1 Introducción

Este documento sirve como guía para comprender una implementación en Haskell para analizar y calcular el límite de una función polinómica en un punto especificado. El código proporcionado utiliza el paradigma de programación funcional de Haskell y aprovecha la biblioteca ‘Text.ParserCombinators.Parsec‘ para analizar expresiones.
El código consta de tres funciones principales:

1. **Análisis de Polinomios:** La función ‘parsePolynomial‘ toma una representación en cadena de un polinomio como entrada y lo analiza en una forma estructurada para su posterior manipulación. Utiliza la función ‘polynomialParser‘, que a su vez utiliza ‘termParser‘ para analizar términos individuales dentro de la expresión polinómica.

2. **Cálculo de Límites:** La función ‘calculateLimit‘ calcula el límite de una función polinómica para un valor dado de ‘x‘. Evalúa la expresión polinómica en el punto ‘x‘ especificado sumando las contribuciones de cada término en el polinomio.

3. **Función Principal:** La función ‘main‘ orquesta la interacción con el usuario. Solicita al usuario que introduzca una expresión polinómica, la analiza, solicita el valor de ‘x‘ y luego calcula y muestra el límite de la función polinómica en ese valor de ‘x‘.

Este documento proporcionará una explicación detallada de cada componente del código, incluyendo cómo se analizan los polinomios, se extraen los términos y se calculan los límites. Además, cubrirá el uso de los constructos de análisis monádicos de Haskell y las técnicas de programación funcional empleadas en la implementación.

2 Análisis de Polinomios

2.1 Función ‘parsePolynomial‘

```haskell
parsePolynomial :: String → Either ParseError Polynomial -- parse polynomial from string
parsePolynomial input = parse polynomialParser "" input -- parse polynomial from string
```

La función parsePolynomial es una función que toma una cadena de caracteres como entrada y devuelve un valor de tipo `Either ParseError Polynomial`.
Nota: El tipo `Either a b` en Haskell es un tipo que puede contener un valor de tipo a o un valor de tipo b. En este caso, a es ParseError y b es Polynomial. Esto significa que la función parsePolynomial puede devolver un error de análisis (ParseError) o un polinomio (Polynomial).
La función `parsePolynomial` utiliza la función `parse` de la biblioteca Parsec. La función parse toma tres argumentos:

1. Un analizador (polynomialParser en este caso), que es una función que intenta convertir una cadena de caracteres en un valor de algún tipo.

2. Un nombre de fuente, que se utiliza en los mensajes de error para indicar de dónde provienen los datos de entrada. En este caso, se pasa una cadena vacía porque los datos de entrada no provienen de ninguna fuente en particular.

3. Los datos de entrada que se van a analizar (input en este caso).

La función parse intenta aplicar el analizador a los datos de entrada. Si el análisis tiene éxito, devuelve `Right polynomial`, donde `polynomial` es el polinomio resultante. Si el análisis falla, devuelve Left `parseError`, donde `parseError` es un mensaje de error que describe por qué falló el análisis.
Por lo tanto, la función `parsePolynomial` intenta convertir una cadena de caracteres en un polinomio utilizando el analizador `polynomialParser`. Si tiene éxito, devuelve el polinomio. Si falla, devuelve un mensaje de error.

2.2 Función 'polynomialParser'

```
polynomialParser :: Parser Polynomial
polynomialParser = do
terms <- sepBy termParser (skipMany space >> (char '+' <|> char '-') >> skipMany
    space) -- parse terms separated by '+' or '-'
return \$ Polynomial terms -- return polynomial
```

La función polynomialParser es un analizador que se utiliza para analizar una cadena de caracteres y convertirla en un polinomio. Un polinomio es una suma de términos, por lo que este analizador busca una serie de términos separados por '+' o '-'.

