

Introducción a la Programación

Datos recursivos

Árboles de expresión, ownership y funciones que siguen a los datos

Gerson Garrido

Semana 13

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Por qué `2 + 3 * x` es un árbol
2. Un tipo que se apunta a sí mismo
3. Construir: quién es owner de cada nodo
4. La función sigue a los datos. Liberar en orden
5. Escribir con los paréntesis necesarios
6. Evaluar: un entorno y una función que puede fallar
7. Transformar: simplificar y derivar

> **Lectura**
>
> Apuntes de la semana 13. Código en `code/week-13`. Los ejemplos en C son de fórmulas lógicas: `Expr` es su laboratorio.

SECCIÓN 01

Árboles

01



La pregunta de la semana

En tres semanas van a escribir un **intérprete***(interpreter)* de un lenguaje pequeño, Mini. ¿Cómo se guarda una expresión?

2 + 3 * X con x = 4

La pregunta de la semana

En tres semanas van a escribir un **intérprete***(interpreter)* de un lenguaje pequeño, Mini. ¿Cómo se guarda una expresión?

```
2  +  3  *  x        con x = 4
```

- Como lista de símbolos, de izquierda a derecha: $2 + 3 = 5$, y $5 \times 4 = 20$.

La pregunta de la semana

En tres semanas van a escribir un **intérprete**(*interpreter*) de un lenguaje pequeño, Mini. ¿Cómo se guarda una expresión?

```
2  +  3  *  x        con x = 4
```

- Como lista de símbolos, de izquierda a derecha: $2 + 3 = 5$, y $5 \times 4 = 20$.
- Pero vale **14**: la multiplicación va primero.

La pregunta de la semana

En tres semanas van a escribir un **intérprete**(*interpreter*) de un lenguaje pequeño, Mini. ¿Cómo se guarda una expresión?

```
2  +  3  *  x        con x = 4
```

- Como lista de símbolos, de izquierda a derecha: $2 + 3 = 5$, y $5 \times 4 = 20$.
- Pero vale **14**: la multiplicación va primero.
- La lista tiene los símbolos. No dice **qué se opera con qué**.

La pregunta de la semana

En tres semanas van a escribir un **intérprete**(*interpreter*) de un lenguaje pequeño, Mini. ¿Cómo se guarda una expresión?

```
2  +  3  *  x        con x = 4
```

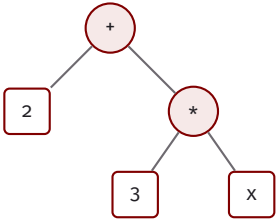
- Como lista de símbolos, de izquierda a derecha: $2 + 3 = 5$, y $5 \times 4 = 20$.
- Pero vale **14**: la multiplicación va primero.
- La lista tiene los símbolos. No dice **qué se opera con qué**.

Esta semana y la próxima

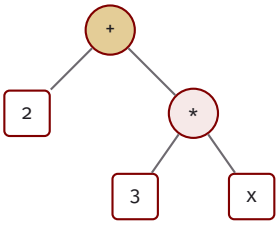
Hoy los árboles se construyen a mano, con funciones. La semana 14 los construirá a partir de texto.

Un árbol de expresión

- Cada símbolo es un **nodo**.

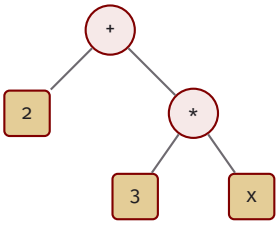


Un árbol de expresión



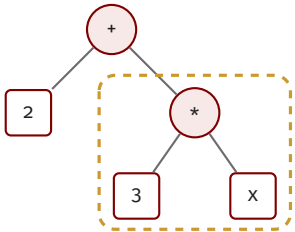
- Cada símbolo es un **nodo**.
- El de arriba es la **raíz**.

Un árbol de expresión



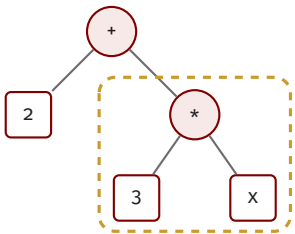
- Cada símbolo es un **nodo**.
- El de arriba es la **raíz**.
- Los que no tienen nada debajo son **hojas**.

Un árbol de expresión



- Cada símbolo es un **nodo**.
- El de arriba es la **raíz**.
- Los que no tienen nada debajo son **hojas**.
- Un hijo, con todo lo que tiene debajo, es un **subárbol**: un árbol completo. Por eso la definición es recursiva.

Un árbol de expresión

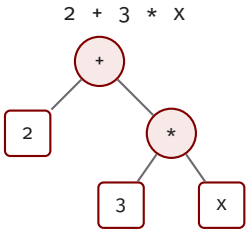


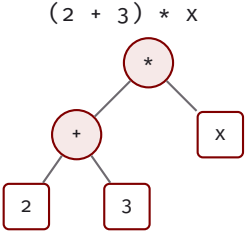
- Cada símbolo es un **nodo**.
- El de arriba es la **raíz**.
- Los que no tienen nada debajo son **hojas**.
- Un hijo, con todo lo que tiene debajo, es un **subárbol**: un árbol completo. Por eso la definición es recursiva.

Frente a la semana 11

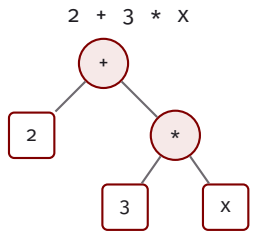
El árbol de decisiones era un dibujo de lo que hacía un programa. Este **existe**: son bloques en el heap, unidos por punteros.

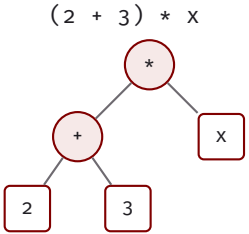
Los paréntesis no están en el árbol





Los paréntesis no están en el árbol



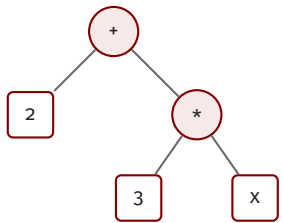


Idea clave

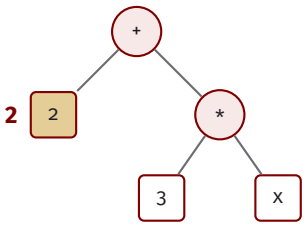
El orden de evaluación, la precedencia y los paréntesis del texto quedaron convertidos en la **forma** del árbol.

Evaluar es recorrer

Para saber cuánto vale un nodo, hay que saber cuánto valen sus hijos.



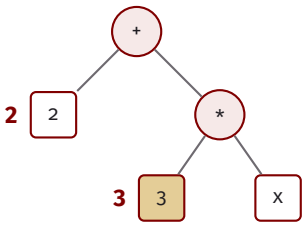
Evaluar es recorrer



Para saber cuánto vale un nodo, hay que saber cuánto valen sus hijos.

- Se baja hasta las hojas: casos base.

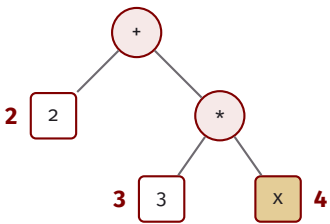
Evaluar es recorrer



Para saber cuánto vale un nodo, hay que saber cuánto valen sus hijos.

- Se baja hasta las hojas: casos base.

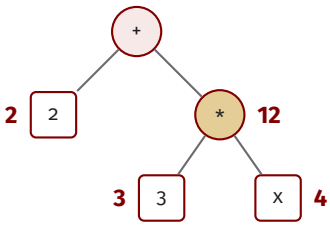
Evaluar es recorrer



Para saber cuánto vale un nodo, hay que saber cuánto valen sus hijos.

- Se baja hasta las hojas: casos base.

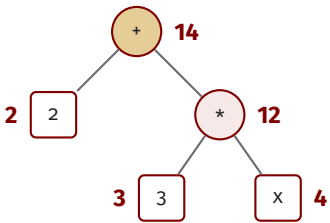
Evaluar es recorrer



Para saber cuánto vale un nodo, hay que saber cuánto valen sus hijos.

- Se baja hasta las hojas: casos base.
- Los resultados se combinan al volver.

Evaluar es recorrer



Para saber cuánto vale un nodo, hay que saber cuánto valen sus hijos.

- Se baja hasta las hojas: casos base.
- Los resultados se combinan al volver.
- Es la recursión de la semana 9.

Lo diferente

El problema más pequeño no se inventa restando 1. **Ya está en los datos**: es el hijo.

SECCIÓN 02

Un tipo recursivo

02



Un nodo no puede contener a otro

errors/self_by_value.c

```c
struct node {
    int kind;
    struct node left;
    struct node right;
};
```

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/self_by_value.c -o build/self_by_value
errors/self_by_value.c:4:17: error: field 'left' has incomplete type
```

Un nodo no puede contener a otro

errors/self_by_value.c

```c
struct node {
    int kind;
    struct node left;
    struct node right;
};
```

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/self_by_value.c -o build/self_by_value
errors/self_by_value.c:4:17: error: field 'left' has incomplete type
```

- Un `struct node` tendría que medir más que dos `struct node`.

