Compiladores 2020-1
Facultad de Ciencias UNAM
Práctica 3

Luis Eduardo Martinez Hernandez 314240028 Emmanuel Cruz Hernandez 314272588
Daniela Susana Vega Monroy 314582915

6 de octubre de 2019

1. Solución a los ejercicios

1.-El ejercicio 1 se resolvió de la siguiente manera:priemro se añadieron las expresiones del tipo (not e),esto lo hicimos para que pudiera hacer el pattern matching de manera correcta y ademas quitamos las primitivas del lenguaje(not,and y or),después de esto simplemente pusismos en nuestro paso que si encontraba una expresión de este tipo (not e),entonces lo sustituyera por su expresión equivalente usando ifs, en el caso de (not e) quedó como ‘(if ,e #f #t).
Las equivalencias que usamos fueron:
(not e) se sustituye por ‘(if ,e #f #t).
(and ,[e0] ,[e1]) se sustituye por ‘(if (,e0) ,e1 #f).
(or ,[e0] ,[e1]) se sustituye por ‘(if (,e0) #t ,e1).
2.-El ejercicio 2 se resolvió de la siguiente manera: priemro se añadieron las expresiones del tipo (+ e0 e1) (para cada operador +-*/),ademas de las expresiones donde solo estab el operador (+-*/),esto lo hicimos para que pudiera hacer el pattern matching de manera correcta,despues de esto simplemente pusismos en nuestro paso que si encontraba una expresión de este tipo (+ e0 e1),entonces lo sustituyera por su expresion equivalente usando los primapps, en el caso de (+ e0 e1) quedo como ‘((lambda ([x Int] [y Int]) (primapp + x y)) ,e0 ,e1),además quitamos las primitivas +-*/ del lenguaje.
Como nota adicional a este ejercicio decidimos de añadir las expresiones (+/*-) debido a que si nos daban una expresión donde solo estaban estos operadores pudieramos hacer el match de ese operador y poderlo sustituir por su respectiva primapp(pero con la diferencia de que en estos casos no se regreso una aplicación de función).
3.-Este ejercicio se resolvio de manera muy sencilla, lo único que se hizo fue añadir esta expresión a nuestro lenguaje (kuote c), y en el paso cada vez que nos encontraramos con una constante cambiarlo por esta expresión (kuote c).
4.-Se creó una función auxiliar obtenNoLambda y nos apoyamos con la función previamente creada organizaLetrec.
5.-Este ejercicio se resolvió de la siguiente manera: dentro del paso del direct-app cada vez que veíamos una lambda, la sustituiamos por ‘(let ([,x* ,t* ,e1] ...) (begin ,body* ... ,body)),es decir el cambio que haciamos era cambiar lambda por let, la expresión que servía como parámetro para la aplicación de función de la lambda lo movimos dentro de los parámetros del let, y el cuerpo de la lambda era el mismo que el cuerpo del let.

2. Comentarios

Si bien los primeros tres ejercicios se completaron de manera rápida el ejercicio 5 que consitía en el reemplazo de quote, tuvimos que cambiar el resultado con ayuda del ayudante y en lugar de que el output fuera quote lo cambiamos por kuote.

3. Gramática

```
<programa> ::= <expr>
```

```
<expr> ::= <const>
         | <list>
         | <var>
         | <string>
         | (<prim> <const> <const>*)
         | (begin <expr> <expr>*)
         | (if <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>

<car> ::= a | b | c | ... | z

<list> ::= empty | (cons <const> <list>)

<string> ::= "" | " <char> <string> "

<char> ::=  a | b | c | ... | z | ... | @ | # | $ | % | & | ...

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

<type> ::= Bool | Int | Char | List | String
```

L1

```
<programa> ::= <expr>

<expr> ::= <const>
         | <list>
         | <var>
         | <string>
         | <void>
         | (<prim> <const> <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>

<car> ::= a | b | c | ... | z

<list> ::= empty | (cons <const> <list>)

<string> ::= "" | " <char> <string> "

<char> ::=  a | b | c | ... | z | ... | @ | # | $ | % | & | ...

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

<type> ::= Bool | Int | Char | List | String
```

L2

```
<programa> ::= <expr>

<expr> ::= <const>
          | <list>
          | <var>
          | (<prim> <const> <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>

<car> ::= a | b | c | ... | z

<list> ::= empty | (cons <const> <list>)
```

```
<char> ::=  a | b | c | ... | z | ... | @ | # | $ | % | & | ...

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

<type> ::= Bool | Int | Char | List
```

L4

```
<programa> ::= <expr>

<expr> ::= <const>
         | <list>
         | <var>
         | (<prim> <const> <const>*)
         | (begin <expr> <expr>*)
         | (and <expr> <expr>)
         | (or <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>

<car> ::= a | b | c | ... | z

<list> ::= empty | (cons <const> <list>)


<char> ::=  a | b | c | ... | z | ... | @ | # | $ | % | & | ...

<prim> ::= + | - | * | / | npr | length | and | or  not | car | cdr

<type> ::= Bool | Int | Char | List
```

L5

```
<programa> ::= <expr>

<expr> ::= <const>
         | <list>
         | <var>
         | (<prim> <const> <const>*)
         | (begin <expr> <expr>*)
         | (+ <expr> <expr>)
```

```
          | (- <expr> <expr>)
          | ( * <expr> <expr>)
          | (/ <expr> <expr>)
          | (primapp (<expr>) <expr>)
          | (primapp (<expr>) <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>

<car> ::= a | b | c | ... | z

<list> ::= empty | (cons <const> <list>)


<char> ::=  a | b | c | ... | z | ... | @ | # | $ | % | & | ...

<prim> ::=  length | car | cdr

<type> ::= Bool | Int | Char | List
```

