

UNIVERSIDAD NACIONAL DE INGENIERÍA

Introducción a la Programación

El intérprete

Variables, funciones y marcos, y el repaso del curso

Gerson Garrido

Semana 15

Hoy

1. Instrucciones y dos variantes más del árbol
2. Completar el parser: `let`, `print`, `def`, `if` y llamadas
3. Ejecutar: el contexto y la tabla de funciones
4. Evaluar una llamada: el modelo de la semana 3, ahora escrito por ustedes
5. Dos pilas a la vez, y una recursión sin fin
6. Errores de ejecución y pruebas por niveles
7. Repaso: qué idea sirvió en cada parte

> **Lectura**
>
> Apuntes de la semana 15. Código en `code/week-15`.

La meta

lab/programs/fact.mini

```
def fact(n) = if n < 2 then 1 else n * fact(n - 1)
let x = 5
print fact(x)     # 120
```

La meta

lab/programs/fact.mini

```
def fact(n) = if n < 2 then 1 else n * fact(n - 1)
let x = 5
print fact(x)     # 120
```

terminal

```
$ build/lab/mini lab/programs/fact.mini
120
```

La meta

lab/programs/fact.mini

```
def fact(n) = if n < 2 then 1 else n * fact(n - 1)
let x = 5
print fact(x)     # 120
```

terminal

```
$ build/lab/mini lab/programs/fact.mini
120
```

- Ya tienen: un árbol que se evalúa (semana 13), un parser de expresiones (semana 14).

La meta

lab/programs/fact.mini

```
def fact(n) = if n < 2 then 1 else n * fact(n - 1)
let x = 5
print fact(x)     # 120
```

terminal

```
$ build/lab/mini lab/programs/fact.mini
120
```

- Ya tienen: un árbol que se evalúa (semana 13), un parser de expresiones (semana 14).
- Faltan tres cosas: instrucciones, un `if` y llamadas.

La meta

lab/programs/fact.mini

```
def fact(n) = if n < 2 then 1 else n * fact(n - 1)
let x = 5
print fact(x)     # 120
```

terminal

```
$ build/lab/mini lab/programs/fact.mini
120
```

- Ya tienen: un árbol que se evalúa (semana 13), un parser de expresiones (semana 14).
- Faltan tres cosas: instrucciones, un `if` y llamadas.
- **La idea de la semana:** evaluar una llamada es hacer lo que ustedes dibujan en papel desde la semana 3.

Las etapas, ya completas



Las etapas, ya completas



- En el laboratorio, lo de las semanas 13 y 14 ya viene hecho.

Las etapas, ya completas



- En el laboratorio, lo de las semanas 13 y 14 ya viene hecho.
- Ustedes escriben: `parse_if`, `parse_call`, `parse_let`, `parse_print`, el final de `parse_def`, `eval_if`, el final de `eval_call`, y dos casos de `exec_statement`.

SECCIÓN 01

Instrucciones y árbol

01

El lenguaje completo

gramática de Mini

```
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 ")"
```

El lenguaje completo

gramática de Mini

```
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 ")"
```

- Todo es `double`. Una comparación da 1 o 0. En un `if`, 0 es falso.

El lenguaje completo

gramática de Mini

```
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 ")"
```

- Todo es `double`. Una comparación da 1 o 0. En un `if`, 0 es falso.
- `let` define o cambia una global. Dentro de una función solo hay parámetros.

El lenguaje completo

gramática de Mini

```
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 ")"
```

- Todo es `double`. Una comparación da 1 o 0. En un `if`, 0 es falso.
- `let` define o cambia una global. Dentro de una función solo hay parámetros.
- El cuerpo de una función es **una expresión**. Sin ciclos: hay recursión.

Dos variantes más para `Expr`

lab/expr.h

```c
struct expr {
    ExprKind kind;
    union {
        double number;
        char *name;                                          // EXPR_VARIABLE: copia propia
        struct { BinaryOp op; Expr *left; Expr *right; } binary;
        struct { Expr *condition; Expr *then_branch; Expr *else_branch; } conditional;
        struct { char *name; Expr **args; int arg_count; } call;    // args es NULL si arg_count
      es 0
    } as;
};
```

Dos variantes más para `Expr`

lab/expr.h

```
1  struct expr {
2      ExprKind kind;
3      union {
4          double number;
5          char *name;                                          // EXPR_VARIABLE: copia propia
6          struct { BinaryOp op; Expr *left; Expr *right; } binary;
7          struct { Expr *condition; Expr *then_branch; Expr *else_branch; } conditional;
8          struct { char *name; Expr **args; int arg_count; } call;    // args es NULL si arg_count
       es 0
9      } as;
10 };
```

- `conditional`, porque `if` es una palabra de C.

Dos variantes más para `Expr`

lab/expr.h

```c
struct expr {
    ExprKind kind;
    union {
        double number;
        char *name;                                          // EXPR_VARIABLE: copia propia
        struct { BinaryOp op; Expr *left; Expr *right; } binary;
        struct { Expr *condition; Expr *then_branch; Expr *else_branch; } conditional;
        struct { char *name; Expr **args; int arg_count; } call;    // args es NULL si arg_count es 0
    } as;
};
```

- `conditional`, porque `if` es una palabra de C.
- `args` es un arreglo dinámico de punteros a `Expr`. Primer nodo con hijos variables.

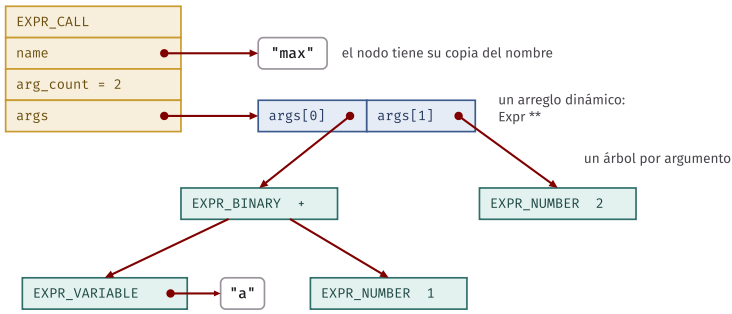
Dos variantes más para `Expr`

lab/expr.h

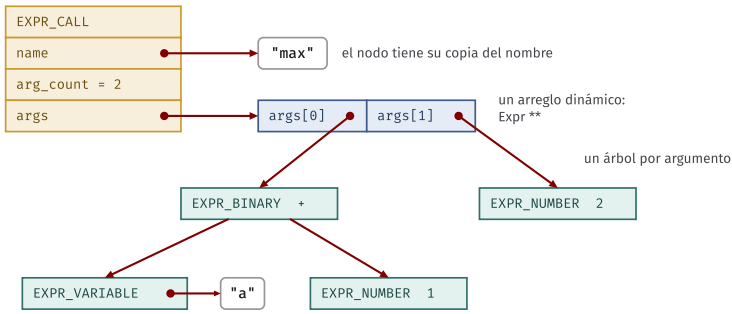
```
1  struct expr {
2      ExprKind kind;
3      union {
4          double number;
5          char *name;                                          // EXPR_VARIABLE: copia propia
6          struct { BinaryOp op; Expr *left; Expr *right; } binary;
7          struct { Expr *condition; Expr *then_branch; Expr *else_branch; } conditional;
8          struct { char *name; Expr **args; int arg_count; } call;    // args es NULL si arg_count
       es 0
9      } as;
10 };
```

- `conditional`, porque `if` es una palabra de C.
- `args` es un arreglo dinámico de punteros a `Expr`. Primer nodo con hijos variables.
- Agregar una variante toca cada `switch`. `-Wall` avisa si falta un caso y no hay `default`.

Una llamada en memoria: `max(a + 1, 2)`

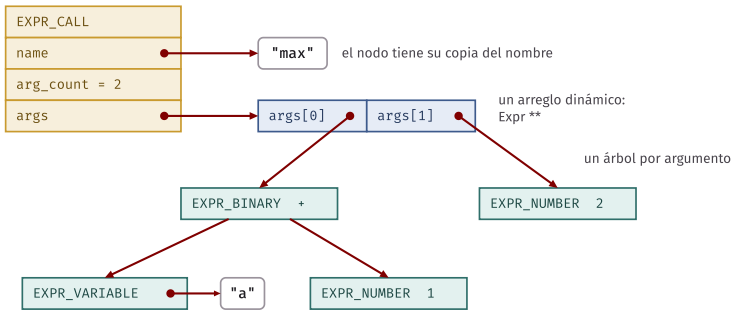


Una llamada en memoria: `max(a + 1, 2)`



- El nodo es owner de su cadena, del arreglo y de cada árbol.

Una llamada en memoria: `max(a + 1, 2)`



- El nodo es owner de su cadena, del arreglo y de cada árbol.
- `expr_free` libera primero lo que cuelga, luego el nodo.

