Universidad Nacional Autónoma de México
Facultad de Ciencias

2024 1

COMPILADORES

Profesor
Manuel Soto Romero

Ayudantes
Braulio Aaron Santiago Carrillo
Javier Enríquez Mendoza
Pedro Ulises Cervantes González

MÓNADAS EN HASKELL: PODER EN POCAS PALABRAS

Integrantes

Zurisadai Uribe García
318223197

Ulises Rodríguez García
318042202

Javier Alejandro Rivera Zavala
311288876

Diciembre 22, 2023

Índice

1. Introducción

El análisis sintáctico, un componente crítico en el campo de los compiladores y el procesamiento de lenguajes de programación, ha sido tradicionalmente un área desafiante debido a su complejidad inherente y la necesidad de manejar eficazmente los estados y los efectos secundarios. En el mundo de la programación funcional, las mónadas emergen como una herramienta potente para abordar estos retos. Este documento se enfoca en explorar el uso de las mónadas en el análisis sintáctico, particularmente en Haskell, un lenguaje de programación funcional puro.

Este documento se propone brindar una visión comprensiva de las mónadas en programación funcional con un enfoque en Haskell, ilustrando cómo pueden ser aplicadas para construir analizadores sintácticos robustos y flexibles. Se examinan diversas implementaciones de analizadores sintácticos para un lenguaje de expresiones aritméticas simples, demostrando la aplicación práctica de las mónadas en diferentes contextos y comparando su eficacia con métodos de análisis sintáctico tradicionales.

A través de esta exploración, se busca no solo entender mejor el papel de las mónadas en la programación funcional, sino también apreciar su potencial para transformar la manera en que abordamos el análisis sintáctico en la construcción de compiladores y otras herramientas de procesamiento de lenguajes. Con este entendimiento, los desarrolladores pueden aprovechar las mónadas para crear soluciones más expresivas, modulares y mantenibles, abriendo así nuevas posibilidades en el campo del procesamiento de lenguajes.

2. Resumen

Este documento pretende proporcionar una visión general y detallada en medida de lo posible, de las mónadas en programación funcional, haciendo enfásis en su implementación en Haskell. Comenzamos con una introducción general a las mónadas, detallando sus conceptos clave como contenedores computacionales, encadenamiento de operaciones y control de efectos secundarios. Luego de lo anterior, presentamos como se definen en Haskell, incluyendo las leyes que rigen su comportamiento y cómo se implementan mediante la clase de tipo `Monad`.

Posteriormente, abordamos el uso de mónadas para el manejo de efectos secundarios y la estructuración de programas. Hablamos de aspectos como la separación de lógica pura e impura, la composición de operaciones complejas, y la utilización de mónadas para abstraer distintos comportamientos computacionales.

Nuestro proyecto también incluye ejemplos prácticos, mostrando cómo algunas de las mónadas más comunes (`Maybe`, `IO`, `List`) se utilizan en Haskell para realizar distintos tipos de operaciones.

En la sección siguiente abordamos aspectos generales del análisis sintáctico monádico, que es una técnica avanzada en la construcción de analizadores y compiladores. Tratamos de explicar cómo las mónadas facilitan la gestión del análisis de lenguajes de programación, para ello mostramos detalles sobre su importancia en compiladores, la flexibilidad y abstracción que ofrecen, y ejemplos de aplicación en proyectos reales.

Luego de lo anterior, presentamos una comparación del análisis sintáctico monádico con métodos tradicionales de análisis sintáctico, destacando las diferencias en términos de flexibilidad, curva de aprendizaje, rendimiento y manejo de estado y errores.

Casi para finalizar, incluimos una descripción de las tres implementaciones distintas que elaboramos, éstas constan de un analizador sintáctico para un lenguaje de expresiones aritméticas sencillas, utilizando diferentes enfoques basados en mónadas. Detallamos las ventajas y desafíos de cada implementación, incluyendo la curva de aprendizaje, rendimiento y manejo de errores.

Por último, presentamos una retrospectiva de los aprendizajes y desafíos enfrentados durante el desarrollo de estas implementaciones, resaltamos la potencia y modularidad de las mónadas en Haskell, y la utilidad de bibliotecas como Parsec y StateT en la construcción de analizadores sintácticos.

3. ¿Qué son las mónadas?

Las mónadas son uno de los conceptos más poderosos y, a veces, uno de los más desafiantes en la programación funcional. En lenguajes como Haskell, las mónadas proporcionan una forma elegante y expresiva de manejar operaciones secuenciales y efectos secundarios, manteniendo al mismo tiempo la pureza funcional del código. Su importancia no solo radica en su capacidad para manejar efectos, sino también en su contribución a la creación de código más modular, reutilizable y mantenible. A continuación, exploraremos en detalle qué son las mónadas, cómo funcionan en el contexto de la programación funcional y proporcionaremos ejemplos concretos para ilustrar su uso práctico.

