

UNIVERSIDAD NACIONAL DE INGENIERÍA

Introducción a la Programación

Análisis léxico y sintáctico

Del texto al árbol: tokens, gramáticas y parsers descendentes recursivos

Gerson Garrido

Semana 14

Hoy

1. Las etapas de un intérprete y el lenguaje Mini
2. Análisis léxico: del texto a los tokens
3. Gramáticas: precedencia y asociatividad
4. Parser descendente recursivo: una función por regla
5. Por qué la recursión por la izquierda no se programa tal cual
6. Errores de sintaxis: la posición, y liberar lo ya construido

> **Lectura**
>
> Apuntes de la semana 14. Código en `code/week-14`.

La pregunta de la semana

La semana pasada, `1 + 2 * x` se construía a mano:

a mano

```
expr_binary(OP_ADD, expr_number(1), expr_binary(OP_MUL, expr_number(2), expr_variable("x")))
```

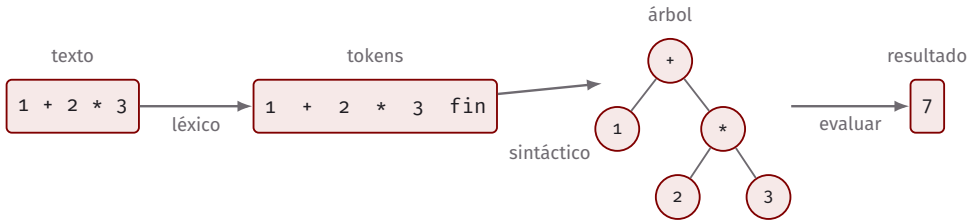
- Nadie programa así. Un programa es **texto**.
- ¿Cómo se pasa de los caracteres `1 + 2 * x` a ese árbol?
- ¿Quién decide que `1 + 2 * 3` vale 7 y no 9?

SECCIÓN 01

Las etapas

01

Un intérprete es una cadena



Cada etapa ignora los detalles de las demás

El analizador léxico sabe de espacios y comentarios. El parser sabe de precedencia. Si cambia uno, el otro no se entera.

El lenguaje Mini

un programa completo

```
# El factorial, con recursión
def fact(n) = if n < 2 then 1 else n * fact(n - 1)
let x = 5
print fact(x)
```

- Números, variables, `let`, `print`, `if`, `def` y llamadas.
- Todo es `double`. Una comparación vale 1 o 0.
- Los saltos de línea son como espacios.
- **Hoy:** solo expresiones. Sin `if` ni llamadas.

Toda la gramática de Mini

diez reglas

```
program   → statement*
statement → "let" NAME "=" expr
          | "print" expr
          | "def" NAME "(" [ NAME ("," NAME)* ] ")" "=" expr
expr      → "if" expr "then" expr "else" expr | compare
compare   → sum [ ("<" | "<=" | ">" | ">=" | "==" | "!=") sum ]
sum       → term (("+" | "-") term)*
term      → unary (("*" | "/") unary)*
unary     → "-" unary | primary
primary   → NUMBER | NAME | NAME "(" [ expr ("," expr)* ] ")" | "(" expr ")"
```

Hoy: de `compare` para abajo, y en `primary` solo `NUMBER`, `NAME` y paréntesis. Lo demás es de la semana 15, en el mismo molde.

SECCIÓN 02

Análisis léxico

02



Token y analizador léxico

Token

Unidad mínima de un texto para un lenguaje: un número, un nombre, un símbolo.

lab/lexer.h

```c
typedef struct {
    TokenKind kind;
    double number;
    char text[64];
    int line;
    int column;
} Token;
```

lab/lexer.h

```c
typedef struct lexer Lexer;
Lexer *lexer_new(const char *source);
void lexer_free(Lexer *lexer);
Token lexer_next(Lexer *lexer);
```

- Tipo opaco, como en la semana 6.
- Se piden de uno en uno. No hay lista guardada.
- No copia el texto: debe seguir vivo.
- Sin la posición, luego no se podría decir dónde hay un error.

