Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación

Práctica 8

Karla Ramírez Pulido
karla@ciencias.unam.mx

J. Ricardo Rodríguez Abreu
ricardo_rodab@ciencias.unam.mx

Manuel Soto Romero
manu@ciencias.unam.mx

Luis A. Patlani Aguilar
ayudantefc@gmail.com

Fecha de inicio: 29 de noviembre de 2017
Fecha de término: 12 de diciembre de 2017
Semestre 2018-1

Objetivos

- Agregar primitivas para manipular cajas al lenguaje de la Práctica 7.
- Modificar el intérprete implementado en la Práctica 7 para que manipule cajas e instrucciones imperativas mediante la técnica *Store Passing Style*.

Descripción general

La práctica consiste en extender el intérprete de la práctica anterior. La gramática en EBNF para las expresiones del lenguaje `BERCFWBAEL/L` (*Boxes Exceptions Recursive Conditionals functions With Boolean Arithmetic Expressions Lists and Lazy*), que se debe implementar en esta práctica es la siguiente:

```
<expr> ::= <id>
         | <num>
         | <bool>
         | <list>
         | {<op> <expr>+}
         | {if <expr> <expr> <expr>}
         | {cond {<expr> <expr>}+ {else <expr>}?}
         | {with {{<id> <expr>}+} <expr>}
         | {with* {{<id> <expr>}+} <expr>}
         | {rec {{<id> <expr>}+} <expr>}
         | {fun {<id>*} <expr>}
         | {<expr> <expr>*}
         | {throws <id>}
         | {try/catch {{<id> <expr>}+} <expr>}
         | {newbox <expr>}
```

```
          | {openbox <expr>}
          | {setbox <expr> <expr>}
          | {seqn <expr> <expr>+}

<id> ::= a |... | z | A | ... | Z | aa | ab | ... | aaa | ...
         (Cualquier combinación de caracteres alfanuméricos
          con al menos uno alfabético)

<num> ::= ... | -2 | -1 | 0 | 1 | 2 | ...
         (Cualquier número admitido por Racket)

<bool> ::= false | true

<list> :: empty
        | {list <expr>+}

<op> ::= + | - | * | / | % | min | max | pow | zero?
       | not | and | or | < | > | <= | >= | = | /=
       | head | tail | empty?
```

Archivos requeridos

El material de esta práctica consta de los siguientes archivos[1]:

- `grammars.rkt` archivo con la definición de los tipos de datos abstractos que definen cada uno de los ASA para el lenguaje `BERCFWBAEL/L`.
- `parser.rkt` archivo en el cual se debe implementar el analizador sintáctico para el lenguaje `BERCFWBAEL/L`.
- `interp.rkt` archivo donde se debe implementar el analizador semántico para el lenguaje `BERCFWBAEL/L`.
- `practica.rkt` archivo con la lógica necesaria para ejecutar el intérprete final de `BERCFWBAEL/L`.
- `test-practica8.rkt` archivo en el cual se deben agregar las pruebas unitarias de la práctica.

Desarrollo de la práctica

Ejercicio 7.1 (0.5 pts.) En el archivo `grammars.rkt` se encuentra el TDA que define los constructores para realizar el análisis sintáctico del lenguaje `BERCFWBAEL/L`.

[1]Los archivos pueden descargarse desde la página del curso `http://lenguajesfc.com`

```
;; TDA para representar los árboles de sintáxis abtracta del
;; lenguaje BERCFWBAEL/L. Este TDA es una versión con azúcar sintáctica.
(define-type BERCFWBAEL/L
   [idS (i symbol?)]
   [numS (n number?)]
   [boolS (b boolean?)]
   [listS (elems (listof BERCFWBAEL/L?))]
   [opS (f procedure?) (args (listof BERCFWBAEL/L?))]
   [ifS (expr BERCFWBAEL/L?) (then-expr BERCFWBAEL/L?) (else-expr BERCFWBAEL/L?)]
   [condS (cases (listof Condition?))]
   [withS (bindings (listof bindingS?)) (body BERCFWBAEL/L?)]
   [withS* (bindings (listof bindingS?)) (body BERCFWBAEL/L?)]
   [recS (bindings (listof bindingS?)) (body BERCFWBAEL/L?)]
   [funS (params (listof symbol?)) (body BERCFWBAEL/L?)]
   [appS (fun-expr BERCFWBAEL/L?) (args (listof ERCFWBAEL/L?))]
   [throwsS (exception-id symbol?)]
   [try/catchS (bindings (listof bindingS?)) (body BERCFWBAEL/L?)]
   [newboxS (contents BERCFWBAEL/L?)]
   [openboxS (box BERCFWBAEL/L?)]
   [setboxS (box BERCFWBAEL/L?) (contents BERCFWBAEL/L?)]
   [seqnS (actions (listof BERCFWBAEL/L?))]

;; TDA para asociar identificadores con valores con azúcar sintáctica.
(define-type BindingS
   [bindingS (name symbol?) (value BERCFWBAEL/L?)])

;; TDA para representar condiciones.
(define-type Condition
   [condition (expr BERCFWBAEL/L?) (then-expr BERCFWBAEL/L?)]
   [else-cond (else-expr BERCFWBAEL/L?)])
```

