SI1001 Teoría de la Computación
Verificación de programas

Andrés Sicard Ramírez

Universidad EAFIT

Semestre 2025-1

Esquema de la presentación

- Introducción
- Lógica de Hoare
- Frama-C y su plugin WP
- Referencias

Tema

- Introducción
- Lógica de Hoare
- Frama-C y su plugin WP
- Referencias

Cinco preguntas

En el prefacio de [Hei2017] el autor señala que la mayoría de los problemas en Ciencias de la Computación involucran una de las siguientes cinco preguntas:

(i) *Can the problem be solved by a computer program?*

(ii) *If not, can you modify the problem so that it can be solved by a program?*

(iii) *If so, can you write a program to solve the problem?*

(iv) *Can you convince another person that your program is correct?*

(v) *Can you convince another person that your program is efficient?*

Clasificación (paradigmas) de los lenguajes de programación

Clasificación

Adaptada de [Sco2015].

- Declarativos (qué)
  - Funcionales (Lisp, ML, Haskell, ...)
  - Lógicos, basados en restricciones (Prolog, CLP, ...)
- Imperativos (cómo)
  - von Neumann (Fortran, C, Ada, ...)
  - Orientados a objetos (Smalltalk, C++, Java, ...)
  - *Scripting* (Perl, Python, PHP, ...)

La arquitectura de von Neumann

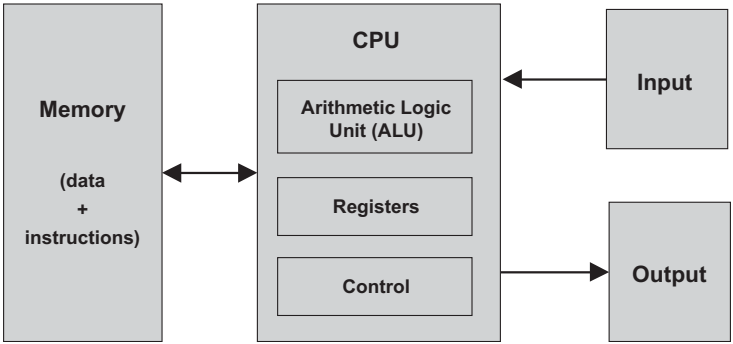


Figura 5.1 en [NS2005]

Programas en lenguajes de máquina

Ejemplo

Programa en lenguaje de máquina que adiciona los números 5 y 6. Figura 1.2 en [LL2011].

```
0010001000000100
0010010000000100
0001011001000010
0011011000000011
1111000000100101
0000000000000101
0000000000000110
```

Programas en lenguajes ensambladores

Ejemplo

Programa en lenguaje ensamblador que adiciona los números en las variables `FIRST` y `SECOND` y almacena el resultado en la variable `SUM`. Figura 1.3 en [LL2011].

```
.ORIG x3000       ; Address (in hexadecimal) of the first instruction
LD  R1, FIRST     ; Copy the number in memory location FIRST to register R1
LD  R2, SECOND    ; Copy the number in memory location SECOND to register R2
ADD R3, R2, R1    ; Add the numbers in R1 and R2 and place the sum in
                  ; register R3
ST  R3, SUM       ; Copy the number in R3 to memory location SUM
HALT              ; Halt the program
FIRST   .FILL #5  ; Location FIRST contains decimal 5
SECOND  .FILL #6  ; Location SECOND contains decimal 6
SUM     .BLKW #1  ; Location SUM (contains 0 by default)
.END              ; End of program
```

Programas en lenguajes imperativos

Ejemplo

Programa en lenguaje C que adiciona los números en las variables `first` y `second` y almacena el resultado en la variable `sum`.

```
int first = 5;
int second = 6;
sum = first + second;
```

Programas en lenguajes imperativos

Descripción

Un programa en un lenguaje imperativo es una secuencia de instrucciones que representan comandos:

- instrucciones de asignación
- instrucciones de secuenciación
- instrucciones *if-then-else*
- instrucciones *while*

Razonamiento sobre programas

Categorías

La investigación acerca el razonamiento sobre programas se puede dividir en cuatro categorías [Ach+2010]:

(i) definir la semántica (nuevos conceptos)

(ii) razonamiento sobre transformaciones de programas

(iii) verificación (formal) de propiedades funcionales (correspondencia de entrada y salida)

(iv) verificación (formal) de propiedades no funcionales (consumo de memoria, rendimiento paralelo, etc.)