Los tokens de una línea

terminal

```
$ build/lab/show_tokens 'let x1 = 3*(y+2)'
1:1   LET           let
1:5   NAME          x1
1:8   ASSIGN        =
1:10  NUMBER        3
1:11  STAR          *
1:12  LPAREN        (
1:13  NAME          y
1:14  PLUS          +
1:15  NUMBER        2
1:16  RPAREN        )
1:17  END
```

- Los espacios desaparecieron.
- `x1` es un solo token.
- `let` es una palabra reservada. `lets` sería un nombre: primero se lee la palabra entera, y luego se compara.
- `END` evita un caso especial: es un token más.

El primer carácter decide

pseudocódigo

```
algoritmo lexer_next(lexer)
    saltar los espacios
    c ← carácter actual
    si c = NULO entonces
        devolver token END
    sino si c es un dígito entonces
        devolver leer_numero(lexer)
    sino si c es una letra o _ entonces
        devolver leer_nombre(lexer)
    sino si c es un símbolo conocido entonces
        consumir c
        devolver el token de ese símbolo
    sino
        consumir c
        devolver token ERROR
    fin si
fin algoritmo
```

Una sola función avanza

lab/lexer.c

```c
static void advance(Lexer *lexer) {
    if (*lexer->current == '\n') {
        lexer->line++;
        lexer->column = 1;
    } else {
        lexer->column++;
    }
    lexer->current++;
}
```

- Si todo pasa por aquí, la posición siempre está bien.
- Línea y columna empiezan en 1, como en `gcc`.
- La columna cuenta bytes: una `ñ` ocupa dos.

Con `ctype.h`

`isdigit((unsigned char) c)`: sin el cast, un byte negativo es comportamiento indefinido.

Los números se leen a mano

terminal

```
$ build/lab/show_tokens '1e5 0x10 3. .5'
1:1   NUMBER          1
1:2   NAME            e5
1:5   NUMBER          0
1:6   NAME            x10
1:10  NUMBER          3
1:11  ERROR           carácter inesperado '.'
1:13  ERROR           carácter inesperado '.'
1:14  NUMBER          5
1:15  END
```

- `strtod` entiende `1e5`, `0x10` e `inf`: mucho más que Mini.
- El lexer busca dónde termina el número, lo copia, y solo entonces llama a `strtod`.
- El punto solo entra si le sigue un dígito.
- `12abc` son dos tokens válidos. Se queja el parser, no el lexer.

El token más largo, y los errores

Lo que falta: el laboratorio

con su lexer terminado

```
$ build/lab/show_tokens 'x <= 1 # hola'
1:1   NAME          x
1:3   LESS_EQUAL    <=
1:6   NUMBER        1
1:14  END
```

- Al leer `<`, hay que mirar un carácter más.
- `===` se lee `==` y `=`: el más largo posible.

`TOKEN_ERROR`

El lexer no escribe nada ni detiene el programa. Devuelve un token con el mensaje en `text`, y consume el carácter.

`snprintf`

Como `printf`, pero en una cadena de tamaño fijo: no se pasa y siempre pone el `'\0'`.

SECCIÓN 03

Gramáticas

03

Cómo se lee una gramática

expresiones lógicas: el ejemplo de la clase

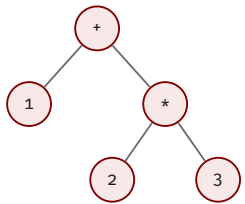
```
or_rule  → and_rule ("or" and_rule)*
and_rule → not_rule ("and" not_rule)*
not_rule → "not" not_rule | atom
atom     → "true" | "false" | "(" or_rule ")"
```

Las reglas se usan a sí mismas: `not_rule` directo, y `atom` a través de los paréntesis.

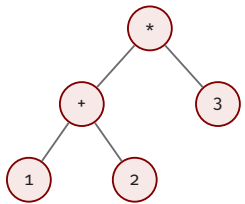
Notación	Se lee
`a → b`	`a` se escribe como `b`
`"or"`	ese token, tal cual
`NAME`	un token de ese tipo
`a b`	`a` y luego `b`
`a | b`	`a` o `b`
`( ... )*`	cero o más veces
`[ ... ]`	cero o una vez

Una gramática ambigua

```
expr → expr "+" expr | expr "*" expr | NUMBER
```



vale 7



vale 9

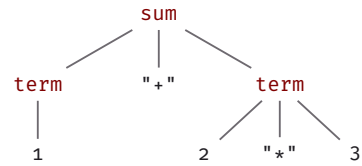
Dos árboles para un mismo texto

Dice bien qué textos son válidos, pero no qué significan. Un programa tendría dos resultados.