Un nodo no puede contener a otro

errors/self_by_value.c

```c
struct node {
    int kind;
    struct node left;
    struct node right;
};
```

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/self_by_value.c -o build/self_by_value
errors/self_by_value.c:4:17: error: field 'left' has incomplete type
```

- Un `struct node` tendría que medir más que dos `struct node`.
- La salida ya la vieron en la semana 6: para reservar un **puntero**, el compilador no necesita saber qué hay al otro lado.

Un nodo no puede contener a otro

errors/self_by_value.c

```c
struct node {
    int kind;
    struct node left;
    struct node right;
};
```

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/self_by_value.c -o build/self_by_value
errors/self_by_value.c:4:17: error: field 'left' has incomplete type
```

- Un `struct node` tendría que medir más que dos `struct node`.
- La salida ya la vieron en la semana 6: para reservar un **puntero**, el compilador no necesita saber qué hay al otro lado.
- El nodo no contiene a sus hijos: **apunta** a ellos.

La fórmula lógica de la clase

logic.h

```h
typedef enum { LOGIC_CONSTANT, LOGIC_INPUT, LOGIC_NOT, LOGIC_AND, LOGIC_OR } LogicKind;

typedef struct logic Logic;

struct logic {
    LogicKind kind;
    union {
        int constant;                          // LOGIC_CONSTANT: 0 o 1
        char input;                            // LOGIC_INPUT: una letra de 'a' a 'z'
        Logic *child;                          // LOGIC_NOT
        struct { Logic *left; Logic *right; } pair;  // LOGIC_AND y LOGIC_OR
    } as;
};
```

- Una unión etiquetada, como el `Payment` de la semana 7, que se apunta a sí misma.

La fórmula lógica de la clase

logic.h

```c
typedef enum { LOGIC_CONSTANT, LOGIC_INPUT, LOGIC_NOT, LOGIC_AND, LOGIC_OR } LogicKind;

typedef struct logic Logic;

struct logic {
    LogicKind kind;
    union {
        int constant;                          // LOGIC_CONSTANT: 0 o 1
        char input;                            // LOGIC_INPUT: una letra de 'a' a 'z'
        Logic *child;                          // LOGIC_NOT
        struct { Logic *left; Logic *right; } pair;  // LOGIC_AND y LOGIC_OR
    } as;
};
```

- Una unión etiquetada, como el `Payment` de la semana 7, que se apunta a sí misma.
- El `typedef` va antes: `Logic *` ya se puede usar adentro.

La fórmula lógica de la clase

logic.h

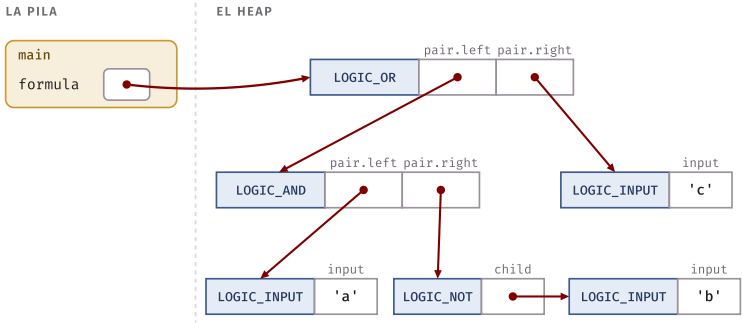
```c
typedef enum { LOGIC_CONSTANT, LOGIC_INPUT, LOGIC_NOT, LOGIC_AND, LOGIC_OR } LogicKind;

typedef struct logic Logic;

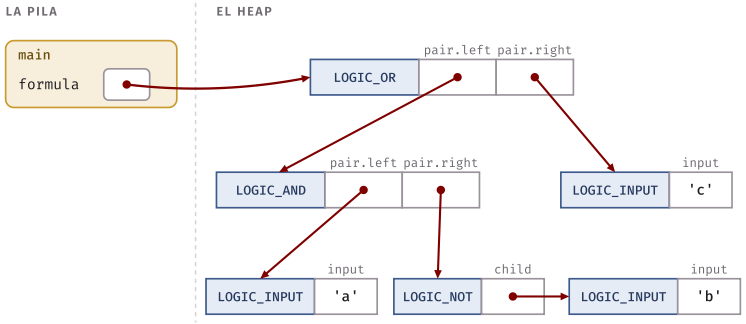
struct logic {
    LogicKind kind;
    union {
        int constant;                          // LOGIC_CONSTANT: 0 o 1
        char input;                            // LOGIC_INPUT: una letra de 'a' a 'z'
        Logic *child;                          // LOGIC_NOT
        struct { Logic *left; Logic *right; } pair;  // LOGIC_AND y LOGIC_OR
    } as;
};
```

- Una unión etiquetada, como el `Payment` de la semana 7, que se apunta a sí misma.
- El `typedef` va antes: `Logic *` ya se puede usar adentro.
- Una variante puede ser un `struct`: `e->as.pair.left`.

En la memoria: `(a and not b) or c`

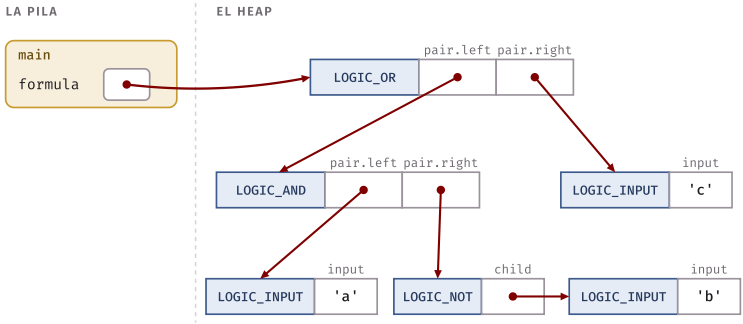


En la memoria: `(a and not b) or c`



- Cada rectángulo es un bloque del heap, con su propio `malloc`.

En la memoria: `(a and not b) or c`



- Cada rectángulo es un bloque del heap, con su propio `malloc`.
- Todos miden lo mismo: una `union` mide lo que su variante más grande.

El tipo del laboratorio: `Expr`

lab/expr.h

```c
typedef enum { EXPR_NUMBER, EXPR_VARIABLE, EXPR_BINARY } ExprKind;
typedef enum { OP_ADD, OP_SUB, OP_MUL, OP_DIV } BinaryOp;

typedef struct expr Expr;

struct expr {
    ExprKind kind;
    union {
        double number;                                          // EXPR_NUMBER
        char *name;                                             // EXPR_VARIABLE: el nodo es owner de su copia
        struct { BinaryOp op; Expr *left; Expr *right; } binary;   // EXPR_BINARY: el nodo es owner de sus hijos
    } as;
};
```

El tipo del laboratorio: `Expr`

lab/expr.h

```
typedef enum { EXPR_NUMBER, EXPR_VARIABLE, EXPR_BINARY } ExprKind;
typedef enum { OP_ADD, OP_SUB, OP_MUL, OP_DIV } BinaryOp;

typedef struct expr Expr;

struct expr {
    ExprKind kind;
    union {
        double number;                                          // EXPR_NUMBER
        char *name;                                             // EXPR_VARIABLE: el nodo es owner de su copia
        struct { BinaryOp op; Expr *left; Expr *right; } binary;   // EXPR_BINARY: el nodo es owner de sus hijos
    } as;
};
```

- Una sola variante para todas las operaciones, con un campo `op`.

El tipo del laboratorio: `Expr`

lab/expr.h

```c
typedef enum { EXPR_NUMBER, EXPR_VARIABLE, EXPR_BINARY } ExprKind;
typedef enum { OP_ADD, OP_SUB, OP_MUL, OP_DIV } BinaryOp;

typedef struct expr Expr;

struct expr {
    ExprKind kind;
    union {
        double number;                                          // EXPR_NUMBER
        char *name;                                             // EXPR_VARIABLE: el nodo es owner de su copia
        struct { BinaryOp op; Expr *left; Expr *right; } binary;   // EXPR_BINARY: el nodo es owner de sus hijos
    } as;
};
```

- Una sola variante para todas las operaciones, con un campo `op`.
- Una variable posee su nombre. **No es opaco**, a propósito: se usa con un `switch`.

SECCIÓN 03

Construir

03

Construir: la regla del ownership

logic.c

```c
Logic *logic_input(char letter) {
    Logic *e = logic_node(LOGIC_INPUT);
    if (e == NULL) {
        return NULL;
    }
    e->as.input = letter;
    return e;
}
```

Una hoja: un bloque nuevo, de quien llama.

Un `and` recibe dos subárboles que ya existen. ¿De quién son después?