L6

```
<programa> ::= <expr>

<expr> ::= <const>
          | <list>
          | <var>
          | (<prim> <const> <const>*)
          | (begin <expr> <expr>*)
          | (+ <const> <const>)
          | (- <const> <const>)
          | ( * <const> <const>)
          | (/ <const> <const>)
          | (primapp (<expr>) <expr>)
          | (primapp (<expr>) <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>

<car> ::= a | b | c | ... | z

<list> ::= empty | (cons <const> <list>)


<char> ::=  (kuote a) | (kuote b) | (kuote b) | ... | (kuote z) | ... | (koute @) |
    (kuote #) | (kuote $) | (kuote %) | (kuote &) | ...

<prim> ::=  length | car | cdr

<type> ::= Bool | Int | Char | List
```

```
(define-language LF
  (terminals
   (variable (x))
   (primitive (pr))
   (constant (c))
   (list (l))
   (string (s))
   (type (t)))
  (Expr (e body)
    x
    pr
    c
    l
    t
    (pr c* ... c)
    (begin e* ... e)
    (if e0 e1)
    (if e0 e1 e2)
    (lambda ([x* t*] ...) body* ... body)
    (let ([x* t* e*] ...) body* ... body)
    (letrec ([x* t* e*] ...) body* ... body)
    (e0 e1 ...)))
```

```
(define-language L1
  (terminals
   (variable (x))
   (primitive (pr))
   (constant (c))
   (list (l))
   (string (s))
```

```
    (type (t)))
  (Expr (e body)
    x
    pr
    c
    l
    t
    (pr c* ... c)
    (begin e* ... e)
    (void (void))
    (if e0 e1 e2)
    (lambda ([x* t*] ...) body* ... body)
    (let ([x* t* e*] ...) body* ... body)
    (letrec ([x* t* e*] ...) body* ... body)
    (e0 e1 ...)))
```

```
(define-language L2
  (terminals
    (variable (x))
    (primitive (pr))
    (constant (c))
    (list (l))
    (type (t)))
  (Expr (e body)
    x
    pr
    c
    l
    t
    (pr c* ... c)
    (begin e* ... e)
    (void (void))
    (if e0 e1 e2)
    (lambda ([x* t*] ...) body* ... body)
    (let ([x* t* e*] ...) body* ... body)
    (letrec ([x* t* e*] ...) body* ... body)
    (e0 e1 ...)))
```

```
(define-language L4
  (terminals
    (variable (x))
    (primitive (npr))
    (constant (c))
    (list (l))
    (string (s))
    (type (t)))
  (Expr (e body)
    x
    Npr
    c
    l
    t
    (pr c* ... c)
    (begin e* ... e)
```

```
    (not e)
    (and e0 e1)
    (Nor e0 e1)
    (void (void))
    (if e0 e1 e2)
    (lambda ([x* t*] ...) body* ... body)
    (let ([x* t* e*] ...) body* ... body)
    (letrec ([x* t* e*] ...) body* ... body)
    (e0 e1 ...)))
```

```
(define-language L5
  (terminals
   (variable (x))
   (constant (c))
   (list (l))
   (string (s))
   (type (t)))
  (Expr (e body)
    x
    npr
    c
    l
    t
    (npr c* ... c)
    (begin e* ... e)
    (not e)
    (and e0 e1)
    (or e0 e1)
    (+ e0 e1)
    (- e0 e1)
    (\* e0 e1)
    (/ e0 e1)
    (void (void))
    (primapp (e0) e1)
    (primapp (e0) e1 e2))
    (if e0 e1 e2)
    (lambda ([x* t*] ...) body* ... body)
    (let ([x* t* e*] ...) body* ... body)
    (letrec ([x* t* e*] ...) body* ... body)
    (e0 e1 ...)))
```

```
(define-language L5
  (terminals
   (variable (x))
   (constant (c))
   (list (l))
   (string (s))
   (type (t)))
  (Expr (e body)
    x
    npr
    c
    l
    t
```

```
    (npr c* ... c)
    (begin e* ... e)
    (not e)
    (and e0 e1)
    (or e0 e1)
    (+ e0 e1)
    (- e0 e1)
    (\* e0 e1)
    (/ e0 e1)
    (void (void))
    (primapp (e0) e1)
    (primapp (e0) e1 e2))
    (if e0 e1 e2)
    (lambda ([x* t*] ...) body* ... body)
    (let ([x* t* e*] ...) body* ... body)
    (letrec ([x* t* e*] ...) body* ... body)
    (e0 e1 ...)))
```

```
(define-language L6
  (terminals
   (variable (x))
   (constant (c))
   (list (l))
   (string (s))
   (type (t)))
  (Expr (e body)
    x
    npr
    c
    l
    t
    (npr c* ... c)
    (begin e* ... e)
    (not e)
    (and e0 e1)
    (or e0 e1)
    (+ e0 e1)
    (- e0 e1)
    (\* e0 e1)
    (/ e0 e1)
    (void (void))
    (primapp (e0) e1)
    (primapp (e0) e1 e2))
    (if e0 e1 e2)
    (lambda ([x* t*] ...) body* ... body)
    (let ([x* t* e*] ...) body* ... body)
    (letrec ([x* t* e*] ...) body* ... body)
    (e0 e1 ...)))
```