La precedencia se escribe con niveles

gramática

```
sum     → term (("+" | "-") term)*
term    → unary (("*" | "/") unary)*
unary   → "-" unary | primary
primary → NUMBER | NAME | "(" expr ")"
```



Se omiten `unary` y `primary`. El `2 * 3` queda dentro de un `term`: vale 7.

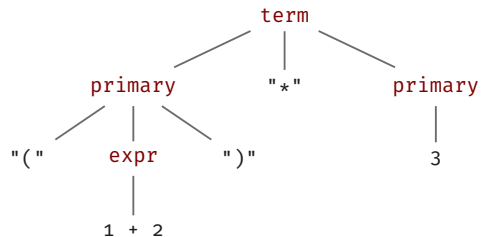
- Una suma es una lista de términos.
- Un término no conoce el `+`. Así que `1 + 2` no puede ser un término.

Cuanto más abajo, más fuerte liga

Nadie le dio a `*` una prioridad numérica. La precedencia es la forma de las reglas.

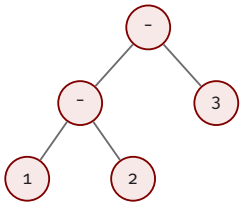
Los paréntesis vuelven a empezar

```
primary → ... | "(" expr ")"
```

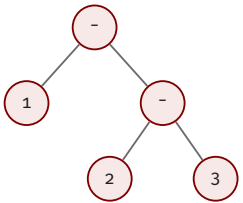


En `(1 + 2) * 3`, dentro de un `primary` se puede volver a escribir una expresión entera. Vale 9.

La asociatividad: `1 - 2 - 3`



`sum` → `sum "-" term`
vale −4. La de Mini y la de C



`sum` → `term "-" sum`
vale 2

El problema de esta semana

La regla de la izquierda es la correcta, y no se puede programar tal cual.

SECCIÓN 04

Parser descendente recursivo

04



Una función por regla

En la gramática	En el programa
una regla	una función
otra regla	una llamada
un token	revisar el actual y avanzar
`a | b`	un `si`, mirando el token
`( ... )*`	un `mientras`
`[ ... ]`	un `si`

Parser descendente recursivo
(recursive descent parser)

Empieza por la primera regla y baja hacia los tokens. Las reglas se llaman entre sí.

Con mirar un token basta: no hay que probar y retroceder como en el backtracking.

Todo el estado cabe en un `struct`

logic_parser.c

```c
typedef struct {
    Lexer *lexer;
    Token current;
    LogicError *error;
} Parser;
```

logic_parser.c

```c
static void advance(Parser *p) {
    p->current = lexer_next(p->lexer);
}
```

- `current` es el token que toca mirar, todavía sin consumir.
- Avanzar es pedir el siguiente.
- Sin variables globales: dos análisis no se pisan.

Una lista con operadores

pseudocódigo

```
algoritmo parse_sum(p)
    izquierda ← parse_term(p)
    mientras p.actual es "+" o "-" hacer
        op ← p.actual
        avanzar(p)
        derecha ← parse_term(p)
        izquierda ← nodo(op, izquierda, derecha)
    fin mientras
    devolver izquierda
fin algoritmo
```

`sum` → `term (("+" | "-") term)*`

- `term`: una llamada.
- `( ... )*`: un `mientras`.
- Lo acumulado queda a la izquierda del nodo nuevo.

Alternativas, y el salto hacia arriba

pseudocódigo

```
algoritmo parse_primary(p)
    si p.actual es NUMBER entonces
        hoja ← nodo_numero(p.actual)
        avanzar(p)
        devolver hoja
    sino si p.actual es "(" entonces
        avanzar(p)
        adentro ← parse_expr(p)
        exigir ")" y avanzar(p)
        devolver adentro
    sino
        error: se esperaba una expresión
    fin si
fin algoritmo
```

- `parse_expr` baja hasta llamar otra vez a `parse_primary`.
- Cada paréntesis abierto es un grupo más de marcos en la pila.

Idea central

La recursión de la gramática se vuelve recursión del programa.

