Spring 5-31-2024

A Survey of Practical Haskell: Parsing, Interpreting, and Testing

Parker Landon
Seattle Pacific University

A Survey of Practical Haskell: Parsing, Interpreting, and Testing

By

Parker Landon

Faculty Mentors

Dr. Carlos R. Arias, Department of Computer Science, Seattle Pacific University

Honors Program Director

Dr. Joshua Tom

A project submitted in partial fulfillment of the requirements
for the Bachelor of Arts degree in Honors Liberal Arts
Seattle Pacific University
2024

Presented at the SPU Honors Research Symposium
May 18, 2024

A Survey of Practical Haskell: Parsing, Interpreting, and Testing

Parker Landon
Seattle Pacific University
Seattle, Washington 98119
landonp@spu.edu

***Abstract*—Strongly typed pure functional programming languages like Haskell have historically been confined to academia as vehicles for programming language research. While features of functional programming have greatly influenced mainstream programming languages, the imperative programming style remains pervasive in practical software development. This paper illustrates the practical utility of Haskell and pure functional programming by exploring "hson," a scripting language for processing JSON developed in Haskell. After introducing the relevant features of Haskell to the unfamiliar reader, this paper reveals how hson leverages functional programming to implement parsing, interpreting, and testing. By showcasing how Haskell's language features enable the creation of expressive, maintainable, and correct code, this paper aims to demonstrate the viability of pure functional programming in real-world software development scenarios.**

I. Introduction

A. Object-Oriented Programming

If you've written code in a contemporary programming language, you're probably familiar with "object-oriented programming." Since its popularity surge among academics and software developers in the 1990s, the object-oriented paradigm has dominated mainstream computer programming, existing as the core of pervasive programming languages like Java and C++ [1]. Object-oriented programming entails encapsulating data within "objects" possessing properties and behaviors defined by the object's "class" [2]. Classes are blueprints for objects: they represent the type, behaviors, and properties that instantiated objects of that class will have. The following `Rectangle` class, for example, defines `_length` and `_width` properties and exposes an `area` method.

```
// The Rectangle Class: a "blueprint" of every Rectangle object
class Rectangle {
  private _length: number;
  private _width: number;

  constructor(length: number, width: number) {
    this._length = length;
    this._width = width;
  }

  area() {
    return this._length * this._width;
  }
}

// myRectangle: an instantiated object of the Rectangle class
const myRectangle = new Rectangle(4, 3);
myRectangle.area(); // 12!
```

A central aspect of the object-oriented paradigm is encoding hierarchical relationships between classes via *inheritance* and *composition* [3]. For example, one could use inheritance to represent the relationship between squares and rectangles (i.e., that squares *are* rectangles) by creating a new `class Square` that *inherits* behaviors and properties from `Rectangle`.

But why undergo this effort of encoding classes, behaviors, properties, and relationships? Much motivation for object-oriented programming stems from the consequences of the *imperative* style of traditional programs.

B. Imperative Programming and Side Effects

Imperative programming is a style in which program behavior is encoded as a sequence of state transitions from an initial state to a final state [4]. Each operation in an imperative program may have the *side effect* of mutating the system's overall state. If intentional or "correct," these side effects incrementally transition the system's state toward the desired final state. However, an unintentional or "incorrect" side effect may cause the system to produce an unexpected result.

A goal of object-oriented programming (i.e., encoding classes, behaviors, properties, relationships, etc.) is to *encapsulate* these state updates by containing them within objects and abstracting them with a defined interface [2, 3]. Interaction with the private `_length` and `_width` properties of the `myRectangle` object, for example, is mediated by the `Rectangle` class and its defined interface (namely, its `constructor` and the `area` method). The `private` access modifiers on the `_length` and `_width` properties prevent any code outside the class definition from mutating those values. This way, state updates are more controlled, and unintended results are more easily identified as occurring within a class instead of "somewhere in the system."

Still, state mutation represents only one kind of side effect. Programs may also

- print text
- read data from a file
- query a database
- execute other programs
- send emails
- launch missiles [4]

Typically, the imperative style allows these side effects to occur anywhere in the program, often without any indication or warning. For example, a program written in TypeScript can perform input/output operations freely, enabling unrestricted interaction with the "outside world."

```
// The `main` function does not indicate that this program
// launches missiles.
function main() {
    console.log("Hello world!");
    foo();
}

function foo() {
    // Indeed, this program launches missiles.
    launchMissiles();
}
```

While the liberty to produce side effects may feel convenient when writing a program, it introduces more possibilities for creating incorrect programs with unexpected results. In general, programming languages make a trade-off between correctness and "convenience." However, an incorrect program is almost always inconvenient, and a language feature that affords convenience during a program's initial implementation may prove inconvenient when refactoring and maintaining that program. The following sections introduce the "functional programming" (or "FP") style, a programming language paradigm emphasizing correctness. As this paper aims to convey, the "restrictions" introduced by this paradigm are liberating in practice as they promote correctness, facilitate tractable program reasoning, and encourage greater developer confidence.

C. Functional Programming and Purity

According to the FP style, a program is a function defined in terms of other functions; programs are compositions of functions [5, 6]. Rather than performing computations as sequences of state transitions, functional computations are carried out by applying functions to arguments. Of course, imperative programming languages often provide the ability to construct and apply functions, too, but the functional programming style distinguishes itself by promoting *purity*.

A function is said to be *pure* if its execution produces no side effects and its output depends on nothing besides its input. In other words, the result of a pure function is wholly determined by its input; it will always produce the same output given the same input. Thus, the behavior of a pure function is captured entirely by its definition: its result neither mutates nor is influenced by any program state [4].

The upshot of purity is that it eliminates the possibility of unwanted side effects by doing away with side effects altogether. While this limitation may seem severe, it affords several benefits that promote correctness. For example, a functional program is easy to reason about. Because a pure function's behavior is captured entirely by its definition, there is never a concern for how a function operates in the broader context of a stateful program. For the same reasons, functional programs tend to be trivial to test. While a test of an impure function might first require arranging the appropriate preconditions (a generally laborious task within an imperative program), testing a pure function simply requires providing it with the expected inputs and asserting about its outputs. Purity also affords several benefits when writing parallel programs, since parallel programming is fundamentally concerned with avoiding unexpected interactions of side effects [7].

Still, it may seem that there exists a fatal problem with the purity restriction: a program composed solely of pure functions is useless. Necessary operations like receiving inputs and displaying outputs are side effects, so they cannot be performed in a pure context. Fortunately, programming languages that enforce purity and the functional style—namely "pure functional programming languages"—offer various methods for integrating and encapsulating side effects. Haskell, the pure functional programming language considered in this work, facilitates effectful programming through *monads*, which encode impure functions and create a boundary distinguishing effectful operations from pure ones [8]. Monads are a part of Haskell's rich *static type-checking system*, another feature like purity that aims to promote correctness. Although it cannot eliminate all unexpected or undesired behaviors, static type checking prevents the programmer from introducing a certain class of errors, such as adding numbers to boolean values [9]. While disagreement exists about the convenience of static type checking and the extent to which it ensures program safety [10, 11], this paper aims to illustrate how Haskell's robust type system promotes correctness while enabling flexibility.

Despite their provided benefits, Haskell and other pure functional programming languages have never reached the popularity of imperative, object-oriented languages like C++ and Java [1]. In 1998, Philip Wadler presented his paper *Why no one uses functional languages*, ascribing this lack of adoption not to ignorance or inferior program performance but to shortcomings of strongly-typed functional programming language implementations and ecosystems at the time [12]. While his concerns with portability, availability, tooling, and libraries were substantial then, they have largely been addressed in functional programming languages like Haskell today.

Yet, strongly typed pure functional programming languages like Haskell still enjoy significantly less *practical application*—for example, in the software engineering industry and commercial world—than their mainstream counterparts [13]. Sure, functional programming has greatly influenced these domains, as features from functional languages like pattern matching, generics, type inference, and first-class functions have been adopted by and become central to the most widely-used programming languages today; however, the functional programming languages themselves have primarily remained tools of researchers and hobbyists [1, 13]. Indeed, the measurable influence of functional features is a testament to the work of these users; though, as Wadler suggested in 1998, there exists a tension between applying a language to building useful systems and using that language to drive programming language research innovations [12].

Despite the relative obscurity of pure functional programming languages and their traditional origins in programming language research, many functional programmers advocate for their fitness in practical settings. Even when pure functional programming languages were radical, slow, and impractical, the legendary John Backus endorsed functional programming as a practical tool in his 1977 Turing Award Lecture *Can programming be liberated from the von Neumann style?* [5, 13].

Thus, this paper seeks to demonstrate how Haskell and strongly typed pure functional programming more broadly are practical tools for creating useful systems. The following section introduces *hson*, a scripting language for processing JSON implemented in Haskell. Section III introduces Haskell and several language features utilized within hson for the uninitiated reader. Finally, sections IV, V, and VI reveal the implementation details of hson, outlining its parser, interpreter, and test suite code, with the primary goal of demonstrating how Haskell and pure functional programming facilitate crafting expressive, maintainable, and correct code.

II. Introducing hson

The hson program is a command-line interface for processing JSON data. Given a script written in the hson language and JSON data, the hson CLI processes the JSON and outputs a result according to the hson script. For example, consider the following JSON data representing a list of restaurants.

```
[
  {
    "name": "Parker's Bar and Grill",
    "city": "Seattle",
    "state": "Washington",
    "rating": 4,
    "price": 1
  },
  {
    "name": "Smashing Sushi",
    "city": "Portland",
    "state": "Oregon",
    "rating": 5,
    "price": 3
  },
  {
    "name": "Barely Barbecue",
    "city": "Seattle",
    "state": "Washington",
    "rating": 1,
    "price": 2
  }
]
```

The following hson script retrieves the names of all Seattle restaurants in the given JSON data.

```
$.filter(|restaurant| =>
  restaurant.city == "Seattle"
).map(|restaurant| =>
  restaurant.name
)
```

When the hson CLI is executed with the provided JSON data and hson script, the JSON data is parsed and bound to the `$` identifier in the hson script. The hson script then *filters* that data for restaurants whose `city` property is `"Seattle"` and *maps* each restaurant object to its `name` property.

If the file **script.hson** contains the above hson script and the file **restaurants.json** contains the JSON data, hson can be run from the command line as follows to produce the desired results.

```
$ hson --hf script.hson --jf restaurants.json
[Parker's Pasta, Barely Barbecue]
```

With an emphasis on readability and familiarity, the syntax of hson was designed to be similar to that of JavaScript. Operations like `filter` and `map` can be chained like methods, and properties of objects are accessed with the dot (`.`) symbol. The construct `|restaurant| => restaurant.name` is an anonymous function that takes a `restaurant` object as its argument and returns the object's `name` property. Functions in hson are first class, so they can be passed as arguments to other higher-order functions like `filter` and `map`.

The above hson script can be modified to format its results as a JSON string with the `toJSON` function. The pipe operator `|>` can also be employed instead of the dot symbol to compose functions, and each `restaurant` object can be mapped to a new object with a single `name` property.

```
$ |> filter(|restaurant| => restaurant.city == "Seattle")
  |> map(|restaurant| => {name: restaurant.name})
  |> toJSON(2)
```

The pipe operator passes its left-hand side as the first argument to the function call on its right-hand side. The argument of `2` in `toJSON()` specifies an indentation of `2` in the output string. Rerunning hson with the newly modified script produces the following output.

```
$ hson --hf script.hson --jf restaurants.json
[
  {
    "name": "Parker's Pasta"
  },
  {
    "name": "Barely Barbecue"
  }
]
```

Variables in hson are declared with the `let` keyword. For example, in the following script, a function `filterSeattleRestaurants` is declared that applies the `filter` and `map` operations from above. Applying that function to the parsed JSON data (bound to `$`) produces the same output as before.

```
let filterSeattleRestaurants = |restaurants| =>
  restaurants
    |> filter(|restaurant| => restaurant.city == "Seattle")
    |> map(|restaurant| => restaurant.name);
filterSeattleRestaurants($)
```

The hson language supports all JSON data types—arrays, objects, strings, numbers, booleans, and `null`—as well as functions. The semantics of these data types and the syntax

of operations on their values are nearly identical to those of JavaScript. For example, arrays are indexed with the standard square bracket notation and, as previously demonstrated, object properties are accessed with the dot symbol. The following snippet accesses the `restaurant` object at index `1` in the input JSON and prints out its `name` property.

```
let restaurant = $[1];
restaurant.name
```

```
$ hson --hf script.hson --jf restaurants.json
Smashing Sushi
```

Like JavaScript, hson also provides array and object *destructuring* for accessing array indices and object properties. The following script produces the same result as above ("Smashing Sushi") but instead utilizes destructuring for index and property access.

```
let [_, restaurant] = $;
let { name } = restaurant;
name
```

As a more sophisticated example, consider the following JSON representation of a Turing Machine, where the `start`, `accept`, and `reject` keys define the start, stop, and reject states of the machine respectively, and the `delta` key contains information about each state transition.

```
{
  "start": "1",
  "accept": "accept",
  "reject": "reject",
  "delta": [
    {
      "from": "1",
      "to": [
        {
          "result": ["reject", "_", "R"],
          "on": "_"
        },
        {
          "result": ["reject", "x", "R"],
          "on": "x"
        },
        {
          "result": ["2", "_", "R"],
          "on": "0"
        }
      ]
    },
    {
      "from": "2",
      "to": [
        ...
      ]
    },
    ...
  ]
}
```

The following hson script counts the number of transitions that result in the reject state.

```
$.delta.reduce(|accumulator, transitions| =>
  transitions.to.some(|transition| =>
    transition.result[0] == $.reject) ?
      accumulator + 1 : accumulator
  , 0)
```

The `reduce` function is inspired by Haskell's `foldr` operation and is nearly identical to the `reduce` array method from JavaScript. Its responsibility in the script above is to keep track of the transition count. The `some` function also has a near-identical analog in JavaScript: it returns true if any element in the provided array satisfies its predicate function. In the script above, `some` returns true if any transition from a given state is the reject state. Finally, the ternary operator `?` conditionally returns the `accumulator` value incremented by one if the transition result is the reject state.

The logic of conditionally tallying list elements can be abstracted to a higher-order function `countWhere`, which will again utilize the `reduce` operation and the ternary operator to count the elements that satisfy a given predicate function.

```
let countWhere = |list, predicate| =>
  list.reduce(|accumulator, element| =>
    predicate(element) ? accumulator + 1 : accumulator
  , 0);
```

The original counting script can now be rewritten to employ `countWhere`, with the provided predicate being the `some` function from before.

```
$.delta |>
  countWhere(|delta| =>
    delta.to.some(|transition| =>
      transition.result[0] == $.reject
    )
  )
```

At a high level, an hson script is a sequence of zero or more variable declarations followed by a single expression. The output of a script is the result of the evaluated final expression. Variables in hson are immutable: they cannot be reassigned after their declaration, and their values cannot change. Values in hson are computed solely from compositions of functions, so hson is itself a functional language.

