1 Type Mismatch (Coincidencia de tipos):

Error: "Couldn't match expected type 'X' with actual type 'Y'"

El error "Couldn't match expected type 'X' with actual type 'Y'" en Haskell indica que hay un conflicto entre el tipo de datos esperado (X) y el tipo de datos real proporcionado (Y) en alguna parte de tu código. Este error es común cuando estás escribiendo código que involucra funciones polimórficas, como funciones de alto orden, funciones que toman tipos de datos genéricos o funciones con tipos de datos complejos.

Aquí hay una explicación detallada de cada parte del mensaje de error:

- **"Couldn't match"**: Esto indica que el compilador no pudo encontrar una coincidencia entre los tipos de datos esperados y los tipos de datos proporcionados.
- **"expected type 'X'"**: Se esperaba que el tipo de datos fuera X en cierto contexto del código. X puede ser cualquier tipo de datos específico, como Int, String, Bool, etc., o puede ser un tipo de datos más general, como una lista, un tipo de dato definido por el usuario, etc.
- **"with actual type 'Y'"**: En cambio, el tipo de datos que se proporcionó realmente fue Y. Y puede ser cualquier tipo de datos que Haskell pueda manejar, como Int, String, Bool, etc., o incluso puede ser otro tipo de datos complejo o definido por el usuario.

Para corregir este error, debes asegurarte de que los tipos de datos coincidan correctamente en todas partes del código. Esto puede implicar revisar las definiciones de funciones, los tipos de datos de entrada y salida, y cualquier otra parte del código donde haya una expectativa sobre el tipo de datos que se está utilizando.

Aquí hay un ejemplo de cómo podría verse este error en código:

```
-- Función que suma dos números
suma :: Int -> Int -> Int
suma x y = x + y

-- Llamada a la función suma con tipos de datos incorrectos
resultado = suma "5" 6
```

En este ejemplo, la función `suma` está definida para tomar dos argumentos de tipo `Int` y devolver un `Int`. Sin embargo, se intenta llamar a la función `suma` con una cadena `"5"` como primer argumento en lugar de un número entero. Esto resulta en un error "Couldn't match expected type 'Int' with actual type 'String'". Para corregir este error, debes asegurarte de proporcionar argumentos que coincidan con los tipos de datos esperados por la función `suma`. Por ejemplo, podrías llamar a la función así: `resultado = suma 5 6`.

2 Indentation Error (Error de indentación):

Error: "parse error (possibly incorrect indentation or mismatched brackets)"

El error "parse error (possibly incorrect indentation or mismatched brackets)" en Haskell indica que el compilador encontró un problema al intentar analizar tu código fuente. Este error puede ser causado por uno de los dos problemas principales:

1. **Indentación incorrecta**: Haskell utiliza la indentación para delimitar bloques de código en lugar de llaves o palabras clave de inicio y fin como en otros lenguajes de programación. Por lo tanto, es importante mantener una indentación coherente en tu código. Si hay una indentación incorrecta, como una línea demasiado indentada o una falta de indentación en un lugar inesperado, Haskell puede arrojar este error.
2. **Paréntesis o corchetes mal cerrados o desbalanceados**: Si tienes paréntesis, corchetes o llaves en tu código, asegúrate de que estén correctamente cerrados y balanceados. Si hay algún paréntesis o corchete que falta o está mal colocado, Haskell no podrá analizar correctamente tu código y te dará este error.

Aquí hay un ejemplo de cómo podrías encontrarte con este error:

```
miFuncion x =
  if x > 0
  then "Positivo"
  else "Negativo"
```

En este caso, la función `miFuncion` está definida para devolver una cadena dependiendo del valor de `x`. Sin embargo, hay un error de indentación: la línea que sigue a `then` debería estar más indentada para indicar que es parte del bloque `then`. Para corregir este error, deberías indentar correctamente esa línea:

```
miFuncion x =
  if x > 0
    then "Positivo"
    else "Negativo"
```

Asegúrate de revisar tu código para corregir cualquier problema de indentación o desbalanceo de paréntesis o corchetes que puedas encontrar. Esto debería solucionar el error "parse error (possibly incorrect indentation or mismatched brackets)".

