

conceptos fundamentales de lenguajes imperativos

Paradigmas de la Programación

FaMAF 2015

capítulo 7.

(adicionales: 4.4. y 5.)

basado en filminas de [John Mitchell](#) y [Vitaly Shmatikov](#)

paradigma imperativo

- un paradigma de programación es una **configuración frecuente (y feliz)** de características de lenguajes de programación
 - el paradigma imperativo es el más antiguo y el que estuvo siempre más pegado a la máquina
 - tradicionalmente se ha opuesto al paradigma **funcional**, pero la mayor parte de lenguajes integran ideas de ambos paradigmas
- (ver 4.4, no entra en el examen pero es lindo)

un minilenguaje Turing-completo

- variables enteras, valores, operaciones
- asignación
- if
- Go To

conceptos fundamentales

- operación básica: **asignación**
 - la asignación tiene efectos secundarios: cambia el estado de la máquina!
- sentencias de **control** de flujo
 - condicionales y sin condición (GO TO), ramas, ciclos
- bloques, para obtener **referencias locales**
- **parametrización**

elementos básicos

- definiciones de **tipos**
- declaraciones de **variables** (normalmente, tipadas)
- expresiones y sentencias de **asignación**
- sentencias de **control de flujo** (normalmente, estructuradas)
- **alcance léxico** y bloques, para poder tener variables con referencias locales
- declaraciones y definiciones de **procedimientos** y **funciones** (bloques parametrizados)

elementos básicos

- definiciones de **tipos**
- **declaraciones de variables (normalmente, tipadas)**
- expresiones y sentencias de **asignación**
- sentencias de **control de flujo** (normalmente, estructuradas)
- **alcance léxico** y bloques, para poder tener variables con referencias locales
- declaraciones y definiciones de **procedimientos** y **funciones** (bloques parametrizados)

declaraciones de variables

- las declaraciones **tipadas** restringen los posibles valores de una variable en la ejecución del programa
 - jerarquía de tipos built-in o personalizada
 - inicialización
- uso de memoria: cuánto espacio de memoria reservar para cada tipo de variable?
 - C en 32-bit : char = 1 byte, short = 2 bytes, int = 4 bytes, char* = 4 bytes

ubicación y valores de variables

- al declarar una variable la estamos ligando a una **ubicación en memoria**, de forma que el nombre de la variable es el **identificador** de la ubicación en memoria
 - la ubicación puede ser global, en la pila o en el heap
- **l-valor**: ubicación en memoria (dirección de memoria)
- **r-valor**: valor que se guarda en la ubicación de memoria identificada por el l-valor
- asignación: $A \text{ (objetivo)} = B \text{ (expresión)}$
 - actualización destructiva: reescribe la ubicación de memoria identificada por A con el valor de la expresión B
 - qué pasa si la variable ocurre en ambos lados de la asignación?

semántica de copia vs. referencia

- **semántica de copia:** la expresión se evalúa a un valor, que se copia al objetivo
 - lenguajes imperativos
- **semántica de referencia:** la expresión se evalúa a un objeto, cuyo puntero se copia al objetivo
 - lenguajes orientados a objetos

elementos básicos

- definiciones de **tipos**
- declaraciones de **variables** (normalmente, tipadas)
- **expresiones y sentencias de asignación**
- sentencias de **control de flujo** (normalmente, estructuradas)
- **alcance léxico** y bloques, para poder tener variables con referencias locales
- declaraciones y definiciones de **procedimientos** y **funciones** (bloques parametrizados)

variables y asignación

- en la parte derecha de una asignación está el r-valor de la variable, en la parte izquierda está su l-valor

$x = x+1$ significa “obtenemos el r-valor que encontramos en el l-valor ligado al identificador de variable x , sumémosle 1, y guardemos el resultado en el l-valor ligado a x ”

cómo se expresa esto en semántica operacional?

- una expresión que no tenga un l-valor no puede aparecer en la parte izquierda de una asignación
- qué expresiones no tienen l-valor?
 - $1=x+1$, $++x++$
 - $a[1] = x+1$?
- el r-valor de un puntero es el l-valor de otra variable (el valor de un puntero es una dirección)
- las constantes sólo tienen r-valor
- las funciones sólo tienen l-valor

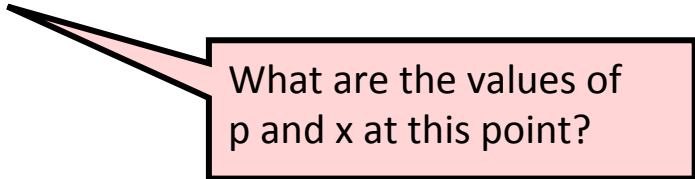
l-valores y r-valores (1)

- Any expression or assignment statement in an imperative language can be understood in terms of l-values and r-values of variables involved
 - In C, also helps with complex pointer dereferencing and pointer arithmetic
- Literal constants
 - Have r-values, but not l-values
- Variables
 - Have both r-values and l-values
 - Example: $x = x * y$ means "compute $rval(x) * rval(y)$ and store it in $lval(x)$ "

I-Values and r-Values (2)

- Pointer variables
 - Their r-values are l-values of another variable
 - Intuition: the value of a pointer is an address
- Overriding r-value and l-value computation in C
 - &x always returns l-value of x
 - *p always return r-value of p
 - If p is a pointer, this is an l-value of another variable

```
int x = 5; // lval(x) is some (stack) address, rval(x) == 5
int *p = &x // rval(p) == lval(x)
*p = 2 * x; // rval(p) <- rval(2) * rval(x)
```



What are the values of
p and x at this point?

l-Values and r-Values (3)

