Universidad Nacional Autónoma de México

Facultad de Ciencias

Práctica 4.
Análisis Sintáctico.

Francisco Arturo Licón Colón
314222095

Kevin Miranda Sánchez
314011163

Victor Hugo Molina Bis
314311212

Compiladores.
Profesor: Lourdes del Carmen González Huesca.
6 de septiembre de 2019

1. Desarrollo del programa

1. **Ejercicio 1.** *Definir el proceso curry-let encargado de currificar las expresiones let y letrec.*

   Currificar es la técnica desarrollada por Moses Schönfinkel y Gottlob Frege que consiste en transformar una función que utiliza múltiples parámetros en una secuencia de fuciones que utilizan un único parámetro.

   Para este ejercicio definimos un nuevo lenguaje L7.
   En este lenguajes se modifican los constructores let y letrec para que tengan una sola asignación.

   La definicón de este lenguaje es la siguiente.

```
(define-language L7
  (extends L6)
  (Expr (e body)
    (- (let ([x* t* e*] ...) body)
     (letrec ([x* t* e*] ...) body))
    (+ (let ([x t e]) body)
     (letrec([x t e]) body))))
```

   Este nuevo lenguaje se define con la siguiente gramática.

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

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

   $< boolean > ::= \#\ \text{t} \mid \#\ \text{f}$

   $< integer > ::= < digit > \mid < digit > < integer >$

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

   $< var > ::= < car > \mid < car > < var > \mid < car > < digit > \mid < car > < digit > < var >$

   $< char > ::= \text{a} \mid \text{b} \mid \text{c} \mid ... \mid \text{z}$

   $< prim > ::= + \mid - \mid * \mid / \mid \text{length} \mid \text{car} \mid \text{cdr}$

   $< type > ::= \text{Bool} \mid \text{Int} \mid \text{Char} \mid \text{List} \mid \text{Lambda}$

Para realizar la implementación, se requirió de una función auxiliar definida dentro de un let, gracias a esta función, el curry let podía ser realizado recursivamente y detenerse en el momento adecuado.
El proceso curry-let se define de la siguiente manera.

```
(define-pass curry-let : L6 (ir) -> L7 ()
 (Expr : Expr (ir) -> Expr ()
   [(let ([,x* ,t* ,[e*]] ...) ,[body])
   (let f ([x* x*] [t* t*] [e* e*])
    (if (not(null? x*))
      `(let ([,(car x*) ,(car t*) ,(car e*)]) ,(f (cdr x*) (cdr t*) (cdr e*)))
      body
      ))]
    [(letrec ([,x* ,t* ,[e*]] ...) ,[body])
   (let f ([x* x*] [t* t*] [e* e*])
    (if (not(null? x*))
      `(letrec ([,(car x*) ,(car t*) ,(car e*)]) ,(f (cdr x*) (cdr t*) (cdr e*)))
      body
      ))]))
```

2. **Ejercicio 2.** *Definir el proceso identity-assigments en el que se detecten los let utilizados para definir funciones y que se remplazan por letrec.*

   Para este ejercicio no fue necesario definir un nuevo lenguaje pues sólo reemplazaremos algunos casos de let por letrec.

   El proceso identity-assigments se define de la siguiente manera.

```
(define-pass identify-assigments : L7 (ir) -> L7 ()
 (Expr : Expr (ir) -> Expr ()
     [(let ([,x ,t ,e]) ,body)
     (if (equal? t 'Lambda)
     `(letrec ([,x Lambda ,e]) ,body)
     `(let ([,x ,t ,e]) ,body))]))
```

   Notemos que sólo en el caso en que t sea sea una asignación de función 'Lambda reemplazamos let por letrec, en otro caso lo dejamos igual.

3. **Ejercicio 3.** *Definir el proceso un-anonymus encargado de asignarle un identificador a las funciones anónimas (lambda).*

   Para este ejercicio definimos un nuevo lenguaje L8.
   En este lenguajes se extiende L7 agregando el constructor de asignación letfun.