3 Variable Not in Scope (Variable no está en el ámbito):

Error: "Not in scope: 'variableName'"

El error "Not in scope: 'variableName'" en Haskell indica que el compilador no puede encontrar una definición válida para la variable 'variableName' en el ámbito actual. Esto puede ocurrir por varias razones:

1. **Variable no definida**: La variable que estás intentando utilizar no ha sido definida en ningún lugar del código.
2. **Ámbito incorrecto**: La variable puede haber sido definida en otro ámbito o módulo y no está disponible en el ámbito actual donde estás intentando utilizarla.
3. **Error tipográfico**: Puede haber un error tipográfico en el nombre de la variable, por lo que el compilador no puede reconocerla.

Para corregir este error, debes asegurarte de que la variable 'variableName' esté definida en un ámbito accesible desde el punto donde estás intentando utilizarla. Si la variable debería estar disponible en el ámbito actual, verifica que no haya errores tipográficos en el nombre de la variable.

Aquí hay un ejemplo que muestra cómo podrías encontrarte con este error:

```
main = do
  putStrLn variableName
```

Si 'variableName' no está definida en ningún lugar del código, obtendrás el error "Not in scope: 'variableName'". Para corregirlo, necesitas definir la variable 'variableName' antes de utilizarla, o verificar si debería estar disponible en el ámbito actual.

Haskell es un lenguaje con un ámbito léxico, lo que significa que la visibilidad de las variables está determinada por su posición en el código. Por lo tanto, si una variable está definida dentro de una función, solo estará disponible dentro de esa función, y si está definida fuera de una función, estará disponible en todo el módulo o archivo en el que se encuentra.

4 Infinite Recursion (Recursión infinita):

Error: El programa no termina y parece estar en un bucle infinito.

Cuando un programa Haskell no termina y parece estar atrapado en un bucle infinito, se debe a que el código está ejecutando una recursión infinita o un bucle infinito. Esto puede ocurrir debido a varios motivos:

1. **Recursión infinita**: En Haskell, es común utilizar la recursión para iterar sobre estructuras de datos o realizar cálculos. Sin embargo, si no se proporciona una condición de terminación adecuada, la función recursiva puede llamarse a sí misma indefinidamente, lo que resulta en un bucle infinito. **En funciones recursivas, este error es muy común cuando no se define adecuadamente el caso base.**
2. **Bucle infinito**: Aunque Haskell no tiene bucles en el sentido tradicional, es posible escribir código que tenga un comportamiento similar a un bucle infinito utilizando la recursión o funciones como `iterate` o `repeat` de manera incorrecta.
3. **Evaluación perezosa (lazy evaluation)**: Haskell utiliza la evaluación perezosa, lo que significa que las expresiones no se evalúan hasta que sea necesario. En algunos casos, esto puede resultar en la evaluación infinita de una expresión si no se maneja adecuadamente.

Para solucionar este problema, debes revisar tu código para identificar dónde se está produciendo el bucle infinito y corregirlo. Algunas estrategias comunes incluyen:

- Agregar una condición de terminación adecuada en las funciones recursivas.
- Utilizar funciones y estructuras de datos que eviten la evaluación infinita.
- Revisar la lógica del código para asegurarse de que no haya errores de lógica que causen bucles infinitos.

5 Pattern Match Exhaustiveness (Exhaustividad en coincidencia de patrones):

Advertencia/Error: "Pattern match(es) are non-exhaustive"

Cuando recibes la advertencia o error "Pattern match(es) are non-exhaustive" en Haskell, significa que has definido una función o una expresión utilizando coincidencias de patrones (pattern matching), pero no has considerado todos los casos posibles para los patrones. Esto es un problema porque Haskell exige que todos los casos posibles estén cubiertos para garantizar que la función sea total y segura.

Veamos un ejemplo para entenderlo mejor. Supongamos que tienes la siguiente función que calcula el factorial de un número:

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

En esta función, has definido dos patrones: `factorial 0` y `factorial n`. Sin embargo, te falta considerar el caso cuando `n` es negativo. Esto significa que la función fallará si se le pasa un número negativo y, por lo tanto, el patrón no es exhaustivo.