Construir: la regla del ownership

logic.c

```c
Logic *logic_input(char letter) {
    Logic *e = logic_node(LOGIC_INPUT);
    if (e == NULL) {
        return NULL;
    }
    e->as.input = letter;
    return e;
}
```

Una hoja: un bloque nuevo, de quien llama.

Un `and` recibe dos subárboles que ya existen. ¿De quién son después?

- **Cada nodo es owner de sus hijos.**

Construir: la regla del ownership

logic.c

```c
Logic *logic_input(char letter) {
    Logic *e = logic_node(LOGIC_INPUT);
    if (e == NULL) {
        return NULL;
    }
    e->as.input = letter;
    return e;
}
```

Una hoja: un bloque nuevo, de quien llama.

Un `and` recibe dos subárboles que ya existen. ¿De quién son después?

- **Cada nodo es owner de sus hijos.**
- El constructor se queda con lo que recibe: es **mover**, de la semana 7.

Construir: la regla del ownership

logic.c

```c
Logic *logic_input(char letter) {
    Logic *e = logic_node(LOGIC_INPUT);
    if (e == NULL) {
        return NULL;
    }
    e->as.input = letter;
    return e;
}
```

Una hoja: un bloque nuevo, de quien llama.

Un `and` recibe dos subárboles que ya existen. ¿De quién son después?

- **Cada nodo es owner de sus hijos.**
- El constructor se queda con lo que recibe: es **mover**, de la semana 7.
- El árbol tiene **un solo owner**, por la raíz. Al liberarla, se libera todo.

Construir: la regla del ownership

logic.c

```c
Logic *logic_input(char letter) {
    Logic *e = logic_node(LOGIC_INPUT);
    if (e == NULL) {
        return NULL;
    }
    e->as.input = letter;
    return e;
}
```

Una hoja: un bloque nuevo, de quien llama.

Un `and` recibe dos subárboles que ya existen. ¿De quién son después?

- **Cada nodo es owner de sus hijos.**
- El constructor se queda con lo que recibe: es **mover**, de la semana 7.
- El árbol tiene **un solo owner**, por la raíz. Al liberarla, se libera todo.

Error común: seguir usando `a` después de `formula = logic_not(a)`. Ya no es de ustedes.

Se queda con los hijos, también cuando falla

logic.c

```c
static Logic *logic_pair(LogicKind kind, Logic *left, Logic *right) {
    if (left == NULL || right == NULL) {
        logic_free(left);
        logic_free(right);
        return NULL;
    }
    Logic *e = logic_node(kind);
    if (e == NULL) {
        logic_free(left);
        logic_free(right);
        return NULL;
    }
    e->as.pair.left = left;
    e->as.pair.right = right;
    return e;
}
```

Se queda con los hijos, también cuando falla

logic.c

```c
static Logic *logic_pair(LogicKind kind, Logic *left, Logic *right) {
    if (left == NULL || right == NULL) {
        logic_free(left);
        logic_free(right);
        return NULL;
    }
    Logic *e = logic_node(kind);
    if (e == NULL) {
        logic_free(left);
        logic_free(right);
        return NULL;
    }
    e->as.pair.left = left;
    e->as.pair.right = right;
    return e;
}
```

Si un hijo es `NULL`, algo falló antes. Libera al otro, y devuelve `NULL`: **el error sube solo**.

Por eso se pueden anidar

logic_demo.c

```
1  Logic *formula = logic_or(logic_and(logic_input('a'), logic_not(logic_input('b'))),
2                            logic_input('c'));
3  if (formula == NULL) {
```

Por eso se pueden anidar

logic_demo.c

```
1  Logic *formula = logic_or(logic_and(logic_input('a'), logic_not(logic_input('b'))),
2                            logic_input('c'));
3  if (formula == NULL) {
```

- La forma de las llamadas es la forma del árbol.

Por eso se pueden anidar

logic_demo.c

```
1  Logic *formula = logic_or(logic_and(logic_input('a'), logic_not(logic_input('b'))),
2                            logic_input('c'));
3  if (formula == NULL) {
```

- La forma de las llamadas es la forma del árbol.
- Se revisa solo el resultado final, y no queda nada reservado.

Por eso se pueden anidar

logic_demo.c

```
1  Logic *formula = logic_or(logic_and(logic_input('a'), logic_not(logic_input('b'))),
2                            logic_input('c'));
3  if (formula == NULL) {
```

- La forma de las llamadas es la forma del árbol.
- Se revisa solo el resultado final, y no queda nada reservado.
- C no fija el orden de los argumentos. Aquí no importa: cada uno crea su subárbol.

Por eso se pueden anidar

logic_demo.c

```
1  Logic *formula = logic_or(logic_and(logic_input('a'), logic_not(logic_input('b'))),
2                            logic_input('c'));
3  if (formula == NULL) {
```

- La forma de las llamadas es la forma del árbol.
- Se revisa solo el resultado final, y no queda nada reservado.
- C no fija el orden de los argumentos. Aquí no importa: cada uno crea su subárbol.
- `expr_binary` sigue esta regla. Las pruebas hacen fallar `malloc`, una reserva a la vez.

Por eso se pueden anidar

logic_demo.c

```c
Logic *formula = logic_or(logic_and(logic_input('a'), logic_not(logic_input('b'))),
                          logic_input('c'));
if (formula == NULL) {
```

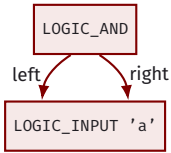
- La forma de las llamadas es la forma del árbol.
- Se revisa solo el resultado final, y no queda nada reservado.
- C no fija el orden de los argumentos. Aquí no importa: cada uno crea su subárbol.
- `expr_binary` sigue esta regla. Las pruebas hacen fallar `malloc`, una reserva a la vez.

> **En C**
>
> Con `-std=c17` no hay `strdup`: el nombre se copia con `malloc` y `strcpy`.

Un nodo, un padre

errors/shared_child.c

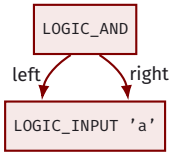
```
1  Logic *a = logic_input('a');
2  Logic *formula = logic_and(a, a);
```



Un nodo, un padre

errors/shared_child.c

```
1  Logic *a = logic_input('a');
2  Logic *formula = logic_and(a, a);
```



Dos padres son dos owners. `logic_free` libera la izquierda, y al ir a la derecha lee un nodo ya liberado:

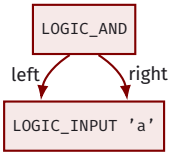
terminal

```
$ build/shared_child
==49==ERROR: AddressSanitizer: heap-use-after-free on address 0x503000000040
    #1 0xaaaad5674850 in logic_free logic.c:86
freed by thread T0 here:
    #2 0xaaaad56747d0 in logic_free logic.c:85
```

Un nodo, un padre

errors/shared_child.c

```
1  Logic *a = logic_input('a');
2  Logic *formula = logic_and(a, a);
```



Dos padres son dos owners. `logic_free` libera la izquierda, y al ir a la derecha lee un nodo ya liberado:

terminal

```
$ build/shared_child
==49==ERROR: AddressSanitizer: heap-use-after-free on address 0x503000000040
    #1 0xaaaad5674850 in logic_free logic.c:86
freed by thread T0 here:
    #2 0xaaaad56747d0 in logic_free logic.c:85
```

Si se necesita en dos lugares, se **copia**. Por eso el laboratorio pide `expr_copy` antes que la derivada.

SECCIÓN 04

La función sigue a los datos

04

Un caso por variante

logic.c

```c
int logic_count(const Logic *e) {
    switch (e->kind) {
    case LOGIC_CONSTANT:
    case LOGIC_INPUT:
        return 1;
    case LOGIC_NOT:
        return 1 + logic_count(e->as.child);
    case LOGIC_AND:
    case LOGIC_OR:
        return 1 + logic_count(e->as.pair.left) + logic_count(e->as.
      pair.right);
    }
    return 0;
}
```

Un caso por variante

logic.c

```c
int logic_count(const Logic *e) {
    switch (e->kind) {
    case LOGIC_CONSTANT:
    case LOGIC_INPUT:
        return 1;
    case LOGIC_NOT:
        return 1 + logic_count(e->as.child);
    case LOGIC_AND:
    case LOGIC_OR:
        return 1 + logic_count(e->as.pair.left) + logic_count(e->as.
      pair.right);
    }
    return 0;
}
```

1. Un `switch` sobre `kind`, **sin `default`**.

Un caso por variante

logic.c

```c
int logic_count(const Logic *e) {
    switch (e->kind) {
    case LOGIC_CONSTANT:
    case LOGIC_INPUT:
        return 1;
    case LOGIC_NOT:
        return 1 + logic_count(e->as.child);
    case LOGIC_AND:
    case LOGIC_OR:
        return 1 + logic_count(e->as.pair.left) + logic_count(e->as.
      pair.right);
    }
    return 0;
}
```

1. Un `switch` sobre `kind`, **sin `default`**.
2. Sin hijos, la respuesta es directa: caso base.

Un caso por variante

logic.c

```c
int logic_count(const Logic *e) {
    switch (e->kind) {
    case LOGIC_CONSTANT:
    case LOGIC_INPUT:
        return 1;
    case LOGIC_NOT:
        return 1 + logic_count(e->as.child);
    case LOGIC_AND:
    case LOGIC_OR:
        return 1 + logic_count(e->as.pair.left) + logic_count(e->as.
      pair.right);
    }
    return 0;
}
```

1. Un `switch` sobre `kind`, **sin `default`**.
2. Sin hijos, la respuesta es directa: caso base.
3. Con hijos, se llama sobre cada uno y se combina.

