Lenguajes de Programación, 2017-2

Práctica Extra: Estado

Karla Ramírez Pulido José Ricardo Rodríguez Abreu
Manuel Soto Romero

Fecha de inicio: 25 de mayo de 2017
Fecha de término: 15 de junio de 2017

1. Objetivos

- Implementar una versión completa del intérprete para el lenguaje diseñado en la Práctica 6 del curso. La implementación deberá soportar condicionales, expresiones aritméticas, expresiones booleanas, expresiones recursivas, identificadores, listas, funciones de primera clase y cajas usando evaluación glotona para interpretar cada una de las expresiones[1].

2. Archivos requeridos

Anexo a este archivo en formato PDF se encuentran los siguientes archivos, necesarios para desarrollar la práctica:

- Un archivo `grammars.rkt` que contiene los TDA necesarios para implementar el intérprete.
- Un archivo `parser.rkt` que contiene las funciones necesarias para convertir sintaxis concreta en la sintaxis abstracta correspondiente.
- Un archivo `interp.rkt` que contiene los procedimientos necesarios para evaluar las expresiones del lenguaje.
- Un archivo `practica-extra.rkt` que se encarga de ejecutar el intérprete final, usando todas las funciones anteriores.

3. Desarrollo de la práctica

Completar los siguientes ejercicios para lograr la correcta ejecución del archivo `practica-extra.rkt` que implementa un intérprete para el siguiente lenguaje descrito en notación EBNF[2]:

```
<expr> ::= <id>
         | <num>
         | <bool>
         | <list>
         | <box>
```

[1] La práctica se entrega siguiendo los lineamientos especificados en la página del curso `http://lenguajesfc.com/lineamientos.html` y por equipos de tres integrantes.

[2] Del inglés Extended Backus–Naur Form

```
          | {<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>*}
          | {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> <list>}

<box> ::= {newbox <expr>}
        | {setbox <expr> <expr>}
        | {openbox <expr>}

<op> ::= + | - | * | / | % | min | max | pow
       | neg | and | or | < | > | <= | >= | = | != | zero?
       | head | tail | empty?
```

Ejercicios

1. (1 pt.) Completar el cuerpo de la función `(parse sexp)` contenida en el archivo `parser.rkt` que recibe una expresión en sintaxis concreta y posteriormente construir el árbol de sintaxis abstracta del lenguaje `BRCFWBAEL`.

2. (1 pt.) Completar el cuerpo de la función `(desugar sexps)` contenida en el archivo `parser.rkt` que recibe una expresión dentro del lenguaje `BRCFWBAEL` y eliminar el azúcar sintáctica, es decir, regresar el árbol de sintaxis abstracta dentro del lenguaje `BRCFBAEL`.

   `BRCFBAEL` es una versión *desendulzada* de `BRCFWBAEL` que no cuenta con constructores para `with`, `with*` ni `cond`. Para eliminar el azúcar sintáctica de este tipo de expresiones, considerar:

   - `with` puede ser expresado como una aplicación de función endulzada.
   - `with*` puede ser expresado como una cadena de expresiones `with` anidadas.
   - `cond` puede ser expresado como una cadena de expresiones `if` anidadas.

3. (8 pts.) Completar el cuerpo de la función `(interp expr env store)` contenida en el archivo `interp.rkt` que recibe un árbol de sintaxis abstracta `BRCFBAEL` y un ambiente de sustitución y a su vez regresa la interpretación del árbol como un valor de tipo `Value*Store`.

   Para esta versión del intérprete, no se almacenan parejas `(id, valor)` en el ambiente, sino pares de la forma `(id, indice)`. El índice indica en qué posición del *store* se encuentra el valor, el *store* almacena parejas `(indice, valor)`.

   El funcionamiento correcto del intérprete, dependerá (1) tanto de cómo se introduce las variables al ambiente y al *store* al aplicar una función, (2) como de la implementación de las funciones `store-lookup` y `env-lookup` que recupere el valor de los identificadores de forma recursiva o no.

   Es importante destacar, que ya no se tienen ambientes recursivos, las cajas con el valor de las definiciones recursivas, se almacenan ahora en el *store*.

   Las nuevas primitivas a implementar son:

   - `newbox` que crea una caja con un valor.
   - `setbox` que modifica el valor de una caja.
   - `openbox` que recupera el valor de una caja.
   - `seqn` que ejecuta n instrucciones, una después de otra y regresa el resultado de la ejecución de la última expresión.

   Por ejemplo, la siguiente expresión:

```
{with {{fac {newbox 1729}}}
   {seqn
      {setbox fac
         {fun {n} {if {zero? n} 1 {* n {{openbox fac} {- n 1}}}}}}
      {{openbox fac} 5}}}
```

   Debe evaluarse a 120.

Referencias

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

[2] Rodrigo Ruiz Murguía, *Manual de prácticas para la asignatura de Lenguajes de Programación*, Reporte de actividad docente, Facultad de Ciencias, 2016.