Compiladores 2020-1
Facultad de Ciencias UNAM
Práctica 3: Análisis Léxico

Licon Colon Francisco Aruturo 314222095

Miranda Sanchez Kevin Ricardo 314011163

Molina Bis Victor Hugo 314311212

22 de Septiembre de 2019

1. Desarrollo del Programa

1. *Definir el proceso remove-logical-operators. Este proceso se encarga de eliminar las primitivas and, or y not sustituyendolas con expresiones if.*

   Para facilitar la realización de este ejercicio, al lenguaje L0 se le agregaron 3 constructores (and,or y not), esto no afecta en lo absoluto, dado que en el lenguaje L4 dichos constructores serán eliminados.

   En la definición del nuevo lenguaje, se eliminaron los constructores agregados anteriormente y se eliminaron las primitivas and, or y not. Para el proceso, se requiría de dos funciones auxiliares, **and-to-if** y **or-to-if**, las cuales reciben dos clausulas (e1 y e2) y, en caso de que e2 no sea null, se devuelve la equivalencia. Se debe notar que en el segundo o tercer parámetro (dependiendo el caso) del if se realiza recursión, esto se debe a que la función debe ser aplicada múltiples veces en caso de que se tengan más de dos cláusulas.

   La definición del lenguaje es la siguiente.

```
(define-language L4
   (extends L0)
   (terminals
    (- (primitive (pr)))
    (+ (primitive-no-logic(pr))))
      (Expr (e body)
      (- (or e* ...)
         (and e* ...)
         (not e))))
```

   El nuevo lenguaje se define con la siguiente gramática.

```
<programa> ::= <expr>
<expr> ::= <const>
     | <list>
     | <var>
     | (begin <expr> <expr>*)
     | (if <expr> <expr> <expr>)
     | (lambda ([<var> <type>]*) <expr>)
```

```
        | (let ([<var> <type> <expr>]*)  <expr>)
        | (letrec ([<var> <type> <expr>]*) <expr>)
        | (list <expr> <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
<list> ::= empty | (cons <const> <list>)
<char> ::=  a | b | c | ... | z | ... | @ | # | $ | % | & | ...
<prim> ::= + | - | * | / | length | car | cdr
<type> ::= Bool | Int | Char | List | String
```

Se definieron dos funciones auxiliares para convertir and y or a una expresión if.

```
(define (or-to-if c1 c2)
     (let f ([c1 c1] [c2 c2])
      (if (null? c2)
          c1
          `(if,c1,c1,(f (car c2) (cdr c2))))))
```

```
(define (and-to-if c1 c2)
        (let f ([c1 c1] [c2 c2])
      (if (null? c2)
          c1
          `(if , c1 , (f (car c2)(cdr c2)) #f))))
```

Las equivalencias quedaron de la siguiente forma:
not c1 = if c1 f t
or c1 c2* = if c1 c1 c2*
and c1 c2* = if c1 c2* f
Se aplica recursión sobre c2, para poder generar más if's, en el caso de que sea necesario.

Por último, se definió el proceso.

```
(define-pass remove-logical-operators : L0 (ir) -> L4 ()
    (Expr : Expr (ir) -> Expr ()
        [(not ,[e0]) `(if ,e0 #f #t)]
        [(and) #t]
        [(or) #f]
        [(and,[e],[e*] ...)
            (and-to-if e e*)]
        [(or,[e],[e*] ...)
            (or-to-if e e*)]))
```

2. *Definir el proceso eta-expand.*

Se definió el constructor primapp y se eliminó pr.
Se toma cada caso de las primitivas, pero se agrupan las que reciben Int's y las que reciben List's; no se hizo inferencia de tipos, como se especificó en el documento del ejercicio, además, los nombres de las variables fueron designados

por el diseñador.

La definición del lenguaje es la siguiente.

```
(define-language L5
   (extends L4)
   (Expr (e body)
       (- pr)
       (+ (primapp pr e* ...))))
```

El nuevo lenguaje se define con la siguiente gramática.

```
<programa> ::= <expr>
<expr> ::= <const>
     | <list>
     | <var>
     | <string>
     | (primapp <prim> <const>*)
     | (begin <expr> <expr>*)
     | (if <expr> <expr> <expr>)
     | (lambda ([<var> <type>]*) <expr>)
     | (let ([<var> <type> <expr>]*)  <expr>)
     | (letrec ([<var> <type> <expr>]*) <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
<list> ::= empty | (cons <const> <list>)
<char> ::=  a | b | c | ... | z | ... | @ | # | $ | % | & | ...
<prim> ::= + | - | * | / | length | car | cdr
<type> ::= Bool | Int | Char | List | String
```

Por último definimos el proceso cuando se hace la aplicación de las primitivas.

```
(define-pass eta-expand : L4 (ir) -> L5 ()
  (Expr : Expr (ir) -> Expr ()
    [(,pr,[e*] ...)
     (if (memq pr '(+ - * /))
         `((lambda ([x Int] [y Int]) (primapp ,pr x y)) ,e*)
          (if (memq pr '(length car cdr))
              `((lambda ([x List]) (primapp ,pr x)) ,e*)
               (error 'error "Invalid primitive")))]))
```

3. *Definir el proceso quote-const encargado de quotear constantes.*

Para este ejercicio fue necesario definir un nuevo lenguaje, donde se eliminó el constructor de las constantes y se añadió cuote. Cabe recalcar que se escribió 'cuote' en lugar de 'quote' para que no tomara el quote como ' , si no como un string. Solo fue necesario tomar el caso de las constantes, ya que ese era el único que debía ser modificado.