Un caso por variante

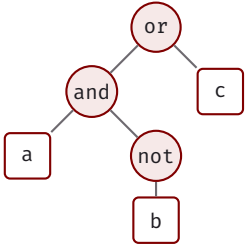
logic.c

```c
int logic_count(const Logic *e) {
    switch (e->kind) {
    case LOGIC_CONSTANT:
    case LOGIC_INPUT:
        return 1;
    case LOGIC_NOT:
        return 1 + logic_count(e->as.child);
    case LOGIC_AND:
    case LOGIC_OR:
        return 1 + logic_count(e->as.pair.left) + logic_count(e->as.
      pair.right);
    }
    return 0;
}
```

1. Un `switch` sobre `kind`, **sin `default`**.
2. Sin hijos, la respuesta es directa: caso base.
3. Con hijos, se llama sobre cada uno y se combina.

Idea clave: no piensen en el recorrido. Supongan que la función ya funciona para los hijos.

Comprobémoslo a mano



terminal

```
$ build/logic_demo
or(and(a, not(b)), c)
nodos: 6
profundidad: 4
a b c | resultado
0 0 0 | 0
0 0 1 | 1
0 1 0 | 0
0 1 1 | 1
1 0 0 | 1
1 0 1 | 1
1 1 0 | 0
1 1 1 | 1
```

Por qué es correcta, y por qué termina

Las dos preguntas de la semana 9:

Por qué es correcta, y por qué termina

Las dos preguntas de la semana 9:

- **¿Es correcta?** Suponemos que da bien para los hijos. Entonces el conteo de un `and` es 1 más el de sus subárboles.

Por qué es correcta, y por qué termina

Las dos preguntas de la semana 9:

- **¿Es correcta?** Suponemos que da bien para los hijos. Entonces el conteo de un `and` es 1 más el de sus subárboles.
- **¿Termina?** Baja un entero: **la cantidad de nodos del subárbol**. Un hijo tiene menos que su padre, y una hoja no llama a nadie.

Por qué es correcta, y por qué termina

Las dos preguntas de la semana 9:

- **¿Es correcta?** Suponemos que da bien para los hijos. Entonces el conteo de un `and` es 1 más el de sus subárboles.
- **¿Termina?** Baja un entero: **la cantidad de nodos del subárbol**. Un hijo tiene menos que su padre, y una hoja no llama a nadie.
- **¿Cuánto cuesta?** Cada nodo se visita una vez: tiempo proporcional a los nodos. Y hay tantos marcos como la profundidad.

Por qué es correcta, y por qué termina

Las dos preguntas de la semana 9:

- **¿Es correcta?** Suponemos que da bien para los hijos. Entonces el conteo de un `and` es 1 más el de sus subárboles.
- **¿Termina?** Baja un entero: **la cantidad de nodos del subárbol**. Un hijo tiene menos que su padre, y una hoja no llama a nadie.
- **¿Cuánto cuesta?** Cada nodo se visita una vez: tiempo proporcional a los nodos. Y hay tantos marcos como la profundidad.

Lo que eso supone

Que el árbol es un árbol: un padre por nodo, sin ciclos. Otra razón para construirlos solo con constructores.

`-Wall` revisa los casos

terminal

```
$ gcc -Wall -Wextra -std=c17 -I. errors/missing_case.c logic.c -o build/missing_case
errors/missing_case.c:7:5: warning: enumeration value 'LOGIC_OR' not handled in switch [-Wswitch]
```

`-Wall` revisa los casos

terminal

```
$ gcc -Wall -Wextra -std=c17 -I. errors/missing_case.c logic.c -o build/missing_case
errors/missing_case.c:7:5: warning: enumeration value 'LOGIC_OR' not handled in switch [-Wswitch]
```

- Sin `default`, `gcc` dice qué `switch` quedó incompleto.

`-Wall` revisa los casos

terminal

```
$ gcc -Wall -Wextra -std=c17 -I. errors/missing_case.c logic.c -o build/missing_case
errors/missing_case.c:7:5: warning: enumeration value 'LOGIC_OR' not handled in switch [-Wswitch]
```

- Sin `default`, `gcc` dice qué `switch` quedó incompleto.
- En la semana 15, `Expr` ganará dos variantes. Este aviso les dirá cada función que hay que cambiar.

`-Wall` revisa los casos

terminal

```
$ gcc -Wall -Wextra -std=c17 -I. errors/missing_case.c logic.c -o build/missing_case
errors/missing_case.c:7:5: warning: enumeration value 'LOGIC_OR' not handled in switch [-Wswitch]
```

- Sin `default`, `gcc` dice qué `switch` quedó incompleto.
- En la semana 15, `Expr` ganará dos variantes. Este aviso les dirá cada función que hay que cambiar.
- El `return 0` del final de cada función no corre con datos bien formados. Está porque `gcc` no lo sabe.

SECCIÓN 05

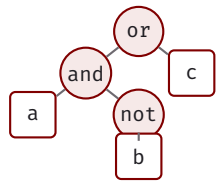
Liberar

05

Primero los hijos, después el nodo

logic.c

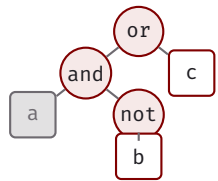
```c
void logic_free(Logic *e) {
    if (e == NULL) {
        return;
    }
    switch (e->kind) {
    case LOGIC_CONSTANT:
    case LOGIC_INPUT:
        break;
    case LOGIC_NOT:
        logic_free(e->as.child);
        break;
    case LOGIC_AND:
    case LOGIC_OR:
        logic_free(e->as.pair.left);
        logic_free(e->as.pair.right);
        break;
    }
    free(e);
}
```



Primero los hijos, después el nodo

logic.c

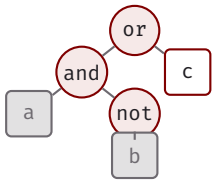
```c
void logic_free(Logic *e) {
    if (e == NULL) {
        return;
    }
    switch (e->kind) {
    case LOGIC_CONSTANT:
    case LOGIC_INPUT:
        break;
    case LOGIC_NOT:
        logic_free(e->as.child);
        break;
    case LOGIC_AND:
    case LOGIC_OR:
        logic_free(e->as.pair.left);
        logic_free(e->as.pair.right);
        break;
    }
    free(e);
}
```



Primero los hijos, después el nodo

logic.c

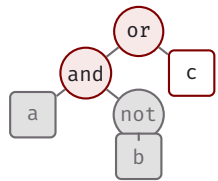
```c
void logic_free(Logic *e) {
    if (e == NULL) {
        return;
    }
    switch (e->kind) {
    case LOGIC_CONSTANT:
    case LOGIC_INPUT:
        break;
    case LOGIC_NOT:
        logic_free(e->as.child);
        break;
    case LOGIC_AND:
    case LOGIC_OR:
        logic_free(e->as.pair.left);
        logic_free(e->as.pair.right);
        break;
    }
    free(e);
}
```



Primero los hijos, después el nodo

logic.c

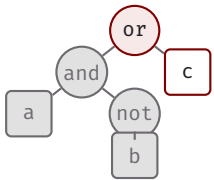
```c
void logic_free(Logic *e) {
    if (e == NULL) {
        return;
    }
    switch (e->kind) {
    case LOGIC_CONSTANT:
    case LOGIC_INPUT:
        break;
    case LOGIC_NOT:
        logic_free(e->as.child);
        break;
    case LOGIC_AND:
    case LOGIC_OR:
        logic_free(e->as.pair.left);
        logic_free(e->as.pair.right);
        break;
    }
    free(e);
}
```



Primero los hijos, después el nodo

logic.c

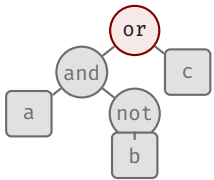
```c
void logic_free(Logic *e) {
    if (e == NULL) {
        return;
    }
    switch (e->kind) {
    case LOGIC_CONSTANT:
    case LOGIC_INPUT:
        break;
    case LOGIC_NOT:
        logic_free(e->as.child);
        break;
    case LOGIC_AND:
    case LOGIC_OR:
        logic_free(e->as.pair.left);
        logic_free(e->as.pair.right);
        break;
    }
    free(e);
}
```