3.1. Mónadas en programación funcional

Las mónadas en Haskell son una abstracción poderosa que permite trabajar con una variedad de operaciones y efectos de manera estructurada y controlada. Para profundizar en su definición, es esencial comprender algunos conceptos clave:

Conceptos clave de las mónadas

1. **Contenedor computacional:** Una mónada se puede visualizar como un contenedor que encapsula un valor o una computación. Este contenedor no solo guarda un valor sino que también describe cómo interactúa esa computación con otras.
2. **Encadenamiento de operaciones:** Las mónadas permiten la composición de funciones que operan sobre estos contenedores. Esto se hace usualmente mediante la función de enlace (bind), que secuencia operaciones manteniendo los contextos monádicos.
3. **Efectos secundarios controlados:** En programación funcional, los efectos secundarios son generalmente indeseados o deben ser manejados con cuidado. Las mónadas proporcionan un marco formal para manejar estos efectos (como la entrada/salida o el manejo de estados) sin comprometer la pureza funcional del código.

La definición de mónadas en Haskell

En Haskell, una mónada se define mediante una clase de tipo llamada Monad, que es parte de la biblioteca estándar. Esta clase de tipo especifica las operaciones fundamentales que deben ser implementadas por cualquier tipo que quiera comportarse como una mónada:

- **Función `return`:** Esta función toma un valor y lo envuelve en un contexto monádico. Es importante para iniciar una secuencia de operaciones monádicas.
- **Función $\gg=$ (bind):** Esta función es el núcleo de la mónada. Toma un valor monádico y una función que toma un valor normal y devuelve un valor monádico, y los combina en una nueva operación monádica. Esta función permite la secuencia de operaciones monádicas.

Leyes de las mónadas

Para que una instancia se considere una mónada en Haskell, debe cumplir con tres leyes que aseguran la consistencia y previsibilidad de las operaciones:

1. **Ley de la identidad a la izquierda:** `return a` $\gg=$ `f` es lo mismo que `f a`. Esto significa que si creamos un valor monádico con return y luego lo aplicamos a una función usando bind, es lo mismo que aplicar la función directamente al valor.
2. **Ley de la identidad a la derecha:** `m` $\gg=$ `return` es lo mismo que m. Esto significa que aplicar bind a un valor monádico y la función return no cambia el valor monádico.
3. **Ley de asociatividad:** `(m` $\gg=$ `f)` $\gg=$ `g` es lo mismo que `m` $\gg=$ `(\x` $\rightarrow$ `f x` $\gg=$ `g)`. Esto asegura que el orden en que se aplican las operaciones no afecta el resultado final.

Estas leyes y definiciones hacen de las mónadas una herramienta poderosa en Haskell, permitiendo estructurar programas de manera clara y eficiente, especialmente cuando se trata de efectos secundarios y operaciones que involucran tipos de datos complejos.

3.2. Efectos secundarios y estructuración de programas

El concepto de mónadas en Haskell es crucial para manejar efectos secundarios y estructurar programas de manera funcional. Vamos a profundizar en cómo las mónadas facilitan estas tareas.

Manejo de efectos secundarios

En programación funcional, los efectos secundarios (como la entrada/salida, el acceso a estado mutable, o la generación de números aleatorios) son un desafío, ya que pueden romper la pureza funcional. Las mónadas proporcionan un marco para manejar estos efectos de manera controlada:

1. **Encapsulación de efectos secundarios:** Las mónadas permiten encapsular efectos secundarios dentro de sus estructuras. Por ejemplo, la mónada `IO` encapsula operaciones de entrada/salida, mientras que la mónada `State` maneja el estado mutable.
2. **Secuenciación y composición:** Las mónadas permiten secuenciar operaciones que tienen efectos secundarios. Esto se logra mediante la función `bind (≫=)`, que asegura que las operaciones se realicen en un orden específico, manteniendo la pureza de las funciones individuales.
3. **Control de efectos impuros:** Al utilizar mónadas, los efectos impuros se aíslan del resto del código. Esto significa que las funciones puras permanecen puras, y los efectos secundarios se manejan de manera explícita y controlada.

Estructuración de programas

Las mónadas no solo manejan efectos secundarios, sino que también ayudan a estructurar programas de manera clara y modular:

1. **Separación de lógica pura e impura:** Las mónadas permiten separar claramente la lógica pura de las operaciones impuras. Esto facilita el razonamiento sobre el código y mejora la mantenibilidad y la legibilidad del programa.
2. **Composición de operaciones complejas:** A través de mónadas, se pueden componer secuencias de operaciones complejas de una manera legible y mantenible. Esto es especialmente útil en casos donde múltiples operaciones dependen unas de otras.
3. **Patrones de diseño y abstracción:** Las mónadas proporcionan un patrón de diseño que puede ser utilizado para abstraer y manejar diferentes tipos de comportamientos computacionales, como la gestión de errores (mónada `Either`), o las operaciones que pueden tener múltiples resultados (mónada `List`).
4. **Manejo de estados y contextos:** Las mónadas como `State` permiten manejar estados de una manera funcional, pasando un estado a través de una serie de funciones sin necesidad de mutabilidad explícita. Esto es especialmente útil en aplicaciones donde el estado global o local es una consideración clave.

Las mónadas en Haskell son fundamentales para manejar efectos secundarios de una manera que se alinea con los principios de la programación funcional. Proporcionan un marco para secuenciar operaciones, manejar diferentes tipos de efectos y estados, y estructurar programas de una manera modular y mantenible.

