Recursive Island Parsing: Monadic Lake Parser Combinators

GERK-JAN HUISMA, University of Twente, The Netherlands

Island parsing is a semi-parsing technique that involves only parsing interesting fragments of an input text, called islands, and leaving uninteresting fragments, called water, unparsed. By allowing these islands to contain lakes and lakes to contain islands, it is possible to support the semi-parsing of nested constructs such as conditional or iteration statements in an imperative programming language. In functional programming, monadic parser combinators are a popular approach to building recursive descent parsers. This research paper outlines the step-by-step design, implementation, and verification of a set of monadic lake parser combinators for recursive island parsing by combining previous work done in the fields of semi-parsing and monadic parser combinators.

Additional Key Words and Phrases: Semi-Parsing, Island Parsing, Lake Symbols, Monadic Parser Combinators, Functional Programming, Haskell

1 INTRODUCTION

In the field of syntactical analysis, semi-parsing is the collective term coined by Dean et al. to describe the process of only partially analysing a string of symbols conforming to the rules of a formal grammar, recognizing only relevant constructs and skipping the analysis of irrelevant constructs. [2]

If we want to compile and execute a program, we need full syntactic analysis and a complete syntax tree. However, on the other side of the spectrum, pure lexical analysis can be used for low-precision tasks such as keyword frequency analysis. [19]

In between these two extremes lies, for example, control flow analysis. If we want to analyse the control flow of a program, only *if-else* and similar statements are constructs of interest. In theory, the rest of the input text could be skipped over. But, without the use of semi-parsing techniques, it can be a resource-intensive and error-prone task to design a proper semi-parser from scratch. [16]

Existing techniques all fall somewhere on the spectrum between almost full syntactic analysis and pure lexical analysis. For example, take agile parsing, fuzzy parsing and skeleton grammars. [2, 9, 10] However, most techniques are often not directly comparable and hard to reuse. One of the more generalizable ideas is that of island- and lake-grammars, and lake symbols. [14, 16, 18]

Island parsing involves only extracting interesting parts of an input text as an *island* and skipping over the rest as *water*. [18] A body of water inside an island is called a *lake*. If it would be possible to define islands within lakes as well, we could semi-parse nested constructs such as the conditional or iteration statements that commonly occur in imperative programming languages. [14] We use the term *recursive island parsers* to denote island parsers that support lakes in islands and islands in lakes.

It would be fairly trivial to define the *sea* in which the islands of interest exist using something similar to a wildcard character. However, it is far more difficult to define the lakes inside an island. The parser needs to know what patterns must not be taken as water to exclude the parts of the island from the lakes. This is a burden to developers implementing a recursive island parser, as this is not an easy task to do. [16]

Previous semi-parsing solutions all involved algorithmic approaches, of which only one enabled the user to design actual recursive island parsers. By shifting to a strictly functional paradigm and using only pure functions, we gain an important benefit. The outputs are always the same for any given inputs. This ensures determinism and predictability, making testing and verification easier and more reliable. [7]

In functional programming, a popular approach to designing recursive descent parsers is to model parsers as functions and to define higher-order functions (or combinators) that implement grammar constructions such as sequencing, choice, and repetition. [8] Combining the research on monadic parser combinators and semi-parsing techniques raises the question:

RQ: To what extent would it be possible to design a set of monadic lake parser combinators for recursive island parsing?

To answer this research question, we performed experimental research, exploring the inner workings of existing monadic parser combinators libraries and related research. [12] Combining these findings with existing semi-parsing techniques, to discover which ideas and principles might translate well into the functional programming domain, and which can or might not.

In this document, we further explore the requirements of the intended system, compare this set of requirements to existing solutions and finally propose and verify a new architecture for designing recursive island parsers.