El primer ejercicio a implementar en el intérprete de esta práctica, consiste en completar el cuerpo de la función `parse` (incluida en el archivo `parser.rkt`) para que realice el análisis sintáctico correspondiente. Además, se deben agregar al menos cinco pruebas unitarias dentro del archivo `test-practica7.rkt`[2].

```
;; parse: s-expression -> BERCFWBAEL/L
(define (parse sexp)
   (error 'parse "Función no implementada."))
```

```
> (parse '{box 1729})
(newboxS (num 1729))
```

Ejercicio 7.2 (0.5 pts.) Una vez que se complete el cuerpo de la función `parse`, implementar el cuerpo de la función `desugar` incluída en el archivo `parser.rkt`, el cual elimina el azúcar sintáctica[3] de las expresiones de `BERCFWBAEL/L`, es decir, las convierte en expresiones de `BERCFBAEL/L`:

[2]Se deben agregar pruebas significativas.

[3]Tipo de sintaxis que hace que un programa sea más "dulce" o fácil de escribir.

```
;; TDA para representar los árboles de sintaxis abstracta del
;; lenguaje BERCFBAEL/L. Este TDA es uan versión sin azúcar sintáctica.
(define-type BERCFBAEL/L
   [id (i symbol?)]
   [num (n number?)]
   [bool (b boolean?)]
   [lisT (elems (listof BERCFBAEL/L?))]
   [op (f procedure?) (args (listof BERCFBAEL/L?))]
   [iF (expr BERCFBAEL/L?) (then-expr BERCFBAEL/L?) (else-expr BERCFBAEL/L?)]
   [fun (params (listof symbol?)) (body BERCFBAEL/L?)]
   [rec (bindings (listof Binding?)) (body BERCFBAEL/L?)]
   [app (fun-expr BERCFBAEL/L?) (args (listof BERCFBAEL/L?))]
   [throws (exception-id symbol?)]
   [try/catch (bindings (listof Binding?)) (body BERCFBAEL/L?)]
   [newbox (contents BERCFBAEL/L?)]
   [openbox (box BERCFBAEL/L?)]
   [setbox (box BERCFBAEL/L?) (contents BERCFWBAEL/L?)]
   [seqn (actions (listof BERCFBAEL/L?))])

;; TDA para asociar identificadores con valores sin azúcar sintáctica.
(define-type Binding
   [binding (name symbol?) (value BERCFBAEL/L?)])
```

Se deben agregar al menos cinco pruebas unitarias dentro del archivo `test-practica8.rkt` que correspondan con las pruebas agregadas en el ejercicio anterior.

Las únicas expresiones que tienen azúcar sintáctica, en esta práctica, son:

- `with` estas expresiones son una versión endulzada de aplicaciones a función. Por ejemplo:

  ```
  {with {{a 3}}
     {+ a 4}}
  ```

  se transforma en

  ```
  {{fun {a} {+ a 4}} 3}
  ```

- `with*` este tipo de expresiones son una versión endulzada de expresiones `with` anidadas que a su vez tienen azúcar sintáctica. Por ejemplo:

  ```
  {with* {{a 2} {b {+ a a}}}
     b}
  ```

  se transforma en

  ```
  {with {{a 2}}
     {with {{b {+ a a}}}
        b}}
  ```

