TALLER 1 FLP. TAD

Juan Carlos Rojas Quintero-2359358

Juan Miguel Palacios Doncel-2359321

Yeifer Ronaldo Muñoz Valencia-2278665

Fundamentos De Interpretación Y Compilación De Lenguajes De Programación

Carlos Andrés Delgado Saavedra

Universidad Del Valle

Sede Tuluá

Facultad De Ingeniería

Ingeniería de Sistemas

Tuluá – Colombia

2024

Instancias de los constructores con las que haremos las pruebas en el archivo “representación-listas.rkt”.

```
(define simple-circuit-1 (simple-circuit '(a) '(b) (prim-chip(chip-and))))
(define simple-circuit-2 (simple-circuit '(a b) '(c) (prim-chip(chip-or))))
(define simple-circuit-3 (simple-circuit '(a) '(b) (prim-chip(chip-not))))
(define simple-circuit-4 (simple-circuit '(a) '(b)  (prim-chip(chip-or))))
(define simple-circuit-5 (simple-circuit '(a b) '(e) (prim-chip(chip-nor))))

;;-------------------------------------------------------------------------------------

(define complex-circuit-1 (complex-circuit simple-circuit-1 (list simple-circuit-2) '(a) '(b)))
(define complex-circuit-2 (complex-circuit simple-circuit-2 (list simple-circuit-1 simple-circuit-2) '(x) '(y)))
(define complex-circuit-3 (complex-circuit complex-circuit-1 (list simple-circuit-1 complex-circuit-2) '(w) '(y z)))
(define complex-circuit-4 (complex-circuit simple-circuit-4 (list simple-circuit-1)'(a d) '(b)))
(define complex-circuit-5 (complex-circuit complex-circuit-4 (list simple-circuit-3 simple-circuit-2) '(x y) '(a b)))

;;-----------------------------------------------------------------------------------------
(define comp-chip-1 (comp-chip '(a b c) '(d) simple-circuit-1))
(define comp-chip-2 (comp-chip '(a b) '(c d) complex-circuit-2))
(define comp-chip-3 (comp-chip '(a) '(b) simple-circuit-4))
(define comp-chip-4 (comp-chip '(a b c) '(e f) simple-circuit-2))
(define comp-chip-5 (comp-chip '(a b c) '(d e f) simple-circuit-3))
;;-----------------------------------------------------------------------------------------

(define chip-or-1(chip-or))
(define chip-and-1(chip-and))
(define chip-nand-1(chip-nand))
(define chip-xor-1(chip-xor))
(define chip-nor-1(chip-nor))
(define chip-xnor-1(chip-xnor))
(define chip-not-1(chip-not))

;;-----------------------------------------------------------------------------------------

(define prim-chip-1(prim-chip chip-and-1))
```

Probaremos los predicados con las instancias anteriores:

- Circuitos simples

```
lleres_FLP\taller-1-flp> racket --repl --eval '(enter! (file \"c:/
Welcome to Racket v8.14 [cs].
"representacion-listas.rkt"> (simple-circuit? simple-circuit-1)
#t
"representacion-listas.rkt"> (simple-circuit? simple-circuit-2)
#t
"representacion-listas.rkt"> (simple-circuit? simple-circuit-3)
#t
"representacion-listas.rkt"> (simple-circuit? simple-circuit-4)
#t
"representacion-listas.rkt"> (simple-circuit? simple-circuit-5)
#t
```

- Circuitos complejos

```
PS C:\Users\USUARIO\Desktop\Talleres_FLP\taller-1-flp> racket --repl
t\"))'
Welcome to Racket v8.14 [cs].
"representacion-listas.rkt"> (complex-circuit? complex-circuit-1)
#t
"representacion-listas.rkt"> (complex-circuit? complex-circuit-2)
#t
"representacion-listas.rkt"> (complex-circuit? complex-circuit-3)
#t
"representacion-listas.rkt"> (complex-circuit? complex-circuit-4)
#t
"representacion-listas.rkt"> (complex-circuit? complex-circuit-5)
#t
"representacion-listas.rkt">
```

