrame = Evaluator $ \state ->
      pure (Done (), Frame poffset [] [] : state)

    popFrame = Evaluator $ \state ->
      pure (Done (), tail state)
```

Finally, the functions `yield` and `raise` are provided for convenience.

```
yield :: Statement -> Evaluator ()
yield statement = Evaluator $ \state ->
  pure (Yield statement (pure ()), state)
```

```
raise :: Error -> Evaluator a
raise error = Evaluator $ \state ->
  pure (Error error, state)
```

4.4.1 Evaluating expressions

The module `Devin.Evaluators` implements functions for evaluating expressions, statements, and definitions. `evalExpression :: Expression -> Evaluator Cell` evaluates expressions and returns a cell containing the resulting value.

Evaluating integer and floating-point literals is trivial: all that's needed is to allocate a new cell containing the value of the literal. For integers, an exception is raised if the literal is out of the 64-bit range.

```
evalExpression expression@IntegerExpression{integer} =
  case toIntegralSized integer of
    Just x -> newCell (Int x)
    Nothing -> raise (IntegerOverflow expression)

evalExpression RationalExpression{rational} =
  newCell (Float (fromRat rational))
```

Variable literals are simply looked up in the running state:

```
evalExpression VarExpression{varName, interval} =
  lookupVar varName >>= \case
    Just (cell, _) -> pure cell
    Nothing -> raise (UnknownVar varName interval)
```

Arrays are processed by recursively evaluating all the elements, cloning the resulting cells. As a general pattern, cloning is performed when it is undesirable for two objects to refer to the same value. For instance, the expression `[x]` should produce a new array with one element which is a copy of `x`'s cell; otherwise, modifying `x` would also appear to modify the newly created array.

```
evalExpression ArrayExpression{elems} = do
  cells <- flip Vector.unfoldrM elems $ \case
    [] -> pure Nothing

    (elem : elems) -> do
      cell <- evalExpression elem
      cell' <- cloneCell cell
      pure (Just (cell', elems))

  newCell (Array cells)
```

Devin arrays are constructed by repeatedly applying a monadic generator function, using `Vector.unfoldrM :: Monad m => (b -> m (Maybe (a, b))) -> b -> m (Vector a)`.

The generator function yields either `Just (`x`, `y`)`, or `Nothing` if there are no more elements; x is the next element and y is the new seed.

Next, access to array elements is evaluated by considering the array to access and its index. It is not assumed that the type checker has been run before: if the expression involves invalid types, a runtime exception is raised.

```
evalExpression AccessExpression{array, index} = do
  arrayCell <- evalExpression array
  arrayVal <- readCell arrayCell

  indexCell <- evalExpression index
  indexVal <- readCell indexCell

  case (arrayVal, indexVal) of
    (Array cells, Int x) -> case toIntegralSized x of
      Just n | Just cell <- cells !? n -> pure cell
      _ -> raise (IndexOutOfBounds index x)

    (Array _, _) -> do
      indexT <- getType indexVal
      raise (InvalidType index Type.Int indexT)

    (_, _) -> do
      arrayT <- getType arrayVal
      raise (InvalidType array (Type.Array Type.Unknown) arrayT)
```

The binary operator `(!?) :: Vector a -> Int -> Maybe a` is used to safely index vectors. It is closely related to `(!) :: Vector a -> Int -> a`; if the index is out of bounds, it returns `Nothing` instead of throwing an exception.

Arithmetic expressions may lead to overflows. The higher-order helper functions `safeUnary` and `safeBinary` are used to wrap Haskell's built-in operators, allowing for overflow detection. Both `toInteger` and `toIntegralSized` are functions from the `base` library. In the context of Devin's evaluator, `toInteger` converts an `Int64` to an unbounded `Integer`; its inverse, `toIntegralSized`, attempts to convert an `Integer` back to an `Int64`. If it is not possible to fit an `Integer` in the 64 bits of an `Int64`, `toIntegralSized` yields `Nothing`.

```
safeUnary op x = toIntegralSized (op (toInteger x))
safeBinary op x y = toIntegralSized (toInteger x `op` toInteger y)
```

In the case of unary arithmetic expressions, the operand is evaluated recursively. If the operator is `-`, the result is negated; if the operator is `+`, no further action is needed.

```
evalExpression UnaryExpression{unary, operand}
  | PlusOperator{} <- unary = do
    cell <- evalExpression operand
    val <- readCell cell
```

```
    case val of
      Int x -> newCell (Int x)
      Float x -> newCell (Float x)

      _ -> do
        operandT <- getType val
        raise (InvalidUnary unary operandT)

evalExpression expression@UnaryExpression{unary, operand}
  | MinusOperator{} <- unary = do
    cell <- evalExpression operand
    val <- readCell cell

    case val of
      Int x | Just y <- safeUnary negate x -> newCell (Int y)
      Int _ -> raise (IntegerOverflow expression)

      Float x -> newCell (Float (negate x))

      _ -> do
        operandT <- getType val
        raise (InvalidUnary unary operandT)
```

The implementations of the binary arithmetic operators `-`, `/` and `%` are similar to each other, with the exception that the operator `%` supports only operands of type `Int`, and that both `/` and `%` require the dividend not to be 0.

```
evalExpression expression@BinaryExpression{left, binary, right}
  | DivideOperator{} <- binary = do
    leftCell <- evalExpression left
    leftVal <- readCell leftCell

    rightCell <- evalExpression right
    rightVal <- readCell rightCell

    case (leftVal, rightVal) of
      (Int _, Int 0) -> raise (DivisionByZero expression)
      (Int x, Int y) | Just z <- safeBinary div x y -> newCell (Int z)
      (Int _, Int _) -> raise (IntegerOverflow expression)

      (Float x, Float y) -> newCell (Float (x / y))

      (_, _) -> do
        leftT <- getType leftVal
        rightT <- getType rightVal
        raise (InvalidBinary binary leftT rightT)
```

The binary operators `+` and `*` are overloaded to work with both arrays and numbers. For `+`, the operands must either be both numbers or both arrays; for `*`, the operands must be either both numbers or one a number and one an array. For brevity, only the implementation of `+` is explained.

First, both the left and right operands of `+` are evaluated. If they are of the same numeric type, addition is performed. Otherwise, if both operands are arrays:

1. The lengths n_1 and n_2 of the arrays are used to compute the length n_3 of the new array, which will contain the elements of the first array concatenated with the second;
2. A new array of length $n_3 = n_1 + n_2$ is created; it is an error if n_3 can't be converted to an `Int` without overflow;
3. Elements of indices 0 through $n_1 - 1$ of the new array are set to clones of the elements of the first array; elements of indices n_1 through $n_3 - 1$ are set to clones of the elements of the second array.

```
evalExpression expression@BinaryExpression{left, binary, right}
  | AddOperator{} <- binary = do
    leftCell <- evalExpression left
    leftVal <- readCell leftCell

    rightCell <- evalExpression right
    rightVal <- readCell rightCell

    case (leftVal, rightVal) of
      (Int x, Int y) | Just z <- safeBinary (+) x y -> newCell (Int z)
      (Int _, Int _) -> raise (IntegerOverflow expression)

      (Float x, Float y) -> newCell (Float (x + y))

      (Array rs1, Array rs2) -> do
        let n1 = Vector.length rs1
        let n2 = Vector.length rs2

        case safeBinary (+) n1 n2 of
          Nothing -> raise (IntegerOverflow expression)

          Just n3 -> do
            let f i = cloneCell (if i < n1 then rs1 ! i else rs2 ! (i - n1))
            cells <- Vector.generateM n3 f
            newCell (Array cells)

      (_, _) -> do
        leftT <- getType leftVal
        rightT <- getType rightVal
        raise (InvalidBinary binary leftT rightT)
```

The operators `==`, `!=`, `<`, `<=`, `>` and `>=` are implemented using `compareCells`. Note that an `Ordering` is itself an instance of `Ord`, making the implementation trivial for all operators:

```
evalExpression BinaryExpression{left, binary, right}
  | LessOperator{} <- binary = do
    leftCell <- evalExpression left
    rightCell <- evalExpression right
```

```
    compareCells leftCell rightCell >>= \case
      Right ordering -> newCell (Bool (ordering < EQ))
      Left (leftT, rightT) -> raise (InvalidBinary binary leftT rightT)
```

The boolean operators `and` and `or` are lazy in their second operand. Thus, their implementation differs from that of other binary operators such as `xor`.

```
evalExpression BinaryExpression{left, binary, right}
  | AndOperator{} <- binary = do
    leftCell <- evalExpression left
    leftVal <- readCell leftCell

    case leftVal of
      Bool False -> newCell (Bool False)

      _ -> do
        rightCell <- evalExpression right
        rightVal <- readCell rightCell

        case (leftVal, rightVal) of
          (Bool _, Bool y) -> newCell (Bool y)

          (_, _) -> do
            leftT <- getType leftVal
            rightT <- getType rightVal
            raise (InvalidBinary binary leftT rightT)

evalExpression BinaryExpression{left, binary, right}
  | XorOperator{} <- binary = do
    leftCell <- evalExpression left
    leftVal <- readCell leftCell

    rightCell <- evalExpression right
    rightVal <- readCell rightCell

    case (leftVal, rightVal) of
      (Bool x, Bool y) -> newCell (Bool (x /= y))

      (_, _) -> do
        leftT <- getType leftVal
        rightT <- getType rightVal
        raise (InvalidBinary binary leftT rightT)
```

Assignments simply update the cell of the left operand to the value of the right operand:

```
evalExpression BinaryExpression{left, binary = PlainAssignOperator{}, right} = do
  leftCell <- evalExpression left

  rightCell <- evalExpression right
  rightVal <- readCell rightCell

  rightVal' <- cloneVal rightVal
  writeCell leftCell rightVal'
```

The other assignment operators are implemented according to the shorthands of Table 3.4. For example, `/=` is implemented as follows:

```
evalExpression expression@BinaryExpression{left, binary, right}
  | DivideAssignOperator{} <- binary = do
    leftCell <- evalExpression left
    leftVal <- readCell leftCell

    rightCell <- evalExpression right
    rightVal <- readCell rightCell

    case (leftVal, rightVal) of
      (Int _, Int 0) -> raise (DivisionByZero expression)
      (Int x, Int y) | Just z <- safeBinary div x y -> writeCell leftCell (Int z)
      (Int _, Int _) -> raise (IntegerOverflow expression)

      (Float x, Float y) -> writeCell leftCell (Float (x / y))

      (_, _) -> do
        leftT <- getType leftVal
        rightT <- getType rightVal
        raise (InvalidBinary binary leftT rightT)
```

Executing function calls is the most complex part of Devin's evaluator. The called function is looked up, raising an error if it is not found. Arguments are evaluated from left to right; if an argument has the wrong type, or if too many or too few arguments are passed, an exception is raised. Built-in functions are handled according to their meaning: `not` negates a boolean, `len` returns the length of an array, `toFloat` and `toInt` convert an `Int` to a `Float` and round a `Float` to an `Int`, respectively. User-defined function calls are handled by binding parameter names to the evaluated arguments and then recursively evaluating the callee's body. If a given argument is passed by value, a copy of its cell is bound; if it is passed by reference, no copy is performed. When a user-defined function's body is evaluated, a new frame is pushed by calling `withNewFrame (depth + 1)`, where `depth` is the nesting level of the callee with respect to the caller.

```
evalExpression expression@CallExpression{funId, args} = do
  let SymbolId{name, interval} = funId
  argCells <- for args evalExpression

  lookupFun name >>= \case
    Just (BuiltinNot, _) | [cell] <- argCells -> do
      val <- readCell cell

      case val of
        Bool x -> newCell (Bool (not x))

        _ -> do
          argT <- getType val
          raise (InvalidType (head args) Type.Bool argT)
```

```
Just (BuiltinNot, _) ->
  raise (InvalidArgCount expression 1 (length argCells))

Just (BuiltinLen, _) | [cell] <- argCells -> do
  val <- readCell cell

  case val of
    Array cells -> newCell (Int (fromIntegral (length cells)))

    _ -> do
      argT <- getType val
      raise (InvalidType (head args) (Type.Array Type.Unknown) argT)

Just (BuiltinLen, _) ->
  raise (InvalidArgCount expression 1 (length argCells))

Just (BuiltinIntToFloat, _) | [cell] <- argCells -> do
  val <- readCell cell

  case val of
    Int x -> newCell (Float (fromIntegral x))

    _ -> do
      argT <- getType val
      raise (InvalidType (head args) Type.Int argT)

Just (BuiltinIntToFloat, _) ->
  raise (InvalidArgCount expression 1 (length argCells))

Just (BuiltinFloatToInt, _) | [cell] <- argCells -> do
  val <- readCell cell

  case val of
    Float x -> newCell (Int (round x))

    _ -> do
      argT <- getType val
      raise (InvalidType (head args) Type.Float argT)

Just (BuiltinFloatToInt, _) ->
  raise (InvalidArgCount expression 1 (length argCells))

Just (UserDefined FunDefinition{params, body}, depth) ->
  withNewFrame (depth + 1) (go 0 params argCells)

  where
    -- Pass argument by value:
    go n ((Nothing, SymbolId{name}, _) : params) (argCell : argCells) = do
      argCell' <- cloneCell argCell
      defineVar name argCell'
      go (n + 1) params argCells
```

```
        -- Pass argument by reference:
        go n ((Just _, SymbolId{name}, _) : params) (argCell : argCells) = do
          defineVar name argCell
          go (n + 1) params argCells

        -- If argument count is correct:
        go _ [] [] = evalStatement body >>= \case
          Just cell -> cloneCell cell
          Nothing -> newCell Unit

        -- If argument count is incorrect:
        go n params argCells = do
          let expected = n + length params
          let actual = n + length argCells
          raise (InvalidArgCount expression expected actual)

    _ -> raise (UnknownFun name interval)
```

4.4.2 Evaluating statements

The function `evalStatement :: Statement -> Evaluator (Maybe Cell)` evaluates single statements; its result is `Just` for statements returning control, and `Nothing` otherwise.

The following two base cases are trivial (`evalDefinitions`'s implementation is discussed in the next section):

```
evalStatement statement@DefinitionStatement{definition} = do
  yield statement
  evalDefinitions [definition]
  pure Nothing

evalStatement statement@ExpressionStatement{effect} = do
  yield statement
  evalExpression effect
  pure Nothing
```

If-else statements are processed by evaluating their predicate; based on its value, the relevant branch is executed. As previously discussed, a declaration can be inserted wherever a statement can be placed; thus, a new frame is pushed to evaluate branches.

```
evalStatement statement@IfStatement{predicate, trueBranch} = do
  yield statement

  cell <- evalExpression predicate
  val <- readCell cell

  case val of
    Bool True -> withNewFrame 1 (evalStatement trueBranch)
    Bool False -> pure Nothing
```

```
    _ -> do
      t <- getType val
      raise (InvalidType predicate Type.Bool t)

evalStatement statement@IfElseStatement{predicate, trueBranch, falseBranch} = do
  yield statement

  val <- evalExpression predicate
  cell <- readCell val

  case cell of
    Bool True -> withNewFrame 1 (evalStatement trueBranch)
    Bool False -> withNewFrame 1 (evalStatement falseBranch)

    _ -> do
      t <- getType cell
      raise (InvalidType predicate Type.Bool t)
```

For while statements, the predicate is evaluated. If true, the body is executed and then the while statement itself is evaluated again. Do-while statements are implemented similarly to while statements.

```
evalStatement statement@WhileStatement{predicate, body} = do
  yield statement

  untilJustM $ do
    cell <- evalExpression predicate
    val <- readCell cell

    case val of
      Bool False -> pure (Just Nothing)

      Bool True -> withNewFrame 1 (evalStatement body) >>= \case
        Just cell -> pure (Just (Just cell))
        Nothing -> pure Nothing

      _ -> do
        t <- getType val
        raise (InvalidType predicate Type.Bool t)
```

Return statements are handled by evaluating the expression representing what's to be returned; if no such expression is present, the statement is assumed to return `unit`.

```
evalStatement statement@ReturnStatement{result = Just result} = do
  yield statement
  cell <- evalExpression result
  pure (Just cell)

evalStatement statement@ReturnStatement{result = Nothing} = do
  yield statement
  cell <- newCell Unit
  pure (Just cell)
```

For assertions, the predicate is evaluated. If it is not of type `Bool`, or if it is `false`, an exception describing the error is raised.

```
evalStatement statement@AssertStatement{predicate} = do
  yield statement

  cell <- evalExpression predicate
  val <- readCell cell

  case val of
    Bool False -> raise (AssertionFailed statement)
    Bool True -> pure Nothing

    _ -> do
      t <- getType val
      raise (InvalidType predicate Type.Bool t)
```

Breakpoint statements are handled trivially:

```
evalStatement statement@BreakpointStatement{} = do
  yield statement
  pure Nothing
```

Blocks are processed in two steps. First, the current frame is populated with all function definitions; then, all other statements and/or definitions are evaluated. With `firstJustM`, execution stops at the first statement which returns control.

```
evalStatement statement@BlockStatement{statements} = do
  yield statement

  withNewFrame 1 $ do
    for_ statements $ \case
      DefinitionStatement{definition} -> evalDefinition1 definition
      _ -> pure ()

    flip firstJustM statements $ \case
      DefinitionStatement{definition} -> do
        evalDefinition2 definition
        pure Nothing

      statement -> evalStatement statement
```

4.4.3 Evaluating variable and function definitions

In Devin's implementation, the evaluator is independent from the type checker: the environment computed by type checking can't be reused during execution; thus, the evaluator needs to maintain its own function and variable bindings. This leads to some code duplication, as bindings are managed similarly in both the type checker and evaluator; however, implementation is so straightforward that there's little benefit in refactoring.

As with type checking, evaluation is performed in two passes. In the first pass, function definitions are bound to the current frame; in the second, variable definitions are evaluated.

```
evalDefinitions :: [Definition] -> Evaluator ()
evalDefinitions definitions = do
  for_ definitions evalDefinition1
  for_ definitions evalDefinition2

evalDefinition1 :: Definition -> Evaluator ()
evalDefinition1 definition = case definition of
  VarDefinition{} -> pure ()
  FunDefinition{funId = SymbolId{name}} -> defineFun name (UserDefined definition)

evalDefinition2 :: Definition -> Evaluator ()
evalDefinition2 = \case
  VarDefinition{varId = SymbolId{name}, value} -> do
    cell <- evalExpression value
    cell' <- cloneCell cell
    defineVar name cell'

  FunDefinition{} -> pure ()
```

4.5 The parser

Devin's parser is implemented with the help of the `parsec` parser combinator library [14]. Parser combinators have been described in section 2.3.

As described, the positions of leaf nodes are stored in the syntax tree. While the `parsec` library does provide functionality to query the current line and column during parsing, it doesn't allow obtaining the character index describing the same position: this is an inconvenient limitation, as the syntax highlighter works with indices. Fortunately, there is a workaround to get such arguably missing functionality back. Usually, the input of a `parsec` parser is some string; by writing new instances for the `Stream` type class, other kinds of inputs can be supported. The trick is to consider the parser's input to be a pair: the first element being an integer character offset, the second being the underlying stream.

```
instance (Num a, Stream s m t) => Stream (a, s) m t where
  uncons :: (a, s) -> m (Maybe (t, (a, s)))
  uncons (offset, stream) = uncons stream >>= \case
    Just (token, rest) -> pure (Just (token, (offset + 1, rest)))
    Nothing -> pure Nothing

getOffset :: Monad m => ParsecT (a, s) u m a
getOffset = do
  State{stateInput = (offset, _)} <- getParserState
  pure offset
```

The module `Devin.Parsers` implements the functions necessary to parse Devin source code. A Devin parser is a `Parsec (Int, s) [Token] a`, where: `Int` is the offset, `s` is the underlying `Stream` (either `String` or `Text`), `[Token]` is the list of comments accumulated in the state, and `a` is the result of the parser.

```
type Parser s a = Parsec (Int, s) [Token] a
type ParserT s m a = ParsecT (Int, s) [Token] m a
```

4.5.1 Parsing expressions

Let's start by examining how an integer is parsed. Syntactically, an integer is composed of one or more digits in the range 0–9; it may be preceded by a sign (`+` or `-`).

```
integerExpression :: Stream s m Char => ParserT s m Expression
integerExpression = flip label "number" $ try $ syntax $ do
  sign <- (char '+' $> 1) <|> (char '-' $> -1) <|> pure 1
  digits <- many1 (satisfy isDigit)
  let magnitude = foldl (\a d -> 10 * a + toInteger (digitToInt d)) 0 digits
  pure (IntegerExpression (sign * magnitude))
```

The do expression first parses the sign, then the digits. It computes the magnitude, which when multiplied by the sign gives the value of the integer.

`char '+'` attempts to parse `+`; if it succeeds, it yields the parsed character. Similarly, `char '-'` parses a `-`. If the parsed character is `+`, the sign is positive; if it is `-`, the sign is negative; if neither a `+` nor a `-` is parsed, the sign is assumed to be positive.

The combinator `(<|>)` implements choice. The parser p `<|>` q first applies p; if it succeeds, the value of p is returned; if p fails without consuming any input, parser q is tried. Thus, `(char '+' $> 1) <|> (char '-' $> -1) <|> pure 1` tries to parse a `+` (in which case the sign is $+1$), then `-` (in which case the sign is -1); if neither character could be parsed, `pure 1` does nothing and yields $+1$.

Digits are parsed with `satisfy isDigit`. The parser `satisfy` p succeeds for any character for which the predicate p holds true, and returns the parsed character. The combinator `many1` p applies a parser p one or more times; thus, `many1 (satisfy isDigit)` parses the longest sequence of one or more digits.

The constructor `IntegerExpression` has two fields, the second of which is an `(Int, Int)`; this field marks the start and end offsets of where the integer was parsed. The `syntax` combinator, used in the previous listing, gets the positions before and after the do expression; this tuple is given to the partially applied `IntegerExpression`. The function `try` makes a parser atomic: it either succeeds or fails without consuming any input. `flip label "number"` gives a description to be used for error reporting.

The helper function `syntax` is implemented as follows:

```
syntax :: Stream s m Char => ParserT s m ((Int, Int) -> a) -> ParserT s m a
syntax mf = do
  start <- getOffset
  f <- mf
  end <- getOffset
  pure (f (start, end))
```

Numbers with decimals are parsed similarly to integers. The dot, parsed with `char '.'`, is ignored by using `(*>)`: this combinator sequences two parsers, discarding the result of the first operand. The expression p `$>` q, used in both earlier and in the following listing, is equivalent to p `*> pure` q.

```
rationalExpression :: Stream s m Char => ParserT s m Expression
rationalExpression = flip label "number" $ try $ syntax $ do
  sign <- (char '+' $> 1) <|> (char '-' $> -1) <|> pure 1
  intDigits <- many1 (satisfy isDigit)
  fracDigits <- char '.' *> many1 (satisfy isDigit)
  let intPart = foldl (\a d -> 10 * a + toRational (digitToInt d)) 0 intDigits
  let fracPart = foldr (\d a -> 0.1 * (a + toRational (digitToInt d))) 0 fracDigits
  pure (RationalExpression (sign * (intPart + fracPart)))
```

Array literals are parsed as a list of comma-separated expressions surrounded by brackets:

```
arrayExpression :: Stream s m Char => ParserT s m Expression
arrayExpression = do
  open <- token "["

  (elems, commas) <- s *> optionMaybe expression >>= \case
    Nothing -> pure ([], [])

    Just first -> do
      rest <- many (liftA2 (,) (try (s *> token ",")) (s *> expression))
      pure (first : map snd rest, map fst rest)

  close <- s *> token "]"
  pure (ArrayExpression open elems commas close)
```

After the opening bracket is matched with `token "["`, zero or more spaces are skipped and an expression is optionally parsed with `optionMaybe expression`. If no expression was parsed, the array is empty and the next character must be a `]`; otherwise, zero or more comma-expression pairs are parsed.

Two helper functions are used: `token` and `s`; the first one matches the input string against a sequence of characters, the second one skips zero or more spaces. They are defined as:

```
token :: Stream s m Char => String -> ParserT s m Token
token literal = syntax $ do
  text literal
  pure Token
```

```
text :: Stream s m Char => String -> ParserT s m String
text literal = try (string literal) <?> ("‘" ++ literal ++ "’")

s :: Stream s m Char => ParserT s m ()
s = skipMany (skipMany1 (space <?> "") <|> void (comment <?> ""))
```

The function `string` is provided by `parsec`; it parses a sequence of characters, consuming characters even if the whole string couldn't be matched: `try` has to be used to disable this odd behavior. `skipMany` and `skipMany1` are similar to `many` and `many1`, except that they ignore the result. The combinator `(<?>)` gives a label to a parser; if an empty string is provided, any previous label is discarded.

As mentioned, comments are accumulated in the parser's state. They are parsed as follows:

```
comment :: Stream s m Char => ParserT s m Token
comment = flip label "comment" $ do
  token <- syntax $ do
    text "//"
    skipMany (noneOf "\n\v\r\x85\x2028\x2029")
    pure Token

  modifyState (++ [token])
  pure token
```

The `noneOf` s function matches any single character which is not included in the string s. Thus, once `//` has been parsed, all characters until the next line break become part of that comment. `modifyState` updates the parser's state according to the provided function.

Variable and function names are parsed following Unicode's definitions of general-purpose identifiers [15], albeit with some minor variations applied to their recommendations. Simplifying, identifiers start with a letter and continue with either letters or numbers; the more general Unicode definition allows identifiers like `πυθαγόρας` to be accepted as well.