  que a su vez se convierte en

```
{{fun {a} {{fun {b} b} {+ a a}}} 2}
```

- cond este tipo de expresiones son una versión endulzada de expresiones `if`. Por ejemplo:

```
{cond
   {{< 2 3} 1}
   {{> 10 2} 2}
   {else 3}}
```

se transforma en

```
{if {< 2 3}
    1
    {if {> 10 2}
        2
        3}}
```

```
;; desugar: BERCFWBAEL/L -> BERCFBAEL/L
(define (desugar expr)
   (error 'desugar "Función no implementada."))
```

```
> (desugar (parse '{cond {{< 2 3} 1} {{> 10 2} 2} {else 3}}))
(iF
    (op < (list (num 2) (num 3))
    (num 1)
    (iF (op > (list (num 10) (num 2))) (num 2) (num 3)))
```

Ejercicio 7.3 (8.5 pts.) El analizador semántico de esta práctica evalúa las expresiones mediante ambientes y cerraduras para implementar alcance estático, para implementar evaluación perezosa se agregan cerraduras de expresiones, para implementar recursividad se agregan ambientes recursivos que hacen uso de cajas y, para implementar el manejo de excepciones se hace uso de continuaciones finalmente, para interpretar las cajas, se utiliza la estructura llamada `Store` y se aplica la técnica *Store Passing Style*. De esta forma se recibe un ASA (sin azúcar sintáctica) y un ambiente de evaluación definido por:

```
;; TDA para representar los ambientes de evaluación.
(define-type Env
   [mtSub]
   [aSub (name symbol?) (location number?) (env Env?)])

;; TDA para representar el store
(define-type Store
   [mtSto]
   [aSto (index number?) (value BERCFBAEL/L-Value?) (sto Store?)])
```

Los resultados devueltos por el analizador semántico, que almacena el ambiente deben ser del tipo de dato BERCFBAEL/L-Value:

```
;; TDA para representar los resultados devueltos por el intérprete.
(define-type ERCFBAEL/L-Value
   [numV (n number?)]
   [boolV (b boolean?)]
   [closureV (params (listof symbol?)) (body ERCFBAEL/L?) (env Env?)]
   [exprV (expr ERCFBAEL/L?) (env Env?)]
   [listV (listof (ERCFBAEL/L-Value?))]
   [exceptionV (exception-id symbol?) (continuation continuation?)]
   [boxV (location number?)]
```

El ejercicio consiste en completar el cuerpo de la función `(interp exp env)` para que realice el análisis semántico correspondiente, es decir, evaluar expresiones de ERCFBAEL/L. Tomar los siguientes puntos a consideración (ver archivo `interp.rkt`):

- El valor de los identificadores debe ser buscando en el ambiente de evaluación mediante la definición de una función `lookup-env`, la cual (1) encuentra el valor asociado en el ambiente y lo regresa o bien (2) se reporta un error, en caso de que no se haya encontrado. En caso de recibir un ambiente recursivo esta función abre la caja asociada para seguir con la ejecución. Ejemplo:

  ```
  > (interp (desugar (parse 'foo)) (mtSub) (mtSto))
  lookup-env: Free identifier
  ```

- Los números se evalúan a valores de tipo `numV`. Ejemplo:

  ```
  > (interp (desugar (parse '1729)) (mtSub) (mtSto))
  (numV 1729)
  ```

- Las expresiones booleanas se evalúan a valores de tipo `boolV`. Ejemplo:

  ```
  > (interp (desugar (parse 'true)) (mtSub) (mtSto))
  (boolV #t)
  ```

- Las listas se evalúan a valores de tipo `listV`. Ejemplo:

  ```
  > (interp (desugar (parse '{list 1 2 3})) (mtSub) (mtSto))
  (listV (list (numV 1) (numV 2) (numV 3)))
  ```

- Los operadores aritméticos, booleanos y sobre listas son n-arios, por lo que esta versión del intérprete tiene un constructor `op` que recibe una función definida en Racket con la cual aplica la operación definida a cada uno de sus parámetros. En este tipo de expresiones se encuentra un punto estricto, se deben evaluar cada uno de sus operandos para aplicar el operador mediante la definición de una función `strict`. Los nuevos operandos para esta práctica son: `zero?`, `head`, `tail` y `empty?`. Ejemplo:

  ```
  > (interp (desugar (parse '{/= 1 2 3 4 5})) (mtSub) (mtSto))
  (boolV #t)
  ```