Correctitud parcial y correctitud total

Al establecer la correctitud funcional (correspondencia de entrada y salida) de un programa se distingue entre:

- Correctitud parcial: El programa es correcto respecto a su especificación
- Correctitud total: Correctitud parcial + terminación

Tema

- Introducción
- Lógica de Hoare
- Frama-C y su plugin WP
- Referencias

Bibliografía

- Mike Gordon. «Background Reading on Hoare Logic». 2016. URL: https://www.cl.cam.ac.uk/archive/mjcg/HL/.

Un pequeño lenguaje de programación imperativo (IMP)

Expresiones aritméticas:

$E ::= N$	(números)
$\mid V$	(variables)
$\mid E_1$ + E_2	(adición)
$\mid E_1$ * E_2	(multiplicación)

Instrucciones:

$I ::= V$:= E	(asignación)
$\mid I_1$; I_2	(secuenciación)
$\mid$ `if` B `then` I_1 `else` I_2	(condicional)
$\mid$ `while` B `do` I	(while)

Expresiones booleanas:

$B ::=$ `true` $\mid$ `false`	(constantes)
$\mid E_1$ == E_2	(igualdad)
$\mid E_1$ <= E_2	(menor o igual)

Tripletas de Hoare

Definición

La **tripletas de Hoare** son de la forma

$$\{P\}\ I\ \{Q\},$$

donde

- I es un programa en IMP,
- P y Q son condiciones (fórmulas de la lógica de predicados de primer orden con identidad) en las variables del programa,
- P es la **precondición**,
- Q es la **postcondición**.

Tripletas de Hoare

Notación

Para escribir las condiciones P y Q usaremos las siguientes constantes lógicas: $\wedge$ (conjunción), $\vee$ (disyunción), $\supset$ (condicional), $\bot$ (falso) y $\top$ (verdadero).

Tripletas de Hoare

Ejemplo

Para la tripleta de Hoare

$$\{x = 1\}\ \texttt{x := x + 1}\ \{x = 2\}$$

- P es la precondición que el valor de x es 1,
- Q is la postcondición que el valor de x es 2,
- I es la instrucción de asignación `x := x + 1`.

Especificaciones de correctitud parcial

Definición

Una tripleta de Hoare $\{P\}\ I\ \{Q\}$ es una **especificación de correctitud parcial**.

Especificaciones de correctitud parcial

Definición

Una tripleta de Hoare $\{P\}\ I\ \{Q\}$ es una **especificación de correctitud parcial**.

Semántica

La especificación de correctitud parcial $\{P\}\ I\ \{Q\}$ es **verdadera** sii cuando el programa I es ejecutado en un estado que satisface la precondición P y si la ejecución del programa I termina, entonces el estado en el cual el programa I termina satisface la postcondición Q.

Especificaciones de correctitud parcial

Ejemplos

- $\{x = 1\}$ `x := x + 1` $\{x = 2\}$ es verdadera.
- $\{x = 1\}$ `x := x + 1` $\{x = 3\}$ es falsa.

Especificaciones de correctitud parcial

Observación

$\{P\}\ I\ \{Q\}$ es una especificación de correctitud parcial porque no es necesario que el programa I termine para que la especificación sea verdadera. Solo es necesario que si el programa I termina entonces la poscondición Q sea satisfecha.

Especificaciones de correctitud parcial

Ejemplos

- $\{x = 1\}$ `while true x := 42` $\{x = 2\}$ es verdadera.
- $\{\bot\}\ I\ \{Q\}$ es verdadera para todo I y Q.
- $\{P\}\ I\ \{\top\}$ es verdadera para todo I y P.

Lógica de Hoare y especificaciones de correctitud parcial

Descripción

La lógica de Hoare es un sistema formal deductivo para la tripletas de Hoare $\{P\}\ I\ \{Q\}$.

Axioma de asignación

$$\frac{}{\vdash \{Q[\,x \mapsto E\,]\}\ \texttt{x := } E\ \{Q\},}$$

donde $Q[\,x \mapsto E\,]$ denota el resultado de substituir cada ocurrencia libre de la variable x por la expresión E en la condición Q.