3.3. Ejemplos prácticos

Para profundizar en el uso práctico de las mónadas en Haskell, exploraremos tres mónadas comunes: `Maybe`, `IO` y `List`, proporcionando más detalles y ejemplos de código.

Mónada Maybe

La mónada `Maybe` se utiliza para representar cálculos que pueden fallar o no retornar un valor. Es muy útil para manejar situaciones donde un valor puede estar ausente sin recurrir a excepciones.

- **Definición:** La mónada `Maybe` se define con dos constructores: `Just a` para un resultado exitoso con un valor `a`, y `Nothing` para un resultado fallido sin valor.
- **Uso:** Se utiliza para encadenar operaciones que pueden fallar, deteniendo la ejecución y devolviendo `Nothing` en caso de fallo.

Ejemplo de código con Maybe:

```
findElement :: Eq a => a -> [a] -> Maybe Int
findElement _ [] = Nothing
findElement element (x:xs)
    | element == x = Just 0
    | otherwise = fmap (+1) (findElement element xs)

-- Uso de Maybe en encadenamiento
result = findElement 5 [1, 2, 3, 4, 5] -- Resultado: Just 4
```

Mónada IO

La mónada `IO` es fundamental en Haskell para manejar operaciones de entrada y salida (E/S), como leer de la consola o escribir en un archivo.

- **Definición:** La `IO` mónada encapsula una operación de E/S, manteniendo la pureza funcional. Cualquier función que realice E/S debe retornar un tipo `IO`.
- **Uso:** Permite secuenciar operaciones de E/S, asegurando que se ejecuten en el orden correcto.

Ejemplo de código con IO:

```
main :: IO ()
main = do
    putStrLn "¿Cuál es tu nombre?"
    nombre <- getLine
    putStrLn $ "Hola, " ++ nombre
```

Mónada List

La mónada `List` se utiliza para representar operaciones que pueden tener múltiples resultados, como en el caso de listas con cero, uno, o varios elementos.

- **Definición:** En el contexto monádico, una lista representa una computación que puede tener varios resultados. La función `bind` aplica una función a cada elemento de la lista y concatena los resultados.
- **Uso:** Se utiliza para operaciones que trabajan con múltiples posibles valores o caminos de ejecución.

Ejemplo de código con IO:

```
powersOfTwo :: Int -> [Int]
powersOfTwo n = [1, 2] >>= (\x -> [x^n])

-- Uso de List en encadenamiento
result = powersOfTwo 3 -- Resultado: [1, 8]
```

Estos ejemplos ilustran cómo las diferentes mónadas en Haskell facilitan distintos tipos de operaciones y patrones de programación. `Maybe` es útil para manejar la ausencia de valores, `IO` para operaciones de entrada/salida, y `List` para trabajar con conjuntos de múltiples posibles resultados, cada uno ofreciendo un enfoque estructurado y seguro para manejar estos escenarios en programación funcional.

4. Análisis sintáctico monádico

El análisis sintáctico monádico es una técnica avanzada en la construcción de analizadores y compiladores, particularmente en el ámbito de la programación funcional. Esta metodología incorpora el concepto de mónadas, una estructura central en la programación funcional, para gestionar el análisis de lenguajes de programación. A continuación, se detallará este concepto y su aplicación en varios aspectos:

4.1. Detalle y su importancia en compiladores

El análisis sintáctico monádico en la construcción de compiladores es una técnica avanzada que emplea el concepto de mónadas para abordar varios desafíos inherentes al análisis sintáctico. Aquí se detalla más sobre esta técnica y su relevancia en el campo de la compilación:

Definición ampliada

- **Integración de mónadas:** El análisis sintáctico monádico integra mónadas para controlar el flujo de análisis. Las mónadas actúan como contenedores que gestionan tanto el estado del analizador (como la posición actual en la entrada) como los posibles efectos secundarios (como errores de análisis).
- **Composición muncional:** Utilizando mónadas, los analizadores sintácticos se construyen mediante la composición de funciones pequeñas y reutilizables. Cada función maneja una pequeña parte del análisis, y las mónadas facilitan la combinación de estas funciones.
- **Gestión de estado y backtracking:** En análisis sintácticos más complejos, el manejo del estado y la capacidad de hacer backtracking (volver atrás en el análisis) son esenciales. Las mónadas permiten implementar estas características de una manera más natural y funcional.

Importancia en la construcción de compiladores

- **Flexibilidad y abstracción:** El análisis sintáctico monádico ofrece un alto nivel de abstracción, permitiendo a los desarrolladores de compiladores centrarse en la lógica del análisis sin preocuparse por detalles de bajo nivel. Esto conduce a un código más limpio y mantenible.
- **Composición de analizadores:** Permite componer fácilmente pequeños analizadores para construir analizadores más complejos. Esto es especialmente útil en la construcción de compiladores, donde diferentes partes del lenguaje pueden requerir diferentes estrategias de análisis.
- **Manejo de efectos secundarios:** En el análisis sintáctico, el manejo de errores y el seguimiento del estado son críticos. Las mónadas brindan una forma estructurada y segura de manejar estos aspectos, manteniendo la pureza del código funcional.

