Universidad Nacional Autónoma de México

Facultad de Ciencias

Práctica 6.
Tabla de símbolos.

Francisco Arturo Licón Colón
314222095

Kevin Miranda Sánchez
314011163

Victor Hugo Molina Bis
314311212

Compiladores.
Profesor: Lourdes del Carmen González Huesca.
10 de noviembre de 2019

1. Desarrollo del programa

1. **Ejercicio 1.** *Definir el proceso uncurry. Este proceso se encarga de descurrificar las expresiones lambda así como las aplicaciones de función.*

   Para este ejercicio definimos un nuevo lenguaje L11.
   En este lenguajes el constructor lambda puede recibir varios parámetros recibe u parámetro.

   La definición de este lenguaje es la siguiente.

```
(define-language L11
  (extends L10)
  (Expr (e body)
      (- (lambda ([x t])body))
       (+ (lambda ([x* t*]...)body))))
```

   Este nuevo lenguaje se define con la siguiente gramática.

   $< programa > ::= < expr >$
   $< expr > ::= < const >$
   $\quad | < var >$
   $\quad |$ (quot $< const >$)
   $\quad |$ (begin $< expr > < expr >$*)
   $\quad |$ (primapp $< prim > < expr >$*)
   $\quad |$ (if $< expr > < expr > < expr >$)
   $\quad |$ (lambda ([$< var > < type >$]*) $< expr >$)
   $\quad |$ (let ([$< var > < type > < expr >$]) $< expr >$)
   $\quad |$ (letrec ([$< var > < type > < expr >$]) $< expr >$)
   $\quad |$ (list $< expr >$*)
   $\quad |$ ($< expr > < expr >$*)

   $< const > ::= < boolean >$
   $\quad | < integer >$
   $\quad | < char >$

   $< boolean > ::=$ # t | # f

   $< integer > ::= < digit > | < digit > < integer >$

   $< digit > ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

   $< var > ::= < car > | < car > < var > | < car > < digit > | < car > < digit > < var >$

   $< char > ::=$ a | b | c | ... | z

   $< prim > ::= + \mid - \mid * \mid / \mid$ length | car | cdr

   $< type > ::=$ Bool | Int | Char | List | Lambda

El proceso uncurry se define de la siguiente manera.

```
(define-pass uncurry : L10 (ir) -> L11 ()
 (Expr : Expr (ir) -> Expr ()
   [(lambda[(lambda ([,x ,t]) ,[body]) `(list ,body)])))
```

2. **Ejercicio 2.***Definir la función symbol-table-var. Esta será la encargada de generar una tabla de símbolos de una expresión del lenguaje.*
Este proceso toma como llave el identificador de la variable y como valor tendrá un par que almacena el tipo de la variable y su valor.

El proceso symbol-table-var se define de la siguiente manera.

```
(define (symbol-table-var expr hash)
 (nanopass-case (L11 Expr) expr
    [(begin ,e* ... ,e) (begin (map (lambda (e) (symbol-table-var e hash)) e*)
        (symbol-table-var e hash))]
    [(primapp ,pr ,[e*]...,e) (begin (map (lambda (e) (symbol-table-var e hash)) e*)
        (symbol-table-var e hash))]
    [(if ,e0 ,e1 ,e2) (begin (symbol-table-var e0 hash)
        (symbol-table-var e1 hash)
        (symbol-table-var e2 hash))]
        [(lambda ([,x ,t]) ,body) (symbol-table-var body hash)]
    [(let ([,x ,t ,e]) ,body) (begin (hash-set*! hash x (cons t (unparse-L11 e)))
        (symbol-table-var body hash))]
    [(letrec ([,x ,t ,e]) ,body) (begin (hash-set*! hash x (cons t (unparse-L11 e)))
        (symbol-table-var body hash))]
     [(letfun ([,x ,t ,e]) ,body) (begin (hash-set*! hash x (cons t (unparse-L11 e)))
        (symbol-table-var body hash))]
            [(list ,e* ... ,e) (begin (map (lambda (e) (symbol-table-var e hash)) e*)
                (symbol-table-var e hash))]
    [(,e0 ,e1) (begin (symbol-table-var e0 hash)
        (symbol-table-var e1 hash))]
    [else hash]))
```

Pra la implementación se utilizó un nanopass-case, como se sugirió en la especificación. Normalmente en los casos en los que se podían tener múltiples funciones (begin, list) la forma en como se revisaban todas se hacía con un **let**, pero al leer la documentación de racker, nos dimos cuenta que esto se podía reducir bastante al utilizar un **map(lambda)**.

