Lenguajes de Programación
Práctica 4

Karla Ramírez Pulido Manuel Soto Romero Alejandro Hernández Mora
Alma Rocío Sánchez Salgado Silvia Díaz Gómez

Semestre 2021-1

Facultad de Ciencias, UNAM
Fecha de inicio: 25 de noviembre de 2020
Fecha de entrega: 8 de enero de 2020

1. Objetivos

Implementar el intérprete para el lenguaje ***CFWAE*** con ambientes. Para llevar a cabo esta tarea, se debe completar el cuerpo de las funciones faltantes dentro de los archivos ***grammars.rkt***, ***parser.rkt*** e ***interp.rkt***[1].

La gramática del lenguaje `CFWAE` se presenta a continuación:

```
<expr> ::= <id>
| <num>
| {<op> <expr>+}
| {if0 <expr> <expr> <expr>}
| {with {{<id> <expr>}+} <expr>}
| {with* {{<id> <expr>}+} <expr>}
| {fun {<id>*} <expr>}
| {<expr>}{<expr>*}
<id> ::= a | b | c | ...
<num> ::= 1 | 2 | 3 | ...
<op> ::= + | - | * | / | modulo | expt | add1 | sub1
```

La gramática anterior se define mediante los siguientes tipos en *Racket*.

```
;; Definición del tipo Binding
(define-type Binding
    [binding (id symbol?) (value CFWAE?)])

;; Definición del tipo CFWAE
(define-type CFWAE
    [id   (i symbol?)]
    [num  (n number?)]
    [op   (f procedure?) (args (listof CFWAE?))]
    [if0  (condicion CFWAE?) (then CFWAE?) (else CFWAE?)]
```

[1] Puedes reutilizar los archivos que entregaste en la práctica anterior, agregando los nuevos tipos incluidos en esta práctica

```
  [with (bindings (listof binding?)) (body CFWAE?)]
  [with* (bindings (listof binding?)) (body CFWAE?)]
  [fun (params (listof symbol?)) (body CFWAE?)]
  [app (fun CFWAE?) (args (listof CFWAE?))])
```

2. Ejercicios

1. (4.5 pts.) Completar el cuerpo de la función `(parse sexp)` dentro del archivo `parser.rkt`. Considerando la función parse de la práctica anterior, agrega las expresiones que aparecen en la nueva gramática.

```
;; parse: s-expression -> CFWAE
(define (parse sexp) ...)
```

   A continuación vemos la gramática y las definiciones de tipos en *Racket* correspondientes al lenguaje de la práctica anterior[2]. El tipo `binding` se mantiene igual en esta práctica.

```
;;Gramática del lenguaje WAE
<expr> ::= <id>
| <num>
| {<op> <expr>+}
| {with {{<id> <expr>}+} <expr>}
| {with* {{<id> <expr>}+} <expr>}
<id>  ::= a | b | c | ...
<num> ::= 1 | 2 | 3 | ...
<op>  ::= + | - | * | / | modulo | expt | add1 | sub1

;; Definición del tipo WAE
(define-type WAE
    [id    (i symbol?)]
    [num   (n number?)]
    [op    (f procedure?) (args (listof WAE?))]
    [with  (bindings (listof binding?)) (body WAE?)]
    [with* (bindings (listof binding?)) (body WAE?)])
```

2. (1 pts.) Completar el cuerpo de la función `(lookup name ds)` dentro del archivo `interp.rkt` la cual busca el el nombre de variable `name` dentro del caché de sustitución `ds`, regresando el valor correspondiente a la variable o arrojando un error en caso de que no lo encuentre.

```
;;(define (lookup name ds)
(define (lookup name ds) ...)
```

   El caché de sustitución guarda símbolos de variables y sus valores, que son expresiones del lenguaje. La sintaxis abstracta del caché de sustituciones se modela mediante los siguientes tipos en *Racket*:

[2] Esto es como referencia, para que tengan toda la información en un mismo documento.

```
;; Data-type que representa un caché de sustituciones
(define-type DefrdSub
    [mtSub]
    [aSub  (name symbol?) (value CFWAE-Value?) (ds DefrdSub?)])
```