Ejemplo ilustrativo

Considere un analizador sintáctico para un lenguaje simple. Usando mónadas, cada parte del analizador (como identificadores, literales, operadores) puede ser escrita como una pequeña función. Estas funciones pueden ser compuestas usando mónadas para construir el analizador completo. Por ejemplo, un analizador para una expresión aritmética podría combinar analizadores para números, operadores y paréntesis, gestionando el flujo y el estado a través de la mónada.

4.2. Comparación entre el análisis sintáctico monádico y otros métodos

El análisis sintáctico monádico en programación funcional, particularmente en Haskell, se distingue de otros enfoques de análisis sintáctico en varios aspectos clave. Exploraremos estas diferencias detalladamente, comparándolo con métodos más tradicionales como los analizadores sintácticos descendentes y ascendentes.

Métodos tradicionales de análisis sintáctico

1. **Análisis descendente (Top-Down):** En este método, el analizador comienza desde la raíz del árbol de análisis y procede hacia las hojas. Los analizadores LL son un ejemplo común. Estos analizadores construyen el árbol de análisis a partir del símbolo inicial de la gramática y descomponen las estructuras sintácticas en sus componentes.
2. **Análisis ascendente (Bottom-Up):** Aquí, el análisis comienza con las hojas y avanza hacia la raíz. Los analizadores LR, como yacc/bison, son ejemplos de este enfoque. Estos analizadores identifican los componentes más pequeños del lenguaje y los combinan progresivamente en estructuras más grandes hasta alcanzar el símbolo inicial de la gramática.

Análisis sintáctico monádico

1. **Enfoque funcional:** A diferencia de los enfoques tradicionales, que a menudo se implementan en un estilo imperativo, el análisis sintáctico monádico se basa en el paradigma funcional. Utiliza mónadas para manejar el estado y los efectos secundarios de una manera que se alinea con los principios de la programación funcional.
2. **Composición y reutilización:** Las mónadas permiten una composición más flexible y modular de los analizadores. Puedes construir analizadores complejos a partir de componentes más pequeños y reutilizables.
3. **Manejo de efectos secundarios:** Mientras que los enfoques tradicionales pueden requerir manejo explícito de estado y efectos secundarios (como el backtracking), las mónadas encapsulan estos aspectos, facilitando un código más claro y mantenible.

Comparación directa

- **Flexibilidad:** El análisis sintáctico monádico es generalmente más flexible que los métodos tradicionales. Permite una mayor facilidad en la modificación y extensión del analizador.
- **Curva de aprendizaje:** Los métodos tradicionales pueden ser más accesibles para aquellos con experiencia en programación imperativa. En contraste, el análisis sintáctico monádico requiere una comprensión de las mónadas y la programación funcional.
- **Rendimiento:** Los analizadores tradicionales, especialmente los optimizados como los LR, pueden ser más rápidos en comparación con los analizadores monádicos, que pueden incurrir en una sobrecarga adicional debido a la abstracción de las mónadas.
- **Manejo de estado y errores:** Los analizadores monádicos tienden a manejar el estado y los errores de manera más elegante y con menos código "pegamento"que los métodos tradicionales, donde el manejo de estado puede ser más explícito y propenso a errores.

Mientras que los métodos tradicionales de análisis sintáctico tienen sus fortalezas en términos de familiaridad y potencialmente mejor rendimiento, el análisis sintáctico monádico sobresale en flexibilidad, modularidad y en la gestión de efectos secundarios. Esta aproximación es particularmente adecuada para entornos de programación funcional y cuando se requiere una alta capacidad de reutilización y extensibilidad del código.

4.3. Casos de estudio o ejemplos de aplicación

El análisis sintáctico monádico ha encontrado aplicaciones prácticas en diversas áreas, particularmente en el desarrollo de compiladores y herramientas de procesamiento de lenguajes. Aquí, exploraremos algunos casos de estudio y ejemplos donde este enfoque ha sido efectivamente aplicado.

Ejemplos en compiladores de lenguajes de programación

- **Haskell y GHC:** El Glasgow Haskell Compiler (GHC), uno de los compiladores más conocidos para Haskell, utiliza análisis sintáctico monádico para procesar el código fuente. Haskell, siendo un lenguaje funcional, se presta naturalmente a este enfoque, y el GHC es un ejemplo destacado de su aplicación efectiva.
- **Compiladores para lenguajes específicos:** Otros lenguajes de programación, particularmente aquellos que tienen características funcionales, también han adoptado el análisis sintáctico monádico en sus compiladores. Esto se debe a su capacidad para manejar de manera eficiente el estado y los efectos secundarios durante el análisis.

Bibliotecas de análisis sintáctico

1. **Parsec en Haskell:** Parsec es una biblioteca de análisis sintáctico monádico en Haskell que permite la construcción de analizadores sintácticos robustos y flexibles. Es ampliamente utilizada para una variedad de aplicaciones, desde el procesamiento de lenguajes de programación hasta la manipulación de archivos de configuración y datos.
2. **Megaparsec:** Otra biblioteca en Haskell, similar a Parsec pero con mejoras en términos de mensajes de error y funcionalidades. Megaparsec es ampliamente utilizado en proyectos que requieren un análisis sintáctico detallado y preciso.