The hson program is responsible for

- reading the hson, JSON, and command line options
- parsing command-line options
- parsing the input hson script
- parsing the input JSON, converting each value to an hson value and binding the root value to `$`
- interpreting the input script
- reporting any syntax or runtime errors that occur

The hson codebase also employs *property-based testing*, which helps ensure the correctness of the hson parser by running it on thousands of randomly generated input programs [14].

Sections IV, V, and VI of this paper reveal the implementation details of parsing, interpreting, and testing within hson. In turn, these sections illustrate how each hson feature is achieved

through Haskell code and demonstrate how the development of hson has benefited from utilizing strongly typed pure functional programming. First, a brief, applied survey of the Haskell programming language is provided in the following section for the uninitiated reader, introducing many language features leveraged within hson.

III. Haskell Background

A. Haskell Types and Functions

Haskell is a statically typed programming language, meaning that the type of each value is known at compile time and that programs with type errors will fail to compile. In Haskell, every value has an associated type. The type of a value is declared explicitly with the `::` symbol.

```
number :: Int
listOfNumbers :: [Int]
numberTuple :: (Int, Int)
```

Functions are values, too, and therefore also have types. The `->` symbol denotes the type of a function mapping.

```
length :: [Int] -> Int
```

The statement above declares that the function `length` maps a list of integers `[Int]` to a single integer `Int`. The semantics of `length` is intuitive: given a list of integers, return the number of elements. However, this type definition for `length` is inflexible: it cannot be applied to a list of characters, for example. Fortunately, Haskell provides *polymorphic* types, enabling type declarations to include *type variables* that can instantiate to any type [4]. For example, the type `[a]` includes the type variable `a` and denotes a homogeneous list containing elements of any type. Utilizing polymorphic types, the type of `length` can be rewritten to support lists of any type.

```
length :: [a] -> Int
```

A function `f` is applied to an argument `a` with the syntax `f a`. Applying the `length` function to a list `list :: [Int]` computes the number of elements `list` contains.

```
list :: [Int]
list = [1,2,3]


length list -- 3
```

What about defining functions with two or more parameters? For example, how could a trivial function `add` be defined to return the `Int` sum of two `Int` arguments? A first pass at the type declaration of `add` might group the two `Int` inputs into a single tuple parameter.

```
add :: (Int, Int) -> Int
add (x, y) = x + y
```

The above definition of `add` maps `(Int, Int)` tuples to `Int` results. Applying `add` to the values `1` and `2` entails providing `add` with the tuple `(1, 2)`.

```
add (1, 2) -- 3
```

Now, consider the following new definition of `add`:

```
add :: Int -> Int -> Int
add x y = x + y
```

The `->` symbol is right-associative, so `Int -> Int -> Int` implicitly means `Int -> (Int -> Int)` and can be read as "a function that maps an `Int` to a function that maps an `Int` to an `Int`." This style of writing a function with two or more parameters as a function mapping a single argument to a function with a single parameter is called *currying* [13].

With the newly curried type definition, the syntax `add 1 2` denotes the application of `add` to the arguments `1` and `2`. Function application is left-associative, so `add 1 2` is equivalent to `(add 1) 2`. Then, `(add 1)` could be extracted into a new function `increment`, and the expression `increment 2` would be equivalent to `(add 1) 2`.

```
increment :: Int -> Int
increment = add 1


increment 2 -- this is the same as (add 1) 2!
```

Currying enables the simple construction of specific operations from general ones, making functions more extensible and robust. Thus, currying is the default style for function definitions in Haskell.

So far, every function definition has been an equation at the top level (e.g., `add x y = x + y`). Alternatively, functions can be defined with lambda expressions, which have arguments and a body like any other function but do not have a name. For example, `add` can be redefined as a lambda expression.

```
add :: Int -> Int -> Int
add = \x y -> x + y
```

Like the original definition of `add`, this new definition takes two arguments, `x` and `y`, and returns their sum. Lambda expressions are commonly used to define arguments for higher-order functions.

B. Typeclasses

The current definition of `add` can only be applied to *integer* arguments. It would be more convenient if `add` supported other kinds of numeric values, like decimals. One attempt at a more reusable `add` function might utilize polymorphic types in its definition to support all parameter types.

```
add :: a -> a -> a
add x y = x + y
```

However, this code won't compile because the `+` operator is not defined for all types: the Haskell compiler wouldn't know how to perform `add True False`, for example. Thus, `add` must only support the *class* of types for which `+` is defined. Indeed, Haskell's *typeclass* feature enables overloading operations for different types and categorizing types according to the operations they support.

A `class` declaration describes a new typeclass according to its operations and their type signatures. A simple typeclass is the built-in `Eq` typeclass; its definition indicates that a type `a` is an instance of `Eq` if it defines the equality operator `==` with type `a -> a -> Bool` [15].

```
class Eq a where
  (==) :: a -> a -> Bool
```

The `add` function can be extended with the `Num` typeclass, which generalizes basic numeric operations like `+`.

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

Now, the `Num` typeclass can serve as the *context* of `a` in the type declaration of `add`, constraining the argument types of `add` to those that define addition with the `+` operator [15].

```
add :: (Num a) => a -> a -> a
add a b = a + b
```

C. Defining Types

To introduce a new numeric type, like a type `Complex` to represent complex numbers, the `data` keyword is used. Before defining `Complex`, consider the following type declaration for `Bool`.

```
data Bool = True | False
```

The declaration introduces the type constructor `Bool` alongside the values (or *data constructors*) it comprises (`True` and `False`) [16]. Having provided its `data` declaration, `Bool` can now serve as a type, and its values `True` and `False` can be used wherever a `Bool` value is expected.

```
not :: Bool -> Bool
not True = False
not False = True
```

The above definition of `not` utilizes *pattern matching* over the values of a type. As demonstrated, functions in Haskell can be redefined for different input patterns. When a function is applied, the pattern on the left-hand side of its equation (e.g., `True` in `not True = False`) is matched against the argument. If the argument aligns with the pattern, the function evaluates and returns the right-hand side of the equation. Otherwise, the argument is matched against the pattern in the next equation, and the process continues until a pattern is matched or all patterns have been visited, in which case an error occurs [17].

```
result :: Bool
result = not True -- False!
```

The type constructor and corresponding data constructors for `Bool` are nullary: they take no arguments. However, when considering a new type `Complex`, one might expect its data constructor to take two arguments, the first representing the real part and the second representing the imaginary part. Indeed, data constructors can be parameterized; after all, they are *functions* that produce results of their corresponding type [17]. For example, the following data declaration for `Complex` introduces a new type `Complex` and a corresponding data constructor `Comp` that takes two `Float` arguments.

```
data Complex = Comp Float Float
```

Having defined `Complex` with `data`, new functions on `Complex` values can be introduced, and the `Comp` constructor can be used wherever a value of type `Complex` is expected.

```
myComplexNumber :: Complex
myComplexNumber = Comp 3 4

magnitude :: Complex -> Float
magnitude (Comp x y) = sqrt (x^2 + y^2)

result :: Float
result = magnitude myComplexNumber -- 5.0
```

Like value constructors, the type constructors in a data declaration can be parameterized to produce polymorphic types [16]. A notable example of a polymorphic type is `Maybe`, which is declared as follows:

```
data Maybe a = Just a | Nothing
```

The `Maybe` *type constructor* can be applied to any other type `t` to produce a new type, `Maybe t` [16]. That new type has two *value constructors*, `Just` and `Nothing`, which could represent success and failure, respectively [17]. For example, a function `unlock` could return a `Maybe String` value that contains a message if the provided `key` is correct and `Nothing` otherwise.

```
unlock :: String -> Maybe String
unlock key = if key == "kitten" then Just "Meow!" else Nothing

test1 = unlock "puppy" -- Nothing
test2 = unlock "kitten" -- Just "Meow!"
```

D. Declaring Instances

The new `Complex` type representing complex numbers can be declared as an instance of the `Num` typeclass. The following *instance* declaration provides for the `Complex` type an appropriate definition of each operation in the `Num` typeclass, thus declaring that the `Complex` type belongs to `Num` [15].

```
instance Num Complex where
  (Comp x1 y1) + (Comp x2 y2) = Comp (x1 + x2) (y1 + y2)
  (Comp x1 y1) - (Comp x2 y2) = Comp (x1 - x2) (y1 - y2)
  (Comp x1 y1) * (Comp x2 y2) =
    Comp (x1 * x2 - y1 * y2) (x1 * y2 + x2 * y1)
  negate (Comp x y) = Comp (negate x) (negate y)
  abs z = Comp (magnitude z) 0
  signum (Comp 0 0) = 0
  signum z@(Comp x y) = Comp (x / r) (y / r)
   where
    r = magnitude z
  fromInteger n = Comp (fromInteger n) 0
```

The `Num` instance declaration for `Complex` enables `Complex` values to be used wherever instances of `Num` are expected.

```
add :: (Num a) => a -> a -> a
...
add (Comp 3 4) (Comp 5 6) -- Comp 8.0 10.0
```

E. Built-in Types

So far, we've considered several functions, types, and typeclasses that are built into Haskell. These base utilities

are provided by the *standard prelude*, which is a library file that all Haskell modules import by default [16].

Two built-in parameterized types—types that include type variables—are list (`[]`) and `Maybe`. The list type, denoted `[a]` or `[] a`, represents a sequence of elements of the same type `a`. The `Maybe` type, `Maybe a`, represents a result that either fails and is `Nothing` or succeeds and contains a value of type `a` within the `Just` constructor [16].

Another useful parameterized type is `Either a b`, which is defined as follows.

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

Similar to the `Maybe` type, `Either` encodes two possibilities; however, unlike `Maybe`, both values of `Either` carry some value. `Either String Int`, for example, represents values that contain a `String` (within `Left`) and values that contain an `Int` (within `Right`).

A unique but critical example of a parameterized type is `IO a`. As mentioned, Haskell's type system enforces purity by distinguishing effectful operations from pure ones. This boundary is constructed primarily by the `IO` type, which represents values whose computation may have demanded the production of input/output side effects [13]. Consider, for example, the type of `getChar`, which reads a character from `stdin`.

```
getChar :: IO Char
```

Intuitively, `getChar` produces a `Char` result. However, because it entails the effectful operation of reading from `stdin`, the type of `getChar` is annotated with `IO`.

Generally, values with parameterized type are deemed "effectful" because those parameterized types capture some *effect*, like an input/output side effect. Some effects can be safely "escaped," with their internal values being separated from the structure of the parameterized type. The `Maybe` type, for example, can be escaped by pattern matching over its values `Just` and `Nothing`.

```
unwrapMessage :: Maybe String -> String
unwrapMessage (Just msg) = "Your message is: " ++ msg
unwrapMessage Nothing = "No messages here!"
```

The upshot of the `IO` type is that there is no "safe escape" from it. Once a value's type is annotated with `IO`, all subsequent operations involving that value must also produce an `IO` result (barring the use of `unsafePerformIO`) [13]. Still, the internal value of an `IO` result can be operated upon within the framework of the `Functor`, `Applicative`, and `Monad` typeclasses, which are introduced in the following section.

F. Built-in Typeclasses: Functor, Applicative, and Monad

Parameterized types are ubiquitous in Haskell as they qualify existing types and express *structure* or *effects*. The standard prelude provides generic operations for working with parameterized types, which are captured in the `Functor`, `Applicative`, and `Monad` typeclasses.

The `Functor` typeclass declares an operation `fmap` that facilitates the application of a function to the elements of a structure while preserving that structure's shape [18].

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

Per the class declaration, a parameterized type `f` is an instance of `Functor` if it provides the operation `fmap` that applies a function `a -> b` to a value `f a` and produces a new value `f b`.

An example of a `Functor` instance is the type `Maybe`, which defines `fmap :: (a -> b) -> Maybe a -> Maybe b` to apply the provided function `a -> b` to the underlying value of a successful result (`Just a`) or propagate a failure (`Nothing`) [16].

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


fmap increment (Just 1) -- Just 2
fmap increment Nothing -- Nothing
```

Similarly, the `Functor` instance `Either` defines `fmap :: (a -> b) -> Either t a -> Either t b` to apply the provided function to the underlying value of a `Right a` result or propagate a `Left t` value.

```
instance Functor (Either t) where
  fmap _ (Left x) = Left x
  fmap f (Right y) = Right (f y)


fmap increment (Left 2) -- Left 2
fmap increment (Right 2) -- Right 3
```

The infix operator `<$>` is equivalent to `fmap` and more commonly used.

```
increment <$> (Left 2) -- Left 2
increment <$> (Right 2) -- Right 3
```

Notice, `fmap` and `<$>` are restricted to applying functions of single arguments. The `Applicative` typeclass provides operations that enable applying functions of several arguments.

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

The syntax `class Functor f => Applicative f` declares that a parameterized type belongs to `Applicative` if it defines the `Applicative` operations—`pure` and `<*>`—*and* is an instance of `Functor` [15]. With currying and the `Applicative` operations, computations over parameterized types can be sequenced and their results combined, thus generalizing `fmap` to functions of several arguments [18]. For example, two `Maybe Int` operands can be summed with the `+`, `<*>`, and `pure` operations, producing `Nothing` if either operand is `Nothing` or a `Just` value otherwise.

```
pure (+) <*> Just 3 <*> Just 5 -- Just 8
pure (+) <*> Just 4 <*> Nothing -- Nothing
```

Similarly, `Applicative` operations on `Either` values propagate `Left` values or produce `Right` results. The `fmap` and `<$>` operations can be used instead of `pure` to apply the initial operation in an applicative sequence.

```
pure (*) <*> Right 3 <*> Right 2 -- Right 6
pure (*) <*> Left 3 <*> Right 2 -- Left 3
```

```
(*) <$> Right 3 <*> Right 2 -- Right 6
(*) <$> Left 3 <*> Right 2 -- Left 3
```

The `Functor` and `Applicative` typeclasses generalize the application of functions producing pure results to effectful arguments. However, they do not capture the common pattern of applying a function producing an effectful result to an effectful value; this operation is instead captured by the `>>=` (pronounced "bind") operator provided by the `Monad` typeclass [16].

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

For example, consider a function `safeSqrt :: Float -> Maybe Float` that returns `Nothing` if the argument is negative and the result of the square root operation otherwise.

```
safeSqrt :: Float -> Maybe Float
safeSqrt x = if x < 0 then Nothing else Just (sqrt x)

safeSqrt 16 -- Just 4.0
safeSqrt (-4) -- Nothing
```

Given that `Maybe` is an instance of `Monad`, `safeSqrt` operations can be chained with the `>>=` operator, repeatedly applying the square root operation to valid inputs and propagating `Nothing` results for invalid inputs.

```
safeSqrt 256 >>= safeSqrt >>= safeSqrt -- Just 2.0
safeSqrt (-4) >>= safeSqrt -- Nothing
```

Now, to write a function `safeSqrtSum` that uses `safeSqrt` to compute $\sqrt{x} + \sqrt{y}$, several `>>=` operators can be sequenced together with lambda expressions that `return` the `Float` result, producing a final `Maybe Float` value.