Lógica de Hoare

Axioma de asignación

$$\frac{}{\vdash \{Q[\,x \mapsto E\,]\}\ \texttt{x := } E\ \{Q\},}$$

donde $Q[\,x \mapsto E\,]$ denota el resultado de substituir cada ocurrencia libre de la variable x por la expresión E en la condición Q.

Ejemplos (instancias del axioma de asignación)

- $\vdash \{y = 2\}\ \texttt{x := 2}\ \{y = x\}$.

Lógica de Hoare

Axioma de asignación

$$\frac{}{\vdash \{Q[\,x \mapsto E\,]\}\ \texttt{x := } E\ \{Q\},}$$

donde $Q[\,x \mapsto E\,]$ denota el resultado de substituir cada ocurrencia libre de la variable x por la expresión E en la condición Q.

Ejemplos (instancias del axioma de asignación)

- $\vdash \{y = 2\}$ `x := 2` $\{y = x\}$.
- $\vdash \{y = x\}$ `z := y` $\{z = x\}$.

Regla de inferencia: fortalecimiento de una precondición

$$\frac{\vdash P \supset P' \qquad \vdash \{P'\}\ I\ \{Q\}}{\vdash \{P\}\ I\ \{Q\}.}$$

Lógica de Hoare

Regla de inferencia: fortalecimiento de una precondición

$$\frac{\vdash P \supset P' \qquad \vdash \{P'\}\ I\ \{Q\}}{\vdash \{P\}\ I\ \{Q\}.}$$

Regla de inferencia: debilitamiento de una postcondición

$$\frac{\vdash \{P\}\ I\ \{Q'\} \qquad \vdash Q' \supset Q}{\vdash \{P\}\ I\ \{Q\}.}$$

Ejemplo (demostración formal)

Demostremos que $\vdash \{r = x\}$ `q := 0` $\{r = x + (y \times q)\}$.

Ejemplo (demostración formal)

Demostremos que $\vdash \{r = x\}$ `q := 0` $\{r = x + (y \times q)\}$.

Demostración.

1. $\vdash r = x \supset r = x \wedge 0 = 0$ (por lógica)
2. $\vdash \{r = x \wedge 0 = 0\}$ `q := 0` $\{r = x \wedge q = 0\}$ (axioma de asignación)
3. $\vdash \{r = x\}$ `q := 0` $\{r = x \wedge q = 0\}$ (fortalecimiento de precondición en 1 y 2)
4. $\vdash r = x \wedge q = 0 \supset r = x + (y \times q)$ (por aritmética)
5. $\vdash \{r = x\}$ `q := 0` $\{r = x + (y \times q)\}$ (debilitamiento de postcondición en 3 y 4)

Regla de inferencia: secuenciación

$$\frac{\vdash \{P\}\ I_1\ \{Q\} \qquad \vdash \{Q\}\ I_2\ \{R\}}{\vdash \{P\}\ I_1\ ;\ I_2\ \{R\}.}$$

Tema

- Introducción
- Lógica de Hoare
- Frama-C y su plugin WP
- Referencias

Preliminares

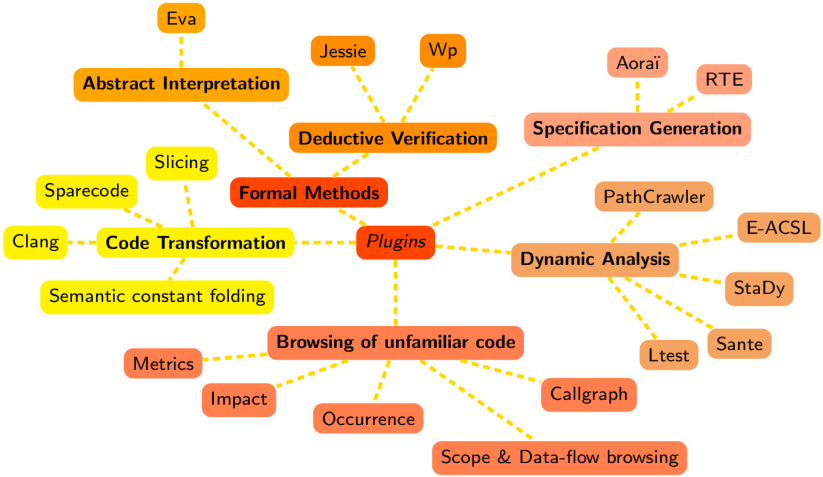
- La bibliografía para esta sección es [Bla2024].
- La numeración (de secciones, definiciones, teoremas, figuras, páginas, etc.) en esta sección corresponde a la numeración en [Bla2024].

