Universidad Nacional Autónoma de México

Facultad de Ciencias

Práctica 07.
Generación de Código en C.

Francisco Arturo Licón Colón
314222095

Kevin Miranda Sánchez
314011163

Victor Hugo Molina Bis
314311212

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

1. Desarrollo del programa

1. **Ejercicio 1.** *Definir el proceso list-to-array encargado de traducir las listas de nuestro lenguaje en arregos..*

   Para este ejercicio definimos un nuevo lenguaje L13.
   En este lenguaje se elimina el constructor list y se agrega un nuevo constructor (array len t[e*...]).

   La definicón de este lenguaje es la siguiente.

```
(define-language L3
  (extends L12)
  (terminals
    (+ (len (len))))
  (Expr (e body)
        (- (list e* ...))
        (+ (array len t [e* ...]))))
```

   Este nuevo lenguaje se define con la siguiente gramática.

   $< programa > ::= < expr >$
   $< expr > ::= < const >$
   $\quad | < var >$
   $\quad | ($ 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 $([< var > < type > < expr >]) < expr >)$
   $\quad | ($ array $< len > < type > [< expr >]^*)$
   $\quad | (< expr > < expr >)$

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

   $< boolean > ::= \#$ 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 | $(< type > \rightarrow < type >)$ | (List of $< type >$)

El proceso list-to-array se define de la siguiente manera.

```
(define-pass list-to-array : L12 (ir) -> L13 ()
  (Expr : Expr (ir) -> Expr ()
    [(list ,[e*] ...) `(array (const Int ,(length e*)) ,(J ir '()) ,(parser-L13(map unparse-L13 e*)))]))
```

Por alguna razón que desconocemos, no nos permitía mostrar la logitud como un número, por lo que tuvimos que mostrarla como un **const**.
Para obtener el tipo de la lista ustilizamos el proceso J definido en una práctica realizada anteriormente. A este algoritmo se le realizaron unas pequeñas modificaciones, primero se le cambió el lenguaje por L12, además, se cambió la forma en la que se muestra el tipo de las listas, para solo mostrar el tipo sin List, es decir, en lugar de que la salida sea **List of Int**, la salida es **Int**.
Por último, por medio de un map y haciendo el unparse a **e***, se pudieron mostrar todos los elementos que conforman a e*; para que esto se mostrara correctamente, al resultado obtenido se le aplicaba el parser-L13.

2. **Ejercicio 2.** *Definir el proceso c, que se encargará de traducir las expresiones de nuestro lenguaje en expresiones del lenguaje de programación C. Para esto se utiliza la siguiente tabla de equivalencias:*

Expresión	C
Variables	Variables
`const`	Valor de la constante
`begin`	Bloques de código
`primapp` aritmética	Operación aritmética correspondiente
`primapp` de listas	Operación correspondiente para arreglos
`if`	`if/else` en C
`lambda`	No es necesario traducir
`let`	Asignaciones, depende del valor
`letrec`	Declaración de función
`letfun`	Declaración de función
`array`	Arreglo
Aplicación de función	Llamada a función

El proceso c se define de la siguiente manera.

```
(define (c expr)
  (nanopass-case (L13 Expr) expr
    [,x x]
    [(const ,t ,c) c]
    [(begin ,[e*] ... ,[e]) `(,e*";",e";")]
    [(primapp ,pr ,[e*] ...)
     (if (memq pr '(+ - * /))
       `([,(car e*) ,pr ,(last e*)])
      (if (equal? pr 'length)
        `(sizeof"[",e*if"]")
       (if (equal? pr 'car)
           `(,e*"["0"]")
           (if (equal? pr 'cdr)
             `(tail "[",e*"]")
             (error "Operación incorrecta"))))))]
[(if ,e0 ,e1 ,e2) `(if (,(c e0))"{" ,(c e1) ";" "}" else"{" ,(c e2) ";" "}")]
[(array ,len ,t [,e* ...]) `("{",(map c e*)"}")]
[(let ([,x ,t ,e]) ,body)
  (if (equal? (car t) 'List)
```

```
      `(,(last t) ,x"[" ,(length (last (unparse-L13 e)))"]" = ,(last e)";" ,(c body)";")
      `(,t ,x = ,(c e)";" ,(c body)";"))]
  [(letrec ([,x ,t ,e]) ,body) `("void" ,x "{" ,(last (unparse-L13 e))";" "}",(c body)";")]
  [(letfun ([,x ,t ,e]) ,body) `("void" ,x "{" ,(last (unparse-L13 e))";" "}",(c body)";")]
  [(,e0 ,e1 ...) `(,(c e0)";" ,(c e1)";")]
  [else "Expr incorrecta"]))
```

La forma en como está diseñada la salida es un poco peculiar, ya que para mostrar caracteres como ";","," "[", debíamos mostrarlos como strings, lo que implicaba que las comillas también se mostraran en la salida. Se decidió mostrarlo así porque de otra forma no había posibilidad de que la sintaxis del lenguaje C pudiera aceptar la salida generada.
La función **cdr** no tiene un equvaente en el lenguaje C, lo que implicaría escribir una cantidad más grandes de líneas de código para poder generarla, esto es poco efectivo si **cdr** es utlizado constantemente, entonces lo conveniente sería escribir un método que replique su funcionamiento. Esto no se genera en esta práctica, pero se tiene planeado realizarlo para el proyecto, por el momento solo se generó el nombre del método,
En el caso de **letrec** y **letfun**, las cuales realizan asignaciones a funciones, era importante conocer cuál es el tipo de la función; nuestro proceso **J** no funciona con la sintaxis del nuevo lenguaje, ya que las funciones **lambda** ya pueden una o varias asignaciones, por lo que obtener su tipo resultaba más complejo. Por el momento se le asignará a todas las funciones como valor de retorno **void**, para el proyecto, se piensa complementar el proceso para que se pueda obtener el tipo de la función que sea introducida.

2. Comentarios

1. El proceso C resultó ser más complejo de lo que se esperaba, manejar las salidas implicaba, en algunos casos, realizar el unparse al lenguaje que se estaba utulizando.

2. Mosrar **e*** por completo fue un poco confuso, obtuvimos varios errores al intentarlo, pero gracias a la función map, se obtuvo el resultado deseado.

3. Obtener los tipos de una lambda, dada que esta ya cambió de definiciçon, resulta ser un pococ confuso más que complejo, ya que está ya puede obtener más de un tipo y variable, por lo tanto, puede contener distintos tipos, caso que no estaba contemplado en nuestro proceso **J**, por lo que la definicón del proceso deberá cambiar en el prpyecto.