TScIT 37, July 8, 2022, Enschede, The Netherlands

2 REQUIREMENTS

To decide on the requirements for the intended system, it is necessary to thoroughly understand the end-user and their needs. In our case, the user will most likely be a software engineer or a researcher. They will want to use the system to successfully create recursive island parsers and use these for analysis purposes. For example, determining code metrics like cyclomatic complexity, or performing further research on the space and time complexity of the system compared to performing full syntactic analysis.
At this preliminary stage in the design process, we are assuming that it is even possible to implement a set of parser combinators for recursive island parsing.
By utilising the MoSCoW method, which is a prioritisation technique used in software development, we can distinguish the following categories of requirements:

2.1 Must-have

R1: The system should be created using, and for use in, a functional programming language.

R2: Using the system, it should be possible to design parsers that can perform recursive island parsing.

2.2 Should-have

R3: The system should automatically determine the *alternative symbols*, which are the symbols that signify the boundaries of the lakes and where the islands begin again, and abstract the complexity of selecting these alternative symbols away from the user.

2.3 Could-have

R4: The system should use monads to define the set of lake parser combinators succinctly.

3 EXISTING SOLUTIONS

In this section we will take a closer look into existing solutions and which of our requirements they satisfy.

3.1 Monadic Parser Combinators

Parser combinators provide a quick and easy method of building functional parsers. One has the full power of a functional language available to define new combinators for special applications. [8]

It was realised early on that parsers form an instance of a monad, an algebraic structure from mathematics. Using monads brings several practical benefits. For example, using a monadic sequencing combinator for parsers avoids the messy manipulation of nested tuples of results present in earlier work. Moreover, monad comprehension notation makes parsers more compact and easier to read. [8, 12]

This flexibility in being able to define new combinators for special applications makes it ideal for extending upon existing solutions with a new set of parser combinators for recursive island parsing. However, such a set of parser combinators does not exist as of yet.

3.2 Island- and Lake-Grammars

In 1999, Deursen and Kuipers introduced the concept of *island grammars* to simplify the construction of documentation generators. [18] Island grammars only define syntactic structures of interest.

In 2001, Moonen used the term *water* to describe uninteresting syntactic structures in island grammars and proposed *lake grammars* and the idea of starting with a complete grammar for a given language and extending that grammar with several bodies of water. [14] An uninteresting body of water inside an island of interest is called a *lake*. Furthermore, in the resulting lake grammar, we can mix bodies of water and islands to support nested constructs such as conditional or iteration statements. This way, it is possible to define islands with lakes and lakes with islands, supporting the idea of recursive island parsing.

3.3 Lake Symbols

In a recent paper from 2021, Okuda and Chiba proposed the use of *lake symbols* for island parsers in an extended Parsing Expression Grammar (PEG). [16] They used the term *alternative symbols* to describe the terminal or nonterminal symbols that indicate the end of a lake and the beginning of the rest of the island.

Lake symbols can be used as a simple wildcard-like symbol in the extended PEG and automate the enumeration of the alternative symbols for the water inside an island. Without this automated enumeration, correctly selecting the right set of alternative symbols is a burden to developers implementing an recursive island parser, as this is not an easy task to do. The paper proposes an algorithm for translating the extended PEG to a normal PEG, which can be given to an existing parser generator based on PEG.

Of all the presented existing solutions, this is the only one that allows users to design parsers that can perform recursive island parsing and automatically determine the alternative symbols. PEGs are closely related to the family of top-down parsing languages, [3] which also includes recursive descent parsers. However, the proposed solution is algorithmic and not suitable for direct use with monadic parser combinators.

3.4 Bounded Seas

In 2015, Kurs et al. proposed *bounded seas*, [11] which can be used to automatically calculate alternative symbols and use them for the not predicate, similar to how lake symbols function. However, the bounded seas calculate only a subset of all the alternative symbols, which corresponds to the SUCCEED set used in the algorithm proposed by Okuda and Chiba. [16]

Thus, its applicability is limited and the solution does not fully satisfy our requirements.