Proyectos de software y herramientas

1. **Herramientas de lenguajes de marcado:** Herramientas para procesar lenguajes de marcado como XML o HTML pueden utilizar análisis sintáctico monádico para manejar la complejidad inherente a estos lenguajes, como anidación de etiquetas y atributos.
2. **Procesadores de lenguajes de dominio específico (DSLs):** Los DSLs a menudo requieren analizadores personalizados. El análisis sintáctico monádico es ideal para estos casos, ya que permite una rápida prototipación y ajustes en los analizadores.

Ventajas observadas en estos casos

- **Manejo de complejidad:** En todos estos casos, el análisis sintáctico monádico ha demostrado ser eficaz para manejar la complejidad del análisis sintáctico, especialmente en situaciones donde el manejo del estado y los errores es crítico.
- **Flexibilidad y reutilización:** La capacidad de construir analizadores componiendo pequeñas unidades reutilizables es una ventaja significativa, permitiendo a los desarrolladores adaptar rápidamente sus analizadores a requisitos cambiantes o a nuevas características del lenguaje.

El análisis sintáctico monádico se ha aplicado con éxito en una variedad de contextos, desde la construcción de compiladores hasta el desarrollo de herramientas de procesamiento de lenguaje. Su capacidad para manejar de manera eficiente y elegante el estado y los efectos secundarios, junto con su flexibilidad y modularidad, lo convierte en una opción valiosa para proyectos que requieren análisis sintáctico complejo y personalizable.

4.4. Ventajas y desafíos del análisis sintáctico monádico

El análisis sintáctico monádico es una técnica poderosa en la construcción de analizadores sintácticos, especialmente en entornos de programación funcional. Sin embargo, como cualquier enfoque, presenta tanto ventajas significativas como desafíos potenciales.

Ventajas del análisis sintáctico monádico

1. **Modularidad y composición:**
   - Los analizadores monádicos permiten una alta modularidad. Puedes componer analizadores pequeños y reutilizables para construir analizadores más complejos.
   - Esta modularidad facilita la mantenibilidad y extensión del código, lo que es particularmente útil en proyectos grandes o en evolución.
2. **Manejo elegante de estado y efectos secundarios:**
   - Las mónadas proporcionan una forma estructurada de manejar el estado y los efectos secundarios, como el backtracking y el manejo de errores, dentro del analizador.
   - Esto se alinea bien con los principios de la programación funcional, manteniendo la pureza del código y la separación de preocupaciones.

3. **Abstracción y expresividad:**
   - Los analizadores monádicos pueden ser más expresivos y fáciles de leer, especialmente para aquellos familiarizados con la programación funcional.
   - Esta abstracción permite a los desarrolladores centrarse en la lógica del análisis sintáctico en lugar de los detalles de implementación.
4. **Seguridad y robustez:**
   - El uso de mónadas puede aumentar la seguridad del código, reduciendo la posibilidad de errores comunes en el manejo de estado y efectos secundarios.
   - La estructura monádica promueve un diseño más robusto y predecible del analizador.

Desafíos en la implementación del análisis sintáctico monádico

1. **Curva de aprendizaje:**
   - Requiere un buen entendimiento de las mónadas y la programación funcional, lo que puede ser un desafío para programadores no familiarizados con estos conceptos.
   - Esto puede aumentar el tiempo de capacitación y adaptación para equipos nuevos en la programación funcional.
2. **Rendimiento:**
   - En algunos casos, el uso de mónadas puede introducir una sobrecarga de rendimiento en comparación con los analizadores sintácticos imperativos más directos.
   - Optimizar analizadores monádicos para rendimiento puede requerir técnicas avanzadas y un profundo conocimiento de la programación funcional.
3. **Depuración y mantenimiento:**
   - La naturaleza abstracta de las mónadas puede hacer que la depuración sea más desafiante, ya que el flujo de control puede ser menos transparente que en los enfoques imperativos.
   - Mantener y modificar un analizador monádico puede requerir un nivel de experiencia más alto en programación funcional.
4. **Adopción y ecosistema:**
   - Dependiendo del lenguaje y el entorno de desarrollo, puede haber menos recursos, herramientas o ejemplos disponibles para el análisis sintáctico monádico en comparación con enfoques más tradicionales.
   - Esto puede influir en la decisión de adoptar este enfoque, especialmente en entornos donde la eficiencia de desarrollo es crítica.

Mientras que el análisis sintáctico monádico ofrece ventajas claras en términos de modularidad, manejo de estado y efectos secundarios, y expresividad, también presenta desafíos en cuanto a la curva de aprendizaje, rendimiento, depuración y mantenimiento. La elección de utilizar el análisis sintáctico monádico dependerá de factores como el contexto del proyecto, las habilidades del equipo de desarrollo y los requisitos específicos del analizador sintáctico a desarrollar.