- Chips primitivos

```
Welcome to Racket v8.14 [cs].
"representacion-listas.rkt"> (prim-chip? prim-chip-1)
#t
"representacion-listas.rkt"> (chip-not? chip-not-1)
#t
"representacion-listas.rkt"> (chip-or? chip-or-1)
#t
"representacion-listas.rkt"> (chip-and? chip-and-1)
#t
"representacion-listas.rkt"> (chip-xor? chip-xor-1)
#t
"representacion-listas.rkt"> (chip-nand? chip-nand-1)
#t
"representacion-listas.rkt"> (chip-nor? chip-nor-1)
#t
"representacion-listas.rkt"> (chip-xnor? chip-xnor-1)
#t
```

- Chips compuestos

```
Welcome to Racket v8.14 [cs].
"representacion-listas.rkt"> (comp-chip? comp-chip-1)
#t
"representacion-listas.rkt"> (comp-chip? comp-chip-2)
#t
"representacion-listas.rkt"> (comp-chip? comp-chip-3)
#t
"representacion-listas.rkt"> (comp-chip? comp-chip-4)
#t
"representacion-listas.rkt"> (comp-chip? comp-chip-5)
#t
```

Ahora probaremos los extractores:

- Circuitos simples

```
Welcome to Racket v8.14 [cs].
"representacion-listas.rkt"> (simple-circuit->in simple-circuit-2)
'(a b)
"representacion-listas.rkt"> (simple-circuit->out simple-circuit-2)
'(c)
"representacion-listas.rkt"> (simple-circuit->chip simple-circuit-2)
'(prim-chip (chip-or))
```

- Circuitos complejos

```
Welcome to Racket v8.14 [cs].
"representacion-listas.rkt"> (complex-circuit->circ complex-circuit-2)
'(simple-circuit (a b) (c) (prim-chip (chip-or)))
"representacion-listas.rkt"> (complex-circuit->lcircs complex-circuit-2)
'((simple-circuit (a) (b) (prim-chip (chip-and)))
  (simple-circuit (a b) (c) (prim-chip (chip-or))))
"representacion-listas.rkt"> (complex-circuit->in complex-circuit-2)
'(x)
"representacion-listas.rkt"> (complex-circuit->out complex-circuit-2)
'(y)
```

- Chips primitivos

```
Welcome to Racket v8.14 [cs].
"representacion-listas.rkt"> (prim-chip->chip-prim prim-chip-1)
'(chip-and)
"representacion-listas.rkt"> (chip-or->chip chip-or-1)
'chip-or
"representacion-listas.rkt"> (chip-and->chip chip-and-1)
'chip-and
"representacion-listas.rkt"> (chip-nand->chip chip-nand-1)
'chip-nand
"representacion-listas.rkt"> (chip-nor->chip chip-nor-1)
'chip-nor
"representacion-listas.rkt"> (chip-not->chip chip-not-1)
'chip-not
"representacion-listas.rkt"> (chip-xnor->chip chip-xnor-1)
'chip-xnor
"representacion-listas.rkt"> (chip-xor->chip chip-xor-1)
'chip-xor
```

- Chips compuestos

```
Welcome to Racket v8.14 [cs].
"representacion-listas.rkt"> (comp-chip->in comp-chip-3)
'(a)
"representacion-listas.rkt"> (comp-chip->out comp-chip-3)
'(b)
"representacion-listas.rkt"> (comp-chip->circ comp-chip-3)
'(simple-circuit (a) (b) (prim-chip (chip-or)))
```

Instancias de los constructores con las que haremos las pruebas en el archivo “representación-procedimientos.rkt”

