Lenguajes de Programación

Práctica Extra - Estado

Semestre 2017-1

Fecha de inicio: 10 de diciembre de 2016

Fecha de término: 5 de enero de 2016

Instrucciones

- Resolver los siguientes ejercicios de manera clara y ordenada.
- Se deben modificar y enviar los archivos `grammars.rkt, parser.rkt` e `interp.rkt` de la práctica 6.
- La entrega es en equipos de máximo 3 integrantes. Seguir los lineamientos especificados en: `http://sites.ciencias.unam.mx/ldp171/formato`.

Descripción general

La mutación es «*el cambio de los valores asociados mediante los nombres*», ésta se encuentra como una característica en la mayoría de los lenguajes de programación. Si se hiciera un programa para modelar el mundo real, entonces dichos programas necesitarán ajustar el hecho de que los eventos del mundo real lo cambian y alteran.

El *store passing style* es una técnica para implementar el concepto de estado y será la que se usará en esta práctica. Se usará esta técnica y no las cajas de Racket ya que ésta nos permitirá analizar a detalle cómo se implementa el concepto de estado desde cero.

En las prácticas pasadas se usó un único repositorio para almacenar variables, el cuál llamamos *ambiente*. En esta práctica haremos uso de un segundo repositorio que por razones históricas llamaremos *store*.

Ejercicios

Se deben agregar modificaciones a las gramáticas, el analizador sintáctico y el intérprete de la **práctica 6** para que agregue aspectos imperativos al lenguaje.

Ahora en vez de almacenar en los ambientes *variable-valor*, se guardarán los pares *variable-ubicación*, donde *ubicación* hace referencia a la posición en el *store* en el cual estará almacenada dicha variable. Por otra parte el *store* almacenará los pares *ubicación-valor*. Si se desea hacer una operación que mute el estado de una caja, en vez de borrar la entrada del *store*, se deberá agregar una nueva, usando el mismo número de localidad. La forma de operar el *store* es igual a la de los ambientes, donde ambos presentan comportamiento de pila.

Por último se debe modificar el valor de regreso del intérprete para que no sólo regrese el valor de cada expresión sino también un *store* actualizado con los cambios que se pueden hacer debido a las mutaciones en el proceso para calcular el valor. A este nuevo tipo de regreso lo llamaremos *ValueXSotre*, pues es un valor pasado por el *store*.

La gramática del nuevo lenguaje objetivo `BRCFBAEL` (*Boxes, Recursion, Conditional, Functions, Booleans, Arithmetic expresions and Lists*) es la siguiente:

```
<expr> ::= <id>
         | <num>
         | <bool>
         | <list>
         | {with {{<id> <expr>}+} <expr>}
         | {rec {{<id> <expr>}+} <expr>}
         | {fun {<id>*} <expr>}
         | {if <expr> <expr> <expr>}
         | {<op> <expr>+}
         | {<expr> <expr>*}
         | {box <expr>}
         | {seqn <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 | ...

<bool> ::= true | false

<list> ::= empty
       | {cons <expr> <expr>}

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

Algunos ejemplos de expresiones para esta gramática son:

```
· foo
· 1729
· true
· false
· {cons 1 {cons 2 empty}}
· {with {{a 2} {b 3}}
     {+ a b}}
· {rec {{fac {fun {n} {if {zero? n} n {* n {fac {- n 1}}}}}}}
     {fac 5}}
· {fun {b h} {/ {* b h} 2}}
· {if {> 10 2} 1 false}
· {+ 1 2 {* 3 4} {pow 5 6} {max 7 8 9}}
· {not {and true {> 10 2} {or true {!= 1 2}}}}
```

```
· {head {tail {cons 1 {cons 2 empty}}}}
· {{fun {b h} {/ {* b h} 2}} 20 27}
· {with {{x 7} {caja {box 3}}}
     {seqn
        {set-box! caja {+ x {unbox caja}}}
        caja}}
```

Para implementar esta práctica se recomienda altamente consultar el capítulo 13 de "*Programing Languages: Aplications and Interpretation*".

1. **Gramáticas**

   Modificar el archivo `grammars.rkt` de la práctica 6 para que acepte las nuevas expresiones de la gramática y que defina los tipos necesarios para operar con el `store` y los nuevos valores de regreso `ValueXStore`.

2. **Análisis sintáctico / Azúcar sintáctica**

   Modificar el archivo `parser.rkt` de la práctica 6 que contiene la definición de dos funciones:

   - Una función `(parse expr)` que toma una expresión en sintaxis concreta y regresa su representación en sintaxis abstracta `BRCFWBAEL`.
   - Una función `(desugar expr)` que toma una expresión en sintaxis abstracta `RCFWBAEL` y la transforma a su respectiva sintaxis en `BRCFBAEL`.

   Modificar el cuerpo de estas funciones para que procesen los tipos de datos definidos en el archivo anexo `grammars.rkt`. No olvidar usar la notación `quote` al ejecutar la función `parse`.

3. **Intérprete**

   Modificar archivo `interp.rkt` de la práctica 6 que contiene la definición de una función `(interp exp env)` que toma un árbol de sintaxis abstracta y regresa su significado. Tomar en cuenta que:

   - Para implementar estado, será necesario hacer las modificaciones necesarias a los ambientes e implementar los `stores` descritos anteriormente. Aunado a esto, este intérprete tiene la singularidad de que los valores de regreso son de tipo `ValueXStore`, es decir, al evaluar una expresión se debe regresar una estructura que contenga tanto el valor de dicha expresión como el `store` que posiblemente se modificó al reducir dicha expresión.
   - La primitiva `box` recibe lo que se desea guardar y crear una nueva localidad en el `store`.
   - La primitiva `unbox` recibe una caja y regresa el valor que almacena.
   - La primitiva `set-box!` recibe una caja y un nuevo valor a almacenar.
   - La primitiva `seqn` recibe varias expresiones, las ejecuta todas y regresa el resultado de evaluar la última. Cualquier cambio en el estado de las variable hecho en alguna de las expresiones se debe ver reflejado al momento de empezar a ejecutar la última expresión.