Los casos más interesantes son el **let** y sus variantes, en los que no solo se realizaba resursión sobre el body, si no que en éste se define la tabla hash, la cual es construida mapeando cada llave con cada valor en el hash. Para generar el par del hash, se utilizó la función **cons**, con la que se concatenaban el tipo y su valor.

3. **Ejercicio 3***Definir el proceso assigment. Esta función que modifica los constructores let, letrec y letfun eliminando el valor asociado a los identificadores y el tipo correspondiente*

Para este proceso definimos un nuevo lenguaje.
Este lenguaje se extiende 11, agregando que los constructores de asignación solo reciben una variable.

La definición del lenguaje es la siguiente:

```
(define-language L12
(extends L11)
 (Expr (e body)
    (- (let ([x t e]) body)
       (letrec ([x t e]) body)
```

```
      (letfun ([x t e]) body))
   (+ (let body)
      (letrec body)
      (letfun body))))
```

Este lenguaje se define con la siguiente gramática

< *programa* > ::= < *expr* >
< *expr* > ::= < *const* >
| < *var* >
| (quot < *const* >)
| (begin < *expr* > < *expr* >*)
| (primapp < *prim* > < *expr* >*)
| (if < *expr* > < *expr* > < *expr* >)
| (lambda ([< *var* > < *type* >]*) < *expr* >)
| (let ([< *var* >]) < *expr* >)
| (letrec ([< *var* >]) < *expr* >)
| (letfun ([< *var* >]) < *expr* >)
| (letrec ([< *var* >]) < *expr* >)
| (list < *expr* >*)
| (< *expr* > < *expr* >*)

< *const* > ::= < *boolean* >
| < *integer* >
| < *char* >

< *boolean* > ::= # t | # f

< *integer* > ::= < *digit* > | < *digit* > < *integer* >

< *digit* > ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

< *var* > ::= < *car* > | < *car* > < *var* > | < *car* > < *digit* > | < *car* > < *digit* > < *var* >

< *char* > ::= a | b | c | ... | z

< *prim* > ::= + | - | * | / | length | car | cdr

< *type* > ::= Bool | Int | Char | List | Lambda

Y por último, assigment se define:

```
(define-pass assigment : L11 (ir) -> L12 (hash)
 (definitions
    (define hash (make-hash)))
     (Expr : Expr (ir) -> Expr ()
     [(let ([,x ,t ,e]) ,[body]) `(let ,body)]
     [(letrec ([,x ,t ,e]) ,[body]) `(letrec ,body)]
     [(letfun ([,x ,t ,e]) ,[body]) `(letfun ,body)])
         (values (Expr ir) (symbol-table-var ir hash)))
```

Para la implementación de esta función solo se tomaron en cuenta los casos de let y sus variantes, y solo se removía todo menos el nombre y cuerpo de la expresión, lo cual resultó sencillo.

La tabla de procesos se pasa por medio de **values**, el cual debía recibir dos valores, primero la expresión y luego la tabla hash.

2. Comentarios

1. Nos gusto usar hash tables, ya es bueno ver el uso de otras estructuras de datos en uso de programacón a un nivel más bajo

2. Hubo bastantes complicaciones en la realización de la práctica, ya que a pesar de lucir sencilla, se debió probar bastantes opciones para llegar a concluir algunos aspectos.

3. En la implemtación de la tabla hash nos salieron varios errores los cuales solucionamos escribiendo pequeños cambios, por ejemplo, al escribir **hash-set** nos salía un error el cual no entendíamos del todo, pero al cambiarlo por **hash-set*!** todo funciono correctamente.

4. Otro error del cual se aprendió algo importante es la utilización de **begin**, ya que este nos permitió utilizar más de una expresión, lo cual era muy necesario, ya que prácticamente todas las expresiones están compuestas de más de una expresión.

5. El primer proceso fue mucho más complejo de lo que parecía, ya que éste aplicaba recursión automáticamente, por lo que, al introducir dos lambas y devolver el body, en lugar de regresar la segunda lambda, regresaba el último body de la expresión; por lo tanto, se debían utilizar funciones auxiliares y no se debía trabajar completamente sobre la expr definida en el caso de lambda. Por lo que resultaba factible utilizar la expresión encontrada en la firma de la función (**ir**).