3. (4.5 pts.) Completar el cuerpo de la función `(interp expr ds)` dentro del archivo `interp.rkt` el cual recibe una expresión, un caché de sustituciones y regresa la evaluación correspondiente de acuerdo con las variables definidas en el caché. Deberás considerar las modificaciones hechas a la gramática.

```
;; interp: CFWAE DefrdSub -> CFWAE-Value
(define (interp expr ds) ...)
```

Para la implementación de `interp`, deberás considerar los siguientes puntos:

- **Condicionales evaluadas a cero (`if0`)**
  El `if0` es una estructura de control que nos permite decidir si ejecutamos una expresión u otra. Está formado de tres partes: `expresión-condicional`, `expresión-then` y `expresión-else`. Ya que aún no contamos con valores de tipo booleano, dependiendo de la evaluación ejecutaremos la `expresión-then`, o la `expresión-else`. Si `expresión-condicional` se evalúa a cero, entonces se ejecuta la `expresión-then`, en otro caso se ejecuta la instrucción `expresión-else`.
- **Funciones (`fun`)**
  Las funciones reciben una lista con cero o más argumentos. La definición asume que el mismo número de argumentos recibidos en una invocación, es el mismo que el número de elementos en la definición del procedimiento. Si la aridad de los argumentos difiere de la aridad de los parámetros, debemos regresar un error.
  La interpretación de una función devuelve un `closure` con la definición de la función y el ambiente en el que se evaluará cuando se requiera, es decir, con el caché de sustitución que exista durante su definición. Por ejemplo:

```
> (interp (parse '(fun (x y z) (+ x y z))) (mtSub))
(closure '(x y z) (op #<procedure:+> (list (id 'x) (id 'y) (id 'z))) (mtSub))
```

- **Asignaciones locales simples (`with`)**
  Deberás modificar la evaluación de `with` de manera que deje de ser parte de la sintaxis abstracta (azúcar sintáctica) y se convierte en una aplicación de función. Por ejemplo:

```
'{with {{x 2} {y 3} {z 4}} {+ x y z}}}
```

  Se transformará en:

```
(app
    (fun (list 'x 'y 'z)
        (op #<procedure:+> '((id 'x) (id 'y) (id 'z))))
(list 1 2 3))
```

- **Asignaciones locales anidadas (`with*`)**
  El `with*` seguirá formando parte de la sintaxis abstracta, así que la evaluación de éste deberá funcionar de la siguiente manera.
  - Para poder evaluar un `id x` en términos de otro `id y`, es necesario que se defina primero el `id y` dentro de la lista de bindings del `with*`.
  - Si un `id x` es definido dos veces en la lista de ligaduras (lista de bindings), éste tomará el último valor en la lista de bindings del `with*` (considerando el orden de izquierda a derecha). Si el valor de un `id y` depende del valor de `x`, el `id y` tomará la asignación del `id x` que esté más cercano a la izquierda de la aparición de `id y`. En otras palabras, si un id depende del valor de otro, que posteriormente es modificado, dicha modificación no afecta el valor del id dependiente. Por ejemplo:

    ```
    (with* ([y {+ x x}] [x 1]) (y)) = "Error: Variable libre: var x"
    (with* ([x 2] [y {+ x x}] [x 1]) (+ 0 y)) = 4
    (with* ([x 2] [y {+ x x}] [x 1]) (+ x y)) = 5
    ```

Se deberán respetar todas las definiciones de las funciones, es decir, está prohibido cambiar el tipo que tengan, ya sea el de algún parámetro o el del resultado.

3. Requerimientos

Se deberá subir los archivos requeridos a la plataforma Google Classroom antes de las 23:59 hrs. del día de la fecha de entrega, conforme lo como lo indican los lineamientos de entrega. No olvides incluir el archivo `ReadMe.txt` con los datos de tus compañeros. Sólo es necesario que una persona suba la práctica, los demás compañeros deberán subir como tarea el contenido del archivo `ReadMe.txt`, de manera que se puedan leer los nombres de los integrantes del equipo desde la plataforma Google Classroom.

El orden en el que aparezcan las funciones en los archivos solicitados, debe ser el orden especificado en este archivo PDF, de lo contrario podrán penalizarse algunos puntos.

La idea es que reutilices funciones de los ejercicios que ya se programaron anteriormente.

¡Que tengas éxito en tu práctica!.