Lenguajes de Programación (2024/2)
Profesores: Federico Olmedo Ismael Figueroa
Auxiliares: Cristian Carrión Bastián Corrales Damián Arquez
Ayudantes: David Ibáñez Javier Kauer Martín Segur
Camilo Urzúa Daniela Assael Kathleen Kohler

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

Tarea 3
Pattern Matching

Para la resolución de la tarea recuerde que

- Toda función debe estar acompañada de su firma, una breve descripción coloquial (en `T3.rkt`) y un conjunto significativo de tests (en `test.rkt`).
- Todo datatype definido por el usuario (via `deftype`) debe estar acompañado de una breve descripción coloquial y de la gramática BNF que lo genera.

Si la función o el datatype no cumple con estas reglas, será ignorado.

Ejercicio 1 **46 Pt**

El objetivo de esta tarea es implementar pattern matching en un lenguaje funcional, como el visto en clases.

El archivo `T3.rkt` contiene una definición inicial del tipo de datos recursivo `Expr` con literales numéricos y adición.

(a) [4 Pt] Extienda `Expr` (y complete su gramática BNF) con pares y un constructor `nil`. Para ello, utilice los siguientes ejemplos como guía (no es necesario implementar la función `parse` aún):

```
>>> (parse '(nil))
(nil)
>>> (parse '(cons 1 2))
(conz (num 1) (num 2))
```

(b) [6 Pt] Implemente la función `parse` de acuerdo a los ejemplos de la parte (a). Además, implemente el azúcar sintáctico para escribir listas en función de `conz` y `nil`:

```
>>> (parse '(list 1 2 3))
(conz (num 1) (conz (num 2) (conz (num 3) (nil))))
```

Recuerde que para hacer pattern matching sobre una lista de cualquier cantidad de elementos puede utilizar la *elipsis* (escrito ...) de la siguiente manera:

```
>>> (match '(2 3 4 5)
        [(list x elems ...) (first elems)]) ; x = 2
3
>>> (match '(2 (3 4 5))
        [(list x (list y rest ...)) (+ x (last rest))]) ; x = 2, y = 3
7
```

En el primer ejemplo, la variable `elems` es una lista que contiene todos los elementos excepto el 2. En el segundo ejemplo, la variable `rest` es una lista que contiene los números 4 y 5.

(c) [4 Pt] La forma más directa de integrar pattern matching en un lenguaje funcional es dándole la capacidad de capturar patrones a las funciones anónimas, es decir, la forma general de una función se vuelve `(fun p e)` donde `p` es un patrón y `e` es el cuerpo de la función. Luego, al aplicar dicha función a un argumento, éste debe calzar con el patrón especificado; de lo contrario, el programa lanza un error.
Complete la definición de `Pattern` (y su gramática BNF) siguiendo los siguientes ejemplos (no es necesario implementar la función `parse-pattern` aún):

- **Patrones constantes**: una expresión hace match si es exactamente igual a la constante especificada.

```
>>> (parse-pattern '3)
(numP 3)
>>> (parse-pattern '(nil))
(nilP)
```

- **Patrones variables**: cualquier expresión hace match con una variable.

```
>>> (parse-pattern 'x)
(varP 'x)
```

- **Patrones pares**: una expresión `e` hace match un patrón de la forma `(cons p1 p2)` si a su vez tiene la forma sintáctica `(cons e1 e2)` donde `e1` hace match con `p1` y `e2` hace match con `p2`.

```
>>> (parse-pattern '(cons 1 x))
(conzP (numP 1) (varP 'x))
```

(d) [4 Pt] Implemente la función `parse-pattern` de acuerdo a los ejemplos de la parte (c). Además, implemente el azúcar sintáctico para escribir patrones sobre listas en función de `conzP` y `nilP`:

```
>>> (parse-pattern '( list  1 x 3))
(conzP (numP 1) (conzP (varP 'x) (conzP (numP 3) (nilP))))
```

(e) [4 Pt] Extienda la definición de `Expr` (con su gramática BNF) y `parse` con funciones de primera clase y aplicación de funciones:

```
>>> (parse '(fun x x))
(fun (varP 'x) (id 'x))
>>> (parse '(fun (cons x xs) x))
(fun (conzP (varP 'x) (varP 'xs)) (id 'x))
>>> (parse '(f x))
(app (id 'f) (id 'x))
```

(f) [2 Pt] El archivo `T3.rkt` cuenta con una definición inicial de `Value`. Ahora que hemos extendido el lenguaje con `nil`, pares y funciones de primera clase, es necesario extender esta definición. Para ello, considere que (1) `(nil)` es un valor y (2) un par se considera valor cuando sus componentes también son valores. Además, queremos implementar scope estático con substitución diferida, por lo tanto es necesario utilizar clausuras como la representación de valores para funciones. Extienda `Value` para que capture la nueva noción de valores del lenguaje, y escriba su gramática.