1. `terms <- sepBy termParser (skipMany space >> (char '+' <|> char '-') >> skipMany space)`: Aquí se utiliza la función `sepBy` de Parsec, que toma dos analizadores como argumentos. El primer analizador, `termParser`, se utiliza para analizar los términos individuales del polinomio. El segundo analizador, `(skipMany space >> (char '+' <|> char '-') >> skipMany space)`, se utiliza para analizar los separadores entre los términos, que en este caso son '+' o '-' rodeados por cualquier cantidad de espacios. `sepBy` aplica estos dos analizadores de forma alternativa hasta que no puede aplicarlos más, y devuelve una lista de todos los términos que pudo analizar.

2. `return $ Polynomial terms`: Finalmente, se construye un polinomio a partir de la lista de términos y se devuelve. En Haskell, return en el contexto de un monad (como Parser) no es como return en la mayoría de los otros lenguajes de programación. No termina la función. En su lugar, envuelve un valor en la monad. En este caso, envuelve el polinomio en el analizador, lo que significa que el resultado del analizador es ese polinomio.

Por lo tanto, polynomialParser es un analizador que convierte una cadena de caracteres en un polinomio, analizando los términos individuales y los separadores entre ellos.

2.3 Función 'termParser'

```
termParser :: Parser Term
termParser = try termWithX <|> termWithoutX
where
  termWithX = do
    sign <- option '+' (skipMany space >> (char '-' <|> char '+'))
    coefficient <- option "1" (many1 (digit <|> char '.')) -- parse coefficient
    skipMany space
    char 'x' -- parse 'x'
    skipMany space
    exponent <- option "1" (char '^' >> many1 digit) -- parse exponent
    let signedCoefficient = if sign == '-' then read ("-" ++ coefficient) else
        read coefficient
    return \$ Term (signedCoefficient :: Double) (read exponent :: Int) --
        return term
  termWithoutX = do
    sign <- option '+' (skipMany space >> (char '-' <|> char '+'))
    coefficient <- many1 (digit <|> char '.')
    let signedCoefficient = if sign == '-' then read ("-" ++ coefficient) else
        read coefficient
    return \$ Term (signedCoefficient :: Double) 0
```

La función `termParser` es un analizador que se utiliza para analizar una cadena de caracteres y convertirla en un término de un polinomio. Un término de un polinomio es de la forma axn̂, donde a es el coeficiente, `x` es la variable y `n` es el exponente. a función `termParser` utiliza la función `try` de `Parsec` para intentar aplicar el analizador `termWithX`. Si `termWithX` falla, entonces `termParser` intentará aplicar el analizador `termWithoutX`.

1. `termWithX`: Este analizador se utiliza para analizar términos que incluyen la variable `x`.
   - `sign <- option '+' (skipMany space >> (char '-' <|> char '+'))`: Aquí se utiliza la función option de Parsec para intentar analizar un signo '+' o '-'. Si no puede analizar un signo, entonces devuelve '+' por defecto.
   - `coefficient <- option "1" (many1 (digit <|> char '.'))`: Aquí se intenta analizar el coeficiente del término. Si no puede analizar un coeficiente, entonces devuelve "1" por defecto.
   - `skipMany space`: Aquí se salta cualquier cantidad de espacios.
   - `char 'x'`: Aquí se analiza la variable 'x'.
   - `exponent <- option "1" (char '^' >> many1 digit)`: Aquí se intenta analizar el exponente del término. Si no puede analizar un exponente, entonces devuelve "1" por defecto.
   - `let signedCoefficient = if sign == '-' then read ("-" ++ coefficient) else read coefficient`: Aquí se calcula el coeficiente firmado en función del signo.
   - `return $ Term (signedCoefficient :: Double) (read exponent :: Int)`: Finalmente, se construye un término a partir del coeficiente firmado y el exponente, y se devuelve.
2. `termWithoutX`: Este analizador se utiliza para analizar términos que no incluyen la variable x.
   - `sign <- option '+' (skipMany space >> (char '-' <|> char '+'))`: Aquí se intenta analizar un signo '+' o '-'. Si no puede analizar un signo, entonces devuelve '+' por defecto.
   - `coefficient <- many1 (digit <|> char '.')`: Aquí se intenta analizar el coeficiente del término.
   - `let signedCoefficient = if sign == '-' then read ("-" ++ coefficient) else read coefficient`: Aquí se calcula el coeficiente firmado en función del signo.
   - `return $ Term (signedCoefficient :: Double) 0`: Finalmente, se construye un término a partir del coeficiente firmado y un exponente de 0 (ya que no hay variable x), y se devuelve.