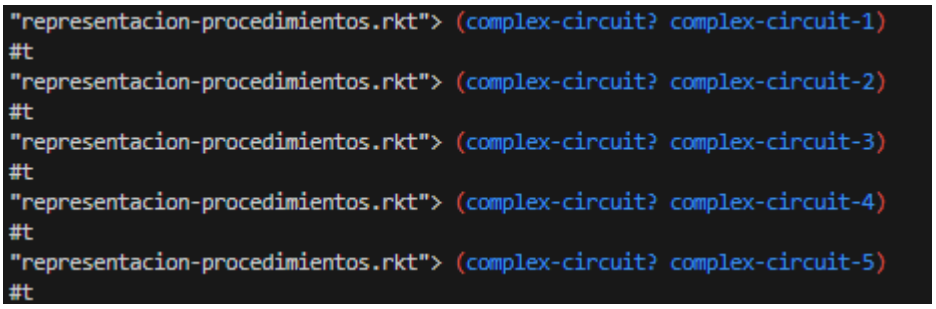
```
;; Pruebas

(define simple-circuit-1 (simple-circuit '(a c) '(b) (prim-chip(chip-and))))
(define simple-circuit-2 (simple-circuit '(a b c) '(c) (prim-chip(chip-or))))
(define simple-circuit-3 (simple-circuit '(a) '(b c) (prim-chip(chip-not))))
(define simple-circuit-4 (simple-circuit '(a b) '(c d) (prim-chip(chip-or))))
(define simple-circuit-5 (simple-circuit '(c d) '(a) (prim-chip(chip-nor))))

;;-------------------------------------------------------------------------

(define complex-circuit-1 (complex-circuit simple-circuit-1 (list simple-circuit-2) '(a) '(b)))
(define complex-circuit-2 (complex-circuit simple-circuit-2 (list simple-circuit-1 simple-circuit-2) '(x) '(y)))
(define complex-circuit-3 (complex-circuit complex-circuit-1 (list simple-circuit-1 complex-circuit-2) '(w) '(y z)))
(define complex-circuit-4 (complex-circuit simple-circuit-4 (list simple-circuit-1)'(a d) '(b)))
(define complex-circuit-5 (complex-circuit complex-circuit-4 (list simple-circuit-3 simple-circuit-2) '(x y) '(a b)))

;;-------------------------------------------------------------------------------

(define chip-or-1(chip-or))
(define chip-and-1(chip-and))
(define chip-nand-1(chip-nand))
(define chip-xor-1(chip-xor))
(define chip-nor-1(chip-nor))
(define chip-xnor-1(chip-xnor))
(define chip-not-1(chip-not))

;;-----------------------------------------------------------------------------------

(define comp-chip-1 (comp-chip '(a b c) '(d) simple-circuit-1))
(define comp-chip-2 (comp-chip '(a b) '(c d) complex-circuit-2))
(define comp-chip-3 (comp-chip '(a) '(b) simple-circuit-4))
(define comp-chip-4 (comp-chip '(a b c) '(e f) simple-circuit-2))
(define comp-chip-5 (comp-chip '(a b c) '(d e f) simple-circuit-3))
;;-----------------------------------------------------------------------------------
(define prim-chip-1(prim-chip chip-and-1))
```

Probaremos los predicados con las instancias anteriores:

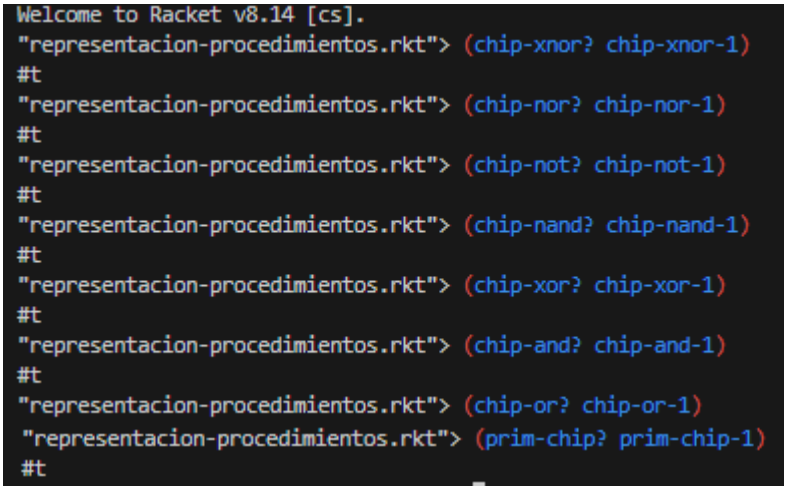
- Circuitos simples

```
Welcome to Racket v8.14 [cs].
"representacion-procedimientos.rkt"> (simple-circuit? simple-circuit-1)
#t
"representacion-procedimientos.rkt"> (simple-circuit? simple-circuit-2)
#t
"representacion-procedimientos.rkt"> (simple-circuit? simple-circuit-3)
#t
"representacion-procedimientos.rkt"> (simple-circuit? simple-circuit-4)
#t
"representacion-procedimientos.rkt"> (simple-circuit? simple-circuit-5)
#t
```