Primero los hijos, después el nodo

logic.c

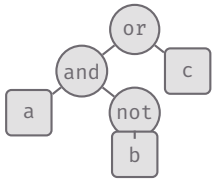
```c
void logic_free(Logic *e) {
    if (e == NULL) {
        return;
    }
    switch (e->kind) {
    case LOGIC_CONSTANT:
    case LOGIC_INPUT:
        break;
    case LOGIC_NOT:
        logic_free(e->as.child);
        break;
    case LOGIC_AND:
    case LOGIC_OR:
        logic_free(e->as.pair.left);
        logic_free(e->as.pair.right);
        break;
    }
    free(e);
}
```



Primero los hijos, después el nodo

logic.c

```c
void logic_free(Logic *e) {
    if (e == NULL) {
        return;
    }
    switch (e->kind) {
    case LOGIC_CONSTANT:
    case LOGIC_INPUT:
        break;
    case LOGIC_NOT:
        logic_free(e->as.child);
        break;
    case LOGIC_AND:
    case LOGIC_OR:
        logic_free(e->as.pair.left);
        logic_free(e->as.pair.right);
        break;
    }
    free(e);
}
```



el orden real

```
$ build/logic_free_order
libero a
libero b
libero un not
libero un and
libero c
libero un or
```

Al revés: leer lo que ya se liberó

errors/free_parent_first.c

```
1  free(e);
2  if (e->kind == LOGIC_AND) {
```

terminal

```
$ build/free_parent_first
==24==ERROR: AddressSanitizer: heap-use-after-free on address 0x503000000a0
READ of size 4 at 0x503000000a0 thread T0
    #0 0xaaaab2c74344 in bad_free errors/free_parent_first.c:11
...
freed by thread T0 here:
...
    #1 0xaaaab2c742c4 in bad_free errors/free_parent_first.c:10
```

Al revés: leer lo que ya se liberó

errors/free_parent_first.c

```
1  free(e);
2  if (e->kind == LOGIC_AND) {
```

terminal

```
$ build/free_parent_first
==24==ERROR: AddressSanitizer: heap-use-after-free on address 0x503000000a0
READ of size 4 at 0x503000000a0 thread T0
    #0 0xaaaab2c74344 in bad_free errors/free_parent_first.c:11
...
freed by thread T0 here:
...
    #1 0xaaaab2c742c4 in bad_free errors/free_parent_first.c:10
```

- Las direcciones de los hijos están **adentro** del nodo.

Al revés: leer lo que ya se liberó

errors/free_parent_first.c

```
1  free(e);
2  if (e->kind == LOGIC_AND) {
```

terminal

```
$ build/free_parent_first
==24==ERROR: AddressSanitizer: heap-use-after-free on address 0x503000000a0
READ of size 4 at 0x503000000a0 thread T0
    #0 0xaaaab2c74344 in bad_free errors/free_parent_first.c:11
...
freed by thread T0 here:
...
    #1 0xaaaab2c742c4 in bad_free errors/free_parent_first.c:10
```

- Las direcciones de los hijos están **adentro** del nodo.
- `gcc` avisó, con `-Wuse-after-free`, porque las líneas están pegadas. No cuenten con eso.

Solo la raíz: una fuga

errors/free_only_root.c

```
1  Logic *formula = logic_and(logic_input('a'), logic_not(logic_input('b')));
2  free(formula);
```

terminal

```
$ build/free_only_root
==35==ERROR: LeakSanitizer: detected memory leaks
...
SUMMARY: AddressSanitizer: 72 byte(s) leaked in 3 allocation(s).
```

Solo la raíz: una fuga

errors/free_only_root.c

```
1  Logic *formula = logic_and(logic_input('a'), logic_not(logic_input('b')));
2  free(formula);
```

terminal

```
$ build/free_only_root
==35==ERROR: LeakSanitizer: detected memory leaks
...
SUMMARY: AddressSanitizer: 72 byte(s) leaked in 3 allocation(s).
```

- El árbol tiene cuatro nodos, y se liberó uno. Quedan tres, de 24 bytes en este entorno.

Solo la raíz: una fuga

errors/free_only_root.c

```
1  Logic *formula = logic_and(logic_input('a'), logic_not(logic_input('b')));
2  free(formula);
```

terminal

```
$ build/free_only_root
==35==ERROR: LeakSanitizer: detected memory leaks
...
SUMMARY: AddressSanitizer: 72 byte(s) leaked in 3 allocation(s).
```

- El árbol tiene cuatro nodos, y se liberó uno. Quedan tres, de 24 bytes en este entorno.
- *Direct leak*: ya nadie apunta al bloque. *Indirect leak*: solo le apunta otro bloque perdido.

SECCIÓN 06

Escribir

06

Solo los paréntesis necesarios: precedencia

`logic_print` escribe llamadas: `or(and(a, not(b)), c)`. Es claro, y feo para `2 + 3 * x`.

Con paréntesis en todo también funciona: `(2 + (3 * x))`. Queremos **solo los necesarios**.

Padre	Hijo izq.	Sin paréntesis	¿Hacen falta?
`*`	`2 + 3`	`2 + 3 * x`: otro árbol	Sí: `(2 + 3) * x`
`+`	`2 * 3`	`2 * 3 + x`: el mismo	No

Solo los paréntesis necesarios: precedencia

`logic_print` escribe llamadas: `or(and(a, not(b)), c)`. Es claro, y feo para `2 + 3 * x`.

Con paréntesis en todo también funciona: `(2 + (3 * x))`. Queremos **solo los necesarios**.

Padre	Hijo izq.	Sin paréntesis	¿Hacen falta?
`*`	`2 + 3`	`2 + 3 * x`: otro árbol	Sí: `(2 + 3) * x`
`+`	`2 * 3`	`2 * 3 + x`: el mismo	No

Primera regla

Un hijo con un operador más débil que el de su padre lleva paréntesis.

Con la misma precedencia, importa el lado

Árbol	Se escribe	Vale
resta de (1 - 2) y 3	`1 - 2 - 3`	-4
resta de 1 y (2 - 3)	`1 - (2 - 3)`	2

Con la misma precedencia, importa el lado

Árbol	Se escribe	Vale
resta de (1 - 2) y 3	`1 - 2 - 3`	-4
resta de 1 y (2 - 3)	`1 - (2 - 3)`	2

- Asocian **por la izquierda**: `1 - 2 - 3` es `(1 - 2) - 3`.

Con la misma precedencia, importa el lado

Árbol	Se escribe	Vale
resta de (1 - 2) y 3	`1 - 2 - 3`	-4
resta de 1 y (2 - 3)	`1 - (2 - 3)`	2

- Asocian **por la izquierda**: `1 - 2 - 3` es `(1 - 2) - 3`.
- Hijo **izquierdo**: paréntesis si su precedencia es **menor**.

Con la misma precedencia, importa el lado

Árbol	Se escribe	Vale
resta de (1 - 2) y 3	`1 - 2 - 3`	-4
resta de 1 y (2 - 3)	`1 - (2 - 3)`	2

- Asocian **por la izquierda**: `1 - 2 - 3` es `(1 - 2) - 3`.
- Hijo **izquierdo**: paréntesis si su precedencia es **menor**.
- Hijo **derecho**: paréntesis si es **menor o igual**.

Con la misma precedencia, importa el lado

Árbol	Se escribe	Vale
resta de (1 - 2) y 3	`1 - 2 - 3`	-4
resta de 1 y (2 - 3)	`1 - (2 - 3)`	2

- Asocian **por la izquierda**: `1 - 2 - 3` es `(1 - 2) - 3`.
- Hijo **izquierdo**: paréntesis si su precedencia es **menor**.
- Hijo **derecho**: paréntesis si es **menor o igual**.
- Por eso `1 + (2 + 3)` los conserva. Como matemática sobran, pero conservan **la forma**: `1 + 2 + 3` es otro árbol.

Con la misma precedencia, importa el lado

Árbol	Se escribe	Vale
resta de (1 - 2) y 3	`1 - 2 - 3`	-4
resta de 1 y (2 - 3)	`1 - (2 - 3)`	2

- Asocian **por la izquierda**: `1 - 2 - 3` es `(1 - 2) - 3`.
- Hijo **izquierdo**: paréntesis si su precedencia es **menor**.
- Hijo **derecho**: paréntesis si es **menor o igual**.
- Por eso `1 + (2 + 3)` los conserva. Como matemática sobran, pero conservan **la forma**: `1 + 2 + 3` es otro árbol.

Error común

Mirar solo al padre o solo al hijo. La pregunta es: ¿este hijo, en este lado, bajo este padre?

SECCIÓN 07

Evaluar

07

Evaluar necesita un entorno

lab/env.h

```
1  Env *env_new(Env *parent);
2  void env_free(Env *env);
3  int env_set(Env *env, const char *name, double
       value);
4  int env_get(const Env *env, const char *name,
       double *value);
```

terminal

```
$ build/env_demo
x vale 10
y vale 2.5
z no existe, y value sigue en 2.5
```

Evaluar necesita un entorno

lab/env.h

```
1  Env *env_new(Env *parent);
2  void env_free(Env *env);
3  int env_set(Env *env, const char *name, double
       value);
4  int env_get(const Env *env, const char *name,
       double *value);
```

terminal

```
$ build/env_demo
x vale 10
y vale 2.5
z no existe, y value sigue en 2.5
```

- Las variables tienen nombres: hace falta una tabla de nombres y valores.