```
safeSqrtSum :: Float -> Float -> Maybe Float
safeSqrtSum x y =
  safeSqrt x >>= \l ->
    safeSqrt y >>= \r ->
      return (l + r)

safeSqrtSum 9 16 -- Just 7.0
safeSqrtSum 4 -1 -- Nothing
```

Haskell provides the `do` syntax to express this common pattern of computation more concisely. The following definition of `safeSqrtSum` is equivalent to the previous.

```
safeSqrtSum x y = do
  l <- safeSqrt x
  r <- safeSqrt y
  return (l + r)
```

As mentioned, there exists no safe escape from the `IO` type. Instead, the `Functor`, `Applicative`, and `Monad` typeclasses provide a framework for working with `IO` values. As an instance of these typeclasses, the `IO` type supports the operators necessary to apply and sequence `IO` operations without needing to escape the `IO` type.

```
-- Read a character from stdin and capitalize it
-- Return the result
getCapitalChar :: IO Char
getCapitalChar = toUpper <$> getChar

-- Read two characters from stdin and compare them with `==`
-- Return the result
compareTwoChars :: IO Bool
compareTwoChars = (==) <$> getChar <*> getChar

-- Print a welcome message and prompt the user for input
-- Return the user input
promptRPS :: IO String
promptRPS = do
  print "Welcome to Rock, Paper, Scissors!"
  print "Do you choose Rock, Paper, or Scissors?"
  getLine
```

G. More Useful Monads: Except and Reader

So far, we've considered several parameterized types, including list (`[]`), `Maybe`, `Either`, and `IO`. Each of these types is an instance of `Monad` and finds use in almost every Haskell program. Some less common, more specialized `Monad` types include `Reader` and `Except`.

The `Reader` monad encapsulates computations that access values held in some fixed, read-only enclosing environment. The `Reader` type constructor is parameterized over two types, `r` and `a` [19–21]. The first type parameter `r` represents the type of values stored in the environment, and the second parameter `a` represents the type of the `Reader` computation's result. For example, a computation of type `Reader String Int` can access a `String` environment value and produces an `Int` result. Such a `Reader` may be constructed with the `reader :: (r -> a) -> Reader r a` function, which produces a `Reader r a` from a function `r -> a`. As a contrived example, consider a `Reader String Int` value constructed from the function `length :: String -> Int`.

```
stringReader :: Reader String Int
stringReader = reader length
```

The `Reader String Int` computation is executed with the `runReader` function, which takes a `Reader` and an initial `String` value for the environment. In this case, the result of running the computation is the result of applying `length` to the provided `String` environment.

```
runReader stringReader "Hello World"
-- 11
```

Typically, a `Reader` computation will access the read-only environment via the `ask` function [21]. For example, if a `String` environment represents the user's name, then the following `Reader String String` computation prints a welcome message to the user when run.

```
welcomeMessage :: Reader String String
welcomeMessage = do
  name <- ask
  return ("Welcome back, " <> name <> ".")

runReader welcomeMessage "Parker"
-- "Welcome back, Parker."
```

The name `Reader` describes these computations because the associated environment data is read-only. Although `Reader`

computations cannot change environments within their own scopes, they can call nested `Reader` computations with modified environments [21]. The function `local :: (r -> r) -> Reader r a -> Reader r a` takes as argument a function that maps the `Reader` environment and performs the provided `Reader` computation with the new environment. The following example defines a new `Reader` computation, `welcomeFirstInitial`, that truncates the `String` environment with `firstLetter` before calling `welcomeMessage`.

```
welcomeFirstInitial :: Reader String String
welcomeFirstInitial = local firstLetter welcomeMessage
 where
  firstLetter = singleton . head

runReader welcomeFirstInitial "Parker"
-- "Welcome back, P."
```

In hson, the `Reader` monad is used in the interpreter to access the program environment, which is a map that associates variable identifiers with the values to which they are bound. The hson interpreter also utilizes the `Except` monad, which adds error handling to its computations. The type `Except e a` represents a computation that either produces a successful result of type `a` or produces an error result of type `e` [21]. Within an `Except` computation, the `throwError :: e -> Except e a` function signals an error. In the following code snippet, a new `safeSqrt2` function produces a result of type `Except String Float` and calls `throwError` for invalid inputs.

```
-- Original `safeSqrt` with Maybe result
safeSqrt :: Float -> Maybe Float
safeSqrt x = if x < 0 then Nothing else Just (sqrt x)

-- New `safeSqrt2` with Except String result
safeSqrt2 :: Float -> Except String Float
safeSqrt2 x = if x < 0
  then throwError "Received negative input!"
  else return (sqrt x)
```

The advantage of using an `Except String` result over a `Maybe` result is that `throwError` can provide contextual information about a failure. Now, when the `Except String Float` computation is run with `runExcept`, it produces an `Either String Float` result that represents a successful computation in `Right Float` or an error message in `Left String`.

```
runSafeSqrt2 x = runExcept (safeSqrt2 x)

runSafeSqrt2 4 -- Right 2.0
runSafeSqrt2 16 -- Right 4.0
runSafeSqrt2 (-1) -- Left "Received negative input!"
```

H. Monad Transformers and Composing Monads

While the features of the `Reader` and `Except` monads are powerful on their own, they grow even mightier when composed. The *monad transformer* framework enables combining monad features into a single new monad through composition [20, 21]. The definitions of `Reader` and `Except` reveal that they are aliases for the `ReaderT` and `ExceptT` monad *transformers* that compose the `Identity` monad.

```
type Reader r a = ReaderT r Identity a
type Except e a = ExceptT e Identity a
```

The `Identity` monad is trivial: it provides no features and represents "no effect" [20, 21]. So, the `Reader` and `Except` monads are special cases of the `ReaderT` and `ExceptT` monad transformers, where the nested monad in composition is `Identity`.

The `ReaderT` and `ExceptT` monad transformers can be composed to produce a new custom monad, `Eval`, that provides access to *both* a read-only environment and error handling.

```
type Eval a = ReaderT String (ExceptT String Identity) a
```

The outermost `ReaderT` monad has `ExceptT` as its base, and the `ExceptT` monad has `Identity` as its base. To run an `Eval` computation, the `runReaderT`, `runExceptT`, and `runIdentity` functions are composed to unwrap all the monad transformers and obtain the final `Either String a` result.

```
runEval :: Eval a -> Either String a
runEval x =
  runIdentity $ runExceptT $ runReaderT x "Initial Environment"
```

Monad transformers also allow the integration of effectful computations with the `IO` monad. When the innermost `Identity` monad in `Eval` is replaced with `IO` and the computation is run, the result is an `IO (Either String a)` value.

```
type Eval a = ReaderT String (ExceptT String IO) a

runEval :: Eval a -> IO (Either String a)
runEval x =
  runExceptT $ runReaderT x "Initial Environment"
```

The hson interpreter utilizes the `ReaderT`, `ExceptT`, and `IO` monad transformers in this way to integrate read-only environment access, error handling, and I/O effects.

In general, all language features introduced in this section are utilized considerably within hson. The following sections demonstrate how hson leverages these language features in its implementation of parsing, interpreting, and testing.

IV. Parsing

The entry point of all Haskell programs is the `main :: IO ()` function. The `main` function of hson is concise and clear: first read and parse the command-line options, then read the input hson script and JSON data, then call `run`.

```
main :: IO ()
main = do
  opts <- hsonOpts
  hsonIn <- readHSON $ hsonInputOpt opts
  jsonIn <- readJSON $ jsonInputOpt opts
  run jsonIn hsonIn opts
```

The `run` function first executes the hson parser on the input hson script. Then, if the hson parse was successful, `run` calls `runProg`, which parses the input JSON and evaluates the hson script, eventually printing its result. Providing a JSON input to hson is optional, so `runProg` accepts the JSON string as a `Maybe` value.

```
runProg :: Maybe BL.ByteString -> Program -> IO ()
runProg Nothing prog = runInterpretNoJSON prog >>= printResult
```

```
runProg (Just json) prog = case decode json of
  Left err -> print . JSONParsingError $ T.pack err
  Right json -> runInterpretWithJSON json prog >>= printResult
```

In total, hson has three parse responsibilities: command-line options, hson script, and JSON data. The following sections highlight how Haskell language features and techniques contributed to implementing these parsers. This survey begins with the hson parser, which utilizes *parser combinators* to encode the hson grammar and execute recursive-descent parsing.

A. The hson Parser

An hson program comprises a series of zero or more variable declarations followed by a single expression, as illustrated by the following type definition.

```
type Program = ([VarStmt], Expr)
```

The hson parser is responsible for parsing an input according to the grammar rules and constructing the appropriate `Program` value, which represents the root of the hson parse tree. The BNF description of the hson language grammar is given in Appendix A.

The `VarStmt` type represents variables declared with either a standalone identifier, a destructured array, or a destructured object.

```
data VarStmt
  = VarDeclStmt VarDecl
  | ObjectDestructureDeclStmt ObjectDestructureDecl
  | ArrayDestructureDeclStmt ArrayDestructureDecl
```

The following code snippet illustrates each kind of variable declaration.

```
let restaurants = $; // identifier
let [_, secondRestaurant] = restaurants; // destructured array
let { name } = secondRestaurant; // destructured object
```

An `Expr` represents any expression node in the hson parse tree.

```
data Expr
  = ArrayInitializerExpr ArrayInitializer
  | ArrowFunctionExpr ArrowFunction
  | BinaryExpr Binary
  | CallExpr Call
  | ConditionalExpr Conditional
  | DollarExpr Dollar
  | GetExpr Get
  | GroupingExpr Grouping
  | IndexExpr Index
  | LiteralExpr Literal
  | LogicalExpr Logical
  | ObjectInitializerExpr ObjectInitializer
  | UnaryExpr Unary
  | VariableExpr Variable
  deriving (Show, Eq)
```

The simple program `1+2`, for example, produces a `Program` value with an empty `VarStmt` list and a `BinaryExpr` as the root of the `Expr` tree.

```
  ( []
  , BinaryExpr
      ( Binary
          { binLeft =
              LiteralExpr
                ( Literal
                    { litTok =
                        Token
                          { tokenType = TokenNumber
                          , literal = Just 1
                          , pos = (line 1, column 1)
                          }
                    }
                )
          , binOp =
              Token
                { tokenType = TokenPlus
                , literal = Nothing
                , pos = (line 1, column 3)
                }
          , binRight =
              LiteralExpr
                ( Literal
                    { litTok =
                        Token
                          { tokenType = TokenNumber
                          , literal = Just 2
                          , pos = (line 1, column 5)
                          }
                    }
                )
          }
      )
  )
```

The `program` function within hson's parser corresponds to the start rule of the hson grammar and is the entry point for the recursive descent parser.

```
program :: HSONParser Program
program = do
  declarations <- many declaration
  expr <- expression
  eof
  return (declarations, expr)
```

The definition of `program` is expressive enough that even a Haskell novice could intuit the semantics of each line

1) Parse many variable declarations
2) Parse an expression
3) Expect the end of the input
4) Return the `Program` parse result

Each function called within the `do` block of `program`—`many`, `declaration`, `expression`, and `eof`—are themselves functions that return a parse result. The `declaration` function, for example, parses variable declarations like `let sum = 1 + 2` and returns a `VarStmt` parse result.

```
declaration :: HSONParser VarStmt
```

This ability to sequence parse operations within the `do` block is afforded by the monadic `HSONParser` type, which wraps the

result of every parse operation. The definition of the `HSONParser` type reveals that it aliases a specific instance of the `ParsecT` type.

```
type HSONParser = ParsecT T.Text () Identity
```

`ParsecT` is a monad transformer provided by the `parsec` library that defines sequencing, failure, and error-handling operations for parse computations. The `parsec` library also defines several primitive *parser combinators*, which are parse operations like `many` that can be sequenced and composed to produce more complex parsers [22]. A `ParsecT` value has four type parameters, `s u m a`, where `s` is the type of the input, `u` is the type of some user state, `m` is an underlying monad, and `a` is the type of the parse result. Thus, the `HSONParser` type simply represents a `ParsecT` value with no user state, `Identity` as its base monad ("no effect"), and an input of type `Text`, which is a time and space-efficient Unicode character string representation [23].

Running parse computations entails calling `runParsecT` with the input string, the initial user state, and an optional source name argument (e.g., the input file name). The hson parser begins its descent at the `program` function, has no user state, and specifies no source name.

```
runHSONParser :: T.Text -> Either ParseError Program
runHSONParser s = runHSONParser' s program


runHSONParser' :: T.Text -> HSONParser a -> Either ParseError a
runHSONParser' s p = runIdentity $ runParserT p () "" s
```

The result of an `HSONParser Program` computation is `Either ParseError Program`, which captures the possibility of failure while parsing `program`.

The `program` parser first parses a sequence of zero or more variable declarations. The `declaration` function handles each kind of variable declaration and produces a `VarStmt` result. The `parseVarDecl`, `parseObjDestDecl`, and `parseArrDestDecl` functions handle parsing identifiers, destructured objects, and destructured arrays, respectively.

```
declaration :: HSONParser VarStmt
declaration = do
  letVar
  stmt <-
    try parseVarDecl
      <|> try parseObjDestDecl
      <|> try parseArrDestDecl
      <?> "identifier or destructuring"
  equal
  initializer <- expression
  semicolon
  return $ stmt initializer
```

The `(<|>)` operator is called the "choice combinator": it returns the parse result on its left-hand side if successful, and returns the parse result on the right-hand side otherwise. The choice operator is combined with the `try` combinator to implement arbitrary lookahead. Without the application of `try`, a parser will consume input even when it fails. The `try p` parser performs the parse operation `p` but behaves as though no input was consumed when it fails [22]. Thus, combining `try` with choice produces parsers that attempt to parse several alternative productions before failing.

The `<?>` operator applies a high-level label to a parser so that, when it fails without consuming input, the associated error message includes that label [22]. For example, consider an incorrect hson program that uses pipe characters (`||`) instead of square brackets (`[]`) for array destructuring.

```
let |_, secondRestaurant| = restaurants;
```

When the script is parsed, `parsec` produces the following error message for hson to print.

```
(line 1, column 5):
unexpected "|"
expecting identifier or destructuring
```

Without the above use of `<?>`, the failed `declaration` parser produces a character-level error message instead.

```
unexpected "|"
expecting identifier, "{" or "["
```

The `declaration` parser executes the `parseVarDecl`, `parseObjDestDecl`, and `parseArrDestDecl` functions to handle each kind of variable declaration statement. These three parse functions each return the same result type: `HSONParser (Expr -> VarStmt)`. Thus, within the `do` block, `stmt` has type `Expr -> VarStmt` and is a *function* that maps an expression to a variable statement.

```
stmt <-
  try parseVarDecl
    <|> try parseObjDestDecl
    <|> try parseArrDestDecl
    <?> "identifier or destructuring"
```

In the last line of `declaration`, the `stmt` function is finally applied to the parsed `initializer` expression, producing the resulting `VarStmt`.

```
initializer <- expression
semicolon
return $ stmt initializer
```

This implementation of `declaration` demonstrates the utility of *higher-order functions*. The `parseVarDecl`, `parseObjDestDecl`, and `parseArrDestDecl` functions return *new functions* that map an eventual `Expr` value to a `VarStmt` result. For example, the `parseVarDecl` function parses an identifier and returns a function that produces a `VarStmt` when eventually provided the parsed initializer expression.