```
-- Regular expression: [\p{L}\p{Nl}\p{Pc}][\p{L}\p{Nl}\p{Pc}\p{Mn}\p{Mc}\p{Nd}]*
identifier :: Stream s m Char => ParserT s m String
identifier = flip label "identifier" $ do
  start <- satisfy (isStart . generalCategory)
  continue <- many (satisfy (isContinue . generalCategory))
  pure (start : continue)

  where
    isStart UppercaseLetter = True
    isStart LowercaseLetter = True
    isStart TitlecaseLetter = True
    isStart ModifierLetter = True
    isStart OtherLetter = True
    isStart LetterNumber = True
    isStart ConnectorPunctuation = True
    isStart _ = False
```

```
    isContinue NonSpacingMark = True
    isContinue SpacingCombiningMark = True
    isContinue DecimalNumber = True
    isContinue category = isStart category
```

Keywords are handled by parsing an identifier and checking if it equals some string.

```
keyword :: Stream s m Char => String -> ParserT s m Token
keyword literal = flip label ("keyword ‘" ++ literal ++ "’") $ syntax $ do
  (name, state) <- try (lookAhead (liftA2 (,) identifier getParserState))
  guard (name == literal)
  setParserState state
  pure Token
```

Unary operators are parsed as follows:

```
unaryOperator :: Stream s m Char => ParserT s m UnaryOperator
unaryOperator = syntax $ choice
  [
    text "+" $> PlusOperator,
    text "-" $> MinusOperator
  ]
```

The `choice` combinator attempts to apply the parsers in a list in order until one of them succeeds; the value provided by the parser which succeeds is returned. This combinator is equivalent to `foldr (<|>) empty`.

Unary expressions are parsed as a unary operator followed by an expression.

```
unaryExpression :: Stream s m Char => ParserT s m Expression
unaryExpression = do
  unary <- unaryOperator
  operand <- s *> expression6
  pure (UnaryExpression unary operand)
```

Parsing binary expressions is somewhat more involved, as operator precedence has to be accounted for. Each level of precedence is handled by a different parsing function.

```
expression :: Stream s m Char => ParserT s m Expression
expression = expression1

expression1 :: Stream s m Char => ParserT s m Expression
expression1 = chainl1 expression2 $ do
  binary <- try $ between s s $ syntax $ choice
    [
      keyword "and" $> AndOperator,
      keyword "or" $> OrOperator,
      keyword "xor" $> XorOperator
    ]

  pure (\left right -> BinaryExpression left binary right)
```

```
expression2 :: Stream s m Char => ParserT s m Expression
expression2 = chainl1 expression3 $ do
  binary <- try $ between s s $ syntax $ choice
    [
      text "==" $> EqualOperator,
      text "!=" $> NotEqualOperator
    ]

  pure (\left right -> BinaryExpression left binary right)

expression3 :: Stream s m Char => ParserT s m Expression
expression3 = chainl1 expression4 $ do
  binary <- try $ between s s $ syntax $ choice
    [
      text ">=" $> GreaterOrEqualOperator,
      text ">" $> GreaterOperator,
      text "<=" $> LessOrEqualOperator,
      text "<" $> LessOperator
    ]

  pure (\left right -> BinaryExpression left binary right)

expression4 :: Stream s m Char => ParserT s m Expression
expression4 = chainl1 expression5 $ do
  binary <- try $ between s s $ syntax $ choice
    [
      text "+" $> AddOperator,
      text "-" $> SubtractOperator
    ]

  pure (\left right -> BinaryExpression left binary right)

expression5 :: Stream s m Char => ParserT s m Expression
expression5 = chainl1 expression6 $ do
  binary <- try $ between s s $ syntax $ choice
    [
      text "*" $> MultiplyOperator,
      text "/" $> DivideOperator,
      text "%" $> ModuloOperator
    ]

  pure (\left right -> BinaryExpression left binary right)
```

The combinator `chainl1` p q parses one or more occurrences of p, separated by q. The recursive nature of the implemented functions implicitly handles operator precedence. `between s s` allows for an arbitrary amount of spaces between operands and operators.

For simplicity, `expression6` is not listed here. This function handles parenthesized expressions, function calls, variables and array access expressions. Interested readers may find the complete implementation in the appendix.

4.5.2 Parsing statements

There are many kinds of statements that can be parsed:

```
statement :: Stream s m Char => ParserT s m Statement
statement = choice
  [
    DefinitionStatement <$> definition,
    ifStatement,
    whileStatement,
    doWhileStatement,
    returnStatement,
    assertStatement,
    breakpointStatement,
    expressionStatement,
    blockStatement
  ]
```

The simplest statement to parse consists of an expression followed by a semicolon:

```
expressionStatement :: Stream s m Char => ParserT s m Statement
expressionStatement = do
  effect <- expression
  semicolon <- s *> token ";"
  pure (ExpressionStatement effect semicolon)
```

If statements are handled by parsing an `if` keyword, then an expression, then a statement. Optionally, an `else` keyword and another statement can follow.

```
ifStatement :: Stream s m Char => ParserT s m Statement
ifStatement = do
  ifKeyword <- keyword "if"
  predicate <- s *> expression
  trueBranch <- s *> statement

  optionMaybe (try (s *> keyword "else")) >>= \case
    Nothing -> pure (IfStatement ifKeyword predicate trueBranch)

    Just elseKeyword -> do
      falseBranch <- s *> statement
      pure (IfElseStatement ifKeyword predicate trueBranch elseKeyword falseBranch)
```

While statements are parsed as a `while` keyword, followed by an expression, followed by a statement. Do-while statements are parsed in a similar manner, except for the additional `do` keyword and the different ordering between the loop body and the predicate.

```
whileStatement :: Stream s m Char => ParserT s m Statement
whileStatement = do
  whileKeyword <- keyword "while"
  predicate <- s *> expression
  body <- s *> statement
  pure (WhileStatement whileKeyword predicate body)
```

```
doWhileStatement :: Stream s m Char => ParserT s m Statement
doWhileStatement = do
  doKeyword <- keyword "do"
  body <- s *> statement
  whileKeyword <- s *> keyword "while"
  predicate <- s *> expression
  semicolon <- s *> token ";"
  pure (DoWhileStatement doKeyword body whileKeyword predicate semicolon)
```

Return and assert statements are both parsed as a keyword, followed by an expression, followed by a semicolon. For return statements, the expression is optional.

```
returnStatement :: Stream s m Char => ParserT s m Statement
returnStatement = do
  returnKeyword <- keyword "return"
  result <- s *> optionMaybe expression
  semicolon <- s *> token ";"
  pure (ReturnStatement returnKeyword result semicolon)

assertStatement :: Stream s m Char => ParserT s m Statement
assertStatement = do
  assertKeyword <- keyword "assert"
  predicate <- s *> expression
  semicolon <- s *> token ";"
  pure (AssertStatement assertKeyword predicate semicolon)
```

Breakpoint statements are just a `breakpoint` keyword followed by a semicolon.

```
breakpointStatement :: Stream s m Char => ParserT s m Statement
breakpointStatement = do
  breakpointKeyword <- keyword "breakpoint"
  semicolon <- s *> token ";"
  pure (BreakpointStatement breakpointKeyword semicolon)
```

Finally, block statements are parsed as an open brace, followed by zero or more statements, followed by a closing brace.

```
blockStatement :: Stream s m Char => ParserT s m Statement
blockStatement = do
  open <- token "{"
  statements <- s *> many (statement <* s)
  close <- token "}"
  pure (BlockStatement open statements close)
```

Note that the operator `(<*)` chains two parsers, discarding the result of the second. Thus, the combinators `(*>)` and `(<*)` serve a similar purpose: sequencing two parsers and discarding the result of one of the two operands. As a mnemonic, the angle bracket points to the parser whose result has to be kept.

4.5.3 Parsing variable and function definitions

Both variable and function definitions use identifiers as names. The function `symbolId` wraps `identifier` by providing information about position in source code.

```
symbolId :: Stream s m Char => ParserT s m SymbolId
symbolId = syntax $ do
  name <- identifier
  pure (SymbolId name)
```

Variables are parsed as a sequence of the following items: a `var` keyword, an identifier, an equal sign, an expression, and a semicolon.

```
varDefinition :: Stream s m Char => ParserT s m Definition
varDefinition = do
  varKeyword <- keyword "var"
  varId <- s *> symbolId
  equalSign <- s *> token "="
  value <- s *> expression
  semicolon <- s *> token ";"
  pure (VarDefinition varKeyword varId equalSign value semicolon)
```

Parsing function definitions is more involved. Syntactically, function definitions are made up of a `def` keyword, an identifier, zero or more comma-separated parameters surrounded by parentheses, and a statement as body. Optionally, type annotations can be added to specify the types of parameters and the return type.

```
funDefinition :: Stream s m Char => ParserT s m Definition
funDefinition = do
  defKeyword <- keyword "def"
  funId <- s *> symbolId
  open <- s *> token "("

  (params, commas) <- s *> optionMaybe param >>= \case
    Nothing -> pure ([], [])

    Just first -> do
      rest <- many (liftA2 (,) (try (s *> token ",")) (s *> param))
      pure (first : map snd rest, map fst rest)

  close <- s *> token ")"

  returnInfo <- s *> optionMaybe (token "->") >>= \case
    Nothing -> pure Nothing

    Just arrow -> do
      returnTypeId <- s *> typeId
      pure (Just (arrow, returnTypeId))

  body <- s *> statement
  pure (FunDefinition defKeyword funId open params commas close returnInfo body)
```

Parameters are parsed by the `param` helper function:

```
param = do
  (refKeyword, paramId) <- choice
    [
      try $ do
        token <- keyword "ref"
        paramId <- s *> symbolId
        pure (Just token, paramId),

      do
        paramId <- symbolId
        pure (Nothing, paramId)
    ]

  paramInfo <- optionMaybe (try (s *> token ":")) >>= \case
    Nothing -> pure Nothing

    Just colon -> do
      paramTypeId <- s *> typeId
      pure (Just (colon, paramTypeId))

  pure (refKeyword, paramId, paramInfo)
```

4.6 The user interface

As explained in section 3.2, Devin's graphical user interface depends on the GTK+ library. For those unfamiliar, GTK+ is the standard GUI library used for Gnome applications; being cross-platform, GTK+ can be used on other platforms besides the Gnome desktop environment. One notable application using GTK+ is GIMP, the GNU Image Manipulation Program; in fact, GTK+ was formerly known as the GIMP ToolKit.

GTK+ is a library written in C and meant to be consumed in C. Given its popularity, many language bindings exist. Devin's implementation utilizes the bindings provided by the package `gi-gtk` [16]. This package is mostly a thin wrapper on top of GTK+; consequently, windows and widgets are to be constructed imperatively rather than declaratively.

Devin's UI implementation is complex, requiring almost 1200 lines of code. Many considerations were made while developing the UI; to be brief, I'll highlight two of them:

- Source code editing features are provided on top of GtkSourceView. This library provides a text widget which can be edited by users; there's built-in support for showing line numbers, automatic indentation, and syntax highlighting.

GtkSourceView's syntax highlighting functionality is quite handy: all that would be required to do is to write a text file describing Devin's grammar and let the library do its magic. However, should such functionality be used? Two grammars would have to be carefully kept in sync: one for Devin's parser, and one to perform syntax highlighting. One could argue that allowing for minor potential discrepancies between the two is good enough: few would notice. Or, one could be pedantic and accept only one source of truth. I chose the second option.

This consideration gives rise to the first problem: using only part of GtkSourceView's features, while performing syntax highlighting based on the output of Devin's parser. GtkSourceView operates on such an abstraction level that understanding how to do syntax highlighting from scratch is not trivial; in fact, I had to read the library's implementation of automatic syntax highlighting in order to do so.

- The second problem lies in the fact that the `gi-gtk` binding is incomplete; in particular, some of the functionality available in C isn't exposed by `gi-gtk`. One example of such missing functionality is the creation of dialog boxes.

  To get around such limitations, I had to investigate the source code of both GTK+ and `gi-gtk`. One of the tricks I employed is to use a subset of GTK+ functionality, exposed through `haskell-gi-base` (a dependency of `gi-gtk`), which allows for `GObject`s (such as widgets) to be instantiated in a different way.

The focus of this thesis is programming language implementation, not GUI development. Rather than explaining all code pertaining to Devin's UI, I'll give one example on how GTK+ may be used within Haskell. Devin's UI implementation is listed in the appendix, starting at section A.18.

Let's consider developing a simple graphical counting application. It should display a number n and provide two buttons: one to increment n, and one to decrement n; the initial value of n should be 0.

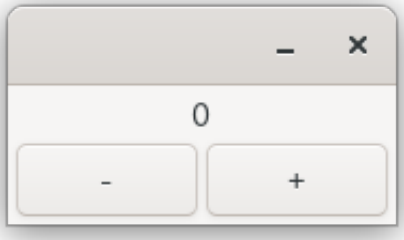

Figure 4.1: A simple counting UI

```
{-# LANGUAGE ImplicitParams #-}

module Main (main) where

import Control.Monad
import System.Exit
import qualified Data.Text as Text
import qualified GI.Gio as G
import qualified GI.Gtk as Gtk

main :: IO ()
main = do
  Just application <- Gtk.applicationNew Nothing [G.ApplicationFlagsDefaultFlags]
  G.onApplicationActivate application onActivate
  status <- G.applicationRun application Nothing
  when (status /= 0) (exitWith (ExitFailure (fromIntegral status)))

onActivate :: (Gtk.IsApplication a, ?self :: a) => G.ApplicationActivateCallback
onActivate = do
  label <- Gtk.labelNew (Just (Text.pack "0"))
  buttonMinus  <- Gtk.buttonNewWithLabel (Text.pack "-")
  buttonPlus <- Gtk.buttonNewWithLabel (Text.pack "+")

  buttonBox <- Gtk.buttonBoxNew Gtk.OrientationHorizontal
  Gtk.boxSetSpacing buttonBox 4
  Gtk.containerAdd buttonBox buttonMinus
  Gtk.containerAdd buttonBox buttonPlus

  box <- Gtk.boxNew Gtk.OrientationVertical 4
  Gtk.widgetSetMarginStart box 4
  Gtk.widgetSetMarginEnd box 4
  Gtk.widgetSetMarginTop box 4
  Gtk.widgetSetMarginBottom box 4
  Gtk.containerAdd box label
  Gtk.containerAdd box buttonBox

  window <- Gtk.applicationWindowNew ?self
  Gtk.windowSetTitle window Text.empty
  Gtk.windowSetResizable window False
  Gtk.containerAdd window box

  Gtk.onButtonClicked buttonMinus $ do
    t <- Gtk.labelGetLabel label
    Gtk.labelSetLabel label (Text.pack (show (read (Text.unpack t) - 1)))

  Gtk.onButtonClicked buttonPlus $ do
    t <- Gtk.labelGetLabel label
    Gtk.labelSetLabel label (Text.pack (show (read (Text.unpack t) + 1)))

  Gtk.widgetShowAll window
```

The entry point `main` is merely boilerplate code that invokes a callback to start the actual application. The window and all its contents are constructed in `onActivate`.

The first three lines of `onActivate` create a label initially set to `0` and two buttons: `buttonMinus` and `buttonPlus`. Note that `gi-gtk` employs the `Text` data type instead of `String` for efficiency; to convert between the two, `Text.pack :: String -> Text` and `Text.unpack :: Text -> String` have to be used.

To position the two buttons horizontally, a `Gtk.ButtonBox` container is used. Buttons are added to this container with `Gtk.containerAdd`, and spacing between buttons is configured with `Gtk.boxSetSpacing`.

To position the button box below the label created earlier, a `Gtk.Box` container is used. Some margin is added on all four sides of the container with `Gtk.widgetSetMarginStart`, `Gtk.widgetSetMarginEnd`, `Gtk.widgetSetMarginTop` and `Gtk.widgetSetMarginBottom`.

The main window is created with `Gtk.applicationWindowNew`. Its title is set to the empty string, and it is configured to be non-resizable. Finally, event handlers for button clicks are registered with `Gtk.onButtonClicked`. Once everything is set up, the application window is displayed to the user with `Gtk.widgetShowAll`.

As demonstrated, using GTK+ within Haskell is not too difficult, even if imperative-style programming is required. In most cases, code that works in C has an almost direct equivalent in Haskell.

The package `gi-gtk-hs` provides an idiomatic API on top of `gi-gtk` which allows loading widgets from XML strings. The following listing illustrates an alternative `onActivate` implementation that utilizes this functionality:

```
import Data.GI.Gtk.BuildFn

onActivate' :: (Gtk.IsApplication a, ?self :: a) => G.ApplicationActivateCallback
onActivate' = do
  let buildFn = do
        label <- getObject Gtk.Label (Text.pack "label")
        buttonMinus <- getObject Gtk.Button (Text.pack "buttonMinus")
        buttonPlus <- getObject Gtk.Button (Text.pack "buttonPlus")
        window <- getObject Gtk.ApplicationWindow (Text.pack "window")

        Gtk.onButtonClicked buttonMinus $ do
          t <- Gtk.labelGetLabel label
          Gtk.labelSetLabel label (Text.pack (show (read (Text.unpack t) - 1)))

        Gtk.onButtonClicked buttonPlus $ do
          t <- Gtk.labelGetLabel label
          Gtk.labelSetLabel label (Text.pack (show (read (Text.unpack t) + 1)))

        pure window
```

```
let string = Text.pack
      "<interface>\n\
      \  <object class=\"GtkApplicationWindow\" id=\"window\">\n\
      \    <property name=\"title\"></property>\n\
      \    <property name=\"resizable\">false</property>\n\
      \    <child>\n\
      \      <object class=\"GtkBox\">\n\
      \        <property name=\"orientation\">vertical</property>\n\
      \        <property name=\"spacing\">4</property>\n\
      \        <property name=\"margin\">4</property>\n\
      \        <child>\n\
      \          <object class=\"GtkLabel\" id=\"label\">\n\
      \            <property name=\"label\">0</property>\n\
      \          </object>\n\
      \        </child>\n\
      \        <child>\n\
      \          <object class=\"GtkButtonBox\">\n\
      \            <property name=\"orientation\">horizontal</property>\n\
      \            <property name=\"spacing\">4</property>\n\
      \            <child>\n\
      \              <object class=\"GtkButton\" id=\"buttonMinus\">\n\
      \                <property name=\"label\">-</property>\n\
      \              </object>\n\
      \            </child>\n\
      \            <child>\n\
      \              <object class=\"GtkButton\" id=\"buttonPlus\">\n\
      \                <property name=\"label\">+</property>\n\
      \              </object>\n\
      \            </child>\n\
      \          </object>\n\
      \        </child>\n\
      \      </object>\n\
      \    </child>\n\
      \  </object>\n\
      \</interface>"

builder <- Gtk.builderNewFromString string (fromIntegral (Text.length string))
window <- buildWithBuilder buildFn builder
Gtk.windowSetApplication window (Just ?self)
Gtk.widgetShowAll window
```

5

Conclusions

In this thesis, I demonstrated how to develop an interpreter for a programming language from scratch. I explained some of the theory which underlies programming languages and gave an introduction to Haskell, which was used to implement Devin.

As for Haskell, I explained how type classes can be encoded as implicit parameters; further, I hinted at some of the advantages of type classes over interfaces or abstract classes from object-oriented languages. The concept of type classes is quite powerful: languages such as Rust and Carbon (Google's experimental successor to C++) employ abstractions that can in many ways be compared to them. Scala, a language for the JVM, has implicit parameters built-in; type classes thus emerge naturally [17]. It would be interesting to witness an imperative language that forgoes object-orientation completely, opting for a type class like mechanism instead.

As for Devin, I implemented enough functionality in order to present a good starting point for serious programming language implementation: after all, Devin only lacks a module system and a standard library. Features such as polymorphism and type classes could be added; of course, this would require extending the type system and runtime implementations.

One of the objectives I had for Devin's graphical user interface was to illustrate how source code corresponds to syntax trees; with the GTK+ library, I developed an interface that not only performs syntax highlighting but also visualizes this correspondence. Additionally, Devin's UI implements a debugger; with the breakpoint statement, execution can be suspended at arbitrary locations, allowing for the program's state to be observed.

I showed how a monadic mechanism can be used to represent pausable computations; more complex solutions might be required in different circumstances. Availability of practical material discussing this matter is sparse: in fact, implementing debugging functionality has been one of the most challenging parts of this whole project.

Developing Devin has been insightful for gaining a better understanding of the matter of programming languages. Actual implementation highlighted the types of challenges that arise and require to be solved; using Haskell for such a non-trivial project has taught me how to solve problems functionally, while providing a new perspective on achieving abstraction through type classes. While developing Devin, many ideas for new language constructs arose: in particular, the way in which Scala provides and makes use of implicitly passed parameters seems to be of great power. I researched which other techniques besides monads could be used for achieving side-effects in a purely functional way. One possible alternative is to use so-called *algebraic effects*, a topic which is subject of ongoing research; languages such as Koka [18] provide a way of experimenting with such features. With an unbounded amount of time at hand, it would've been interesting to develop a purely functional language with implicit parameters and an effect system. Alas, this project has taken long enough; still, I'm quite satisfied with what I learned from this experience.

A

Devin source code

A.1 `devin.cabal`

```
cabal-version: 2.2

name: devin
version: 1.4.0
synopsis: Devin programming language
author: Matteo Morena
category: Language

common common
  default-language: Haskell2010

  ghc-options:
    -Wall
    -Wextra
    -Wno-name-shadowing
    -Wno-type-defaults
    -Wno-unused-do-bind

library
  import: common
  hs-source-dirs: src

  exposed-modules:
    Devin.Display,
    Devin.Error,
    Devin.Evaluator,
    Devin.Evaluators,
    Devin.Interval,
    Devin.Parsec,
    Devin.Parsers,
    Devin.Ratio,
    Devin.Syntax,
    Devin.Type,
    Devin.Typer,
    Devin.Typers

  build-depends:
    -- Haskell Platform, extra
    base,
```

```
    vector,
    parsec,
    extra,

test-suite devin-test
  import: common
  type: exitcode-stdio-1.0
  hs-source-dirs: test
  main-is: Main.hs

  other-modules:
    Devin.EvaluatorsSpec,
    Devin.ParsersSpec,
    Devin.TypersSpec

  build-depends:
    -- Haskell Platform
    base,
    parsec,

    -- Others
    hspec,
    devin

  build-tool-depends:
    hspec-discover:hspec-discover

executable devin
  import: common
  hs-source-dirs: app
  main-is: Main.hs

  other-modules:
    Devin.Debug.Evaluator,
    Devin.Debug.Syntax,
    Devin.Highlight,
    Devin.Highlight.Brackets,
    Devin.Highlight.Syntax,
    Devin.Levenshtein

  build-depends:
    -- Haskell Platform, extra
    base,
    containers,
    text,
    vector,
    parsec,
    extra,

    -- haskell-gi
    haskell-gi-overloading == 0.0,
    haskell-gi-base,
    gi-gobject,
    gi-glib,
    gi-gio,
    gi-gdk,
    gi-gtk < 4,
```

```
    gi-gtksource,
    gi-gtk-hs,

    -- Others
    devin

  ghc-options:
    -threaded
```

A.2 src/Devin/Display.hs

```
module Devin.Display (Display (..)) where


class Display a where
  {-# MINIMAL display | displays #-}


  display :: a -> String
  display x = displays x ""


  displays :: a -> ShowS
  displays x = showString (display x)
```

A.3 src/Devin/Error.hs

```
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE DisambiguateRecordFields #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE InstanceSigs #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE NamedFieldPuns #-}

module Devin.Error (Error (..)) where

import Data.Data
import Data.Int

import Devin.Display
import Devin.Interval
import Devin.Syntax
import Devin.Type


data Error where
  UnknownVar :: {
    varName :: String,
    interval :: (Int, Int)
  } -> Error

  UnknownFun :: {
    funName :: String,
    interval :: (Int, Int)
```

```
} -> Error

UnknownType :: {
  typeName :: String,
  interval :: (Int, Int)
} -> Error

InvalidUnary :: {
  unary :: UnaryOperator,
  operandT :: Type
} -> Error

InvalidBinary :: {
  binary :: BinaryOperator,
  leftT :: Type,
  rightT :: Type
} -> Error

InvalidType :: {
  expression :: Expression,
  expectedT :: Type,
  actualT :: Type
} -> Error

-- Static errors:

MissingReturnValue :: {
  statement :: Statement,
  expectedT :: Type
} -> Error

MissingReturnStatement :: {
  funId :: SymbolId
} -> Error

-- Runtime errors:

IntegerOverflow :: {
  expression :: Expression
} -> Error

DivisionByZero :: {
  expression :: Expression
} -> Error

IndexOutOfBounds :: {
  expression :: Expression,
  value :: Int64
} -> Error

InvalidArgCount :: {
  expression :: Expression,
  expected :: Int,
  actual :: Int
} -> Error

