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 2
Parsing e Intérpretes

Para la resolución de la tarea recuerde que

- Toda función debe estar acompañada de su firma, una breve descripción coloquial (en `T2.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 **30 Pt**

El archivo `T2.rkt` contiene una definición inicial del tipo de datos recursivo `Prop` con literales booleanos.

(a) [4 Pt] Extienda `Prop` (y su gramática BNF) con operadores booleanos (not, or y and) y definiciones locales. Para ello, utilice los siguientes ejemplos como guía (no es necesario implementar la función `parse-prop` aún):

- `(p-not p)`, `(p-and ps)` y `(p-or ps)`: Operadores booleanos.

```
>>> (parse-prop '(not true))
(p-not (tt))
>>> (parse-prop '(and true false true))
(p-and (list (tt) (ff) (tt)))
>>> (parse-prop '(or true false))
(p-or (list (tt) (ff)))
```

- `id` y `(expr where [id expr])`: Identificadores y definiciones locales.

```
>>> (parse-prop '(false where [x true]))
(p-where (ff) 'x (tt))
>>> (parse-prop '(x where [x true]))
(p-where (p-id 'x) 'x (tt))
```

(b) [8 Pt] Implemente la función `parse-prop` de acuerdo a los ejemplos de la parte (a). Los operadores `and` y `or` deben siempre recibir **al menos dos operandos**. De lo contrario, la función `parse-prop` debe lanzar una excepción:

```
>>> (parse-prop '(and true))
parse-prop: and expects at least two operands
>>> (parse-prop '(or))
parse-prop: or expects at least two operands
```

Adicionalmente, para hacer pattern matching sobre una lista de cualquier cantidad de elementos (como en el caso de `and` y `or`) puede utilizar la *elipsis* (escrito `...`) de la siguiente manera:

```
>>> (match '(2 3 4 5)
       [(list x elems ...) (first elems)]) ; x captura el 2
3
>>> (match '(2 (3 4 5))
       [(list x (y rest ...)) (+ x (last rest))]) ; x captura 2, y captura 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) [2 Pt] Defina el tipo de datos recursivo `PValue` que capture la noción de valores del lenguaje, y escriba su gramática. Adicionalmente, implemente la función `(from-PValue)` que convierte un `PValue` en elemento `Prop`.

```
>>> (from-PValue (ttV))
(tt)
>>> (from-PValue (ffV))
(ff)
```

(d) [6 Pt] Implemente la función `p-subst` que realiza la substitución de una proposición por un identificador.

```
>>> (p-subst (p-id 'x) 'x (tt))
(tt)
>>> (p-subst (p-id 'x) 'y (tt))
(p-id 'x)
>>> (p-subst (p-where (p-id 'x) 'x (tt)) 'x (ff))
(p-where (p-id 'x) 'x (tt))
>>> (p-subst (p-where (p-id 'x) 'y (tt)) 'x (ff))
(p-where (ff) 'y (tt))
```

(e) [10 Pt] Implemente la función `p-eval` que reduce una proposición (`Prop`) en un valor del lenguaje (`PValue`).
Su interprete debe implementar la funcionalidad de corto circuito, es decir:

- Si se está evaluando una expresión `and` y uno de los operandos reduce a `false` (más específicamente `(ffV)`), entonces toda la expresión debe reducir a `false`, sin necesidad de seguir reduciendo los siguientes operandos.

- De la misma manera, si se está evaluando una expresión `or` y uno de los operandos reduce a `true` (más específicamente `(ttV)`), entonces toda la expresión debe reducir a `true`, sin necesidad de seguir reduciendo los siguientes operandos.

Para implementar el corto circuito de `and` (y de `or`) se le sugiere implementar

una función auxiliar, mutuamente recursiva con `p-eval`, `eval-and` (y `eval-or`, respectivamente) de tipo `(Listof Prop) -> PValue`.
Asegúrese de escribir los tests para este comportamiento. Para ello, piense en cómo puede testear que **no** se evaluó una proposición.

Ejercicio 2 **30 Pt**

El archivo `T2.rkt` contiene una definición inicial del tipo de datos recursivo `Expr`, basado en el lenguaje visto en clases.

(a) [4 Pt] Extienda `Expr` (y su gramática BNF) con operadores números reales e imaginarios. Además, agregue una forma de introducir varias definiciones locales en una sola expresión, es decir, una forma de `with` con varias definiciones. Para ello, utilice los siguientes ejemplos como guía (no es necesario implementar la función `parse` aún):

- `n` y `(n i)`: Números reales e imaginarios.

```
>>> (parse '1)
(real 1)
>>> (parse '(1 i))
(imaginary 1)
>>> (parse '(+ 1 (2 i)))
(add (real 1) (imaginary 2))
```

- `id` y `(with [(id expr)*] expr)`: Identificadores y definiciones locales.

```
>>> (parse '(with [(x 1) (y 1)] (+ x y)))
(with (list (cons 'x (real 1)) (cons 'y (real 1))) (add (id 'x) (id 'y)))
```

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

```
>>> (parse '(with [] 1))
parse: 'with' expects at least one definition
```

(c) [4 Pt] En el archivo base, se encuentra una definición de valores para nuestro lenguaje: `CValue`. Dado que nuestro lenguaje soporta números reales e imaginarios, naturalmente los valores son los números complejos (o números reales o imaginarios, representados por un número complejo con algún componente en 0). Implemente la función `(from-CValue)` que convierte un `CValue` en un elemento `Expr`. Además, implemente las funciones `cmplx+`, `cmplx-` y `cmplx0`. Las primeras dos funciones suman y restan, respectivamente, dos valores de nuestro lenguaje. La última función retorna `true` si el número complejo es 0.

```
>>> (cmplx+ (compV 1 2) (compV 3 4))
(compV 4 6)
>>> (cmplx- (compV 1 2) (compV 3 4))
(compV -2 -2)
>>> (cmplx0? (compV 0 1))
#f
```

(d) [10 Pt] Implemente la función `subst` que realiza la substitución de una expresión por un identificador. Note que una expresión `with` introduce identificadores que pueden ser utilizados en próximas definiciones. Por ejemplo, la siguiente expresión es válida:

```
(with [(x 2) (y (+ x 1))] (+ x y))
```

Note que la definición de `y` utiliza `x`.
Su función de substitución debe tomar esto en cuenta y evitar substituir variables que han sido "oscurecidas" por otra definición local. Por ejemplo:

```
;; este ejemplo no tiene "shadowing"
>>> (subst (parse '(with [(x 2) (y z)] (+ x z))) 'z (real 1))
(with (list (cons 'x (real 2)) (cons 'y (real 1))) (add (id 'x) (real 1)))

;; en este ejemplo si ocurre "shadowing" -> no se realiza la substitucion
>>> (subst (parse '(with [(x 2) (y x)] (+ x x))) 'x (real 1))
(with (list (cons 'x (real 2)) (cons 'y (id 'x))) (add (id 'x) (id 'x)))
```

Hint Le puede ser útil definir una función auxiliar que realice una substitución sobre una lista de definiciones.

(e) [8 Pt] Implemente la función `interp` que reduce una expresión (`Expr`) en un valor del lenguaje (`CValue`).