```
parseVarDecl :: HSONParser (Expr -> VarStmt)
parseVarDecl = do
  declName <- identifier
  return $
    \expr ->
      VarDeclStmt
        VarDecl
          { declName = declName
          , initializer = expr
          }
```

In an imperative language, one might return a partially-constructed `VarStmt` from a function like `parseVarDecl` and utilize

mutability or reassignment later to supply the missing initializer expression. However, if the caller fails to eventually provide that initializer expression, a `VarStmt` could float around the program with missing data and produce unexpected outcomes. Thus, while Haskell's immutability restrictions and strong type system demand thoughtful solutions, they help mitigate uncertainty by eliminating the possibility of partial and ever-changing values.

For each production in the hson grammar, the hson parser implements a function that utilizes combinators and patterns like those found in the definition of `declaration`. The original `program` function calls `declaration` with the `many` parser, which applies the `declaration` parser zero or more times and returns a list of parse results [24].

```
program :: HSONParser Program
program = do
  declarations <- many declaration
  expr <- expression
  eof
  return (declarations, expr)
```

Next, `program` calls `expression`, which is responsible for producing the root `Expr` node that will be evaluated and output by hson. The `expression` function descends through precedence levels, eventually parsing logical expressions like the null-coalescing operator `??`, the logical "or" operator `||`, and the logical "and" operator `&&`.

```
nullCoalesce :: HSONParser Expr
nullCoalesce = do
  chainl1 logicOr parseNullCoalesce
 where
  parseNullCoalesce = parseLogicalOp questionQuestion

logicOr :: HSONParser Expr
logicOr = do
  chainl1 logicAnd parseOr
 where
  parseOr = parseLogicalOp orOr

logicAnd :: HSONParser Expr
logicAnd = do
  chainl1 equality parseAnd
 where
  parseAnd = parseLogicalOp andAnd
```

The `chainl1` combinator is common to each of these logical expression parsers and is responsible for parsing left-associative binary operations. Specifically, `chainl1` parses one or more occurrences of a particular expression, each separated by an operator. The first argument to `chainl1` is an expression parser, `HSONParser Expr`, and the second argument is an operator parser with the type `HSONParser (Expr -> Expr -> Expr)`. The operator parser is responsible for parsing an operator token (e.g., the "logical and" operator `&&`) and returning a function that produces a higher-level expression from left-hand and right-hand side operand expressions. The final result of `chainl1` is given by the left-associative application of each operator function to each consecutive operand expression. Effectively, `chainl1` encodes left-associativity while eliminating left-recursion [22, 24]. This functionality is also useful for parsing left-associative binary expressions.

```
equality :: HSONParser Expr
equality = do
  chainl1 comparison (try parseNeq <|> parseEq)
 where
  parseEq = parseBinaryOp equalEqual
  parseNeq = parseBinaryOp bangEqual

comparison :: HSONParser Expr
comparison = do
  chainl1
    term
    ( try parseGte
        <|> try parseGt
        <|> try parseLte
        <|> parseLt
    )
 where
  parseGt = parseBinaryOp greater
  parseGte = parseBinaryOp greaterEqual
  parseLt = parseBinaryOp less
  parseLte = parseBinaryOp lessEqual

term :: HSONParser Expr
term = do
  chainl1 factor (try parsePlus <|> parseMinus)
 where
  parseMinus = parseBinaryOp minus
  parsePlus = parseBinaryOp plus

factor :: HSONParser Expr
factor = do
  chainl1 unary (try parseDiv <|> parseMult)
 where
  parseMult = parseBinaryOp star
  parseDiv = parseBinaryOp slash
```

In summary, the hson parser *is* the `program` parser, which itself composes and sequences several more specific parsers, each of which comprises several more fundamental parsers with the most fundamental parsers provided by `parsec`. The parser continues its recursive descent down through each parser combinator until eventually parsing the terminals of the hson grammar. For example, parsing an array initializer expression entails applying the `brackets` parser to the `arguments` parser, thus parsing a comma-separated list of expressions between a pair of square brackets. Terminal parsers also record positions and relevant tokens for producing friendly error messages within the hson interpreter.

```
parseArray :: HSONParser Expr
parseArray = do
  bracketPos <- getPosition
  elems <- brackets arguments
  return $
```

```
ArrayInitializerExpr
  ArrayInitializer
    { bracket =
        Token
          { tokenType = TokenLeftBracket
          , literal = Nothing
          , pos = bracketPos
          }
    , elements = elems
    }
```

Parser combinators are a robust tool for implementing parsers. One can quickly encode an entire language grammar through composing and sequencing a relatively small collection of elementary combinators (like that which `parsec` provides). Moreover, the declarative style of parser combinators results in parser code that *looks* like the associated grammar, with each function corresponding to a variable and each definition corresponding to the associated production. Additionally, because parser combinators are utilized within Haskell, the parser implementation has access to Haskell's many features, and the parser results can be easily integrated with the rest of the Haskell program.

B. JSON Parsing

The hson program also accepts JSON data as an optional input and makes it available via the `$` identifier. Before hson can bind the JSON data to `$`, it must first parse and convert the raw JSON string to the appropriate hson value.

The following `HSONValue` data type represents all values within an hson program. The `Function`, `Method`, and `Closure` values represent functions within the hson interpreter, and the other values represent JSON data types.

```
data HSONValue
  = Function Func
  | Method (HSONValue -> Func)
  | Closure Func Environment
  | Array (V.Vector HSONValue)
  | Object (Map.Map T.Text HSONValue)
  | String T.Text
  | Number Scientific
  | Bool Bool
  | Null
```

The goal of the JSON parser within hson is to construct the appropriate `HSONValue` for each node in the JSON parse tree. The popular Haskell library `aeson` makes this JSON parsing task simple: it provides a typeclass `FromJSON` that declares an associated `parseJSON` operation [25].

```
class FromJSON a where
  parseJSON :: Value -> Parser a
```

Like the `HSONValue` type defined in `hson`, the `Value` type defined in `aeson` represents JSON values as Haskell values, so the `parseJSON :: Value -> Parser a` operation maps JSON values to parser results of the given type. Then, converting JSON values into `HSONValue` results entails declaring `HSONValue` as an instance of `FromJSON` and providing the appropriate definition of `parseJSON :: Value -> Parser HSONValue`. Because the `Value` and `HSONValue` types are similar, the implementation of `parseJSON` for `HSONValue` is straightforward: map JSON arrays to hson arrays, JSON objects to hson objects, JSON strings to hson strings, and so forth. The `mapM` function is utilized for data structures like arrays and objects to apply `parseJSON` to each element.

```
-- `A` is the `aeson` namespace, `H` is the `hson` namespace
instance A.FromJSON H.HSONValue where
  parseJSON (A.Array v) = H.Array <$> mapM parseJSON v
  parseJSON (A.Object v) = H.Object <$> mapM parseJSON (A.toMapText v)
  parseJSON (A.String v) = return $ H.String v
  parseJSON (A.Number v) = return $ H.Number v
  parseJSON (A.Bool v) = return $ H.Bool v
  parseJSON A.Null = return H.Null
```

With the `HSONValue` type declared as an instance of `FromJSON`, its values can be produced as the result of JSON decode operations. In hson, the `eitherDecode` function is utilized to produce an `Either String HSONValue` result, where the `Right` value contains the parsed JSON and the `Left` value contains an error message.

```
runProg (Just json) prog = case eitherDecode json of
runProg (Just json) prog = case decode json of
  Left err ->
    print . JSONParsingError $ T.pack err
  Right parsedJSON ->
    runInterpretWithJSON parsedJSON prog >>= printResult
```

If the JSON parse is successful, hson runs the interpreter with the parsed hson script and JSON data. Otherwise, hson prints the JSON parse error.

As demonstrated, `aeson` manages most of the necessary JSON parsing logic. Thanks to the generalized `parseJSON` operation facilitated by Haskell's typeclasses, the only responsibility of hson concerning JSON parsing is to define an instance of `FromJSON` and run the appropriate decode operation on the raw JSON input, handling any parse errors that occur.

C. Applicative Command Line Option Parsing

The hson command-line interface accepts a few options that must also be parsed and handled. Invoking `hson` with the `--help` flag gives the following output that lists each available option.

```
hson - json processing language

Usage:
hson ((--hf|--hfile|--hsonfile FILENAME.HSON) | SCRIPT)
     [(--jf|--jfile|--jsonfile FILENAME.JSON) | (-n|--no-json)]
     [-a|--ast] [-p|--pretty-print] [-o|--omit-eval]

  Parse JSON according to given hson input.

Available options:
  --hf,--hfile,--hsonfile FILENAME.HSON
                           hson input file.
  SCRIPT                   hson script to be run.
  --jf,--jfile,--jsonfile FILENAME.JSON
                           JSON input file.
```

```
-n,--no-json            Run hson without a JSON input.
-a,--ast                Print the hson parse tree.
-p,--pretty-print       Pretty-print the provided hson script.
-o,--omit-eval          Omit the evaluated expression output.
-h,--help               Show this help text
```

To summarize,

- The input hson script can be provided from the command line or a file with the `-hf` flag.
- By default, hson reads JSON from `stdin`. The JSON data can instead be read from a file with the `-jf` flag, or hson can be run without a JSON input with the `-n` flag.
- The `-a` flag prints the parse tree generated from the provided hson script.
- The `-p` flag pretty-prints the provided hson script.
- The `-o` flag prevents hson from printing the evaluated expression.

The parsing of each command-line option and the printing and formatting of the above help message is facilitated by the `optparse-applicative` library [26]. Like `parsec`, `optparse-applicative` provides several combinators that can be sequenced and composed to create new parsers. However, the combinators that `optparse-applicative` provides are specific to parsing command-line options. Moreover, as library name implies, the `optparse-applicative` combinators are applicative, not monadic, so they are sequenced with the `fmap` (`<$>`) and applicative sequencing (`<*>`) operators instead of with the bind (`>>=`) operator or within a `do` block.

Implementing command-line options begins by defining a data type with a field corresponding to the value of each option.

```
data Options = Options
  { hsonInputOpt :: HSONInput
  , jsonInputOpt :: JSONInput
  , doPrintAst :: Bool
  , doPrettyPrint :: Bool
  , doOmitEval :: Bool
  }

data HSONInput
  = HSONFileInput FilePath
  | CmdLineIn String

data JSONInput
  = JSONFileInput FilePath
  | StdIn
  | NoJSONInput
```

The `hsonInputOpt` field determines whether the hson script is provided from the command line (`CmdLineIn`) or from within a file (`HSONFileInput`). The `jsonInputOpt` field determines whether JSON will be read from standard input (`StdIn`), a file (`JSONFileInput`), or not at all (`NoJSONInput`). The other fields correspond to the debugging flags and comprise only `True` and `False` values.

Next, a `Parser` is created for every possible option value. Each parser defines the flags that invoke the respective option (`long` or `short`), the placeholder text for the option's argument (`metavar`), and the associated `help` text. For example, the following snippet defines the two parsers `hsonFileInput` and `cmdLineIn` that correspond to the `HSONFileInput` and `CmdLineIn` values, respectively.

```
hsonFileInput :: Parser HSONInput
hsonFileInput =
  HSONFileInput
    <$> strOption
      ( long "hf"
          <> long "hfile"
          <> long "hsonfile"
          <> metavar "FILENAME.HSON"
          <> help "hson input file."
      )

cmdLineIn :: Parser HSONInput
cmdLineIn =
  CmdLineIn
    <$> argument
      str
      ( metavar "SCRIPT"
          <> help "hson script to be run."
      )
```

Once a parser is implemented for each option value, a single `Options` parser can be constructed to handle all possible options. Within this aggregate parser, the applicative operations sequence options, and the choice operator `<|>` signals mutually exclusive options. For example, a user can provide hson either by file or via the command line, so the choice operator distinguishes the `hsonFileInput` and `cmdLineIn` parsers.

```
opts :: Parser Options
opts =
  Options
    <$> (hsonFileInput <|> cmdLineIn)
    <*> (jsonFileInput <|> stdin)
    <*> printParseTree
    <*> prettyPrint
    <*> hideEval
```

The definition of `opts` expresses the possible command-line options for hson. The set of command-line options can easily be extended by modifying the `Options` data type and defining and sequencing new option parsers with `<*>` and `<|>`.

When hson runs, it first calls `execParser` to parse the command-line options, passing in `info` to specify the information displayed in the help text.

```
runHsonOpts :: IO Options
runHsonOpts =
  execParser $
    info
      (opts <**> helper)
      ( fullDesc
          <> progDesc "Parse JSON according to given hson input."
          <> header "hson - json processing language"
      )
```

Once hson parses the command-line options, it reads the provided hson and JSON according to the values of those options.

```
-- Main.hs
main :: IO ()
main = do
  -- first, read command line options
  opts <- runHsonOpts
  -- read hson according to the `hsonInputOpt` option
  hsonIn <- readHSON $ hsonInputOpt opts
  -- read JSON according to the `jsonInputOpt` option
  jsonIn <- readJSON $ jsonInputOpt opts
  -- run the program
  run jsonIn hsonIn opts
```

Then, `main` calls `run`, which utilizes `when` and `unless` to conditionally execute functions depending on the values of each debug option.

```
run json hson opts = case runHSONParser hson of
  Left err -> TIO.putStrLn $ T.pack $ show err
  Right prog -> do
    when
      (doPrintAst opts)
      (TIO.putStrLn $ T.pack $ show prog)
    when
      (doPrettyPrint opts)
      (TIO.putStrLn $ prettyPrintProg prog)
    unless
      (doOmitEval opts)
      (runProg json prog)
```

Like monadic parsing with parser combinators, applicative parsing is concise, expressive, and extensible. In general, the `Monad` and `Applicative` type classes capture patterns of computation common to many domains. Although the examples in this section pertain to parsing, the abstract operations for sequencing and composing effectful operations can extend naturally to other applications, providing a common language for computing across many domains. The following section illustrates how monads can be composed to implement modular interpreters with enclosed effects.

V. Interpreting

A. The Environment

Once hson finishes parsing its inputs, it passes the parse results to the interpreter to execute the script. The hson interpreter is responsible for visiting each node in the provided parse tree, recursively executing the statement nodes, and evaluating the expression nodes. This process begins with a call to the `runInterpret` function by either `runInterpretWithJSON` or `runInterpretNoJSON`.

```
runInterpretWithJSON ::
  HSONValue -> Program -> IO (Either HSONError HSONValue)
runInterpretWithJSON json prog = runInterpret (mkEnv json) prog

runInterpretNoJSON ::
  Program -> IO (Either HSONError HSONValue)
runInterpretNoJSON prog = runInterpret (mkEnv Null) prog
```

The `mkEnv` function constructs the script's initial environment, a map that associates identifiers with the `HSONValue` values to which they are bound. If JSON data was provided, hson binds the parsed value to the `$` identifier; otherwise, `$` evaluates to `null`.

```
type Environment = Map.Map T.Text HSONValue

mkEnv :: HSONValue -> Environment
mkEnv json = Map.singleton "$" json `Map.union` builtInFunctions
```

The initial environment also includes all built-in functions and their associated identifiers, which are defined in the `builtInFunctions` table.