3.5 Formal Foundations for Semi-Parsing

In 2014, Zaytsev compiled what is probably the most comprehensive list of semi-parsing techniques known today. [19] Both island- and lake-grammars appear on the listing in this paper, which boasts a total of 22 existing methods of semi-parsing. However, none of these other solutions can perform something similar to recursive island parsing, which is one of our main requirements. The most promising ideas include:

- *Fuzzy Parsing*, [10] which involves parsing triggered by anchor terminals encountered during the scanning phase. In the framework proposed by Koppler, the user must manually implement the scanner.
- *Skeleton Grammars*, [9] which introduces default rules for water in a baseline complete grammar. The parsers derived from skeleton grammars could be considered island parsers. However, the proposal by Klusener and Lämmel does not include support for lakes.
- *Agile Parsing*, [2] in which additional rules are included to connect several grammars. This approach could be considered as a variant of island parsing when using the not operator in TXL [1] to manually specify where a lake ends and an island starts again.

4 NEW ARCHITECTURE

In this section, we will propose a new architecture, defining monadic parser combinators for the purpose of recursive island parsing. A Git repository containing a functioning implementation has been made available.[1]

4.1 Defining Primitive Monadic Parser Combinators

We are going to start off by defining our primitive monadic parser combinators, which we can later extend with combinators for the purpose of recursive island parsing. From the works of Hutton and Meijer, we have learned that we can define our `Parser` as seen below. [8]

```
type Parser = StateT String Maybe

runParser :: Parser a
          -> String
          -> Maybe (a, String)
runParser = runStateT
```

The definition of `Parser` at this point is just a simple type synonym for the `StateT` monad transformer, carrying a state of type `String` and an inner monad `Maybe`. Monad transformers are type constructors which take a monad as an argument and return a monad as a result. The **mtl** package defines several useful monad transformers besides `StateT`, some of which we will make use of in later sections. [6]

[1] https://github.com/ghuisma/lake-parser-combinators

```
data Maybe a = Nothing
             | Just a

newtype StateT s m a = StateT
    {
    -- unwrap a state monad computation
    -- as a function
    runStateT :: StateT s m a
              -> s
              -> m (a, s)
    }
```

By defining `Parser` using the `StateT` monad transformer, which is already an instance of several important type classes, including `Functor`, `Applicative`, `Alternative`, `Monad` and `MonadState`, we gain access to a number of powerful functions and parser combinators.

The `Functor` type class defines the function `fmap`, which applies a function of type `(a -> b)` to a value of type `f a` where `f` is a functor, to produce a value of type `f b`. [15]

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

The `Applicative` type class describes a structure intermediate between a functor and a monad. It defines operations to embed pure expressions (`pure`), and sequence computations and combine their results (`<*>`) [13]

```
class Functor f => Applicative f where
    -- lift a value
    pure :: a -> f a
    -- sequential application
    (<*>) :: f (a -> b) -> f a -> f b
    -- sequence actions, discarding the
    -- value of the first argument
    (*>) :: f a -> f b -> f b
    -- sequence actions, discarding the
    -- value of the second argument
    (<*) :: f a -> f b -> f a
```

The `Alternative` type class describes a monoid on applicative functors. A monoid is a set equipped with an associative binary operation (`<|>`) and an identity element (`empty`). [13] It provides our `Parser` type with a combinator for prioritised choice (`<|>`) and failure (`empty`).

```
class Applicative f => Alternative f where
    -- The identity of <|>
    empty :: f a
    -- Prioritised choice
    (<|>) :: f a -> f a -> f a
    -- One or more
```

```
    some :: f a -> f [a]
    -- Zero or more
    many :: f a -> f [a]
```

The Monad type class defines the basic operations over a monad, a concept from a branch of mathematics known as category theory. [15] These include the binding operation (>>=), which sequentially composes two actions, passing any value produced by the first as an argument to the second, and return, which injects a value into the monadic type, and should give the same result as pure from the Applicative type class.

```
class Applicative m => Monad m where
    -- inject a value into the monadic type
    return :: a -> m a
    -- sequentially compose two actions
    (>>=) :: forall a b .
             m a -> (a -> m b) -> m b
```