AssertionFailed :: {
```

```
    statement :: Statement
  } -> Error

  deriving (Show, Read, Data)


instance Interval Error where
  start :: Num a => Error -> a
  start UnknownVar{interval} = start interval
  start UnknownFun{interval} = start interval
  start UnknownType{interval} = start interval
  start InvalidUnary{unary} = start unary
  start InvalidBinary{binary} = start binary
  start InvalidType{expression} = start expression
  start MissingReturnValue{statement} = start statement
  start MissingReturnStatement{funId} = start funId
  start IntegerOverflow{expression} = start expression
  start DivisionByZero{expression} = start expression
  start IndexOutOfBounds{expression} = start expression
  start InvalidArgCount{expression} = start expression
  start AssertionFailed{statement} = start statement


  end :: Num a => Error -> a
  end UnknownVar{interval} = end interval
  end UnknownFun{interval} = end interval
  end UnknownType{interval} = end interval
  end InvalidUnary{unary} = end unary
  end InvalidBinary{binary} = end binary
  end InvalidType{expression} = end expression
  end MissingReturnValue{statement} = end statement
  end MissingReturnStatement{funId} = end funId
  end IntegerOverflow{expression} = end expression
  end DivisionByZero{expression} = end expression
  end IndexOutOfBounds{expression} = end expression
  end InvalidArgCount{expression} = end expression
  end AssertionFailed{statement} = end statement


instance Display Error where
  displays :: Error -> ShowS
  displays = \case
    UnknownVar{varName} ->
      showString "Unknown variable: " .
      showString varName

    UnknownFun{funName} ->
      showString "Unknown function: " .
      showString funName

    UnknownType{typeName} ->
      showString "Unknown type: " .
      showString typeName

    InvalidUnary{unary, operandT} ->
      showString "Can't apply `" .
      displays unary .
```

```
      showString "' to " .
      displays operandT

    InvalidBinary{binary, leftT, rightT} ->
      showString "Can't apply `" .
      displays binary .
      showString "' to " .
      displays leftT .
      showString " and " .
      displays rightT

    InvalidType{expectedT, actualT} ->
      showString "Invalid type: expected " .
      displays expectedT .
      showString ", but got " .
      displays actualT

    MissingReturnValue{} ->
      showString "Missing return value"

    MissingReturnStatement{} ->
      showString "Missing return statement"

    IntegerOverflow{} ->
      showString "Integer overflow"

    DivisionByZero{} ->
      showString "Division by zero"

    IndexOutOfBounds{value} ->
      showString "Index out of bounds: " .
      shows value

    InvalidArgCount{expected, actual} ->
      showString "Invalid argument count: expected " .
      shows expected .
      showString " arguments, but got " .
      shows actual

    AssertionFailed{statement} | AssertStatement{predicate} <- statement ->
      showString "Assertion failed: " .
      displays predicate

    AssertionFailed{} ->
      showString "Assertion failed"
```

A.4 src/Devin/Evaluator.hs

```
{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE InstanceSigs #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE NamedFieldPuns #-}

module Devin.Evaluator (
```

```
  Evaluator,
  Result (..),
  State,
  Frame (..),
  Function (..),
  Value (..),
  Cell,
  runEvaluatorStep,
  runEvaluator,
  makePredefinedState,
  getType,
  cloneVal,
  compareVals,
  newCell,
  readCell,
  writeCell,
  cloneCell,
  compareCells,
  defineFun,
  lookupFun,
  defineVar,
  lookupVar,
  withNewFrame,
  yield,
  raise
) where

import Control.Applicative
import Control.Monad.IO.Class
import Data.Data
import Data.Int
import Data.IORef

import qualified Data.Vector as Vector
import Data.Vector (Vector, (!))

import Data.Foldable.Extra (allM)

import Devin.Error
import Devin.Syntax
import qualified Devin.Type as Type
import Devin.Type (Type, (<:))


newtype Evaluator a = Evaluator (State -> IO (Result a, State))
  deriving Functor


data Result a
  = Done a
  | Yield Statement (Evaluator a)
  | Error Error
  deriving Functor


type State = [Frame]
```

```
data Frame = Frame {
  label :: Maybe String,
  poffset :: Int,  -- Static link
  funs :: [(String, Function)],
  vars :: [(String, Cell)]
}


data Function
  = BuiltinNot
  | BuiltinLen
  | BuiltinIntToFloat
  | BuiltinFloatToInt
  | UserDefined Definition
  deriving (Eq, Show, Read, Data)


data Value
  = Unit
  | Bool Bool
  | Int Int64
  | Float Double
  | Array (Vector Cell)


newtype Cell = Cell (IORef Value)


instance Applicative Evaluator where
  pure :: a -> Evaluator a
  pure x = Evaluator (\state -> pure (Done x, state))


  liftA2 :: (a -> b -> c) -> Evaluator a -> Evaluator b -> Evaluator c
  liftA2 f mx my = Evaluator $ \state -> do
    (result, state') <- runEvaluatorStep mx state

    case result of
      Done x -> runEvaluatorStep (f x <$> my) state'
      Yield statement mx -> pure (Yield statement (liftA2 f mx my), state')
      Error error -> pure (Error error, state')


instance Monad Evaluator where
  (>>=) :: Evaluator a -> (a -> Evaluator b) -> Evaluator b
  mx >>= f = Evaluator $ \state -> do
    (result, state') <- runEvaluatorStep mx state

    case result of
      Done x -> runEvaluatorStep (f x) state'
      Yield statement mx -> pure (Yield statement (f =<< mx), state')
      Error error -> pure (Error error, state')


instance MonadIO Evaluator where
  liftIO :: IO a -> Evaluator a
```

```
  liftIO mx = Evaluator $ \state -> do
    x <- mx
    pure (Done x, state)


instance MonadFail Evaluator where
  fail :: String -> Evaluator a
  fail message = liftIO (fail message)


runEvaluatorStep :: MonadIO m => Evaluator a -> State -> m (Result a, State)
runEvaluatorStep (Evaluator f) state = liftIO (f state)


runEvaluator :: MonadIO m => Evaluator a -> State -> m (Either Error a, State)
runEvaluator mx state = do
  (result, state') <- runEvaluatorStep mx state

  case result of
    Done x -> pure (Right x, state')
    Yield _ mx -> runEvaluator mx state'
    Error error -> pure (Left error, state')


makePredefinedState :: MonadIO m => m State
makePredefinedState = liftIO $ do
  let f1 = ("not", BuiltinNot)
  let f2 = ("len", BuiltinLen)
  let f3 = ("intToFloat", BuiltinIntToFloat)
  let f4 = ("floatToInt", BuiltinFloatToInt)

  v1 <- (,) "true" <$> newCell (Bool True)
  v2 <- (,) "false" <$> newCell (Bool False)
  v3 <- (,) "unit" <$> newCell Unit

  pure [Frame Nothing 0 [f4, f3, f2, f1] [v3, v2, v1]]


getType :: MonadIO m => Value -> m Type
getType = \case
  Unit -> pure Type.Unit
  Bool _ -> pure Type.Bool
  Int _ -> pure Type.Int
  Float _ -> pure Type.Float

  Array cells | Vector.null cells -> pure (Type.Array Type.Unknown)

  Array cells -> do
    cell <- readCell (Vector.head cells)
    t <- getType cell

    let f Type.Unknown = False
        f t' = t' <: t

    ok <- allM (\cell -> f <$> (getType =<< readCell cell)) cells
    pure (Type.Array (if ok then t else Type.Unknown))
```

```
cloneVal :: MonadIO m => Value -> m Value
cloneVal = \case
  Unit -> pure Unit
  Bool x -> pure (Bool x)
  Int x -> pure (Int x)
  Float x -> pure (Float x)

  Array cells -> do
    cells' <- Vector.forM cells cloneCell
    pure (Array cells')


compareVals :: MonadIO m => Value -> Value -> m (Either (Type, Type) Ordering)
compareVals val1 val2 = case (val1, val2) of
  (Unit, Unit) -> pure (Right EQ)
  (Bool x, Bool y) -> pure (Right (compare x y))
  (Int x, Int y) -> pure (Right (compare x y))
  (Float x, Float y) -> pure (Right (compare x y))

  (Array cells1, Array cells2) -> go (Vector.length cells1) (Vector.length cells2) 0
    where
      go n1 n2 i | i >= n1 || i >= n2 =
        pure (Right (compare n1 n2))

      go n1 n2 i = do
        val1 <- readCell (cells1 ! i)
        val2 <- readCell (cells2 ! i)

        compareVals val1 val2 >>= \case
          Right EQ -> go n1 n2 (i + 1)
          Right ordering -> pure (Right ordering)
          Left (t1, t2) -> pure (Left (t1, t2))

  (_, _) -> do
    t1 <- getType val1
    t2 <- getType val2
    pure (Left (t1, t2))


newCell :: MonadIO m => Value -> m Cell
newCell val = liftIO $ do
  ref <- newIORef val
  pure (Cell ref)


readCell :: MonadIO m => Cell -> m Value
readCell (Cell ref) = liftIO (readIORef ref)


writeCell :: MonadIO m => Cell -> Value -> m Cell
writeCell (Cell ref) val = liftIO $ do
  writeIORef ref val
  pure (Cell ref)


cloneCell :: MonadIO m => Cell -> m Cell
```

```
cloneCell cell = do
  val <- readCell cell
  val' <- cloneVal val
  newCell val'


compareCells :: MonadIO m => Cell -> Cell -> m (Either (Type, Type) Ordering)
compareCells cell1 cell2 = do
  val1 <- readCell cell1
  val2 <- readCell cell2
  compareVals val1 val2


defineFun :: String -> Function -> Evaluator ()
defineFun name fun = Evaluator $ \case
  [] -> pure (Done (), [Frame Nothing 0 [(name, fun)] []])

  frame : frames -> do
    let funs' = (name, fun) : funs frame
    pure (Done (), frame{funs = funs'} : frames)


lookupFun :: String -> Evaluator (Maybe (Function, Int))
lookupFun name = Evaluator (\state -> pure (Done (go 0 state), state))
  where
    go _ [] = Nothing

    go depth (Frame{poffset, funs} : frames) = case lookup name funs of
      Just fun -> Just (fun, depth)
      Nothing -> go (depth + max 1 poffset) (drop (poffset - 1) frames)


defineVar :: String -> Cell -> Evaluator ()
defineVar name cell = Evaluator $ \case
  [] -> pure (Done (), [Frame Nothing 0 [] [(name, cell)]])

  frame : frames -> do
    let vars' = (name, cell) : vars frame
    pure (Done (), frame{vars = vars'} : frames)


lookupVar :: String -> Evaluator (Maybe (Cell, Int))
lookupVar name = Evaluator (\state -> pure (Done (go 0 state), state))
  where
    go _ [] = Nothing

    go depth (Frame{poffset, vars} : frames) = case lookup name vars of
      Just cell -> Just (cell, depth)
      Nothing -> go (depth + max 1 poffset) (drop (poffset - 1) frames)


withNewFrame :: Maybe String -> Int -> Evaluator a -> Evaluator a
withNewFrame label poffset mx = do
  pushFrame
  x <- mx
  popFrame
  pure x
```

```
  where
    pushFrame = Evaluator $ \state ->
      pure (Done (), Frame label poffset [] [] : state)

    popFrame = Evaluator $ \state ->
      pure (Done (), tail state)


yield :: Statement -> Evaluator ()
yield statement = Evaluator $ \state ->
  pure (Yield statement (pure ()), state)


raise :: Error -> Evaluator a
raise error = Evaluator $ \state ->
  pure (Error error, state)
```

A.5 src/Devin/Evaluators.hs

```
{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE DisambiguateRecordFields #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE NamedFieldPuns #-}

module Devin.Evaluators (
  evalDevin,
  evalDefinition,
  evalStatement,
  evalExpression
) where

import Data.Bits
import Data.Foldable
import Data.Traversable
import Numeric

import qualified Data.Vector as Vector
import Data.Vector ((!), (!?))

import Control.Monad.Extra

import Devin.Error
import Devin.Evaluator
import qualified Devin.Type as Type
import Devin.Syntax


evalDevin :: Devin -> Evaluator ()
evalDevin Devin{definitions} = void $ do
  for_ definitions evalDefinition1
  for_ definitions evalDefinition2
  Just (UserDefined FunDefinition{params = [], body}, depth) <- lookupFun "main"
  withNewFrame (Just "main") (depth + 1) (evalStatement body)
```

```
evalDefinition :: Definition -> Evaluator ()
evalDefinition definition = do
  evalDefinition1 definition
  evalDefinition2 definition


evalDefinition1 :: Definition -> Evaluator ()
evalDefinition1 definition = case definition of
  VarDefinition{} -> pure ()
  FunDefinition{funId = SymbolId{name}} -> defineFun name (UserDefined definition)


evalDefinition2 :: Definition -> Evaluator ()
evalDefinition2 = \case
  VarDefinition{varId = SymbolId{name}, value} -> do
    cell <- evalExpression value
    cell' <- cloneCell cell
    defineVar name cell'

  FunDefinition{} -> pure ()


evalStatement :: Statement -> Evaluator (Maybe Cell)
evalStatement statement = do
  yield statement

  case statement of
    DefinitionStatement{definition} -> do
      evalDefinition definition
      pure Nothing

    ExpressionStatement{effect} -> do
      evalExpression effect
      pure Nothing

    IfStatement{predicate, trueBranch} -> do
      cell <- evalExpression predicate
      val <- readCell cell

      case val of
        Bool True -> withNewFrame Nothing 1 (evalStatement trueBranch)
        Bool False -> pure Nothing

        _ -> do
          t <- getType val
          raise (InvalidType predicate Type.Bool t)

    IfElseStatement{predicate, trueBranch, falseBranch} -> do
      val <- evalExpression predicate
      cell <- readCell val

      case cell of
        Bool True -> withNewFrame Nothing 1 (evalStatement trueBranch)
        Bool False -> withNewFrame Nothing 1 (evalStatement falseBranch)

        _ -> do
          t <- getType cell
```

```
      raise (InvalidType predicate Type.Bool t)

WhileStatement{predicate, body} -> untilJustM $ do
  cell <- evalExpression predicate
  val <- readCell cell

  case val of
    Bool False -> pure (Just Nothing)

    Bool True -> withNewFrame Nothing 1 (evalStatement body) >>= \case
      Just cell -> pure (Just (Just cell))
      Nothing -> pure Nothing

    _ -> do
      t <- getType val
      raise (InvalidType predicate Type.Bool t)

DoWhileStatement{body, predicate} -> untilJustM $
  withNewFrame Nothing 1 (evalStatement body) >>= \case
    Just cell -> pure (Just (Just cell))

    Nothing -> do
      cell <- evalExpression predicate
      val <- readCell cell

      case val of
        Bool False -> pure (Just Nothing)
        Bool True -> pure Nothing

        _ -> do
          t <- getType val
          raise (InvalidType predicate Type.Bool t)

ReturnStatement{result = Just result} -> do
  cell <- evalExpression result
  pure (Just cell)

ReturnStatement{result = Nothing} -> do
  cell <- newCell Unit
  pure (Just cell)

AssertStatement{predicate} -> do
  cell <- evalExpression predicate
  val <- readCell cell

  case val of
    Bool False -> raise (AssertionFailed statement)
    Bool True -> pure Nothing

    _ -> do
      t <- getType val
      raise (InvalidType predicate Type.Bool t)

BreakpointStatement{} -> do
  pure Nothing

BlockStatement{statements} -> withNewFrame Nothing 1 $ do
```

```
      for_ statements $ \case
        DefinitionStatement{definition} -> evalDefinition1 definition
        _ -> pure ()

      flip firstJustM statements $ \case
        DefinitionStatement{definition} -> do
          evalDefinition2 definition
          pure Nothing

        statement -> evalStatement statement


evalExpression :: Expression -> Evaluator Cell
evalExpression expression = case expression of
  IntegerExpression{integer} -> case toIntegralSized integer of
    Just x -> newCell (Int x)
    Nothing -> raise (IntegerOverflow expression)

  RationalExpression{rational} -> newCell (Float (fromRat rational))

  VarExpression{varName, interval} -> lookupVar varName >>= \case
    Just (cell, _) -> pure cell
    Nothing -> raise (UnknownVar varName interval)

  ArrayExpression{elems} -> do
    cells <- flip Vector.unfoldrM elems $ \case
      [] -> pure Nothing

      (elem : elems) -> do
        cell <- evalExpression elem
        cell' <- cloneCell cell
        pure (Just (cell', elems))

    newCell (Array cells)

  AccessExpression{array, index} -> do
    arrayCell <- evalExpression array
    arrayVal <- readCell arrayCell

    indexCell <- evalExpression index
    indexVal <- readCell indexCell

    case (arrayVal, indexVal) of
      (Array cells, Int x) -> case toIntegralSized x of
        Just n | Just cell <- cells !? n -> pure cell
        _ -> raise (IndexOutOfBounds index x)

      (Array _, _) -> do
        indexT <- getType indexVal
        raise (InvalidType index Type.Int indexT)

      (_, _) -> do
        arrayT <- getType arrayVal
        raise (InvalidType array (Type.Array Type.Unknown) arrayT)

  CallExpression{funId = SymbolId{name, interval}, args} -> do
    argCells <- for args evalExpression
```

```
lookupFun name >>= \case
  Just (BuiltinNot, _) | [cell] <- argCells -> do
    val <- readCell cell

    case val of
      Bool x -> newCell (Bool (not x))

      _ -> do
        argT <- getType val
        raise (InvalidType (head args) Type.Bool argT)

  Just (BuiltinNot, _) ->
    raise (InvalidArgCount expression 1 (length argCells))

  Just (BuiltinLen, _) | [cell] <- argCells -> do
    val <- readCell cell

    case val of
      Array cells -> newCell (Int (fromIntegral (length cells)))

      _ -> do
        argT <- getType val
        raise (InvalidType (head args) (Type.Array Type.Unknown) argT)

  Just (BuiltinLen, _) ->
    raise (InvalidArgCount expression 1 (length argCells))

  Just (BuiltinIntToFloat, _) | [cell] <- argCells -> do
    val <- readCell cell

    case val of
      Int x -> newCell (Float (fromIntegral x))

      _ -> do
        argT <- getType val
        raise (InvalidType (head args) Type.Int argT)

  Just (BuiltinIntToFloat, _) ->
    raise (InvalidArgCount expression 1 (length argCells))

  Just (BuiltinFloatToInt, _) | [cell] <- argCells -> do
    val <- readCell cell

    case val of
      Float x -> newCell (Int (round x))

      _ -> do
        argT <- getType val
        raise (InvalidType (head args) Type.Float argT)

  Just (BuiltinFloatToInt, _) ->
    raise (InvalidArgCount expression 1 (length argCells))

  Just (UserDefined FunDefinition{params, body}, depth) ->
    withNewFrame (Just name) (depth + 1) (go 0 params argCells)
```

```
      where
        -- Pass argument by value:
        go n ((Nothing, SymbolId{name}, _) : params) (argCell : argCells) = do
          argCell' <- cloneCell argCell
          defineVar name argCell'
          go (n + 1) params argCells

        -- Pass argument by reference:
        go n ((Just _, SymbolId{name}, _) : params) (argCell : argCells) = do
          defineVar name argCell
          go (n + 1) params argCells

        -- If argument count is correct:
        go _ [] [] = evalStatement body >>= \case
          Just cell -> cloneCell cell
          Nothing -> newCell Unit

        -- If argument count is incorrect:
        go n params argCells = do
          let expected = n + length params
          let actual = n + length argCells
          raise (InvalidArgCount expression expected actual)

    _ -> raise (UnknownFun name interval)

UnaryExpression{unary, operand} | PlusOperator{} <- unary -> do
  cell <- evalExpression operand
  val <- readCell cell

  case val of
    Int x -> newCell (Int x)
    Float x -> newCell (Float x)

    _ -> do
      operandT <- getType val
      raise (InvalidUnary unary operandT)

UnaryExpression{unary, operand} | MinusOperator{} <- unary -> do
  cell <- evalExpression operand
  val <- readCell cell

  case val of
    Int x | Just y <- safeUnary negate x -> newCell (Int y)
    Int _ -> raise (IntegerOverflow expression)

    Float x -> newCell (Float (negate x))

    _ -> do
      operandT <- getType val
      raise (InvalidUnary unary operandT)

BinaryExpression{left, binary, right} | AddOperator{} <- binary -> do
  leftCell <- evalExpression left
  leftVal <- readCell leftCell

  rightCell <- evalExpression right
  rightVal <- readCell rightCell
```

```
  case (leftVal, rightVal) of
    (Int x, Int y) | Just z <- safeBinary (+) x y -> newCell (Int z)
    (Int _, Int _) -> raise (IntegerOverflow expression)

    (Float x, Float y) -> newCell (Float (x + y))

    (Array rs1, Array rs2) -> do
      let n1 = Vector.length rs1
      let n2 = Vector.length rs2

      case safeBinary (+) n1 n2 of
        Nothing -> raise (IntegerOverflow expression)

        Just n3 -> do
          let f i = cloneCell (if i < n1 then rs1 ! i else rs2 ! (i - n1))
          cells <- Vector.generateM n3 f
          newCell (Array cells)

    (_, _) -> do
      leftT <- getType leftVal
      rightT <- getType rightVal
      raise (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | SubtractOperator{} <- binary -> do
  leftCell <- evalExpression left
  leftVal <- readCell leftCell

  rightCell <- evalExpression right
  rightVal <- readCell rightCell

  case (leftVal, rightVal) of
    (Int x, Int y) | Just z <- safeBinary (-) x y -> newCell (Int z)
    (Int _, Int _) -> raise (IntegerOverflow expression)

    (Float x, Float y) -> newCell (Float (x - y))

    (_, _) -> do
      leftT <- getType leftVal
      rightT <- getType rightVal
      raise (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | MultiplyOperator{} <- binary -> do
  leftCell <- evalExpression left
  leftVal <- readCell leftCell

  rightCell <- evalExpression right
  rightVal <- readCell rightCell

  case (leftVal, rightVal) of
    (Int x, Int y) | Just z <- safeBinary (*) x y -> newCell (Int z)
    (Int _, Int _) -> raise (IntegerOverflow expression)

    (Float x, Float y) -> newCell (Float (x * y))

    (Int x, Array _) | x <= 0 -> newCell (Array Vector.empty)
    (Array _, Int y) | y <= 0 -> newCell (Array Vector.empty)
```

```
    (Int x, Array cells) -> do
      let n = Vector.length cells

      case safeBinary (*) x n of
        Nothing -> raise (IntegerOverflow expression)

        Just n' -> do
          cells' <- Vector.generateM n' (\i -> cloneCell (cells ! (i `mod` n)))
          newCell (Array cells')

    (Array cells, Int y) -> do
      let n = Vector.length cells

      case safeBinary (*) n y of
        Nothing -> raise (IntegerOverflow expression)

        Just n' -> do
          cells' <- Vector.generateM n' (\i -> cloneCell (cells ! (i `mod` n)))
          newCell (Array cells')

    (_, _) -> do
      leftT <- getType leftVal
      rightT <- getType rightVal
      raise (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | DivideOperator{} <- binary -> do
  leftCell <- evalExpression left
  leftVal <- readCell leftCell

  rightCell <- evalExpression right
  rightVal <- readCell rightCell

  case (leftVal, rightVal) of
    (Int _, Int 0) -> raise (DivisionByZero expression)
    (Int x, Int y) | Just z <- safeBinary div x y -> newCell (Int z)
    (Int _, Int _) -> raise (IntegerOverflow expression)

    (Float x, Float y) -> newCell (Float (x / y))

    (_, _) -> do
      leftT <- getType leftVal
      rightT <- getType rightVal
      raise (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | ModuloOperator{} <- binary -> do
  leftCell <- evalExpression left
  leftVal <- readCell leftCell

  rightCell <- evalExpression right
  rightVal <- readCell rightCell

  case (leftVal, rightVal) of
    (Int _, Int 0) -> raise (DivisionByZero expression)
    (Int x, Int y) | Just z <- safeBinary mod x y -> newCell (Int z)
    (Int _, Int _) -> raise (IntegerOverflow expression)
```

```
    (_, _) -> do
      leftT <- getType leftVal
      rightT <- getType rightVal
      raise (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | EqualOperator{} <- binary -> do
  leftCell <- evalExpression left
  rightCell <- evalExpression right

  compareCells leftCell rightCell >>= \case
    Right ordering -> newCell (Bool (ordering == EQ))
    Left (leftT, rightT) -> raise (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | NotEqualOperator{} <- binary -> do
  leftCell <- evalExpression left
  rightCell <- evalExpression right

  compareCells leftCell rightCell >>= \case
    Right ordering -> newCell (Bool (ordering /= EQ))
    Left (leftT, rightT) -> raise (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | LessOperator{} <- binary -> do
  leftCell <- evalExpression left
  rightCell <- evalExpression right

  compareCells leftCell rightCell >>= \case
    Right ordering -> newCell (Bool (ordering < EQ))
    Left (leftT, rightT) -> raise (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | LessOrEqualOperator{} <- binary -> do
  leftCell <- evalExpression left
  rightCell <- evalExpression right

  compareCells leftCell rightCell >>= \case
    Right ordering -> newCell (Bool (ordering <= EQ))
    Left (leftT, rightT) -> raise (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | GreaterOperator{} <- binary -> do
  leftCell <- evalExpression left
  rightCell <- evalExpression right

  compareCells leftCell rightCell >>= \case
    Right ordering -> newCell (Bool (ordering > EQ))
    Left (leftT, rightT) -> raise (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | GreaterOrEqualOperator{} <- binary -> do
  leftCell <- evalExpression left
  rightCell <- evalExpression right

  compareCells leftCell rightCell >>= \case
    Right ordering -> newCell (Bool (ordering >= EQ))
    Left (leftT, rightT) -> raise (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | AndOperator{} <- binary -> do
  leftCell <- evalExpression left
  leftVal <- readCell leftCell
```

```
    case leftVal of
      Bool False -> newCell (Bool False)