Frama-C y su plugin WP

Descripción

- Frama-C (*FRAmework for Modular Analysis of* C *code*) es un plataforma para análisis de programas escritos en C creada por la CEA LIST y el Inria.
- ACSL (*ANSI/ISO* C *Specification Language*) (pronunciado «Axel») es el lenguaje de especificación para ANSI/ISO C usado por Frama-C.
- WP (*Weakest Precondition*) es un plugin para realizar verificación formal de programas en Frama-C.

Frama-C y su plugin WP



(Figura [pág. 13])

Contratos para funciones

*«The goal of a **function contract** is to state the properties of the input that are expected by the function, and in exchange the properties that will be assured for the output.»* [pág. 22]

Los contratos para funciones están compuestos de **pre-condiciones** y **post-condiciones**, las cuales se escriben en ACSL.

Ejemplo

Calcular el valor absoluto de un número entero (archivo `src/frama-c/abs-1.c`).

```
1  int abs(int val) {
2    if ( val < 0 ) return -val;
3    return val;
4  }
```

(continua en la próxima diapositiva)

Ejemplo (continuación)

Adicionamos una primera post-condición escrita en ACSL (archivo `src/frama-c/abs-2.c`).

```
1  /*@
2    ensures \result >= 0;
3  */
4  int abs(int val) {
5    if ( val < 0 ) return -val;
6    return val;
7  }
```

(continua en la próxima diapositiva)

Ejemplo (continuación)

Adicionamos una segunda post-condición escrita en ACSL (archivo `src/frama-c/abs-3.c`).

```
1  /*@
2    ensures \result >= 0;
3    ensures (val >= 0 ==> \result == val) && (val < 0 ==> \result == -val);
4  */
5  int abs(int val) {
6    if ( val < 0 ) return -val;
7    return val;
8  }
```

(continua en la próxima diapositiva)

Ejemplo (continuación)

Ejecutamos las instrucciones

```
$ cd src/frama-c
$ frama-c-gui abs-3.c
```

y demostramos las post-condiciones utilizando el plugin WP.

(continua en la próxima diapositiva)

Ejemplo (continuación)

¿Es nuestro programa libre de errores? ¿Qué acerca de errores en tiempo de ejecución?

Ejemplo (continuación)

¿Es nuestro programa libre de errores? ¿Qué acerca de errores en tiempo de ejecución?

El pluging RTE (*runtime errors*) adiciona anotaciones al programa para evitar errores en tiempo de ejecución (*integer overflow*, *invalid pointer dereferencing*, división por cero, etc.)

(continua en la próxima diapositiva)

Ejemplo (continuación)

La anotación adicionada por el pluging RTE es debido a la representación de números en el computador. Como es señalado por *The GNU C Library Reference Manual* para la versión 2.36:

> *Most computers use a two's complement integer representation, in which the absolute value of* `INT_MIN` *(the smallest possible* **`int`** *) cannot be represented; thus,* `abs (INT_MIN)` *is not defined.*

(continua en la próxima diapositiva)

Ejemplo (continuación)

Si un número entero es representado con n bits usando la representación de complemento a dos los valores que se pueden representar están en el intervalo $[-2^{n-1}, 2^{n-1} - 1]$:

- Un `int` de C en GCC es representado con 32 bits (en mi computador), es decir, los valores que se pueden representar son $[-2^{31}, 2^{31} - 1] = [-2147483648, 2147483647]$.
- Un `Int` de HASKELL en GHC es representado con 64 bits (en mi computador), es decir, los valores que se pueden representar son $[-2^{63}, 2^{63} - 1] = [-9223372036854775808, -9223372036854775807]$.

