# Taller 1
# Semántica de Lenguajes de Programación

**César David Peñaranda Melo - 2266265**
**Kevin Lorza - 2266098**
**Juan David Peres - 2266289**

**Universidad del Valle - Seccional Tuluá**
**Facultad de Ingeniería**
**Fundamentos de Interpretación y Compilación de Lenguajes de Programación**
**Carlos Andres Delgado S, Msc**

**23 de sept. de 24**

**Representación Listas**

**Ejemplo 1: Un circuito simple con dos entradas (a b) y una salida (s) que utiliza un chip AND.**

```
;; Ejemplo 1: Circuito simple con AND
(define circuito-simple-ejemplo1
  (circuito-simple '(a b) '(s) (prim-chip (chip-and))))

;; Prueba para el Ejemplo 1 (Circuito simple con AND)
(circuito-simple-inputs circuito-simple-ejemplo1)
(circuito-simple-outputs circuito-simple-ejemplo1)
(circuito-simple-chip circuito-simple-ejemplo1)
(newline)
```

**RESULTADO**

```
[Running] racket "c:\Users\LENOVO\Documents\5 SEMESTRE\FLP\listas.rkt"
'(a b)
'(s)
'(chip-and)
```

**Ejemplo 2: Un circuito simple con entradas (x y) y salida (z) que utiliza un chip OR.**

```
;; Ejemplo 2: Circuito simple con OR
(define circuito-simple-ejemplo2
  (circuito-simple '(x y) '(z) (prim-chip (chip-or))))

;; Prueba para el Ejemplo 2 (Circuito simple con OR)
(circuito-simple-inputs circuito-simple-ejemplo2)
(circuito-simple-outputs circuito-simple-ejemplo2)
(circuito-simple-chip circuito-simple-ejemplo2)
(newline)
```

**RESULTADO**

```
'(x y)
'(z)
'(chip-or)
```

**Ejemplo 3: Un circuito simple con entradas (i1 i2) y salida (o1) que utiliza un chip NOT.**

```
;; Ejemplo 3: Circuito simple con NOT
(define circuito-simple-ejemplo3
  (circuito-simple '(i1 i2) '(o1) (prim-chip (chip-not))))
;; Prueba para el Ejemplo 3 (Circuito simple con NOT)
(circuito-simple-inputs circuito-simple-ejemplo3)
(circuito-simple-outputs circuito-simple-ejemplo3)
(circuito-simple-chip circuito-simple-ejemplo3)
(newline)
```

**RESULTADO**

```
'(i1 i2)
'(o1)
'(chip-not)
```

**Ejemplo 4: Un circuito complejo que combina dos sub-circuitos: uno con AND y otro con OR.**

```
;; Ejemplo 4: Circuito complejo que combina dos circuitos simples (AND y OR)
(define circuito-complejo-ejemplo1
  (circuito-complejo '(a b c) '(d)
    (list (circuito-simple '(a b) '(x) (prim-chip (chip-and)))
          (circuito-simple '(x c) '(d) (prim-chip (chip-or))))))
;; Prueba para el Ejemplo 4 (Circuito complejo con AND y OR)
(circuito-complejo-inputs circuito-complejo-ejemplo1)
(circuito-complejo-sub-circuitos circuito-complejo-ejemplo1)
(newline)
```

**RESULTADO**

```
'(a b c)
'((circuito-simple (a b) (x) (chip-and)) (circuito-simple (x c) (d) (chip-or)))
```

**Ejemplo 5: Un circuito complejo que combina un sub-circuito con un chip XOR y otro sub-circuito con un chip NOR.**

```
;; Ejemplo 5: Circuito complejo que combina XOR y NOR
(define circuito-complejo-ejemplo2
  (circuito-complejo '(p q) '(r)
    (list (circuito-simple '(p) '(s) (prim-chip (chip-xor)))
          (circuito-simple '(q s) '(r) (prim-chip (chip-nor))))))

;; Prueba para el Ejemplo 5 (Circuito complejo con XOR y NOR)
(circuito-complejo-inputs circuito-complejo-ejemplo2)
(circuito-complejo-sub-circuitos circuito-complejo-ejemplo2)
(newline)
```

**RESULTADO**

```
'(p q)
'((circuito-simple (p) (s) (chip-xor)) (circuito-simple (q s) (r) (chip-nor)))
```

## Representación por procedimientos

**para los ejemplos necesitamos crear primero unos cuantos chips primitivos:**

```
;Creación de chips primitivos
(define chip-and (crear-chip-prim 'AND))
(define chip-or (crear-chip-prim 'OR))
(define chip-not (crear-chip-prim 'NOT))
```