- Las expresiones `if` deben evaluar su condición y regresar el valor correspondiente en caso de que la condición se evalúe a verdadero o falso, según sea el caso. En este tipo de expresiones se encuentra un punto estricto, por lo que se debe evaluar la condición para regresar el valor correcto mediante la definición de una función `strict`. Ejemplo:

```
> (interp (desugar (parse '{if true true false})) (mtSub) (mtSto))
(boolV #t)
```

- Las expresiones `cond` deben evaluar cada condición y regresar el valor correspondiente en caso de que alguna sea verdadera, en caso contrario, se deben seguir evaluando el resto de condiciones o ejecutar el caso `else` si ninguna condición fue verdadera. Al ser azúcar sintáctica de expresiones `if`, cada condición de este tipo de expresiones también es un punto estricto. Ejemplo:

```
> (interp (desugar (parse '{cond {true 1} {true 2} {else 3}})) (mtSub) (mtSto))
(numV 1)
```

- Las expresiones `with` son multiparamétricas, es decir, tienen más de un identificador. Ejemplo:

```
> (interp (desugar (parse '{with {{a 2} {b 3}} {+ a b}})) (mtSub) (mtSto))
(numV 5)
```

- Las expresiones `with*` presentan un comportamiento parecido al de la primitiva `with`, sin embargo, estas expresiones permiten definir identificadores en términos de otros definidos anteriormente. Por ejemplo: `{with* {{a 2} {b {+ a a}}} b}`. Se interpreta similar a `with`, la única diferencia es que también se deben procesar los identificadores. Ejemplo:

```
> (interp (desugar (parse '{with* {{a 2} {b {+ a a}}} b})) (mtSub) (mtSto))
(exprV
    (op + (list (id 'a) (id 'a)))
    (aSub 'a (exprV (num 2) (mtSub)) (mtSub)))
```

- Las expresiones `rec` presentan un comportamiento parecido al de la primitiva `with*`, sin embargo, estas expresiones permiten definir identificadores recursivos, es decir, que se definen en términos de sí mismos. Por ejemplo: `{rec {{fac {fun {n} {if {zero? n} 1 {* n {fac {- n 1}}}}}} {n 5}} {fac n}}`. Se interpreta similar a `with*`, la diferencia es que se hace uso de la estructura `Store` para preservar el cuerpo de la función. Ejemplo:

```
> (define expr
    '{rec {
        {fac {fun {n}
            {if {zero? n}
                1
                {* n {fac {- n 1}}}}}}
        {n 5}}
        {fac n}})
> (interp (desugar (parse expr)) (mtSub) (mtSto))
(numV 120)
```

- Las funciones deben evaluarse a una cerradura que incluya los parámetros de la función, el cuerpo de la función y el ambiente dónde fue definida, con el fin de evaluarlas usando alcance estático. Ejemplo:

  ```
  > (interp (desugar (parse '{fun {x} {+ x 2}})) (mtSub) (mtSto))
  (closureV '(x) (op + (list (id 'x) (num 2)) (mtSub))
  ```

- En las aplicaciones de función se encuentra un punto estricto, éste deberá evaluar la función para poder aplicarla con los argumentos correspondientes mediante la definición de una función `strict`. Este tipo de expresiones deben de evaluar el cuerpo de la función correspondiente considerando el ambiente donde ésta fue definida y agregando las expresiones al ambiente sin evaluar pues es un intérprete perezoso. Por ejemplo, en la siguiente llamada, el ambiente de evaluación es:
  `(aSub 'a (exprV (op + (list (num 2) (num 3))) (mtSub)) (mtSub))`

  ```
  > (interp (desugar (parse '{{fun {x} {+ x 2}} {+ 2 3}})) (mtSub) (mtSto))
  (numV 7)
  ```

- Al lanzar excepciones se debe obtener un valor de tipo `exceptionV` que guarde el identificador de la excepción que está siendo lanzada y la pila de ejecución actual para conocer en qué punto del programa ocurrió el error, para ello se puede obtener la continuación actual usando `call/cc` o `let/cc` según corresponda.

  ```
  > (interp (desugar (parse '{throws DivisionEntreCero})) (mtSub) (mtSto))
  (exceptionV 'DivisionEntreCero #<continuation>)
  ```