- Declared functions and procedures
 - Have l-values, but no r-values

```
int f(int y); // lval(f) is some global address
typedef int (*IFP)(int); // pointer to an int function that takes an int argument
IFP g = &f; // lval(g) <- lval(f)
(*g)(5);    // (rval(g)) == lval(f), so *g invokes f with argument rval(5)
            // the function call operator () has higher precedence than * so
            // we have to write (*g)(5) to deference g to invoke f(5)
```

elementos básicos

- definiciones de **tipos**
- declaraciones de **variables** (normalmente, tipadas)
- expresiones y sentencias de **asignación**
- **sentencias de control de flujo (normalmente, estructuradas)**
- **alcance léxico** y bloques, para poder tener variables con referencias locales
- declaraciones y definiciones de **procedimientos** y **funciones** (bloques parametrizados)

flujo de control estructurado

- se piensa como secuencial
 - las instrucciones se ejecutan en el orden en el que están escritas
 - en algunos casos soporta ejecución concurrente
- un programa es estructurado si el flujo de control es evidente en la estructura sintáctica del texto del programa
 - útil para poder razonar intuitivamente leyendo el texto del programa
 - se crean construcciones del lenguaje para patrones comunes de control: iteración, selección, procedimientos / funciones...

estructura de control en Fortran

```
10 IF (X .GT. 0.000001) GO TO 20
11 X = -X
    IF (X .LT. 0.000001) GO TO 50
20 IF (X*Y .LT. 0.000001) GO TO 30
    X = X-Y-Y
30 X = X+Y
    ...
50 CONTINUE
    X = A
    Y = B-A
    GO TO 11
    ...
```



código espagueti!

una estructura parecida puede aparecer en ensamblador

debate histórico sobre GO TO

- Dijkstra, “GO TO Statement Considered Harmful”
 - Letter to Editor, Comm. ACM, March 1968
- Knuth, “Structured Prog. with Go To Statements”
 - se puede usar goto, pero de forma estructurada

las reglas sintácticas fuerzan un buen estilo de programación? ayudan?

elementos básicos

- definiciones de **tipos**
- declaraciones de **variables** (normalmente, tipadas)
- expresiones y sentencias de **asignación**
- sentencias de **control de flujo** (normalmente, estructuradas)
- **alcance léxico y bloques, para poder tener variables con referencias locales**
- declaraciones y definiciones de **procedimientos** y **funciones** (bloques parametrizados)

estilo moderno

- construcciones estándar que estructuran los saltos

if ... then ... else ... end

while ... do ... end

for ... { ... }

case ...

- agrupan el código en bloques lógicos
- se evitan saltos explícitos (excepto retorno de función)
- no se puede saltar al medio de un bloque o función

iteración

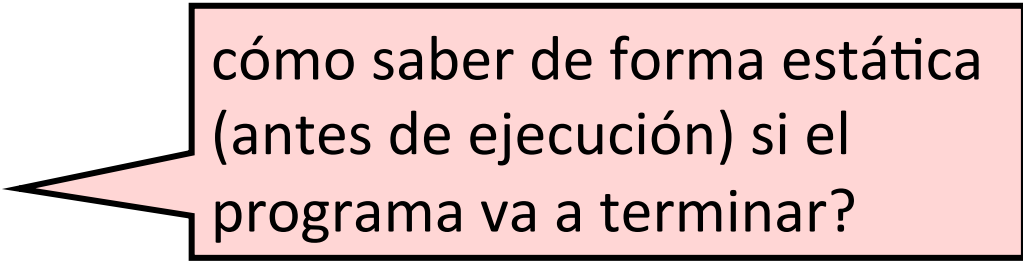
- Definida

```
for (int i = 0; i < 10; i++) {  
    a[i] = 0; // initialize each array element to zero  
}
```

- Indefinida

- la terminación depende de un valor dinámico (calculado en tiempo de ejecución)

```
int m = 0;  
while (n > 0) {  
    m = m * n;  
    n = n - 1;  
}
```



cómo saber de forma estática (antes de ejecución) si el programa va a terminar?

construcciones de iteración en C

- while (condition) stmt;
while (condition) { stmt; stmt; ...; }
- do stmt while (condition);
do { stmt; stmt; ...; } while (condition);
- for (<initialize>; <test>; <step>) stmt;
 - una forma restringida de “while”, lo mismo que
<initialize>; while (<test>) { stmt; <step> }
- for (<initialize>; <test>; <step>) { stmt;
stmt; ...; }
- foreach

“escapar” de un ciclo en C

```
int y; // y is in the "outer" scope
...
while (cond == true) {
    int x; // x is local to the while blocks scope (its extent and lifetime)
    ...
    if (x < y) { // special case...
        break; // leave while loop
    }
    ... // normal case
}
```