Los constructores se apropian, aunque fallen

lab/expr.h

```
1  // Estos tres se vuelven owners de lo que reciben, aunque fallen: en ese caso lo liberan y
2  // devuelven NULL. Un hijo NULL cuenta como falla.
3  Expr *expr_binary(BinaryOp op, Expr *left, Expr *right);
4  Expr *expr_if(Expr *condition, Expr *then_branch, Expr *else_branch);
5  Expr *expr_call(const char *name, Expr **args, int arg_count);
```

Los constructores se apropian, aunque fallen

lab/expr.h

```
1  // Estos tres se vuelven owners de lo que reciben, aunque fallen: en ese caso lo liberan y
2  // devuelven NULL. Un hijo NULL cuenta como falla.
3  Expr *expr_binary(BinaryOp op, Expr *left, Expr *right);
4  Expr *expr_if(Expr *condition, Expr *then_branch, Expr *else_branch);
5  Expr *expr_call(const char *name, Expr **args, int arg_count);
```

- ¿Por qué se copia `name`, y `args` no?

Los constructores se apropian, aunque fallen

lab/expr.h

```
1  // Estos tres se vuelven owners de lo que reciben, aunque fallen: en ese caso lo liberan y
2  // devuelven NULL. Un hijo NULL cuenta como falla.
3  Expr *expr_binary(BinaryOp op, Expr *left, Expr *right);
4  Expr *expr_if(Expr *condition, Expr *then_branch, Expr *else_branch);
5  Expr *expr_call(const char *name, Expr **args, int arg_count);
```

- ¿Por qué se copia `name`, y `args` no?
- `const char *`: lo miro. Puede estar dentro de un token que está por cambiar.

Los constructores se apropian, aunque fallen

lab/expr.h

```c
// Estos tres se vuelven owners de lo que reciben, aunque fallen: en ese caso lo liberan y
// devuelven NULL. Un hijo NULL cuenta como falla.
Expr *expr_binary(BinaryOp op, Expr *left, Expr *right);
Expr *expr_if(Expr *condition, Expr *then_branch, Expr *else_branch);
Expr *expr_call(const char *name, Expr **args, int arg_count);
```

- ¿Por qué se copia `name`, y `args` no?
- `const char *`: lo miro. Puede estar dentro de un token que está por cambiar.
- `Expr **`: ahora es mío. Se reservó justo para este nodo.

Los constructores se apropian, aunque fallen

lab/expr.h

```
1  // Estos tres se vuelven owners de lo que reciben, aunque fallen: en ese caso lo liberan y
2  // devuelven NULL. Un hijo NULL cuenta como falla.
3  Expr *expr_binary(BinaryOp op, Expr *left, Expr *right);
4  Expr *expr_if(Expr *condition, Expr *then_branch, Expr *else_branch);
5  Expr *expr_call(const char *name, Expr **args, int arg_count);
```

- ¿Por qué se copia `name`, y `args` no?
- `const char *`: lo miro. Puede estar dentro de un token que está por cambiar.
- `Expr **`: ahora es mío. Se reservó justo para este nodo.

> **Semana 5**
>
> Cada bloque tiene un solo owner, y la firma con su comentario dice quién es.

Las instrucciones son otra unión etiquetada

lab/stmt.h

```c
struct stmt {
    StmtKind kind;
    union {
        struct { char *name; Expr *value; } let;
        struct { Expr *value; } print;
        struct { char *name; char **params; int param_count; Expr *body; } def;     // params es NULL si son 0
    } as;
    Stmt *next;
};
```



Las instrucciones son otra unión etiquetada

lab/stmt.h

```c
struct stmt {
    StmtKind kind;
    union {
        struct { char *name; Expr *value; } let;
        struct { Expr *value; } print;
        struct { char *name; char **params; int param_count; Expr *body; } def;     // params es
  NULL si son 0
    } as;
    Stmt *next;
};
```



- Una expresión tiene un valor. Una instrucción **hace** algo.

Las instrucciones son otra unión etiquetada

lab/stmt.h

```c
struct stmt {
    StmtKind kind;
    union {
        struct { char *name; Expr *value; } let;
        struct { Expr *value; } print;
        struct { char *name; char **params; int param_count; Expr *body; } def;     // params es
  NULL si son 0
    } as;
    Stmt *next;
};
```



- Una expresión tiene un valor. Una instrucción **hace** algo.
- `stmt_free` usa un ciclo: si no, la profundidad sería el largo del programa.

SECCIÓN 02

Completar el parser

02



El primer token decide

pseudocódigo

```
algoritmo analizar_let(p)
    consumir "let"
    nombre ← token actual
    exigir un NAME
    exigir "="
    valor ← analizar_expr(p)
    si falló la lectura de valor entonces
        devolver "falló"
    fin si
    devolver una instrucción let
fin algoritmo
```

El primer token decide

pseudocódigo

```
algoritmo analizar_let(p)
    consumir "let"
    nombre ← token actual
    exigir un NAME
    exigir "="
    valor ← analizar_expr(p)
    si falló la lectura de valor entonces
        devolver "falló"
    fin si
    devolver una instrucción let
fin algoritmo
```

- Un token basta para saber la regla. Nunca se deshace: aquí no hay backtracking.

El primer token decide

pseudocódigo

```
algoritmo analizar_let(p)
    consumir "let"
    nombre ← token actual
    exigir un NAME
    exigir "="
    valor ← analizar_expr(p)
    si falló la lectura de valor entonces
        devolver "falló"
    fin si
    devolver una instrucción let
fin algoritmo
```

- Un token basta para saber la regla. Nunca se deshace: aquí no hay backtracking.
- Por eso no hace falta separador: tras un `let`, un `print` ya no es parte de la expresión.

El primer token decide

pseudocódigo

```
algoritmo analizar_let(p)
    consumir "let"
    nombre ← token actual
    exigir un NAME
    exigir "="
    valor ← analizar_expr(p)
    si falló la lectura de valor entonces
        devolver "falló"
    fin si
    devolver una instrucción let
fin algoritmo
```

- Un token basta para saber la regla. Nunca se deshace: aquí no hay backtracking.
- Por eso no hace falta separador: tras un `let`, un `print` ya no es parte de la expresión.

Error común: `current.text` cambia con `advance`. Copien antes el token: `Token name = p->current;`

La lista: dónde enganchar la siguiente

lab/parser.c

```
1  Stmt *first = NULL;
2  Stmt **tail = &first;   // dónde se engancha la pró
       xima instrucción
3  ...
4  *tail = s;
5  tail = &s->next;
```

La lista: dónde enganchar la siguiente

lab/parser.c

```
1  Stmt *first = NULL;
2  Stmt **tail = &first;   // dónde se engancha la pró
       xima instrucción
3  ...
4  *tail = s;
5  tail = &s->next;
```

`tail` es la dirección de la casilla que hay que llenar: al inicio `first`, luego un `next`.

La lista: dónde enganchar la siguiente

`tail` es la dirección de la casilla que hay que llenar: al inicio `first`, luego un `next`.

lab/parser.c

```
1  Stmt *first = NULL;
2  Stmt **tail = &first;   // dónde se engancha la pró
       xima instrucción
3  ...
4  *tail = s;
5  tail = &s->next;
```

- Un archivo vacío es un programa correcto, y su lista es `NULL`.

La lista: dónde enganchar la siguiente

`tail` es la dirección de la casilla que hay que llenar: al inicio `first`, luego un `next`.

lab/parser.c

```
1  Stmt *first = NULL;
2  Stmt **tail = &first;   // dónde se engancha la pró
       xima instrucción
3  ...
4  *tail = s;
5  tail = &s->next;
```

- Un archivo vacío es un programa correcto, y su lista es `NULL`.
- Pero `NULL` también es error de sintaxis. Los distingue `error->message`.

lab/mini.c

```
1  if (program == NULL && syntax.message[0] != '\0') {
```

El `if`: una escalera de liberaciones

pseudocódigo

```
algoritmo analizar_if(p)
    consumir "if"
    condicion ← analizar_expr(p)
    si falló la lectura de condicion entonces
        devolver "falló"
    fin si
    si el token actual no es "then" entonces
        anotar el error
        liberar condicion
        devolver "falló"
    fin si
    consumir "then"
    rama_si ← analizar_expr(p)
    ...
fin algoritmo
```

El `if`: una escalera de liberaciones

pseudocódigo

```
algoritmo analizar_if(p)
    consumir "if"
    condicion ← analizar_expr(p)
    si falló la lectura de condicion entonces
        devolver "falló"
    fin si
    si el token actual no es "then" entonces
        anotar el error
        liberar condicion
        devolver "falló"
    fin si
    consumir "then"
    rama_si ← analizar_expr(p)
    ...
fin algoritmo
```

- Cada paso que falla libera todo lo que se tiene hasta ahí.