      _ -> do
        rightCell <- evalExpression right
        rightVal <- readCell rightCell

        case (leftVal, rightVal) of
          (Bool _, Bool y) -> newCell (Bool y)

          (_, _) -> do
            leftT <- getType leftVal
            rightT <- getType rightVal
            raise (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary, right} | OrOperator{} <- binary -> do
    leftCell <- evalExpression left
    leftVal <- readCell leftCell

    case leftVal of
      Bool True -> newCell (Bool True)

      _ -> do
        rightCell <- evalExpression right
        rightVal <- readCell rightCell

        case (leftVal, rightVal) of
          (Bool _, Bool y) -> newCell (Bool y)

          (_, _) -> do
            leftT <- getType leftVal
            rightT <- getType rightVal
            raise (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary, right} | XorOperator{} <- binary -> do
    leftCell <- evalExpression left
    leftVal <- readCell leftCell

    rightCell <- evalExpression right
    rightVal <- readCell rightCell

    case (leftVal, rightVal) of
      (Bool x, Bool y) -> newCell (Bool (x /= y))

      (_, _) -> do
        leftT <- getType leftVal
        rightT <- getType rightVal
        raise (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary = PlainAssignOperator{}, right} -> do
    leftCell <- evalExpression left

    rightCell <- evalExpression right
    rightVal <- readCell rightCell

    rightVal' <- cloneVal rightVal
    writeCell leftCell rightVal'
```

```
BinaryExpression{left, binary, right} | AddAssignOperator{} <- binary -> do
  leftCell <- evalExpression left
  leftVal <- readCell leftCell

  rightCell <- evalExpression right
  rightVal <- readCell rightCell

  case (leftVal, rightVal) of
    (Int x, Int y) | Just z <- safeBinary (+) x y -> writeCell leftCell (Int z)
    (Int _, Int _) -> raise (IntegerOverflow expression)

    (Float x, Float y) -> writeCell leftCell (Float (x + y))

    (_, _) -> do
      leftT <- getType leftVal
      rightT <- getType rightVal
      raise (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | SubtractAssignOperator{} <- binary -> do
  leftCell <- evalExpression left
  leftVal <- readCell leftCell

  rightCell <- evalExpression right
  rightVal <- readCell rightCell

  case (leftVal, rightVal) of
    (Int x, Int y) | Just z <- safeBinary (-) x y -> writeCell leftCell (Int z)
    (Int _, Int _) -> raise (IntegerOverflow expression)

    (Float x, Float y) -> writeCell leftCell (Float (x - y))

    (_, _) -> do
      leftT <- getType leftVal
      rightT <- getType rightVal
      raise (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | MultiplyAssignOperator{} <- binary -> do
  leftCell <- evalExpression left
  leftVal <- readCell leftCell

  rightCell <- evalExpression right
  rightVal <- readCell rightCell

  case (leftVal, rightVal) of
    (Int x, Int y) | Just z <- safeBinary (*) x y -> writeCell leftCell (Int z)
    (Int _, Int _) -> raise (IntegerOverflow expression)

    (Float x, Float y) -> writeCell leftCell (Float (x * y))

    (Array cells, Int y) -> do
      let n = Vector.length cells

      case safeBinary (*) n y of
        Nothing -> raise (IntegerOverflow expression)

        Just n' -> do
```

```
            cells' <- Vector.generateM n' (\i -> cloneCell (cells ! (i `mod` n)))
            writeCell leftCell (Array cells')

      (_, _) -> do
        leftT <- getType leftVal
        rightT <- getType rightVal
        raise (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary, right} | DivideAssignOperator{} <- binary -> do
    leftCell <- evalExpression left
    leftVal <- readCell leftCell

    rightCell <- evalExpression right
    rightVal <- readCell rightCell

    case (leftVal, rightVal) of
      (Int _, Int 0) -> raise (DivisionByZero expression)
      (Int x, Int y) | Just z <- safeBinary div x y -> writeCell leftCell (Int z)
      (Int _, Int _) -> raise (IntegerOverflow expression)

      (Float x, Float y) -> writeCell leftCell (Float (x / y))

      (_, _) -> do
        leftT <- getType leftVal
        rightT <- getType rightVal
        raise (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary, right} | ModuloAssignOperator{} <- binary -> do
    leftCell <- evalExpression left
    leftVal <- readCell leftCell

    rightCell <- evalExpression right
    rightVal <- readCell rightCell

    case (leftVal, rightVal) of
      (Int _, Int 0) -> raise (DivisionByZero expression)
      (Int x, Int y) | Just z <- safeBinary mod x y -> writeCell leftCell (Int z)
      (Int _, Int _) -> raise (IntegerOverflow expression)

      (_, _) -> do
        leftT <- getType leftVal
        rightT <- getType rightVal
        raise (InvalidBinary binary leftT rightT)

  ParenthesizedExpression{inner} -> evalExpression inner

  where
    safeUnary op x = toIntegralSized (op (toInteger x))
    safeBinary op x y = toIntegralSized (toInteger x `op` toInteger y)
```

A.6 src/Devin/Interval.hs

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE InstanceSigs #-}

module Devin.Interval (Interval (..)) where
```

```
class Interval a where
  start :: Num b => a -> b
  end :: Num b => a -> b


instance Integral a => Interval (a, a) where
  start :: Num b => (a, a) -> b
  start interval = fromIntegral (fst interval)


  end :: Num b => (a, a) -> b
  end interval = fromIntegral (snd interval)
```

A.7 src/Devin/Parsec.hs

```
{-# OPTIONS_GHC -Wno-orphans #-}

{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE InstanceSigs #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE UndecidableInstances #-}

module Devin.Parsec (
  module Text.Parsec,
  getOffset,
  toOffsetT,
  toOffset
) where

import Data.Functor.Identity

import Text.Parsec
import Text.Parsec.Pos


instance (Num a, Stream s m t) => Stream (a, s) m t where
  uncons :: (a, s) -> m (Maybe (t, (a, s)))
  uncons (offset, stream) = uncons stream >>= \case
    Just (token, rest) -> pure (Just (token, (offset + 1, rest)))
    Nothing -> pure Nothing


getOffset :: Monad m => ParsecT (a, s) u m a
getOffset = do
  State{stateInput = (offset, _)} <- getParserState
  pure offset


toOffsetT :: (Num a, Stream s m Char) => SourcePos -> s -> m a
toOffsetT sourcePos stream = go 0 (initialPos "") stream
  where
```

```
    go result sourcePos' _ | sourcePos' >= sourcePos = pure result

    go result sourcePos' stream = uncons stream >>= \case
      Just (c, rest) -> go (result + 1) (updatePosChar sourcePos' c) rest
      Nothing -> pure result


toOffset :: (Num a, Stream s Identity Char) => SourcePos -> s -> a
toOffset sourcePos stream = runIdentity (toOffsetT sourcePos stream)
```

A.8 src/Devin/Parsers.hs

```
{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE DisambiguateRecordFields #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE NamedFieldPuns #-}

module Devin.Parsers (
  Parser,
  ParserT,
  State,
  parse,
  parseT,
  devin,
  definition,
  statement,
  expression,
  unaryOperator,
  binaryOperator,
  symbolId,
  typeId,
  comment
) where

import Control.Applicative hiding ((<|>), many)
import Control.Monad
import Data.Char
import Data.Functor
import Data.Functor.Identity

import Control.Monad.Extra

import Devin.Parsec hiding (State, parse, token)
import Devin.Syntax hiding (definition)


type Parser s a = Parsec (Int, s) State a
type ParserT s m a = ParsecT (Int, s) State m a
newtype State = State { unState :: [Token] -> [Token] }


parse ::
  Stream s Identity t =>
  Parser s a -> SourceName -> (Int, s) -> Either ParseError (a, [Token])
parse mx sourceName stream =
```

```
  runIdentity (parseT mx sourceName stream)


parseT ::
  Stream s m t =>
  ParserT s m a -> SourceName -> (Int, s) -> m (Either ParseError (a, [Token]))
parseT mx =
  runParserT (liftA2 (,) mx ((\state -> unState state []) <$> getState)) (State id)


devin :: Stream s m Char => ParserT s m Devin
devin = do
  definitions <- s *> many (definition <* s) <* eof
  pure (Devin definitions)


definition :: Stream s m Char => ParserT s m Definition
definition = varDefinition <|> funDefinition


varDefinition :: Stream s m Char => ParserT s m Definition
varDefinition = do
  varKeyword <- keyword "var"
  varId <- s *> symbolId
  equalSign <- s *> token "="
  value <- s *> expression
  semicolon <- s *> token ";"
  pure (VarDefinition varKeyword varId equalSign value semicolon)


funDefinition :: Stream s m Char => ParserT s m Definition
funDefinition = do
  defKeyword <- keyword "def"
  funId <- s *> symbolId
  open <- s *> token "("

  (params, commas) <- s *> optionMaybe param >>= \case
    Nothing -> pure ([], [])

    Just first -> do
      rest <- many (liftA2 (,) (try (s *> token ",")) (s *> param))
      pure (first : map snd rest, map fst rest)

  close <- s *> token ")"

  returnInfo <- s *> optionMaybe (token "->") >>= \case
    Nothing -> pure Nothing

    Just arrow -> do
      returnTypeId <- s *> typeId
      pure (Just (arrow, returnTypeId))

  body <- s *> statement
  pure (FunDefinition defKeyword funId open params commas close returnInfo body)

  where
    param = do
```

```
      (refKeyword, paramId) <- choice
        [
          try $ do
            token <- keyword "ref"
            paramId <- s *> symbolId
            pure (Just token, paramId),

          do
            paramId <- symbolId
            pure (Nothing, paramId)
        ]

      paramInfo <- optionMaybe (try (s *> token ":")) >>= \case
        Nothing -> pure Nothing

        Just colon -> do
          paramTypeId <- s *> typeId
          pure (Just (colon, paramTypeId))

      pure (refKeyword, paramId, paramInfo)


statement :: Stream s m Char => ParserT s m Statement
statement = choice
  [
    try expressionStatement,
    definitionStatement,
    ifStatement,
    whileStatement,
    doWhileStatement,
    returnStatement,
    assertStatement,
    breakpointStatement,
    blockStatement
  ]


definitionStatement :: Stream s m Char => ParserT s m Statement
definitionStatement = DefinitionStatement <$> definition


expressionStatement :: Stream s m Char => ParserT s m Statement
expressionStatement = do
  value <- expression6 True
  semicolon <- s *> token ";"
  pure (ExpressionStatement value semicolon)


ifStatement :: Stream s m Char => ParserT s m Statement
ifStatement = do
  ifKeyword <- keyword "if"
  predicate <- s *> expression
  trueBranch <- s *> statement

  optionMaybe (try (s *> keyword "else")) >>= \case
    Nothing -> pure (IfStatement ifKeyword predicate trueBranch)
```

```
    Just elseKeyword -> do
      falseBranch <- s *> statement
      pure (IfElseStatement ifKeyword predicate trueBranch elseKeyword falseBranch)


whileStatement :: Stream s m Char => ParserT s m Statement
whileStatement = do
  whileKeyword <- keyword "while"
  predicate <- s *> expression
  body <- s *> statement
  pure (WhileStatement whileKeyword predicate body)


doWhileStatement :: Stream s m Char => ParserT s m Statement
doWhileStatement = do
  doKeyword <- keyword "do"
  body <- s *> statement
  whileKeyword <- s *> keyword "while"
  predicate <- s *> expression
  semicolon <- s *> token ";"
  pure (DoWhileStatement doKeyword body whileKeyword predicate semicolon)


returnStatement :: Stream s m Char => ParserT s m Statement
returnStatement = do
  returnKeyword <- keyword "return"
  result <- s *> optionMaybe expression
  semicolon <- s *> token ";"
  pure (ReturnStatement returnKeyword result semicolon)


assertStatement :: Stream s m Char => ParserT s m Statement
assertStatement = do
  assertKeyword <- keyword "assert"
  predicate <- s *> expression
  semicolon <- s *> token ";"
  pure (AssertStatement assertKeyword predicate semicolon)


breakpointStatement :: Stream s m Char => ParserT s m Statement
breakpointStatement = do
  breakpointKeyword <- keyword "breakpoint"
  semicolon <- s *> token ";"
  pure (BreakpointStatement breakpointKeyword semicolon)


blockStatement :: Stream s m Char => ParserT s m Statement
blockStatement = do
  open <- token "{"
  statements <- s *> many (statement <* s)
  close <- token "}"
  pure (BlockStatement open statements close)


expression :: Stream s m Char => ParserT s m Expression
expression = expression1
```

```
expression1 :: Stream s m Char => ParserT s m Expression
expression1 = chainl1 expression2 $ do
  binary <- try $ between s s $ syntax $ choice
    [
      keyword "and" $> AndOperator,
      keyword "or" $> OrOperator,
      keyword "xor" $> XorOperator
    ]

  pure (\left right -> BinaryExpression left binary right)


expression2 :: Stream s m Char => ParserT s m Expression
expression2 = chainl1 expression3 $ do
  binary <- try $ between s s $ syntax $ choice
    [
      text "==" $> EqualOperator,
      text "!=" $> NotEqualOperator
    ]

  pure (\left right -> BinaryExpression left binary right)


expression3 :: Stream s m Char => ParserT s m Expression
expression3 = chainl1 expression4 $ do
  binary <- try $ between s s $ syntax $ choice
    [
      text ">=" $> GreaterOrEqualOperator,
      text ">" $> GreaterOperator,
      text "<=" $> LessOrEqualOperator,
      text "<" $> LessOperator
    ]

  pure (\left right -> BinaryExpression left binary right)


expression4 :: Stream s m Char => ParserT s m Expression
expression4 = chainl1 expression5 $ do
  binary <- try $ between s s $ syntax $ choice
    [
      text "+" $> AddOperator,
      text "-" $> SubtractOperator
    ]

  pure (\left right -> BinaryExpression left binary right)


expression5 :: Stream s m Char => ParserT s m Expression
expression5 = chainl1 (expression6 False) $ do
  binary <- try $ between s s $ syntax $ choice
    [
      text "*" $> MultiplyOperator,
      text "/" $> DivideOperator,
      text "%" $> ModuloOperator
    ]
```

```
  pure (\left right -> BinaryExpression left binary right)


expression6 :: Stream s m Char => Bool -> ParserT s m Expression
expression6 requireSideEffects = do
  term <- choice
    [
      callOrVarExpression,
      rationalExpression,
      integerExpression,
      arrayExpression,
      unaryExpression,
      parenthesizedExpression
    ]

  left <- flip loopM term $ \term -> optionMaybe (try (s *> token "[")) >>= \case
    Nothing -> pure (Right term)

    Just open -> do
      index <- s *> expression
      close <- s *> token "]"
      pure (Left (AccessExpression term open index close))

  let leftIsCall = case left of
        CallExpression{} -> True
        _ -> False

  if not requireSideEffects || leftIsCall then
    optionMaybe (try (s *> assignOperator)) >>= \case
      Nothing -> pure left

      Just binary -> do
        right <- s *> expression
        pure (BinaryExpression left binary right)
  else do
    binary <- s *> assignOperator
    right <- s *> expression
    pure (BinaryExpression left binary right)

  where
    assignOperator = syntax $ choice
      [
        (try (char '=' <* notFollowedBy (char '=')) <?> "‘=’") $> PlainAssignOperator,
        text "+=" $> AddAssignOperator,
        text "-=" $> SubtractAssignOperator,
        text "*=" $> MultiplyAssignOperator,
        text "/=" $> DivideAssignOperator,
        text "%=" $> ModuloAssignOperator
      ]


integerExpression :: Stream s m Char => ParserT s m Expression
integerExpression = flip label "number" $ try $ syntax $ do
  sign <- (char '+' $> 1) <|> (char '-' $> -1) <|> pure 1
  digits <- many1 (satisfy isDigit)
  let magnitude = foldl (\a d -> 10 * a + toInteger (digitToInt d)) 0 digits
  pure (IntegerExpression (sign * magnitude))
```

```
rationalExpression :: Stream s m Char => ParserT s m Expression
rationalExpression = flip label "number" $ try $ syntax $ do
  sign <- (char '+' $> 1) <|> (char '-' $> -1) <|> pure 1
  intDigits <- many1 (satisfy isDigit)
  fracDigits <- char '.' *> many1 (satisfy isDigit)
  let intPart = foldl (\a d -> 10 * a + toRational (digitToInt d)) 0 intDigits
  let fracPart = foldr (\d a -> 0.1 * (a + toRational (digitToInt d))) 0 fracDigits
  pure (RationalExpression (sign * (intPart + fracPart)))


arrayExpression :: Stream s m Char => ParserT s m Expression
arrayExpression = do
  open <- token "["

  (elems, commas) <- s *> optionMaybe expression >>= \case
    Nothing -> pure ([], [])

    Just first -> do
      rest <- many (liftA2 (,) (try (s *> token ",")) (s *> expression))
      pure (first : map snd rest, map fst rest)

  close <- s *> token "]"
  pure (ArrayExpression open elems commas close)


callOrVarExpression :: Stream s m Char => ParserT s m Expression
callOrVarExpression = do
  SymbolId{name, interval} <- symbolId

  optionMaybe (try (s *> token "(")) >>= \case
    Nothing -> pure (VarExpression name interval)

    Just open -> do
      (args, commas) <- s *> optionMaybe expression >>= \case
        Nothing -> pure ([], [])

        Just first -> do
          rest <- many (liftA2 (,) (try (s *> token ",")) (s *> expression))
          pure (first : map snd rest, map fst rest)

      close <- s *> token ")"
      pure (CallExpression (SymbolId name interval) open args commas close)


unaryExpression :: Stream s m Char => ParserT s m Expression
unaryExpression = do
  unary <- unaryOperator
  operand <- s *> expression6 False
  pure (UnaryExpression unary operand)


parenthesizedExpression :: Stream s m Char => ParserT s m Expression
parenthesizedExpression = do
  open <- token "("
  inner <- s *> expression
```

```
  close <- s *> token ")"
  pure (ParenthesizedExpression open inner close)


unaryOperator :: Stream s m Char => ParserT s m UnaryOperator
unaryOperator = syntax $ choice
  [
    text "+" $> PlusOperator,
    text "-" $> MinusOperator
  ]


binaryOperator :: Stream s m Char => ParserT s m BinaryOperator
binaryOperator = syntax $ choice
  [
    text "+=" $> AddAssignOperator,
    text "-=" $> SubtractAssignOperator,
    text "*=" $> MultiplyAssignOperator,
    text "/=" $> DivideAssignOperator,
    text "%=" $> ModuloAssignOperator,
    (try (char '=' <* notFollowedBy (char '=')) <?> "'='") $> PlainAssignOperator,
    text "*" $> MultiplyOperator,
    text "/" $> DivideOperator,
    text "%" $> ModuloOperator,
    text "+" $> AddOperator,
    text "-" $> SubtractOperator,
    text ">=" $> GreaterOrEqualOperator,
    text ">" $> GreaterOperator,
    text "<=" $> LessOrEqualOperator,
    text "<" $> LessOperator,
    text "==" $> EqualOperator,
    text "!=" $> NotEqualOperator,
    keyword "and" $> AndOperator,
    keyword "or" $> OrOperator,
    keyword "xor" $> XorOperator
  ]


symbolId :: Stream s m Char => ParserT s m SymbolId
symbolId = syntax $ do
  name <- identifier
  pure (SymbolId name)


typeId :: Stream s m Char => ParserT s m TypeId
typeId = choice
  [
    syntax $ do
      name <- identifier
      pure (PlainTypeId name),

    do
      open <- token "["
      innerTypeId <- s *> typeId
      close <- s *> token "]"
      pure (ArrayTypeId open innerTypeId close)
  ]
```

```
comment :: Stream s m Char => ParserT s m Token
comment = flip label "comment" $ do
  token <- syntax $ do
    text "//"
    skipMany (noneOf "\n\v\r\x85\x2028\x2029")
    pure Token

  modifyState (\state -> State (unState state . (token :)))
  pure token


syntax :: Stream s m Char => ParserT s m ((Int, Int) -> a) -> ParserT s m a
syntax mf = do
  start <- getOffset
  f <- mf
  end <- getOffset
  pure (f (start, end))


keyword :: Stream s m Char => String -> ParserT s m Token
keyword literal = flip label ("keyword ‘" ++ literal ++ "’") $ syntax $ do
  (name, state) <- try (lookAhead (liftA2 (,) identifier getParserState))
  guard (name == literal)
  setParserState state
  pure Token


token :: Stream s m Char => String -> ParserT s m Token
token literal = syntax $ do
  text literal
  pure Token


text :: Stream s m Char => String -> ParserT s m String
text literal = try (string literal) <?> ("‘" ++ literal ++ "’")


-- Regular expression: [\p{L}\p{Nl}\p{Pc}][\p{L}\p{Nl}\p{Pc}\p{Mn}\p{Mc}\p{Nd}]*
identifier :: Stream s m Char => ParserT s m String
identifier = flip label "identifier" $ do
  start <- satisfy (isStart . generalCategory)
  continue <- many (satisfy (isContinue . generalCategory))
  pure (start : continue)

  where
    isStart UppercaseLetter = True
    isStart LowercaseLetter = True
    isStart TitlecaseLetter = True
    isStart ModifierLetter = True
    isStart OtherLetter = True
    isStart LetterNumber = True
    isStart ConnectorPunctuation = True
    isStart _ = False

    isContinue NonSpacingMark = True
```

```
    isContinue SpacingCombiningMark = True
    isContinue DecimalNumber = True
    isContinue category = isStart category


s :: Stream s m Char => ParserT s m ()
s = skipMany (skipMany1 (space <?> "") <|> void (comment <?> ""))
```

A.9 src/Devin/Ratio.hs

```
module Devin.Ratio (
  module Data.Ratio,
  displayRatio,
  displaysRatio,
) where

import Data.Ratio


displayRatio :: (Integral a, Show a) => Ratio a -> String
displayRatio ratio = displaysRatio ratio ""


displaysRatio :: (Integral a, Show a) => Ratio a -> ShowS
displaysRatio ratio = case denominator ratio of
  0 -> case compare (numerator ratio) 0 of
    LT -> showString "-ϖ"
    EQ -> showString "NaN"
    GT -> showString "ϖ"

  _ ->
    let num = abs (numerator ratio)
        den = abs (denominator ratio)
        (d, m) = num `divMod` den

        go 0 = showChar '0'
        go m | (d', 0) <- (10 * m) `divMod` den = shows d'
        go m | (d', m') <- (10 * m) `divMod` den = shows d' . go m'

    in (if ratio < 0 then showChar '-' else id) . shows d . showChar '.' . go m
```