Para corregir este error, debes considerar todos los casos posibles para los patrones. En este ejemplo, podrías agregar un patrón para números negativos y manejarlos adecuadamente, quizás lanzando una excepción, devolviendo un valor especial o manejándolos de otra manera dependiendo de los requisitos de tu programa. Por ejemplo:

```
factorial :: Int -> Int
factorial 0 = 1
factorial n
    | n < 0     = error "No se puede calcular el factorial de un número negativo"
    | otherwise = n * factorial (n - 1)
```

Ahora, el patrón es exhaustivo porque cubre todos los casos posibles para la entrada `n`. Entonces, cuando recibes el error o advertencia "Pattern match(es) are non-exhaustive" en Haskell, debes revisar tus patrones de coincidencia y asegurarte de considerar todos los casos posibles para garantizar que tu función sea total y segura.

6 Ambiguous Type Variable (Variable de tipo ambigua):

Error: "Ambiguous type variable 'a' in the constraints"

El error "Ambiguous type variable 'a' in the constraints" en Haskell ocurre cuando el compilador no puede determinar de manera única el tipo de una variable 'a' basándose en los contextos de uso dentro de una expresión o una función. Esto generalmente sucede cuando hay ambigüedad en la inferencia de tipos.

En Haskell, el sistema de tipos es poderoso pero también puede ser complejo. A menudo, el compilador puede inferir los tipos automáticamente a partir de los contextos en los que se utilizan las variables y las funciones. Sin embargo, en algunos casos, puede haber múltiples opciones válidas para el tipo de una variable 'a', lo que genera ambigüedad.

Por ejemplo, considera la siguiente función que concatena dos listas:

```
concatenar :: [a] -> [a] -> [a]
concatenar xs ys = xs ++ ys
```

En esta función, la variable 'a' se usa tanto en el tipo de entrada `[a]` como en el tipo de salida `[a]`. Esto podría llevar a una ambigüedad si las listas de entrada no tienen el mismo tipo. Por ejemplo, si llamamos a la función de la siguiente manera:

```
resultado = concatenar [1,2,3] ['a','b','c']
```

El compilador no puede determinar de manera única el tipo de 'a' porque podría ser tanto `Int` como `Char`. Esto resulta en el error "Ambiguous type variable 'a' in the constraints".

Para corregir este error, necesitas hacer que el tipo de 'a' sea más específico o proporcionar más información al compilador para ayudarlo a inferir el tipo correctamente. Puedes hacer esto cambiando la función para que solo funcione con un tipo específico, o utilizando anotaciones de tipo explícitas para especificar el tipo de 'a'. Por ejemplo:

```
concatenar :: [Int] -> [Char] -> [a]
concatenar xs ys = xs ++ ys
```

o

```
resultado = concatenar [1,2,3 :: Int] ['a','b','c']
```

Entonces, el error "Ambiguous type variable 'a' in the constraints" ocurre cuando el compilador no puede determinar de manera única el tipo de una variable 'a'. Esto suele ser el resultado de una ambigüedad en la inferencia de tipos y se puede corregir haciendo que el tipo de 'a' sea más específico o proporcionando más información al compilador.

7 Monomorphism Restriction (Restricción de monomorfismo):

Advertencia/Error: "Monomorphism restriction"

Cuando recibes la advertencia o error "The monomorphism restriction..." en Haskell, se refiere a una regla de inferencia de tipos que limita la generalización de variables polimórficas en ciertos contextos. Esta restricción se implementa para evitar problemas de rendimiento y ambigüedad en la inferencia de tipos.

La monomorfización es el proceso de restringir variables polimórficas a tipos concretos. Esto significa que las variables polimórficas se especializan a tipos específicos en lugar de permanecer como variables genéricas. La monomorfización se aplica automáticamente por el compilador Haskell en ciertos casos para mejorar el rendimiento y evitar ambigüedades en la inferencia de tipos.

La advertencia o error "The monomorphism restriction..." generalmente ocurre en los siguientes casos:

1. **Asignaciones de variables**: Cuando defines una función o un valor utilizando asignaciones de variables sin especificar explícitamente su tipo y el tipo de la variable se vuelve ambiguo debido a la monomorfización.