El `if`: una escalera de liberaciones

pseudocódigo

```
algoritmo analizar_if(p)
    consumir "if"
    condicion ← analizar_expr(p)
    si falló la lectura de condicion entonces
        devolver "falló"
    fin si
    si el token actual no es "then" entonces
        anotar el error
        liberar condicion
        devolver "falló"
    fin si
    consumir "then"
    rama_si ← analizar_expr(p)
    ...
fin algoritmo
```

- Cada paso que falla libera todo lo que se tiene hasta ahí.
- Al final, `expr_if` se vuelve owner de los tres.

El `if`: una escalera de liberaciones

pseudocódigo

```
algoritmo analizar_if(p)
    consumir "if"
    condicion ← analizar_expr(p)
    si falló la lectura de condicion entonces
        devolver "falló"
    fin si
    si el token actual no es "then" entonces
        anotar el error
        liberar condicion
        devolver "falló"
    fin si
    consumir "then"
    rama_si ← analizar_expr(p)
    ...
fin algoritmo
```

- Cada paso que falla libera todo lo que se tiene hasta ahí.
- Al final, `expr_if` se vuelve owner de los tres.
- `1 + if a then b else c` es un error: en un `sum`, el `if` entra con paréntesis.

Las llamadas: un token más, y un arreglo que crece

lab/parser.c, en parse_parameters

```c
if (count == capacity) {
    int new_capacity = capacity == 0 ? 4 : capacity * 2;
    char **grown = realloc(names, (size_t) new_capacity
      * sizeof *grown);
    if (grown == NULL) {
        no_memory(p);
        free_names(names, count);
        return 0;
    }
    names = grown;
    capacity = new_capacity;
}
```

Las llamadas: un token más, y un arreglo que crece

lab/parser.c, en parse_parameters

```c
if (count == capacity) {
    int new_capacity = capacity == 0 ? 4 : capacity * 2;
    char **grown = realloc(names, (size_t) new_capacity
      * sizeof *grown);
    if (grown == NULL) {
        no_memory(p);
        free_names(names, count);
        return 0;
    }
    names = grown;
    capacity = new_capacity;
}
```

- Como en la semana 5: `names` se reemplaza solo si `realloc` funcionó.

Las llamadas: un token más, y un arreglo que crece

lab/parser.c, en parse_parameters

```c
if (count == capacity) {
    int new_capacity = capacity == 0 ? 4 : capacity * 2;
    char **grown = realloc(names, (size_t) new_capacity
      * sizeof *grown);
    if (grown == NULL) {
        no_memory(p);
        free_names(names, count);
        return 0;
    }
    names = grown;
    capacity = new_capacity;
}
```

- Como en la semana 5: `names` se reemplaza solo si `realloc` funcionó.
- Se libera con `count`: solo esas casillas tienen algo.

Las llamadas: un token más, y un arreglo que crece

lab/parser.c, en parse_parameters

```c
if (count == capacity) {
    int new_capacity = capacity == 0 ? 4 : capacity * 2;
    char **grown = realloc(names, (size_t) new_capacity
      * sizeof *grown);
    if (grown == NULL) {
        no_memory(p);
        free_names(names, count);
        return 0;
    }
    names = grown;
    capacity = new_capacity;
}
```

- Como en la semana 5: `names` se reemplaza solo si `realloc` funcionó.
- Se libera con `count`: solo esas casillas tienen algo.
- `parse_call` es igual, con `Expr **`. En `f(a + 1, b * 2, c - )` cada función libera lo suyo.

Las llamadas: un token más, y un arreglo que crece

lab/parser.c, en parse_parameters

```c
if (count == capacity) {
    int new_capacity = capacity == 0 ? 4 : capacity * 2;
    char **grown = realloc(names, (size_t) new_capacity
      * sizeof *grown);
    if (grown == NULL) {
        no_memory(p);
        free_names(names, count);
        return 0;
    }
    names = grown;
    capacity = new_capacity;
}
```

- Como en la semana 5: `names` se reemplaza solo si `realloc` funcionó.
- Se libera con `count`: solo esas casillas tienen algo.
- `parse_call` es igual, con `Expr **`. En `f(a + 1, b * 2, c - )` cada función libera lo suyo.

Error común: olvidar `f()`. Con cero argumentos no se reserva nada: `malloc(0)` puede devolver `NULL`.

SECCIÓN 03

Ejecutar

03

El contexto del intérprete

lab/interp.c

```c
typedef struct {
    Env *global;            // el marco global de Mini
    Function *functions;    // la tabla de funciones de Mini
    FILE *out;
    RunError *error;
} Interp;
```

El contexto del intérprete

lab/interp.c

```c
typedef struct {
    Env *global;            // el marco global de Mini
    Function *functions;    // la tabla de funciones de Mini
    FILE *out;
    RunError *error;
} Interp;
```

- El contexto de la semana 12, sin el `void *`: aquí se sabe el tipo.

El contexto del intérprete

lab/interp.c

```c
typedef struct {
    Env *global;            // el marco global de Mini
    Function *functions;    // la tabla de funciones de Mini
    FILE *out;
    RunError *error;
} Interp;
```

- El contexto de la semana 12, sin el `void *`: aquí se sabe el tipo.
- "Global" significa global **en Mini**. En C no hay ninguna variable global.

El contexto del intérprete

lab/interp.c

```c
typedef struct {
    Env *global;            // el marco global de Mini
    Function *functions;    // la tabla de funciones de Mini
    FILE *out;
    RunError *error;
} Interp;
```

- El contexto de la semana 12, sin el `void *`: aquí se sabe el tipo.
- "Global" significa global **en Mini**. En C no hay ninguna variable global.
- Se pueden correr dos programas seguidos sin que se mezclen.

Ejecutar instrucciones

pseudocódigo

```
algoritmo ejecutar(s)
    si s es let entonces
        valor ← evaluar(expresión de s, marco global)
        definir el nombre de s en el marco global, con valor
    sino si s es print entonces
        valor ← evaluar(expresión de s, marco global)
        escribir valor
    sino si s es def entonces
        registrar s en la tabla de funciones
    fin si
fin algoritmo
```

Ejecutar instrucciones

pseudocódigo

```
algoritmo ejecutar(s)
    si s es let entonces
        valor ← evaluar(expresión de s, marco global)
        definir el nombre de s en el marco global, con valor
    sino si s es print entonces
        valor ← evaluar(expresión de s, marco global)
        escribir valor
    sino si s es def entonces
        registrar s en la tabla de funciones
    fin si
fin algoritmo
```

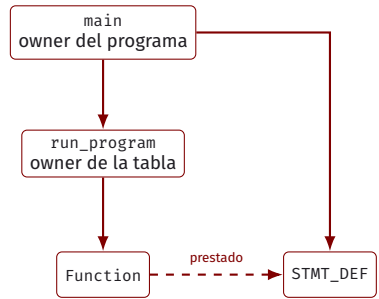
Lean el último caso

Un `def` no evalúa su cuerpo. Solo lo guarda. `def f(n) = n / 0` es válido hasta que alguien llame a `f`.

La tabla toma prestado

lab/interp.c

```c
typedef struct function {
    const Stmt *definition;
    struct function *next;
} Function;
```

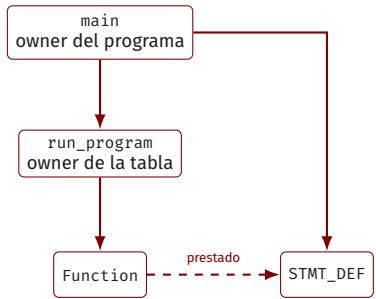


La tabla toma prestado

lab/interp.c

```c
typedef struct function {
    const Stmt *definition;
    struct function *next;
} Function;
```

- La entrada es de la tabla. El `def` es del programa.

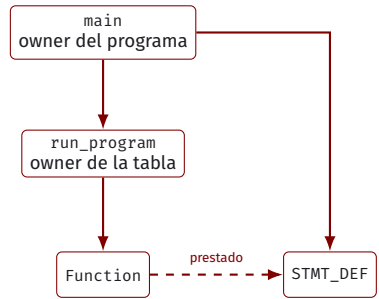


La tabla toma prestado

lab/interp.c

```c
typedef struct function {
    const Stmt *definition;
    struct function *next;
} Function;
```

- La entrada es de la tabla. El `def` es del programa.
- Seguro: el programa vive más que la tabla.

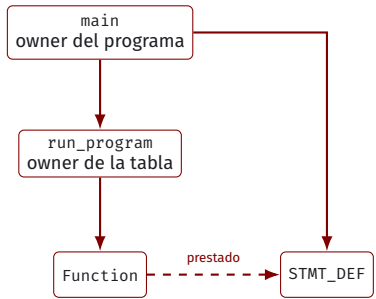


La tabla toma prestado

lab/interp.c

```c
typedef struct function {
    const Stmt *definition;
    struct function *next;
} Function;
```

- La entrada es de la tabla. El `def` es del programa.
- Seguro: el programa vive más que la tabla.
- El `const` hace que `free` se queje.