Evaluar necesita un entorno

lab/env.h

```c
Env *env_new(Env *parent);
void env_free(Env *env);
int env_set(Env *env, const char *name, double
      value);
int env_get(const Env *env, const char *name,
      double *value);
```

terminal

```
$ build/env_demo
x vale 10
y vale 2.5
z no existe, y value sigue en 2.5
```

- Las variables tienen nombres: hace falta una tabla de nombres y valores.
- Es lo que dibujan en cada marco desde la semana 3.

Evaluar necesita un entorno

lab/env.h

```c
Env *env_new(Env *parent);
void env_free(Env *env);
int env_set(Env *env, const char *name, double
      value);
int env_get(const Env *env, const char *name,
      double *value);
```

terminal

```
$ build/env_demo
x vale 10
y vale 2.5
z no existe, y value sigue en 2.5
```

- Las variables tienen nombres: hace falta una tabla de nombres y valores.
- Es lo que dibujan en cada marco desde la semana 3.
- **Se les da hecha**. Se usa solo por su encabezado.

Evaluar necesita un entorno

lab/env.h

```c
Env *env_new(Env *parent);
void env_free(Env *env);
int env_set(Env *env, const char *name, double
      value);
int env_get(const Env *env, const char *name,
      double *value);
```

terminal

```
$ build/env_demo
x vale 10
y vale 2.5
z no existe, y value sigue en 2.5
```

- Las variables tienen nombres: hace falta una tabla de nombres y valores.
- Es lo que dibujan en cada marco desde la semana 3.
- **Se les da hecha**. Se usa solo por su encabezado.
- `parent` va en `NULL`. Ya le tocará en la semana 15.

Evaluar puede fallar. ¿Cómo se avisa?

Diseño	Problema

Evaluar puede fallar. ¿Cómo se avisa?

Diseño	Problema
Devolver un `double` especial, como -1	Todo `double` es un resultado válido

Evaluar puede fallar. ¿Cómo se avisa?

Diseño	Problema
Devolver un `double` especial, como -1	Todo `double` es un resultado válido
Terminar con `exit`	Quien usa el módulo no puede decidir qué hacer

Evaluar puede fallar. ¿Cómo se avisa?

Diseño	Problema
Devolver un `double` especial, como -1	Todo `double` es un resultado válido
Terminar con `exit`	Quien usa el módulo no puede decidir qué hacer
Dejar que la división dé infinito	Oculta el error

Evaluar puede fallar. ¿Cómo se avisa?

Diseño	Problema
Devolver un `double` especial, como -1	Todo `double` es un resultado válido
Terminar con `exit`	Quien usa el módulo no puede decidir qué hacer
Dejar que la división dé infinito	Oculta el error
Un `int` de éxito, y el valor por un puntero	Quien llama está obligado a mirar. Como `env_get`

lab/expr.h

```
1  int expr_eval(const Expr *e, const Env *env, double *result);
```

La forma de `expr_eval`

pseudocódigo

```
algoritmo evaluar(e, entorno)
    si e es un número entonces
        devolver éxito con su valor
    sino si e es una variable entonces
        devolver lo que diga buscar(entorno, nombre de e)
    sino
        izquierda ← evaluar(hijo izquierdo de e, entorno)
        si izquierda falló entonces
            devolver fracaso
        fin si
        derecha ← evaluar(hijo derecho de e, entorno)
        si derecha falló entonces
            devolver fracaso
        fin si
        si el operador es la división y derecha = 0 entonces
            devolver fracaso
        fin si
        devolver éxito con la operación de izquierda y derecha
    fin si
fin algoritmo
```

La forma de `expr_eval`

pseudocódigo

```
algoritmo evaluar(e, entorno)
    si e es un número entonces
        devolver éxito con su valor
    sino si e es una variable entonces
        devolver lo que diga buscar(entorno, nombre de e)
    sino
        izquierda ← evaluar(hijo izquierdo de e, entorno)
        si izquierda falló entonces
            devolver fracaso
        fin si
        derecha ← evaluar(hijo derecho de e, entorno)
        si derecha falló entonces
            devolver fracaso
        fin si
        si el operador es la división y derecha = 0 entonces
            devolver fracaso
        fin si
        devolver éxito con la operación de izquierda y derecha
    fin si
fin algoritmo
```

- **El error sube solo**: cada padre ve el fracaso y devuelve fracaso.

La forma de `expr_eval`

pseudocódigo

```
algoritmo evaluar(e, entorno)
    si e es un número entonces
        devolver éxito con su valor
    sino si e es una variable entonces
        devolver lo que diga buscar(entorno, nombre de e)
    sino
        izquierda ← evaluar(hijo izquierdo de e, entorno)
        si izquierda falló entonces
            devolver fracaso
        fin si
        derecha ← evaluar(hijo derecho de e, entorno)
        si derecha falló entonces
            devolver fracaso
        fin si
        si el operador es la división y derecha = 0 entonces
            devolver fracaso
        fin si
        devolver éxito con la operación de izquierda y derecha
    fin si
fin algoritmo
```

- **El error sube solo**: cada padre ve el fracaso y devuelve fracaso.
- Es la misma recursión. No hace falta nada especial.

La forma de `expr_eval`

pseudocódigo

```
algoritmo evaluar(e, entorno)
    si e es un número entonces
        devolver éxito con su valor
    sino si e es una variable entonces
        devolver lo que diga buscar(entorno, nombre de e)
    sino
        izquierda ← evaluar(hijo izquierdo de e, entorno)
        si izquierda falló entonces
            devolver fracaso
        fin si
        derecha ← evaluar(hijo derecho de e, entorno)
        si derecha falló entonces
            devolver fracaso
        fin si
        si el operador es la división y derecha = 0 entonces
            devolver fracaso
        fin si
        devolver éxito con la operación de izquierda y derecha
    fin si
fin algoritmo
```

- **El error sube solo**: cada padre ve el fracaso y devuelve fracaso.
- Es la misma recursión. No hace falta nada especial.

Error común

Pasar `result` a las dos llamadas: la derecha pisaría a la izquierda.

Copiar y comparar: la misma receta

`expr_copy`

`expr_equal`

Copiar y comparar: la misma receta

`expr_copy`

- Un árbol nuevo, sin nada compartido: ni nodos ni nombres.

`expr_equal`

Copiar y comparar: la misma receta

`expr_copy`

- Un árbol nuevo, sin nada compartido: ni nodos ni nombres.
- Es la *copia de verdad* de la semana 7.

`expr_equal`

Copiar y comparar: la misma receta

`expr_copy`

- Un árbol nuevo, sin nada compartido: ni nodos ni nombres.
- Es la *copia de verdad* de la semana 7.
- Si falla una copia, ¿qué hace `expr_binary` con un hijo `NULL`?

`expr_equal`

Copiar y comparar: la misma receta

`expr_copy`

- Un árbol nuevo, sin nada compartido: ni nodos ni nombres.
- Es la *copia de verdad* de la semana 7.
- Si falla una copia, ¿qué hace `expr_binary` con un hijo `NULL`?

`expr_equal`

- Igual **forma y contenido**, no igual valor.

Copiar y comparar: la misma receta

`expr_copy`

- Un árbol nuevo, sin nada compartido: ni nodos ni nombres.
- Es la *copia de verdad* de la semana 7.
- Si falla una copia, ¿qué hace `expr_binary` con un hijo `NULL`?

`expr_equal`

- Igual **forma y contenido**, no igual valor.
- `1 + 2` y `2 + 1` no son iguales.

Copiar y comparar: la misma receta

`expr_copy`

- Un árbol nuevo, sin nada compartido: ni nodos ni nombres.
- Es la *copia de verdad* de la semana 7.
- Si falla una copia, ¿qué hace `expr_binary` con un hijo `NULL`?

`expr_equal`

- Igual **forma y contenido**, no igual valor.
- `1 + 2` y `2 + 1` no son iguales.
- `a == b` dice si son *el mismo* árbol. Y los nombres se comparan con `strcmp`.

Copiar y comparar: la misma receta

`expr_copy`

- Un árbol nuevo, sin nada compartido: ni nodos ni nombres.
- Es la *copia de verdad* de la semana 7.
- Si falla una copia, ¿qué hace `expr_binary` con un hijo `NULL`?

`expr_equal`

- Igual **forma y contenido**, no igual valor.
- `1 + 2` y `2 + 1` no son iguales.
- `a == b` dice si son *el mismo* árbol. Y los nombres se comparan con `strcmp`.

En C

Aquí sí se comparan `double` con `==`: no son resultados de un cálculo, sino valores guardados.

