Guía de lectura para la materia

Paradigmas de la Programación

Licenciatura en Ciencias de la Computación
FaMAF-UNC

16 de marzo de 2020

Índice general

Capítulo 1

Introducción y Motivación

1.1. Nuestro objeto de estudio

Esta materia se llama *Paradigmas de la Programación*, pero en otros lugares la pueden encontrar bajo el nombre *Lenguajes de Programación*, porque el objetivo es aprender conceptos y propiedades de los lenguajes de programación.

1.2. Cuestiones fundamentales

¿Cuáles son las atribuciones de un lenguaje de programación?
¿Cómo funcionan los lenguajes de programación?
¿Cómo podemos hablar del "*significado*" de un programa?
¿Cómo podemos comparar diferentes lenguajes de programación?

1.3. Objetivos del curso "Paradigmas de la Programación"

- Objetivizar e instrumentar los conceptos involucrados en la constitución de los lenguajes de programación.

1.4. Para qué sirve esta materia

En la currícula de un profesional de la computación, esta materia nos capacita para analizar, entender y tomar decisiones respecto a las fortalezas y desventajas de los diferentes lenguajes de programación en diferentes contextos y para diferentes tareas.

Por ejemplo, para un proyecto en el que la eficiencia sea crítica (p.ej. dispositivos móviles, grandes volúmenes de datos) podemos elegir un lenguaje que priorice la eficiencia aunque requiera más experiencia de parte del programador, como C. En cambio, si el

equipo de desarrollo del software está compuesto de personas con poca experiencia, será mejor elegir un lenguaje que ayude a prevenir los errores humanos, como por ejemplo Python.

1.5. Estructura de este apunte

Este apunte es un breve resumen en castellano de los contenidos de la materia. Está concebido para acompañar a la lectura del libro Concepts in Programming Languages de John Mitchell. Cada capítulo del apunte tiene un puntero a las secciones correspondientes del libro, así como a las filminas correspondientes de clase. Al final de cada capítulo se encuentran los ejercicios prácticos correspondientes. También se encuentran varias preguntas de tipo verdadero-falso sobre el contenido del capítulo.

Al final del apunte se encuentran exámenes parciales y finales de ediciones pasadas de la materia.

El capítulo 2 trata sobre el alcance de los lenguajes de programación y sus componentes fundamentales: la forma y el significado. En el capítulo 3 describimos cómo funcionan los lenguajes de programación en una computadora.

En el capítulo 4 describimos mecanismos para organizar y manipular los datos, jerarquizándolos en estructuras de conjuntos y subconjuntos, y cómo estas estructuras otorgan mayor seguridad a los lenguajes de programación.

Después, el capítulo 5 presenta la primera abstracción en lenguajes de programación, la estructura de bloques, y a partir de esa abstracción nuestra unidad de significado básica: el activation record. En este capítulo damos ejemplos de dos lenguajes con abstracción de bloques: Algol en el paradigma imperativo y ML en el paradigma funcional. Estos dos lenguajes están en la base de la mayor parte de lenguajes modernos.

En capítulo 6 mostramos diferentes construcciones de los lenguajes para manejar de forma eficiente e intuitiva el flujo de ejecución de un programa. Vemos las implicaciones en eficiencia y expresividad que tienen las diferentes construcciones.

En el capítulo 7 describimos las abstracciones propias del paradigma de orientación a objetos, con ejemplos de C++ y Java. En el capítulo 8 describimos las abstracciones del paradigma de framework, con ejemplos de varios frameworks. Finalmente, en el capítulo 9 describimos las abstracciones del paradigma funcional.

En el capítulo 10 presentamos el paradigma de programación lógica, con su único exponente, Prolog. En el capítulo 11 presentamos cómo se introduce en lenguajes de programación secuenciales la semántica de concurrencia, con ejemplos de concurrencia declarativa con lenguajes como Scala.

Y finalmente en el capítulo 12 mostramos las decisiones de diseño y las concreciones del paradigma de *scripting*, con ejemplos de glsjavascript y Perl.

Este apunte está en construcción y se agradecen comentarios y sugerencias a alemany@famaf.unc.edu.ar.

Capítulo 2

Qué es y qué puede hacer un lenguaje de programación

Un lenguaje de programación es un lenguaje formal diseñado para realizar procesos que pueden ser llevados a cabo por máquinas como las computadoras. Pueden usarse para crear programas que controlen el comportamiento físico y lógico de una máquina o para expresar algoritmos con precisión (fuente: wikipedia).

2.1. Sintaxis y semántica

Los lenguajes son sistemas que se sirven de una **forma** para comunicar un **significado**. Lo que tiene que ver con la forma recibe el nombre de **sintaxis** y lo que tiene que ver con el significado recibe el nombre de **semántica**.

En los lenguajes naturales, como el castellano, el inglés o las lenguas de signos, las palabras son la forma, y el contenido proposicional de las oraciones es el significado.

En los lenguajes de programación, que son lenguajes artificiales creados por hombres (lenguajes *formales*), la forma son **los programas** y el significado es **lo que los programas hacen**, usualmente, en una computadora. En la definición de arriba, se ha descrito lo que los programas como "controlar el comportamiento físico y lógico de una máquina".

Un lenguaje de programación se describe con su sintaxis (qué es lo que se puede escribir legalmente en ese lenguaje) y su semántica (qué efectos tiene en la máquina lo que se escribe en ese lenguaje).

2.2. Alcance de los lenguajes de programación

Mitchell 2.