Un `def` tiene efecto cuando corre

before_def.mini

```
print 1
print twice(4)
def twice(n) = n * 2
```

even_odd.mini

```
def even(n) = if n == 0 then 1 else odd(n - 1)
def odd(n) = if n == 0 then 0 else even(n - 1)
print even(10)
print even(7)
```

Un `def` tiene efecto cuando corre

before_def.mini

```
print 1
print twice(4)
def twice(n) = n * 2
```

even_odd.mini

```
def even(n) = if n == 0 then 1 else odd(n - 1)
def odd(n) = if n == 0 then 0 else even(n - 1)
print even(10)
print even(7)
```

terminal

```
$ build/lab/mini lab/programs/before_def.mini
1
lab/programs/before_def.mini: error de ejecución: función desconocida: twice
$ build/lab/mini lab/programs/even_odd.mini
1
0
```

Un `def` tiene efecto cuando corre

before_def.mini

```
print 1
print twice(4)
def twice(n) = n * 2
```

even_odd.mini

```
def even(n) = if n == 0 then 1 else odd(n - 1)
def odd(n) = if n == 0 then 0 else even(n - 1)
print even(10)
print even(7)
```

terminal

```
$ build/lab/mini lab/programs/before_def.mini
1
lab/programs/before_def.mini: error de ejecución: función desconocida: twice
$ build/lab/mini lab/programs/even_odd.mini
1
0
```

El cuerpo de `even` nombra a `odd`, pero no corre hasta la llamada.

SECCIÓN 04

Evaluar

04

El `if` no es una función

lab/programs/my_if.mini

```
def my_if(c, a, b) = if c then a else b
print my_if(1, 10, 20)
def fact(n) = my_if(n < 2, 1, n * fact(n - 1))
print fact(3)
```

El `if` no es una función

lab/programs/my_if.mini

```
def my_if(c, a, b) = if c then a else b
print my_if(1, 10, 20)
def fact(n) = my_if(n < 2, 1, n * fact(n - 1))
print fact(3)
```

- `eval_if` evalúa la condición, y luego solo la rama elegida.

El `if` no es una función

lab/programs/my_if.mini

```
def my_if(c, a, b) = if c then a else b
print my_if(1, 10, 20)
def fact(n) = my_if(n < 2, 1, n * fact(n - 1))
print fact(3)
```

- `eval_if` evalúa la condición, y luego solo la rama elegida.
- Una llamada, en cambio, evalúa **todos** sus argumentos antes de entrar.

El `if` no es una función

lab/programs/my_if.mini

```
def my_if(c, a, b) = if c then a else b
print my_if(1, 10, 20)
def fact(n) = my_if(n < 2, 1, n * fact(n - 1))
print fact(3)
```

- `eval_if` evalúa la condición, y luego solo la rama elegida.
- Una llamada, en cambio, evalúa **todos** sus argumentos antes de entrar.
- Para llamar a `my_if` hay que calcular `fact(n - 1)`. El caso base está escrito, y nunca se usa.

El `if` no es una función

lab/programs/my_if.mini

```
def my_if(c, a, b) = if c then a else b
print my_if(1, 10, 20)
def fact(n) = my_if(n < 2, 1, n * fact(n - 1))
print fact(3)
```

- `eval_if` evalúa la condición, y luego solo la rama elegida.
- Una llamada, en cambio, evalúa **todos** sus argumentos antes de entrar.
- Para llamar a `my_if` hay que calcular `fact(n - 1)`. El caso base está escrito, y nunca se usa.

> **Idea clave**
>
> El `if` no se comporta como una llamada. El `if`, `&&` y `||` de C tienen esa misma propiedad.

El modelo de la semana 3, en código

Regla de la semana 3	En el intérprete
Calcular argumentos en el marco de quien llama	`eval` con el `frame` actual

El modelo de la semana 3, en código

Regla de la semana 3	En el intérprete
Calcular argumentos en el marco de quien llama	`eval` con el `frame` actual
Crear un marco nuevo	`env_new(in->global)`

El modelo de la semana 3, en código

Regla de la semana 3	En el intérprete
Calcular argumentos en el marco de quien llama	`eval` con el `frame` actual
Crear un marco nuevo	`env_new(in->global)`
Cada parámetro recibe una copia, por posición	`env_set` de cada parámetro

El modelo de la semana 3, en código

Regla de la semana 3	En el intérprete
Calcular argumentos en el marco de quien llama	`eval` con el `frame` actual
Crear un marco nuevo	`env_new(in->global)`
Cada parámetro recibe una copia, por posición	`env_set` de cada parámetro
Buscar en el marco, y al final afuera. Nunca en otra llamada	`env_get`: marco y luego padres; el único padre es el global

El modelo de la semana 3, en código

Regla de la semana 3	En el intérprete
Calcular argumentos en el marco de quien llama	`eval` con el `frame` actual
Crear un marco nuevo	`env_new(in->global)`
Cada parámetro recibe una copia, por posición	`env_set` de cada parámetro
Buscar en el marco, y al final afuera. Nunca en otra llamada	`env_get`: marco y luego padres; el único padre es el global
Calcular el retorno, y el marco desaparece	`eval` del cuerpo, y `env_free`

El modelo de la semana 3, en código

Regla de la semana 3	En el intérprete
Calcular argumentos en el marco de quien llama	`eval` con el `frame` actual
Crear un marco nuevo	`env_new(in->global)`
Cada parámetro recibe una copia, por posición	`env_set` de cada parámetro
Buscar en el marco, y al final afuera. Nunca en otra llamada	`env_get`: marco y luego padres; el único padre es el global
Calcular el retorno, y el marco desaparece	`eval` del cuerpo, y `env_free`

Lo que dibujaban

Uno a uno. Lo que hacían a mano, ahora lo escribe `eval_call`.

Evaluar una llamada

pseudocódigo

```
algoritmo evaluar_llamada(f, argumentos, n, marco)
    valores ← arreglo nuevo de n números
    para i desde 0 hasta n - 1 hacer
        valores[i] ← evaluar(argumentos[i], marco)
    fin para
    nuevo ← marco vacío, con el marco global como padre
    para i desde 0 hasta n - 1 hacer
        definir el parámetro i de f en nuevo, con valores[i]
    fin para
    liberar valores
    resultado ← evaluar(el cuerpo de f, nuevo)
    liberar nuevo
    devolver resultado
fin algoritmo
```

Evaluar una llamada

pseudocódigo

```
algoritmo evaluar_llamada(f, argumentos, n, marco)
    valores ← arreglo nuevo de n números
    para i desde 0 hasta n - 1 hacer
        valores[i] ← evaluar(argumentos[i], marco)
    fin para
    nuevo ← marco vacío, con el marco global como padre
    para i desde 0 hasta n - 1 hacer
        definir el parámetro i de f en nuevo, con valores[i]
    fin para
    liberar valores
    resultado ← evaluar(el cuerpo de f, nuevo)
    liberar nuevo
    devolver resultado
fin algoritmo
```

Antes, dos revisiones: **la función existe** y **el número de argumentos coincide**.

Evaluar una llamada

pseudocódigo

```
algoritmo evaluar_llamada(f, argumentos, n, marco)
    valores ← arreglo nuevo de n números
    para i desde 0 hasta n - 1 hacer
        valores[i] ← evaluar(argumentos[i], marco)
    fin para
    nuevo ← marco vacío, con el marco global como padre
    para i desde 0 hasta n - 1 hacer
        definir el parámetro i de f en nuevo, con valores[i]
    fin para
    liberar valores
    resultado ← evaluar(el cuerpo de f, nuevo)
    liberar nuevo
    devolver resultado
fin algoritmo
```

Antes, dos revisiones: **la función existe** y **el número de argumentos coincide**.
Falta el manejo de errores: cada línea que falla libera lo que ya existe.

Los argumentos, en el marco de quien llama

Mini

```
def sub(a, b) = a - b
let a = 10
let b = 3
print sub(b, a)
```

Los argumentos, en el marco de quien llama

Mini

```
def sub(a, b) = a - b
let a = 10
let b = 3
print sub(b, a)
```

- Bien hecho: `b` y `a` son las globales, 3 y 10. Da -7.

Los argumentos, en el marco de quien llama

Mini

```
def sub(a, b) = a - b
let a = 10
let b = 3
print sub(b, a)
```

- Bien hecho: `b` y `a` son las globales, 3 y 10. Da -7.
- Si se evaluara cada argumento en el marco nuevo: `a` valdría 3 al calcular el segundo. Da 0.