```
builtInFunctions =
  Map.fromList
    [ ("keys", mkFunction keys)
    , ("values", mkFunction values)
    , ("hasProperty", mkFunction hasProperty)
    , ("toJSON", mkFunction hsonToJSON)
    , ("toString", mkFunction hsonToString)
    , ("length", mkFunction hsonLength)
    , ("at", mkFunction hsonAt)
    , ("reverse", mkFunction hsonReverse)
    , ("map", mkFunction hsonMap)
    , ("filter", mkFunction hsonFilter)
    , ("reduce", mkFunction hsonReduce)
    ]
```

Each `mkFunction` call constructs a `Function` from a `FunctionDefinition`. A `FunctionDefinition` is a mapping from a list of values representing arguments to an `Eval HSONValue` result.

```
type FunctionDefinition = [HSONValue] -> Eval HSONValue

mkFunction :: FunctionDefinition -> HSONValue
mkFunction f = Function (Func f)
```

The `Eval` monad wraps every evaluated result of the hson interpreter. The definition of `Eval` reveals a composition of monad transformers, each introducing a specific feature to interpreter computations [27].

```
newtype Eval a = Eval
  { unEval :: ReaderT Environment (ExceptT HSONError IO) a
  }
  deriving
    ( Applicative
    , Functor
    , Monad
    , MonadError HSONError
    , MonadIO
    , MonadReader Environment
    )
```

The `ReaderT Environment` monad transformer provides all interpreter computations with read-only access to the program environment. The `ExceptT HSONError` monad transformer enables interpreter computations to produce and handle errors by calling

`throwError` with `HSONError` values. The base monad `IO` integrates I/O capabilities into the interpreter [21].

The `deriving` clause generates standard instance declarations of various typeclasses (e.g., `Functor`, `Applicative`, and `Monad`) for the `Eval` type. Without `deriving`, `Eval` would need an explicit instance declaration for every relevant typeclass. In this way, the `deriving` clause shortcuts those instance declarations by inheriting each instance from its representation (i.e., the `ReaderT` type it contains). By default, `deriving` is limited to only a few typeclasses, but is extended within `hson` through the `GeneralizedNewtypeDeriving` Haskell language extension [28].

B. Defining Built-in Functions

Creating a new built-in function entails defining a `FunctionDefinition`, which maps a list of `HSONValue` arguments to an `Eval HSONValue` result, and binding it to a name within the `builtInFunctions` table. Consider, for example, the built-in `reverse` function, which reverses the elements of a provided string or array. Under the hood, the environment associates the `reverse` identifier with the `hsonReverse` Haskell function. When `reverse` is called within an hson script, the corresponding `hsonReverse` Haskell function is executed. The definition of `hsonReverse` utilizes pattern matching to operate appropriately on valid arguments and throw errors for invalid arguments.

```
builtInFunctions =
  Map.fromList
    [ ("keys", mkFunction keys)
    , ("values", mkFunction values)
    ...
    , ("reverse", mkFunction hsonReverse)
    ...
    ]

...

hsonReverse :: FunctionDefinition
hsonReverse [Array arr] =
  return $ Array $ V.reverse arr
hsonReverse [String str] =
  return $ String $ T.reverse str
hsonReverse [arg] =
  throwError $ UnexpectedType "array or string" (showType arg)
hsonReverse args =
  throwError $ ArgumentCount 1 args
```

The first two equations in the definition of `hsonReverse` handle valid arguments. If a single array or string is passed to `reverse`, the `hsonReverse` function applies the appropriate Haskell `reverse` function to the argument and returns the result. The third `hsonReverse` equation is responsible for throwing an `UnexpectedType` error if `reverse` is called with a single argument that is not a string or array. Otherwise, the final equation handles all other argument lists by throwing an `ArgumentCount` error.

Compare the above definition of `hsonReverse` with the near-identical definition of the built-in `length` function, which, like `reverse`, takes as argument either a string or an array, and returns the number of elements.

```
hsonLength :: FunctionDefinition
hsonLength [Array arr] =
  return $ Number $ fromIntegral $ V.length arr
hsonLength [String str] =
  return $ Number $ fromIntegral $ T.length str
hsonLength [arg] =
  throwError $ UnexpectedType "array or string" (showType arg)
hsonLength args =
  throwError $ ArgumentCount 1 args
```

Each built-in function definition follows a similar pattern of executing the appropriate operations on valid arguments and throwing the appropriate errors for invalid arguments. Haskell's pattern-matching allows for the expressive and direct handling of every possible argument combination, which helps prevent mistakes while implementing complex parameter lists. For example, the `includes` function takes two string arguments, `haystack` and `needle`, and returns `true` if `needle` is a substring of `haystack`. If `includes` does not receive exactly two string arguments, hson reports a type error.

```
includes("hello world", "world") // true
includes("hello world", "Parker") // false


// Call error at (line 1, column 9):
// Expected string, received number.
includes("hello world", 3)
```

The associated `hsonIncludes` Haskell function defines the behavior for the hson `includes` function. Pattern matching is used to apply the Haskell `isInfixOf` function to an argument list consisting of exactly two string elements. Otherwise, `hsonIncludes` throws the appropriate error.

```
hsonIncludes :: FunctionDefinition
hsonIncludes [String haystack, String needle] =
  return $ Bool $ needle `T.isInfixOf` haystack
hsonIncludes [String _, arg] =
  throwError $ UnexpectedType "string" (showType arg)
hsonIncludes [arg, _] =
  throwError $ UnexpectedType "string" (showType arg)
hsonIncludes args =
  throwError $ ArgumentCount 2 args
```

C. First-Class Function Evaluation

Some built-in hson functions are higher-order and accept other functions as input. For example, the hson `map` function accepts two arguments—an array and a function—and applies the function to each array element.

```
map([1,2,3], |n| => n + 1) // [2,3,4]
```

Under the hood, the function argument is represented as a *closure*, which, alongside the function behavior, contains the variables surrounding that function declaration. This way, a function only has access to the variables defined within the environment that is active when the function is declared [29]. To appreciate this behavior, consider the following two

examples. In the first example, the `addX` function can access the variable `x` because `x` is defined before the declaration of `addX`.

```
let x = 2;
let addX = |n| => n + x;
map([1,2,3], addX) // [3,4,5]
```

However, in the following example, `addY` *cannot* access the value of `y`. Even though `addY` is called after `y` is defined, `y` is not in the environment when `addY` is declared.

```
let addY = |n| => n + y;
let y = 4;
// Call error at (line 4, column 4):
// Undefined variable "y" at (line 1, column 23).
map([1,2,3], addY)
```

The Haskell function underlying the hson `map` function, `hsonMap`, reveals how this behavior is encoded.

```
hsonMap :: FunctionDefinition
hsonMap [Array arr, Closure (Func f) env] =
  Array <$> local (const env) (V.mapM (f . L.singleton) arr)
hsonMap [Array _, arg] =
  throwError $ UnexpectedType "function" (showType arg)
hsonMap [arg, _] =
  throwError $ UnexpectedType "array" (showType arg)
hsonMap args =
  throwError $ ArgumentCount 2 args
```

When `hsonMap` receives valid arguments (i.e., an array and a closure), it calls the `local` operation from `ReaderT`, which executes the operation in its second argument with an environment modified by the function passed to its first argument. Here, `local` receives `const env`, which effectively replaces the current environment with the environment stored in the `Closure`. The second argument to `local` is the Haskell `map` function that applies the provided function argument to each element in the array argument.

D. Evaluating hson Programs

The hson interpreter begins by calling `runInterpret`. At a high level, `runInterpret` maps a parsed hson program and an initial environment to an `HSONValue` if it succeeds or an `HSONError` if it fails. The `runInterpret` function directly calls `interpret`, which returns an `Eval HSONValue`, and unwraps all of the monad transformers that `Eval` comprises.

```
runInterpret ::
  Environment -> Program -> IO (Either HSONError HSONValue)
runInterpret env prog =
  runExceptT $ runReaderT (unEval $ interpret prog) env
```

The `interpret` function maps an initial `Program` value to an `Eval HSONValue` result representing the evaluated expression at the end of the hson script. The `interpret` function utilizes pattern matching and recursion to handle each variable declaration before evaluating the final expression in its base case. For each variable declaration, the interpreter evaluates the associated initializer expression and binds the identifier to the result. Then, it recursively calls `interpret` on the remaining variable declarations with the newly updated environment. A variable declaration is either a single identifier, a destructured object, or a destructured array, so pattern matching is again utilized to handle each possibility.

```
interpret :: Program -> Eval HSONValue
-- base case:
interpret ([], expr) = eval expr
-- recursive case:
interpret (stmt : stmts, expr) = do
  -- pattern match over the variable declaration types
  case stmt of
    VarDeclStmt
      ( VarDecl
          (Token _ (Just (String name)) _)
          initializer
        ) -> do
        val <- eval initializer
        local (Map.insert name val) $ interpret (stmts, expr)
    ObjectDestructureDeclStmt
      (ObjectDestructureDecl kvs initializer) -> do
        val <- eval initializer
        case val of
          Object o ->
            local (Map.union $ bindDestObj kvs o) $
              interpret (stmts, expr)
          v -> throwError $ UnexpectedType "object" (showType v)
    ArrayDestructureDeclStmt
      (ArrayDestructureDecl elems initializer) -> do
        val <- eval initializer
        case val of
          Array arr ->
            local (Map.union $ bindDestArr elems arr) $
              interpret (stmts, expr)
          v -> throwError $ UnexpectedType "array" (showType v)
```

Most of the hson interpreter logic is contained within the `eval` function, which evaluates parsed `Expr` values and produces `Eval HSONValue` results.

```
eval :: Expr -> Eval HSONValue
```

The `eval` function pattern matches over all possible expressions, evaluating each expression type appropriately. Evaluation of literal expressions is the simplest example of `eval`: given the token contained with the `Literal` expression, `eval` returns the appropriate value.

```
newtype Literal = Literal {litTok :: Token}
  deriving (Show, Eq)

...

data Token = Token
  { tokenType :: TokenType
  , literal :: Maybe HSONValue
  , pos :: SourcePos
  }
  deriving (Show)

...
```

```
eval (LiteralExpr (Literal (Token _ (Just v) _))) =
  return v
eval (LiteralExpr (Literal (Token TokenTrue _ _))) =
  return $ Bool True
eval (LiteralExpr (Literal (Token TokenFalse _ _))) =
  return $ Bool False
eval (LiteralExpr (Literal (Token TokenNull _ _))) =
  return Null
```

If a `LiteralExpr` represents a string or a number literal, then the `literal` field contains a `Just` result with the corresponding value, which `eval` returns. Otherwise, the `literal` field is `Nothing`, and `eval` pattern matches on the `tokenType` field.

Evaluating a `BinaryExpr` entails first evaluating left- and right-hand side operand expressions. Then, `eval` pattern matches on the operator token to apply the appropriate operation. An operation is either a numeric comparison (`numCmp`), a numeric operation (`numOp`), or a "plus" operation (`valuePlus`) that applies addition to numeric operands and string concatenation otherwise.

```
eval (BinaryExpr (Binary l opTok r)) = do
  left <- eval l
  right <- eval r
  case opTok of
    Token TokenEqualEqual _ _ -> return $ Bool $ left == right
    Token TokenBangEqual _ _ -> return $ Bool $ left == right
    Token TokenGreater _ _ -> numCmp opTok (>) left right
    Token TokenGreaterEqual _ _ -> numCmp opTok (>=) left right
    Token TokenLess _ _ -> numCmp opTok (<) left right
    Token TokenLessEqual _ _ -> numCmp opTok (<=) left right
    Token TokenMinus _ _ -> numOp opTok (-) left right
    Token TokenStar _ _ -> numOp opTok (*) left right
    Token TokenSlash _ _ -> numOp opTok (/) left right
    Token TokenPlus _ _ -> valuePlus opTok left right
    _otherToken -> throwError $ UnhandledOperator opTok

valuePlus :: Token -> HSONValue -> HSONValue -> Eval HSONValue
valuePlus _ (Number x) (Number y) =
  return $ Number $ x + y
valuePlus _ (String x) (String y) =
  return $ String $ x <> y
valuePlus _ (Number x) (String y) =
  return $ String $ T.pack (show x) <> y
valuePlus _ (String x) (Number y) =
  return $ String $ x <> T.pack (show y)
valuePlus opTok _ _ =
  throwError $
    TypeError opTok "operands must be either strings or numbers"
```

Logical expressions are evaluated similarly, but the interpreter utilizes short-circuiting instead of immediately evaluating both operands. The Haskell code that implements `eval` for each logical operation is so expressive that it almost reads like English!

```
eval (LogicalExpr (Logical l opTok r)) = do
  left <- eval l
  case opTok of
    Token TokenOrOr _ _ ->
      if isTruthy left then return left else eval r
    Token TokenAndAnd _ _ ->
      if isTruthy left then eval r else return left
    Token TokenQuestionQuestion _ _ ->
      if left == Null then eval r else return left
    _unrecognizedOperator ->
      throwError $ UnhandledOperator opTok
```

Evaluating a `VariableExpr` entails finding the associated value of the identifier in the environment. If the value exists, the expression evaluates to it; otherwise, an `UndefinedVariable` error is thrown. The `eval` function retrieves the environment from the `ReaderT` monad with `ask` and pattern matches over the `Maybe` result of `Map.lookup`.

```
eval
  ( VariableExpr
      (Variable tok@(Token TokenIdentifier (Just (String s)) _))
  ) = do
  val <- asks (Map.lookup s)
  case val of
    Nothing -> throwError (UndefinedVariable tok)
    Just v -> return v
```

A `CallExpr` represents a function call like `doSomething(x, y)`: it comprises a `callee` expression (`doSomething`) and a list of argument expressions (`x` and `y`). The interpreter evaluates a `CallExpr` by first evaluating the `callee` expression. Then, if the result is *callable*, `eval` will evaluate each argument and pass their values as arguments to the `callee`. Otherwise, the interpreter throws an `UncallableExpression` error.