   La definicón de este lenguaje es la siguiente.

```
(define-language L8
 (extends L7)
 (Expre (e body)
     (+ (letfun ([x t e]) body))))
```

Este nuevo lenguaje se define con la siguiente gramática.

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

El proceso un-anonymus se define de la siguiente manera.

```
(define-pass un-anonymous : L7 (ir) -> L8 ()
 (Expr : Expr (ir) -> Expr ()
  [(lambda (,x ,t] ...) ,body)
  (let ([name (unique-vars 'foo)])
  `(letfun ([,name Lambda (lambda (,x ,t] ...) ,body)]) ,name))]))
```

Para realizar el proceso anterior se requirió de una función auxiliar, la cual genera nombres únicos para variables o funciones. La función auxiliar esta definida afuera del proceso porque cada vez que sea lamada generarrá un nombre nuevo, es decir, no hay forma de que haya dos funciones llamadas foo0.

```
(define unique-vars
 (let ()
  (define count 0)
  (lambda (name)
   (let ([c count])
    (set! count (+ count 1))
    (string->symbol
    (string-append (symbol->string name) (number->string c)))))))
```

4. **Ejercicio 4.** *Define el proceso verify-arity. Este proceso funciona como verificador de la sintaxis de las expresiones. El proceso consiste en verificar que el número de parámetros corresponda con la aridad de las primitivas. Si corresponde, se regresa la misma expresión en caso contrario se lanza un error, especificando que la expresión está mal construida.*

   Recordemos la aridad de las primitivas.

Primitiva	Aridad
+	2
-	2
*	2
/	2
length	1
car	1
cdr	1

   Para este ejercicio no fue necesario definir un nuevo lenguaje pues sólo verificaremos que el número de parámetros de las expresiones sea el correcto. En el caso de las expresiones lógicas, el compilador solo permite que estas tengan uno o dos de aridad.
   Para la implementación de este proceso se verificó primitiva por primitiva su aridad, agrupando las que tengan la misma para tener un código más corto.

   El proceso verify-arity se define de la siguiente manera.

```
(define-pass verify-arity : L8 (ir) -> L8 ()
 (Expr : Expr (ir) -> Expr ()
    [(primapp ,pr ,[e*]...)
    (let ([e e*])
     (if (memq pr '(+ - * /))
      (if (= 2 (length e))
       `(primapp ,pr ,e* ...)
       (error 'error "Arity mismatch"))
     (if (memq pr '(length car cdr not))
      (if (= 1 (length e))
       `(primapp ,pr ,e* ...)
       (error 'error "Arity mismatch"))
     (if (memq pr '(and or))
      (if (or (= 1 (length e)) (= 2 (length e)))
       `(primapp ,pr ,e* ...)
       (error 'error "Arity mismatch"))
     (error 'error "Invalid primitive"))))]))
```

5. **Ejercicio 5.** *Define el proceso verify-vars. Este proceso funciona como verificador de la sintaxis de las expresiones. El proceso consiste en verificar que la expresión no tenga variables libres, de existir variables libres se regresa un error en caso contrario la salida es la misma expresión.*

   Para el proceso no se necesitó definir un nuevo lenguaje, se continuó trabajando con el L8.
   Para este ejercicio se tomo caso por caso, en algunos casos se debía hacer recursión sobre las expresiones, la función solo revisa si las funciones son simples, es decir, no tiene if dentro de un let o expresiones así.

2. Comentarios

1. Para el último proceso (verify-vars) tuvimos algunos problemas, ya que no nos fue muy claro el cómo definir la función para obtener las variables libres.

Se intentó hacer la recursión dentro del mismo proceso, pero esto no dió resultados.

2. La definición de los lenguajes fue muy sencilla gracias a que se nos proporcionó el lenguaje L6, lenguje resultnte de prácticas anteriores.

3. En general para los demas procesos no tuvimos complicaciones, pues el uso de funciones auxiliares (unique-vars y unique-vars) hizo que los procesos fueran más sencillos de definir. Además, después de un tiempo, hemos entendido mucho mejor nanopass, aunque aún se nos complican ciertas implementaciones.