Finally, we reached the MonadState type class, which defines the actions we can perform on the state monad. get returns the state from the internals of the monad. In the case of our Parser type, it Maybe Just returns a String containing the remaining input characters we still have to parse, or Nothing, in which case somewhere failure occurred. The put function can be used to replace or update the current state inside the monad. [6]

```
class Monad m => MonadState s m | m -> s where
    -- return state from the internals
    -- of the monad
    get :: m s
    -- replace the state inside the monad
    put :: s -> m ()
```

Because of our use of StateT, the only primitive parser combinator we still need to define is item, which always succeeds, as long as there is input left, and parses one character from the input.

```
item :: Parser Char
item = do
    (x:xs) <- get
    put xs
    return x
```

By carefully analysing previous works on semi-parsing techniques, we realised there are two key ingredients that are necessary in order to support recursive island parsing. The first is prioritised choice, which would allow us to attempt parsing islands first and if all parser combinators fail, parse water. We already have the prioritised choice combinator available (<|>), as our Parser is just a type synonym for StateT, which in turn instantiates the Alternative type class. The second key ingredient is the ability to perform negative lookahead. In order to define our water parser combinator, we need to be able to look ahead and determine that the character we are trying to parse is not equal to an alternative symbol.

We took the idea of a notFollowedBy parser combinator from the Parsec library. [12] We can define notFollowedBy such that it takes as input a parser we want to check is not ahead. We can run the input parser and let notFollowedBy succeed if the provided parser fails. Vice versa, if the supplied parser succeeds, notFollowedby fails.

```
notFollowedBy :: Parser a -> Parser ()
notFollowedBy p = do
    xs <- get
    case runParser p xs of
        Just _ -> empty
        Nothing -> return ()
```

Using the newly defined item and notFollowedBy parsers, it is also possible to define a parser combinator to verify if we reached the end of a file.

```
eof :: Parser ()
eof = notFollowedBy item
```

The works of Hutton and Meijer [8] also describe a range of other potentially useful parser combinators. And, once more inspired by the Parsec library [12], we also built a helper module to parse lexical elements (tokens) using our previously defined Parser and primitive parser combinators.[2] Using this module, it is possible to create a language definition describing the structure of comments and identifiers in our target language, and generate several important token parsers, including symbol, identifier and whiteSpace. We used this module to create a lexer for C-style languages.[3] Finally, we have all the necessary ingredients to write recursive island parsers by hand.[4]

4.2 Recursive Island Parser Combinators

By carefully analysing our handwritten recursive island parser, we quickly find a recurring pattern in our implementation. We first try to parse the islands that might occur at that specific point in the execution and if none are found, we parse water. This means that we first perform negative lookahead on our manually selected set of alternative symbols for that specific point in the execution and if none are matched, we just parse the current character of the input stream and discard it as water. First, let us define an abstract data type for differentiating between islands and water.

[2] https://github.com/ghuisma/lake-parser-combinators/tree/main/src/V1/Parser
[3] https://github.com/ghuisma/lake-parser-combinators/blob/main/src/V1/JavaScriptLexer.hs
[4] https://github.com/ghuisma/lake-parser-combinators/blob/main/src/V1/TrivialLakeParser.hs

```
data IslandOrWater a = Island a
                     | Water
```

We can use a `Set` to keep track of the alternative symbols at a certain point in the execution of the recursive island parser.

```
-- alternative symbols
type Alt = Char
-- set of current alternative symbols
type AltSet = Set Alt
```

We can now define our water combinator, which receives a set of alternative symbols. By folding over the alternative symbols in the set and performing negative lookahead on them, we ensure that the current character does not match the ending of an island. If it does, the water parser fails. If it does not, we just parse the current character and discard it as `Water`.

```
water :: AltSet -> Parser (IslandOrWater a)
water s = do
    foldl (\acc alt -> acc *>
              (notFollowedBy . char) alt
          ) (return ()) s
    item
    return Water
```