```
eval (CallExpr (Call callee tok args)) = do
  res <- eval callee
  case res of
    Function f -> do
      args <- mapM eval args
      fn f args `catchError` (throwError . CallError tok)
    Closure f env -> do
      args <- mapM eval args
      local (const env) $
        fn f args `catchError` (throwError . CallError tok)
    _uncallableValue -> throwError $ UncallableExpression tok
```

Notice that a callable value is either a `Function` or a `Closure`. The `Function` value represents a built-in function and does not have an associated environment. For a `Closure`, which is associated with an environment, `eval` calls `local` to replace the current environment with the closure's environment before executing the function. Finally, if the function call fails, `eval` catches the error with `catchError` and re-throws it as a `CallError`.

The `eval` function definition comprises several other equations, each handling a particular expression type. The goal of walking through various equations in its definition has been to reveal how various Haskell features facilitate a readable, modular, and correct implementation. As illustrated, Haskell's support for pattern matching and recursion simplifies the process of evaluating each parse tree node while also promoting expressiveness. Discovering how a particular expression type

is evaluated equates to finding the matching equation within `eval`. Adding interpreter support for a new expression type entails creating a new equation within `eval` for that type. Finally, because all `eval` computations produce a result in the `Eval` monad, the interpreter has access to all computational features afforded by the monad transformers that `Eval` comprises. In turn, each computational feature of the interpreter is clearly expressed in the definition of `Eval`, and the interpreter computations and their effects are naturally contained and separated from the rest of the program.

VI. Testing

A. Unit Testing

The restrictions enforced by Haskell about types, purity, and side effects aim to emphasize program correctness and bolster developer confidence. Still, developers working with other programming languages, even those that are imperative and lacking a type system, can feel confident about their code by *testing* its behavior. The "unit test," for example, demonstrates that an individual software component behaves according to the original design specification and intention. In a unit test, the target software component is isolated from the rest of the system, and the tester defines and controls its input [30]. Essentially, a unit test seeks to answer the question, "Given a specific input, does the unit of code produce the expected output (or side effects)?" A standard framework for writing unit tests is the *arrange-act-assert* pattern [31]:

1) Organize the predefined input data and environment ("arrange")
2) Invoke the target software component with the input ("act")
3) Verify that the resulting output matches the expected output ("assert")

Ideally, a test ensures that the target software component behaves correctly for *all possible inputs*. For example, a test for a `divide(a, b)` function seeks to ensure that it produces an expected result for all numeric `a` and `b`. However, testing the behavior of `divide` over its entire input domain is infeasible. So, the arrange-act-assert process might be repeated with, for example, positive, negative, and fractional inputs.

```
describe('divide(a, b)', () => {
  it('divides positive inputs', () => {
    // arrange
    const a = 6;
    const b = 3;

    // act
    const result = divide(a, b);

    // assert
    expect(result).toBe(2);
  });

  it('divides negative inputs', () => {
    ...
  });

  it('divides fractional inputs', () => {
    ...
  });
});
```

Generally, it is hopeless to test every possible input, but developers can remain somewhat confident in their code if the tests pass for various inputs. However, the burden of generating a sufficient variety of inputs falls on the developers, and knowing which inputs are worth testing may demand considerable expertise or creativity. If, for example, one forgets to test how the `divide(a, b)` function behaves when `b` is 0, one might encounter unexpected behavior or a runtime error in circumstances where `b` is possibly 0.

So, code verification via unit testing demands understanding the possible inputs that the target code may receive. Unfortunately, this means that, in practice, unit tests written for existing code are only somewhat effective at revealing unexpected behavior. If all possible inputs are known when testing a chunk of code, all possible inputs were likely considered and handled when that code was written. And if an edge case was not considered when a chunk of code was written, it's unlikely that it will suddenly be considered when that code is tested. Still, unit tests are powerful tools for providing developers with quick feedback and bolstering code against regressions [32]. Thus, even many Haskell codebases employ unit tests, and Haskell's purity restrictions make those tests relatively easy to write. Where developers working in imperative languages might spend significant time setting up the complex system state or environment with which their target software interacts [32], Haskell developers need only provide the appropriate inputs to the target functions, since purity guarantees that function outputs depend only on their inputs and modify no system state [4].

B. Property-Based Testing

The hson code base does not yet employ unit tests but leverages a different technique called *property-based testing*. As the name implies, property-based tests work by verifying that a chunk of code abides by a property [4]. For example, one could implement a property-based test for the following property of the `divide(a, b)` function.

```
for all numeric a, b
such that b != 0
divide(a, b) * b == a
```

The advantage of property-based testing over unit testing is that assertions are no longer made about *specific* inputs and outputs. Instead, assertions are made about *all* outputs given *any* input that satisfies the property's conditions. Then, instead of manually checking that a property holds for some inputs, an assertion is run against the function for thousands of randomly generated inputs. And, because Haskell functions are pure, there is no need to set up and tear down the correct state and environment for each input.

C. Parser Testing with QuickCheck

The popular Haskell library QuickCheck facilitates this random, repeated input generation and output assertion for property-based tests [14]. In the hson codebase, QuickCheck tests the parser, ensuring a positive answer to the question, "Given any valid string expression in the hson language, does the hson parser generate the appropriate parse tree?"

Verifying this behavior with property-based testing requires generating arbitrary valid expressions for the parser. In hson, this random string generation is accomplished by generating arbitrary valid parse trees and *pretty-printing* them. In this case, pretty-printing refers to the inverse of parsing: instead of constructing a parse tree from an input program string, a pretty-printer maps a parse tree to a program string that could have generated it [33].

The hson codebase pretty-prints parse trees with the `pretty` package [34, 35]. The following code snippet contains functions responsible for pretty-printing hson expressions.

```
prettyPrintExpr :: Expr -> T.Text
prettyPrintExpr = prettyPrint ppExpr


ppExpr :: Expr -> Doc
ppExpr (ArrayInitializerExpr (ArrayInitializer _ elems)) =
  brackets $ commaSep $ map ppExpr elems
ppExpr (ArrowFunctionExpr (ArrowFunction params body)) =
  pipes (commaSep $ map ppTok params)
    <+> text "=>"
    <+> ppExpr body
ppExpr (BinaryExpr (Binary l op r)) =
  ppExpr l <+> ppTok op <+> ppExpr r
ppExpr (CallExpr (Call callee _ args)) =
  ppExpr callee <> parens (commaSep $ map ppExpr args)
ppExpr (ConditionalExpr (Conditional cond matched unmatched)) =
  ppExpr cond
    <+> char '?'
    <+> ppExpr matched
    <+> char ':'
    <+> ppExpr unmatched
ppExpr (DollarExpr (Dollar tok)) = ppTok tok
ppExpr (GetExpr (Get obj prop)) =
  ppExpr obj <> char '.' <> ppTok prop
ppExpr (GroupingExpr (Grouping expr)) =
  parens $ ppExpr expr
ppExpr (IndexExpr (Index indexed _ index)) =
  ppExpr indexed <> brackets (ppExpr index)
ppExpr (LiteralExpr (Literal tok)) = ppTok tok
ppExpr (LogicalExpr (Logical l op r)) =
  ppExpr l <+> ppTok op <+> ppExpr r
ppExpr (ObjectInitializerExpr (ObjectInitializer _ entries)) =
  ppObjectLiteral entries
ppExpr (UnaryExpr (Unary op r)) = ppTok op <> ppExpr r
ppExpr (VariableExpr (Variable name)) = ppTok name
```

With a pretty-printer implemented for hson expressions, property-based testing can be applied to the expression parser. The approach taken in hson is to generate an arbitrary valid parse tree, pretty-print it, and then parse the pretty-printed program [36]. If the pretty-printer is correct and the original arbitrary parse tree is valid, then the parser is correct if the output parse tree matches the input parse tree. The property can be expressed as follows.

```
for all parse trees a
such that a is valid
parse(prettyPrint(a)) == a
```

This property asserts that two functions, namely `parse` and `prettyPrint`, are inverses of each other. These "round-trip properties" are popular targets for property-based testing thanks to their succinct form [37]. The property's "valid" condition requires that the given parse tree is "possible," or that some program generates it. In practice, the "valid" condition means, for example, that the parse tree follows the precedence rules of the language. Expressions with lower precedence are never direct children of expressions with higher precedence; binary expressions representing addition should never be children of binary expressions representing multiplication.

Encoding this test in Haskell requires only a function that maps an input `Expr` value to a `Bool` test result. The type `Expr` represents a node in the hson expression parse tree, so the `checkExpressionParser` test takes a parse tree as input and produces `True` if the test passes and `False` otherwise.

```
checkExpressionParser :: Expr -> Bool
checkExpressionParser ast =
  case runHSONExprParser (prettyPrintExpr ast) of
    Left _ -> False
    Right a -> ast == a
```

The `runHSONExprParser (prettyPrintExpr ast)` operation corresponds to `parse(prettyPrint(a))` in the original formulation of the property; it's responsible for pretty-printing and parsing the input `Expr` tree. If the parser produces an error, the test returns `False`. Otherwise, the test asserts that the resulting parse tree `a` matches the input parse tree `ast`, returning `True` if so and `False` otherwise.

The test runs with a call to `quickCheck checkExpressionParser`, but QuickCheck first requires a definition of the `Expr` type as an instance of the `Arbitrary` typeclass. Declaring an `Arbitrary` instance entails defining how QuickCheck should generate arbitrary values for the given data type, which is accomplished by defining an operation `arbitrary`. The `arbitrary` definition for the `Expr` type requires encoding the precedence rules for operations to eliminate the possibility of generating invalid parse trees. The instance declaration also requires *sizing* rules to prevent QuickCheck from generating infinitely large parse trees. Otherwise, parse tree nodes like `ArrayInitializerExpr` that can have an arbitrarily large number of children may blow up and grow forever. Fortunately, QuickCheck provides the `sized` and `resize` functions that constrain the depth of the generated parse tree. The primary expression generator in hson utilizes the `sized` and `resized` functions to halve the size parameter every time a recursive node like `ArrayInitializerExpr` is generated [14].

```haskell
primaryExprGenerator :: Gen Expr
primaryExprGenerator =
  QC.oneof
    [ LiteralExpr <$> arbitrary
    , DollarExpr <$> arbitrary
    , VariableExpr <$> arbitrary
    , GroupingExpr
        <$> QC.sized (\n -> QC.resize (n `div` 2) arbitrary)
    , ArrayInitializerExpr
        <$> QC.sized (\n -> QC.resize (n `div` 2) arbitrary)
    , ObjectInitializerExpr
        <$> QC.sized (\n -> QC.resize (n `div` 2) arbitrary)
    ]
```

D. Interpreting Property-Based Test Results

Besides the errors that arose from defining the `Arbitrary` instance for `Expr`, implementing property-based testing for the hson parser presented two other error types.

1) The pretty-printer implementation did not correctly convert a parse tree to a string program (e.g., it printed the contents of a string literal without surrounding it with quotes).
2) The parser did not correctly parse a generated input.

The property test has three sources of failure in total: the parse tree generator, the pretty-printer, and the parser. Discerning the source of a particular error from among these three possibilities is straightforward.

1) An invalid parse tree indicates an error with the parse tree generator and, more specifically, the `Arbitrary` instance definition.
2) An unexpected program string given a valid parse tree (e.g., a string literal expression printed without surrounding quotes) indicates a bug with the pretty-printer.
3) If the parse tree and pretty-printed program are correct, an error has occurred within the parser.

The first two error cases represent unintended behavior within the test setup and are undesirable. The third error type, however, implies that a test has successfully identified a bug by generating an unhandled edge case. The property test helped identify several bugs related to precedence and ambiguity this way. The following test output concerning a `GetExpr` is a notable example of a successful test.

```
*** Failed! Falsified (after 9993 tests and 3 shrinks):
GetExpr
  ( Get
      { object =
          VariableExpr
            ( Variable
                { varName =
                    Token
                      { tokenType = TokenIdentifier
                      , literal = Just leaf
                      }
                }
            )
      , property =
          Token
            { tokenType = TokenIdentifier
            , literal = Just let
            }
      }
  )
```

The QuickCheck test output reveals that, after running 9993 randomly generated tests, it identified a counterexample that falsifies the target property. The output also includes the defective input, which QuickCheck produced after three "shrinks." A `shrink` is an operation performed by QuickCheck when it discovers a counterexample—a generated input that leads to a failed assertion. The goal of `shrink` is to produce the smallest similar counterexample that also falsifies the property [38]. Implementing `shrink` is an optional aspect of declaring an `Arbitrary` instance and entails defining a function `shrink :: a -> [a]` that returns a list of simpler values from a given generated value. The `Arbitrary` instance declaration for `Expr` contains both `arbitrary` and `shrink` function definitions.

```haskell
instance Arbitrary Expr where
  arbitrary =
    QC.oneof
      [ ArrayInitializerExpr <$> arbitrary
      , ArrowFunctionExpr <$> arbitrary
      , BinaryExpr <$> arbitrary
      , CallExpr <$> arbitrary
      , ConditionalExpr <$> arbitrary
      , DollarExpr <$> arbitrary
      , GetExpr <$> arbitrary
      , GroupingExpr <$> arbitrary
      , IndexExpr <$> arbitrary
      , LiteralExpr <$> arbitrary
      , LogicalExpr <$> arbitrary
      , ObjectInitializerExpr <$> arbitrary
      , UnaryExpr <$> arbitrary
      , VariableExpr <$> arbitrary
      ]
  shrink
    (ArrayInitializerExpr (ArrayInitializer tok elems)) =
      exprLeaves
        ++ elems
        ++ [ ArrayInitializerExpr (ArrayInitializer tok elems')
           | elems' <- QC.shrink elems
           ]
  shrink (ArrowFunctionExpr (ArrowFunction params body)) =
    exprLeaves
      ++ [body]
      ++ [ ArrowFunctionExpr (ArrowFunction params body')
         | body' <- QC.shrink body
         ]
  shrink (BinaryExpr (Binary l tok r)) =
    exprLeaves
      ++ [l, r]
      ++ [ BinaryExpr (Binary l' tok r')
         | (l', r') <-
```

```
          QC.shrink (l, r)
        ]
shrink (CallExpr (Call callee tok args)) =
  exprLeaves
    ++ (callee : args)
    ++ [ CallExpr (Call callee' tok args')
       | (callee', args') <- QC.shrink (callee, args)
       ]
shrink
  ( ConditionalExpr
      (Conditional cond matched unmatched)
    ) =
    exprLeaves
      ++ [cond, matched, unmatched]
      ++ [ ConditionalExpr
           ( Conditional
               cond'
               matched'
               unmatched'
           )
         | (cond', matched', unmatched') <-
             QC.shrink (cond, matched, unmatched)
         ]
shrink (DollarExpr (Dollar _)) = []
shrink (GetExpr (Get obj tok)) =
  exprLeaves
    ++ [obj]
    ++ [ GetExpr (Get obj' tok)
       | obj' <- QC.shrink obj
       ]
shrink (GroupingExpr (Grouping expr)) =
  exprLeaves
    ++ [expr]
    ++ [ GroupingExpr (Grouping expr')
       | expr' <- QC.shrink expr
       ]
shrink (IndexExpr (Index indexed tok index)) =
  exprLeaves
    ++ [indexed, index]
    ++ [ IndexExpr (Index indexed' tok index')
       | (indexed', index') <- QC.shrink (indexed, index)
       ]
shrink (LiteralExpr (Literal _)) = []
shrink (LogicalExpr (Logical l tok r)) =
  exprLeaves
    ++ [l, r]
    ++ [ LogicalExpr (Logical l' tok r')
       | (l', r') <- QC.shrink (l, r)
       ]
shrink
  (ObjectInitializerExpr (ObjectInitializer tok entries)) =
    exprLeaves
      ++ mapMaybe snd entries
      ++ [ ObjectInitializerExpr
           ( ObjectInitializer tok entries'
           )
         | entries' <- QC.shrink entries
         ]
shrink (UnaryExpr (Unary tok r)) =
  exprLeaves
    ++ [r]
    ++ [ UnaryExpr (Unary tok r')
       | r' <- QC.shrink r
       ]
shrink (VariableExpr (Variable _)) = []
```

When QuickCheck attempts to shrink a counterexample, it calls the appropriate `shrink` function to produce an ordered list of candidates, then tries to pick a simpler counterexample from that list [39]. If the `Arbitrary` instance declaration for a type does not define `shrink`, the function defaults to producing an empty list, and QuickCheck does not try to simplify counterexamples of that type. However, meticulously implementing `shrink` for complex recursive structures like `Expr` has a substantial payoff. Instead of producing a large parse tree, the test above produced a single `GetExpr` node counterexample, which can be pretty-printed to produce the following hson program.