**y un circuito simple que usamos en varios ejemplos:**

```
;Creación de un circuito simple
(define circuito-simple (crear-circuito-simple '(a b) '(out) chip-and))
```

**luego, procedemos a crear los ejemplos con los chips y circuitos anteriores, en este caso creamos un circuito compuesto con subcircuitos:**

```
;Ejemplo 1: Creación de un circuito compuesto con subcircuitos
(define subcircuito-1 (crear-circuito-simple '(x y) '(out1) chip-or))
(define subcircuito-2 (crear-circuito-simple '(out1 z) '(out2) chip-not))
(define circuito-compuesto (crear-circuito-compuesto '(x y z) '(out2) (list subcircuito-1 subcircuito-2)))
```

**ejemplo 2, es un circuito simple:**

```
;Ejemplo 2: Circuito Simple con un chip XOR
(define circuito-simple-xor (crear-circuito-simple '(a b) '(out) (crear-chip-prim 'XOR)))
```

**ejemplo 3, es un circuito compuesto:**

```
;Ejemplo 3: Circuito Compuesto con un chip AND y un chip NAND
(define subcircuito-3 (crear-circuito-simple '(a b) '(out1) (crear-chip-prim 'AND)))
(define subcircuito-4 (crear-circuito-simple '(out1 c) '(out2) (crear-chip-prim 'NAND)))
(define circuito-compuesto-1 (crear-circuito-compuesto '(a b c) '(out2) (list subcircuito-3 subcircuito-4)))
```

**ejemplo 4:**

```
;Ejemplo 4: Circuito Simple con un chip NOR
(define circuito-simple-nor (crear-circuito-simple '(x y) '(out) (crear-chip-prim 'NOR)))
```

**ejemplo 5:**

```
;Ejemplo 5: Circuito Compuesto con chips OR y NOT
(define subcircuito-5 (crear-circuito-simple '(x y) '(out1) (crear-chip-prim 'OR)))
(define subcircuito-6 (crear-circuito-simple '(out1 z) '(out2) (crear-chip-prim 'NOT)))
(define circuito-compuesto-2 (crear-circuito-compuesto '(x y z) '(out2) (list subcircuito-5 subcircuito-6)))
```

**Luego, llamamos todos los ejemplos para imprimirlos y comprobar la calidad de la implementación:**

```
;Obtener información de todos los circuitos creados

(informacion-circuito circuito-compuesto)
(informacion-circuito circuito-simple-xor)
(informacion-circuito circuito-compuesto-1)
(informacion-circuito circuito-simple-nor)
(informacion-circuito circuito-compuesto-2)
```

**y finalmente verificamos si los ejemplos se imprimieron en consola correctamente:**

```
'(circuito-compuesto (x y z) (out2) ((circuito-simple (x y) (out1) OR) (circuito-simple (out1 z) (out2) NOT)))
'(circuito-simple (a b) (out) XOR)
'(circuito-compuesto (a b c) (out2) ((circuito-simple (a b) (out1) AND) (circuito-simple (out1 c) (out2) NAND)))
'(circuito-simple (x y) (out) NOR)
'(circuito-compuesto (x y z) (out2) ((circuito-simple (x y) (out1) OR) (circuito-simple (out1 z) (out2) NOT)))
```

## DataType

**Se define un tipo de dato llamado "circuito", que puede ser un circuito simple o compuesto. Un circuito simple incluye listas de símbolos para entradas y salidas, junto con un chip. Un circuito compuesto, en cambio, contiene otro circuito y una lista de circuitos, además de listas de entradas y salidas. También se define un tipo de dato llamado "chip", que puede ser un chip primitivo, que representa un chip lógico específico, o un chip compuesto, que incluye listas de símbolos para entradas y salidas y un circuito asociado.**

```
(define-datatype circuito circuito?
    (cir_simple (c1 (list-of symbol?))
                (c2 (list-of symbol?))
                (ch chip?))
    (cir_comp   (cir1 circuito?)
                (cir2 (list-of circuito?))
                (in (list-of symbol?))
                (out (list-of symbol?)))
                )

(define-datatype chip chip?
    (prim_chip
        (cp chip_prim?))
    (comp_chip
        (c1 (list-of symbol?))
        (c2 (list-of symbol?))
        (cir circuito?))
        )
```