Los argumentos, en el marco de quien llama

Mini

```
def sub(a, b) = a - b
let a = 10
let b = 3
print sub(b, a)
```

- Bien hecho: `b` y `a` son las globales, 3 y 10. Da -7.
- Si se evaluara cada argumento en el marco nuevo: `a` valdría 3 al calcular el segundo. Da 0.
- “Se llaman igual, y son otras casillas”: la advertencia de la semana 3.

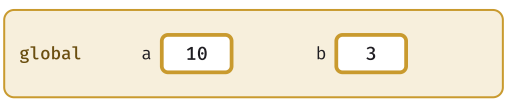
sub(b, a), paso a paso

```
print sub(b, a)
```

```
valores[i] ← evaluar(argumentos[i], marco)
```

VALORES





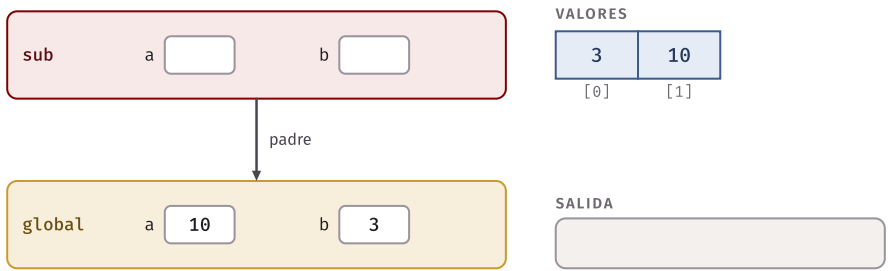
SALIDA

Los argumentos se calculan en el marco de quien llama: b vale 3, y a vale 10.

sub(b, a), paso a paso

```
print sub(b, a)
```

nuevo ← marco vacío, con el marco global como padre

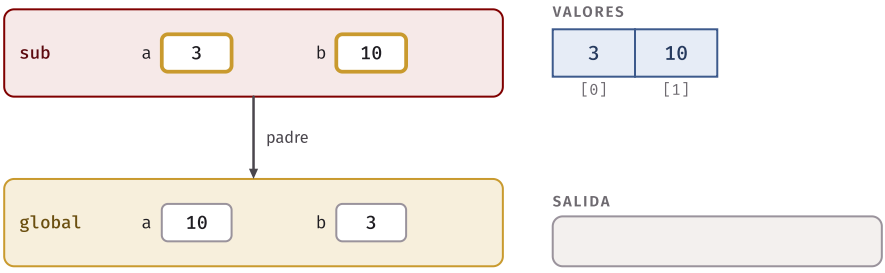


El marco nuevo cuelga del global: env_new(in->global).

sub(b, a), paso a paso

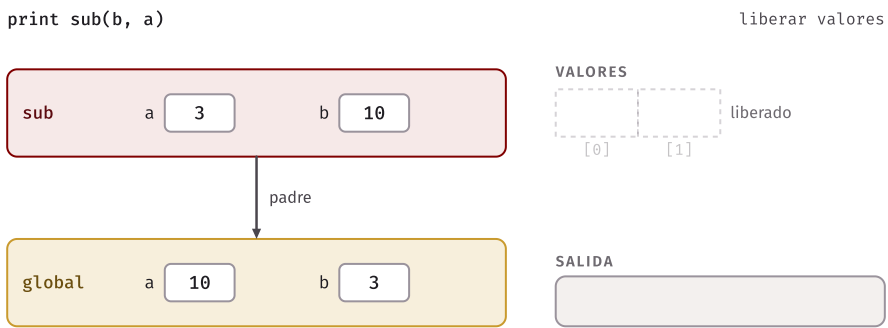
```
print sub(b, a)
```

definir el parámetro i de f en nuevo, con valores[i]



Cada parámetro recibe su valor, por posición: a vale 3, y b vale 10.

`sub(b, a)`, paso a paso

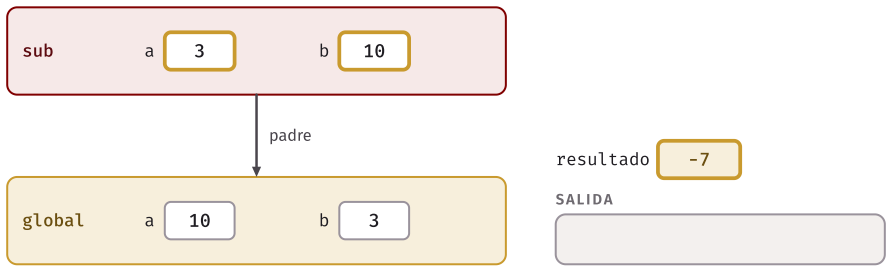


El arreglo ya cumplió, y se libera antes de correr el cuerpo.

sub(b, a), paso a paso

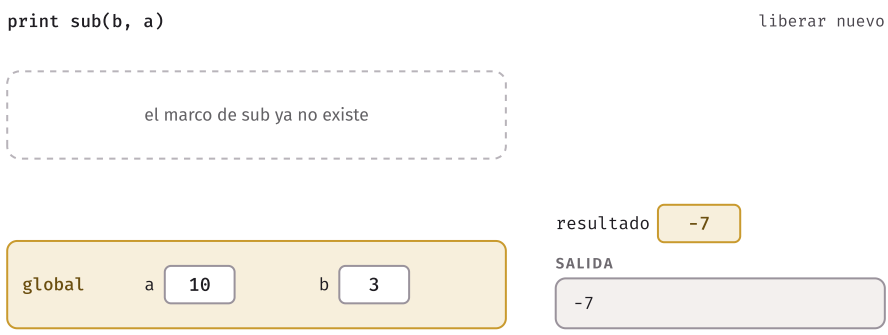
```
print sub(b, a)
```

```
resultado ← evaluar(el cuerpo de f, nuevo)
```



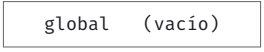
El cuerpo a - b busca en el marco nuevo: 3 - 10 da -7.

sub(b, a), paso a paso



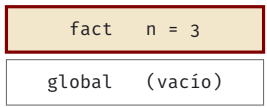
env_free destruye el marco. El a y el b globales nunca cambiaron, y print escribe -7.

Seguir `fact(3)`: el marco activo, arriba



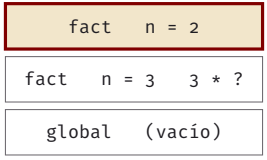
El `def` solo llena la tabla. El `print` corre en el marco global.

Seguir `fact(3)`: el marco activo, arriba



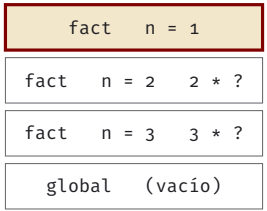
El argumento vale 3. Se crea un marco, con su `n`. Como 3 < 2 da 0, sigue la rama `n * fact(n - 1)`.

Seguir `fact(3)`: el marco activo, arriba



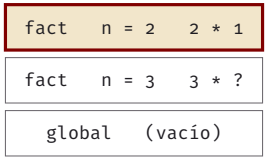
El argumento `n - 1` se calcula **en este marco**, y da 2. Nace otro marco, con otra `n`.

Seguir `fact(3)`: el marco activo, arriba



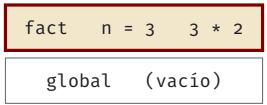
Tres `n` distintas conviven, y cada `eval` ve la suya porque recibe su `frame`. Aquí, 1 < 2 da 1.

Seguir `fact(3)`: el marco activo, arriba



Ese marco devolvió 1, y `env_free` lo destruyó. Ahora 2 * 1.

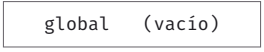
Seguir `fact(3)`: el marco activo, arriba



Ahora 3 * 2.

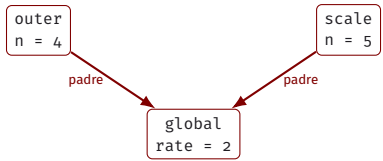
Seguir `fact(3)`: el marco activo, arriba

el `print` escribe 6

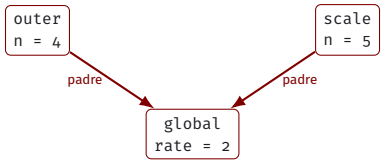


Como máximo hubo **tres marcos de `fact`**, además del global, que siempre está.

¿Por qué cuelga del global?

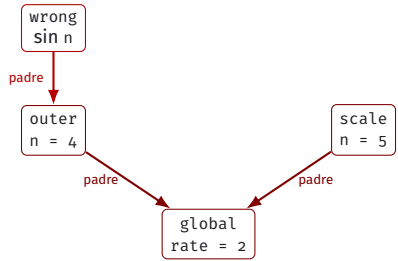


¿Por qué cuelga del global?