```
leaf.let
```

The counterexample's simplicity helped immediately reveal the source of the error. The parser rejected `leaf.let` because the property `let` is a reserved word in hson, and the `identifier` parser combinator used for object properties fails when it encounters reserved words.

```
parseGet :: HSONParser (Expr -> Expr)
parseGet = do
  dot
  -- `identifier` fails if the parsed string is a reserved word
  property <- identifier
  return $
    \object -> GetExpr Get{object = object, property = property}
```

This behavior of the hson parser is unintentional and potentially disruptive. If an object contained the property `let`, a user could not access that property by writing `$.let` within hson.

The failed test also suggested that hson might reject reserved words that appear as properties in object literal expressions like the following:

```
{ false: true }
```

Indeed, when input to the hson parser, the program produced the following error.

```
unexpected reserved word "false"
expecting expression
```

Again, the use of the `identifier` parser combinator was the source of the error.

```
keyValue = do
  k <- try tokenString <|> identifier -- the culprit!
  colon
  v <- expression
  return (k, Just v)
```

In summary, although implementing property-based testing for the hson parser demanded significant setup effort, it revealed

several bugs and their sources through simple counterexamples. The bugs preventing reserved words from being defined and accessed as properties of objects, for example, would not have been identified without the thousands of random tests generated by QuickCheck. In general, each success of property-based testing represents an edge case that might not have been considered while writing unit tests.

VII. Conclusion

This paper has demonstrated how strongly typed pure functional programming languages can be applied practically to crafting safe and expressive code for building useful software. The techniques utilized within the hson implementation—parser combinators, applicative interfaces, monad transformers, and property-based testing—are likely novel to the common programmer as they stem from an approach fundamentally different than the mainstream imperative style. The goal of detailing these features has been to illustrate how the seemingly restrictive nature of types and purity is an advantage that, when wielded, promotes correctness, maintainability, and tractable program reasoning. In turn, this paper has sought to show that functional programming language features are not merely exciting for the sake of programming language research but also genuinely benefit those leveraging them in practical applications. The reader who finds these points interesting or compelling is encouraged to explore functional programming and experience the delight and confidence afforded by the guarantee that type errors, untamed side effects, and unexpected state mutations are made improbable.

The range of Haskell language features surveyed in this paper is not exhaustive and only begins to demonstrate the language's capabilities. A possible extension to this paper and the hson implementation is to employ compile-time meta-programming with Template Haskell [40]. Many code patterns are repeated throughout the implementation of hson, especially within the built-in function definitions (see Section V-B), and could be abstracted into compile-time code generation. This work could also be extended to demonstrate how strong typing and purity simplify writing concurrent programs [7]. Specifically, the hson parser and interpreter implementations could be augmented to leverage Concurrent Haskell and Software Transactional Memory for modular concurrency that preserves correctness [41, 42].

Appendix A
hson Language Grammar

⟨*program*⟩ ::= {⟨*declaration*⟩} ⟨*expr*⟩

⟨*declaration*⟩ ::= `let` ⟨*binding*⟩ `=` ⟨*expr*⟩`;`

⟨*binding*⟩ ::= ⟨*ident*⟩
| ⟨*object-dest*⟩
| ⟨*array-dest*⟩

⟨*object-destructure*⟩ ::= `{` [⟨*object-destructure-entry*⟩{`,` ⟨*object-destructure-entry*⟩ }] `}`

⟨*object-destructure-entry*⟩ ::= ⟨*ident*⟩[`:` ⟨*ident*⟩]
| ⟨*string-literal*⟩`:` ⟨*ident*⟩

⟨*array-destructure*⟩ ::= `[` [⟨*ident*⟩{`,` ⟨*ident*⟩}] `]`

⟨*expr*⟩ ::= ⟨*arrow-function*⟩

⟨*arrow-function*⟩ ::= `|` [⟨*ident*⟩{`,` ⟨*ident*⟩}] `|` `=>` ⟨*expr*⟩
| ⟨*pipe-forward*⟩

⟨*pipe-forward*⟩ ::= ⟨*ternary*⟩ { `|>` ⟨*call*⟩ }

⟨*ternary*⟩ ::= ⟨*null-coalesce*⟩ [`?` ⟨*expr*⟩ `:` ⟨*expr*⟩]

⟨*null-coalesce*⟩ ::= ⟨*logical-or*⟩ { `??` ⟨*logical-or*⟩ }

⟨*logical-or*⟩ ::= ⟨*logical-and*⟩ { `||` ⟨*logical-and*⟩ }

⟨*logical-and*⟩ ::= ⟨*equality*⟩ { `&&` ⟨*equality*⟩ }

⟨*equality*⟩ ::= ⟨*comparison*⟩ { (`!=` | `==`) ⟨*comparison*⟩ }

⟨*comparison*⟩ ::= ⟨*term*⟩ { (`>=` | `>` | `<=` | `<`) ⟨*term*⟩ }

⟨*term*⟩ ::= ⟨*factor*⟩ { (`+` | `-`) ⟨*factor*⟩ }

⟨*factor*⟩ ::= ⟨*unary*⟩ { (`*` | `*`) ⟨*unary*⟩ }

⟨*unary*⟩ ::= { (`!!` | `!` | `-`) } ⟨*unary*⟩
| ⟨*call*⟩

⟨*call*⟩ ::= ⟨*primary*⟩ { (`(` ⟨*arguments*⟩ `)` | `[` ⟨*expr*⟩ `]` | `.`⟨*ident*⟩) }

⟨*primary*⟩ ::= `$`
| `true`
| `false`
| `null`
| ⟨*ident*⟩
| ⟨*array*⟩
| ⟨*object*⟩
| ⟨*string-literal*⟩
| ⟨*number-literal*⟩
| `(` ⟨*expr*⟩ `)`

⟨*ident*⟩ ::= (⟨*letter*⟩ | `_`) { ⟨*alphaNum*⟩ | `_` | `'` }

⟨*array*⟩ ::= `[` ⟨*arguments*⟩ `]`

⟨*arguments*⟩ ::= [⟨*expr*⟩{`,` ⟨*expr*⟩}]

⟨*object*⟩ ::= `{` [⟨*object-entry*⟩{`,` ⟨*object-entry*⟩ }] `}`

⟨*object-entry*⟩ ::= (⟨*string-literal*⟩ | ⟨*ident*⟩) `:` ⟨*expr*⟩
| ⟨*ident*⟩

⟨*string-literal*⟩ ::= As described in the Haskell Report [43].

⟨*number-literal*⟩ ::= ⟨*natural*⟩
| ⟨*float*⟩

⟨*natural*⟩ ::= As described in the Haskell Report [43].

⟨*float*⟩ ::= As described in the Haskell Report [43].

APPENDIX B
HSON REFERENCE

$ is the identifier to which the parsed JSON data is bound.

filter(array, predicate) returns a copy of `array` filtered to contain only the elements for which `predicate` returns `true`.

map(array, function) returns a copy of `array` in which each new element is the result of applying `function` to the original element.

toJSON(value, indent) returns a string containing the JSON representation of `value`. The optional `indent` parameter specifies the number of spaces of indentation in the resulting JSON string.

some(array, predicate) returns `true` if there exists some element in `array` for which `predicate` returns true.

reduce(array, function, initial) first applies a binary `function` to `initial` and the first element of `array`, then applies `function` to that result and the second element of `array`, and so on. Once `function` has been applied to every element in `array`, `reduce` returns the final accumulated result.

reverse(value) returns a copy of `value` with its elements in reverse order if `value` is an array. If `value` is a string, `reverse` returns a copy of `value` with its characters in reverse order.

length(value) returns the number of elements in `value` if `value` is an array. If `value` is a string, `length` returns the number of characters in `value`.

includes(haystack, needle) returns `true` if the string `haystack` contains the substring `needle` and returns `false` otherwise.

APPENDIX C
SPU HONORS RESEARCH SYMPOSIUM MAY 18, 2024
BRIDGING THE GAP: DECONSTRUCTING ACADEMIC ATTITUDES FOR ACCESSIBILITY

Hello everyone! My name is Parker Landon. I'm a computer science and applied math major, and the topic of my honors project is programming languages. Specifically, my research focuses on a particular programming language paradigm called "functional programming." This programming style has historically been confined to the margins of software development and used primarily in academia as a vehicle for programming language research. However, functional programming is hailed by its users for facilitating expressive, maintainable, and correct code. Thus, my research explores how functional programming can be applied as a practical tool for creating useful software. I used functional programming to implement a program called "hson," a scripting language and command-line interface for processing JSON data. Before I discuss my work, allow me to provide more context about programming language paradigms.

Programming languages provide the means of instructing computers to carry out tasks; they are the tools of computer programmers for expressing ideas and implementing programs. Even among the uninitiated, for whom computer programming seems like a black box, the names of popular programming languages are widely known. Perhaps you've heard of C, C++, Java, or JavaScript. Or Python, R, or Matlab. Or Swift, Kotlin, C#, Ruby, Go, Rust, Lisp, Perl, or, the topic of this research, Haskell. There are many programming languages, each offering different features that make them better suited for certain applications. The C and Go programming languages, for example, are simple languages with similar syntaxes. However, C programs are responsible for manual memory management, while Go programs handle memory management with garbage collection. Consequently, C is more commonly used in low-level hardware applications where fine-grained control over memory is necessary. Go, on the other hand, is a better choice for building reliable, resilient web servers.

Still, C and Go share a common style. Besides their similar syntaxes, programs in both languages are sequences of statements that transition the system from an initial state to a final state. This characteristic is the hallmark of the "imperative" programming style. Each statement in an imperative program may have the "side effect" of mutating the system's overall state. For example, an imperative program that computes the result of some number x raised to the nth power might read as follows. "Let i equal 0, and let s equal 1. Then, while i is less than n, update the value of s to be x times s, then increment i by 1. Return s as the result." Throughout this example program, the values associated with the variables i and s are mutated several times until s contains our desired result. However, even in this small, contrived example, the program would be very wrong if I had accidentally used "less than or equal" instead of "less than" or forgotten to increment i by one during each iteration. In this way, unintentional mutations or side effects can produce unexpected results that lead to incorrect programs. While these mistakes are somewhat easy to spot in small programs, a real program with many variables and functions that may or may not mutate those variables can be challenging to navigate and impossible to reason about.

Since the 1990s, a paradigm that aims to manage these side effects called object-oriented programming has dominated mainstream computer programming, and languages like Java, C++, and C# that embraced this style have remained among the most popular languages of the tech industry. Object-oriented programming entails structuring data within "objects" that possess properties and behaviors defined by that object's "class." A class is a blueprint for objects: it describes their type, behaviors, and properties. The upshot of object-oriented programming is that it attempts to contain the complexity of imperative programs by *encapsulating* state and side effects within objects and abstracting them away with a defined interface. A `Rectangle` class, for example, might encapsulate the `length` and `width` properties of a rectangle and expose an interface for computing its area. This way, mutating a rectangle's length and width properties is confined to the behaviors of the `Rectangle` class, so the source of "rectangle errors" can be more easily

identified as occurring within the `Rectangle` class instead of "somewhere in the system."

Still, state mutation represents only one kind of side effect. Statements within an imperative program can also print text, read from a file, query a database, execute other programs, send emails, and launch missiles. Typically, the imperative style allows these side effects to occur anywhere in the program, often without any indication or warning. While this liberty to produce side effects may feel convenient when writing a program, it introduces more possibilities for creating incorrect programs with unexpected results. In this way, it appears that programming languages must compromise between correctness and convenience. The object-oriented style, while attempting to manage the complexity of state and mutations, still emphasizes convenience, allowing unrestricted side effects but encapsulating them within classes.

Meanwhile, throughout this object-oriented tidal wave, a marginalized world of researchers and hobbyists has enjoyed and praised a different style, namely functional programming. In contrast to object-oriented programs, functional programs heavily emphasize correctness, expressiveness, and elegant correspondence to formal mathematical models. According to the functional programming style, a program is a composition of functions that each produce an output given an input. Rather than performing computations as sequences of state transitions, functional programs compute by applying functions to arguments. Of course, the imperative programming style may utilize functions, too, but the functional programming style distinguishes itself by promoting *purity*.

A function is said to be *pure* if its execution produces no side effects and its output depends on nothing besides its input. In other words, the result of a pure function is wholly determined by its input; it will always produce the same output given the same input. Thus, the behavior of a pure function is captured entirely by its definition: its result neither mutates nor is influenced by any program state.

The upshot of purity is that it eliminates the possibility of unwanted side effects by doing away with side effects altogether. While this limitation may seem severe, it affords several benefits that promote correctness. For example, a functional program is easy to reason about. Because a pure function's behavior is captured entirely by its definition, there is never a concern for how a function operates in the broader context of a stateful program. For the same reasons, pure functions are typically trivial to test and are much easier to parallelize.

Still, it may seem that there exists a fatal problem with the purity restriction: a program composed solely of pure functions is useless. Necessary operations like receiving inputs and displaying outputs are side effects, so they cannot be performed in a pure context. Fortunately, programming languages that enforce purity and the functional style—namely "pure functional programming languages"—offer various methods for integrating and encapsulating side effects. The pure functional programming language considered in this work, Haskell, facilitates effectful programming through *monads,* which encode impure functions and create a boundary distinguishing effectful operations from pure ones. Monads are a part of Haskell's rich *static type-checking system,* another feature like purity that aims to promote correctness. Static type-checking ensures that a written program contains no errors concerning the types of values, like adding a number to a string of characters. Although static type-checking cannot eliminate *all* unexpected or undesired behaviors, it prevents the programmer from introducing a common class of errors.

So, functional programming leverages "restrictions" like purity and static typing to promote correctness and facilitate tractable program reasoning. These "restrictions," in practice, actually feel liberating as they encourage elegant solutions and greater developer confidence. Yet, strongly typed pure functional programming languages still enjoy significantly less *practical application*—for example, in the software engineering industry and commercial world—than their mainstream, imperative counterparts. Yes, functional programming has greatly influenced these domains, as functional language features like pattern matching, generics, type inference, and first-class functions have been adopted by and become central to the most widely-used programming languages today; however, the functional programming languages themselves have primarily remained tools of researchers and hobbyists. Indeed, the measurable influence of functional features is a testament to the work of these users; though, as researchers have acknowledged, there exists a tension between applying a language to building useful systems and using that language to drive programming language research innovations.

Still, despite the traditional origins of functional programming in academic research, those initiated to the style tend to advocate heartily for its fitness in practical settings. Motivated by an interest in this sentiment, I settled on pursuing the following question in my research: How could a pure functional programming language be applied practically to creating useful software? I sought to answer this question by first learning a pure functional programming language, familiarizing myself with the functional style, and then using it to build a program that affords tangible benefits.

The programming language I set out to learn is called Haskell. Haskell is notorious among software engineers for its strangeness and supposedly steep learning curve. Its adherence to the functional paradigm and explicit correspondence to a relatively obscure field of mathematics called "category theory" lead many to avoid it. However, Haskell's historical commitment to purity and strong typing made it the perfect subject of this research, and its nearly legendary status as esoteric supplemented my intrigue.