A.10 src/Devin/Syntax.hs

```
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE DuplicateRecordFields #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE InstanceSigs #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE NamedFieldPuns #-}

module Devin.Syntax (
  Devin (..),
  Definition (..),
  Statement (..),
```

```
  Expression (..),
  UnaryOperator (..),
  BinaryOperator (..),
  SymbolId (..),
  TypeId (..),
  Token (..)
) where

import Data.Data

import Devin.Display
import Devin.Interval
import Devin.Ratio


newtype Devin = Devin { definitions :: [Definition] }
  deriving (Eq, Show, Read, Data)


data Definition where
  VarDefinition :: {
    varKeyword :: Token,
    varId :: SymbolId,
    equalSign :: Token,
    value :: Expression,
    semicolon :: Token
  } -> Definition

  FunDefinition :: {
    defKeyword :: Token,
    funId :: SymbolId,
    open :: Token,
    params :: [(Maybe Token, SymbolId, Maybe (Token, TypeId))],
    commas :: [Token],
    close :: Token,
    returnInfo :: Maybe (Token, TypeId),
    body :: Statement
  } -> Definition

  deriving (Eq, Show, Read, Data)


data Statement where
  DefinitionStatement :: {
    definition :: Definition
  } -> Statement

  ExpressionStatement :: {
    effect :: Expression,
    semicolon :: Token
  } -> Statement

  IfStatement :: {
    ifKeyword :: Token,
    predicate :: Expression,
    trueBranch :: Statement
  } -> Statement
```

```
  IfElseStatement :: {
    ifKeyword :: Token,
    predicate :: Expression,
    trueBranch :: Statement,
    elseKeyword :: Token,
    falseBranch :: Statement
  } -> Statement

  WhileStatement :: {
    whileKeyword :: Token,
    predicate :: Expression,
    body :: Statement
  } -> Statement

  DoWhileStatement :: {
    doKeyword :: Token,
    body :: Statement,
    whileKeyword :: Token,
    predicate :: Expression,
    semicolon :: Token
  } -> Statement

  ReturnStatement :: {
    returnKeyword :: Token,
    result :: Maybe Expression,
    semicolon :: Token
  } -> Statement

  AssertStatement :: {
    assertKeyword :: Token,
    predicate :: Expression,
    semicolon :: Token
  } -> Statement

  BreakpointStatement :: {
    breakpointKeyword :: Token,
    semicolon :: Token
  } -> Statement

  BlockStatement :: {
    open :: Token,
    statements :: [Statement],
    close :: Token
  } -> Statement

  deriving (Eq, Show, Read, Data)


data Expression where
  VarExpression :: {
    varName :: String,
    interval :: (Int, Int)
  } -> Expression

  IntegerExpression :: {
    integer :: Integer,
```

```
    interval :: (Int, Int)
  } -> Expression

  RationalExpression :: {
    rational :: Rational,
    interval :: (Int, Int)
  } -> Expression

  ArrayExpression :: {
    open :: Token,
    elems :: [Expression],
    commas :: [Token],
    close :: Token
  } -> Expression

  AccessExpression :: {
    array :: Expression,
    open :: Token,
    index :: Expression,
    close :: Token
  } -> Expression

  CallExpression :: {
    funId :: SymbolId,
    open :: Token,
    args :: [Expression],
    commas :: [Token],
    close :: Token
  } -> Expression

  UnaryExpression :: {
    unary :: UnaryOperator,
    operand :: Expression
  } -> Expression

  BinaryExpression :: {
    left :: Expression,
    binary :: BinaryOperator,
    right :: Expression
  } -> Expression

  ParenthesizedExpression :: {
    open :: Token,
    inner :: Expression,
    close :: Token
  } -> Expression

  deriving (Eq, Show, Read, Data)


data UnaryOperator
  = PlusOperator { interval :: (Int, Int) }
  | MinusOperator { interval :: (Int, Int) }
  deriving (Eq, Show, Read, Data)


data BinaryOperator
```

```
  = AddOperator { interval :: (Int, Int) }
  | SubtractOperator { interval :: (Int, Int) }
  | MultiplyOperator { interval :: (Int, Int) }
  | DivideOperator { interval :: (Int, Int) }
  | ModuloOperator { interval :: (Int, Int) }
  | EqualOperator { interval :: (Int, Int) }
  | NotEqualOperator { interval :: (Int, Int) }
  | LessOperator { interval :: (Int, Int) }
  | LessOrEqualOperator { interval :: (Int, Int) }
  | GreaterOperator { interval :: (Int, Int) }
  | GreaterOrEqualOperator { interval :: (Int, Int) }
  | AndOperator { interval :: (Int, Int) }
  | OrOperator { interval :: (Int, Int) }
  | XorOperator { interval :: (Int, Int) }
  | PlainAssignOperator { interval :: (Int, Int) }
  | AddAssignOperator { interval :: (Int, Int) }
  | SubtractAssignOperator { interval :: (Int, Int) }
  | MultiplyAssignOperator { interval :: (Int, Int) }
  | DivideAssignOperator { interval :: (Int, Int) }
  | ModuloAssignOperator { interval :: (Int, Int) }
  deriving (Eq, Show, Read, Data)


data SymbolId = SymbolId { name :: String, interval :: (Int, Int) }
  deriving (Eq, Show, Read, Data)


data TypeId
  = PlainTypeId { name :: String, interval :: (Int, Int) }
  | ArrayTypeId { open :: Token, innerTypeId :: TypeId, close :: Token }
  deriving (Eq, Show, Read, Data)


newtype Token = Token { interval :: (Int, Int) }
  deriving (Eq, Show, Read, Data)


instance Interval Definition where
  start :: Num a => Definition -> a
  start VarDefinition{varKeyword} = start varKeyword
  start FunDefinition{defKeyword} = start defKeyword


  end :: Num a => Definition -> a
  end VarDefinition{semicolon} = end semicolon
  end FunDefinition{body} = end body


instance Interval Statement where
  start :: Num a => Statement -> a
  start DefinitionStatement{definition} = start definition
  start ExpressionStatement{effect} = start effect
  start IfStatement{ifKeyword} = start ifKeyword
  start IfElseStatement{ifKeyword} = start ifKeyword
  start WhileStatement{whileKeyword} = start whileKeyword
  start DoWhileStatement{doKeyword} = start doKeyword
  start ReturnStatement{returnKeyword} = start returnKeyword
```

```
  start AssertStatement{assertKeyword} = start assertKeyword
  start BreakpointStatement{breakpointKeyword} = start breakpointKeyword
  start BlockStatement{open} = start open


  end :: Num a => Statement -> a
  end DefinitionStatement{definition} = end definition
  end ExpressionStatement{semicolon} = end semicolon
  end IfStatement{trueBranch} = end trueBranch
  end IfElseStatement{falseBranch} = end falseBranch
  end WhileStatement{body} = end body
  end DoWhileStatement{semicolon} = end semicolon
  end ReturnStatement{semicolon} = end semicolon
  end AssertStatement{semicolon} = end semicolon
  end BreakpointStatement{semicolon} = end semicolon
  end BlockStatement{close} = end close


instance Interval Expression where
  start :: Num a => Expression -> a
  start VarExpression{interval} = start interval
  start IntegerExpression{interval} = start interval
  start RationalExpression{interval} = start interval
  start ArrayExpression{open} = start open
  start AccessExpression{array} = start array
  start CallExpression{funId} = start funId
  start UnaryExpression{unary} = start unary
  start BinaryExpression{left} = start left
  start ParenthesizedExpression{open} = start open


  end :: Num a => Expression -> a
  end VarExpression{interval} = end interval
  end IntegerExpression{interval} = end interval
  end RationalExpression{interval} = end interval
  end ArrayExpression{close} = end close
  end AccessExpression{close} = end close
  end CallExpression{close} = end close
  end UnaryExpression{operand} = end operand
  end BinaryExpression{right} = end right
  end ParenthesizedExpression{close} = end close


instance Interval UnaryOperator where
  start :: Num a => UnaryOperator -> a
  start PlusOperator{interval} = start interval
  start MinusOperator{interval} = start interval


  end :: Num a => UnaryOperator -> a
  end PlusOperator{interval} = end interval
  end MinusOperator{interval} = end interval


instance Interval BinaryOperator where
  start :: Num a => BinaryOperator -> a
  start AddOperator{interval} = start interval
```

```
  start SubtractOperator{interval} = start interval
  start MultiplyOperator{interval} = start interval
  start DivideOperator{interval} = start interval
  start ModuloOperator{interval} = start interval
  start EqualOperator{interval} = start interval
  start NotEqualOperator{interval} = start interval
  start LessOperator{interval} = start interval
  start LessOrEqualOperator{interval} = start interval
  start GreaterOperator{interval} = start interval
  start GreaterOrEqualOperator{interval} = start interval
  start AndOperator{interval} = start interval
  start OrOperator{interval} = start interval
  start XorOperator{interval} = start interval
  start PlainAssignOperator{interval} = start interval
  start AddAssignOperator{interval} = start interval
  start SubtractAssignOperator{interval} = start interval
  start MultiplyAssignOperator{interval} = start interval
  start DivideAssignOperator{interval} = start interval
  start ModuloAssignOperator{interval} = start interval


  end :: Num a => BinaryOperator -> a
  end AddOperator{interval} = end interval
  end SubtractOperator{interval} = end interval
  end MultiplyOperator{interval} = end interval
  end DivideOperator{interval} = end interval
  end ModuloOperator{interval} = end interval
  end EqualOperator{interval} = end interval
  end NotEqualOperator{interval} = end interval
  end LessOperator{interval} = end interval
  end LessOrEqualOperator{interval} = end interval
  end GreaterOperator{interval} = end interval
  end GreaterOrEqualOperator{interval} = end interval
  end AndOperator{interval} = end interval
  end OrOperator{interval} = end interval
  end XorOperator{interval} = end interval
  end PlainAssignOperator{interval} = end interval
  end AddAssignOperator{interval} = end interval
  end SubtractAssignOperator{interval} = end interval
  end MultiplyAssignOperator{interval} = end interval
  end DivideAssignOperator{interval} = end interval
  end ModuloAssignOperator{interval} = end interval


instance Interval SymbolId where
  start :: Num a => SymbolId -> a
  start SymbolId{interval} = start interval


  end :: Num a => SymbolId -> a
  end SymbolId{interval} = end interval


instance Interval TypeId where
  start :: Num a => TypeId -> a
  start PlainTypeId{interval} = start interval
  start ArrayTypeId{open} = start open
```

```
  end :: Num a => TypeId -> a
  end PlainTypeId{interval} = end interval
  end ArrayTypeId{close} = end close


instance Interval Token where
  start :: Num a => Token -> a
  start Token{interval} = start interval


  end :: Num a => Token -> a
  end Token{interval} = end interval


instance Display Expression where
  displays :: Expression -> ShowS
  displays = \case
    VarExpression{varName} ->
      showString varName

    IntegerExpression{integer} ->
      shows integer

    RationalExpression{rational} ->
      displaysRatio rational

    AccessExpression{array, index} ->
      displays array .
      showChar '[' .
      displays index .
      showChar ']'

    ParenthesizedExpression{inner} ->
      showChar '(' .
      displays inner .
      showChar ')'

    UnaryExpression{unary = PlusOperator{}, operand} ->
      showChar '+' .
      displays operand

    UnaryExpression{unary = MinusOperator{}, operand} ->
      showChar '-' .
      displays operand

    BinaryExpression{left, binary, right} ->
      displays left .
      showChar ' ' .
      displays binary .
      showChar ' ' .
      displays right

    ArrayExpression{elems = []} ->
      showString "[]"
```

```
    ArrayExpression{elems = elem : elems} ->
      showChar '[' .
      displays elem .
      go elems

      where
        go [] = showChar ']'
        go (elem : elems) = showString ", " . displays elem . go elems

    CallExpression{funId = SymbolId{name}, args = []} ->
      showString name .
      showString "()"

    CallExpression{funId = SymbolId{name}, args = arg : args} ->
      showString name .
      showChar '(' .
      displays arg .
      go args

      where
        go [] = showChar ')'
        go (arg : args) = showString ", " . displays arg . go args


instance Display UnaryOperator where
  displays :: UnaryOperator -> ShowS
  displays PlusOperator{} = showChar '+'
  displays MinusOperator{} = showChar '-'


instance Display BinaryOperator where
  displays :: BinaryOperator -> ShowS
  displays AddOperator{} = showChar '+'
  displays SubtractOperator{} = showChar '-'
  displays MultiplyOperator{} = showChar '*'
  displays DivideOperator{} = showChar '/'
  displays ModuloOperator{} = showChar '%'
  displays EqualOperator{} = showString "=="
  displays NotEqualOperator{} = showString "!="
  displays LessOperator{} = showChar '<'
  displays LessOrEqualOperator{} = showString "<="
  displays GreaterOperator{} = showChar '>'
  displays GreaterOrEqualOperator{} = showString ">="
  displays AndOperator{} = showString "and"
  displays OrOperator{} = showString "or"
  displays XorOperator{} = showString "xor"
  displays PlainAssignOperator{} = showChar '='
  displays AddAssignOperator{} = showString "+="
  displays SubtractAssignOperator{} = showString "-="
  displays MultiplyAssignOperator{} = showString "*="
  displays DivideAssignOperator{} = showString "/="
  displays ModuloAssignOperator{} = showString "%="
```

A.11 src/Devin/Type.hs

```
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE InstanceSigs #-}

module Devin.Type (
  Type (..),
  (<:),
  merge
) where

import Data.Data
import Data.Maybe

import Devin.Display


data Type
  = Unknown
  | Unit
  | Bool
  | Int
  | Float
  | Array Type
  | Placeholder String
  deriving (Show, Read, Data)


instance Display Type where
  displays :: Type -> ShowS
  displays Unknown = showChar '?'
  displays Unit = showString "Unit"
  displays Bool = showString "Bool"
  displays Int = showString "Int"
  displays Float = showString "Float"
  displays (Array t) = showChar '[' . displays t . showChar ']'
  displays (Placeholder name) = showString name


(<:) :: Type -> Type -> Bool
t1 <: t2 = isJust (merge t1 t2)


merge :: Type -> Type -> Maybe Type
merge Unknown _ = Just Unknown
merge _ Unknown = Just Unknown
merge Unit Unit = Just Unit
merge Bool Bool = Just Bool
merge Int Int = Just Int
merge Float Float = Just Float
merge (Array t1) (Array t2) = Array <$> merge t1 t2
merge (Placeholder n1) (Placeholder n2) | n1 == n2 = Just (Placeholder n1)
merge _ _ = Nothing
```

A.12 src/Devin/Typer.hs

```
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE InstanceSigs #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE NamedFieldPuns #-}

module Devin.Typer (
  Typer,
  Environment,
  Scope (..),
  predefinedEnv,
  runTyper,
  defineType,
  lookupType,
  defineFunSignature,
  lookupFunSignature,
  defineVarType,
  lookupVarType,
  withNewScope,
  report
) where

import Control.Applicative
import Data.Data

import Devin.Error
import Devin.Type


newtype Typer a
  = Typer { unTyper :: Environment -> (a, Environment, [Error] -> [Error]) }
  deriving Functor


type Environment = [Scope]


data Scope = Scope {
  types :: [(String, Type)],
  funs :: [(String, ([Type], Type))],
  vars :: [(String, Type)]
} deriving (Show, Read, Data)


instance Applicative Typer where
  pure :: a -> Typer a
  pure x = Typer (\env -> (x, env, id))


  liftA2 :: (a -> b -> c) -> Typer a -> Typer b -> Typer c
  liftA2 f mx my = Typer $ \env ->
    let (x, env', errorDL1) = unTyper mx env
        (y, env'', errorDL2) = unTyper my env'
     in (f x y, env'', errorDL1 . errorDL2)
```

```
instance Monad Typer where
  (>>=) :: Typer a -> (a -> Typer b) -> Typer b
  mx >>= f = Typer $ \env ->
    let (x, env', errorDL1) = unTyper mx env
        (y, env'', errorDL2) = unTyper (f x) env'
     in (y, env'', errorDL1 . errorDL2)


predefinedEnv :: Environment
predefinedEnv =
  let t1 = ("Unit", Unit)
      t2 = ("Bool", Bool)
      t3 = ("Int", Int)
      t4 = ("Float", Float)

      f1 = ("not", ([Bool], Bool))
      f2 = ("len", ([Array Unknown], Int))
      f3 = ("intToFloat", ([Int], Float))
      f4 = ("floatToInt", ([Float], Int))

      v1 = ("true", Bool)
      v2 = ("false", Bool)
      v3 = ("unit", Unit)

   in [Scope [t4, t3, t2, t1] [f4, f3, f2, f1] [v3, v2, v1]]


runTyper :: Typer a -> Environment -> (a, Environment, [Error])
runTyper mx env =
  let (x, env', errorDL) = unTyper mx env
   in (x, env', errorDL [])


defineType :: String -> Type -> Typer Type
defineType name t = Typer $ \case
  [] -> (t, [Scope [(name, t)] [] []], id)

  scope : scopes ->
    let types' = (name, t) : types scope
     in (t, scope{types = types'} : scopes, id)


lookupType :: String -> Typer (Maybe (Type, Int))
lookupType name = Typer (\env -> (go 0 env, env, id))
  where
    go _ [] = Nothing

    go depth (Scope{types} : scopes) = case lookup name types of
      Just t -> Just (t, depth)
      Nothing -> go (depth + 1) scopes


defineFunSignature :: String -> ([Type], Type) -> Typer ()
defineFunSignature name signature = Typer $ \case
  [] -> ((), [Scope [] [(name, signature)] []], id)
```

```
  scope : scopes ->
    let funs' = (name, signature) : funs scope
     in ((), scope{funs = funs'} : scopes, id)


lookupFunSignature :: String -> Typer (Maybe (([Type], Type), Int))
lookupFunSignature name = Typer (\env -> (go 0 env, env, id))
  where
    go _ [] = Nothing

    go depth (Scope{funs} : scopes) = case lookup name funs of
      Just signature -> Just (signature, depth)
      Nothing -> go (depth + 1) scopes


defineVarType :: String -> Type -> Typer ()
defineVarType name t = Typer $ \case
  [] -> ((), [Scope [] [] [(name, t)]], id)

  scope : scopes ->
    let vars' = (name, t) : vars scope
     in ((), scope{vars = vars'} : scopes, id)


lookupVarType :: String -> Typer (Maybe (Type, Int))
lookupVarType name = Typer (\env -> (go 0 env, env, id))
  where
    go _ [] = Nothing

    go depth (Scope{vars} : scopes) = case lookup name vars of
      Just t -> Just (t, depth)
      Nothing -> go (depth + 1) scopes


withNewScope :: Typer a -> Typer a
withNewScope mx = Typer $ \env ->
  let (x, env', errors) = unTyper mx (Scope [] [] [] : env)
   in (x, tail env', errors)


report :: Error -> Typer ()
report error = Typer (\env -> ((), env, (error :)))
```

A.13 src/Devin/Typers.hs

```
{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE DisambiguateRecordFields #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE NamedFieldPuns #-}

module Devin.Typers (
  checkDevin,
  checkDefinition,
  checkStatement,
  checkExpression
) where
```

```
import Control.Monad
import Data.Foldable
import Data.Traversable

import Control.Monad.Extra

import Devin.Error
import Devin.Syntax
import Devin.Type
import Devin.Typer


checkDevin :: Devin -> Typer ()
checkDevin Devin{definitions} = do
  for_ definitions checkDefinition1
  for_ definitions checkDefinition2


checkDefinition :: Definition -> Typer ()
checkDefinition definition = do
  checkDefinition1 definition
  checkDefinition2 definition


checkDefinition1 :: Definition -> Typer ()
checkDefinition1 = \case
  VarDefinition{} -> pure ()

  FunDefinition{funId = SymbolId{name}, params, returnInfo} -> do
    paramTs <- for params $ \(_, _, typeInfo) -> case typeInfo of
      Just (_, paramTypeId) -> getType paramTypeId
      Nothing -> pure Unknown

    returnT <- case returnInfo of
      Just (_, returnTypeId) -> getType returnTypeId
      Nothing -> pure Unknown

    defineFunSignature name (paramTs, returnT)


checkDefinition2 :: Definition -> Typer ()
checkDefinition2 = \case
  VarDefinition{varId = SymbolId{name}, value} -> do
    t <- checkExpression value
    defineVarType name t

  FunDefinition{funId, params, returnInfo, body} -> withNewScope $ do
    for_ params $ \(_, SymbolId{name}, typeInfo) -> case typeInfo of
      Just (_, paramTypeId) -> do
        paramT <- getType paramTypeId
        defineVarType name paramT

      Nothing -> defineVarType name Unknown

    returnT <- case returnInfo of
      Just (_, returnTypeId) -> getType returnTypeId
```

```
      Nothing -> pure Unknown

    case returnT of
      Unknown -> void (checkStatement Unknown body)
      Unit -> void (checkStatement Unit body)

      _ -> do
        doesReturn <- checkStatement returnT body
        unless doesReturn (report (MissingReturnStatement funId))


checkStatement :: Type -> Statement -> Typer Bool
checkStatement expectedT statement = case statement of
  DefinitionStatement{definition} -> do
    checkDefinition definition
    pure False

  ExpressionStatement{effect} -> do
    checkExpression effect
    pure False

  IfStatement{predicate, trueBranch} -> do
    t <- checkExpression predicate
    unless (t <: Bool) (report (InvalidType predicate Bool t))
    withNewScope (checkStatement expectedT trueBranch)
    pure False

  IfElseStatement{predicate, trueBranch, falseBranch} -> do
    t <- checkExpression predicate
    unless (t <: Bool) (report (InvalidType predicate Bool t))
    trueBranchDoesReturn <- withNewScope (checkStatement expectedT trueBranch)
    falseBranchDoesReturn <- withNewScope (checkStatement expectedT falseBranch)
    pure (trueBranchDoesReturn && falseBranchDoesReturn)

  WhileStatement{predicate, body} -> do
    t <- checkExpression predicate
    unless (t <: Bool) (report (InvalidType predicate Bool t))
    withNewScope (checkStatement expectedT body)
    pure False

  DoWhileStatement{body, predicate} -> do
    doesReturn <- withNewScope (checkStatement expectedT body)
    t <- checkExpression predicate
    unless (t <: Bool) (report (InvalidType predicate Bool t))
    pure doesReturn

  ReturnStatement{result = Just result} -> do
    t <- checkExpression result
    unless (t <: expectedT) (report (InvalidType result expectedT t))
    pure True

  ReturnStatement{result = Nothing} -> do
    unless (Unit <: expectedT) (report (MissingReturnValue statement expectedT))
    pure True

  AssertStatement{predicate} -> do
    t <- checkExpression predicate
```

```
    unless (t <: Bool) (report (InvalidType predicate Bool t))
    pure False

  BreakpointStatement{} -> pure False

  BlockStatement{statements} -> withNewScope $ do
    for_ statements $ \case
      DefinitionStatement{definition} -> checkDefinition1 definition
      _ -> pure ()

    foldlM f False statements

    where
      f doesReturn DefinitionStatement{definition} = do
        checkDefinition2 definition
        pure doesReturn

      f doesReturn statement = do
        doesReturn' <- checkStatement expectedT statement
        pure (doesReturn || doesReturn')


checkExpression :: Expression -> Typer Type
checkExpression expression = case expression of
  IntegerExpression{} -> pure Int

  RationalExpression{} -> pure Float

  VarExpression{varName, interval} -> lookupVarType varName >>= \case
    Just (t, _) -> pure t
    Nothing -> report' (UnknownVar varName interval)

  ArrayExpression{elems = []} -> pure (Array Unknown)

  ArrayExpression{elems = elem : elems} -> do
    t <- checkExpression elem

    flip loopM elems $ \case
      [] -> pure (Right (Array t))

      elem : elems -> checkExpression elem >>= \case
        Unknown -> do
          for_ elems checkExpression
          pure (Right (Array Unknown))

        t' | t' <: t -> pure (Left elems)

        t' -> do
          report (InvalidType elem t t')
          pure (Left elems)

  AccessExpression{array, index} -> do
    arrayT <- checkExpression array
    indexT <- checkExpression index

    case (arrayT, indexT) of
      (Unknown, Unknown) -> pure Unknown
```

```
    (Unknown, Int) -> pure Unknown
    (Unknown, _) -> report' (InvalidType index Int indexT)
    (Array t, Unknown) -> pure t
    (Array t, Int) -> pure t
    (Array _, _) -> report' (InvalidType index Int indexT)
    (_, _) -> report' (InvalidType array (Array Unknown) arrayT)

CallExpression{funId = SymbolId{name, interval}, args} ->
  lookupFunSignature name >>= \case
    Nothing -> report' (UnknownFun name interval)

    Just ((paramTs, returnT), _) -> go 0 args paramTs
      where
        go _ [] [] = pure returnT

        go n (arg : args) (paramT : paramTs) = do
          argT <- checkExpression arg
          unless (argT <: paramT) (report (InvalidType arg paramT argT))
          go (n + 1) args paramTs

        go n args paramTs = do
          let expected = n + length paramTs
          let actual = n + length args
          report' (InvalidArgCount expression expected actual)

UnaryExpression{unary, operand} | PlusOperator{} <- unary -> do
  operandT <- checkExpression operand

  case operandT of
    Unknown -> pure Unknown
    Int -> pure Int
    Float -> pure Float
    _ -> report' (InvalidUnary unary operandT)

UnaryExpression{unary, operand} | MinusOperator{} <- unary -> do
  operandT <- checkExpression operand

  case operandT of
    Unknown -> pure Unknown
    Int -> pure Int
    Float -> pure Float
    _ -> report' (InvalidUnary unary operandT)

BinaryExpression{left, binary, right} | AddOperator{} <- binary -> do
  leftT <- checkExpression left
  rightT <- checkExpression right

  case (leftT, rightT) of
    (Unknown, _) -> pure Unknown
    (_, Unknown) -> pure Unknown
    (Int, Int) -> pure Int
    (Float, Float) -> pure Float
    (Array t1, Array t2) | Just t <- merge t1 t2 -> pure (Array t)
    (_, _) -> report' (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | SubtractOperator{} <- binary -> do
  leftT <- checkExpression left
```

```
    rightT <- checkExpression right

    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
      (Int, Int) -> pure Int
      (Float, Float) -> pure Float
      (_, _) -> report' (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary, right} | MultiplyOperator{} <- binary -> do
    leftT <- checkExpression left
    rightT <- checkExpression right

    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
      (Int, Int) -> pure Int
      (Float, Float) -> pure Float
      (Array t, Int) -> pure (Array t)
      (Int, Array t) -> pure (Array t)
      (_, _) -> report' (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary, right} | DivideOperator{} <- binary -> do
    leftT <- checkExpression left
    rightT <- checkExpression right

    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
      (Int, Int) -> pure Int
      (Float, Float) -> pure Float
      (_, _) -> report' (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary, right} | ModuloOperator{} <- binary -> do
    leftT <- checkExpression left
    rightT <- checkExpression right

    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
      (Int, Int) -> pure Int
      (_, _) -> report' (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary, right} | EqualOperator{} <- binary -> do
    leftT <- checkExpression left
    rightT <- checkExpression right

    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
      (_, _) | Just _ <- merge leftT rightT -> pure Bool
      (_, _) -> report' (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary, right} | NotEqualOperator{} <- binary -> do
    leftT <- checkExpression left
    rightT <- checkExpression right
```

```
  case (leftT, rightT) of
    (Unknown, _) -> pure Unknown
    (_, Unknown) -> pure Unknown
    (_, _) | Just _ <- merge leftT rightT -> pure Bool
    (_, _) -> report' (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | LessOperator{} <- binary -> do
  leftT <- checkExpression left
  rightT <- checkExpression right

  case (leftT, rightT) of
    (Unknown, _) -> pure Unknown
    (_, Unknown) -> pure Unknown
    (_, _) | Just _ <- merge leftT rightT -> pure Bool
    (_, _) -> report' (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | LessOrEqualOperator{} <- binary -> do
  leftT <- checkExpression left
  rightT <- checkExpression right

  case (leftT, rightT) of
    (Unknown, _) -> pure Unknown
    (_, Unknown) -> pure Unknown
    (_, _) | Just _ <- merge leftT rightT -> pure Bool
    (_, _) -> report' (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | GreaterOperator{} <- binary -> do
  leftT <- checkExpression left
  rightT <- checkExpression right

  case (leftT, rightT) of
    (Unknown, _) -> pure Unknown
    (_, Unknown) -> pure Unknown
    (_, _) | Just _ <- merge leftT rightT -> pure Bool
    (_, _) -> report' (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | GreaterOrEqualOperator{} <- binary -> do
  leftT <- checkExpression left
  rightT <- checkExpression right

  case (leftT, rightT) of
    (Unknown, _) -> pure Unknown
    (_, Unknown) -> pure Unknown
    (_, _) | Just _ <- merge leftT rightT -> pure Bool
    (_, _) -> report' (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | AndOperator{} <- binary -> do
  leftT <- checkExpression left
  rightT <- checkExpression right

  case (leftT, rightT) of
    (Unknown, _) -> pure Unknown
    (_, Unknown) -> pure Unknown
    (Bool, Bool) -> pure Bool
    (_, _) -> report' (InvalidBinary binary leftT rightT)

BinaryExpression{left, binary, right} | OrOperator{} <- binary -> do
```

```
    leftT <- checkExpression left
    rightT <- checkExpression right

    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
      (Bool, Bool) -> pure Bool
      (_, _) -> report' (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary, right} | XorOperator{} <- binary -> do
    leftT <- checkExpression left
    rightT <- checkExpression right

    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
      (Bool, Bool) -> pure Bool
      (_, _) -> report' (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary, right} | PlainAssignOperator{} <- binary -> do
    leftT <- checkExpression left
    rightT <- checkExpression right

    if rightT <: leftT then
      pure rightT
    else
      report' (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary, right} | AddAssignOperator{} <- binary -> do
    leftT <- checkExpression left
    rightT <- checkExpression right

    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
      (Int, Int) -> pure Int
      (Float, Float) -> pure Float
      (_, _) -> report' (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary, right} | SubtractAssignOperator{} <- binary -> do
    leftT <- checkExpression left
    rightT <- checkExpression right

    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
      (Int, Int) -> pure Int
      (Float, Float) -> pure Float
      (_, _) -> report' (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary, right} | MultiplyAssignOperator{} <- binary -> do
    leftT <- checkExpression left
    rightT <- checkExpression right

    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
```

```
      (Int, Int) -> pure Int
      (Float, Float) -> pure Float
      (Array t, Int) -> pure (Array t)
      (_, _) -> report' (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary, right} | DivideAssignOperator{} <- binary -> do
    leftT <- checkExpression left
    rightT <- checkExpression right

    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
      (Int, Int) -> pure Int
      (Float, Float) -> pure Float
      (_, _) -> report' (InvalidBinary binary leftT rightT)

  BinaryExpression{left, binary, right} | ModuloAssignOperator{} <- binary -> do
    leftT <- checkExpression left
    rightT <- checkExpression right

    case (leftT, rightT) of
      (Unknown, _) -> pure Unknown
      (_, Unknown) -> pure Unknown
      (Int, Int) -> pure Int
      (_, _) -> report' (InvalidBinary binary leftT rightT)

  ParenthesizedExpression{inner} -> checkExpression inner


getType :: TypeId -> Typer Type
getType = \case
  PlainTypeId{name, interval} -> lookupType name >>= \case
    Just (t, _) -> pure t

    Nothing -> do
      report (UnknownType name interval)
      defineType name (Placeholder name)

  ArrayTypeId{innerTypeId} -> do
    t <- getType innerTypeId
    pure (Array t)


report' :: Error -> Typer Type
report' error = do
  report error
  pure Unknown
```