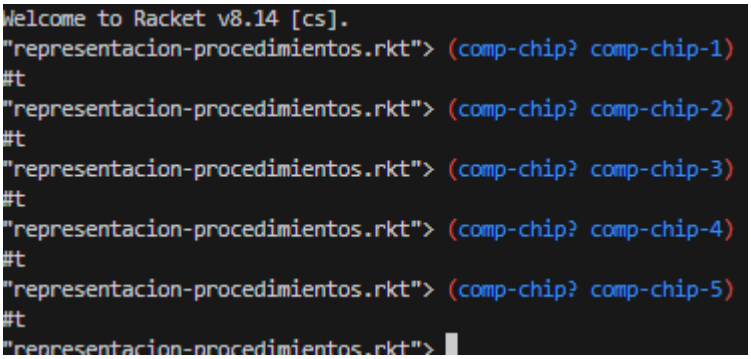
- Circuitos complejos



- Chips primitivos



- Chips compuestos



Ahora probaremos los extractores:

- Circuitos simples

```
"representacion-procedimientos.rkt"> (simple-circuit->in  simple-circuit-4)
'(a b)
"representacion-procedimientos.rkt"> (simple-circuit->out  simple-circuit-4)
'(c d)
"representacion-procedimientos.rkt"> (simple-circuit->chip  simple-circuit-4)
#<procedure:...-procedimientos.rkt:60:4>
```

- Circuitos complejos

```
Welcome to Racket v8.14 [cs].
"representacion-procedimientos.rkt"> (complex-circuit->circ complex-circuit-4)
#<procedure:...-procedimientos.rkt:16:4>
"representacion-procedimientos.rkt"> (complex-circuit->lcircs complex-circuit-4)
'(#<procedure:...-procedimientos.rkt:16:4>)
"representacion-procedimientos.rkt"> (complex-circuit->in complex-circuit-4)
'(a d)
"representacion-procedimientos.rkt"> (complex-circuit->out complex-circuit-4)
'(b)
"representacion-procedimientos.rkt">
```

- Chips primitivos

```
Welcome to Racket v8.14 [cs].
"representacion-procedimientos.rkt"> (prim-chip->chip-prim prim-chip-1)
#<procedure:...-procedimientos.rkt:131:4>
"representacion-procedimientos.rkt"> (chip-or->symbol chip-or-1)
'chip-or
"representacion-procedimientos.rkt"> (chip-and->symbol chip-and-1)
'chip-and
"representacion-procedimientos.rkt"> (chip-not->symbol chip-not-1)
'chip-not
"representacion-procedimientos.rkt"> (chip-xor->symbol chip-xor-1)
'chip-xor
"representacion-procedimientos.rkt"> (chip-nand->symbol chip-nand-1)
'chip-nand
"representacion-procedimientos.rkt"> (chip-nor->symbol chip-nor-1)
'chip-nor
"representacion-procedimientos.rkt"> (chip-xnor->symbol chip-xnor-1)
'chip-xnor
```

- Chips compuestos

```
"representacion-procedimientos.rkt"> (comp-chip->in comp-chip-4)
'(a b c)
"representacion-procedimientos.rkt"> (comp-chip->out comp-chip-4)
'(e f)
"representacion-procedimientos.rkt"> (comp-chip->circ comp-chip-4)
#<procedure:...-procedimientos.rkt:16:4>
```