Para "controlar el comportamiento físico y lógico de una máquina" usamos algoritmos, un conjunto de instrucciones bien definidas, ordenadas y finitas que permite realizar algo de forma inambigua mediante pasos.

Las funciones computables son la formalización de la noción intuitiva de algoritmo, en el sentido de que una función es computable si existe un algoritmo que puede hacer el trabajo de la función, es decir, dada una entrada del dominio de la función puede devolver la salida correspondiente.

El concepto de función computable es intuitivo, se usa para hablar de computabilidad sin hacer referencia a ningún modelo concreto de computación. Cualquier definición, sin embargo, debe hacer referencia a algún modelo específico de computación, como por ejemplo máquina de Turings, las funciones μ recursivas, el lambda cálculo o las máquinas de registro.

La tesis de Church-Turing[1] dice que las funciones computables son exactamente las funciones que se pueden calcular utilizando un dispositivo de cálculo mecánico dada una cantidad ilimitada de tiempo y espacio de almacenamiento. De manera equivalente, esta tesis establece que cualquier función que tiene un algoritmo es computable.

Algunas función no computable famosas son *Halting problem* o calcular la Complejidad de Kolmogorov.

2.3. Sintaxis a través de gramáticas

Nuestro objetivo con respecto a la sintaxis de los lenguajes de programación es describir de forma compacta e inambigua el conjunto de los programas válidos en ese lenguaje. El instrumento formal básico para describir la sintaxis de los lenguajes de programación son las gramáticas independientes de contexto. El estándar de facto para gramáticas independientes de contexto de lenguajes de programación es EBNF.

Sin embargo, las gramáticas independientes de contexto no son suficientemente expresivas para describir adecuadamente la mayor parte de lenguajes de programación. Por esa razón, en la práctica el análisis de la forma de los programas suele incorporar diferentes mecanismos, entre ellos, el análisis de un programa mediante una gramática independiente de contexto, que se complementa con otros mecanismos para alcanzar la expresividad propia de una máquina de Turing. El proceso completo de análisis lo realiza el compilador del lenguaje, como se describe en la sección 3.1.

No obstante, las gramáticas independientes de contexto describen el grueso de la forma de los lenguajes de programación.

Una gramática de ejemplo clásica es la que genera los strings que representan expresiones aritméticas con los cuatro operadores $+, -, *, /$ y los números como operandos:

```
<expresion> --> numero
<expresion> --> ( <expresion> )
<expresion> --> <expresion> + <expresion>
<expresion> --> <expresion> - <expresion>
```

[1] La historia de la colaboración entre Church y Turing es un ejemplo muy interesante de como dos individuos geniales quieren y pueden trabajar juntos para potenciarse mútuamente y llegar a alcanzar logros mucho mayores de los que habrían alcanzado cada uno por su lado. Fíjense también en los estudiantes a los que dirigió Church.

```
<expresion> --> <expresion> * <expresion>
<expresion> --> <expresion> / <expresion>
```

El único símbolo no terminal en esta gramática es `expresion`, que también es el símbolo inicial. Los terminales son `{+,-,*,/,(,),numero}`, donde `numero` representa cualquier número válido.

La primera regla (o producción) dice que una `<expresion>` se puede reescribir como (o ser reemplazada por) un número. Por lo tanto, un número es una expresión válida. La segunda regla dice que una expresión entre paréntesis es una expresión válida, usando una definición recursiva de expresión. El resto de reglas dicen que la suma, resta, producto o división de dos expresiones también son expresiones válidas.

Pueden encontrar algunos ejemplos simples de gramáticas independientes de contexto en `https://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html`

Algunos fenómenos de los lenguajes de programación no se pueden expresar con naturalidad mediante gramáticas independientes de contexto. Por ejemplo es difícil expresar la obligación de que una variable sea declarada antes de ser usada, o bien describir la asignación múltiple de variables, como podemos hacer por ejemplo en Python: `(foo, bar, baz) = 1, 2, 3`.

> **Pregunta 1:**
> *Piense cómo sería una gramática independiente de contexto para expresar la obligación de que una variable sea declarada antes de ser usada, o la asignación múltiple de variables. ¿Cómo es la gramática resultante? ¿Resulta fácil de pensar, fácil de entender?*

Las gramáticas que describen los lenguajes de programación se usan para verificar la correctitud de un programa escrito en ese lenguaje, mediante un compilador. En un compilador, estas limitaciones se solucionan en parte porque hay módulos de procesamiento del programa posteriores a la gramática que tratan algunos de los fenómenos que las gramáticas no pueden capturar.

Sin embargo, también se usan las gramáticas para describir los lenguajes para **consumo humano**. En ese uso, uno desea que la gramática pueda expresar de forma compacta todas las propiedades relevantes del lenguaje. Para conseguirlo, se suelen usar mecanismos ad-hoc para aumentar su poder expresivo, como por ejemplo predicados asociados a reglas. Por ejemplo, si queremos establecer una restricción de tipos en una determinada regla, podemos hacerlo de la siguiente forma:

```
<expresion> --> numero
<expresion> --> ( <expresion> )
<expresion> --> <expresion> + <expresion>
<expresion> --> <expresion> - <expresion>
<expresion> --> <expresion> * <expresion>
<expresion> --> <expresion> / <expresion> ^ <expresion>
   es_de_tipo Float
```

```
<expresion> --> <expresion> div <expresion> ^ <expresion>
   es_de_tipo Int
```

En la práctica, las restricciones de tipos las realiza el análisis semántico del compilador.