A.14 test/Main.hs

```
{-# OPTIONS_GHC -F -pgmF hspec-discover #-}
```

A.15 test/Devin/EvaluatorsSpec.hs

```
{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE LambdaCase #-}

module Devin.EvaluatorsSpec (spec) where

import Test.Hspec

import Devin.Display
import Devin.Evaluator
import Devin.Evaluators
import Devin.Parsers


spec :: Spec
spec = do
  describe "evalDevin" $ do
    it "should succeed on program 1" $
      executionShouldSucceed
        "def main() {\n\
        \    var x = 1;\n\
        \    var y = 2;\n\
        \    var z = 2 * y + x;\n\
        \    assert z == 5;\n\
        \}"

    it "should succeed on program 2" $
      executionShouldSucceed
        "def main() {\n\
        \    var array1 = [4, -2, 1, 0];\n\
        \    var array2 = array1;\n\
        \    array1[1] = 7;\n\
        \    assert array1 == [4, 7, 1, 0];\n\
        \    assert array2 == [4, -2, 1, 0];\n\
        \}"

    it "should succeed on program 3" $
      executionShouldSucceed
        "def main() {\n\
        \    var array = [1, 2];\n\
        \    assert array * 5 == [1, 2, 1, 2, 1, 2, 1, 2, 1, 2];\n\
        \    assert array * 0 == [];\n\
        \    assert array * -2 == [];\n\
        \}"

    it "should succeed on program 4" $
      executionShouldSucceed
        "def main()\n\
        \    assert sum(34, 35) == 69;\n\
        \\n\
        \def sum(a, b)\n\
        \    return a + b;"

    it "should succeed on program 5" $
      executionShouldSucceed
        "def main()\n\
```

```
    \    assert factorial(6) == 720;\n\
    \\n\
    \def factorial(n) {\n\
    \    assert n >= 0;\n\
    \\n\
    \    if n == 0\n\
    \        return 1;\n\
    \\n\
    \    return n * factorial(n - 1);\n\
    \}"

it "should succeed on program 6" $
  executionShouldSucceed
    "def main()\n\
    \    assert factorial(6) == 720;\n\
    \\n\
    \def factorial(n) {\n\
    \    assert n >= 0;\n\
    \    var result = 1;\n\
    \\n\
    \    while n > 1 {\n\
    \        result *= n;\n\
    \        n -= 1;\n\
    \    }\n\
    \\n\
    \    return result;\n\
    \}"

it "should succeed on program 7" $
  executionShouldSucceed
    "def main()\n\
    \    assert factorial(6) == 720;\n\
    \\n\
    \def factorial(n) {\n\
    \    if n == 0\n\
    \        return 1;\n\
    \\n\
    \    assert n > 0;\n\
    \    var result = 1;\n\
    \\n\
    \    do {\n\
    \        result *= n;\n\
    \        n -= 1;\n\
    \    } while n > 1;\n\
    \\n\
    \    return result;\n\
    \}"

it "should succeed on program 8" $
  executionShouldSucceed
    "def main() {\n\
    \    var array = [9, 7, 2, 5];\n\
    \    update(array, 1, -42);\n\
    \    assert array == [9, -42, 2, 5];\n\
    \}\n\
    \\n\
    \def update(ref array, index, value)\n\
```

```
    \    array[index] = value;"

  it "should succeed on program 9" $
    executionShouldSucceed
      "def main() {\n\
      \    var array = [9, 7, 2, 5];\n\
      \    noupdate(array, 1, -42);\n\
      \    assert array == [9, 7, 2, 5];\n\
      \}\n\
      \\n\
      \def noupdate(array, index, value)\n\
      \    array[index] = value;"

  it "should succeed on program 10" $
    executionShouldSucceed
      "def main() {\n\
      \    assert isOdd(69);\n\
      \    assert isEven(420);\n\
      \}\n\
      \\n\
      \def isEven(n) {\n\
      \    assert n >= 0;\n\
      \\n\
      \    if n == 0\n\
      \        return true;\n\
      \    else\n\
      \        return isOdd(n - 1);\n\
      \}\n\
      \\n\
      \def isOdd(n) {\n\
      \    assert n >= 0;\n\
      \\n\
      \    if n == 0\n\
      \        return false;\n\
      \    else\n\
      \        return isEven(n - 1);\n\
      \}"

  it "should succeed on program 11" $
    executionShouldSucceed
      "def main() {\n\
      \    var c = -1;\n\
      \\n\
      \    def count() {\n\
      \        c += 1;\n\
      \        return c;\n\
      \    }\n\
      \\n\
      \    assert count() == 0;\n\
      \    assert count() == 1;\n\
      \    assert count() == 2;\n\
      \    assert count() == 3;\n\
      \}"

  it "should succeed on program 12" $
    executionShouldSucceed
      "def main() -> Unit {\n\
```

```
        \    var array = [9, 2, 1, 21, -2, 4];\n\
        \    bubbleSort(array);\n\
        \    assert array == [-2, 1, 2, 4, 9, 21];\n\
        \}\n\
        \\n\
        \def bubbleSort(ref array: [Int]) -> Unit {\n\
        \    var i = 0;\n\
        \\n\
        \    while i < len(array) {\n\
        \        var j = i + 1;\n\
        \\n\
        \        while j < len(array) {\n\
        \            if array[i] > array[j] {\n\
        \                var t = array[i];\n\
        \                array[i] = array[j];\n\
        \                array[j] = t;\n\
        \            }\n\
        \\n\
        \            j += 1;\n\
        \        }\n\
        \\n\
        \        i += 1;\n\
        \    }\n\
        \}"

    it "should succeed on program 13" $
      executionShouldSucceed
        "var a = -1;\n\
        \var b = -2;\n\
        \\n\
        \def f(a: Int) -> Unit {\n\
        \    assert b == -2;\n\
        \\n\
        \    if (a < 100)\n\
        \        g(a + 1);\n\
        \}\n\
        \\n\
        \def g(b: Int) -> Unit {\n\
        \    assert a == -1;\n\
        \\n\
        \    if (b < 100)\n\
        \        f(b + 1);\n\
        \}\n\
        \\n\
        \def main() -> Unit {\n\
        \    f(0);\n\
        \    g(0);\n\
        \}"


executionShouldSucceed :: String -> Expectation
executionShouldSucceed source = case parse devin "" (0, source) of
  Left parseError -> expectationFailure (show parseError)

  Right (syntaxTree, _) -> do
    state <- makePredefinedState
```

```
    runEvaluator (evalDevin syntaxTree) state >>= \case
      (Left error, _) -> expectationFailure (display error)
      (Right _, _) -> pure ()
```

A.16 test/Devin/ParsersSpec.hs

```
{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE DisambiguateRecordFields #-}
{-# LANGUAGE NegativeLiterals #-}

module Devin.ParsersSpec (spec) where

import Test.Hspec

import Devin.Parsers
import Devin.Syntax


spec :: Spec
spec = do
  describe "expression" $ do
    it "should accept fragment 1" $
      shouldParse expression "1 += 2 " BinaryExpression{
        left = IntegerExpression 1 (0, 1),
        binary = AddAssignOperator (2, 4),
        right = IntegerExpression 2 (5, 6)
      }

    it "should accept fragment 2" $
      shouldParse expression "1+ 2*3 -(1) " BinaryExpression{
        left = BinaryExpression{
          left = IntegerExpression 1 (0, 1),

          binary = AddOperator (1, 2),

          right = BinaryExpression{
            left = IntegerExpression 2 (3, 4),
            binary = MultiplyOperator (4, 5),
            right = IntegerExpression 3 (5, 6)
          }
        },

        binary = SubtractOperator (7, 8),

        right = ParenthesizedExpression{
          open = Token (8, 9),
          inner = IntegerExpression 1 (9, 10),
          close = Token (10, 11)
        }
      }

    it "should accept fragment 3" $
      shouldParse expression "a+ -2.1= f (x, y ,z ) *=88 " BinaryExpression{
        left = VarExpression "a" (0, 1),

        binary = AddOperator (1, 2),
```

```
        right = BinaryExpression{
          left = RationalExpression -2.1 (3, 7),

          binary = PlainAssignOperator (7, 8),

          right = BinaryExpression{
            left = CallExpression{
              funId = SymbolId "f" (9, 10),

              open = Token (11, 12),

              args = [
                VarExpression "x" (12, 13),
                VarExpression "y" (15, 16),
                VarExpression "z" (18, 19)
              ],

              commas = [
                Token (13, 14),
                Token (17, 18)
              ],

              close = Token (20, 21)
            },

            binary = MultiplyAssignOperator (22, 24),

            right = IntegerExpression 88 (24, 26)
          }
        }
      }

  describe "binaryOperator" $
    it "should accept fragment 4" $
      shouldParse binaryOperator "+ " (AddOperator (0, 1))


shouldParse :: (Eq a, Show a) => Parser String a -> String -> a -> Expectation
shouldParse parser source x = case parse parser "" (0, source) of
  Left parseError -> expectationFailure (show parseError)
  Right (x', _) -> x' `shouldBe` x
```

A.17 test/Devin/TypersSpec.hs

```
{-# LANGUAGE ApplicativeDo #-}

module Devin.TypersSpec (spec) where

import Data.List

import Test.Hspec

import Devin.Display
import Devin.Parsers
import Devin.Typer
```

```
import Devin.Typers


spec :: Spec
spec = do
  describe "checkDevin" $ do
    it "should fail on program 1" $
      typeCheckingShouldFail
        "def sum(list: [Int]) -> Int {}"

    it "should succeed on program 2" $
      typeCheckingShouldSucceed
        "def sum(list: [Int]) -> Int\n\
        \    return 0;"

    it "should succeed on program 3" $
      typeCheckingShouldSucceed
        "def main()\n\
        \    var x = sum(1, 2.0);\n\
        \\n\
        \def sum(a: Int, b) -> Int\n\
        \     return 0;"

    it "should fail on program 4" $
      typeCheckingShouldFail
        "def main()\n\
        \    var x = sum(1, 2.0);\n\
        \\n\
        \def sum(a: Int, b: Int) -> Int\n\
        \     return 0;"

    it "should succeed on program 5" $
      typeCheckingShouldSucceed
        "def main()\n\
        \    var x = sum(1, 2);\n\
        \\n\
        \def sum(a: Int, b: Int) -> Int\n\
        \     return 0;"

    it "should fail on program 6" $
      typeCheckingShouldFail
        "def main()\n\
        \    var x = sum([1, false, 3]);\n\
        \\n\
        \def sum(list: [Int]) -> Int\n\
        \    return 0;\n\
        \"

    it "should succeed on program 7" $
      typeCheckingShouldSucceed
        "def main() {\n\
        \    def f(x) return x;\n\
        \    var x = sum([f(1), false, 3]);\n\
        \}\n\
        \\n\
        \def sum(list: [Int]) -> Int\n\
        \    return 0;\n\
```

```
        \"


typeCheckingShouldSucceed :: String -> Expectation
typeCheckingShouldSucceed source = case parse devin "" (0, source) of
  Left parseError -> expectationFailure (show parseError)

  Right (syntaxTree, _) -> case runTyper (checkDevin syntaxTree) predefinedEnv of
    (_, _, []) -> pure ()
    (_, _, errors) -> expectationFailure (intercalate "\n" (map display errors))


typeCheckingShouldFail :: String -> Expectation
typeCheckingShouldFail source = case parse devin "" (0, source) of
  Left parseError -> expectationFailure (show parseError)

  Right (syntaxTree, _) -> case runTyper (checkDevin syntaxTree) predefinedEnv of
    (_, _, []) -> expectationFailure "Expected type checking to fail, but it succeeded"
    (_, _, _) -> pure ()
```