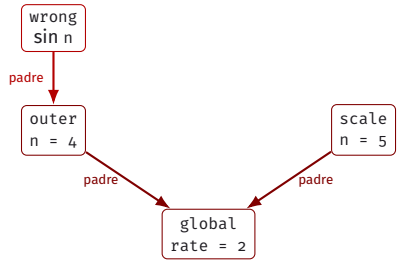
- `env_new(in->global)`: cada marco ve lo suyo y lo global. Es la regla 2 de la semana 3.

¿Por qué cuelga del global?



- `env_new(in->global)`: cada marco ve lo suyo y lo global. Es la regla 2 de la semana 3.
- `wrong` cuelga de quien llama, y encuentra una `n` que nunca definió.

¿Por qué cuelga del global?



- `env_new(in->global)`: cada marco ve lo suyo y lo global. Es la regla 2 de la semana 3.
- `wrong` cuelga de quien llama, y encuentra una `n` que nunca definió.
- Una función dependería de quién la llama. A eso se le llama alcance dinámico *(dynamic scope)*.

Con `Env`, a mano: `frames.c`

frames.c

```c
Env *global = env_new(NULL);
Env *outer = env_new(global);
Env *scale = env_new(global);       // cuelga del global, no de outer
Env *wrong = env_new(outer);        // lo que NO hace Mini: colgar del marco de quien llama
```

terminal

```
$ build/frames
global      rate  = 2
global      n     no existe
outer       n     = 4
outer       rate  = 2
scale       n     = 5
scale       rate  = 2
wrong       n     = 4
```

Y Mini lo cumple

lab/programs/caller_local.mini

```
def show() = n
def outer(n) = show()
print outer(7)
```

Y Mini lo cumple

lab/programs/caller_local.mini

```
def show() = n
def outer(n) = show()
print outer(7)
```

terminal

```
$ build/lab/mini lab/programs/caller_local.mini
lab/programs/caller_local.mini: error de ejecución: variable desconocida: n
```

Y Mini lo cumple

lab/programs/caller_local.mini

```
def show() = n
def outer(n) = show()
print outer(7)
```

terminal

```
$ build/lab/mini lab/programs/caller_local.mini
lab/programs/caller_local.mini: error de ejecución: variable desconocida: n
```

- `show` no tiene `n`, y el marco de `outer` no le sirve.

Y Mini lo cumple

lab/programs/caller_local.mini

```
def show() = n
def outer(n) = show()
print outer(7)
```

terminal

```
$ build/lab/mini lab/programs/caller_local.mini
lab/programs/caller_local.mini: error de ejecución: variable desconocida: n
```

- `show` no tiene `n`, y el marco de `outer` no le sirve.
- Es el error de `caller_local.c` en la semana 3: allá lo detectó `gcc`, y aquí aparece al correr.

Un parámetro con el nombre de una global

lab/programs/shadow.mini

```
let x = 10
def double(x) = x * 2
print double(3)
print x
```

terminal

```
$ build/lab/mini lab/programs/shadow.mini
6
10
```

lab/programs/globals.mini

```
let rate = 2
def scale(n) = n * rate
print scale(5)
let rate = 3
print scale(5)
```

terminal

```
$ build/lab/mini lab/programs/globals.mini
10
15
```

Un parámetro con el nombre de una global

lab/programs/shadow.mini

```
let x = 10
def double(x) = x * 2
print double(3)
print x
```

terminal

```
$ build/lab/mini lab/programs/shadow.mini
6
10
```

lab/programs/globals.mini

```
let rate = 2
def scale(n) = n * rate
print scale(5)
let rate = 3
print scale(5)
```

terminal

```
$ build/lab/mini lab/programs/globals.mini
10
15
```

- `env_get` busca primero en el marco, y `env_set` no toca el global: el parámetro la tapa.

Un parámetro con el nombre de una global

lab/programs/shadow.mini

```
let x = 10
def double(x) = x * 2
print double(3)
print x
```

terminal

```
$ build/lab/mini lab/programs/shadow.mini
6
10
```

lab/programs/globals.mini

```
let rate = 2
def scale(n) = n * rate
print scale(5)
let rate = 3
print scale(5)
```

terminal

```
$ build/lab/mini lab/programs/globals.mini
10
15
```

- `env_get` busca primero en el marco, y `env_set` no toca el global: el parámetro la tapa.
- Si se prohibiera, un `let` nuevo podría romper una función ya escrita.

Un parámetro con el nombre de una global

lab/programs/shadow.mini

```
let x = 10
def double(x) = x * 2
print double(3)
print x
```

terminal

```
$ build/lab/mini lab/programs/shadow.mini
6
10
```

lab/programs/globals.mini

```
let rate = 2
def scale(n) = n * rate
print scale(5)
let rate = 3
print scale(5)
```

terminal

```
$ build/lab/mini lab/programs/globals.mini
10
15
```

- `env_get` busca primero en el marco, y `env_set` no toca el global: el parámetro la tapa.
- Si se prohibiera, un `let` nuevo podría romper una función ya escrita.
- Una global se lee cuando el cuerpo corre, no cuando se define.

Programas de verdad, y %g

lab/programs/fib.mini

```
def fib(n) = if n < 2 then n else fib(n - 1) + fib(n - 2)
print fib(1)
print fib(10)
print fib(20)
```

terminal

```
$ build/lab/mini lab/programs/fib.mini
1
55
6765
```

terminal

```
$ build/lab/mini lab/programs/big.mini
720
3.6288e+06
6.22702e+09
```

Programas de verdad, y %g

lab/programs/fib.mini

```
def fib(n) = if n < 2 then n else fib(n - 1) + fib(n - 2)
print fib(1)
print fib(10)
print fib(20)
```

terminal

```
$ build/lab/mini lab/programs/fib.mini
1
55
6765
```

terminal

```
$ build/lab/mini lab/programs/big.mini
720
3.6288e+06
6.22702e+09
```

- `%g` usa 6 cifras significativas: `fact(13)` es 6227020800. El valor adentro sigue completo.

Programas de verdad, y `%g`

lab/programs/fib.mini

```
def fib(n) = if n < 2 then n else fib(n - 1) + fib(n - 2)
print fib(1)
print fib(10)
print fib(20)
```

terminal

```
$ build/lab/mini lab/programs/fib.mini
1
55
6765
```

terminal

```
$ build/lab/mini lab/programs/big.mini
720
3.6288e+06
6.22702e+09
```

- `%g` usa 6 cifras significativas: `fact(13)` es 6227020800. El valor adentro sigue completo.
- Todo es `double`: `0.1 + 0.2 == 0.3` también da 0 en Mini.

SECCIÓN 05

Dos pilas

05

La recursión del intérprete sostiene la del lenguaje

depende del sistema: extracto, en el caso base de fact(3)

```
#0  eval (in=..., e=..., frame=..., result=...) at interp.c:144
...
#3  eval_call (in=..., e=..., frame=..., result=...) at interp.c:138
...
#9  eval_call (in=..., e=..., frame=..., result=...) at interp.c:138
...
#15 eval_call (in=..., e=..., frame=..., result=...) at interp.c:138
#16 eval (in=..., e=..., frame=..., result=...) at interp.c:158
#17 exec_statement (in=..., s=...) at interp.c:176
#18 run_program (program=..., out=..., error=...) at interp.c:195
#19 main (argc=..., argv=...) at mini.c:53
```

La recursión del intérprete sostiene la del lenguaje

depende del sistema: extracto, en el caso base de fact(3)

```
#0  eval (in=..., e=..., frame=..., result=...) at interp.c:144
...
#3  eval_call (in=..., e=..., frame=..., result=...) at interp.c:138
...
#9  eval_call (in=..., e=..., frame=..., result=...) at interp.c:138
...
#15 eval_call (in=..., e=..., frame=..., result=...) at interp.c:138
#16 eval (in=..., e=..., frame=..., result=...) at interp.c:158
#17 exec_statement (in=..., s=...) at interp.c:176
#18 run_program (program=..., out=..., error=...) at interp.c:195
#19 main (argc=..., argv=...) at mini.c:53
```

- Tres veces `eval_call`, una por cada llamada de Mini viva.