Para ejercitar este mecanismo de expresividad, resuelvan el ejercicio 2.1.

2.4. Semántica operacional vs. lambda cálculo o denotacional

Hay diferentes maneras de describir y manipular formalmente la semántica de un programa. Algunas de ellas son el lambda cálculo de Church (Mitchell 4.2), la semántica denotacional de Strachey & Scott (Mitchell 4.3) y diferentes tipos de semántica operacional. En nuestro curso vamos a estar usando un tipo de semántica operacional.

La semántica operacional (de pequeños pasos) describe formalmente cómo se llevan a cabo cada uno de los pasos de un cálculo en un sistema computacional. Para eso se suele trabajar sobre un modelo simplificado de la máquina.

Cuando describimos la semántica de un programa mediante semántica operacional, describimos cómo un programa válido se interpreta como secuencias de pasos computacionales. Estas secuencias son el significado del programa.

Tal vez la primera formalización de semántica operacional fue el uso del cálculo lambda para definir la semántica de LISP que hizo [John McCarthy. "Funciones recursivas de expresiones simbólicas y su cómputo por máquina, Parte I". Consultado el 2006-10-13.].

Pregunta 2:
¿Por qué puede haber más de un tipo de semántica operacional, mientras que el lambda cálculo y la semántica denotacional son sólo uno?

2.5. Para saber más

[John McCarthy. "Funciones recursivas de expresiones simbólicas y su cómputo por máquina, Parte I". Consultado el 2006-10-13.]

2.6. Ejercicios

2.1. En EBNF no podemos expresar lenguajes Turing completos, por esa razón necesitamos aumentar el poder expresivo de las gramáticas libres de contexto. Esto se hace en la implementación con el paso posterior al análisis sintáctico, el análisis semántico. Informalmente, podemos aumentar una gramática libre de contexto con predicados.

Escriba una EBNF aumentada con los predicados necesarios para poder expresar:

- que una variable debe ser declarada antes de ser usada.
- asignación múltiple de variables, con un número arbitrario de variables.

2.2. Considerando la asignación múltiple de variables en Python, ¿es Python un lenguaje independiente de contexto? Justifique su respuesta.

2.3. Cuando se usa una gramática independiente de contexto para describir un lenguaje puede suceder que la gramática resulte ambigua. ¿Cuándo decimos que una gramática es ambigua? ¿Qué problemas presenta la ambigüedad, y cómo se pueden solucionar?

2.4. Dada la siguiente gramática:

```
<e> ::= <n> | <e>+<e> | <e>-<e>
<n> ::= <d> | <n><d>
<d> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

a) Enumere los conjuntos de no terminales y terminales.

b) Dé 3 ejemplos de expresiones válidas en el lenguaje definido por la gramática. Grafique los árboles de derivación correspondientes.

c) Las siguientes secuencias de símbolos: `<e>`,`e-5`, `e + 6 - A` ¿son expresiones válidas en el lenguaje? Justifique su respuesta.

d) ¿Es la gramática definida ambigua? Si lo es, de un ejemplo que pruebe su ambigüedad.

e) Si la gramática es ambigua, ¿qué condiciones agregaría para desambiguarla?

2.5. De la misma forma que obtuvimos la derivación de una expresión algebraica usando la gramática de dígitos, use la siguiente gramática:

```
program = 'PROGRAM', espacio, identificador, espacio,
          'BEGIN', espacio,
          { asignacion, ";", espacio },
          'END.' ;
identificador = caracter_alfabetico, {
   caracter_alfabetico | digito } ;
numero = [ "-" ], digito, { digito } ;
string = '"' , { todos_caracteres - '"' }, '"' ;
```

```
asignacion = identificador , ":=" , ( numero |
   identificador | string ) ;
caracter_alfabetico = "A" | "B" | "C" | "D" | "E" | "F" |
   "G"
                     | "H" | "I" | "J" | "K" | "L" | "M"
   | "N"
                     | "O" | "P" | "Q" | "R" | "S" | "T"
   | "U"
                     | "V" | "W" | "X" | "Y" | "Z" ;
digito = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
   "8" | "9" ;
espacio = ? caracteres_espacio ? ;
all caracteres = ? todos_caracteres_visibles ? ;
```

Para obtener la derivación de la siguiente expresión:

```
PROGRAM DEMO1
BEGIN
  A:=3;
  TEXT:="Hello world!";
END.
```

2.6. Muestre con un ejemplo que la siguiente gramática es ambigua, y modifíquela para que se convierta en una gramática inambigua. ¿Qué estrategias puede usar para hacerlo?