Instancias de los constructores con las que haremos las pruebas en el archivo “representacion-datatype.rkt”

```
;; Pruebas

;;-------------------------------------------------------------------------------------

(define simple-circuit-1 (simple-circuit '(a) '(b) (prim-chip(chip_and))))
(define simple-circuit-2 (simple-circuit '(a b c) '(c d) (prim-chip(chip_or))))
(define simple-circuit-3 (simple-circuit '(a) '(b) (prim-chip(chip_not))))
(define simple-circuit-4 (simple-circuit '(a) '(b)  (prim-chip(chip_or))))
(define simple-circuit-5 (simple-circuit '(a b) '(c d) (prim-chip(chip_nor))))

;;-------------------------------------------------------------------------------------

(define complex-circuit-1 (complex-circuit simple-circuit-1 (list simple-circuit-2) '(a) '(b)))
(define complex-circuit-2 (complex-circuit simple-circuit-2 (list simple-circuit-1 simple-circuit-2) '(x) '(y)))
(define complex-circuit-3 (complex-circuit complex-circuit-1 (list simple-circuit-1 complex-circuit-2) '(w) '(y z)))
(define complex-circuit-4 (complex-circuit simple-circuit-4 (list simple-circuit-1)'(a d) '(b)))
(define complex-circuit-5 (complex-circuit complex-circuit-4 (list simple-circuit-3 simple-circuit-2) '(x y) '(a b)))

;;-----------------------------------------------------------------------------------------

(define comp-chip-1 (comp-chip '(a b c) '(d) simple-circuit-1))
(define comp-chip-2 (comp-chip '(a b) '(c d) complex-circuit-2))
(define comp-chip-3 (comp-chip '(a) '(b) simple-circuit-4))
(define comp-chip-4 (comp-chip '( a b c d) '(e f) complex-circuit-5))
(define comp-chip-5 (comp-chip '(a b c) '(d e f) simple-circuit-3))

;;-------------------------------------------------------------------------------------------

(define prim-chip-1(prim-chip(chip_and)))
(define chip-prim-1 (chip_nand))
(define chip-prim-2(chip_or))
(define chip-prim-3(chip_xor))
```

Empezaremos probando los predicados:

- Circuitos

```
Welcome to Racket v8.14 [cs].
"representacion-datatype.rkt"> (circuito? simple-circuit-1)
#t
"representacion-datatype.rkt"> (circuito? simple-circuit-3)
#t
"representacion-datatype.rkt"> (circuito? simple-circuit-5)
#t
"representacion-datatype.rkt"> (circuito? complex-circuit-2)
#t
"representacion-datatype.rkt"> (circuito? complex-circuit-4)
#t
"representacion-datatype.rkt"> (circuito? complex-circuit-5)
#t
"representacion-datatype.rkt"> 
```

- Chips

```
Welcome to Racket v8.14 [cs].
"representacion-datatype.rkt"> (chip? comp-chip-1)
#t
"representacion-datatype.rkt"> (chip? comp-chip-4)
#t
"representacion-datatype.rkt"> (chip? comp-chip-5)
#t
"representacion-datatype.rkt"> (chip-prim? chip-prim-1)
#t
"representacion-datatype.rkt">
```

- Chips primitivos

```
Welcome to Racket v8.14 [cs].
"representacion-datatype.rkt"> (chip-prim? chip-prim-1)
#t
"representacion-datatype.rkt"> (chip-prim? chip-prim-2)
#t
"representacion-datatype.rkt"> (chip-prim? chip-prim-3)
#t
"representacion-datatype.rkt">
```