(continua en la próxima diapositiva)

Ejemplo (continuación)

Programa que muestra el problema empleando la función `abs` de la *GNU C library* (archivo `src/glibc.c`).

```
1  #include <limits.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main() {
6    printf("Maximum int (INT_MAX, 2^31 - 1): %d\n", INT_MAX);
7    printf("Minimum int (INT_MIN, -2^31): %d\n", INT_MIN);
8    printf("Absolute value of INT_MIN (error): %d\n", abs(INT_MIN));
9  }
```

(continua en la próxima diapositiva)

Ejemplo (continuación)

Compilar y ejecutar el programa con las siguientes instrucciones:

```
$ cd src
$ make glibc-abs
gcc-15.1.0 -Wall -Wextra -Wpedantic -Werror -std=c23 \
  -o glibc-abs glibc-abs.c
$ ./glibc-abs
Maximum int (INT_MAX, 2^31 - 1): 2147483647
Minimum int (INT_MIN, -2^31): -2147483648
Absolute value of INT_MIN (error): -2147483648
```

(continua en la próxima diapositiva)

Ejemplo (continuación)

Programa que muestra el problema empleando la función `abs` de GHC (archivo `src/ghc-abs.hs`).

```
1  main :: IO ()
2  main = do
3    let maxInt = maxBound :: Int
4        minInt = minBound :: Int
5    print $ "Maximum Int (maxBound :: Int, 2^63 - 1): " ++ show maxInt
6    print $ "Minimum Int (minBound :: Int, -2^63): " ++ show minInt
7    print $ "Absolute value of (minBound :: Int) (error): " ++
8      show (abs minInt)
```

(continua en la próxima diapositiva)

Ejemplo (continuación)

Compilar y ejecutar el programa con las siguientes instrucciones:

```
$ cd src
$ make ghc-abs
ghc -Wall -Werror -o ghs-abs ghc-abs.hs
$ ./ghc-abs
"Maximum Int (maxBound :: Int, 2^63 -1): 9223372036854775807"
"Minimum Int (minBound :: Int, -2^63): -9223372036854775808"
"Absolute value of (minBound :: Int) (error): -9223372036854775808"
```

(continua en la próxima diapositiva)

Ejemplo (continuación)

Adicionamos la pre-condición asociada a la representación de los números enteros (archivo `src/frama-c/abs-4.hs`).

```
1  #include <limits.h>
2
3  /*@
4    requires INT_MIN < val;
5    ensures \result >= 0;
6    ensures (val >= 0 ==> \result == val) && (val < 0 ==> \result == -val);
7  */
8  int abs(int val) {
9    if ( val < 0 ) return -val;
10   return val;
11 }
```

Ejemplo (continuación)

Ejecutamos FRAMA-C con las instrucciones

```
$ cd src/frama-c
$ frama-c-gui -wp -wp-rte abs-4.c
```

(continua en la próxima diapositiva)

Ejemplo (continuación)

También podemos verificar que las llamadas a la función `abs` satisfagan las pre-condiciones (archivo `src/frama-c/abs-5.hs`).

```
1  void foo(int a) {
2    int b = abs(42);
3    int c = abs(-42);
4    int d = abs(a);       // Warning: "a" may be INT_MIN
5    int e = abs(INT_MIN); // False: the parameter is INT_MIN
6  }
```

(continua en la próxima diapositiva)

La función `main`

La función `main` es el punto de entrada del programa. Por esta razón al contexto de su evaluación Frama-C adiciona el valor inicial de la variables globales.

Ejemplo

Una función `main` (archivo `src/frama-c/main.c`).

```
1  int h = 42;
2
3  //@ requires 0 <= h <= 100 ;
4  int main() {
5    //@ assert h == 42 ;
6  }
7
8  //@ requires 0 <= h <= 100 ;
9  void foo() {
10   //@ assert h == 42 ;
11 }
```

(continua en la próxima diapositiva)

Ejemplo (continuación)

Ejecutamos FRAMA-C con las instrucciones

```
$ cd src/frama-c
$ frama-c-gui -wp -wp-rte main.c
```

Apuntadores

El libro [KR1988] comienza la presentación de los apuntadores de la siguiente manera:

> *A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways... (pág. 83)*
>
> *Pointers have been lumped with the* `goto` *statement as a marvelous way to create impossible-to-understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can also be used to achieve clarity and simplicity. (pág. 83)*

Apuntadores

En C:

- el operador unario `&` da la dirección de un objeto (variable, función, etc) y
- el operador unario `*` es el operador de desreferenciación *«dereferencing operator»*, que cuando aplicado a un apuntador, accede al objeto apuntado por el apuntador.