5. Implementación

El lenguaje propuesto para su análisis, es una variante del lenguaje para expresiones aritméticas sencillas, planteamos esta modesta alternativa para enriquecer en medida de lo posible nuestro analizador y así darle cierta variedad. La idea tras su planteamiento, es construir expresiones que tengan ya de por si una forma similar a la de los ASA vistos hasta ahora, lo cuál hace un poco más sencillo el análisis. Además de lo anterior, incluimos un manejo de paréntesis que a simple vista nos permita identificar que operador corresponde a que operandos sin ser ambiguo. En términos generales, tratamos de construir un lenguaje de expresiones aritméticas original (o eso pretendemos), fácil de implementar, que reúna algunas ventajas de la notación infija y algunas otras de la notación prefija, esperamos haber alcanzado un resultado satisfactorio. La gramática que define a nuestro lenguaje es la siguiente:

```
S := (E) | E
E := X | - E | (E) | + A A | - A A | * A A | / A A
A := (E) | X
X := -X | 0, 1, 2, 3, ...
```

Probamos 3 implementaciones distintas, una de ellas implementa directamente la instanciación de Functor, Applicative, Monad y Alternative, para construir un parser a través de monadas. La segunda emplea la monada Parsec definida dentro de las bibliotecas de Haskell y la tercera emplea la monada State.

5.1. Primera implementación

Definimos el tipo de datos Expr que emplearemos para representar las expresiones de nuestro lenguaje en todas las implementaciones. Pueden ser sumas, restas, productos, divisiones, números enteros, etc. Después de lo anterior, definimos un type para parser, mismo que encapsula una función de parseo, es decir, una función que toma una cadena de entrada y devuelve una lista de pares con el resultado del parseo y el resto de la cadena no procesada. Ahora explicaremos en términos generales como está compuesto nuestro código:

- **Parsers básicos**

  itemParser: Parsea un solo carácter de la cadena de entrada.

  satisfy: Asegura que un carácter cumple con un predicado específico.

  charParser: Parsea un carácter específico de la cadena de entrada.

- **Instancias de Type Classes**

  Functor: Permite aplicar una función a los resultados de un parser. Por ejemplo, para transformar el resultado de un parser de caracteres a un parser de expresiones.

  Applicative: Permite combinar parsers y aplicar funciones embebidas en el resultado. Lo utilizamos para construir parsers más complejos a partir de parsers más simples.

  Monad: Permite la composición secuencial de parsers. Lo utilizamos para encadenar parsers y procesar la entrada de manera estructurada.

  Alternative: Brinda una abstracción para la elección no determinista entre parsers, así podemos definir alternativas en el parseo.

- **Parsers para espacios y tokens**

  spaceParser: Parsea cero o más espacios en blanco. En las 3 implementaciones, los espacios son descartables salvo para la separación de números. Una expresión "- 3 2" es igual a la expresión "-3 2", no se tomará el primer elemento de "-3 2" como negativo.

  stringParser: Parsea una cadena específica permitiendo espacios opcionales.

  tokenParser: Parsea un token específico permitiendo espacios opcionales.

- **Parsers para expresiones aritméticas**

  multParser, sumParser: Parsean operadores de multiplicación y división, y suma y resta, respectivamente.

  negativoParser: Parsea el signo negativo de una expresión.

  enteroParser: Parsea valores enteros, tanto positivos como negativos.

  factor: Parsea factores, que pueden ser enteros o expresiones entre paréntesis.

  term, expr: Parsean términos y expresiones aritméticas.

  entreParentesis: Parsea expresiones dentro de paréntesis.

- **Funciones principales**

  buildASA: Construye el Árbol de Sintaxis Abstracta (ASA) a partir de los resultados del parseo.

  parserAritmetico: Función principal que maneja el caso de error y devuelve el ASA en caso de recibir una expresión válida.

Esta implementación utiliza composición de parsers y las instancias de type classes para construir un parser modular y extensible que puede reconocer expresiones aritméticas según la gramática definida. Los parsers más simples se combinan para construir parsers más complejos, permitiendo analizar la estructura de las expresiones y construir un árbol de sintaxis abstracta (ASA) correspondiente.

5.2. Segunda implementación

En este caso se empleó directamente la biblioteca Parsec de Haskell, misma que nos brinda muchas herramientas para la construcción de analizadores sintácticos, de modo que es visible una simplificación del código respecto de la implementación anterior. Parsec fue diseñada para ser fácil de usar y expresiva, permitiendo la definición de gramáticas de manera clara y concisa. Explicaremos como está compuesto nuestro código.

- **Parsers básicos**

  itemParser: Parsea un solo carácter.

  stringParser: Parsea una cadena específica.

  spaceParser: Parsea cero o más espacios.

  tokenParser: Parsea un token específico y consume espacios.

- **Parsers para expresiones aritméticas**

  multParser, sumParser: Parsean operadores de multiplicación y división, y suma y resta, respectivamente.

  enteroParser: Parsea números enteros, incluyendo la posibilidad de ser negativos.

  factor: Parsea factores, que pueden ser números enteros o expresiones entre paréntesis.

  entreParentesis: Parsea expresiones entre paréntesis.

  term, expr: Parsean términos y expresiones aritméticas.