Our island combinator can be defined as a function that receives the alternative symbol of that particular island, a function that takes an `AltSet` and produces a `Parser`, and the current `AltSet`. We first add the provided alternative symbol to our `AltSet`. Then we can pass the updated `AltSet` to the second argument and produce a `Parser` that takes into account the updated `AltSet`. Finally, we map `Island` over the resulting parser to indicate that we care about the result of executing this parser.

```
island :: Alt
       -> (AltSet -> Parser a)
       -> AltSet
       -> Parser (IslandOrWater a)
island alt f s1 = fmap Island (f s2)
    where s2 = (Set.insert alt s1)
```

It is now possible to define our lake parser combinator as a function that receives a list of islands and an `AltSet`. We fold over the list of island parsers, providing them with the current `AltSet`. The starting accumulator is `empty`, which is a parser that always fails. Because we accumulate using prioritised choice (`<|>`), if no islands are provided, we just parse `water`. If islands are provided, we first try to parse the islands and if all island parsers fail, we just parse `water`.

```
lake :: [ AltSet
          -> Parser (IslandOrWater a)
        ]
     -> AltSet
     -> Parser [IslandOrWater a]
lake fs s = many (islands <|> water s)
    where islands = foldl (\acc f -> acc
                                  <|> f s
                          ) empty fs
```

Now, it is also possible to define an empty lake, which just receives the current AltSet and calls lake with no islands provided.

```
emptyLake :: AltSet
          -> Parser [IslandOrWater a]
emptyLake = lake []
```

We can now use these `island` and `lake` parser combinators to redefine our handwritten implementation.[5] Not only did we drastically reduce the number of lines it takes to define the same recursive island parser, but it also becomes significantly easier to select the right alternative symbols for our islands. Thereby making the process of designing recursive island parsers a much less error-prone task.

4.3 Hiding the Set of Alternative Symbols

One downside of our previous implementation is that we need to keep passing our `AltSet` around to all of our parsers. However, we can also use the `ReaderT` monad transformer to provide our parsers with a read-only environment, containing the current `AltSet`.

```
newtype ReaderT r m a = ReaderT
    {
    runReaderT :: ReaderT r m a
               -> r
               -> m a
    }
```

`ReaderT` is an instance of `MonadReader`, which permits us to ask for the current environment containing our `AltSet`, and to update the local environment by adding alternative symbols to our `AltSet`. [6]

```
class Monad m => MonadReader r m | m -> r where
    -- retrieve the monad environment
    ask :: m r
    -- execute a computation in the
    -- modified environment
    local :: (r -> r) -> m a -> m a
```

[5] https://github.com/ghuisma/lake-parser-combinators/blob/main/src/V2/TrivialLakeParser.hs

Both `ReaderT` and `StateT` are instances of the `MonadReader` and `MonadState` type classes. This allows us to stack both monad transformers together and retain access to the functionality described in the definition of both type classes. [6] By utilising the `GeneralizedNewtypeDeriving` pragma [17], we can redefine `Parser` and hide the implementation details from the end-user.

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype Parser a = Parser
    { getParser :: ReaderT AltSet
                   (StateT Input Maybe) a
    } deriving ( Monad
               , Applicative
               , Functor
               , Alternative
               , MonadFail
               , MonadState Input
               , MonadReader AltSet
               )

runParser :: Parser a
          -> String
          -> Maybe (a, String)
runParser p = runStateT
    ( runReaderT ( getParser p ) Set.empty )
```

Now, it is possible to redefine our lake combinators using our redefined `Parser`. The `water` parser asks the local reader environment for the current `AltSet`. The `lake` combinator directly receives a parser for all the islands it can contain. And, the `island` parser updates the local environment of the provided parser, by inserting the provided alternative symbol into the current `AltSet`. We can now redefine our previous implementations of recursive island parsers by removing `AltSet`, as this is now part of our `Parser` definition. [6]