```
<bloque> ::= { <sentencias> }
<sentencias> ::= <sentencias> <sentencia> | <sentencia>
<sentencia> ::= <asignacion> | <sentenciaIf> | <bloque>
<sentenciaIf> ::= if ( <expresion> ) <sentencia> |
                  if ( <expresion> ) <sentencia> else
   <sentencia>
<asignacion> ::= <variable> := <expresion>
<expresion> ::= <variable> | <variable> <operador>
   <variable>
<operador> ::= + | - | * | / | < | > | 'and' | 'or'
<variable> ::= x | y | z
```

Capítulo 3

Cómo funcionan los lenguajes de programación

3.1. Estructura de un compilador

Mitchell 4.1.1.

3.2. Estructuras de datos de bajo nivel

3.2.1. Variables

Una de las estructuras de datos básicas de los lenguajes de programación son las variables. Una variable está formada por una ubicación en la memoria y un identificador asociado a esa ubicación. Esa ubicación contiene un valor, que es una cantidad o información conocida o desconocida. El nombre de la variable es la forma usual de referirse al valor almacenado: esta separación entre nombre y contenido permite que el nombre sea usado independientemente de la información exacta que representa.

Los compiladores reemplazan los nombres simbólicos de las variables con la real ubicación de los datos. Mientras que el nombre, tipo y ubicación de una variable permanecen fijos, los datos almacenados en la ubicación pueden ser cambiados durante la ejecución del programa.

El identificador en el código fuente puede estar ligado a un valor durante el tiempo de ejecución y el valor de la variable puede por lo tanto cambiar durante el curso de la ejecución del programa. De esta forma, es muy sencillo usar la variable en un proceso repetitivo: puede asignársele un valor en un sitio, ser luego utilizada en otro, más adelante reasignársele un nuevo valor para más tarde utilizarla de la misma manera, por ejemplo, en procedimientos iterativos.

Diferentes identificadores del código pueden referirse a una misma ubicación en memoria, lo cual se conoce como aliasing. En esta configuración, si asignamos un valor a una variable utilizando uno de los identificadores también cambiará el valor al que se puede

acceder a través de los otros identificadores.

Veamos un ejemplo:

```
x: int;
y: int;
x: = y + 3;
```

En la asignación, el valor almacenado en la variable y se añade a 3 y el resultado se almacena en la ubicación para x. Notemos que las dos variables se utilizan de manera diferente: usamos el valor almacenado en y, independientemente de su ubicación, pero en cambio usamos la ubicación de x, independientemente del valor almacenado en x antes de que ocurra la asignación.

La ubicación de una variable se llama su L-valor y el valor almacenado en esta ubicación se llama el R-valor de la variable.

En ML, los L-valores y los R-valores tienen diferentes tipos. En otras palabras, una región de la memoria asignable tiene un tipo diferente de un valor que no se puede cambiar. En ML, un L-valor o región asignable de la memoria se llama celda de referencia. El tipo de una celda de referencia indica que es una celda de referencia y especifica el tipo de valor que contiene. Por ejemplo, una celda de referencia que contiene un número entero tiene tipo `int ref`.

Cuando se crea una celda de referencia, se debe inicializar a un valor del tipo correcto. ML no tiene variables no inicializadas o punteros colgantes. Cuando una asignación cambia el valor almacenado en una celda de referencia, la asignación debe ser coherente con el tipo de la celda de referencia: una celda de referencia de tipo entero siempre contendrá un entero, una celda de referencia de tipo lista contendrá siempre (o hará referencia a) una lista, y así.

ML tiene operaciones para crear celdas de referencia, para acceder a su contenido y para cambiar su contenido: `ref`, `!` y `:=`, que se comportan como sigue:

- `ref v` – crea una celda de referencia que contiene el valor v
- `! r` – devuelve el valor contenido en la celda r
- `r: = v` – ubica el valor v en la celda de referencia r

Capítulo 4

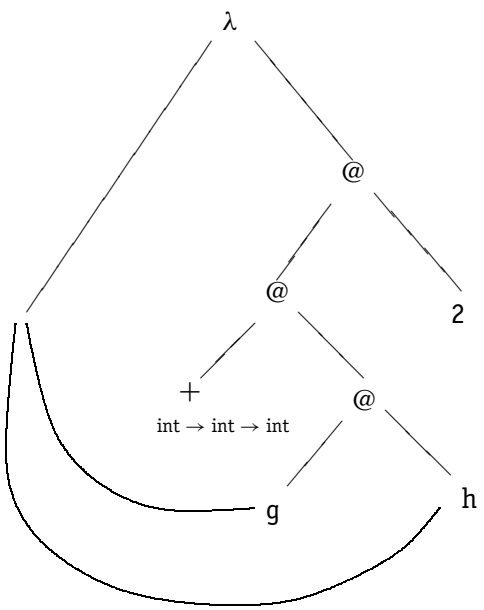
Tipos de datos

Mitchell 6.1., 6.2., 6.3., 6.4., 13.3

4.1. Ejercicios

4.1. En base al siguiente árbol de tipos, calcule el tipo de datos de la función implementada en ML.

```
fun f(g,h) = g(h) + 2;
```



4.2. Calcule el tipo de datos de la función implementada en ML.

```
fun f(g) = g(g(3)) + 2;
```

4.3. Calcule el tipo de datos de las siguientes funciones en ML. Provea el árbol sintáctico de la función y aplique de forma explícita el algoritmo de inferencia de tipos, ya sea sobre el árbol mismo o como sistema de ecuaciones.

```
fun a(x,y) = (x > 2) orelse (y < 10);

fun a(f,x,y) = f(x*2+3) andalso y;

fun d(f,x) = f(x*2) andalso true

fun aplicar(f,x) = f(x)
```

4.4. La siguiente expresión está mal tipada:

```
1 f(g,x) = g(x) + x && g(x)
```

Muestre con el árbol para inferencia de tipos dónde se encuentra el conflicto y en qué consiste. Cómo podría resolver este conflicto un lenguaje de tipado fuerte? y uno de tipado débil?

4.5. La siguiente expresión está mal tipada:

```
1 f (g, x) = g(x) > x(g)
```

Muestre con el árbol para inferencia de tipos dónde se encuentra el conflicto y en qué consiste. Cómo podría resolver este conflicto un lenguaje de tipado fuerte? y uno de tipado débil?

4.6. En el siguiente texto, explique si el diseño de Crystal con *duck typing* mantiene los principios de seguridad del tipado estático, qué errores se podrían producir y qué errores no.

> *The language allows you to program using duck-typing, in a way. For example if you have a variable "duck.ªnd you assign a Duck to it and invoke "quack.ºn it, given a Duck quacks, it will work. Now if later you assign a Horse to it, and the horse doesn't quack, you will get a compile error. So, the compiler catches the error at compile time. Now, if you assign another type, say, a Goose, and it quacks, then the code continue to compile fine (provided that it returns the correct type, which again is determined by its methods, and so on). The new type will be Duck | Goose. In short: there's no "undefined method" errors at runtime in Crystal.*

Capítulo 5

Estructura en bloques

5.1. Código estructurado vs. código *spaghetti*

Mitchell 8.1

En lenguajes como ensamblador o Fortran es muy fácil escribir programas que resulten incomprensibles porque al que lee le cuesta entender la estructura de control del programa. A este tipo de código se le llama código *spaghetti*, y aquí mostramos unos ejemplos:

```
10 IF (X .GT. 0.000001) GO TO 20
   X = -X
11 Y = X*X - SIN(Y)/(X+1)
   IF (X .LT. 0.000001) GO TO 50
20 IF (X*Y .LT. 0.00001) GO TO 30
   X = X-Y-Y
30 X = X+Y
   ...
50 CONTINUE
   X=A
   Y = B-A + C*C
   GO TO 11
   ...
```

En este ejemplo de Fortran de Mitchell (8:204) se ven algunos efectos del código *spaghetti*.

```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
70 END
```

Este fragmento de código *spaghetti* podría traducirse a un lenguaje con bloques de la siguiente forma:

```
10 FOR i = 1 TO 10
20     PRINT i; " squared = "; i * i
30 NEXT i
40 PRINT "Program Completed."
50 END
```

Para evitar este problema, la mayoría de los lenguajes de programación modernos proporcionan alguna forma de bloque. Un bloque es una región del texto del programa con inicio y fin explícitos e inambiguos. Esta región del texto permite organizar de forma explícita la lógica del programa. Pero además, posibilita sucesivas abstracciones sobre el flujo de ejecución.

En primer lugar, los bloques nos permiten hacer declaraciones de variables locales. Por ejemplo:

```
{ int x = 2;
  { int y = 3;
    x = y+2;
  }
}
```

En esta sección del código hay dos bloques. Cada bloque comienza con una llave izquierda, `{`, y termina con una llave derecha, `}`[1]. El bloque exterior comienza con la primera llave izquierda y termina con la última llave derecha. El bloque interno se anida en el interior del bloque exterior, comienza con la segunda llave izquierda y termina con la primera llave derecha.

La variable `x` se declara en el bloque exterior y la variable `y` se declara en el bloque interior. Una variable declarada dentro de un bloque se dice que es una variable local para ese bloque. Las variables que no están declaradas en un bloque, sino en algún otro bloque que lo contiene, se dice que es una variable global para el bloque más interno. En este ejemplo, `x` es local en el bloque exterior, `y` es local para el bloque interior, y `x` es global para el bloque interior.

Gracias a la estructura de bloques se pueden implementar mecanismos de gestión de la memoria que permiten, por ejemplo, el llamado a funciones de forma recursiva.

Las versiones de Fortran de los años 1960 y 1970 no eran estructuradas por bloques. En Fortran histórico, todas las variables, incluyendo todos los parámetros de cada procedimiento (llamado subrutina en Fortran) tenían una ubicación fija en memoria. Esto hacía imposible llamar a un procedimiento de forma recursiva, ya fuera directa o indirectamente. Si el procedimiento Fortran P llama Q, Q llama R, R y luego intenta llamar a P, la segunda

[1] Los bloques se pueden delimitar con cualquier símbolo arbitrario, de hecho, los lenguajes de programación modernos ya no usan llaves sino espacios en blanco, que resultan más naturales para el humano. Un buen ejercicio es pensar la gramática que describiría la sintaxis de delimitadores de bloque de Python.

llamada a P no se puede realizar. Si P se llama por segunda vez en esta cadena de llamadas, la segunda llamada escribiría sobre los parámetros y la dirección de la primera llamada, por lo tanto, no se podría volver correctamente a la primera llamada.

5.2. Estructura de bloque

Mitchell 7.1

Los lenguajes con estructrua de bloque se caracterizan por las siguientes propiedades:

- Las nuevas variables se pueden declarar en varios puntos de un programa.
- Cada declaración es visible dentro de una determinada región de texto del programa, llamada bloque. Los bloques pueden ser anidados, pero no pueden superponerse parcialmente. En otras palabras, si dos bloques contienen expresiones o declaraciones en común, entonces un bloque debe estar enteramente contenida dentro del otro.

  Pregunta 1:
  Den un ejemplo de bloques superpuestos parcialmente, y argumenten qué problemas podemos encontrar si permitimos este tipo de estructuras en nuestros programas.

- Cuando un programa inicia la ejecución de las instrucciones contenidas en un bloque en tiempo de ejecución, se asigna memoria a las variables declaradas en ese bloque.
- Cuando un programa sale de un bloque, parte o toda la memoria asignada a las variables declaradas en ese bloque se libera.
- Un identificador de variable que no está declarado en el bloque actual se considera global a ese bloque, y su referencia es a la entidad con el mismo identificador nombre que se encuentra en el bloque más cercano que contiene al bloque actual.

Aunque la mayoría de los lenguajes de programación de propósito general modernos son estructurados por bloques, muchos lenguajes importantes no explotan todas las capacidades que permite la estructura por bloques. Por ejemplo, C y C ++ no permiten declaraciones de función locales dentro de los bloques anidados, y por lo tanto no se ocupan de las cuestiones de aplicación asociados con el retorno de las funciones de los bloques anidados.

En este capítulo, nos fijamos en cómo se manejan en memoria tres clases de variables:

variables locales que se almacenan en la pila de ejecución, en el activation record asociado al bloque.

parámetros de función que también se almacenan en el activation record asociada con el bloque.

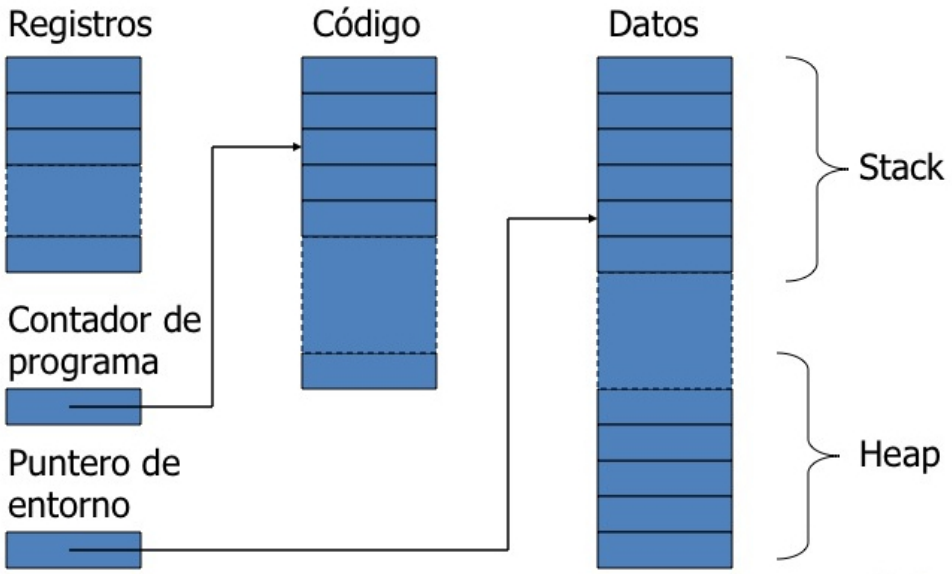


Figura 5.1: Modelo simplificado de la computadora que usamos para describir la semántica operacional.

variables globales que se declaran en algún bloque que contiene al bloque actual y por lo tanto hay que acceder a ellos desde un activation record que se colocó en la pila de ejecución antes del bloque actual.

La mayoría de complicaciones y diferencias entre lenguajes están relacionadas con el acceso a las variables globales, a consecuencia de la estructura de la pila: para dejar más a mano las variables locales, una variable global puede quedar enterrado en la pila bajo un número arbitrario de activation records.

5.3. Activation records

Para describir la semántica de los lenguajes de programación usamos una semántica operacional, es decir, describimos los efectos de las diferentes expresiones del lenguaje sobre una máquina. Para eso usamos un modelo simplificado de la computadora, el que se ve en la figura 5.1.

Vamos a usar esta simplificación para describir cómo funcionan en memoria los lenguajes estructurados por bloques.

Vemos que este modelo de máquina separa la memoria en la que se guarda el código de la memoria en la que se guardan los datos. Usamos dos variables para saber a qué parte de la memoria necesitamos acceder en cada momento de la ejecución del programa: el **contador de programa** y el **puntero de entorno**. El contador de programa es una dirección en la parte de la memoria donde se guarda el código, en concreto, la dirección donde se