Por lo tanto, termParser es un analizador que convierte una cadena de caracteres en un término de un polinomio, ya sea con o sin la variable x.

3 Cálculo de Límites

3.1 Función 'calculateLimit'

```
1 calculateLimit :: Polynomial → Double → Double -- calculate limit
2 calculateLimit (Polynomial terms) x = sum $ map (\(Term coefficient exponent) →
      coefficient * (x ** fromIntegral exponent)) terms -- calculate limit
```

La función `calculateLimit` toma un polinomio y un valor de `x` como entrada y calcula el límite del polinomio cuando `x` tiende a ese valor.

1. `(Polynomial terms)`: Este es un patrón de coincidencia que extrae la lista de términos del polinomio de entrada.
2. `map (`
   `(Term coefficient exponent) -> coefficient (x fromIntegral exponent)) terms`: Aquí se utiliza la función map para aplicar una función a cada término de la lista de términos. La función que se aplica es una función lambda que toma un término y calcula el valor del término para el valor dado de x. Para hacer esto, multiplica el coeficiente del término por `x` elevado al exponente del término. La función `fromIntegral` se utiliza para convertir el exponente a un número de punto flotante, ya que la operación de exponenciación () requiere que ambos operandos sean de punto flotante.
3. `sum`: Finalmente, se utiliza la función sum para sumar todos los valores de los términos calculados. Esto da el valor del polinomio para el valor dado de x.

Por lo tanto, calculateLimit es una función que calcula el límite de un polinomio cuando x tiende a un valor dado.

4 Función Principal

```haskell
main :: IO ()
main = do
  putStrLn "Enter a polynomial:" -- prompt user for polynomial
  input <- getLine -- read input
  let polynomial = parsePolynomial input -- parse polynomial
  case polynomial of -- handle result
    Left error → print error -- print error
    Right polynomial → do -- calculate limit
      putStrLn "Enter a value for x:" -- prompt user for x
      x <- readLn -- read x
      print $ calculateLimit polynomial x -- print limit
```

La función main es la función principal que se ejecuta cuando se inicia el programa. Aquí está el desglose de cómo funciona:

1. `putStrLn "Enter a polynomial:"`: Aquí se imprime un mensaje para solicitar al usuario que introduzca un polinomio.
2. `input <- getLine`: Aquí se lee una línea de entrada del usuario y se almacena en la variable `input`.
3. `let polynomial = parsePolynomial input`: Aquí se analiza el polinomio introducido por el usuario utilizando la función `parsePolynomial` y se almacena en la variable `polynomial`.
4. `case polynomial of`: Aquí se utiliza la expresión de caso para manejar el resultado del análisis del polinomio.
   - `Left error -> print error`: Si el análisis del polinomio falla, se imprime el error.
   - `Right polynomial -> do`: Si el análisis del polinomio tiene éxito, se procede a calcular el límite.
     - `putStrLn "Enter a value for x:"`: Aquí se imprime un mensaje para solicitar al usuario que introduzca un valor para `x`.
     - `x <- readLn`: Aquí se lee un valor de `x` del usuario y se almacena en la variable `x`.
     - `print $ calculateLimit polynomial x`: Aquí se calcula el límite del polinomio para el valor dado de `x` y se imprime.

Por lo tanto, la función main orquesta la interacción con el usuario, solicitando un polinomio, analizándolo, solicitando un valor para `x` y calculando y mostrando el límite del polinomio para ese valor de `x`.