- **Parser principal**
  parserAritmetico: Esta es la función principal para el análisis sintáctico de expresiones aritméticas. Utiliza parse de Parsec para analizar la cadena de entrada.
  spaceParser $*>$ *expr* $<*$ *eof:* Define la estructura general de la expresión permitiendo espacios al inicio y al final, y asegurándose de que no haya más caracteres después de analizar la expresión.
- **Manejo de errores**
  Empleamos Either ParseError Expr para manejar el resultado del parseo. Si hay un error, se devuelve un Left con un mensaje personalizado.

Este código con Parsec realiza el análisis sintáctico de expresiones aritméticas de manera similar al anterior, pero para ello emplea la biblioteca Parsec para simplificar la definición de parsers y manejar errores de manera más efectiva. La sintaxis y los combinadores específicos de Parsec contribuyen a una implementación más clara y concisa en comparación con la implementación manual de combinadores de parsers. Los parsers en este caso, también se componen de manera modular para construir parsers más complejos y analizar la estructura de las expresiones según la gramática definida.

5.3. Tercera implementación

En este caso de igual forma que en el anterior, nos ahorramos algunas definiciones al emplear una biblioteca monádica como lo es State, por ello veremos un código más suscinto respecto de la primer implementación aunque no tanto respecto de la segunda, veremos al final algunas diferencias respecto de las implementaciones anteriores. Explicaremos cuál es la estructura de nuestro código. Nota: esta implementación tiene un pequeño fallo que se explica en la sección siguiente.

- **Uso de la monada Transformer StateT**
  ParserState a: Es una instancia de StateT String [] a. Combina el monad State para el manejo del estado con el monad no determinista [].
- **Parsers básicos**
  itemParser: Parsea un solo carácter en el contexto del monad StateT.
  satisfy: Parsea un carácter que cumple un predicado en el contexto del monad StateT.
  charParser: Parsea un carácter específico.
  spaceParser: Parsea cero o más espacios en el contexto del monad StateT.
  stringParser: Parsea una cadena específica en el contexto del monad StateT.
  tokenParser: Parsea un token específico y consume espacios.
- **Parsers para expresiones aritméticas**
  multParser, sumParser: Parsean operadores de multiplicación y división, y suma y resta, respectivamente.
  enteroParser: Parsea números enteros, permitiendo la posibilidad de ser negativos.
  factor: Parsea factores, que pueden ser números enteros o expresiones entre paréntesis.
  entreParentesis: Parsea expresiones entre paréntesis.
  term, expr: Parsean términos y expresiones aritméticas.
- **Parser principal**
  parserAritmetico: Es la función principal para el análisis sintáctico de expresiones aritméticas.Utiliza evalStateT para ejecutar el parser en el estado inicial (cadena de entrada). Esta función devuelve todos los posibles resultados del parser en una lista debido al monad no determinista [].

A diferencia de la implementación con instances, la implementación con StateT utiliza el monad transformer para combinar los efectos de estado y no determinismo de manera más directa, por ende nos ahorramos escribir algunas definiciones. Además de lo anterior, los combinadores son más específicos y están definidos directamente en el contexto del monad StateT. Por otro lado, cuándo comparamos esta implementación con la segunda, hemos de considerar que Parsec maneja automáticamente la generación de mensajes de error detallados y personalizados, mientras que la implementación con StateT utiliza una lógica manual para lanzar errores. Es importante señalar que Parsec tiende a ser más expresivo y conciso que StateT para definir parsers, debido a su diseño específico para el análisis sintáctico.

5.4. Aprendizajes y desafíos

El desarrollo del código presentado para este proyecto, conllevó una serie de desafíos inherentes a las exigencias del mismo en primer lugar. Familiarizarse con el concepto y uso de las monadas fue complejo, toda vez que en la facultad no recibimos un curso de teoría de las categorías ni nada parecido. Además de lo antes mencionado, el concepto es en si mismo muy abstracto y esquivo, es decir, justo cuándo crees que ya comprendiste lo que es una monada, va y resulta que no era de la forma en que lo habías asimilado. Una vez tuvimos una mayor certeza de estar manejando adecuadamente el concepto, pasamos a la implementación de los parsers empleando Haskell, donde nos encontramos con un hallazgo interesante: el lenguaje mismo ya incluye bibliotecas orientadas uso del análisis monádico. Ese hallazgo nos sirvió para desarrollar una de nuestras implementaciones, lo que supuso tener que familiarizarnos con esta nueva biblioteca, aunque de hecho, en las 3 implementaciones un desafío constante fue estudiar y hacer buen uso de bibliotecas hasta entonces desconocidas.

En la primera implementación fue complicado procesar la abstracción y la modularidad al definir operaciones sobre el tipo Parser, pero una vez le hallamos el truco, eso nos sirvió para desarrollar las otras 2 implementaciones de forma mucho más sencilla. Sin embargo, entender el uso de estos typeclasses en particular, aún es un trauma del que tendremos que reponernos. Durante el desarrollo de de la segunda implementación, ya más familiarizados con el uso de monadas, notamos lo expresivo que puede ser Haskell y lo clara que puede hacer la estructura del programa, facilitando así su modificación y mantenimiento. A pesar de estas ventajas, al igual que en la implementación anterior, usar nuevas bibliotecas siempre tiene su lado no tan amigable.