```
emptyLake :: Parser [IslandOrWater a]
emptyLake = lake empty

lake :: Parser (IslandOrWater a)
        -> Parser [IslandOrWater a]
lake islands = many (islands <|> water)

water :: Parser (IslandOrWater a)
water = do
    s <- ask
    foldl (\acc alt -> acc *>
              (notFollowedBy . char) alt
          ) (return ()) s
    item
    return Water
```

[6] https://github.com/ghuisma/lake-parser-combinators/blob/main/src/V3/TrivialLakeParser.hs

```
island :: Alt
       -> Parser a
       -> Parser (IslandOrWater a)
island a = (fmap Island) . local (Set.insert a)
```

4.4 Determining Alternative Symbols

While our current implementation certainly makes it easier to select the appropriate alternative symbols, we still do not automatically determine them. When analysing our implementations, we quickly see that the alternative symbol we provide to `island` is the same as the last character provided to the `symbol` token parser. One idea would be to use the `Writer` monad to accumulate this final character and add it to the current `AltSet`.

```
newtype WriterT w m a = WriterT
    {
    runWriterT :: Writer w m a
               -> m (a, w)
    }

type Writer w = WriterT w Identity
runWriter :: Writer w a -> (a, w)
```

`Writer` is an instance of `MonadWriter`, which defines `tell`, a function that allows us to tell `w`, an action that produces an output `w`. [6] The definition of `MonadWriter` specifies that `w` must be a `Monoid`, which in turn specifies that it must be a `Semigroup`. A `Monoid` instance specifies an associative operation and the identity of this operation, while a `Semigroup` instance only specifies an associative operation. [5]

```
class Semigroup a where
    -- an associative operation
    (<>) :: a -> a -> a

class Semigroup a => Monoid a where
    -- identity of mappend
    mempty :: a
    -- an associative operation
    mappend :: a -> a -> a

class (Monoid w, Monad m)
      => MonadWriter w m | m -> w where
    tell :: w -> m ()
```

We can redefine our `Alt` type synonym as a `newtype` and create an instance of `Semigroup` where the associative binary operation only retains the last character. For example, `Just (Alt 'a') <> Just (Alt 'b') == Just (Alt 'b')`

```haskell
newtype Alt = Alt Char

instance Semigroup Alt where
    _ <> y = y
```

We can use the `Monoid` instance of `Maybe` to provide us with `mempty`, which is `Nothing`.

```haskell
instance Semigroup a
         => Monoid (Maybe a) where
    mempty :: Maybe a
    mappend :: Maybe a -> Maybe a -> Maybe a
```

As before, `ReaderT`, `StateT` and `Writer` are all instances of the `MonadState`, `MonadReader`, and `MonadWriter` type classes. This allows us to stack these monad transformers together and retain access to the functionality described in the definition of the type classes. [6]

We can use the `MaybeT` monad transformer to insert `Writer` at the bottom of our `Parser` monad stack. [4] Now we can have both the result of parsing the input string as well as the last character parsed as our output.

```haskell
newtype MaybeT m a = MaybeT
    {
    runMaybeT :: m (Maybe a)
    }

newtype Parser a = Parser
    { getParser :: ReaderT AltSet (
                   StateT Input (
                   MaybeT (
                   Writer (Maybe Alt)))) a
    } deriving ( Monad
               , Applicative
               , Functor
               , Alternative
               , MonadFail
               , MonadState Input
               , MonadReader AltSet
               )

runParser :: Parser a
          -> String
          -> Maybe (a, String)
runParser p input = fst
    $ runWriter
    $ runMaybeT
    $ runStateT (
        runReaderT (getParser p) Set.empty
    ) input

accAlt :: Parser a
       -> String
       -> (Maybe Alt)
accAlt p input = snd
    $ runWriter
    $ runMaybeT
    $ runStateT (
        runReaderT (getParser p) Set.empty
    ) input
```

We now have two execution modes, parsing (`runParser`) and accumulating alternative symbols (`accAlt`).

The `sat` parser combinator takes a predicate and if the parsed character satisfies the predicate, successfully returns the result. It stands at the basis of our character and token parsers.[7]

```haskell
sat :: (Char -> Bool) -> Parser Char
sat p = do
    x <- item
    if p x then return x else empty
```

We now want to `tell` the last character that satisfies the given predicate. This way when calling `accAlt` on a `Parser`, we get the final character that could be parsed by the provided parser.