encuentra la instrucción de programa que se está ejecuntando actualmente. Normalmente se incrementa después de ejecutar cada instrucción.

El puntero de entorno nos sirve para saber cuáles son los valores que se asignan a las variables que se están usando en una parte determinada del código. En los lenguajes no estructurados por bloques, la memoria de datos es no estructurada, todo el espacio es heap, y por lo tanto los valores que se asignan a las variables son visibles desde cualquier parte del código. En cambio, en los lenguajes estructurados por bloques, se guardan en el *heap* algunos datos para los que necesitamos persistencia (por ejemplo, variables globales), en los registros guardamos algunos datos para los que queremos rápido acceso, y en la pila de ejecución o stack se guardan los datos estructurados en forma de pila, lo cual hace posible que se instancien variables locales y argumentos distintos para cada llamada de función. Esto posibilita que las funciones puedan ser más abstractas y que se puedan hacer llamadas recursivas.

Pregunta 2:
¿Por qué no se pueden hacer llamadas recursivas sin bloques ni activation records? Explique usando Fortran como ejemplo.

El stack o pila de ejecución funciona de la siguiente forma: cuando el programa entra en un nuevo bloque, se agrega a la pila una estructura de datos que se llama activation record o marco de pila (*stack frame*), que contiene el espacio para las variables locales declaradas en el bloque, normalmente, por la parte de arriba de la pila. Entonces, el puntero de entorno apunta al nuevo activation record. Cuando el programa sale del bloque, se retira el activation record de la pila y el puntero de entorno se restablece a su ubicación anterior, es decir, al puntero de entorno correspondiente a la función que llamaba a la función que ha sido desapilada. El activation record que se apila más recientemente es el primero en ser desapilado, a esto se le llama *disciplina de pila*.

En la figura 5.2 podemos ver la información que hay en un activation record de arquitectura x86. Vemos que en esta arquitectura hay, además de espacio para variables locales, también espacio para los argumentos que puede recibir una función, que funcionan como variables locales con respecto a la función pero que son escritos por la función que la llama. También encontramos espacio para guardar resultados intermedios, en el caso de que sea necesario. Podemos observar que hay dos direcciones de memoria donde se guardan datos importantes: una, el control link, contiene el que será el puntero de entorno cuando se desapile el activation record actual, es decir, el puntero de entorno del activation record correspondiente a la función que llamaba a la función del activation record actual. La otra dirección de memoria distinguida es la llamada *dirección de retorno*, que es donde se va a guardar el resultado de la ejecución de la función, si es que lo hay.

Los activation records pueden tener tamaños variables, por lo tanto, las operaciones que ponen y sacan activation records de la pila guardan también en cada activation record una variable con la dirección de memoria que corresponde a la parte superior del registro de activación anterior. Esta variable o puntero (porque apunta a una dirección de memoria) se llama control link, porque es el enlace que se sigue cuando se devuelve el control a la instrucción en el bloque precedente.

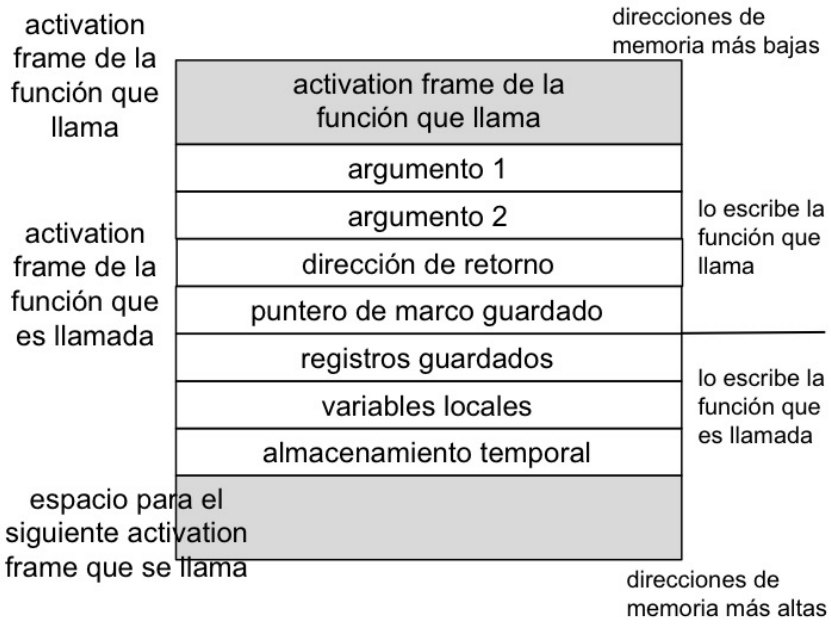


Figura 5.2: Esquema de los contenidos de un activation record típico de la arquitectura x86.

Pregunta 3:
Aunque la mayoría de lenguajes estructurados por bloques se ejecutan con una pila, las funciones de alto orden pueden hacer que falle la disciplina de pila. ¿Por qué?

Pregunta 4:
Se pueden hacer algunas optimizaciones sobre el funcionamiento de los activation records, cuáles se le ocurren?

Pregunta 5:
Los programas estructurados por bloques y su traslado a activation records en una parte de la memoria permiten las llamadas recursivas. Pero hacer muchas llamadas recursivas puede crear un problema en la memoria. ¿Cómo se llama este problema y cómo lo solucionan los compiladores de algunos lenguajes?

Pregunta 6:
¿Por qué pueden tener tamaños variables los activation records?

El lenguaje de programación C está diseñado para ser fácil de compilar y ejecutar, evitando varias de las propiedades de alcance y de manejo de memoria en general que se describen en este capítulo. Sin embargo, algunas implementaciones comerciales de C y C++ permiten tomar funciones como parámetros de otras funciones y devolver funciones como resultado de otras funciones, preservando el alcance estático con clausuras.

En la siguiente sección vamos a ver cómo funcionan los activation records en ejemplos de ejecución. Para una explicación alternativa, pueden consultar la sección correspondiente de la wikipedia sobre la estructura de pila de llamadas.

5.3.1. Detalle de ejecución de un activation record

Mitchell 7.2

Describiremos el comportamiento de los bloques *in line* primero, ya que estos son más simples que los bloques asociados a las llamadas de función. Un bloque *in line* es un bloque que no es el cuerpo de una función o procedimiento.

Cuando un programa en ejecución entra en un bloque *in line*, se asigna espacio en memoria para las variables que se declaran en el bloque, un conjunto de posiciones de memoria en la pila, el activation record.

Veamos cómo funciona paso a paso, con el siguiente ejemplo de código.