2. **Funciones con tipos de datos polimórficos**: Cuando defines una función que toma argumentos polimórficos pero el compilador no puede generalizar el tipo de la función debido a la monomorfización.

Por ejemplo, considera el siguiente código:

```
f x = x + 1
```

En este caso, la función `f` toma un argumento `x` y devuelve `x + 1`. Si no especificamos el tipo de `x`, el compilador puede aplicar la monomorfización y deducir un tipo específico para `x`, lo que podría no ser lo que esperábamos. Para evitar esta advertencia, podríamos especificar explícitamente el tipo de `x`:

```
f :: Num a => a -> a
f x = x + 1
```

Esto asegura que `x` sea de tipo numérico (`Num`) pero sigue siendo polimórfico en el sentido de que puede ser cualquier tipo numérico.

8 No Instance for (Num a) arising from a use of 'someFunction':

Error: "No instance for (Num a) arising from a use of 'someFunction'"

El error "No instance for (Num a) arising from a use of 'someFunction'" en Haskell indica que el compilador no puede encontrar una instancia de la clase de tipos `Num` para el tipo de dato `a` en el contexto de uso de la función `someFunction`.

La clase de tipos `Num` en Haskell representa los tipos numéricos, como `Int`, `Integer`, `Float`, `Double`, entre otros. Cuando se encuentra este error, significa que estás utilizando una función que espera que su argumento tenga un tipo que sea una instancia de la clase `Num`, pero el tipo específico que estás proporcionando no es uno de esos tipos numéricos.

Por ejemplo, considera la siguiente función:

```
doubleValue :: Num a => a -> a
doubleValue x = x * 2
```

Esta función `doubleValue` espera un argumento de cualquier tipo que sea una instancia de la clase `Num`, y luego multiplica ese valor por 2. Sin embargo, si intentamos llamar a esta función con un argumento que no es numérico, obtendremos el error "No instance for (Num a) arising from a use of 'doubleValue'".

Por ejemplo:

```
stringValue = doubleValue "hello"
```

En este caso, estamos intentando pasar una cadena de caracteres `"hello"` a la función `doubleValue`, pero las cadenas de caracteres no son tipos numéricos y, por lo tanto, no son instancias de la clase `Num`. Esto provoca el error mencionado.

Para corregir este error, debes asegurarte de que el tipo del argumento que estás pasando a la función `someFunction` sea un tipo numérico, o bien, si la función debería aceptar otros tipos, necesitarás modificar la firma de tipo de la función para que sea más general o proporcionar una implementación específica para los tipos no numéricos.

9 Undefined Function or Variable (Función o variable no definida):

Error: "Not in scope: 'functionName'"

El error "Not in scope: 'functionName'" en Haskell indica que estás intentando utilizar una función llamada 'functionName' que no está definida en el ámbito actual del código. Esto puede ocurrir por varias razones:

1. **Falta de definición**: No has definido la función 'functionName' en ninguna parte del código.
2. **Ámbito incorrecto**: La función 'functionName' puede haber sido definida en otro módulo o lugar del código que no está accesible desde el punto donde estás intentando utilizarla.
3. **Error tipográfico**: Puede haber un error tipográfico en el nombre de la función, lo que hace que el compilador no la reconozca.

Veamos un ejemplo para entenderlo mejor. Supongamos que tienes el siguiente código:

```
main = do
  putStrLn "Hello, world!"
  result <- someFunction
  print result
```

Si la función `someFunction` no está definida en ningún lugar del código o en ningún módulo importado, obtendrás el error "Not in scope: 'someFunction'".

Para corregir este error, debes asegurarte de que la función 'functionName' esté definida en un ámbito accesible desde el punto donde estás intentando utilizarla. Esto puede implicar definir la función en el mismo archivo o módulo donde la estás utilizando, o importar el módulo que contiene la definición de la función utilizando una declaración `import`.

Por ejemplo, si `someFunction` está definida en el módulo `Module`, puedes corregir el error importando el módulo en tu archivo Haskell:

```
import Module

main = do
  putStrLn "Hello, world!"
  result <- someFunction
  print result
```