```haskell
sat :: (Char -> Bool) -> Parser Char
sat p = do
    x <- item
    if p x
       then do
           tell (Just (Alt x))
           return x
       else empty
```

Now we can modify `island` such that it no longer takes an alternative symbol as input, but executes the provided parser using `accAlt` and inserts the result of this execution into the `AltSet` in the read-only environment of the provided parser.

```haskell
island :: Parser a
       -> Parser (IslandOrWater a)
island p = fmap Island (do
    xs <- get
    case accAlt p xs of
        Nothing -> p
        Just alt ->
            local (Set.insert alt) p
    )
```

This approach comes with one obstacle. `sat` does not discard the character it evaluated against the provided predicate

[7] https://github.com/ghuisma/lake-parser-combinators/tree/main/src/V1/Parser

as water in case of failure. When parsing this is the desired behaviour. However, when accumulating the alternative symbol of an island, if the predicate is not met, we want to discard it as water and attempt to match the next character.

One way to achieve this behaviour is by adding a boolean flag (`accAltFlag`) to the read-only environment and modifying `sat` to keep consuming characters from the input by applying itself recursively. We can now run our parsers and specify, by setting this flag, that we are interested in either parsing or accumulating the alternative symbol of an island.

```
sat :: (Char -> Bool) -> Parser Char
sat p = do
    accAltFlag <- askAccAltFlag
    x <- item
    if accAltFlag
        then do
            if p x
                then do
                    tell (Just (Alt x))
                    return x
                else do
                    tell Nothing
                    sat p
        else if p x
            then return x
            else empty
```

When accumulating the alternative symbol of a parser, we need to make sure the whitespace parsers always have the `accAltFlag` set to `False`. Otherwise they will keep trying to accumulate whitespace or comments as the alternative symbol.

Furthermore, the `island` and `lake` combinators should just be skipped over, that is, return successfully without parsing any input, because we are only interested in determining the alternative symbol of the outermost island and not any of the islands that may occur within it.

We encourage the reader of this document to take a closer look at the final implementation using the `Writer` monad, in the provided Git repository.[8]

5 VERIFICATION

In this section we will attempt to verify if the requirements set in section 2 were met. Because of our use of Haskell, which is a functional programming language, **R1** was obviously met. The same holds for **R4**, as we used monad and monad transformers to define the `Parser` type used by our set of lake parser combinators. What remains to be done is to verify the correctness of our solutions using manual selection of alternative symbols (**R2**), and our final solution, which attempts to automatically determine the alternative symbols of an island (**R3**).

For this purpose, we created an implementation of a recursive island parser for each of our proposed solutions. These implementations only recognize function definitions and function calls in JavaScript programs, which is all we need to construct a call graph of the input program. See below for a trivial example of such a program.

```
function a (a) {
    return a(a);
}
```

When running our implementations on the program defined above, we would expect to see a result with a general structure as follows.