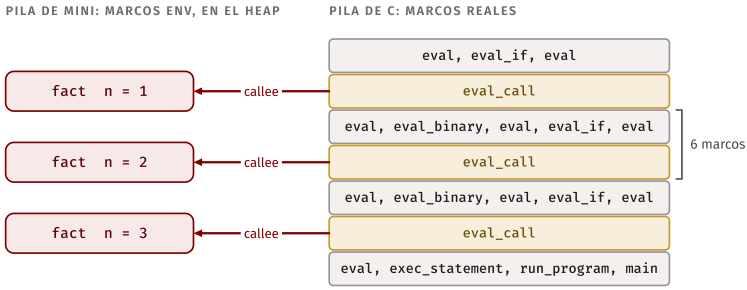
La recursión del intérprete sostiene la del lenguaje

depende del sistema: extracto, en el caso base de fact(3)

```
#0  eval (in=..., e=..., frame=..., result=...) at interp.c:144
...
#3  eval_call (in=..., e=..., frame=..., result=...) at interp.c:138
...
#9  eval_call (in=..., e=..., frame=..., result=...) at interp.c:138
...
#15 eval_call (in=..., e=..., frame=..., result=...) at interp.c:138
#16 eval (in=..., e=..., frame=..., result=...) at interp.c:158
#17 exec_statement (in=..., s=...) at interp.c:176
#18 run_program (program=..., out=..., error=...) at interp.c:195
#19 main (argc=..., argv=...) at mini.c:53
```

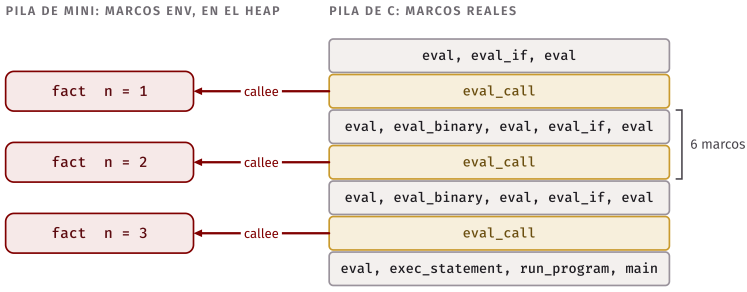
- Tres veces `eval_call`, una por cada llamada de Mini viva.
- No hay una “pila de Mini” escrita en ningún lado: `eval` llama a `eval_call`, y este a `eval`.

Dos pilas a la vez



Cada llamada de Mini es un eval_call, y entre una y la siguiente hay otros cinco marcos de C.

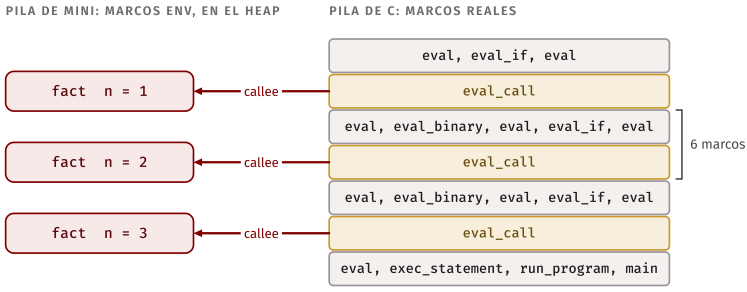
Dos pilas a la vez



Cada llamada de Mini es un eval_call, y entre una y la siguiente hay otros cinco marcos de C.

- Aquí, cada llamada de Mini cuesta seis marcos de C.

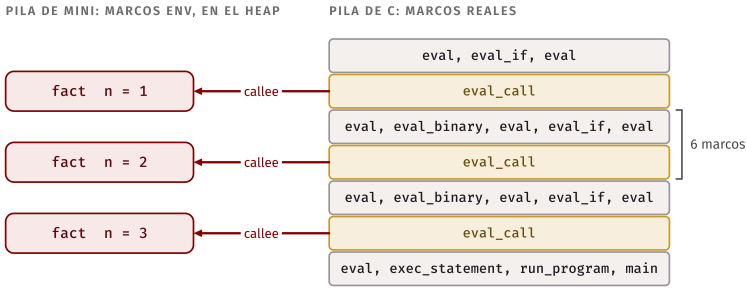
Dos pilas a la vez



Cada llamada de Mini es un eval_call, y entre una y la siguiente hay otros cinco marcos de C.

- Aquí, cada llamada de Mini cuesta seis marcos de C.
- El salto de fe de la semana 9: `eval_call` confía en que `eval` del cuerpo devuelve lo correcto.

Dos pilas a la vez



Cada llamada de Mini es un eval_call, y entre una y la siguiente hay otros cinco marcos de C.

- Aquí, cada llamada de Mini cuesta seis marcos de C.
- El salto de fe de la semana 9: `eval_call` confía en que `eval` del cuerpo devuelve lo correcto.
- Si se acaba la pila de C, se acaba también la de Mini.

Una recursión sin fin

lab/programs/runaway.mini

```
def forever(n) = forever(n + 1)
print forever(0)
```

extracto: las direcciones y las líneas cambian

```
==29==ERROR: AddressSanitizer: stack-overflow on address 0xffffebe09f90 ...
    #1 0xaaaac2707270 in env_get lab/env.c:67
    #2 0xaaaac270f374 in eval lab/interp.c:149
    #5 0xaaaac270ed8c in eval_call lab/interp.c:119
    #7 0xaaaac270f10c in eval_call lab/interp.c:138
```

Una recursión sin fin

lab/programs/runaway.mini

```
def forever(n) = forever(n + 1)
print forever(0)
```

extracto: las direcciones y las líneas cambian

```
==29==ERROR: AddressSanitizer: stack-overflow on address 0xffffebe09f90 ...
    #1 0xaaaac2707270 in env_get lab/env.c:67
    #2 0xaaaac270f374 in eval lab/interp.c:149
    #5 0xaaaac270ed8c in eval_call lab/interp.c:119
    #7 0xaaaac270f10c in eval_call lab/interp.c:138
```

- Lo que se repite es `eval_call` y `eval` en C, no `forever`.

Una recursión sin fin

lab/programs/runaway.mini

```
def forever(n) = forever(n + 1)
print forever(0)
```

extracto: las direcciones y las líneas cambian

```
==29==ERROR: AddressSanitizer: stack-overflow on address 0xffffebe09f90 ...
    #1 0xaaaac2707270 in env_get lab/env.c:67
    #2 0xaaaac270f374 in eval lab/interp.c:149
    #5 0xaaaac270ed8c in eval_call lab/interp.c:119
    #7 0xaaaac270f10c in eval_call lab/interp.c:138
```

- Lo que se repite es `eval_call` y `eval` en C, no `forever`.
- Se cae el programa de C entero. Sin sanitizers es `Segmentation fault`, estado 139.

Una recursión sin fin

lab/programs/runaway.mini

```
def forever(n) = forever(n + 1)
print forever(0)
```

extracto: las direcciones y las líneas cambian

```
==29==ERROR: AddressSanitizer: stack-overflow on address 0xffffebe09f90 ...
    #1 0xaaaac2707270 in env_get lab/env.c:67
    #2 0xaaaac270f374 in eval lab/interp.c:149
    #5 0xaaaac270ed8c in eval_call lab/interp.c:119
    #7 0xaaaac270f10c in eval_call lab/interp.c:138
```

- Lo que se repite es `eval_call` y `eval` en C, no `forever`.
- Se cae el programa de C entero. Sin sanitizers es `Segmentation fault`, estado 139.
- Medido aquí, con 8192 KB de pila: unas 22 000 llamadas con `-O0`, y 13 000 con ASan. Depende de la máquina.

SECCIÓN 06

Errores y pruebas

06

Cuatro errores de ejecución

Error	Dónde se detecta	Ejemplo
Variable desconocida	`eval`, si `env_get` da 0	`print prcie * 2`
Función desconocida	`eval_call`, antes de todo	`twice(4)` antes del `def`
Número de argumentos	`eval_call`, antes de los argumentos	`add(1)`
División entre cero	`eval_binary`	`inverse(0)`

Cuatro errores de ejecución

Error	Dónde se detecta	Ejemplo
Variable desconocida	`eval`, si `env_get` da 0	`print prcie * 2`
Función desconocida	`eval_call`, antes de todo	`twice(4)` antes del `def`
Número de argumentos	`eval_call`, antes de los argumentos	`add(1)`
División entre cero	`eval_binary`	`inverse(0)`

- El 0 sube por la recursión, y cada `eval_call` libera su marco antes de devolverlo.

Cuatro errores de ejecución

Error	Dónde se detecta	Ejemplo
Variable desconocida	`eval`, si `env_get` da 0	`print prcie * 2`
Función desconocida	`eval_call`, antes de todo	`twice(4)` antes del `def`
Número de argumentos	`eval_call`, antes de los argumentos	`add(1)`
División entre cero	`eval_binary`	`inverse(0)`

- El 0 sube por la recursión, y cada `eval_call` libera su marco antes de devolverlo.
- Con 51 marcos vivos no hay nada especial que hacer, si cada camino está bien.

Cuatro errores de ejecución

Error	Dónde se detecta	Ejemplo
Variable desconocida	`eval`, si `env_get` da 0	`print prcie * 2`
Función desconocida	`eval_call`, antes de todo	`twice(4)` antes del `def`
Número de argumentos	`eval_call`, antes de los argumentos	`add(1)`
División entre cero	`eval_binary`	`inverse(0)`

- El 0 sube por la recursión, y cada `eval_call` libera su marco antes de devolverlo.
- Con 51 marcos vivos no hay nada especial que hacer, si cada camino está bien.