**Después, se define un tipo de dato para chips primitivos. Estos incluyen diferentes tipos de compuertas lógicas, como AND, OR, NOR, NAND, NOT, XOR y XNOR.**

```
(define-datatype chip_prim chip_prim?
    (prim_or)
    (prim_and)
    (prim_nor)
    (prim_nand)
    (prim_not)
    (prim_xor)
    (prim_xnor)
    )
```

**Finalmente, se crean instancias de circuitos y chips. El primer chip se define como un chip compuesto que tiene cuatro entradas y una salida, y contiene un circuito compuesto que a su vez incluye un circuito simple con una compuerta AND y otras conexiones. La variable "circuito1" representa un circuito compuesto similar al del chip, mostrando cómo se pueden conectar varios circuitos y chips entre sí**

```
(define chip1       Juan David Perez, 2 hours ago •
    (comp_chip
    '(INA INB INC IND)
    '(OUTA)
    (cir_comp
        (cir_simple '(a b) '(e)
        (prim_chip (prim_and)))
        (list
        (cir_simple '(c d) '(f)
            (prim_chip (prim_and)))
        (cir_simple '(e f) '(g)
            (prim_chip (prim_or))))
    '(a b c d)
    '(g)))
    )

(define circuito1
    (cir_comp
        (cir_simple '(a b) '(e)
        (prim_chip (prim_and)))
        (list
        (cir_simple '(c d) '(f)
            (prim_chip (prim_and)))
        (cir_simple '(e f) '(g)
            (prim_chip (prim_or))))
    '(a b c d)
    '(g))
    )
```

**Aquí algunas comprobaciones para el datatype:**

- **comprobacion**

```
"datatype.rkt"> circuito1
(cir_comp
 (cir_simple '(a b) '(e) (prim_chip (prim_and)))
 (list
  (cir_simple '(c d) '(f) (prim_chip (prim_and)))
  (cir_simple '(e f) '(g) (prim_chip (prim_or))))
 '(a b c d)
 '(g))
```

- **aquí se comprueba la construcción del circuito simple**

```
"datatype.rkt"> (circuito? (cir_simple '(a b) '(e) (prim_chip (prim_and))))
#t
```

- **aqui se comprueba si un chip primitivo dentro de un circuito compuesto sería envolver el chip primitivo en un circuito simple**

```
"datatype.rkt"> (comp_chip '(a b) '(out) (cir_simple '(a b) '(out) (prim_chip (prim_or))))
(comp_chip '(a b) '(out) (cir_simple '(a b) '(out) (prim_chip (prim_or))))
"datatype.rkt"> (chip? (comp_chip '(a b) '(out) (cir_simple '(a b) '(out) (prim_chip (prim_or)))))
#t
```

## Parser y Unparser

**Se definen tipos de datos como "circuito", que puede ser simple o compuesto. Un circuito simple tiene entradas, salidas y un chip, mientras que un circuito compuesto incluye otros circuitos y listas de entradas y salidas**

```
(define-datatype circuito circuito?
  (cir_simple (c1 (list-of symbol?))
              (c2 (list-of symbol?))
              (ch chip?))
  (cir_comp   (cir1 circuito?)
              (cir2 (list-of circuito?))
              (in (list-of symbol?))
              (out (list-of symbol?))))

(define-datatype chip chip?
  (prim_chip
   (cp chip_prim?))
  (comp_chip
   (c1 (list-of symbol?))
   (c2 (list-of symbol?))
   (cir circuito?)))

(define-datatype chip_prim chip_prim?
  (prim_or)
  (prim_and)
  (prim_nor)
  (prim_nand)
  (prim_not)
  (prim_xor)
  (prim_xnor))
```