```
[ FunDef "a" (Block [FunCall "a"]) ]
```

We ran our implementations against a number of such hand-written input programs and manually verified the correctness of the data structures resulting from the execution of the implementations given the input programs. From these tests, we can with reasonable certainty conclude that we satisfied **R2**.

However, despite the fact that our final implementation, which uses the `Writer` monad to accumulate the alternative symbols of islands, did pass our tests, we surmise that it is probably possible to construct a recursive island parser that gives incorrect results for the input programs provided. More extensive testing is required to somewhat guarantee our final solution works as expected under all various circumstances that may be expected when designing recursive island parsers. Therefore, we only partially satisfy **R3**.

6 CONCLUSION

We have shown it possible to design a set of lake parser combinators for recursive island parsing. We have hidden the implementation details of our work using generalised `newtype` deriving and monad transformers. We proposed a solution using the `Writer` monad to accumulate the alternative symbols of islands, removing this burden from the developers of recursive island parsers.

We suggest further work is done on the verification of the final proposed solution, potentially refining or redefining parts of it based on the results of the additional tests performed. Furthermore, it remains to be shown if it is possible to improve the space and time complexity of the proposed solution and if the proposed solution is performant enough to be used in real-life applications.

[8] https://github.com/ghuisma/lake-parser-combinators/tree/main/src/V4

REFERENCES

[1] J. R. Cordy, C. D. Halpern, and E. Promislow. 1991. TXL: A Rapid Prototyping System for Programming Language Dialects. *Computer Languages* 16, 1 (1991), 97–107.

[2] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider. 2003. Agile Parsing in TXL. *Journal of Automated Software Engineering* 10, 4 (2003), 311–336.

[3] B. Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '04)* (Jan 2004), 111–122.

[4] Y. Gale and E. Kidd. 2007. transformers-0.6.0.4: Concrete functor and monad transformers. https://hackage.haskell.org/package/transformers-0.6.0.4/docs/Control-Monad-Trans-Maybe.html.

[5] A. Gill. 2001. Data.Monoid. https://hackage.haskell.org/package/base-4.16.1.0/docs/Data-Monoid.html.

[6] A. Gill and E. Kmett. 2018. mtl: Monad classes, using functional dependencies. https://hackage.haskell.org/package/mtl.

[7] J. Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (April 1989), 98–107. https://doi.org/10.1093/comjnl/32.2.98 arXiv:https://academic.oup.com/comjnl/article-pdf/32/2/98/1445644/320098.pdf

[8] G. Hutton and E. Meijer. 1996. Monadic Parser Combinators. http://www.cs.nott.ac.uk/~pszgmh/monparsing.pdf.

[9] S. Klusener and R. Lämmel. 2003. Deriving Tolerant Grammars from a Base-line Grammar. *Proceedings of the 19th IEEE International Conference on Software Maintenance (ICSM 2003)* (September 2003), 179–188.

[10] R. Koppler. 1997. A Systematic Approach to Fuzzy Parsing. *Software: Practice & Experience* 27, 6 (1997), 637–649.

[11] J. Kurs, M. Lungu, R. Iyadurai, and O. Nierstrasz. 2015. Bounded Seas. *Computer Languages, Systems and Structures* 44, A (2015), 114–140.

[12] D. Leijen. 2001. Parsec: a fast combinator parser. http://www.cs.nott.ac.uk/~pszgmh/monparsing.pdf.

[13] C. McBride and R. Paterson. 2005. Control.Applicative. https://hackage.haskell.org/package/base-4.16.1.0/docs/Control-Applicative.html.

[14] L. Moonen. 2001. Generating Robust Parsers using Island Grammars. *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)* (October 2001), 13–22.

[15] The University of Glasgow. 2001. Control.Monad. https://hackage.haskell.org/package/base-4.16.1.0/docs/Control-Monad.html.

[16] K. Okuda and S. Chiba. 2021. Lake Symbols for Island Parsing. *The Art, Science, and Engineering of Programming* 5, 2 (2021), 11.

[17] GHC Team. 2020. Generalised derived instances for newtypes. https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/newtype_deriving.html.

[18] A. van Deursen and T. Kuipers. 1999. Building Documentation Generators. *Proceedings of International Conference on Software Maintenance (ICSM 1999)* (1999), 40–49.

[19] V. Zaytsev. 2014. Formal foundations for semi-parsing. *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)* (Feb 2014), 313–317.