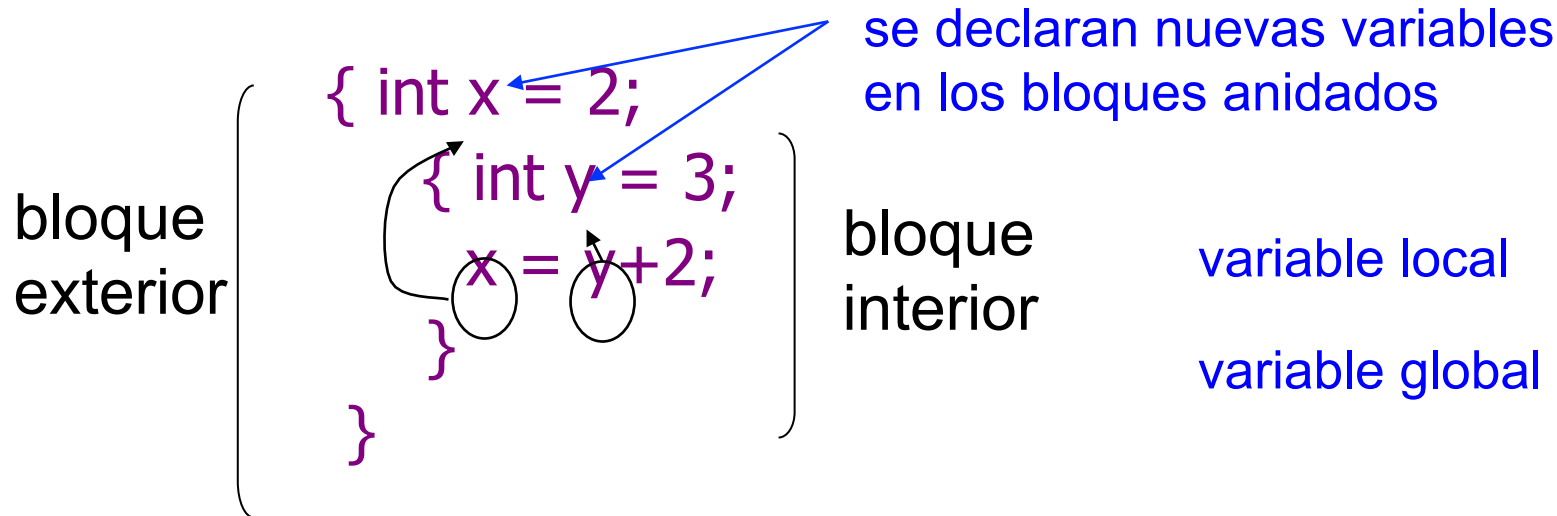
```
while (cond1 == true) {
    while (cond2 == true) {
        if (x < y) // special case
            break; // leave inner loop, but not outer loop
        ...
    }
    ... // control resumes here after a break from the inner loop
}
```

re-entrada forzada a un ciclo en C

```
while (cond-expr == true) {  
    ... // do something while cond is true  
    if (a == b) {  
        ... // do something special  
        continue; // transfer to start of while and re-evaluate cond  
    }  
    ... // remaining statements of while loop  
}
```


lenguajes con estructura de bloques

- bloques anidados con variables locales



– manejo de memoria

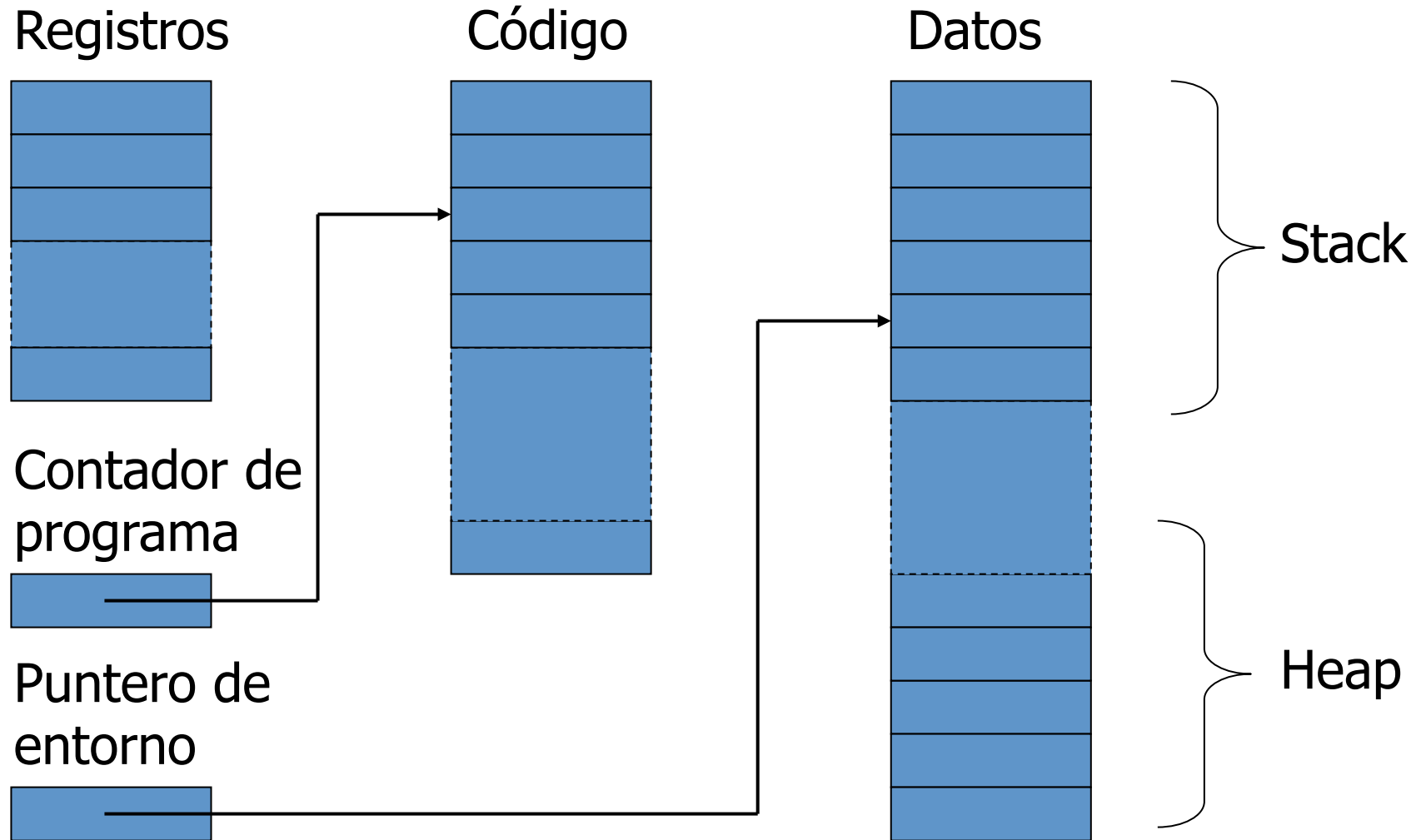
- al entrar al bloque reservamos espacio para variables
- al salir del bloque se puede liberar parte o todo el espacio

bloques en lenguajes comunes

- C, JavaScript * { ... }
- Algol begin ... end
- ML let ... in ... end
- dos formas de bloques
 - Inline
 - bloques asociados con funciones o procedimientos