**A continuación, se implementan las funciones de parser, que son responsables de convertir listas en las estructuras de datos definidas. La función "parser-circuito" convierte listas en circuitos, identificando si son simples o compuestos. La función "parser-chip" convierte listas en chips, diferenciando entre chips primitivos y compuestos. La función "parser-chip_prim" convierte listas en chips primitivos, devolviendo el tipo de compuerta lógica correspondiente**

```
(define parser-circuito
  (lambda (list)
    (cond
      [(eq? (car list) 'cir_simple) (cir_simple (cadr list) (caddr list) (parser-chip (cadddr list)))]
      [(eq? (car list) 'cir_comp) (cir_comp (parser-circuito (cadr list))  (map parser-circuito (caddr list)) (cadddr list) (car (cdddr list)))]
    )
  )
)

(define parser-chip
  (lambda (list)
    (cond
      [(eq? (car list) 'prim_chip) (prim_chip (parser-chip_prim (cadr list)))]
      [(eq? (car list) 'comp_chip) (comp_chip (cadr list) (caddr list) (parser-circuito (cadddr list)))]
    )
  )
)

(define parser-chip_prim
  (lambda (list)
    (cond
      [(eq? (car list) 'prim_and) (prim_and)]
      [(eq? (car list) 'prim_or) (prim_or)]
      [(eq? (car list) 'prim_not) (prim_not)]
      [(eq? (car list) 'prim_xor) (prim_xor)]
      [(eq? (car list) 'prim_nand) (prim_nand)]
      [(eq? (car list) 'prim_nor) (prim_nor)]
      [(eq? (car list) 'prim_xnor) (prim_xnor)]
    )
  )
)
```

**Finalmente, se implementan las funciones de unparser, que convierten las estructuras de datos de vuelta a listas. La función "unparser-circuito" genera la representación correspondiente para circuitos, mientras que "unparser-chip" y "unparser-chip_prim" hacen lo mismo para chips y chips primitivos, respectivamente**

```
(define parser-circuito
  (lambda (list)
    (cond
      [(eq? (car list) 'cir_simple) (cir_simple (cadr list) (caddr list) (parser-chip (cadddr list)))]
      [(eq? (car list) 'cir_comp) (cir_comp (parser-circuito (cadr list))  (map parser-circuito (caddr list)) (cadddr list) (car (cdddr list)))]
    )
  )
)

(define parser-chip
  (lambda (list)
    (cond
      [(eq? (car list) 'prim_chip) (prim_chip (parser-chip_prim (cadr list)))]
      [(eq? (car list) 'comp_chip) (comp_chip (cadr list) (caddr list) (parser-circuito (cadddr list)))]
    )
  )
)

(define parser-chip_prim
  (lambda (list)
    (cond
      [(eq? (car list) 'prim_and) (prim_and)]
      [(eq? (car list) 'prim_or) (prim_or)]
      [(eq? (car list) 'prim_not) (prim_not)]
      [(eq? (car list) 'prim_xor) (prim_xor)]
      [(eq? (car list) 'prim_nand) (prim_nand)]
      [(eq? (car list) 'prim_nor) (prim_nor)]
      [(eq? (car list) 'prim_xnor) (prim_xnor)]
    )
  )
)
```

**ejemplos para comprobar paser y unparser**

```
"parser_unparser.rkt"> (chip? chip1-list)
#f
```

```
"parser_unparser.rkt"> (chip? (parser-chip chip1-list))
#t
"parser_unparser.rkt"> (parser-circuito cir2-list)
(cir_comp
 (cir_simple '(a b) '(e) (prim_chip (prim_and)))
 (list
  (cir_simple '(c d) '(f) (prim_chip (prim_and)))
  (cir_simple '(e f) '(g) (prim_chip (prim_or))))
 '(a b c d)
 '(a b c d))
```

```
"parser_unparser.rkt"> (chip? (parser-chip chip1-list))
#t
"parser_unparser.rkt"> (parser-circuito cir2-list)
(cir_comp
 (cir_simple '(a b) '(e) (prim_chip (prim_and)))
 (list
  (cir_simple '(c d) '(f) (prim_chip (prim_and)))
  (cir_simple '(e f) '(g) (prim_chip (prim_or))))
 '(a b c d)
 '(a b c d))
```

```
"parser_unparser.rkt"> (parser-circuito cir2-list)
(cir_comp
 (cir_simple '(a b) '(e) (prim_chip (prim_and)))
 (list
  (cir_simple '(c d) '(f) (prim_chip (prim_and)))
  (cir_simple '(e f) '(g) (prim_chip (prim_or))))
 '(a b c d)
 '(a b c d))
```

```
"parser_unparser.rkt"> (parser-chip chip1-list)
(cir_simple '(a b) '(e) (prim_chip (prim_and)))
(list
 (cir_simple '(c d) '(f) (prim_chip (prim_and)))
 (cir_simple '(e f) '(g) (prim_chip (prim_or))))
'(a b c d)
'(a b c d))
```

```
"parser_unparser.rkt"> (circuito? (unparser-circuito (paser-circuito cir1-list)))
  '(a b c d)
  '(a b c d)))
```

```
"parser_unparser.rkt"> (circuito? (unparser-circuito (parser-circuito cir1-list)))
#f
```