Compiladores 2020-1
Facultad de Ciencias UNAM
Práctica 6

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

10 de noviembre de 2019

1. Solución a los ejercicios

Para realizar los ejercicios nos auxiliamos con las siguientes funciones:

- (uncurryLambda params cuerpo)
- (tablaSimbolos (make-hash))
- (tablita (make-hash))

(uncurryLambda params cuerpo) se utilizó para el primer ejercicio **uncurry**, (tablaSimbolos (make-hash)) para el segundo ejercicio **symbol-table-var** y (tablita (make-hash)) para el tercer ejercicio **assigment**.

2. Comentarios

A diferencia de otras prácticas, en esta solo se necesitó una función auxiliar por ejercicio, no fue la práctica más fácil sin embargo entendimos bien los ejercios el martes que se explicó la práctica y se terminó a bien tiempo.

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
```

L7

```
<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
```

L8

```
<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>)
         | (leftfun ([<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
```

L9

```
<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>)
         | (leftfun ([<var> <type> <expr>]*)  <expr>)
         | (<expr> <expr>*)

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

<boolean> ::= (const bool #t) | (const bool #f)

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

<digit> ::= (const int 0) | (const int 1) | (const int 2) | (const int 3) | (const
   int 4) | (const int 5) | (const int 6) | (const int 7) | (const int 8) | (const
   int 9)| ... |

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

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

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


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

<prim> ::=  length | car | cdr

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

L10

```
<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>)
         | (leftfun ([<var> <type> <expr>]*)  <expr>)
         | (<expr> <expr>*)

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

<boolean> ::= (const bool #t) | (const bool #f)

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

<digit> ::= (const int 0) | (const int 1) | (const int 2) | (const int 3) | (const
   int 4) | (const int 5) | (const int 6) | (const int 7) | (const int 8) | (const
   int 9)| ... |

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

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

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


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

<prim> ::=  length | car | cdr

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

L11

```
<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>)
         | (leftfun ([<var> <type> <expr>]*)  <expr>)
         | (<expr> <expr>*)

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

<boolean> ::= (const bool #t) | (const bool #f)

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

<digit> ::= (const int 0) | (const int 1) | (const int 2) | (const int 3) | (const
   int 4) | (const int 5) | (const int 6) | (const int 7) | (const int 8) | (const
   int 9)| ... |

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

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

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


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

<prim> ::=  length | car | cdr

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

L12

```
<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 <expr>)
         | (letrec <expr>)
         | (leftfun <expr>)
         | (<expr> <expr>*)

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

<boolean> ::= (const bool #t) | (const bool #f)

<integer> ::= <digit> | <digit><integer>
```

```
<digit> ::= (const int 0) | (const int 1) | (const int 2) | (const int 3) | (const
   int 4) | (const int 5) | (const int 6) | (const int 7) | (const int 8) | (const
   int 9)| ... |

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

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

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


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

<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 ...)))
```

```
(define-language L7
  (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)
    (\* e0 e1)
    (/ e0 e1)
    (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)
    (letrec ([x t e]) body)
    (let ([x t e]) body)
    (e0 e1 ...)))
```

```
(define-language L8
  (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)
    (\* e0 e1)
    (/ e0 e1)
    (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)
    (letrec ([x t e]) body)
    (let ([x t e]) body)
    (letfun ([x t e]) body)
    (e0 e1 ...)))
```

```
(define-language L9
  (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)
    (\* e0 e1)
    (/ e0 e1)
    (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)
    (letrec ([x t e]) body)
    (let ([x t e]) body)
    (letfun ([x t e]) body)
    (e0 e1 ...)))
```

```
(define-language L10
  (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)
    (\* e0 e1)
    (/ e0 e1)
    (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)
    (letrec ([x t e]) body)
    (let ([x t e]) body)
    (letfun ([x t e]) body)
    (e0 e1 ...)))
```

```
(define-language L11
  (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)
    (\* e0 e1)
    (/ e0 e1)
    (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)
    (letrec ([x t e]) body)
    (let ([x t e]) body)
    (letfun ([x t e]) body)
    (e0 e1 ...)))
```

```
(define-language L12
  (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)
    (\* e0 e1)
    (/ e0 e1)
    (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)
    (letrec body)
    (let body)
    (letfun body)
    (e0 e1 ...)))
```