* JavaScript functions provides blocks

modelo de máquina simplificado



manejo de memoria

- el **stack** tiene los datos sobre entrada y salida de bloques
- el **heap** tiene datos de diferente lifetime
- el **puntero de entorno** (*environment*) apunta a la posición actual en el stack
- al entrar a un bloque: se añade un nuevo activation record al stack
- al salir de un bloque: se elimina el activation record más reciente del stack

alcance y lifetime

- **alcance:** región del texto del programa donde una declaración es visible
- **lifetime:** período de tiempo en que una ubicación de memoria es asignada a un programa

```
{ int x = ... ;  
    { int y = ... ;  
        { int x = ... ;  
            ....  
        };  
    };  
};
```

ejemplo:

la declaración más interior de x tapa a la más exterior (“hueco en el alcance”)

el lifetime de la declaración exterior incluye el tiempo en el que el bloque interior se ejecuta

activation records

- para cada **bloque** se usa un activation record
 - **estructura de datos** que se guarda en la pila de ejecución
 - tiene lugar para **variables locales**

```
{ int x=0;  
  int y=x+1;  
    { int z=(x+y)*(x-y);  
      };  
};
```

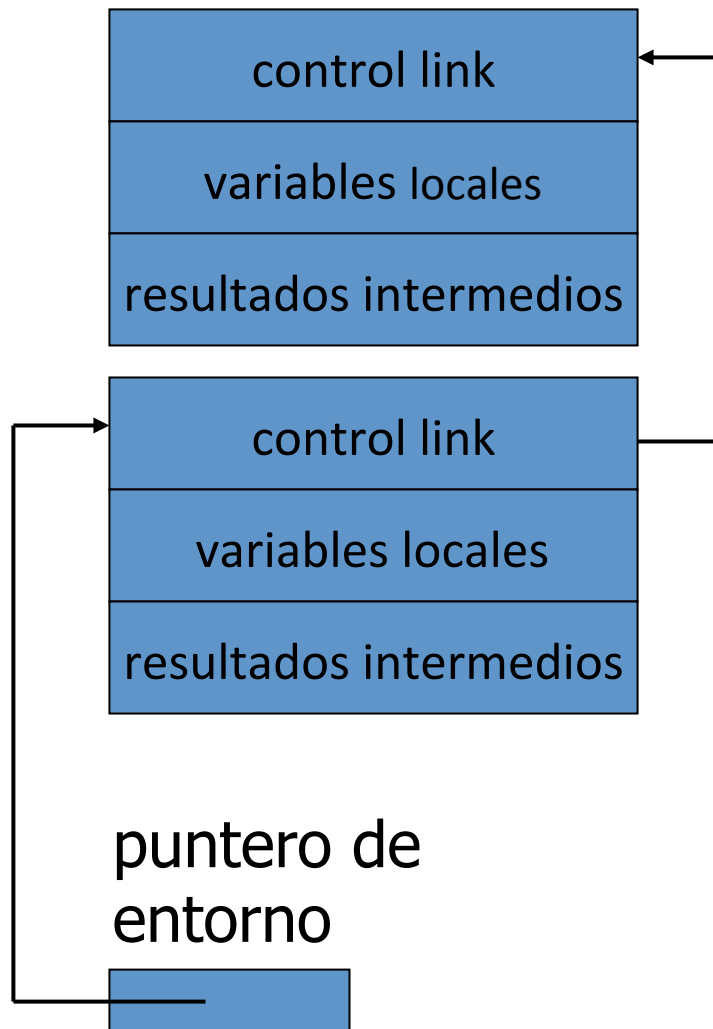
apilar record con espacio para x, y
fijar valores de x, y

apilar record para bloque interno
fijar valor de z

desapilar record para bloque interno
desapilar record para bloque externo

May need space for variables and intermediate results like $(x+y)$, $(x-y)$

Activation Record para bloque inline



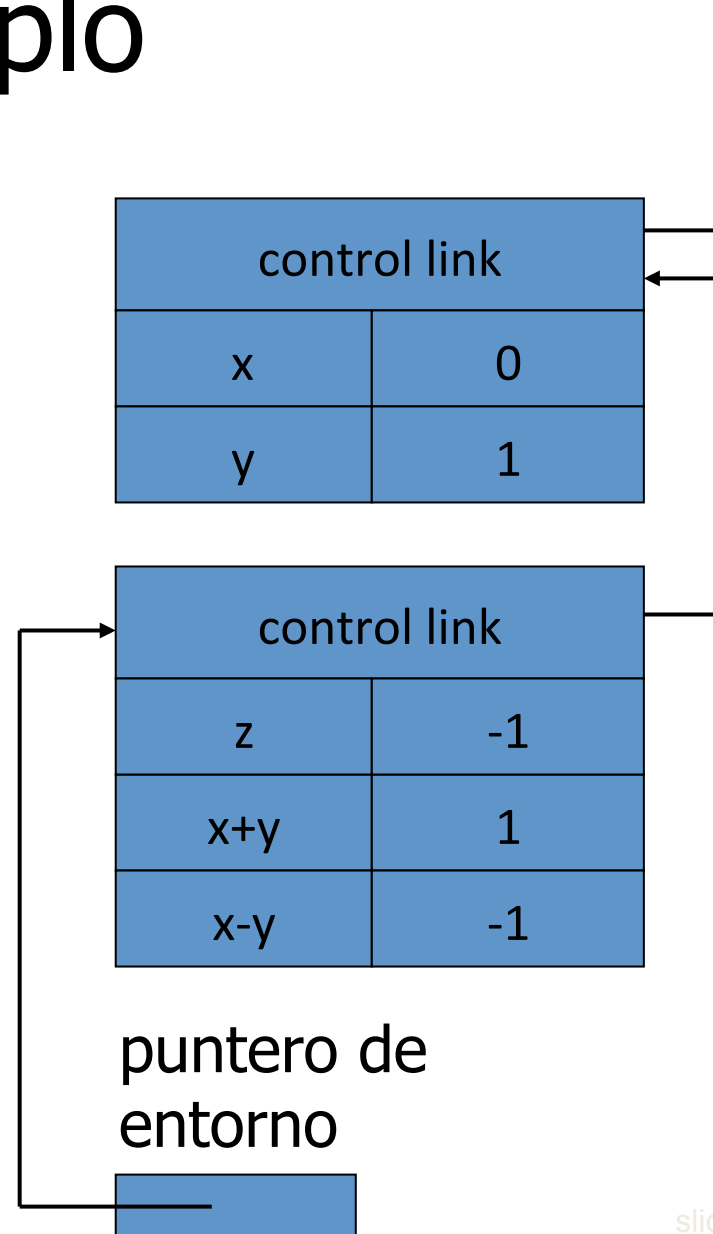
- **Control link**
 - puntero al record anterior en la pila
- **apilar**
 - fijar el nuevo control link para que apunte al viejo puntero de entorno
 - fijar el puntero de entorno al nuevo record
- **desapilar**