Instancias de los constructores con las que haremos las pruebas de parser en el archivo “parser-unparser.rkrt”.

```
;; Pruebas para Parser

;;-------------------------------------------------------------------------------------

(define simple-circuit-1 (simple-circuit-list '(a) '(b) (prim-chip-list(chip-and-list))))
(define simple-circuit-2 (simple-circuit-list '(a b c) '(c d) (prim-chip-list(chip-or-list))))
(define simple-circuit-3 (simple-circuit-list '(a) '(b) (prim-chip-list(chip-not-list))))
(define simple-circuit-4 (simple-circuit-list '(a) '(b)  (prim-chip-list(chip-or-list))))
(define simple-circuit-5 (simple-circuit-list '(a b) '(c d) (prim-chip-list(chip-nor-list))))

;;-------------------------------------------------------------------------------------

(define complex-circuit-1 (complex-circuit-list simple-circuit-1 (list simple-circuit-2) '(a) '(b)))
(define complex-circuit-2 (complex-circuit-list simple-circuit-2 (list simple-circuit-1 simple-circuit-2) '(x) '(y)))
(define complex-circuit-3 (complex-circuit-list complex-circuit-1 (list simple-circuit-1 complex-circuit-2) '(w) '(y z)))
(define complex-circuit-4 (complex-circuit-list simple-circuit-4 (list simple-circuit-1)'(a d) '(b)))
(define complex-circuit-5 (complex-circuit-list complex-circuit-4 (list simple-circuit-3 simple-circuit-2) '(x y) '(a b)))

;;-------------------------------------------------------------------------------------

(define comp-chip-1 (comp-chip-list '(a b c) '(d) simple-circuit-1))
(define comp-chip-2 (comp-chip-list '(a b) '(c d) complex-circuit-2))
(define comp-chip-3 (comp-chip-list '(a) '(b) simple-circuit-4))
(define comp-chip-4 (comp-chip-list '( a b c d) '(e f) complex-circuit-5))
(define comp-chip-5 (comp-chip-list '(a b c) '(d e f) simple-circuit-3))

;;-------------------------------------------------------------------------------------

(define prim-chip-1(prim-chip-list(chip-and-list)))
(define chip-prim-1 (chip-nand-list))
(define chip-prim-2(chip-or-list))
(define chip-prim-3(chip-xor-list))
```

- Circuitos simples

```
Welcome to Racket v8.14 [cs].
"parser-unparser.rkt"> (parser simple-circuit-1)
(simple-circuit '(a) '(b) (prim-chip (chip_and)))
"parser-unparser.rkt"> (parser simple-circuit-2)
(simple-circuit '(a b c) '(c d) (prim-chip (chip_or)))
"parser-unparser.rkt"> (parser simple-circuit-3)
(simple-circuit '(a) '(b) (prim-chip (chip_not)))
"parser-unparser.rkt"> (parser simple-circuit-4)
(simple-circuit '(a) '(b) (prim-chip (chip_or)))
"parser-unparser.rkt"> (parser simple-circuit-5)
(simple-circuit '(a b) '(c d) (prim-chip (chip_nor)))
"parser-unparser.rkt"> 
```

- Circuitos complejos

```
"parser-unparser.rkt"> (parser complex-circuit-1)
(complex-circuit
 (simple-circuit '(a) '(b) (prim-chip (chip_and)))
 (list (simple-circuit '(a b c) '(c d) (prim-chip (chip_or))))
 '(a)
 '(b))
"parser-unparser.rkt"> (parser complex-circuit-2)
(complex-circuit
 (simple-circuit '(a b c) '(c d) (prim-chip (chip_or)))
 (list
  (simple-circuit '(a) '(b) (prim-chip (chip_and)))
  (simple-circuit '(a b c) '(c d) (prim-chip (chip_or))))
 '(x)
 '(y))
"parser-unparser.rkt"> (parser complex-circuit-3)
(complex-circuit
 (complex-circuit
  (simple-circuit '(a) '(b) (prim-chip (chip_and)))
  (list (simple-circuit '(a b c) '(c d) (prim-chip (chip_or))))
  '(a)
  '(b))
 (list
  (simple-circuit '(a) '(b) (prim-chip (chip_and)))
  (complex-circuit
   (simple-circuit '(a b c) '(c d) (prim-chip (chip_or)))
   (list
    (simple-circuit '(a) '(b) (prim-chip (chip_and)))
    (simple-circuit '(a b c) '(c d) (prim-chip (chip_or))))
   '(x)
   '(y)))
 '(w)
 '(y z))
"parser-unparser.rkt"> (parser complex-circuit-4)
(complex-circuit
 (simple-circuit '(a) '(b) (prim-chip (chip_or)))
 (list (simple-circuit '(a) '(b) (prim-chip (chip_and))))
 '(a d)
 '(b))
"parser-unparser.rkt"> (parser complex-circuit-5)
(complex-circuit
 (complex-circuit
  (simple-circuit '(a) '(b) (prim-chip (chip_or)))
  (list (simple-circuit '(a) '(b) (prim-chip (chip_and))))
  '(a d)
  '(b))
 (list
  (simple-circuit '(a) '(b) (prim-chip (chip_not)))
  (simple-circuit '(a b c) '(c d) (prim-chip (chip_or))))
 '(x y)
 '(a b))
```