La definición del lenguaje es la siguiente.

```
(define-language L6
    (extends L5)
    (Expr (e body)
      (- c)
      (+ (cuote c))))
```

El nuevo lenguaje se define con la siguiente gramática.

```
<programa> ::= <expr>
<expr> ::=
     | <list>
     | <var>
     | (cuote <const>)
     | (primapp <prim> <const>*)
     | (begin <expr> <expr>*)
     | (if <expr> <expr> <expr>)
     | (lambda ([<var> <type>]*) <expr>)
     | (let ([<var> <type> <expr>]*)  <expr>)
     | (letrec ([<var> <type> <expr>]*) <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
<list> ::= empty | (cons <const> <list>)
<char> ::=  a | b | c | ... | z | ... | @ | # | $ | % | & | ...
<prim> ::= + | - | * | / | length | car | cdr
<type> ::= Bool | Int | Char | List | String
```

Por último definimos el proceso:.

```
(define-pass quote-const : L5 (ir) -> L6 ()
 (Expr : Expr (ir) -> Expr ()
   [,c `(cuote ,c)]))
```

4. *Define el proceso purify-recursion. Este proceso esta encargado de verificar que las expresiones letrec asignen únicamente expresiones lambda a variables.*

   Para este proceso solo se tomó el caso de letrec, para la implementación se utilizó una función recursiva en el proceso que se llamaba con cada asignación dentro del letrec; dicha función verificaba si el tipo de la variable era Lambda, en caso de que si lo fuera continuaba con la siguiente, en caso contrario agregaba un let y continuaba con la siguiente asignación.

```
(define-pass purify-recursion : L6 (ir) -> L6 ()
 (Expr : Expr (ir) -> Expr ()
       [(letrec ([,x* ,t* ,e*] ...) ,body)
        (let f ([x* x*][t* t*] [e* e*])
          (if (null? t*)
              body
```

```
                (if (equal? 'Lambda (car t*))
                    `(letrec ([,(car x*) ,(car t*) ,(car e*)]) ,(f (cdr x*) (cdr
                        t*) (cdr e*)))
                    `((let ([,(car x*) ,(car t*)
                    ,(car e*)])),(f (cdr x*) (cdr t*)
                    (cdr e*))))))]))
```

5. *Define el proceso direct-app. Este proceso se encarga de traducir una aplicación de función a una expresión let tomando como nombres de variables los parámetros formales y como valores los parámetros reales.*

   Se siguió la estructura del ejemplo, en el que se cambia la posición de la primera expresión con el resto, escribiendo la primera al final del let.

   Al let se le agrega e* después de t* para que cada expresión esté al lado de su tipo. Por lo tanto, no importa cuántas expresiones se tengan, todas estarán junto a su variable y tipo correspondiente.
   Para este ejercicio no fue necesario crear un nuevo lenguaje, se utilizó el lenguaje L6

   Definición del proceso.

```
   (define-pass direct-app : L6 (ir) -> L6 ()
    (Expr : Expr (ir) -> Expr ()
      [((lambda ([,x* ,t*] ...) ,e0) ,e* ...)
       `(let ([,x* ,t*,e*] ...) ,e0)]))
```

2. Comentarios

1. En esta práctica notamos algo que nos resulto muy interesante, el tomar a las primitivas como casos particulares de funciones.

2. La realización del ejercicio 2 se nos dificultó porque no sabíamos cómo tomar caso por caso cada primitiva, es decir, escribir lo que pasa cuando aparece un +,-,etc. Teníamos en mente que esto se solucionaba con un simple **case**, en el que se escribía algo así:

```

       (define-pass eta-expand : L4 (ir) -> L5 ()
        (Expr : Expr (ir) -> Expr ()
         [(,pr,[e*] ...) (primapp ,pr,e* ...)]
         [(,pr ((case (pr)
                ['(+) ...]
                ['(-) ...]
                ...))])))
```

   Esto no fue una solución, lo que nos impidió llevar a cabo nuestra idea.
   Nota: Esto fue arreglado en la última versión.

3. Para el ejercicio 4 no se nos ocurrió una implementación adaptable facilmente a nanopass.
   Nota: Gracias al tiempo que se le ha invertido a nanopass, la implementación fue lograda en la última versión.

4. El principal problema que estamos teniendo con las prácticas es por nanopass, se nos ha complicado bastante, a pesar de que la dificultad de los ejericios no es muy alta, la falta de comprensión del lenguaje nos impide llevar a cabo todos lo ejercicios.