pila

 - seguir el control link del record actual para reasignar el puntero de entorno

ejemplo

```
{ int x=0;  
  int y=x+1;  
    { int z=(x+y)*(x-y);  
      };  
};
```



elementos básicos

- definiciones de **tipos**
- declaraciones de **variables** (normalmente, tipadas)
- expresiones y sentencias de **asignación**
- sentencias de **control de flujo** (normalmente, estructuradas)
- **alcance léxico** y bloques, para poder tener variables con referencias locales
- **declaraciones y definiciones de procedimientos y funciones (bloques parametrizados)**

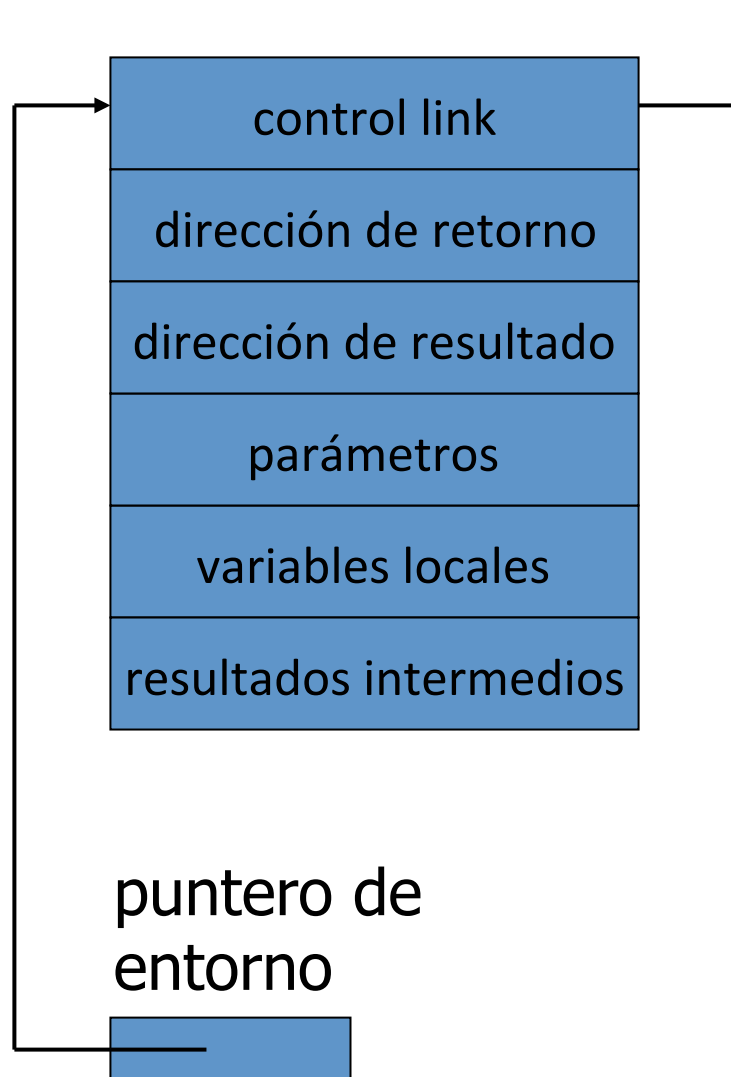
abstracción procedural

- un procedimiento es un **alcance parametrizado con nombre**
 - el programador se puede abstraer de los detalles de implementación, enfocándose en la interfaz
- funciones que **retornan valores**
$$x = (b*b - \text{sqrt}(4*a*c))/2*a$$
- funciones que **NO retornan valores**
 - “procedimientos” (Ada), “subrutinas” (Fortran), “funciones vacías / métodos” (C, C++, Java)
 - tienen **efectos secundarios visibles**, cambian el estado de algún valor de datos que no se define dentro de la función
$$\text{strcpy}(s1,s2)$$