Instancias de los constructores con las que haremos las pruebas de unparser en el archivo “parser-unparser.rkrt”.

```
;;Pruebas para  Unparser

(define simple-circuit-Unp-1 (simple-circuit '(a) '(b) (prim-chip(chip_and))))
(define simple-circuit-Unp-2 (simple-circuit '(a b c) '(c d) (prim-chip(chip_or))))
(define simple-circuit-Unp-3 (simple-circuit '(a) '(b) (prim-chip(chip_not))))
(define simple-circuit-Unp-4 (simple-circuit '(a) '(b)  (prim-chip(chip_or))))
(define simple-circuit-Unp-5 (simple-circuit '(a b) '(c d) (prim-chip(chip_nor))))

;;-------------------------------------------------------------------------------

(define complex-circuit-Unp-1 (complex-circuit simple-circuit-Unp-1 (list simple-circuit-Unp-2) '(a) '(b)))
(define complex-circuit-Unp-2 (complex-circuit simple-circuit-Unp-2 (list simple-circuit-Unp-1 simple-circuit-Unp-2) '(x) '(y)))
(define complex-circuit-Unp-3 (complex-circuit complex-circuit-Unp-1 (list simple-circuit-Unp-1 complex-circuit-Unp-2) '(w) '(y z)))
(define complex-circuit-Unp-4 (complex-circuit simple-circuit-Unp-4 (list simple-circuit-Unp-1)'(a d) '(b)))
(define complex-circuit-Unp-5 (complex-circuit complex-circuit-Unp-4 (list simple-circuit-Unp-3 simple-circuit-Unp-2) '(x y) '(a b)))

;;-----------------------------------------------------------------------------------

(define comp-chip-Unp-1 (comp-chip '(a b c) '(d) simple-circuit-Unp-1))
(define comp-chip-Unp-2 (comp-chip '(a b) '(c d) complex-circuit-Unp-2))
(define comp-chip-Unp-3 (comp-chip '(a) '(b) simple-circuit-Unp-4))
(define comp-chip-Unp-4 (comp-chip '( a b c d) '(e f) complex-circuit-Unp-5))
(define comp-chip-Unp-5 (comp-chip '(a b c) '(d e f) simple-circuit-Unp-3))

;;---------------------------------------------------------------------------------------

(define prim-chip-Unp-1(prim-chip(chip_and)))
(define chip-prim-Unp-1 (chip_nand))
(define chip-prim-Unp-2(chip_or))
(define chip-prim-Unp-3(chip_xor))
```

- Circuitos simples

```
Welcome to Racket v8.14 [cs].
"parser-unparser.rkt"> (unparser simple-circuit-Unp-1)
'(simple-circuit (a) (b) (prim-chip (chip-and)))
"parser-unparser.rkt"> (unparser simple-circuit-Unp-2)
'(simple-circuit (a b c) (c d) (prim-chip (chip-or)))
"parser-unparser.rkt"> (unparser simple-circuit-Unp-3)
'(simple-circuit (a) (b) (prim-chip (chip-not)))
"parser-unparser.rkt"> (unparser simple-circuit-Unp-4)
'(simple-circuit (a) (b) (prim-chip (chip-or)))
"parser-unparser.rkt"> (unparser simple-circuit-Unp-5)
'(simple-circuit (a b) (c d) (prim-chip (chip-nor)))
"parser-unparser.rkt">
```

- Circuitos complejos