SECCIÓN 08

Transformar

08

Simplificar: un contrato de entrega

logic.h y logic_simplify_demo.c

```
1  Logic *logic_simplify(Logic *e);
2  e = logic_simplify(e);
```

Simplificar: un contrato de entrega

logic.h y logic_simplify_demo.c

```
1  Logic *logic_simplify(Logic *e);
2  e = logic_simplify(e);
```

- Sin `const`: quien llama **entrega** el árbol, y recibe otro.

Simplificar: un contrato de entrega

logic.h y logic_simplify_demo.c

```
1  Logic *logic_simplify(Logic *e);
2  e = logic_simplify(e);
```

- Sin `const`: quien llama **entrega** el árbol, y recibe otro.
- `a and true` vale lo mismo que `a`. `a and false` siempre es falso.

Simplificar: un contrato de entrega

logic.h y logic_simplify_demo.c

```
1  Logic *logic_simplify(Logic *e);
2  e = logic_simplify(e);
```

- Sin `const`: quien llama **entrega** el árbol, y recibe otro.
- `a and true` vale lo mismo que `a`. `a and false` siempre es falso.
- **Primero los hijos**: las reglas se aplican en cadena. En `or(and(true, a), not(true))`, el `or` recién ve `false` después de simplificar su hijo.

Simplificar: un contrato de entrega

logic.h y logic_simplify_demo.c

```
1  Logic *logic_simplify(Logic *e);
2  e = logic_simplify(e);
```

- Sin `const`: quien llama **entrega** el árbol, y recibe otro.
- `a and true` vale lo mismo que `a`. `a and false` siempre es falso.
- **Primero los hijos**: las reglas se aplican en cadena. En `or(and(true, a), not(true))`, el `or` recién ve `false` después de simplificar su hijo.

Error común

`logic_simplify(e);` sin guardar el resultado. La raíz pudo cambiar, o quedar liberada.

Las reglas de `and` y `or`

logic.c

```c
static Logic *simplify_pair(Logic *e) {
    e->as.pair.left = logic_simplify(e->as.pair.left);
    e->as.pair.right = logic_simplify(e->as.pair.right);
    Logic *left = e->as.pair.left;
    Logic *right = e->as.pair.right;
    // En AND, true no cambia nada y false decide. En OR, al revés.
    int neutral = e->kind == LOGIC_AND;
    if (is_constant(left, neutral)) {
        return keep_child(e, right, left);
    }
    if (is_constant(right, neutral)) {
        return keep_child(e, left, right);
    }
    ...
    return e;
}
```

Las reglas de and y or

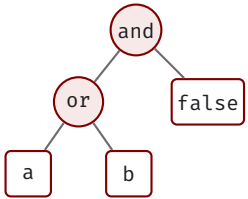
logic.c

```c
static Logic *simplify_pair(Logic *e) {
    e->as.pair.left = logic_simplify(e->as.pair.left);
    e->as.pair.right = logic_simplify(e->as.pair.right);
    Logic *left = e->as.pair.left;
    Logic *right = e->as.pair.right;
    // En AND, true no cambia nada y false decide. En OR, al revés.
    int neutral = e->kind == LOGIC_AND;
    if (is_constant(left, neutral)) {
        return keep_child(e, right, left);
    }
    if (is_constant(right, neutral)) {
        return keep_child(e, left, right);
    }
    ...
    return e;
}
```

Se guarda lo que devuelve cada llamada: un hijo puede cambiar.

Tres destinos, tres herramientas

`and(or(a, b), false)`



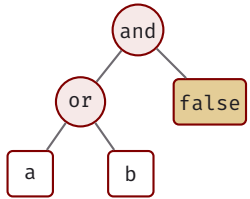
Nodo	Con qué

logic.c

```c
static Logic *keep_child(Logic *node,
       Logic *kept, Logic *dropped) {
    logic_free(dropped);
    free(node);
    return kept;
}
```

Tres destinos, tres herramientas

`and(or(a, b), false)`



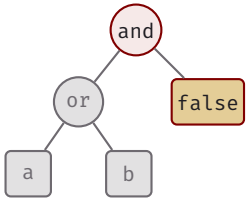
Nodo	Con qué
El que se conserva	`return`

logic.c

```c
static Logic *keep_child(Logic *node,
       Logic *kept, Logic *dropped) {
    logic_free(dropped);
    free(node);
    return kept;
}
```

Tres destinos, tres herramientas

`and(or(a, b), false)`



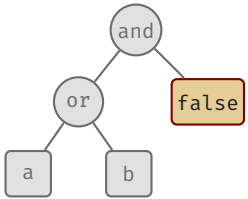
logic.c

```c
static Logic *keep_child(Logic *node,
       Logic *kept, Logic *dropped) {
    logic_free(dropped);
    free(node);
    return kept;
}
```

Nodo	Con qué
El que se conserva	`return`
El subárbol que se descarta, entero	`logic_free`

Tres destinos, tres herramientas

`and(or(a, b), false)`



logic.c

```c
static Logic *keep_child(Logic *node,
       Logic *kept, Logic *dropped) {
    logic_free(dropped);
    free(node);
    return kept;
}
```

Nodo	Con qué
El que se conserva	`return`
El subárbol que se descarta, entero	`logic_free`
El nodo que desaparece, solo él	`free`

Se puede fallar en las dos direcciones

Olvidar el subárbol descartado

errors/simplify_leak.c

```
1  Logic *right = e->as.pair.right;
2  free(e);
3  return right;
```

terminal

```
$ build/simplify_leak
==69==ERROR: LeakSanitizer: detected memory leaks
SUMMARY: ... 72 byte(s) leaked in 3 allocation(s).
```

Liberar el hijo que se conservaba

errors/simplify_dangling.c

```
1  Logic *left = e->as.pair.left;
2  logic_free(e);
3  return left;
```

terminal

```
$ build/simplify_dangling
==81==ERROR: AddressSanitizer: heap-use-after-free
freed by thread T0 here:
    #3 ... in bad_simplify errors/simplify_dangling.c:10
```

Se puede fallar en las dos direcciones

Olvidar el subárbol descartado

errors/simplify_leak.c

```
1  Logic *right = e->as.pair.right;
2  free(e);
3  return right;
```

terminal

```
$ build/simplify_leak
==69==ERROR: LeakSanitizer: detected memory leaks
SUMMARY: ... 72 byte(s) leaked in 3 allocation(s).
```

Liberar el hijo que se conservaba

errors/simplify_dangling.c

```
1  Logic *left = e->as.pair.left;
2  logic_free(e);
3  return left;
```

terminal

```
$ build/simplify_dangling
==81==ERROR: AddressSanitizer: heap-use-after-free
freed by thread T0 here:
    #3 ... in bad_simplify errors/simplify_dangling.c:10
```

Si `x * 0` no libera el subárbol de `x`, nadie apunta a él. Por grande que sea, se pierde.

La versión correcta

terminal

```
$ build/logic_simplify_demo
and(a, true)  ->  a  (nodos: de 3 a 1)
and(or(a, b), false)  ->  false  (nodos: de 5 a 1)
not(not(a))  ->  a  (nodos: de 3 a 1)
or(and(true, a), not(true))  ->  a  (nodos: de 6 a 1)
and(a, or(b, c))  ->  and(a, or(b, c))  (nodos: de 5 a 5)
```

La versión correcta

terminal

```
$ build/logic_simplify_demo
and(a, true)  ->  a  (nodos: de 3 a 1)
and(or(a, b), false)  ->  false  (nodos: de 5 a 1)
not(not(a))  ->  a  (nodos: de 3 a 1)
or(and(true, a), not(true))  ->  a  (nodos: de 6 a 1)
and(a, or(b, c))  ->  and(a, or(b, c))  (nodos: de 5 a 5)
```

- La cuarta se aplica en cadena: primero los hijos, luego el `or`.

La versión correcta

terminal

```
$ build/logic_simplify_demo
and(a, true)  ->  a  (nodos: de 3 a 1)
and(or(a, b), false)  ->  false  (nodos: de 5 a 1)
not(not(a))  ->  a  (nodos: de 3 a 1)
or(and(true, a), not(true))  ->  a  (nodos: de 6 a 1)
and(a, or(b, c))  ->  and(a, or(b, c))  (nodos: de 5 a 5)
```

- La cuarta se aplica en cadena: primero los hijos, luego el `or`.
- La última no cambia, y es un resultado válido: `e` se devuelve tal cual.

Las reglas de `expr_simplify`

Si el nodo es	Queda
`X + 0`, `0 + X`, `X - 0`, `X * 1`, `1 * X`, `X / 1`	`X`
`X * 0`, `0 * X`	`0`
Una operación entre dos números	El resultado

Las reglas de `expr_simplify`

Si el nodo es	Queda
`x + 0`, `0 + x`, `x - 0`, `x * 1`, `1 * x`, `x / 1`	`x`
`x * 0`, `0 * x`	`0`
Una operación entre dos números	El resultado

- `1 / 0` no se simplifica: el error le toca a quien evalúe. Tampoco `0 - x` ni `0 / x`.