En C: `not_rule` → `"not" not_rule | atom`

logic_parser.c

```c
static Logic *parse_not(Parser *p) {
    if (!is_word(p, "not")) {
        return parse_atom(p);
    }
    advance(p);
    Logic *operand = parse_not(p);
    if (operand == NULL) {
        return NULL;
    }
    Logic *node = logic_new(LOGIC_NOT, 0, operand, NULL)
      ;
    if (node == NULL) {
        fail(p, "no hay memoria");
    }
    return node;
}
```

- El `|` es el `if`.
- Primero **consume** el `not`, y luego se llama.
- `NULL` quiere decir que el error ya está anotado: solo se pasa hacia arriba.
- `logic_new` toma posesión de sus hijos, como `expr_binary`.

En C: `or_rule` → `and_rule ("or" and_rule)*`

logic_parser.c, abreviado

```c
static Logic *parse_or(Parser *p) {
    Logic *left = parse_and(p);
    ...
    while (is_word(p, "or")) {
        advance(p);
        Logic *right = parse_and(p);
        if (right == NULL) {
            logic_free(left);
            return NULL;
        }
        left = logic_new(LOGIC_OR, 0, left, right);
        ...
    }
    return left;
}
```

terminal

```
$ build/logic_parser 'true or false and false'
árbol: (true or (false and false))
valor: true
$ build/logic_parser '(true or false) and false'
árbol: ((true or false) and false)
valor: false
```

`and` liga más fuerte: `and_rule` está más abajo.

`true or false and false`, paso a paso

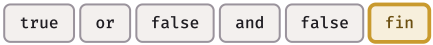
Token	Pila	Qué pasa
`true`	`or` → `and` → `not` → `atom`	crea la hoja `true` y avanza
`or`	`or` → `and`	`and` ve `or`, no es lo suyo, y regresa la hoja
`or`	`or`	`left` es `true`. Ve `or`: entra al ciclo y avanza
`false`	`or` → `and` → `not` → `atom`	crea la hoja `false` y avanza
`and`	`or` → `and`	`left` de `and` es `false`. Entra al ciclo y avanza
`false`	`or` → `and` → `not` → `atom`	crea la segunda hoja `false` y avanza
fin	`or` → `and`	arma `(false and false)` y regresa
fin	`or`	arma `(true or (false and false))` y regresa

La promesa de cada función

Al entrar, el token actual es el primero de su regla. Al salir, ya consumió lo suyo, y el actual es el primero que no le pertenece.

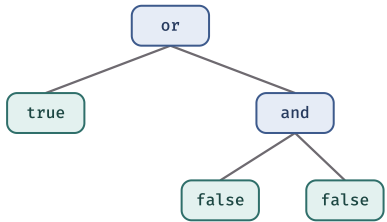
El mismo recorrido, en la pila y en el heap

TOKEN ACTUAL



PILA DE LLAMADAS

LO QUE YA ESTÁ EN EL HEAP




```
parse_or
left = (true or (false and false))
```

Arma todo el árbol. No hay otro or: regresa.

SECCIÓN 05

Recursión por la izquierda

05



Copiar la regla tal cual

gramática

```
chain ⟶ chain "-" NUMBER | NUMBER
```

errors/left_recursion.c

```c
static double parse_chain(Parser *p) {
    double left = parse_chain(p);       // primero la regla misma, sin consumir nada
```

Mismo token actual, mismos datos, otra vez. No hay caso base. `gcc` ya lo nota:

terminal

```
$ gcc -Wall -Wextra -std=c17 -Ilab errors/left_recursion.c lab/lexer.c -o /dev/null
errors/left_recursion.c:15:15: warning: infinite recursion detected [-Winfinite-recursion]
errors/left_recursion.c:16:19: note: recursive call
```

La pila se llena

extracto de ASan; las direcciones cambian

```
$ build/left_recursion
==154==ERROR: AddressSanitizer: stack-overflow on address 0xfffff2cdefd0 (pc 0xaaaae95e437c ...)
    #1 0xaaaae95e438c in parse_chain errors/left_recursion.c:16
    #2 0xaaaae95e438c in parse_chain errors/left_recursion.c:16
    #3 0xaaaae95e438c in parse_chain errors/left_recursion.c:16
```

La misma firma de la semana 9: `stack-overflow`, con una línea repetida. Esto es **recursión por la izquierda** *(left recursion)*: una regla que empieza por sí misma.