activation records para funciones

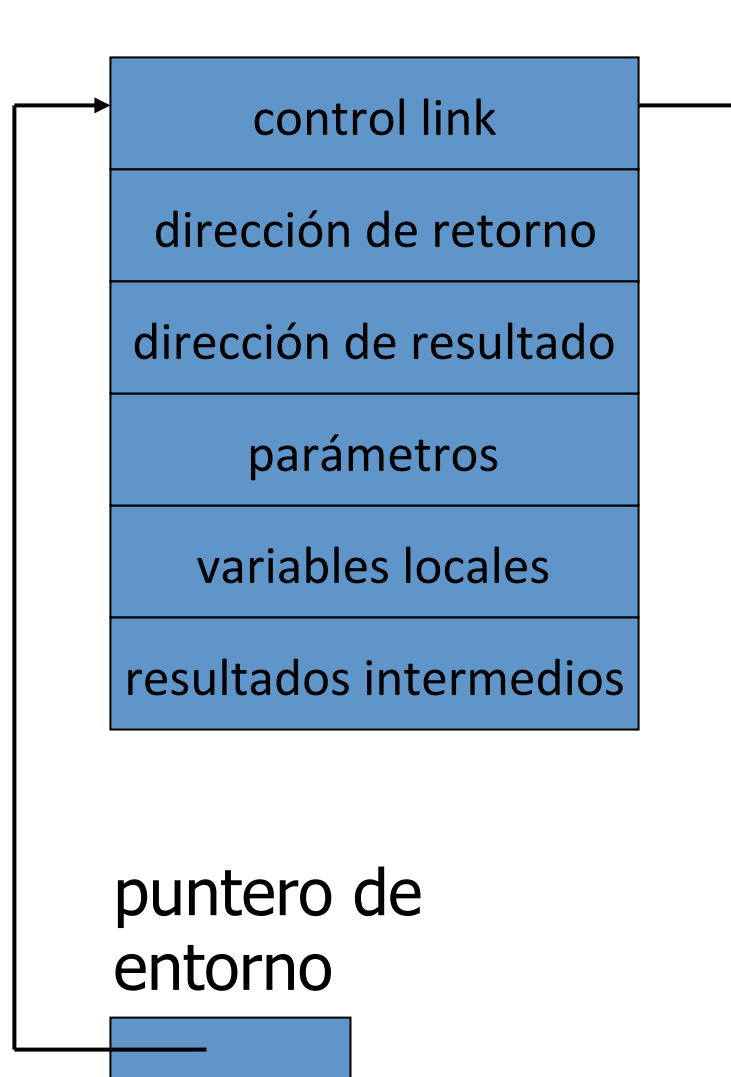
- información de bloque (“*frame*”) asociada a cada llamada de función:
 - parámetros
 - variables locales
 - dirección de retorno
 - ubicación para el valor de retorno al finalizar la función
 - control link al activation record de quien la llamó
 - registros guardados
 - variables temporales y resultados intermedios
 - (no siempre) access link al padre estático de la función

esquema de activation record



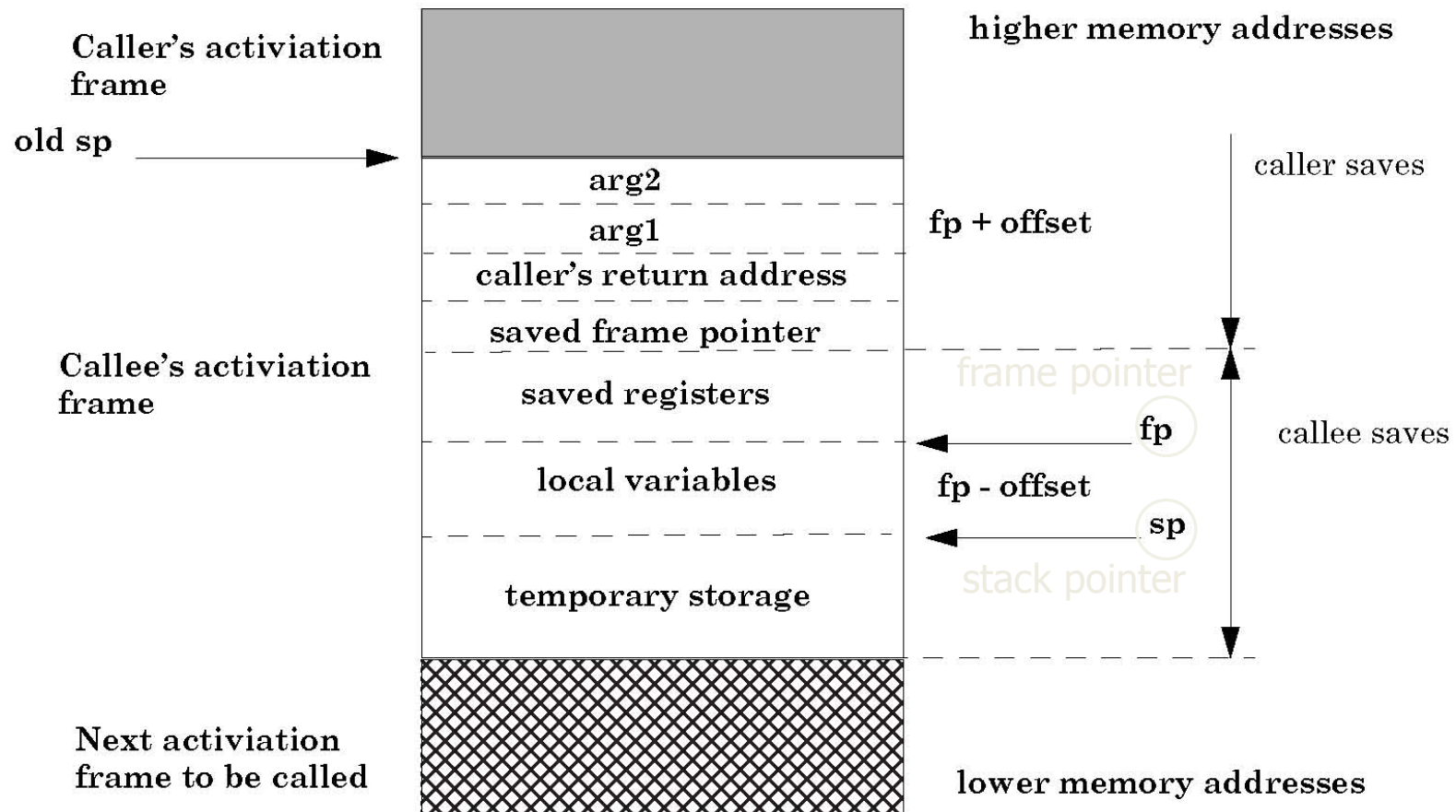
- dirección de retorno
 - ubicación del código a ejecutar cuando retorna la función
- dirección de resultado de retorno
 - dirección donde se guardará el valor que retorna la función, se encuentra en el activation record del bloque que llama a la función
- Parámetros
 - ubicaciones con los datos del bloque que llama a la función