La fuga se puede contar

Sin el `env_free`, `fact(5)` deja `in 15 allocation(s)`: 5 marcos, de 3 bloques: marco, entrada y nombre.

Un programa grande se prueba por niveles

Nivel	Prueba	Atrapa
Módulo	`expr`, `lexer`, `parse_expression`	Un error local
Interfaz	Forma de los árboles, `run_program`	Un acuerdo roto
Programa	Texto de Mini y su salida	Que todo junto funciona
Errores	`RunErrors`, `SyntaxErrors`	Los caminos que nadie recorre
Memoria	ASan y UBSan, en todas	Fugas y accesos inválidos

Un programa grande se prueba por niveles

Nivel	Prueba	Atrapa
Módulo	`expr`, `lexer`, `parse_expression`	Un error local
Interfaz	Forma de los árboles, `run_program`	Un acuerdo roto
Programa	Texto de Mini y su salida	Que todo junto funciona
Errores	`RunErrors`, `SyntaxErrors`	Los caminos que nadie recorre
Memoria	ASan y UBSan, en todas	Fugas y accesos inválidos

Una buena prueba busca el camino que nadie recorre

`print f(1, 2, 3, 4, 5, 6, 7, 8, 9, *)` falla justo después de que el arreglo creció, con árboles adentro.

¿Sus pruebas sirven? Rompan el programa

solo un programa que distinga atrapa el bug

```
# Atrapa un marco que cuelga de quien llama: rate debe ser la global, no el parámetro de outer.
let rate = 2
def scale(n) = n * rate
def outer(rate) = scale(rate)
print outer(10)
```

¿Sus pruebas sirven? Rompan el programa

solo un programa que distinga atrapa el bug

```
# Atrapa un marco que cuelga de quien llama: rate debe ser la global, no el parámetro de outer.
let rate = 2
def scale(n) = n * rate
def outer(rate) = scale(rate)
print outer(10)
```

- Cambien `env_new(in->global)` por `env_new((Env *) frame)`.

¿Sus pruebas sirven? Rompan el programa

solo un programa que distinga atrapa el bug

```
# Atrapa un marco que cuelga de quien llama: rate debe ser la global, no el parámetro de outer.
let rate = 2
def scale(n) = n * rate
def outer(rate) = scale(rate)
print outer(10)
```

- Cambien `env_new(in->global)` por `env_new((Env *) frame)`.
- `fact` y `fib` siguen dando lo mismo: no prueban esto.

¿Sus pruebas sirven? Rompan el programa

solo un programa que distinga atrapa el bug

```
# Atrapa un marco que cuelga de quien llama: rate debe ser la global, no el parámetro de outer.
let rate = 2
def scale(n) = n * rate
def outer(rate) = scale(rate)
print outer(10)
```

- Cambien `env_new(in->global)` por `env_new((Env *) frame)`.
- `fact` y `fib` siguen dando lo mismo: no prueban esto.
- Este programa da 20 bien hecho, y 100 con el bug.

¿Sus pruebas sirven? Rompan el programa

solo un programa que distinga atrapa el bug

```
# Atrapa un marco que cuelga de quien llama: rate debe ser la global, no el parámetro de outer.
let rate = 2
def scale(n) = n * rate
def outer(rate) = scale(rate)
print outer(10)
```

- Cambien `env_new(in->global)` por `env_new((Env *) frame)`.
- `fact` y `fib` siguen dando lo mismo: no prueban esto.
- Este programa da 20 bien hecho, y 100 con el bug.
- Escribir esa prueba obliga a entender qué distingue al programa correcto.

SECCIÓN 07

Repaso

07

Qué idea sirvió en cada parte (I)

Parte del intérprete	Idea	Sem.
Leer los módulos dados antes de extenderlos	Leer código	2
`eval_call`: argumentos, marco, parámetros, cuerpo	Modelo de entornos	3
`Env`, y parámetros que tapan a los globales	Alcance	3
`double *result`, y devolver 1 o 0	Paso por puntero	4
`const Expr *`, `const Env *`	`const` y punteros	4
`Expr`, `args`, `expr_binary` que se apropia	Ownership	5
El arreglo que crece	`realloc`	5
La tabla que toma prestados los `def`	Prestar, y quién vive más	5
`env.h`, `lexer.h`, `parser.h`, `interp.h`	Módulos	6
`Lexer` y `Env`, solo por su encabezado	Tipos opacos	6

Qué idea sirvió en cada parte (II)

Parte del intérprete	Idea	Sem.
`Stmt` y `next`	Bloques enlazados	6
`Expr` y `Stmt`, un `switch` por variante	Uniones etiquetadas	7
`eval`, `expr_free`, `expr_print`, el parser	Recursión, salto de fe	9
`stack-overflow` con una recursión de Mini	La pila no es infinita	9
`fib` en Mini	Recursión en árbol	10
`Interp` por parámetro	Contexto	12
Funciones que siguen la forma de los datos	Datos recursivos	13
Una función por regla	Análisis sintáctico	14

Qué idea sirvió en cada parte (II)

Parte del intérprete	Idea	Sem.
`Stmt` y `next`	Bloques enlazados	6
`Expr` y `Stmt`, un `switch` por variante	Uniones etiquetadas	7
`eval`, `expr_free`, `expr_print`, el parser	Recursión, salto de fe	9
`stack-overflow` con una recursión de Mini	La pila no es infinita	9
`fib` en Mini	Recursión en árbol	10
`Interp` por parámetro	Contexto	12
Funciones que siguen la forma de los datos	Datos recursivos	13
Una función por regla	Análisis sintáctico	14

No aparece: el backtracking de la semana 11: la gramática de Mini se decide con un token. De la semana 12 solo quedó el contexto.

Preguntas de repaso

1. En `def f(n) = if n < 2 then 1 else n * f(n - 1)`, ¿cuántos marcos de Mini existen durante `f(3)`?
2. ¿Qué pasa en el intérprete de C si un programa de Mini tiene una recursión sin fin?
3. ¿Qué cambiaría si un parámetro se llama igual que una variable global?
4. ¿Qué parte del curso usaron en el analizador léxico, en el parser y en el evaluador?

Preguntas de repaso

1. En `def f(n) = if n < 2 then 1 else n * f(n - 1)`, ¿cuántos marcos de Mini existen durante `f(3)`?
2. ¿Qué pasa en el intérprete de C si un programa de Mini tiene una recursión sin fin?
3. ¿Qué cambiaría si un parámetro se llama igual que una variable global?
4. ¿Qué parte del curso usaron en el analizador léxico, en el parser y en el evaluador?

> **Sin mirar**
>
> Háganlas primero en papel: la 1 con el dibujo de marcos, y la 4 con la tabla tapada.

SECCIÓN 08

Laboratorio

08

El laboratorio

Sesión 1: unir las partes

1. Leer, con preguntas. 15 min.
2. `let` y `print`: ya corre un archivo. 15 min.
3. `parse_if` y `eval_if`. 15 min.
4. `parse_def`, `parse_call`, `eval_call`. Antes, dibujar `fact(3)`. 45 min.

Sesión 2: probar y repasar

5. `make test-lab`, por grupos. 25 min.
6. Un programa propio y pruebas que atrapen dos bugs metidos a propósito. 30 min.
7. Repaso. 35 min.

Antes de empezar

Lean `lab/README.md`. `make test-lab` falla hasta que terminen, y una fuga también lo hace fallar.

Resumen

- Un programa es una **lista de instrucciones**, y cada una tiene sus árboles, con un owner para cada bloque.
- Cada función del parser libera lo que ya tiene si falla. Lo que entrega a un constructor ya no es suyo.
- Un `def` no evalúa su cuerpo: lo registra, y la tabla lo **toma prestado**.
- El `if` solo evalúa la rama elegida: por eso no se puede escribir como una función.
- **Evaluar una llamada es el modelo de entornos de la semana 3.**
- El marco nuevo cuelga del **global**: una función nunca ve lo local de quien la llama.
- La recursión de `eval` **sostiene** la de Mini. Si no hay fin, cae el intérprete entero.
- Un programa grande se prueba por niveles, y las pruebas se comprueban rompiéndolo.

Para la próxima clase

- Terminen el laboratorio: `make test-lab`, sin informes de ASan.
- Resuelvan los ejercicios marcados como repaso en los apuntes.
- Respondan las preguntas de la guía de la semana 15.
- La semana 16 es el examen final: cubre todo el curso, con énfasis en las semanas 9 a 15.

Antes de cerrar el curso

Abran `interp.c` y busquen una línea de cada semana. Si la encuentran, ya vieron cómo encaja todo.