> **La regla**
>
> Toda llamada recursiva debe haber consumido al menos un token antes. Así cada nivel tiene menos texto por delante, y el texto es finito. `not_rule` → `"not" not_rule` lo cumple.

Voltear la regla tampoco sirve

gramática

```
chain ⟶ NUMBER [ "-" chain ]
```

subtract_chain.c

```c
static double chain_right(Parser *p) {
    double value = read_number(p);
    if (p->current.kind == TOKEN_MINUS) {
        advance(p);
        return value - chain_right(p);
    }
    return value;
}
```

- Termina: consume un número antes de llamarse.
- Acepta los mismos textos.
- Pero agrupa `1 - (2 - 3)`.

terminal

```
$ build/subtract_chain '1 - 2 - 3'
recursión por la derecha: 2
con un ciclo:             -4
```

No se cae, no tiene fugas, y calcula mal.

El ciclo reemplaza a la recursión

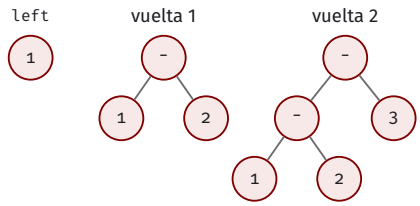
`sum` → `sum "-" term | term` produce un `term`, y luego cero o más `- term`:

gramática

```
sum → term ("-" term)*
```

subtract_chain.c

```c
static double chain_loop(Parser *p) {
    double value = read_number(p);
    while (p->current.kind == TOKEN_MINUS)
      {
        advance(p);
        value = value - read_number(p);
    }
    return value;
}
```



Lo acumulado pasa a ser el **hijo izquierdo**. Es el `array_fold` de la semana 12.

Tres reglas para lo mismo

Regla	¿Termina?	Agrupa	¿Sirve?
`sum → sum "-" term | term`	no	por la izquierda	correcta, no se programa
`sum → term "-" sum | term`	sí	por la derecha	da mal `1 - 2 - 3`
`sum → term ("-" term)*`	sí	por la izquierda	**esta**

No todo lleva ciclo

El - unario asocia por la derecha: `- - x` es `-(-x)`. Ahí `unary → "-" unary` es lo correcto.

SECCIÓN 06

Errores de sintaxis

06



Dónde y qué

logic_parser.c

```c
static void fail(Parser *p, const char *message) {
    p->error->line = p->current.line;
    p->error->column = p->current.column;
    const char *text = p->current.kind == TOKEN_ERROR ? p->current.text : message;
    snprintf(p->error->message, sizeof p->error->message, "%s", text);
}
```

- El culpable es el token actual, que trae su posición.
- `TOKEN_ERROR` ya trae su mensaje.
- Se acabó el texto: el error está en `END`.
- Al final se exige `END`, o `1 2` pasa.

terminal

```
$ build/logic_parser 'true @'
línea 1, columna 6: carácter inesperado '@'
$ build/logic_parser 'true or (false'
línea 1, columna 15: se esperaba ')'
$ build/logic_parser 'true and false)'
línea 1, columna 15: sobra texto después de la expresión
```

Qué hay vivo cuando falla

```
terminal

$ build/logic_parser 'true or false and (true or'
línea 1, columna 27: se esperaba true, false o '('
```

Función activa	Su `left`	Al recibir `NULL`
`parse_or`, la de afuera	hoja `true`	la libera y devuelve `NULL`
`parse_and`, la de afuera	hoja `false`	la libera y devuelve `NULL`
`parse_atom`, por el paréntesis		devuelve `NULL`
`parse_or`, la de adentro	hoja `true`	la libera y devuelve `NULL`
`parse_atom`, la última		reporta el error y devuelve `NULL`

No hay un árbol: hay hojas sueltas en marcos distintos

Quien tiene la variable local es el owner. El `NULL` sube por todos los marcos, y cada uno libera lo suyo.

Las ocho funciones activas al fallar



¿Quién libera?

Situación	¿Quién libera?
`right` falló, y `left` ya está	esta función libera `left`
`logic_new(kind, left, right)` falló	`logic_new` ya los liberó. Si los liberan otra vez, es un double free
Falló el `)`, e `inside` ya está	esta función libera `inside`
Sobra texto, y `tree` ya está	quien hizo esa comprobación libera `tree`

Prueben con entradas malas

Las fugas de un parser están en los caminos de error, y esos solo corren con entradas que fallan tarde.