(g) [10 Pt] El principal desafio de un lenguaje con pattern matching es determinar si un valor calza con un patrón dado, encontrando además el conjunto de asociaciones entre variables y valores. Por ejemplo, el valor `(cons 1 (cons 2 3))` calza con el patrón `(cons x y)`, donde implícitamente se asocia `x` con `1` e `y` con `(cons 2 3)`. De manera dual, el valor `(cons 1 2)` **no calza** con el patrón `(cons x (nil))`, porque `2` no calza con `(nil)`.

En general, en un lenguaje con pattern matching, es sumamente importante saber porque un valor **no hizo match** con un patrón. Por lo tanto, en este ejercicio, además de enfocarnos en saber si un valor calza con un patrón, también pondremos énfasis en saber cual fue error de *matching*. Para lograrlo, utilizaremos el tipo de datos `Result` (definido en `T3.rkt`) que permite retornar un resultado exitoso (`(success v)`) o un mensaje de error (`(failure e)`), sin necesidad de lanzar un error explícitamente. Por ejemplo, utilizando `Result`, podríamos definir una división *safe* que previene la división por cero:

```
;; safe-/ : Number Number -> (Result String Number)
(define (safe-/ x y)
  (if (= y 0) (failure "DivisionByZero") (success (/ x y))))
```

El tipo de retorno `(Result String Number)` especifica que la función retorna un string en caso de falla, o un número en caso de éxito.

Al definir una función utilizando este enfoque, se le permite a quien invoca la función *safe* determinar qué hacer en caso de falla, es decir, el manejo de errores se vuelve explícito.

Implemente la función `generate-substs` de tipo `Pattern Value -> (Result String (Listof (Symbol * Value)))`, que a partir de un patrón y un valor, retorna (1) una lista de substituciones, o (2) un mensaje de error.

Importante: Asuma que en un patrón nunca se repite el nombre de una variable.

```
>>> (generate-substs (numP 3) (numV 3))
(success '())
>>> (generate-substs (numP 3) (numV 4))
(failure "MatchError: given number does not match pattern")
>>> (generate-substs (varP 'x) (numV 3))
(success (list (cons 'x (numV 3))))
```

En particular, su función debe ser capaz de retornar los siguientes mensajes de error:

- `"MatchError: given number does not match pattern"`

- `"MatchError: expected a number"`
- `"MatchError: expected nil"`
- `"MatchError: expected a cons constructor"`

(h) [12 Pt] Implemente la función `interp` que reduce una expresión (`Expr`) en un valor del lenguaje (`Value`).
En el caso de aplicación de función, si el argumento no calza con el patrón especificado por la función, se debe lanzar el error de *matching* correspondiente.

Ejercicio 2 **14 Pt**

Ahora extenderemos el lenguaje con una expresión `match` que permita hacer análisis de casos, basado en patrones.

(a) [2 Pt] Extienda `Expr` (y su gramática BNF) con una expresión `match`. Para ello, utilice los siguientes ejemplos como guía (no es necesario extender la función `parse` aún):

```
>>> (parse '(match 2 [1 1] [x 3]))
(pmatch (num 2) (list (cons (numP 1) (num 1)) (cons (varP 'x) (num 3))))
```

(b) [4 Pt] Extienda la función `parse` de acuerdo a los ejemplos de la parte (a).
Una expresión `match` debe siempre recibir **al menos un caso**. De lo contrario, la función `parse` debe lanzar una excepción:

```
>>> (parse '(match (list 1 2 3)))
SyntaxError: match expression must have at least one case
```

(c) [6 Pt] Extienda la función `interp` para poder reducir una expresión `match`. Dicha expresión debe testear los patrones en el orden especificado y ejecutar el cuerpo del primer caso que calce. Si ningún patrón calza, se debe lanzar un error con el mensaje `"MatchError: expression does not match any pattern"`.

(d) [2 Pt] En función de lo implementado en la pregunta anterior, argumente porqué es útil que la función `generate-subst` no lance un error (cuando el valor no calza con el patrón) y, en cambio, retorne un mensaje.