A.18 app/Main.hs

```
{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE ImplicitParams #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE MonoLocalBinds #-}
{-# LANGUAGE NegativeLiterals #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE TypeApplications #-}

module Main (main) where

import Prelude hiding (getLine)
import Control.Concurrent
import Control.Exception
import Control.Monad
import Control.Monad.IO.Class
import Data.Foldable
import Data.Int
import Data.IORef
import Data.Maybe
import Data.String
import System.Exit

import Data.Tree

import qualified Data.Text as Text
import qualified Data.Text.IO as Text

import Text.Parsec.Error

import Control.Monad.Extra

import Data.GI.Base
import Data.GI.Base.GObject
import qualified GI.GObject as G
```

```
import qualified GI.GLib as G
import qualified GI.Gio as G
import qualified GI.Gdk as Gdk
import qualified GI.Gtk as Gtk
import qualified GI.GtkSource as GtkSource
import Data.GI.Gtk.ModelView.CellLayout
import Data.GI.Gtk.ModelView.ForestStore
import Data.GI.Gtk.ModelView.SeqStore
import Data.GI.Gtk.ModelView.Types
import Data.GI.Gtk.Threading

import Devin.Display
import Devin.Evaluator
import Devin.Evaluators
import Devin.Interval
import Devin.Parsec hiding (Error, try)
import Devin.Parsers
import Devin.Syntax
import Devin.Typer
import Devin.Typers

import Devin.Debug.Evaluator
import Devin.Debug.Syntax
import Devin.Highlight
import Devin.Highlight.Brackets
import Devin.Highlight.Syntax
import Devin.Levenshtein


main :: IO ()
main = Gtk.applicationNew Nothing [G.ApplicationFlagsDefaultFlags] >>= \case
  Nothing -> exitFailure

  Just application -> do
    G.onApplicationActivate application onActivate
    status <- G.applicationRun application Nothing
    when (status /= 0) (exitWith (ExitFailure (fromIntegral status)))


onActivate :: (Gtk.IsApplication a, ?self :: a) => G.ApplicationActivateCallback
onActivate = do
  userStateDirName <- G.getUserStateDir
  stateDirName <- G.buildFilenamev [userStateDirName, "devin"]
  keyFileName <- G.buildFilenamev [stateDirName, "ui.ini"]
  codeFileName <- G.buildFilenamev [stateDirName, "main.devin"]

  -- Add a fallback .monospace class for systems that lack it, such as KDE:

  cssProvider <- Gtk.cssProviderNew
  Gtk.cssProviderLoadFromData cssProvider ".monospace { font-family: monospace; }"

  displayManager <- Gdk.displayManagerGet
  displays <- Gdk.displayManagerListDisplays displayManager

  for_ displays $ \display -> do
    screen <- Gdk.displayGetDefaultScreen display
    let priority = fromIntegral Gtk.STYLE_PROVIDER_PRIORITY_FALLBACK
```

```
  Gtk.styleContextAddProviderForScreen screen cssProvider priority

-- Build the UI:

let iconName = Just "media-playback-stop-symbolic"
let iconSize = fromIntegral (fromEnum Gtk.IconSizeButton)
stopButton <- Gtk.buttonNewFromIconName iconName iconSize
Gtk.widgetSetSensitive stopButton False

let iconName = Just "media-playback-start-symbolic"
let iconSize = fromIntegral (fromEnum Gtk.IconSizeButton)
playButton <- Gtk.buttonNewFromIconName iconName iconSize
Gtk.widgetSetSensitive playButton False

codeBuffer <- GtkSource.bufferNew (Nothing @Gtk.TextTagTable)
GtkSource.bufferSetHighlightSyntax codeBuffer False
GtkSource.bufferSetHighlightMatchingBrackets codeBuffer False

codeView <- GtkSource.viewNewWithBuffer codeBuffer
GtkSource.viewSetShowLineNumbers codeView True
GtkSource.viewSetAutoIndent codeView True
GtkSource.viewSetTabWidth codeView 4
Gtk.textViewSetMonospace codeView True

syntaxTreeModel <- forestStoreNew []
syntaxTreeView <- Gtk.treeViewNewWithModel syntaxTreeModel
Gtk.treeViewSetHeadersVisible syntaxTreeView False
Gtk.treeViewSetEnableSearch syntaxTreeView False
Gtk.treeViewSetGridLines syntaxTreeView Gtk.TreeViewGridLinesVertical
addMonospaceClass syntaxTreeView

stateModel <- forestStoreNew []
stateView <- Gtk.treeViewNewWithModel stateModel
Gtk.treeViewSetHeadersVisible stateView False
Gtk.treeViewSetEnableSearch stateView False
Gtk.treeViewSetGridLines stateView Gtk.TreeViewGridLinesVertical
addMonospaceClass stateView

logModel <- seqStoreNew []
logView <- Gtk.treeViewNewWithModel logModel
Gtk.treeViewSetHeadersVisible logView False
Gtk.treeViewSetEnableSearch logView False
addMonospaceClass logView

blankView <- Gtk.treeViewNew
Gtk.treeViewSetHeadersVisible blankView False
Gtk.treeViewSetEnableSearch blankView False
Gtk.treeViewSetGridLines blankView Gtk.TreeViewGridLinesVertical
addMonospaceClass blankView

let adjustment = Nothing @Gtk.Adjustment
codeScrolledWindow <- Gtk.scrolledWindowNew adjustment adjustment
Gtk.containerAdd codeScrolledWindow codeView

let adjustment = Nothing @Gtk.Adjustment
rightScrolledWindow <- Gtk.scrolledWindowNew adjustment adjustment
Gtk.containerAdd rightScrolledWindow blankView
```

```
let adjustment = Nothing @Gtk.Adjustment
logScrolledWindow <- Gtk.scrolledWindowNew adjustment adjustment
Gtk.containerAdd logScrolledWindow logView

horizontalPaned <- Gtk.panedNew Gtk.OrientationHorizontal
Gtk.panedPack1 horizontalPaned codeScrolledWindow True False
Gtk.panedPack2 horizontalPaned rightScrolledWindow True False

verticalPaned <- Gtk.panedNew Gtk.OrientationVertical
Gtk.panedPack1 verticalPaned horizontalPaned True False
Gtk.panedPack2 verticalPaned logScrolledWindow False False

headerBar <- Gtk.headerBarNew
Gtk.headerBarSetTitle headerBar (Just "Devin")
Gtk.headerBarSetShowCloseButton headerBar True
Gtk.containerAdd headerBar stopButton
Gtk.containerAdd headerBar playButton

window <- Gtk.applicationWindowNew ?self
Gtk.windowSetDefaultSize window 1280 720
Gtk.windowSetTitlebar window (Just headerBar)
Gtk.containerAdd window verticalPaned

-- Set up the columns for syntaxTreeView, stateView, logView:

appendColumnWithDataFunction syntaxTreeView syntaxTreeModel $ \renderer row ->
  Gtk.setCellRendererTextText renderer (fst row)

appendColumnWithDataFunction syntaxTreeView syntaxTreeModel $ \renderer row ->
  Gtk.setCellRendererTextText renderer (snd row)

appendColumnWithDataFunction stateView stateModel $ \renderer row ->
  Gtk.setCellRendererTextText renderer (fst row)

appendColumnWithDataFunction stateView stateModel $ \renderer row ->
  Gtk.setCellRendererTextText renderer (snd row)

appendColumnWithDataFunction logView logModel $ \renderer row ->
  Gtk.setCellRendererTextText renderer (fst row)

appendColumnWithDataFunction logView logModel $ \renderer row ->
  Gtk.setCellRendererTextText renderer (snd row)

-- Set up the the tag table. This is needed for syntax highlighting.

styleScheme <- GtkSource.bufferGetStyleScheme codeBuffer
tags <- getHighlightingTags styleScheme
tagTable <- Gtk.textBufferGetTagTable codeBuffer
addHighlightingTags tagTable tags

-- Load the last UI state from $XDG_STATE_HOME/devin/ui.ini:

keyFile <- G.keyFileNew

try @GError $ do
  G.keyFileLoadFromFile keyFile keyFileName [G.KeyFileFlagsKeepComments]
```

```
  try @GError (G.keyFileGetIntegerList keyFile "window" "size") >>= \case
    Right [width, height] -> Gtk.windowSetDefaultSize window width height
    _ -> pure ()

  try @GError $ do
    value <- G.keyFileGetValue keyFile "window" "maximized"
    let maximized = Text.stripEnd value `elem` ["true", "1"]
    when maximized (Gtk.windowMaximize window)

-- Connect window callbacks for "window-state-event" and "configure-event"
-- signals. These keep track of the UI state which needs to be saved.

Gtk.onWidgetWindowStateEvent window $ \event -> do
  state <- Gdk.getEventWindowStateNewWindowState event
  let maximized = Gdk.WindowStateMaximized `elem` state
  G.keyFileSetBoolean keyFile "window" "maximized" maximized
  pure Gdk.EVENT_PROPAGATE

Gtk.onWidgetConfigureEvent window $ const $ do
  try @GError $ do
    value <- G.keyFileGetValue keyFile "window" "maximized"
    let maximized = Text.stripEnd value `elem` ["true", "1"]

    unless maximized $ do
      (width, height) <- Gtk.windowGetSize ?self
      G.keyFileSetIntegerList keyFile "window" "size" [width, height]

  pure Gdk.EVENT_PROPAGATE

-- Connect codeBuffer callbacks for "changed" and "notify::cursor-position"
-- signals. The former callback signals that parsing, type checking, and
-- syntax highlighting need to be performed again; the latter takes care of
-- highlighting matching braces.

syntaxTreeRef <- newIORef Nothing
parseAndTypeCheckCond <- newEmptyMVar

Gtk.onTextBufferChanged codeBuffer $ void $ do
  Gtk.widgetSetSensitive playButton False
  tryPutMVar parseAndTypeCheckCond ()

G.onObjectNotify codeBuffer (Just "cursor-position") $ const $ do
  (startIter, endIter) <- Gtk.textBufferGetBounds ?self
  Gtk.textBufferRemoveTag ?self (bracketTag tags) startIter endIter

  whenJustM (readIORef syntaxTreeRef) $ \syntaxTree -> void $ do
    insertMark <- Gtk.textBufferGetInsert ?self
    insertIter <- Gtk.textBufferGetIterAtMark ?self insertMark
    highlightDevinBrackets ?self tags insertIter syntaxTree

-- Connect playButton and stopButton callbacks for "clicked" signals.
--
-- The play button exhibits different behavior depending on context:
-- initially, its function is to start the debugging process; pressing it
-- again will advance the debugger to the next breakpoint.
--
```

```
-- While debugging, the window is switched to a different state where the
-- syntax tree preview is replaced by the evaluator state.

evaluatorAndStateRef <- newIORef Nothing
debuggerCond <- newEmptyMVar

Gtk.onButtonClicked playButton $ void $ do
  Gtk.widgetSetSensitive playButton False
  Gtk.widgetSetSensitive stopButton True

  readIORef evaluatorAndStateRef >>= \case
    Nothing -> do
      Gtk.textViewSetEditable codeView False
      setChild rightScrolledWindow stateView

      -- Set up the evaluator and its state
      Just syntaxTree <- readIORef syntaxTreeRef
      let evaluator = evalDevin syntaxTree
      state <- makePredefinedState
      writeIORef evaluatorAndStateRef (Just (evaluator, state))

    Just _ -> do
      -- Clear previous highlighting
      (startIter, endIter) <- Gtk.textBufferGetBounds codeBuffer
      Gtk.textBufferRemoveTag codeBuffer (highlightTag tags) startIter endIter

  -- Signal to start or resume debugging
  tryPutMVar debuggerCond ()

Gtk.onButtonClicked stopButton $ void $ do
  Gtk.widgetSetSensitive playButton False
  Gtk.widgetSetSensitive stopButton False
  writeIORef evaluatorAndStateRef Nothing
  tryPutMVar debuggerCond ()

-- Connect window callback for "delete-event" signals:

Gtk.onWidgetDeleteEvent window $ const $ do
  -- Save the current Devin code to $XDG_STATE_HOME/devin/main.devin
  (startIter, endIter) <- Gtk.textBufferGetBounds codeBuffer
  code <- Gtk.textIterGetText startIter endIter
  G.mkdirWithParents stateDirName 0o777
  try @IOException (Text.writeFile codeFileName code)

  -- Save the current UI state to $XDG_STATE_HOME/devin/ui.ini
  G.mkdirWithParents stateDirName 0o777
  try @GError (G.keyFileSaveToFile keyFile (Text.pack keyFileName))

  pure Gdk.EVENT_PROPAGATE

-- Load the last Devin code from $XDG_STATE_HOME/devin/main.devin:

try @IOException $ do
  code <- Text.readFile codeFileName
  Gtk.textBufferSetText codeBuffer code (fromIntegral (Text.length code))

  startIter <- Gtk.textBufferGetStartIter codeBuffer
```

```
  Gtk.textBufferPlaceCursor codeBuffer startIter

-- Start the worker thread responsible for parsing, type checking, and
-- performing syntax highlighting:

var1 <- newEmptyMVar
var2 <- newEmptyMVar
var3 <- newEmptyMVar

forkOS $ forever $ do
  readMVar parseAndTypeCheckCond

  postGUIASync $ do
    (startIter, endIter) <- Gtk.textBufferGetBounds codeBuffer
    code <- Gtk.textIterGetText startIter endIter
    takeMVar parseAndTypeCheckCond
    putMVar var1 code

  code <- takeMVar var1

  -- Run the parser
  parserResult <- parseT devin "" (0, code)

  case parserResult of
    -- Parser failure:
    Left parseError -> do
      offset <- toOffsetT (errorPos parseError) code
      let messages = errorMessages parseError

      postGUIASync $ do
        list <- seqStoreToList logModel
        startIter <- Gtk.textBufferGetIterAtOffset codeBuffer offset
        (line, column) <- getLineColumn startIter
        putMVar var2 (list, (line, column))

      (list, (line, column)) <- takeMVar var2
      let list' = [(displayLineColumn (line, column), showMessages messages)]
      let edits = diff list list'

      postGUIASync $ do
        writeIORef syntaxTreeRef Nothing

        -- Clear previous highlighting
        (startIter, endIter) <- Gtk.textBufferGetBounds codeBuffer
        clearSyntaxHighlighting codeBuffer tags startIter endIter
        clearBracketsHighlighting codeBuffer tags startIter endIter
        Gtk.textBufferRemoveTag codeBuffer (errorTag tags) startIter endIter

        -- Highlight as error, starting from the parseError offset
        startIter <- Gtk.textBufferGetIterAtOffset codeBuffer offset
        Gtk.textBufferApplyTag codeBuffer (errorTag tags) startIter endIter

        -- Hide the syntax tree preview
        setChild rightScrolledWindow blankView

        -- Update the error log
        patchSeqStore logModel edits
```

```
      Gtk.treeViewColumnsAutosize logView

  -- Parser success:
  Right (syntaxTree, comments) -> do
    postGUIASync $ do
      forest <- forestStoreGetForest syntaxTreeModel
      putMVar var3 forest

    forest <- takeMVar var3
    let forest' = map definitionTree (definitions syntaxTree)
    let edits = forestDiff forest forest'

    postGUIASync $ do
      writeIORef syntaxTreeRef (Just syntaxTree)

      -- Clear previous highlighting
      (startIter, endIter) <- Gtk.textBufferGetBounds codeBuffer
      clearSyntaxHighlighting codeBuffer tags startIter endIter
      clearBracketsHighlighting codeBuffer tags startIter endIter
      Gtk.textBufferRemoveTag codeBuffer (errorTag tags) startIter endIter

      -- Highlight syntax
      highlightDevin codeBuffer tags syntaxTree
      for_ comments (highlightInterval (commentTag tags) codeBuffer)

      -- Highlight matching braces
      insertMark <- Gtk.textBufferGetInsert codeBuffer
      insertIter <- Gtk.textBufferGetIterAtMark codeBuffer insertMark
      highlightDevinBrackets codeBuffer tags insertIter syntaxTree

      -- Show and update the syntax tree preview
      setChild rightScrolledWindow syntaxTreeView
      let expandPredicate (label, _) = not (Text.isSuffixOf "Expression" label)
      patchForestStore syntaxTreeModel edits syntaxTreeView expandPredicate
      Gtk.treeViewColumnsAutosize syntaxTreeView

      -- Clear the error log
      seqStoreClear logModel
      Gtk.treeViewColumnsAutosize logView

    -- Run the type checker
    let typerResult = runTyper (checkDevin syntaxTree) predefinedEnv

    case typerResult of
      -- Type checker success:
      ((), env, []) -> do
        let (hasMain, _, _) = runTyper checkHasMain env
        postGUIASync (Gtk.widgetSetSensitive playButton hasMain)

        where
          checkHasMain = lookupFunSignature "main" >>= \case
            Just (([], _), _) -> pure True
            _ -> pure False

      -- Type checker failure:
      ((), _, errors) -> postGUIASync $ for_ errors $ \error -> do
        -- Highlight the part of code which caused the failure
```

```
          startIter <- Gtk.textBufferGetIterAtOffset codeBuffer (start error)
          endIter <- Gtk.textBufferGetIterAtOffset codeBuffer (end error)
          Gtk.textBufferApplyTag codeBuffer (errorTag tags) startIter endIter

          -- Append the error description to the log
          (line, column) <- getLineColumn startIter
          let datum = (displayLineColumn (line, column), Text.pack (display error))
          seqStoreAppend logModel datum
          Gtk.treeViewColumnsAutosize logView

-- Start worker thread responsible for Devin code evaluation:

var1 <- newEmptyMVar
var2 <- newEmptyMVar

forkOS $ forever $ do
  readMVar debuggerCond

  postGUIASync $ do
    evaluatorAndState <- readIORef evaluatorAndStateRef
    takeMVar debuggerCond
    putMVar var1 evaluatorAndState

  evaluatorAndState <- takeMVar var1

  case evaluatorAndState of
    Nothing -> postGUIASync $ do
      -- Clear previous highlighting
      (startIter, endIter) <- Gtk.textBufferGetBounds codeBuffer
      Gtk.textBufferRemoveTag codeBuffer (highlightTag tags) startIter endIter

      -- Clear the evaluator state preview
      forestStoreClear stateModel

      -- Revert back to the initial window state
      Gtk.widgetSetSensitive playButton True
      Gtk.widgetSetSensitive stopButton False
      Gtk.textViewSetEditable codeView True
      setChild rightScrolledWindow syntaxTreeView

    Just (evaluator, state) -> do
      -- Evaluate a single statement
      (result, state') <- runEvaluatorStep evaluator state

      case result of
        -- Evaluator done:
        Done _ -> postGUIASync $ void $ do
          writeIORef evaluatorAndStateRef Nothing
          tryPutMVar debuggerCond ()

        -- Evaluator yield (breakpoint statement):
        Yield statement evaluator' | BreakpointStatement{} <- statement -> do
          postGUIASync $ do
            forest <- forestStoreGetForest stateModel
            putMVar var2 forest

          forest <- takeMVar var2
```

```
    forest' <- stateForest state'
    let edits = forestDiff forest forest'

    postGUIASync $ do
      -- Highlight the breakpoint statement
      highlightInterval (highlightTag tags) codeBuffer statement

      -- Update the evaluator state preview
      patchForestStore stateModel edits stateView (const True)
      Gtk.treeViewColumnsAutosize stateView

      -- Enable the play button to resume execution
      writeIORef evaluatorAndStateRef (Just (evaluator', state'))
      Gtk.widgetSetSensitive playButton True

  -- Evaluator yield (any other statement):
  Yield _ evaluator' -> postGUIASync $
    whenM (isJust <$> readIORef evaluatorAndStateRef) $ void $ do
      writeIORef evaluatorAndStateRef (Just (evaluator', state'))
      tryPutMVar debuggerCond ()

  -- Evaluator error:
  Error error -> do
    forest' <- stateForest state'

    postGUIASync $ void $ do
      -- Disable the stop button
      Gtk.widgetSetSensitive stopButton False

      -- Highlight the segment of code which caused the error
      startIter <- Gtk.textBufferGetIterAtOffset codeBuffer (start error)
      endIter <- Gtk.textBufferGetIterAtOffset codeBuffer (end error)
      Gtk.textBufferApplyTag codeBuffer (errorTag tags) startIter endIter

      -- Update the evaluator state preview
      forest <- forestStoreGetForest stateModel
      let edits = forestDiff forest forest'
      patchForestStore stateModel edits stateView (const True)
      Gtk.treeViewColumnsAutosize stateView

      -- Display the error message.
      --
      -- gi-gtk doesn't provide bindings for gtk_message_dialog_new().
      -- To get around this, we use new' from haskell-gi-base.

      (line, column) <- getLineColumn startIter
      let string = displaysLineColumn (line, column) (' ' : display error)

      headerBar <- Gtk.headerBarNew
      Gtk.headerBarSetTitle headerBar (Just "Error")
      Gtk.headerBarSetShowCloseButton headerBar True

      messageDialog <- new' Gtk.MessageDialog
        [
          Gtk.constructMessageDialogMessageType Gtk.MessageTypeError,
          Gtk.constructMessageDialogButtons Gtk.ButtonsTypeClose,
          Gtk.constructMessageDialogText (Text.pack string)
```

```
                ]

              messageArea <- Gtk.messageDialogGetMessageArea messageDialog
              addMonospaceClass messageArea

              Gtk.windowSetTitlebar messageDialog (Just headerBar)
              Gtk.windowSetTransientFor messageDialog (Just window)
              Gtk.widgetShowAll messageDialog
              Gtk.dialogRun messageDialog

              -- Prepare for cleanup. Since debuggerCond is signalled, the
              -- debugger worker thread should proceed by reverting back to the
              -- initial window state.

              Gtk.widgetDestroy messageDialog
              Gtk.textBufferRemoveTag codeBuffer (errorTag tags) startIter endIter
              writeIORef evaluatorAndStateRef Nothing
              tryPutMVar debuggerCond ()

  -- Display the UI:

  Gtk.widgetShowAll syntaxTreeView
  Gtk.widgetShowAll stateView
  Gtk.widgetShowAll window


addMonospaceClass :: (Gtk.IsWidget a, MonadIO m) => a -> m ()
addMonospaceClass widget = do
  context <- Gtk.widgetGetStyleContext widget
  Gtk.styleContextAddClass context Gtk.STYLE_CLASS_MONOSPACE


setChild :: (Gtk.IsBin a, Gtk.IsWidget b, MonadIO m) => a -> b -> m ()
setChild bin widget = do
  let bin' = bin `asA` Gtk.Bin
  whenJustM (Gtk.binGetChild bin') (Gtk.containerRemove bin')
  Gtk.containerAdd bin' widget


appendColumnWithDataFunction ::
  (Gtk.IsTreeView a, IsTypedTreeModel model, Gtk.IsTreeModel (model row), MonadIO m) =>
  a -> model row -> (Gtk.CellRendererText -> row -> IO ()) -> m Int32
appendColumnWithDataFunction view model f = do
  renderer <- Gtk.cellRendererTextNew
  column <- Gtk.treeViewColumnNew
  cellLayoutSetDataFunction column renderer model (f renderer)
  Gtk.cellLayoutPackStart column renderer True
  Gtk.treeViewAppendColumn view column


patchSeqStore :: MonadIO m => SeqStore a -> [Edit a] -> m ()
patchSeqStore model = foldM_ f 0
  where
    f i (Copy _ _) =
      pure (i + 1)

    f i (Insert x) = do
```

```
      seqStoreInsert model i x
      pure (i + 1)

    f i (Delete _) = do
      seqStoreRemove model i
      pure i

    f i (Replace _ x) = do
      seqStoreInsert model i x
      seqStoreRemove model (i + 1)
      pure (i + 1)


patchForestStore ::
  Gtk.IsTreeView b =>
  ForestStore a -> [TreeEdit a] -> b -> (a -> Bool) -> IO ()
patchForestStore model edits view expandPredicate = do
  path <- Gtk.treePathNew
  go path 0 edits

  where
    go _ _ [] =
      pure ()

    go path i (TreeCopy _ _ : edits) =
      go path (i + 1) edits

    go path i (TreeInsert tree : edits) = do
      forestStoreInsertTree model path (fromIntegral i) tree
      Gtk.treePathAppendIndex path i
      expand path
      Gtk.treePathUp path
      go path (i + 1) edits

    go path i (TreeDelete _ : edits) = do
      Gtk.treePathAppendIndex path i
      forestStoreRemove model path
      Gtk.treePathUp path
      go path i edits

    go path i (TreeReplace _ tree : edits) = do
      forestStoreInsertTree model path (fromIntegral i) tree
      Gtk.treePathAppendIndex path i
      expand path
      Gtk.treePathNext path
      forestStoreRemove model path
      Gtk.treePathUp path
      go path (i + 1) edits

    go path i (TreeUpdate _ edits' : edits) = do
      Gtk.treePathAppendIndex path i
      go path 0 edits'
      Gtk.treePathUp path
      go path (i + 1) edits

    expand path = do
      tree <- forestStoreGetTree model path
```

```
      when (expandPredicate (rootLabel tree)) $ void $ do
        Gtk.treeViewExpandRow view path False
        Gtk.treePathDown path

        for_ (subForest tree) $ const $ do
          expand path
          Gtk.treePathNext path

        Gtk.treePathUp path


getLine :: (Num a, MonadIO m) => Gtk.TextIter -> m a
getLine iter = do
  line <- Gtk.textIterGetLine iter
  pure (fromIntegral line + 1)


getColumn :: (Num a, MonadIO m) => Gtk.TextIter -> m a
getColumn iter = do
  iter' <- Gtk.textIterCopy iter
  Gtk.textIterSetLineOffset iter' 0

  flip loopM 1 $ \column -> do
    result <- Gtk.textIterCompare iter' iter

    if result >= 0 then
      pure (Right column)
    else do
      Gtk.textIterForwardCursorPosition iter'
      pure (Left (column + 1))


getLineColumn :: (Num a, MonadIO m) => Gtk.TextIter -> m (a, a)
getLineColumn iter = do
  line <- getLine iter
  column <- getColumn iter
  pure (line, column)


displayLineColumn :: (Show a, IsString b) => (a, a) -> b
displayLineColumn (line, column) = fromString (displaysLineColumn (line, column) "")


displaysLineColumn :: Show a => (a, a) -> ShowS
displaysLineColumn (line, column) =
  showChar '[' . shows line . showChar ':' . shows column . showChar ']'


showMessages :: IsString a => [Message] -> a
showMessages messages =
  let s1 = "or"
      s2 = "Unknown parse error"
      s3 = "Expecting"
      s4 = "Unexpected"
      s5 = "end of input"
```

```
   in case showErrorMessages s1 s2 s3 s4 s5 (filter f messages) of
        ('\n' : string) -> fromString string
        string -> fromString string

  where
    f (Expect _) = True
    f (Message _) = True
    f _ = False
```

A.19 app/Devin/Debug/Evaluator.hs

```
{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE OverloadedStrings #-}

module Devin.Debug.Evaluator (
  stateForest,
  displayVal,
  displaysVal
) where

import Control.Monad.IO.Class
import Data.Foldable
import Data.String
import Numeric

import Data.Tree

import qualified Data.Text as Text
import Data.Text (Text)

import qualified Data.Vector as Vector
import Data.Vector ((!))

import Devin.Evaluator


stateForest :: MonadIO m => State -> m (Forest (Text, Text))
stateForest = \case
  [] -> pure []

  frames -> do
    (tree, frames') <- go [] frames
    forest' <- stateForest frames'

    case tree of
      Node label [] -> pure (Node label [Node ("-", "") []] : forest')
      _ -> pure (tree : forest')

    where
      go result [] = pure (Node ("-", "") result, [])

      go result (frame : frames) = do
        result' <- foldlM f result (vars frame)

        case label frame of
```

```
          Nothing -> go result' frames
          Just label -> pure (Node (Text.pack label, "") result', frames)

      f varsForest (name, cell) = do
        val <- readCell cell
        valText <- displayVal val
        pure (Node (Text.pack name, valText) [] : varsForest)


displayVal :: (MonadIO m, IsString a) => Value -> m a
displayVal val = do
  s <- displaysVal val
  pure (fromString (s ""))


displaysVal :: MonadIO m => Value -> m ShowS
displaysVal = \case
  Unit -> pure (showString "unit")
  Bool x -> pure (showString (if x then "true" else "false"))
  Int x -> pure (shows x)

  Float x | isNaN x -> pure (showString "NaN")
  Float x | isInfinite x -> pure (showString (if x < 0 then "-ϖ" else "ϖ"))
  Float x -> pure (showFFloat Nothing x)

  Array cells | Vector.null cells -> pure (showString "[]")

  Array cells -> do
    val <- readCell (cells ! 0)
    s1 <- displaysVal val
    s2 <- go (Vector.length cells) 1
    pure (showChar '[' . s1 . s2)

    where
      go n i | i >= n = pure (showChar ']')

      go n i = do
        cell <- readCell (cells ! i)
        s1 <- displaysVal cell
        s2 <- go n (i + 1)
        pure (showString ", " . s1 . s2)
```

A.20 app/Devin/Debug/Syntax.hs

```
{-# LANGUAGE DisambiguateRecordFields #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE NamedFieldPuns #-}
{-# LANGUAGE OverloadedStrings #-}

module Devin.Debug.Syntax (
  devinTree,
  definitionTree,
  statementTree,
  expressionTree,
  unaryOperatorTree,
  binaryOperatorTree,
```

```
  symbolIdTree,
  tokenTree
) where

import Data.Tree

import qualified Data.Text as Text
import Data.Text (Text)

import Devin.Ratio
import Devin.Syntax


devinTree :: Devin -> Tree (Text, Text)
devinTree Devin{definitions} =
  Node ("Devin", "") (map definitionTree definitions)


definitionTree :: Definition -> Tree (Text, Text)
definitionTree = \case
  VarDefinition{varKeyword, varId, equalSign, value, semicolon} ->
    Node ("VarDefinition", "") [
      tokenTree "var" varKeyword,
      symbolIdTree varId,
      tokenTree "=" equalSign,
      expressionTree value,
      tokenTree ";" semicolon
    ]

  FunDefinition{defKeyword, funId, open, params, commas, close, returnInfo, body} ->
    Node ("FunDefinition", "") $ concat [
      [tokenTree "def" defKeyword],
      [symbolIdTree funId],
      [tokenTree "(" open],
      go params commas,
      [tokenTree ")" close],
      maybe [] (\(arrow, id) -> [tokenTree "->" arrow, typeIdTree id]) returnInfo,
      [statementTree body]
    ]

    where
      go ps [] = concatMap paramTree ps
      go [] cs = map (tokenTree ",") cs
      go (p : ps) (c : cs) = paramTree p ++ [tokenTree "," c] ++ go ps cs

      paramTree (refKeyword, paramId, paramInfo) =
        concat [
          maybe [] (\token -> [tokenTree "ref" token]) refKeyword,
          [symbolIdTree paramId],
          maybe [] (\(colon, t) -> [tokenTree ":" colon, typeIdTree t]) paramInfo
        ]


statementTree :: Statement -> Tree (Text, Text)
statementTree statement = case statement of
  DefinitionStatement{definition} ->
    Node ("DefinitionStatement", "") [definitionTree definition]
```

```
ExpressionStatement{effect, semicolon} ->
  Node ("ExpressionStatement", "") [
    expressionTree effect,
    tokenTree ";" semicolon
  ]

IfStatement{ifKeyword, predicate, trueBranch} ->
  Node ("IfStatement", "") [
    tokenTree "if" ifKeyword,
    expressionTree predicate,
    statementTree trueBranch
  ]

IfElseStatement{ifKeyword, predicate, trueBranch, elseKeyword, falseBranch} ->
  Node ("IfElseStatement", "") [
    tokenTree "if" ifKeyword,
    expressionTree predicate,
    statementTree trueBranch,
    tokenTree "else" elseKeyword,
    statementTree falseBranch
  ]

WhileStatement{whileKeyword, predicate, body} ->
  Node ("WhileStatement", "") [
    tokenTree "while" whileKeyword,
    expressionTree predicate,
    statementTree body
  ]

DoWhileStatement{doKeyword, body, whileKeyword, predicate, semicolon} ->
  Node ("DoWhileStatement", "") [
    tokenTree "do" doKeyword,
    statementTree body,
    tokenTree "while" whileKeyword,
    expressionTree predicate,
    tokenTree ";" semicolon
  ]

ReturnStatement{returnKeyword, result = Just result, semicolon} ->
  Node ("ReturnStatement", "") [
    tokenTree "return" returnKeyword,
    expressionTree result,
    tokenTree ";" semicolon
  ]

ReturnStatement{returnKeyword, result = Nothing, semicolon} ->
  Node ("ReturnStatement", "") [
    tokenTree "return" returnKeyword,
    tokenTree ";" semicolon
  ]

AssertStatement{assertKeyword, predicate, semicolon} ->
  Node ("AssertStatement", "") [
    tokenTree "assert" assertKeyword,
    expressionTree predicate,
    tokenTree ";" semicolon
```

```
    ]

  BreakpointStatement{breakpointKeyword, semicolon} ->
    Node ("BreakpointStatement", "") [
      tokenTree "breakpoint" breakpointKeyword,
      tokenTree ";" semicolon
    ]

  BlockStatement{open, statements, close} ->
    Node ("BlockStatement", "") $ concat [
      [tokenTree "{" open],
      map statementTree statements,
      [tokenTree "}" close]
    ]


expressionTree :: Expression -> Tree (Text, Text)
expressionTree = \case
  IntegerExpression{integer} ->
    Node ("IntegerExpression", Text.pack (show integer)) []

  RationalExpression{rational} ->
    Node ("RationalExpression", Text.pack (displayRatio rational)) []

  VarExpression{varName} ->
    Node ("VarExpression", Text.pack varName) []

  ArrayExpression{open, elems = es, commas = cs, close} ->
    Node ("ArrayExpression", "") $ concat [
      [tokenTree "[" open],
      go es cs,
      [tokenTree "]" close]
    ]

    where
      go es [] = map expressionTree es
      go [] cs = map (tokenTree ",") cs
      go (e : es) (c : cs) = expressionTree e : tokenTree "," c : go es cs

  AccessExpression{array, open, index, close} ->
    Node ("AccessExpression", "") [
      expressionTree array,
      tokenTree "[" open,
      expressionTree index,
      tokenTree "]" close
    ]

  CallExpression{funId, open, args = as, commas = cs, close} ->
    Node ("CallExpression", "") $ concat [
      [symbolIdTree funId, tokenTree "(" open],
      go as cs,
      [tokenTree ")" close]
    ]

    where
      go as [] = map expressionTree as
      go [] cs = map (tokenTree ",") cs
```

```
      go (a : as) (c : cs) = expressionTree a : tokenTree "," c : go as cs

  UnaryExpression{unary, operand} ->
    Node ("UnaryExpression", "") [
      unaryOperatorTree unary,
      expressionTree operand
    ]

  BinaryExpression{left, binary, right} ->
    Node ("BinaryExpression", "") [
      expressionTree left,
      binaryOperatorTree binary,
      expressionTree right
    ]

  ParenthesizedExpression{open, inner, close} ->
    Node ("ParenthesizedExpression", "") [
      tokenTree "(" open,
      expressionTree inner,
      tokenTree ")" close
    ]


unaryOperatorTree :: UnaryOperator -> Tree (Text, Text)
unaryOperatorTree PlusOperator{} = Node ("PlusOperator", "+") []
unaryOperatorTree MinusOperator{} = Node ("MinusOperator", "-") []


binaryOperatorTree :: BinaryOperator -> Tree (Text, Text)
binaryOperatorTree = \case
  AddOperator{} -> Node ("AddOperator", "+") []
  SubtractOperator{} -> Node ("SubtractOperator", "-") []
  MultiplyOperator{} -> Node ("MultiplyOperator", "*") []
  DivideOperator{} -> Node ("DivideOperator", "/") []
  ModuloOperator{} -> Node ("ModuloOperator", "%") []
  EqualOperator{} -> Node ("EqualOperator", "==") []
  NotEqualOperator{} -> Node ("NotEqualOperator", "!=") []
  LessOperator{} -> Node ("LessOperator", "<") []
  LessOrEqualOperator{} -> Node ("LessOrEqualOperator", "<=") []
  GreaterOperator{} -> Node ("GreaterOperator", ">") []
  GreaterOrEqualOperator{} -> Node ("GreaterOrEqualOperator", ">=") []
  AndOperator{} -> Node ("AndOperator", "and") []
  OrOperator{} -> Node ("OrOperator", "or") []
  XorOperator{} -> Node ("XorOperator", "xor") []
  PlainAssignOperator{} -> Node ("PlainAssignOperator", "=") []
  AddAssignOperator{} -> Node ("AddAssignOperator", "+=") []
  SubtractAssignOperator{} -> Node ("SubtractAssignOperator", "-=") []
  MultiplyAssignOperator{} -> Node ("MultiplyAssignOperator", "*=") []
  DivideAssignOperator{} -> Node ("DivideAssignOperator", "/=") []
  ModuloAssignOperator{} -> Node ("ModuloAssignOperator", "%=") []


symbolIdTree :: SymbolId -> Tree (Text, Text)
symbolIdTree SymbolId{name} = Node ("SymbolId", Text.pack name) []


typeIdTree :: TypeId -> Tree (Text, Text)
```

```
typeIdTree = \case
  PlainTypeId{name} ->
    Node ("PlainTypeId", Text.pack name) []

  ArrayTypeId{open, innerTypeId, close} ->
    Node ("ArrayTypeId", "") [
      tokenTree "[" open,
      typeIdTree innerTypeId,
      tokenTree "]" close
    ]


tokenTree :: String -> Token -> Tree (Text, Text)
tokenTree lexeme Token{} = Node ("Token", Text.pack lexeme) []
```