Si se olvida: una fuga

errors/leak_on_error.c

```
1           if (right == NULL) {
2               return NULL;                // ¿y left?
```

extracto de ASan; las direcciones cambian

```
$ build/leak_on_error
==163==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 16 byte(s) in 1 object(s) allocated from:
    #1 0xaaaab7fe4954 in node_new errors/leak_on_error.c:31
    #2 0xaaaab7fe4b40 in parse_atom errors/leak_on_error.c:47
    #3 0xaaaab7fe4b60 in parse_or errors/leak_on_error.c:51
```

- `gcc` no dice nada, y el mensaje de error sale bien.
- El informe dice dónde se **reservó**. Hay que seguir al owner: `parse_atom` lo crea, y `parse_or` lo recibe en `left`.

Un límite que no vamos a arreglar

extracto de ASan; el límite exacto depende de la máquina

```
$ build/logic_parser "$(python3 -c 'print("(" * 100000)')"
ERROR: AddressSanitizer: stack-overflow
```

- Cada paréntesis abierto son varios marcos en la pila, y la pila es finita.
- Con mil paréntesis no hay problema. Con ASan los marcos son más grandes.
- Un parser serio cuenta la profundidad y devuelve un error. Es uno de los ejercicios.

SECCIÓN 07

El laboratorio

07

Lo que van a escribir

expresiones de Mini, esta semana

```
expr    → compare
compare → sum [ ("<" | "<=" | ">" | ">=" | "==" | "!=") sum ]
sum     → term (("+" | "-") term)*
term    → unary (("*" | "/") unary)*
unary   → "-" unary | primary
primary → NUMBER | NAME | "(" expr ")"
```

- `-x` se construye como `0 - x`.
- `[ ]` es un `si`. Por eso `1 < 2 < 3` es un error.
- Un constructor con `NULL`: no hay memoria. También va un mensaje.

con su parser terminado

```
$ build/lab/parse_main '-x * 2'
árbol: (0 - x) * 2
valor: -4
$ build/lab/parse_main '(x + 2'
(x + 2
     ^
línea 1, columna 7: se esperaba ')'
```

Laboratorio: una sesión, unos 90 minutos

1. **El lexer** (25 min). Leerlo con sus pruebas. Agregar comparadores y comentarios.
2. **El parser** (45 min). Una función por regla, que produce el `Expr` de la semana 13.
3. **Los errores** (20 min). Línea, columna y mensaje, sin fugas.

Reciben hecho

`expr.c` con las comparaciones, `env.c` y el lexer base.

Antes de empezar

Lean `lab/README.md`. `make test-lab` falla hasta que terminen. La otra sesión es para la Práctica 4.

Resumen

- Texto $\rightarrow$ tokens $\rightarrow$ árbol $\rightarrow$ resultado. Cada etapa ignora los detalles de las demás.
- El lexer entrega tokens de uno en uno, con posición. Se queda con el más largo. `TOKEN_ERROR` lleva su mensaje.
- Una gramática **ambigua** da dos árboles. La **precedencia** se escribe con niveles: más abajo, más fuerte.
- Parser descendente recursivo: una función por regla. La recursión de la gramática se vuelve recursión del programa.
- **Recursión por la izquierda**: se llama sin consumir, y la pila se llena. Por la derecha, asocia al revés.
- Un **ciclo** la reemplaza: lo acumulado pasa a ser el hijo izquierdo.
- Un error usa la posición del token actual. El `NULL` sube, y cada marco libera lo suyo.
- Al final se exige `END`. Si no, `1 2` pasa como válido.

Para la próxima clase

- Terminen el laboratorio: `make test-lab`, sin informes de ASan.
- Resuelvan los ejercicios de los apuntes, sobre todo el 3, el 6 y el 8.
- Respondan las preguntas de repaso de la guía de la semana 14.
- La semana 15 cierra el curso: instrucciones, funciones y marcos en Mini.

Antes de escribir una regla

¿Cuál es su primer token? ¿La función consume algo antes de llamarse a sí misma? Si falla, ¿qué tiene en sus variables locales?