ejemplo



- función
$$\text{fact}(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n * \text{fact}(n-1) & \text{else} \end{cases}$$
- dirección de resultado de retorno: ubicación donde poner $\text{fact}(n)$
- parámetro: el valor de n que asigna la secuencia de llamada
- resultados intermedios: ubicaciones con los valores de $\text{fact}(n-1)$

Typical x86 Activation Record



pila de ejecución

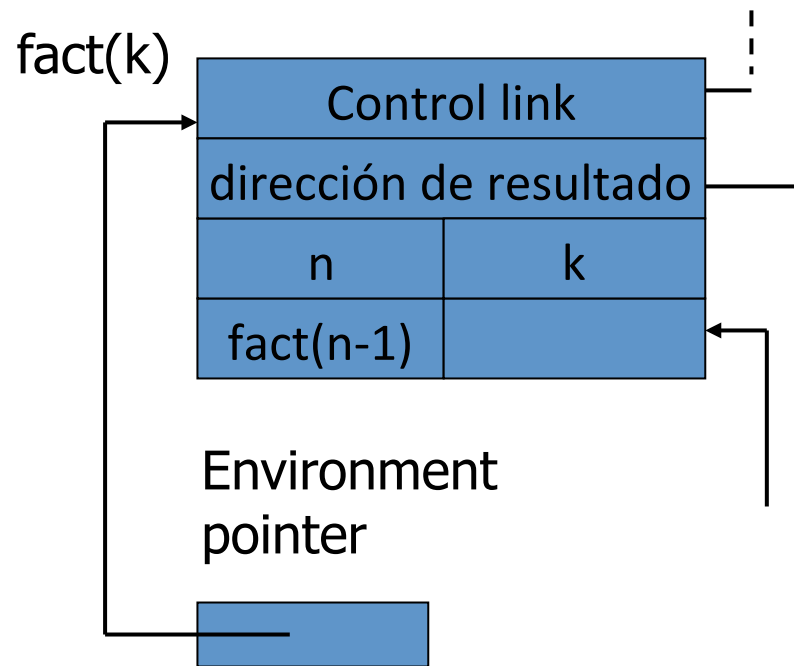
los registros de activación se guardan en la pila

- cada nueva llamada apila un activation record
- cada llamada finalizada desapila el activation record de la punta
- la pila tiene todos los records de todas las llamadas activas en un momento de la ejecución, siendo el record de la punta la llamada más reciente

ejemplo: fact(3)

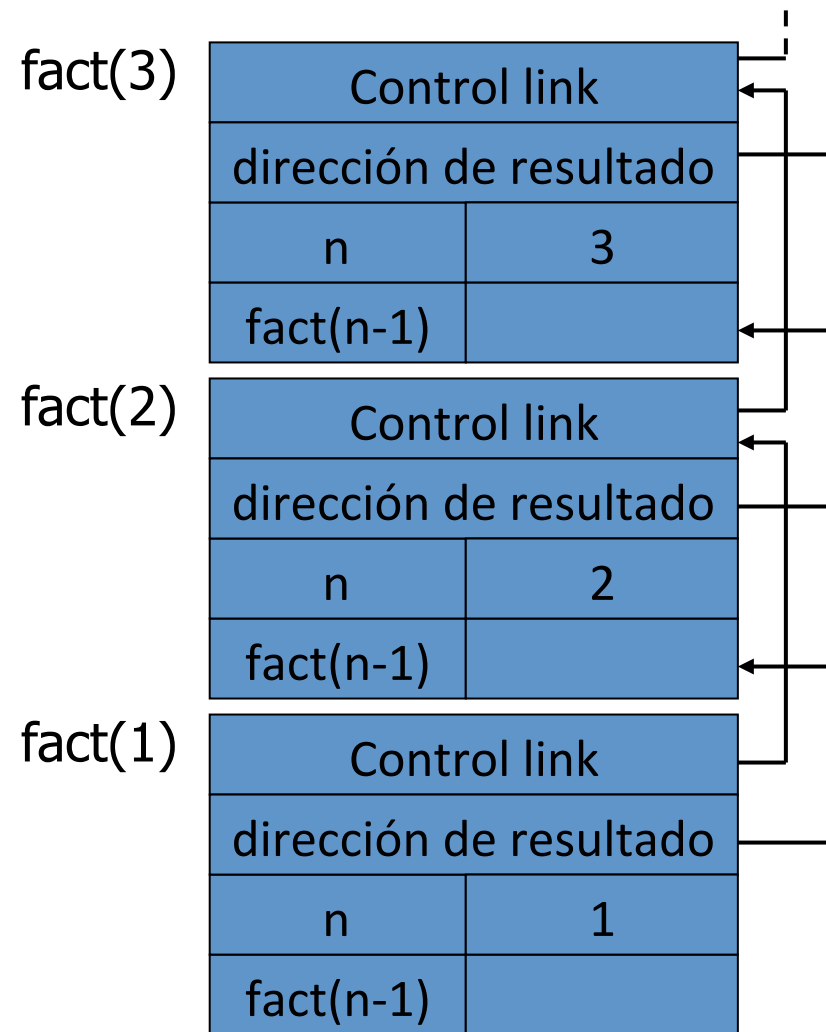
- apila un activation record a la pila, llama a fact(2)
- esta llamada apila otro record, y llama a fact(1)
- esta llamada apila otro record, lo cual resulta en tres activation records en la pila