Por último, la tercera implementación, parecía ser la más sencilla de todas, no por qué inherentemente sea menos compleja, sino por que ya para ese momento teníamos más dominados los conceptos y herramientas. Sin embargo, esta implementación resulta menos expresiva y requiere más código en comparación con Parsec ya que StateT no está expresamente orientado al análisis sintáctico. A pesar de lo antes dicho, nos enfrentamos al problema de que su no determinismo nos dio problemas a la hora de manejar los casos de error, pues nos devuelve la parte de la cadena que si alcanzó a procesar para ciertas expresiones. En principio, el manejo de errores fue una tarea desafiante, pues suponía hacer un rastreo paso por paso de las declaraciones de nuestros parsers modulares para detectar los fallos y ver por qué el parser aceptaba/rechazaba cadenas que no debía. Algo que nos llamó fuertemente la atención, precisamente derivado de la modularidad ya mencionada, es lo potencialmente poderosa que puede ser esta herramienta a la hora de descomponer un problema en subproblemas más sencillos. La descomposición como antes mencionamos, aunque en principio da problemas para familiarizarnos con ella, al final puede servir de mucha ayuda cuando se trabaja en equipo o cuándo se quieren aislar componentes críticos de nuestros programas.

Incluimos dentro del código algunos ejemplos de expresiones para probar los analizadores.

6. Conclusiones

Esta investigación nos ha llevado a reflexionar sobre las capacidades, elegancia y versatilidad de los lenguajes que siguen el paradigma funcional como Haskell. Hemos profundizado en un concepto que ha transformado la forma en que manejamos efectos secundarios y cómputos complicados en programación funcional. También nos ha servido para notar cómo las monadas son una herramienta poderosa y versátil para gestionar de manera elegante el estado y la composición de parsers. Entre otras cosas, la capacidad de encadenar computos de forma concisa y declarativa, mediante la composición monádica, ha hecho que podamos simplificar en gran medida la implementación de analizadores sintácticos, haciendo que el código sea más claro y mantenible.

La implementación práctica de estos conceptos en Haskell, es un aspecto que nos agradó bastante del lenguaje. Las monadas han sido integradas en Haskell sin alterar la sintaxis concisa y expresiva que lo caracteriza. La flexibilidad y composabilidad inherentes a las monadas hacen que sea más fácil diseñar programas robustos y modulares, aún cuando agarrarle el truco a su declaración y diseño toma algo de tiempo. Otra gran ventaja de las monadas en Haskell, es que nos permiten abordar problemas complejos de manera incremental y estructurada.

En resumen, esta investigación nos ha permitido apreciar la elegancia y eficacia de las monadas, en el contexto del análisis sintáctico en Haskell. La combinación de conceptos abstractos dentro del paradigma funcional funcional, como el discutido a lo largo de este proyecto, ha enriquecido nuestra comprensión de la programación, brindándonos herramientas poderosas para abordar desafíos computacionales de una mejor forma.

En lo que se refiere a nuestra experiencia en el desarrollo de este proyecto, podemos asegurar que reforzó nuestra capacidad para resolver problemas complejos mediante la aplicación de conceptos teóricos. Insistimos bastante en lo mucho que nos sirvió este proyecto y el curso de compiladores en general, para apreciar que la programación funcional es muy clara, poderosa y elegante, lo cuál ha despertado nuestro interés por seguir trabajando con lenguajes que sigan este paradigma en proyectos futuros.

Finalmente, respecto al proyecto en si mismo, esta clase de trabajos fomentan en nosotros la adquisición de una mentalidad para la investigación o dicho de otra forma, propiamente científica. La experiencia práctica en la resolución de problemas como el abordado, a través de la investigación sobre herramientas y conceptos antes desconocidos, sientan las bases para que podamos llevar a cabo soluciones más avanzadas y mejor planteadas para distintos problemas en el área de las ciencias de la computación.

Aprovechamos esta pequeña sección para agradecer a todos los profesores, Manuel, Braulio, Pedro y Javier, por su excelente trabajo a lo largo del curso. Deseamos que tengan una carrera y una vida largas y prósperas, además, en tanto que las fechas ya están cerca, les deseamos una feliz navidad y un buen comienzo en el año de 2024.

7. Referencias

- Haskell Monads — Monday morning Haskell. (s.f.). Monday Morning Haskell.
- Jitani, V. (2023, 20 julio). Monads in functional programming explained. Built In.
- Haskell Language => Mónadas. (s.f.).
- DevX. (2023, 18 agosto). Monad: Definition, Examples - DevX.
- Deciphering Haskell's applicative and Monadic Parsers - Eli Bendersky's website. (s.f.).
- A gentle introduction to Haskell: About Monads. (s.f.).
- Dcc/fcup, P. V. (s.f.). Monadic Parsing.
- Parsec. (s.f.). Hackage.
- MoNAD (Functional Programming) - HandWiki. (s.f.).
- Monadic Parser Combinators in Haskell - DeepSource. (s.f.).
- Quick and dirty guide to Monadic parsers and Angstrom. (s.f.). OCamlverse.