A.21 app/Devin/Highlight.hs

```
{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE MonoLocalBinds #-}
{-# LANGUAGE OverloadedStrings #-}

module Devin.Highlight (
  HighlightingTags (..),
  getHighlightingTags,
  addHighlightingTags,
  highlightInterval
) where

import Control.Monad.IO.Class

import qualified Data.Set as Set

import Data.Text (Text)

import Control.Monad.Extra

import qualified GI.Gtk as Gtk
import qualified GI.GtkSource as GtkSource

import Devin.Interval


data HighlightingTags = HighlightingTags {
  highlightTag :: GtkSource.Tag,
  bracketTag :: GtkSource.Tag,
  keywordTag :: GtkSource.Tag,
  varIdTag :: GtkSource.Tag,
  funIdTag :: GtkSource.Tag,
  typeTag :: GtkSource.Tag,
  numberTag :: GtkSource.Tag,
  operatorTag :: GtkSource.Tag,
  commentTag :: GtkSource.Tag,
  errorTag :: GtkSource.Tag
} deriving Eq
```

```
getHighlightingTags ::
  (GtkSource.IsStyleScheme a, MonadIO m) =>
  Maybe a -> m HighlightingTags
getHighlightingTags scheme = do
  languageManager <- GtkSource.languageManagerGetDefault
  defaultLanguage <- GtkSource.languageManagerGetLanguage languageManager "def"

  tag01 <- getTag defaultLanguage scheme "search-match"
  tag02 <- getTag defaultLanguage scheme "bracket-match"
  tag03 <- getTag defaultLanguage scheme "def:keyword"
  tag04 <- getTag defaultLanguage scheme "def:identifier"
  tag05 <- getTag defaultLanguage scheme "def:function"
  tag06 <- getTag defaultLanguage scheme "def:type"
  tag07 <- getTag defaultLanguage scheme "def:number"
  tag08 <- getTag defaultLanguage scheme "def:operator"
  tag09 <- getTag defaultLanguage scheme "def:comment"
  tag10 <- getTag defaultLanguage scheme "def:error"

  pure (HighlightingTags tag01 tag02 tag03 tag04 tag05 tag06 tag07 tag08 tag09 tag10)


addHighlightingTags ::
  (Gtk.IsTextTagTable a, MonadIO m) =>
  a -> HighlightingTags -> m Bool
addHighlightingTags tagTable tags = do
  Gtk.textTagTableAdd tagTable (highlightTag tags)
  Gtk.textTagTableAdd tagTable (bracketTag tags)
  Gtk.textTagTableAdd tagTable (keywordTag tags)
  Gtk.textTagTableAdd tagTable (varIdTag tags)
  Gtk.textTagTableAdd tagTable (funIdTag tags)
  Gtk.textTagTableAdd tagTable (typeTag tags)
  Gtk.textTagTableAdd tagTable (numberTag tags)
  Gtk.textTagTableAdd tagTable (operatorTag tags)
  Gtk.textTagTableAdd tagTable (commentTag tags)
  Gtk.textTagTableAdd tagTable (errorTag tags)


highlightInterval ::
  (Gtk.IsTextTag a, Gtk.IsTextBuffer b, Interval c, MonadIO m) =>
  a -> b -> c -> m ()
highlightInterval tag buffer interval = do
  startIter <- Gtk.textBufferGetIterAtOffset buffer (start interval)
  endIter <- Gtk.textBufferGetIterAtOffset buffer (end interval)
  Gtk.textBufferApplyTag buffer tag startIter endIter


getTag ::
  (GtkSource.IsLanguage a, GtkSource.IsStyleScheme b, MonadIO m) =>
  Maybe a -> Maybe b -> Text -> m GtkSource.Tag
getTag language scheme styleId = do
  maybeStyle <- case (scheme, language) of
    (Just scheme, Just language) -> getStyle language scheme styleId
    (Just scheme, Nothing) -> GtkSource.styleSchemeGetStyle scheme styleId
    (Nothing, _) -> pure Nothing

  tag <- GtkSource.tagNew Nothing
```

```
  whenJust maybeStyle (\style -> GtkSource.styleApply style tag)
  pure tag


getStyle ::
  (GtkSource.IsLanguage a, GtkSource.IsStyleScheme b, MonadIO m) =>
  a -> b -> Text -> m (Maybe GtkSource.Style)
getStyle language scheme styleId = go Set.empty styleId
  where
    go seen styleId = GtkSource.styleSchemeGetStyle scheme styleId >>= \case
      Just style -> pure (Just style)

      Nothing | Set.member styleId seen -> pure Nothing

      Nothing -> GtkSource.languageGetStyleFallback language styleId >>= \case
        Just fallbackStyleId -> go (Set.insert styleId seen) fallbackStyleId
        Nothing -> pure Nothing
```

A.22 app/Devin/Highlight/Brackets.hs

```
{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE DisambiguateRecordFields #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE MonoLocalBinds #-}
{-# LANGUAGE NamedFieldPuns #-}

module Devin.Highlight.Brackets (
  clearBracketsHighlighting,
  highlightDevinBrackets,
  highlightDefinitionBrackets,
  highlightStatementBrackets,
  highlightExpressionBrackets
) where

import Control.Monad.IO.Class

import Control.Monad.Extra

import qualified GI.Gtk as Gtk

import Devin.Interval
import Devin.Syntax

import Devin.Highlight


clearBracketsHighlighting ::
  (Gtk.IsTextBuffer a, MonadIO m) =>
  a -> HighlightingTags -> Gtk.TextIter -> Gtk.TextIter -> m ()
clearBracketsHighlighting buffer tags =
  Gtk.textBufferRemoveTag buffer (bracketTag tags)


highlightDevinBrackets ::
  (Gtk.IsTextBuffer a, MonadIO m) =>
  a -> HighlightingTags -> Gtk.TextIter -> Devin -> m Bool
```

```
highlightDevinBrackets buffer tags insertIter Devin{definitions} =
  anyM (highlightDefinitionBrackets buffer tags insertIter) definitions


highlightDefinitionBrackets ::
  (Gtk.IsTextBuffer a, MonadIO m) =>
  a -> HighlightingTags -> Gtk.TextIter -> Definition -> m Bool
highlightDefinitionBrackets buffer tags insertIter = \case
  VarDefinition{value} ->
    highlightExpressionBrackets buffer tags insertIter value

  FunDefinition{open, close, body} ->
    orM [
      highlightBrackets buffer tags insertIter open close,
      highlightStatementBrackets buffer tags insertIter body
    ]


highlightStatementBrackets ::
  (Gtk.IsTextBuffer a, MonadIO m) =>
  a -> HighlightingTags -> Gtk.TextIter -> Statement -> m Bool
highlightStatementBrackets buffer tags insertIter = \case
  ReturnStatement{result = Nothing} -> pure False
  BreakpointStatement{} -> pure False

  ExpressionStatement{effect} ->
    highlightExpressionBrackets buffer tags insertIter effect

  DefinitionStatement{definition} ->
    highlightDefinitionBrackets buffer tags insertIter definition

  IfStatement{predicate, trueBranch} ->
    orM [
      highlightExpressionBrackets buffer tags insertIter predicate,
      highlightStatementBrackets buffer tags insertIter trueBranch
    ]

  IfElseStatement{predicate, trueBranch, falseBranch} ->
    orM [
      highlightExpressionBrackets buffer tags insertIter predicate,
      highlightStatementBrackets buffer tags insertIter trueBranch,
      highlightStatementBrackets buffer tags insertIter falseBranch
    ]

  WhileStatement{predicate, body} ->
    orM [
      highlightExpressionBrackets buffer tags insertIter predicate,
      highlightStatementBrackets buffer tags insertIter body
    ]

  DoWhileStatement{body, predicate} ->
    orM [
      highlightStatementBrackets buffer tags insertIter body,
      highlightExpressionBrackets buffer tags insertIter predicate
    ]

  ReturnStatement{result = Just result} ->
```

```
    highlightExpressionBrackets buffer tags insertIter result

  AssertStatement{predicate} ->
    highlightExpressionBrackets buffer tags insertIter predicate

  BlockStatement{open, statements, close} ->
    orM [
      anyM (highlightStatementBrackets buffer tags insertIter) statements,
      highlightBrackets buffer tags insertIter open close
    ]


highlightExpressionBrackets ::
  (Gtk.IsTextBuffer a, MonadIO m) =>
  a -> HighlightingTags -> Gtk.TextIter -> Expression -> m Bool
highlightExpressionBrackets buffer tags insertIter = \case
  IntegerExpression{} -> pure False
  RationalExpression{} -> pure False
  VarExpression{} -> pure False

  ArrayExpression{open, elems, close} ->
    orM [
      anyM (highlightExpressionBrackets buffer tags insertIter) elems,
      highlightBrackets buffer tags insertIter open close
    ]

  AccessExpression{array, open, index, close} ->
    orM [
      highlightExpressionBrackets buffer tags insertIter array,
      highlightExpressionBrackets buffer tags insertIter index,
      highlightBrackets buffer tags insertIter open close
    ]

  CallExpression{open, args, close} ->
    orM [
      anyM (highlightExpressionBrackets buffer tags insertIter) args,
      highlightBrackets buffer tags insertIter open close
    ]

  UnaryExpression{operand} ->
    highlightExpressionBrackets buffer tags insertIter operand

  BinaryExpression{left, right} ->
    orM [
      highlightExpressionBrackets buffer tags insertIter left,
      highlightExpressionBrackets buffer tags insertIter right
    ]

  ParenthesizedExpression{open, inner, close} ->
    orM [
      highlightExpressionBrackets buffer tags insertIter inner,
      highlightBrackets buffer tags insertIter open close
    ]


highlightBrackets ::
  (Gtk.IsTextBuffer a, Interval b, Interval c, MonadIO m) =>
```

```
  a -> HighlightingTags -> Gtk.TextIter -> b -> c -> m Bool
highlightBrackets buffer tags insertIter open close = do
  openStartIter <- Gtk.textBufferGetIterAtOffset buffer (start open)
  openEndIter <- Gtk.textBufferGetIterAtOffset buffer (end open)
  closeStartIter <- Gtk.textBufferGetIterAtOffset buffer (start close)
  closeEndIter <- Gtk.textBufferGetIterAtOffset buffer (end close)

  applyBracketTag <- anyM (Gtk.textIterEqual insertIter)
    [openStartIter, openEndIter, closeStartIter, closeEndIter]

  if applyBracketTag then do
    Gtk.textBufferApplyTag buffer (bracketTag tags) openStartIter openEndIter
    Gtk.textBufferApplyTag buffer (bracketTag tags) closeStartIter closeEndIter
    pure True
  else
    pure False
```

A.23 app/Devin/Highlight/Syntax.hs

```
{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE DisambiguateRecordFields #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE MonoLocalBinds #-}
{-# LANGUAGE NamedFieldPuns #-}

module Devin.Highlight.Syntax (
  clearSyntaxHighlighting,
  highlightDevin,
  highlightDefinition,
  highlightStatement,
  highlightExpression,
) where

import Control.Monad.IO.Class
import Data.Foldable

import Control.Monad.Extra

import qualified GI.Gtk as Gtk

import Devin.Syntax

import Devin.Highlight


clearSyntaxHighlighting ::
  (Gtk.IsTextBuffer a, MonadIO m) =>
  a -> HighlightingTags -> Gtk.TextIter -> Gtk.TextIter -> m ()
clearSyntaxHighlighting buffer tags startIter endIter = do
  Gtk.textBufferRemoveTag buffer (keywordTag tags) startIter endIter
  Gtk.textBufferRemoveTag buffer (varIdTag tags) startIter endIter
  Gtk.textBufferRemoveTag buffer (funIdTag tags) startIter endIter
  Gtk.textBufferRemoveTag buffer (typeTag tags) startIter endIter
  Gtk.textBufferRemoveTag buffer (numberTag tags) startIter endIter
  Gtk.textBufferRemoveTag buffer (operatorTag tags) startIter endIter
  Gtk.textBufferRemoveTag buffer (commentTag tags) startIter endIter
```

```
  Gtk.textBufferRemoveTag buffer (errorTag tags) startIter endIter


highlightDevin ::
  (Gtk.IsTextBuffer a, MonadIO m) =>
  a -> HighlightingTags -> Devin -> m ()
highlightDevin buffer tags Devin{definitions} =
  for_ definitions (highlightDefinition buffer tags)


highlightDefinition ::
  (Gtk.IsTextBuffer a, MonadIO m) =>
  a -> HighlightingTags -> Definition -> m ()
highlightDefinition buffer tags = \case
  VarDefinition{varKeyword, varId, value} -> do
    highlightInterval (keywordTag tags) buffer varKeyword
    highlightInterval (varIdTag tags) buffer varId
    highlightExpression buffer tags value

  FunDefinition{defKeyword, funId, params, returnInfo, body} -> do
    highlightInterval (keywordTag tags) buffer defKeyword
    highlightInterval (funIdTag tags) buffer funId

    for_ params $ \(refKeyword, paramId, paramInfo) -> do
      whenJust refKeyword (highlightInterval (keywordTag tags) buffer)
      highlightInterval (varIdTag tags) buffer paramId
      whenJust paramInfo $ \(_, id) -> highlightInterval (typeTag tags) buffer id

    whenJust returnInfo $ \(_, id) -> highlightInterval (typeTag tags) buffer id
    highlightStatement buffer tags body


highlightStatement ::
  (Gtk.IsTextBuffer a, MonadIO m) =>
  a -> HighlightingTags -> Statement -> m ()
highlightStatement buffer tags = \case
  DefinitionStatement{definition} ->
    highlightDefinition buffer tags definition

  ExpressionStatement{effect} ->
    highlightExpression buffer tags effect

  IfStatement{ifKeyword, predicate, trueBranch} -> do
    highlightInterval (keywordTag tags) buffer ifKeyword
    highlightExpression buffer tags predicate
    highlightStatement buffer tags trueBranch

  IfElseStatement{ifKeyword, predicate, trueBranch, elseKeyword, falseBranch} -> do
    highlightInterval (keywordTag tags) buffer ifKeyword
    highlightExpression buffer tags predicate
    highlightStatement buffer tags trueBranch
    highlightInterval (keywordTag tags) buffer elseKeyword
    highlightStatement buffer tags falseBranch

  WhileStatement{whileKeyword, predicate, body} -> do
    highlightInterval (keywordTag tags) buffer whileKeyword
    highlightExpression buffer tags predicate
```

```
    highlightStatement buffer tags body

  DoWhileStatement{doKeyword, body, whileKeyword, predicate} -> do
    highlightInterval (keywordTag tags) buffer doKeyword
    highlightStatement buffer tags body
    highlightInterval (keywordTag tags) buffer whileKeyword
    highlightExpression buffer tags predicate

  ReturnStatement{returnKeyword, result} -> do
    highlightInterval (keywordTag tags) buffer returnKeyword
    whenJust result (highlightExpression buffer tags)

  AssertStatement{assertKeyword, predicate} -> do
    highlightInterval (keywordTag tags) buffer assertKeyword
    highlightExpression buffer tags predicate

  BreakpointStatement{breakpointKeyword} ->
    highlightInterval (keywordTag tags) buffer breakpointKeyword

  BlockStatement{statements} ->
    for_ statements (highlightStatement buffer tags)


highlightExpression ::
  (Gtk.IsTextBuffer a, MonadIO m) =>
  a -> HighlightingTags -> Expression -> m ()
highlightExpression buffer tags expression = case expression of
  IntegerExpression{} ->
    highlightInterval (numberTag tags) buffer expression

  RationalExpression{} ->
    highlightInterval (numberTag tags) buffer expression

  VarExpression{} ->
    highlightInterval (varIdTag tags) buffer expression

  ArrayExpression{elems} ->
    for_ elems (highlightExpression buffer tags)

  AccessExpression{array, index} -> do
    highlightExpression buffer tags array
    highlightExpression buffer tags index

  CallExpression{funId, args} -> do
    highlightInterval (funIdTag tags) buffer funId
    for_ args (highlightExpression buffer tags)

  UnaryExpression{unary, operand} -> do
    highlightInterval (operatorTag tags) buffer unary
    highlightExpression buffer tags operand

  BinaryExpression{left, binary, right} -> do
    highlightExpression buffer tags left
    highlightInterval (operatorTag tags) buffer binary
    highlightExpression buffer tags right

  ParenthesizedExpression{inner} ->
```

```
    highlightExpression buffer tags inner
```

A.24 app/Devin/Levenshtein.hs

```
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE DeriveTraversable #-}
{-# LANGUAGE LambdaCase #-}

module Devin.Levenshtein (
  Edit (..),
  TreeEdit (..),
  levenshteinBy,
  levenshteinOn,
  levenshtein,
  distanceBy,
  distanceOn,
  distance,
  diffWith,
  diffOn,
  diff,
  treeDiffBy,
  treeDiffOn,
  treeDiff,
  forestDiffBy,
  forestDiffOn,
  forestDiff
) where

import Data.Data
import Data.Function
import qualified Data.List.NonEmpty as NonEmpty
import Data.List.NonEmpty (NonEmpty ((:|)), (<|))

import Data.Tree

import Data.List.Extra


data Edit a
  = Copy a a
  | Insert a
  | Delete a
  | Replace a a
  deriving (Eq, Foldable, Traversable, Functor, Show, Read, Data)


data TreeEdit a
  = TreeCopy (Tree a) (Tree a)
  | TreeInsert (Tree a)
  | TreeDelete (Tree a)
  | TreeReplace (Tree a) (Tree a)
  | TreeUpdate (Tree a) [TreeEdit a]
  deriving (Eq, Foldable, Traversable, Functor, Show, Read, Data)


levenshteinBy :: Real b => (a -> a -> Bool) -> [a] -> [a] -> (b, [Edit a])
```

```
levenshteinBy eq xs ys =
  let f (cost, editDL) y = (cost + 1, editDL . (Insert y :))
      row0 = NonEmpty.scanl f (0, id) ys
      (cost, editDL) = NonEmpty.last (foldl nextRow row0 xs)
   in (cost, editDL [])

  where
    nextRow ((cost, editDL) :| cells) x =
      go (cost + 1, editDL . (Delete x :)) ((cost, editDL) :| cells) ys

      where
        go wCell (nwCell :| nCell : cells) (y : ys) =
          let cell = nextCell wCell nwCell nCell x y
           in wCell <| go cell (nCell :| cells) ys

        go wCell _ _ = NonEmpty.singleton wCell

    nextCell (wCost, wEditDL) (nwCost, nwEditDL) (nCost, nEditDL) x y =
      if x `eq` y then
        (nwCost, nwEditDL . (Copy x y :))
      else
        minimumOn fst [
          (wCost + 1, wEditDL . (Insert y :)),
          (nCost + 1, nEditDL . (Delete x :)),
          (nwCost + 1, nwEditDL . (Replace x y :))
        ]


levenshteinOn :: (Eq b, Real c) => (a -> b) -> [a] -> [a] -> (c, [Edit a])
levenshteinOn f = levenshteinBy ((==) `on` f)


levenshtein :: (Eq a, Real b) => [a] -> [a] -> (b, [Edit a])
levenshtein = levenshteinOn id


distanceBy :: Real b => (a -> a -> Bool) -> [a] -> [a] -> b
distanceBy eq xs ys = fst (levenshteinBy eq xs ys)


distanceOn :: (Eq b, Real c) => (a -> b) -> [a] -> [a] -> c
distanceOn f = distanceBy ((==) `on` f)


distance :: (Eq a, Real b) => [a] -> [a] -> b
distance = distanceOn id


diffWith :: (a -> a -> Bool) -> [a] -> [a] -> [Edit a]
diffWith eq xs ys = snd (levenshteinBy eq xs ys)


diffOn :: Eq b => (a -> b) -> [a] -> [a] -> [Edit a]
diffOn f = diffWith ((==) `on` f)


diff :: Eq a => [a] -> [a] -> [Edit a]
```

```
diff = diffOn id


treeDiffBy :: (a -> a -> Bool) -> Tree a -> Tree a -> TreeEdit a
treeDiffBy eq tree1 tree2
  | rootLabel tree1 `eq` rootLabel tree2 = treeDiffHelper eq tree1 tree2
  | otherwise = TreeReplace tree1 tree2


treeDiffOn :: Eq b => (a -> b) -> Tree a -> Tree a -> TreeEdit a
treeDiffOn f = treeDiffBy ((==) `on` f)


treeDiff :: Eq a => Tree a -> Tree a -> TreeEdit a
treeDiff = treeDiffOn id


forestDiffBy :: (a -> a -> Bool) -> Forest a -> Forest a -> [TreeEdit a]
forestDiffBy eq forest1 forest2 =
  flip map (diffWith (eq `on` rootLabel) forest1 forest2) $ \case
    Copy tree1 tree2 -> treeDiffHelper eq tree1 tree2
    Insert tree2 -> TreeInsert tree2
    Delete tree1 -> TreeDelete tree1
    Replace tree1 tree2 -> TreeReplace tree1 tree2


forestDiffOn :: Eq b => (a -> b) -> Forest a -> Forest a -> [TreeEdit a]
forestDiffOn f = forestDiffBy ((==) `on` f)


forestDiff :: Eq a => Forest a -> Forest a -> [TreeEdit a]
forestDiff = forestDiffOn id


treeDiffHelper :: (a -> a -> Bool) -> Tree a -> Tree a -> TreeEdit a
treeDiffHelper eq tree1 tree2 =
  let edits = forestDiffBy eq (subForest tree1) (subForest tree2)
   in if all isCopy edits then TreeCopy tree1 tree2 else TreeUpdate tree1 edits

  where
    isCopy (TreeCopy _ _) = True
    isCopy _ = False
```

Bibliography

[1] *Haskell*. URL: https://www.haskell.org/.

[2] *Haskell 2010 Language Report*. Chapter 18. URL: https://www.haskell.org/onlinereport/haskell2010/haskellch18.html#x26-22400018.1.

[3] *Haskell 2010 Language Report*. Chapter 13. URL: https://www.haskell.org/onlinereport/haskell2010/haskellch13.html#x21-19400013.1.

[4] Philip Wadler. "Monads for functional programming." In: *Program Design Calculi*. Edited by Manfred Broy. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pages 233–264. ISBN: 978-3-662-02880-3. DOI: 10.1007/978-3-662-02880-3_8.

[5] Donald E. Knuth. "On the translation of languages from left to right." In: *Information and Control* 8.6 (1965), pages 607–639. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(65)90426-2.

[6] Robert Nystrom. *Crafting Interpreters*. Chapter 6. URL: https://craftinginterpreters.com/parsing-expressions.html#recursive-descent-parsing.

[7] Gilad Bracha. "Pluggable Type Systems." 2004. URL: https://bracha.org/pluggableTypesPosition.pdf.

[8] *GTK+ 3 Reference Manual*. URL: https://developer-old.gnome.org/gtk3/stable/gtk.html.

[9] *GtkSourceView*. URL: https://developer-old.gnome.org/gtksourceview/stable/intro.html.

[10] *MDN Web Docs*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/isSafeInteger.

[11] *The Go Programming Language Specification*. URL: https://go.dev/ref/spec#Numeric_types.

[12] *Rust By Example*. Chapter 6. URL: https://doc.rust-lang.org/rust-by-example/conversion.html.

[13] *extra*. URL: https://hackage.haskell.org/package/extra.

[14] *parsec*. URL: https://hackage.haskell.org/package/parsec.

[15] *Unicode Identifier and Pattern Syntax*. URL: https://unicode.org/reports/tr31/.

[16] *gi-gtk*. URL: https://hackage.haskell.org/package/gi-gtk.

[17] *Scala 3 Reference*. URL: https://docs.scala-lang.org/scala3/reference/contextual/.

[18] *Koka*. URL: https://koka-lang.org/.