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

Arturo García Campos Alejandro Valderrama Silva

22 de septiembre de 2019

## Desarrollo

Antes de iniciar el desarrollo de esta práctica, recuperamos las funciones y predicados declarados en la práctica 2. Esto para continuar con la implementación de un parser que poco a poco va transformando tipos en expresiones.

## Gramática Original

La gramática que describe el lenguaje utilizado para el desarrollo de esta práctica es la siguiente:

```
<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
```

## Ejercicio 1

Para implementar la función **remove-logical-operators** utilizamos las equivalencias siguientes:

**AND:**

- Si no recibe parámetros, por definición regresará **true**.
- Si sólo tiene un parámetro, también devuelve **true**.
- Cuando recibe dos parámetros, si el primero de ellos es **true**, entonces se evaluará al valor del segundo parámetro. De lo contrario, se evalua a **false**.

**OR:**

- Si no recibe parámetros, por definición regresará **true**.
- Si sólo tiene un parámetro, también devuelve **true**.
- Cuando recibe dos parámetros, si el primero de ellos es **true**, entonces se evaluará a **true**.

**NOT:**

- Si se cumple la expresión, devuelve el valor opuesto.

### Ejercicio 2

Para este ejercicio, definimos las siguientes eta-expansiones:

```
       [(,pr) `(lambda ,(hash-ref types `,pr) (primapp ,pr ,(if (
          hash-has-key? binarios `,pr) `,'(x0 x1) `,'(x0))))]
        [((,pr) ,[e0]) `((lambda ,(hash-ref unarios `,pr) (primapp ,pr
          x0)) ,e0)]
        [((,pr) ,[e0] ,[e1]) `((lambda ,(hash-ref binarios `,pr) (
          primapp ,pr x0 x1)) ,e0 ,e1)]))
```

### Ejercicio 3

Para convertir constante a expresiones quoteadas utilizamos el siguiente proceso:

```
       [,q `(cuote ,q)]))
```

### Ejercicio 5

Nuestra propuesta de solución es la siguiente

```
       [((lambda ([,x* ,t*] ...) ,[body]) ,[e*] ...)
         `(let ([,x* ,t* ,e*] ...) ,body)]))
```

## Lenguajes Usados

### NO-LOGIC-OPS

```
(define-language NO-LOGIC-OPS
  (extends L2)
  (terminals
   (- (primitive (pr)))
   (+ (primitiva (pr))))
  (Expr (e body)
        (- pr)
        (+ (pr)
           (and e* ...)
           (or e* ...)
           (not e))))
```

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

<expr> ::= <const>
         | <list>
         | <var>
```

```
          | <string>
          | (and <expr>*)
          | (or <expr>*)
          | (not <expr>)
          | (<primitiva> <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 | ... | @ | # | $ | % | & | ...

<primitiva> ::= + | - | * | / | length | car | cdr

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

## L4

```
(define-language L4
  (extends NO-LOGIC-OPS)
  (Expr (e body)
        (- (and e* ...)
           (or e* ...)
```

```
            (not e))))
```

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

<expr> ::= <const>
         | <list>
         | <var>
         | <string>
         | (<primitiva> <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 | ... | @ | # | $ | % | & | ...

<primitiva> ::= + | - | * | / | length | car | cdr

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

## L5

```
(define-language L5
```

```
  (extends L4)
  (terminals
    (+ (quotable (q))))
  (Expr (e body)
        (- (pr))
        (+ q
           (primapp pr e* ... e0))))
```

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

<expr> ::= <const>
         | <list>
         | <var>
         | <string>
         | <quotable>
         | (<primitiva> <const> <const>*)
         | (primapp <primitiva> <expr>* <expr>)
         | (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 | ... | @ | # | $ | % | & | ...
```

```
<primitiva> ::= + | - | * | / | length | car | cdr

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

<quotable> ::= q
```

## L6

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

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

<expr> ::= <list>
         | <string>
         | (cuote <quotable>)
         | (<primitiva> <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> ::= + | - | * | / | length | car | cdr

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

<quotable> ::= q
```

## Predicados

```
(define (variable? x) (and (symbol? x) (not (primitive? x)) (not (
   constant? x))))

(define (primitive? x) (memq x '(+ - * / length car cdr and or not)))

(define (constant? x)
  (or (integer? x)
      (char? x)
      (boolean? x)))

(define (type? x) (memq x '(Bool Char Int List String Lambda)))

(define (primitiva? pr) (memq pr '(+ - * / length car cdr)))

(define (quotable? q) (or (constant? q) (and (symbol? q) (not (
   primitive? q)))))
```

## Comentarios

Sólo nos faltó definir la función 4, la cual continuamos trabajando.

Nos encontramos con ciertas dificultades que tenían que ver con modificaciones de las primitivas. Observamos que ocurrían conflictos cuando intentábamos llamar los operadores aritméticos luego de realizar cambios en el predicado original (defniendo el nuestro).

El ejercicio 2 fue el que más tiempos nos tomó en desarrollar por las dificultades que aún enfrentamos respecto a la sintaxis correcta de nano-pass.