Las reglas de `expr_simplify`

Si el nodo es	Queda
`x + 0`, `0 + x`, `x - 0`, `x * 1`, `1 * x`, `x / 1`	`x`
`x * 0`, `0 * x`	`0`
Una operación entre dos números	El resultado

- `1 / 0` no se simplifica: el error le toca a quien evalúe. Tampoco `0 - x` ni `0 / x`.
- Revisen `x * 0` antes de operar constantes, o `-3 * 0` se escribe `-0`.

Las reglas de `expr_simplify`

Si el nodo es	Queda
`x + 0`, `0 + x`, `x - 0`, `x * 1`, `1 * x`, `x / 1`	`x`
`x * 0`, `0 * x`	`0`
Una operación entre dos números	El resultado

- `1 / 0` no se simplifica: el error le toca a quien evalúe. Tampoco `0 - x` ni `0 / x`.
- Revisen `x * 0` antes de operar constantes, o `-3 * 0` se escribe `-0`.
- Un nodo puede cambiar de variante, sin `malloc`: se liberan los hijos, y se cambia `kind`.

Las reglas de `expr_simplify`

Si el nodo es	Queda
`x + 0`, `0 + x`, `x - 0`, `x * 1`, `1 * x`, `x / 1`	`x`
`x * 0`, `0 * x`	`0`
Una operación entre dos números	El resultado

- `1 / 0` no se simplifica: el error le toca a quien evalúe. Tampoco `0 - x` ni `0 / x`.
- Revisen `x * 0` antes de operar constantes, o `-3 * 0` se escribe `-0`.
- Un nodo puede cambiar de variante, sin `malloc`: se liberan los hijos, y se cambia `kind`.

No es del todo inocente: si `x` no existe, o es `1 / 0`, evaluar `x * 0` daba error. Simplificado, da 0.

Derivar: un recorrido que construye

La expresión	Su derivada
Un número	`0`
La variable pedida	`1`
Otra variable	`0`
`f + g`	`f' + g'`
`f - g`	`f' - g'`
`f * g`	`f' * g + f * g'`
`f / g`	`(f' * g - f * g') / (g * g)`

- Un caso por variante, y cada uno devuelve un **árbol nuevo**.

Derivar: un recorrido que construye

La expresión	Su derivada
Un número	`0`
La variable pedida	`1`
Otra variable	`0`
`f + g`	`f' + g'`
`f - g`	`f' - g'`
`f * g`	`f' * g + f * g'`
`f / g`	`(f' * g - f * g') / (g * g)`

- Un caso por variante, y cada uno devuelve un **árbol nuevo**.
- En `f * g`, cada una aparece dos veces.

Derivar: un recorrido que construye

La expresión	Su derivada
Un número	`0`
La variable pedida	`1`
Otra variable	`0`
`f + g`	`f' + g'`
`f - g`	`f' - g'`
`f * g`	`f' * g + f * g'`
`f / g`	`(f' * g - f * g') / (g * g)`

- Un caso por variante, y cada uno devuelve un **árbol nuevo**.
- En `f * g`, cada una aparece dos veces.
- En `f / g`, `g` aparece tres veces.

Derivar: un recorrido que construye

La expresión	Su derivada
Un número	`0`
La variable pedida	`1`
Otra variable	`0`
`f + g`	`f' + g'`
`f - g`	`f' - g'`
`f * g`	`f' * g + f * g'`
`f / g`	`(f' * g - f * g') / (g * g)`

- Un caso por variante, y cada uno devuelve un **árbol nuevo**.
- En `f * g`, cada una aparece dos veces.
- En `f / g`, `g` aparece tres veces.
- Un nodo tiene un solo padre: cada aparición tal cual es un `expr_copy`.

Derivar: un recorrido que construye

La expresión	Su derivada
Un número	`0`
La variable pedida	`1`
Otra variable	`0`
`f + g`	`f' + g'`
`f - g`	`f' - g'`
`f * g`	`f' * g + f * g'`
`f / g`	`(f' * g - f * g') / (g * g)`

- Un caso por variante, y cada uno devuelve un **árbol nuevo**.
- En `f * g`, cada una aparece dos veces.
- En `f / g`, `g` aparece tres veces.
- Un nodo tiene un solo padre: cada aparición tal cual es un `expr_copy`.
- `e` es `const`: se mira, y sigue igual.

Comprobar con otro método

con la solución del laboratorio

```
$ build/solution/algebra
expresión:    (x + 1) * (x - 2)
copia igual:  sí
derivada:     (1 + 0) * (x - 2) + (x + 1) * (1 - 0)
simplificada: x - 2 + (x + 1)
  x = 0   derivada = -1.0000   numérica = -1.0000   bien
  x = 1   derivada = 1.0000    numérica = 1.0000    bien
  x = 2.5 derivada = 4.0000    numérica = 4.0000    bien
```

Comprobar con otro método

con la solución del laboratorio

```
$ build/solution/algebra
expresión:    (x + 1) * (x - 2)
copia igual:  sí
derivada:     (1 + 0) * (x - 2) + (x + 1) * (1 - 0)
simplificada: x - 2 + (x + 1)
  x = 0   derivada = -1.0000   numérica = -1.0000   bien
  x = 1   derivada = 1.0000    numérica = 1.0000    bien
  x = 2.5 derivada = 4.0000    numérica = 4.0000    bien
```

- La columna `numérica` es la pendiente entre dos puntos muy cercanos: $(f(x+h) - f(x-h))/2h$.

Comprobar con otro método

con la solución del laboratorio

```
$ build/solution/algebra
expresión:    (x + 1) * (x - 2)
copia igual:  sí
derivada:     (1 + 0) * (x - 2) + (x + 1) * (1 - 0)
simplificada: x - 2 + (x + 1)
  x = 0   derivada = -1.0000   numérica = -1.0000   bien
  x = 1   derivada = 1.0000    numérica = 1.0000    bien
  x = 2.5 derivada = 4.0000    numérica = 4.0000    bien
```

- La columna `numérica` es la pendiente entre dos puntos muy cercanos: $(f(x+h) - f(x-h))/2h$.
- No usa las reglas, así que no comparte sus errores.

Comprobar con otro método

con la solución del laboratorio

```
$ build/solution/algebra
expresión:    (x + 1) * (x - 2)
copia igual:  sí
derivada:     (1 + 0) * (x - 2) + (x + 1) * (1 - 0)
simplificada: x - 2 + (x + 1)
  x = 0   derivada = -1.0000   numérica = -1.0000   bien
  x = 1   derivada = 1.0000    numérica = 1.0000    bien
  x = 2.5 derivada = 4.0000    numérica = 4.0000    bien
```

- La columna `numérica` es la pendiente entre dos puntos muy cercanos: $(f(x+h) - f(x-h))/2h$.
- No usa las reglas, así que no comparte sus errores.

Error común: comparar el texto de la derivada. Hay muchas formas correctas: lo que debe coincidir es el valor.

SECCIÓN 09

Laboratorio

09

Laboratorio: álgebra de expresiones

Sesión 1

1. Leer `expr.h`, `env.h` y las pruebas.
2. Los constructores y `expr_free`.
3. `expr_print`, con los paréntesis necesarios.
4. `expr_eval`, con `Env` por su interfaz.

Sesión 2

5. `expr_copy` y `expr_equal`.
6. `expr_simplify`, cuidando quién libera cada nodo.
7. `expr_derive`, y comprobar con `algebra`.

Antes de empezar

Lean `lab/README.md`. Una fuga también hace fallar una prueba. Este árbol existe para representar expresiones: los árboles de búsqueda y las tablas hash le tocan a Estructuras de Datos.

Resumen

- Un **árbol de expresión** guarda qué se opera con qué. Los paréntesis no se guardan.
- Un tipo recursivo es una unión etiquetada con punteros a sí misma.
- **Cada nodo es owner de sus hijos**. Un nodo tiene un solo padre. Si se necesita dos veces, se copia.
- Un constructor se queda con sus hijos aunque falle, y acepta `NULL`. Por eso se anidan.
- **La función sigue a los datos**: un `switch` sin `default`, un caso por variante.
- Para liberar, primero los hijos. Termina porque un subárbol tiene menos nodos.
- Paréntesis: izquierdo si es menor, derecho si es menor o igual.
- Lo que puede fallar devuelve un `int`, y el valor por un puntero.
- Transformar: `e = logic_simplify(e);`. Lo que queda se devuelve, lo descartado se libera.

Para la próxima clase

- Terminen el laboratorio: `make test-lab`, sin informes de ASan.
- Resuelvan los ejercicios de los apuntes, sobre todo el 2, el 4 y el 7.
- Respondan las preguntas de repaso de la guía de la semana 13.
- La semana 14 convierte texto en estos árboles: análisis léxico y análisis sintáctico.

Antes de escribir una función sobre un árbol

¿Qué hago en cada variante? ¿Qué nodos reservo, y quién es su owner cuando termine?