Admittedly, Haskell was challenging to learn. The novelty of the functional style and the intricacy of Haskell's type system made for slow initial progress. However, the intimidating world of Haskell and functional programming eventually grew

comfortable, and critical concepts like functors and monads felt less like terrors to be dreaded and more like tools to be wielded. At this point, I was confident I could begin working on a software project to demonstrate the efficacy and practicality of pure functional programming. This project, which I'm calling "hson," is a command-line program for processing data in the JSON format. JSON is a pervasive data format in contemporary software engineering, finding use in applications like configuration files and data interchange. Consequently, reading and working with the contents of JSON data is a common operation within the terminal. Hence, the hson program seeks to provide a simple and efficient command-line interface for processing and manipulating that data.

The hson program takes two inputs—the JSON data to be processed and a script in the hson language—and processes the JSON data according to the hson script. For example, suppose you had a web response that contained a JSON-formatted list of restaurant information. You could pipe that data into the hson program alongside an hson script that filters the list for restaurants in Seattle, and hson would produce the appropriately filtered list.

As suggested, an hson script comprises code in the hson language, which I designed and implemented for this project. The syntax and semantics of the hson language draw much inspiration from JavaScript to preserve familiarity. Some of hson's features include destructuring assignment, anonymous functions, the dot and bracket notations for property access, the pipe operator, and built-in higher-order functions like `map`, `filter`, and `reduce`. The ability to manipulate the input JSON data from within hson is enabled by the dollar sign symbol, which is bound to the parsed JSON data before the script is executed.

At a high level, an hson script is a sequence of zero or more variable declarations followed by a single expression. The output of a script is the result of the evaluated final expression. Variables in hson are immutable: they cannot be reassigned after their declaration, and their values cannot change. Computations are performed solely through sequencing and composing functions, so hson is, fittingly, a functional language.

Creating a scripting language like hson entails implementing two key components: a *parser,* which turns text into lexical tokens and structures those tokens into a *parse tree* according to the grammar rules of the language; and an *interpreter,* which traverses the parse tree produced by the parser and executes the appropriate code. The hson program is also responsible for parsing the input JSON data, parsing other command-line options, and reporting any syntax or runtime errors that occur. Finally, hson leverages property-based testing to ensure the correctness of the hson parser. To summarize, the hson program has three sweeping responsibilities: parsing, interpreting, and testing. My research demonstrates how Haskell and pure functional programming can deliver these features while facilitating expressive, maintainable, and correct code in the process.

Parsing, the first of hson's sweeping responsibilities, encompasses three distinct tasks: parsing the JSON data, parsing the hson script, and parsing the command-line options. Each of these tasks is implemented using a different technique. The JSON parsing task has the simplest implementation, as it leverages an existing JSON parsing library and Haskell's *type class* feature to define how each JSON value should be converted into hson.

The hson script parser, in contrast, leverages higher-order functions called "parser combinators" to sequence and compose parsers. A parser combinator takes another parser as input and produces a new, composite parser as output. For example, a parser responsible for parsing variable declarations can be composed with the `many` parser combinator to produce a new parser that handles "many variable declarations." The results of these parser computations are contained within the `Parser` monad, which defines and manages sequencing, failure, choice, and error-handling operations. The advantage of leveraging monadic parser combinators is that one can quickly encode an entire language grammar by composing and sequencing a relatively small collection of elementary combinators. Moreover, the declarative style of parser combinators results in parser code that *looks* like the associated grammar, with each function corresponding to a variable and each definition corresponding to the related production. Additionally, because parser combinators are utilized within Haskell, the parser implementation has access to Haskell's many features, and the parser results can be easily integrated with the rest of the Haskell program.

Finally, the command-line option parser leverages an applicative parsing interface that, like monadic parser combinators, affords operations for sequencing and composing, but is aptly more concise for the simpler task.

The second responsibility of hson, interpreting, utilizes *pattern-matching* to descend the parse tree and evaluate each node appropriately. Pattern-matching is a Haskell feature that allows a function to be redefined for various input types. Then, when the function is called with a particular value, Haskell calls the first function whose input pattern matches that value's type. In the case of interpreting, an `evaluate` function can be defined for each expression type, and Haskell will handle calling the appropriate definition of `evaluate` for each parse tree node. Pattern-matching is also used within the hson interpreter to implement built-in functions like `map` and `reduce`. Each built-in function definition follows the similar, readable pattern of pattern-matching over the argument types to execute the appropriate operations on valid arguments and throw proper errors for invalid arguments. Finally, the hson interpreter utilizes monad transformers, which enable monad features to be combined into a single new monad through composition. Specifically, interpreter results are contained within a custom `Eval` monad that comprises more fundamental

monads, introducing features like a read-only environment for variable bindings, operations for error handling, and the ability to produce input/output side effects. In sum, these Haskell features facilitate an intepreter implementation that is modular, expressive, and extensible.

Finally, hson leverages property-based testing to ensure the correctness of its parser. Property-based testing is a robust technique that is especially well-suited for functional programming and deviates from a more typical style of testing called "unit testing." In a unit test, a target software component is isolated from the rest of the system, and the tester defines and controls its input. Essentially, a unit test seeks to answer the question, "Given a specific input, does the unit of code produce the expected output or side effects?" While unit tests can bolster confidence about how the code handles certain kinds of inputs, it is hopeless to test every possible input, so the code may still be susceptible to unconsidered edge cases. Property-based testing mitigates this risk by generating thousands of random inputs and asserting that a particular property holds for a unit of code given those inputs. Then, instead of asserting about *specific* inputs and outputs, property-based tests assert about *all* outputs given *any* input that satisfies the property's conditions. For the hson parser, property-based testing revealed several bugs and their sources through simple counterexamples to the tested properties, identifying edge cases that may have otherwise been missed.

As mentioned, pure functional programming languages have primarily remained tools of academia. Indeed, Haskell has developed a reputation for being the language of academic white papers, not real software applications. Furthermore, Haskell's esoteric status and adherence to formal theory have bolstered a widespread allegation that it's indecipherable by mere mortals and lowly engineers. However, through my past year's journey, I've discovered that learning Haskell is both an attainable and rewarding feat for non-academic engineers. Not only is Haskell rich enough to build practical software, as I've demonstrated in implementing hson, but it also constitutes the origin of many modern programming language features. Mainstream languages are gradually adopting functional constructs and robust type systems, so the skills acquired by learning Haskell are more transferable than ever.

Recently, several promising technologies have fully embraced the functional paradigm. The Nix package manager for Linux enables a declarative approach to system configuration with its purely functional configuration language. More recently, the version one release of the strongly typed functional language Gleam has impressed the software engineering community with its approach to marrying simplicity with expressiveness and fault-tolerant concurrency.

To conclude, I'll address the apparent problem that my effort to bring Haskell down from the ivory tower of academia and to the practical masses has culminated in my producing an academic paper and presenting at an academic research symposium. Researching programming languages has made it abundantly clear that academic papers and computer science scholarship are not a means by which engineers consume information or communicate. The primary forums for discussing software engineering are Twitter, YouTube, GitHub, and blogs: platforms that encourage brevity and facilitate low barriers to entry. Recognizing this, I've made an effort throughout my research to share learnings and condensed versions of my work in several blog posts and tweets, which are all made available on my website *parkerlandon.com*.

Haskell is a remarkable manifestation of theory colliding with practice, and its many features offer all programmers something to enjoy and learn from. Pure functional programming languages are robust tools for crafting expressive, maintainable, and correct code, and they need not be limited to the bounds of programming language research. Understanding problems, expressing ideas, and implementing solutions are part of what it means to be human, and the pure functional style offers programmers an elegant means of accomplishing these duties within the domain of software engineering. If nothing else, I hope my work encourages another curious programmer like myself to try functional programming and discover the *fun* of programming. Thank you.

REFERENCES

[1] D. Syme, "The early history of F#," in *2020 Proceedings of the ACM on Programming Languages*, Jun. 2020. [Online]. Available: https://www.microsoft.com/en-us/research/publication/the-early-history-of-f/

[2] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," in *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, ser. OOPSLA '86. New York, NY, USA: Association for Computing Machinery, 1986, p. 38–45. [Online]. Available: https://doi.org/10.1145/28697.28702

[3] A. T. Cohen, "Data abstraction, data encapsulation and object-oriented programming," *SIGPLAN Not.*, vol. 19, no. 1, p. 31–35, jan 1984. [Online]. Available: https://doi.org/10.1145/948415.948418

[4] Z. Hu, J. Hughes, and M. Wang, "How functional programming mattered," *National Science Review*, vol. 2, no. 3, pp. 349–370, 07 2015. [Online]. Available: https://doi.org/10.1093/nsr/nwv042

[5] J. Backus, "Can programming be liberated from the von neumann style? a functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 8, p. 613–641, aug 1978. [Online]. Available: https://doi.org/10.1145/359576.359579

[6] J. Hughes, "Why Functional Programming Matters," *The Computer Journal*, vol. 32, no. 2, pp. 98–107, 01 1989. [Online]. Available: https://doi.org/10.1093/comjnl/32.2.98

[7] K. Hammond, "Why Parallel Functional Programming Matters: Panel Statement," in *Reliable Software Technologies - Ada-Europe 2011*, A. Romanovsky and T. Vardanega, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 201–205.

[8] P. Wadler, "The essence of functional programming," in *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '92. New York, NY, USA: Association for Computing Machinery, 1992, p. 1–14. [Online]. Available: https://doi.org/10.1145/143165.143169

[9] B. C. Pierce, *Types and programming languages*. MIT press, 2002.

[10] E. Meijer and P. Drayton, "Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages," 01 2004.

[11] J. Ousterhout, "Scripting: higher level programming for the 21st century," *Computer*, vol. 31, no. 3, pp. 23–30, 1998.

[12] P. Wadler, "Why no one uses functional languages," *ACM Sigplan Notices*, vol. 33, no. 8, pp. 23–27, 1998.

[13] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of haskell: being lazy with class," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007, pp. 12–1.

[14] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," *SIGPLAN Not.*, vol. 35, no. 9, p. 268–279, sep 2000. [Online]. Available: https://doi.org/10.1145/357766.351266
[15] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler, "Type classes in haskell," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 2, p. 109–138, mar 1996. [Online]. Available: https://doi.org/10.1145/227699.227700
[16] G. Hutton, *Programming in haskell*. Cambridge University Press, 2016.
[17] P. Hudak and J. H. Fasel, "A gentle introduction to haskell," *ACM Sigplan Notices*, vol. 27, no. 5, pp. 1–52, 1992.
[18] R. Paterson, "Constructing applicative functors," in *International Conference on Mathematics of Program Construction*. Springer, 2012, pp. 300–323.
[19] M. P. Jones and L. Duponcheel, "Composing monads," Technical Report YALEU/DCS/RR-1004, Department of Computer Science. Yale . . . , Tech. Rep., 1993.
[20] M. P. Jones, "Functional programming with overloading and higher-order polymorphism," in *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1*. Springer, 1995, pp. 97–136.
[21] M. Grabmuller, "Monad transformers step by step," 01 2006.
[22] D. Leijen and E. Meijer, "Parsec: Direct style monadic parser combinators for the real world," Tech. Rep. UU-CS-2001-27, July 2001, user Modeling 2007, 11th International Conference, UM 2007, Corfu, Greece, June 25-29, 2007. [Online]. Available: https://www.microsoft.com/en-us/research/publication/parsec-direct-style-monadic-parser-combinators-for-the-real-world/
[23] B. O'Sullivan, "text: An efficient packed unicode text type." 2024. [Online]. Available: https://hackage.haskell.org/package/text
[24] D. Leijen, P. Martini, and A. Latter, "parsec: Monadic parser combinators," 2023. [Online]. Available: https://hackage.haskell.org/package/parsec
[25] B. O'Sullivan, "aeson: Fast json parsing and encoding," 2023. [Online]. Available: https://hackage.haskell.org/package/aeson
[26] P. Capriotti and H. Campbell, "optparse-applicative: Utilities and combinators for parsing command line options," 2023. [Online]. Available: https://hackage.haskell.org/package/optparse-applicative
[27] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 333–343. [Online]. Available: https://doi.org/10.1145/199448.199528
[28] G. Team, "Ghc user's guide documentation," 2023. [Online]. Available: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/index.html
[29] R. Nystrom, *Crafting interpreters*. Genever Benning, 2021.
[30] J. Whittaker, "What is software testing? and why is it so hard?" *IEEE Software*, vol. 17, no. 1, pp. 70–79, 2000.
[31] D. D. Ma'ayan, "The quality of junit tests: an empirical study report," in *Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies*, ser. SQUADE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 33–36. [Online]. Available: https://doi.org/10.1145/3194095.3194102
[32] P. Runeson, "A survey of unit testing practices," *IEEE Software*, vol. 23, no. 4, pp. 22–29, 2006.
[33] P. Wadler, "A prettier printer," *The Fun of Programming, Cornerstones of Computing*, pp. 223–243, 2003.
[34] J. Hughes, "The design of a pretty-printing library," 01 2006, pp. 53–96.
[35] D. Terei, "pretty: Pretty-printing library," 2018. [Online]. Available: https://hackage.haskell.org/package/pretty-1.1.3.6
[36] "Parsec parser testing with quickcheck," 2007. [Online]. Available: https://lstephen.wordpress.com/2007/07/29/parsec-parser-testing-with-quickcheck/
[37] H. Goldstein, J. W. Cutler, D. Dickstein, B. C. Pierce, and A. Head, "Property-based testing in practice," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 971–971.
[38] H. Goldstein, S. Frohlich, M. Wang, and B. C. Pierce, "Reflecting on random generation," *Proc. ACM Program. Lang.*, vol. 7, no. ICFP, aug 2023. [Online]. Available: https://doi.org/10.1145/3607842
[39] J. Hughes, "Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane," in *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, S. Lindley, C. McBride, P. Trinder, and D. Sannella, Eds. Cham: Springer International Publishing, 2016, pp. 169–186. [Online]. Available: https://doi.org/10.1007/978-3-319-30936-1_9
[40] T. Sheard and S. P. Jones, "Template meta-programming for haskell," in *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 1–16. [Online]. Available: https://doi.org/10.1145/581690.581691
[41] S. P. Jones, A. Gordon, and S. Finne, "Concurrent haskell," in *POPL*, vol. 96, 1996, pp. 295–308.
[42] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 48–60. [Online]. Available: https://doi.org/10.1145/1065944.1065952
[43] S. Marlow *et al.*, "Haskell 2010 language report," 2010.

All links were last followed on May 31, 2024.