- Para atrapar excepciones se debe evaluar el cuerpo del bloque `try/catch`, si éste genera un error, se debe obtener el tipo de excepción generado y compararlo con la lista de posibles excepciones para saber qué valor regresar en caso de que ocurra un valor de ese tipo. Para recuperarse del error, se debe aplicar la continuación capturada por el tipo `exceptionV` con el valor en caso de falla.

  ```
  > (define expr
      '{try/catch {{DivisionPorCero 2} {MiExcepcion 4}}
          {+ 1 {throws MiExcepcion}}})
  > (desugar (parse (interp expr (mtSub) (mtSto))))
  (numV 5)
  ```

- Al crear una caja se debe crear un nuevo registro en la estructura `Store`, para generar los índices de dicha estructura se recomienda definir una función `next-location` que genere secuencias de números enteros. Una vez interpretada una instrucción `newbox` se encontrará el contenido de la caja en el `Store` y se regresará la dirección en memoria de la misma.

  ```
  > (interp (desugar (parse '{newbox 1729})) (mtSub) (mtSto))
  (boxV 0)
  ```

- Al abrir una caja se debe buscar el índice en el `Store`, para ello se debe definir una función `lookup-sto` que devuelva el resultado asociado a un índice en el `Store`.

  ```
  > (interp (desugar (parse '{openbox {newbox 1729}})) (mtSub) (mtSto))
  (numV 1729)
  ```

- Al modificar una caja se debe crear un nuevo registro en el `Store` con la misma dirección en memoria de la caja que se está mutando, no se debe eliminar el registro anterior, pues la función `lookup-sto` busca el último valor agregado en el `Store`. Debido a que todas las expresiones del lenguaje deben regresar un valor, al modificar el valor de una caja, se optará por regresar el nuevo valor que se incluyó.

  ```
  > (interp (desugar (parse '{set {newbox 1729} 1835})) (mtSub) (mtSto))
  (numV 1835)
  ```

- La primitiva `seqn` debe permitir ejecutar *instrucciones* una después de otra. Esta versión de `seqn` permite múltiples instrucciones, debe ejecutar todas pero sólo debe regresarse el resultado de la última instrucción. Si funcionalidad es equivalente a la de la primitiva `begin` de Racket.

  ```
  > (define expr
      '{with {{a {newbox 1729}}}
         {seqn
           {setbox a 1835}
           {setbox a 405}
           {openbox a}}})
  > (desugar (parse (interp expr (mtSub) (mtSto))))
  (numV 405)
  ```

Además, se deben agregar al menos cinco pruebas unitarias dentro del archivo `test-practica8.rkt` que correspondan con las pruebas agregadas en los ejercicios anteriores.

```
;; interp: BERCFBAEL/L Env Store -> BERCFBAEL/L-Value
(define (interp expr env store)
   (error 'interp "Función no implementada."))
```

```
> (interp (desugar (parse '{box 1729})) (mtSub) (mtSto))
(boxV 1)
```

Ejercicio 7.4 (0.5 pt.) Una vez finalizados los ejercicios anteriores, modificar el archivo `practica7.rkt` para que procese los resultados devueltos por el intérprete. Ejecutarlo y realizar pruebas significativas para verificar que la práctica se completó con éxito.

```
Bienvenido a DrRacket, versión 6.10.1 [3m].
Lenguaje: plai, with debugging; memory limit: 128 MB.
(λ) {if {> 2 3} 1 2}
2
(λ) {cond {{> 2 3} 1} {{> 2 10} 2} {else 3}}
3
```

Figura 1: *Ejecución de* `RCFWBAEL/L`

Puntaje total: *10 puntos*

Entrega

Los archivos que se deben enviar a los ayudantes de laboratorio son:

- `parser.rkt`
- `interp.rkt`
- `practica8.rkt`
- `test-practica8.rkt`

Recordando seguir los lineamientos de entrega de prácticas especificados en la sección correspondiente de la página del curso: `http://lenguajesfc.com/lineamientos.html`.

Referencias

Algunas referencias de consulta:

[1] Karla Ramírez, Manuel Soto, *Notas del curso de Lenguajes de Programación*, Semestre 2018-1, Facultad de Ciencias, UNAM. Disponibles en: [`http://lenguajesfc.com/notas.html`].

[2] Shriram Krishnamurthi, *Programming Languages: Application and Interpretation*, Primera edición, Brown University, 2007.