Ejemplo

```
1  int x = 1;
2  int y = 2;
3  int* p;  /* ip is a pointer to int */
4
5  p = &x;  /* p now points to x */
6  y = *p;  /* y is now 1 */
7  *p = 0;  /* x is now 0 */
```

Paso de parámetros a una función

El paso de parámetros a una función puede ser **por valor** *«call by value»* o **por referencia** *«call by reference»*.

- Paso por valor: La función recibe el valor de la variable que se pasa y crea una copia local de ésta. En consecuencia, no es posible modificar la variable pasada.
- Paso por referencia: La función recibe la dirección de la variable que se pasa. En consecuencia, los cambios realizados por la función o el método se realizarán sobre la variable pasada.

En C utilizamos apuntador para pasar los argumentos por referencia.

Ejemplo

Contrato para un función recibiendo los argumentos por referencia (archivo `src/frama-c/swap-1.c`).

```
1  /*@
2    ensures *a == \old(*b) && *b == \old(*a);
3  */
4  void swap(int* a, int* b) {
5    int tmp = *a;
6    *a = *b;
7    *b = tmp;
8  }
```

(continua en la próxima diapositiva)

Ejemplo

Debemos verificar que los apuntadores sean válidos (archivo `src/frama-c/swap-2.c`).

```
1  /*@
2    requires \valid(a) && \valid(b);
3    ensures  *a == \old(*b) && *b == \old(*a);
4  */
5  void swap(int* a, int* b) {
6    int tmp = *a;
7    *a = *b;
8    *b = tmp;
9  }
```

(continua en la próxima diapositiva)

Ejemplo

¿Por qué falla el contrato? (archivo `src/frama-c/swap-3.c`).

```
1  int h = 42;
2
3  int main() {
4    int a = 37;
5    int b = 91;
6
7    //@ assert h == 42;
8    swap(&a, &b);
9    //@ assert h == 42;
10 }
```

(continua en la próxima diapositiva)

Ejemplo

Debemos especificar los efectos colaterales *«side effects»* de la función (archivo `src/frama-c/swap-4.c`).

```
1  /*@
2    requires \valid(a) && \valid(b);
3    assigns *a, *b;
4    ensures  *a == \old(*b) && *b == \old(*a);
5  */
6  void swap(int* a, int* b) {
7    int tmp = *a;
8    *a = *b;
9    *b = tmp;
10 }
```

Tema

- Introducción
- Lógica de Hoare
- Frama-C y su plugin WP
- Referencias

Referencias

[Ach+2010] Peter Achten, Marko van Eekelen, Pieter Koopam y Marco T. Morazán. «Trends in *Trends in Functional Programming* 1999/2000 versus 2007/2008». En: *Higher-Order Symbolic Computation* 23.4 (2010), págs. 465-487. DOI: 10.1007/s10990-011-9074-z (vid. pág. 11).

[Bla2024] Allan Blanchard. «Introduction to C program proof with Frama-C and its WP plugin». Recent version from the GitHub repository. 2 de nov. de 2024. URL: https://github.com/AllanBlanchard/tutoriel_wp (vid. pág. 34).

[Gor2016] Mike Gordon. «Background Reading on Hoare Logic». 2016. URL: https://www.cl.cam.ac.uk/archive/mjcg/HL/ (vid. pág. 14).

[Hei2017] James L. Hein. *Discrete Structures, Logic, and Computability*. 4.ª ed. Jones & Bartlett Learning, 2017 (1995) (vid. pág. 4).

[KR1988] Brian W. Kernighan y Dennis M. Ritchie. *The C Programming Language*. 2.ª ed. Sofware Series. Prentice-Hall, 1988 (1978) (vid. pág. 56).

Referencias

[LL2011] Kenneth C. Louden y Kenneth A. Lambert. *Programming Languages. Principles and Practice*. 3.ª ed. Cengage Learning, 2011 (1993) (vid. págs. 7, 8).

[NS2005] Noam Nisan y Schocken Shimon. *The Elements of Computing Systems. Building a Modern Computer from First Principles*. MIT Press, 2005 (vid. pág. 6).

[Sco2015] Michael L. Scott. *Programming Language Pragmatics*. 4.ª ed. Morgan Kaufmann, 2015 (1999) (vid. pág. 5).