llamada de función

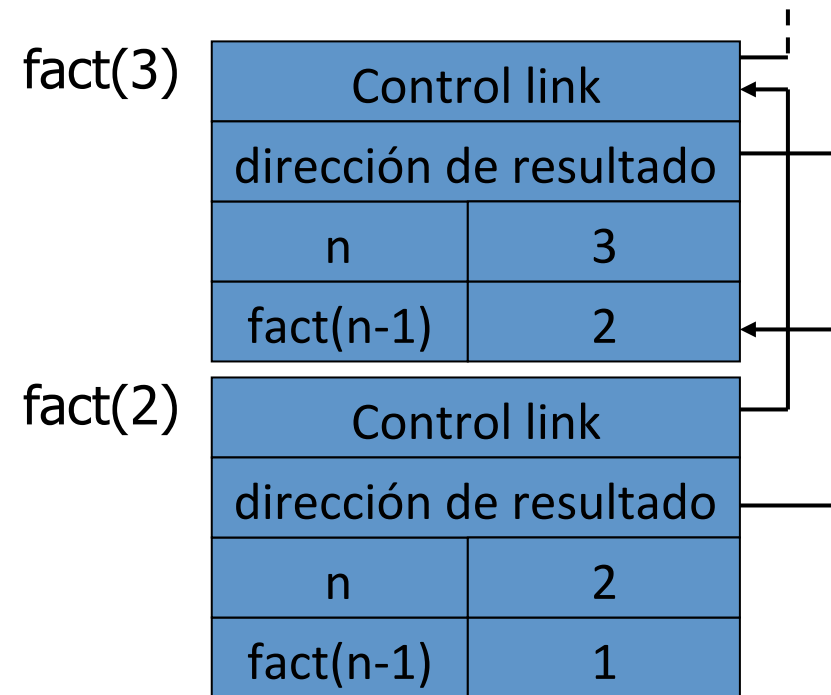
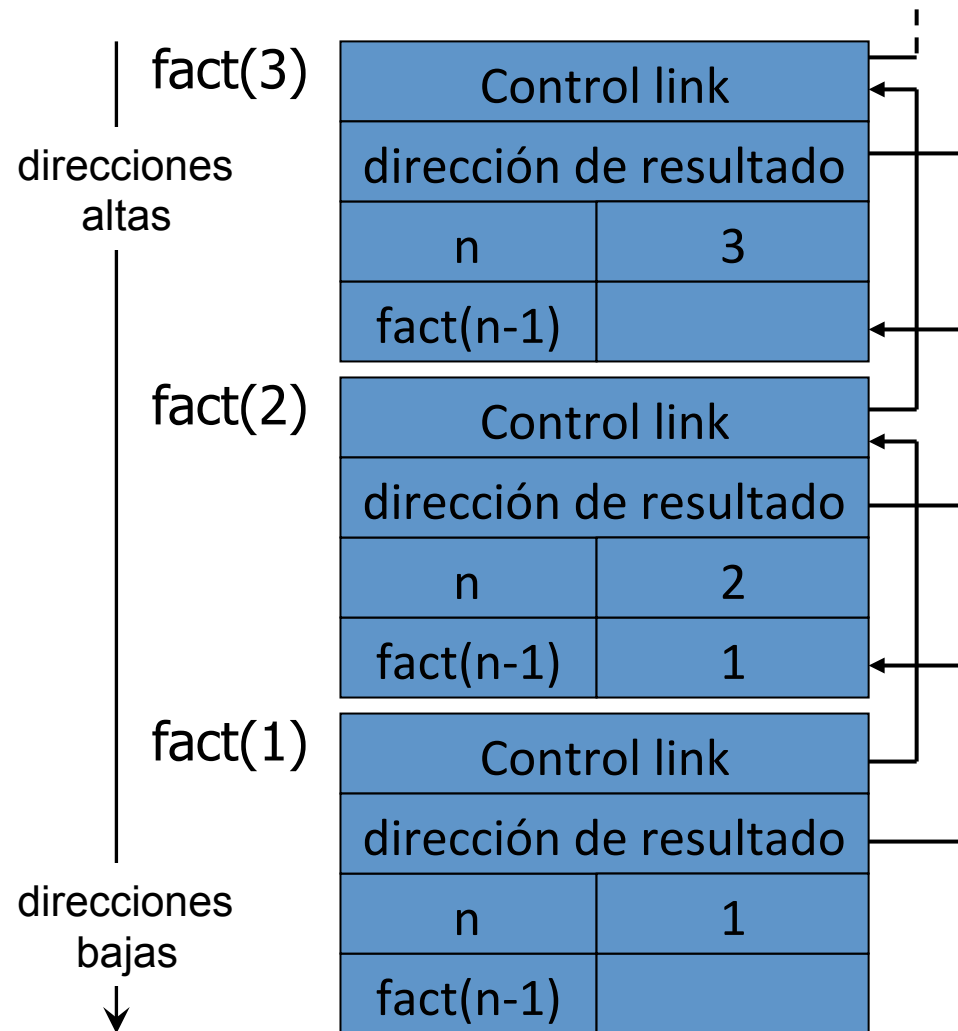


fact(n) = if $n \leq 1$ then 1
else $n * \text{fact}(n-1)$

omitimos la dirección de retorno



retorno de función



$\text{fact}(n) = \text{if } n \leq 1 \text{ then } 1$
 $\text{else } n * \text{fact}(n-1)$