```
1 { int x=0;
  int y=x+1;
3     { int z=(x+y)*(x-y);
      };
5 }
```

Si este código es parte de un programa más grande, la pila puede contener espacio para otras variables antes de ejecutar este bloque.

Este código estará escrito en código máquina, traducido por el compilador, y guardado en la parte de la memoria en la que se guarda el programa. El contador de programa recorrerá cada una de sus instrucciones, y realizará las acciones que indiquen en memoria, tal como sigue.

Cuando se introduce el bloque exterior, se pone en la parte superior de la pila un activation record que contiene espacio para las variables `x` e `y`. A continuación se ejecutan las declaraciones que establecen valores de `x` e `y`, y los valores de `x` e `y` se almacenan en las direcciones de memoria del activation record.

Cuando se introduce el bloque interior, se apila un nuevo activation record que con direcciones en memoria para `z`. Después de establecer el valor de `z`, el activation record que contiene este valor se retira de la pila y el puntero de entorno deja de referirse a este activation record y pasa a referirse al activation record que está ahora en la cabeza de la pila. Por último, cuando se sale del bloque exterior, el activation record que con `x` e `y` se retira de la pila y el puntero de entorno se refiere al activation record que esté en la cabeza de la pila o a ninguno si no hay más activation records en la pila.

Podemos visualizar esta ejecución usando una secuencia de gráficos que describen la secuencia de estados por los que pasa la pila, como se ve en la figura 5.3.

El número de direcciones en memoria que va a necesitar un activation record en tiempo de ejecución depende del número de variables declaradas en el bloque y sus tipos. Debido a que estas cantidades se conocen en tiempo de compilación, el compilador puede determinar

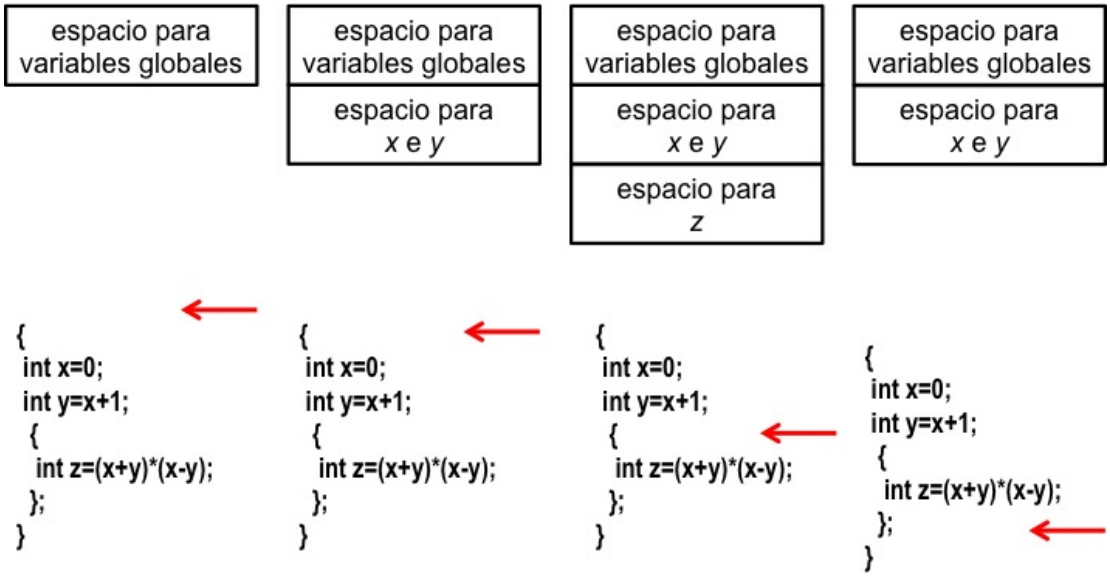


Figura 5.3:

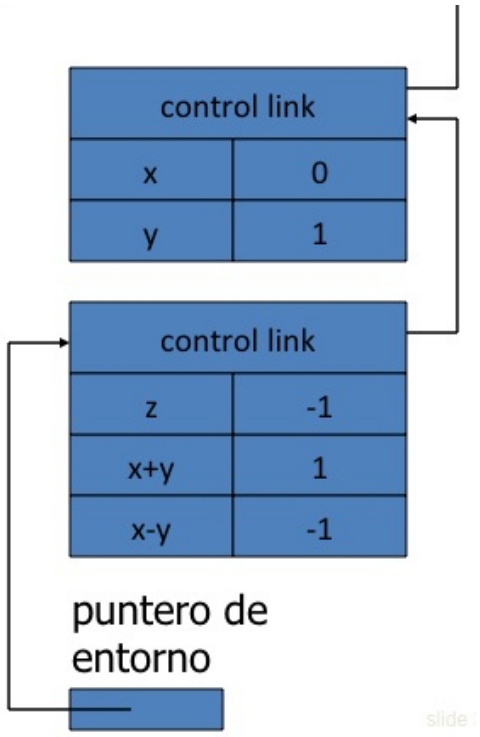


Figura 5.4:

el formato de cada activation record y almacenar esta información como parte del código compilado.

En general, un activation record también puede contener espacio para resultados intermedios, como hemos visto en la figura 5.2. Estos son valores que no reciben un identificador de variable explícito en el código, pero que se guardan temporalmente para facilitar algún cálculo.

Por ejemplo, el registro de activación para este bloque,

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

puede tener espacio para asignar valor a la variable `z` pero también para los valores `x+y` y `x-y`, porque los valores de estas subexpresiones pueden tener que ser evaluados y almacenados en algún lugar antes de multiplicarse. Si hacemos zoom en el tercer estado de la pila de la figura 5.3, veremos lo que se muestra en la figura 5.4, con espacio para valores temporales.

En las computadoras modernas, hay suficientes registros en memoria para guardar estos resultados intermedios en los registros y no en la pila. No hablamos de ubicación de registros porque es un tema que compete a los compiladores y no al diseño de lenguajes de programación.

5.4. Ejercicios

5.1. El siguiente es un ejemplo de "*spaghetti code*". Reescríbalo de forma que NO use saltos (GOTO), y en cambio use programación estructurada.

```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
70 END
```

5.2. Diagrame los estados de la pila de ejecución en los diferentes momentos de la ejecución del siguiente programa, mostrando cómo se apilan y desapilan los diferentes activation records a medida que se va ejecutando el programa. Puede representar los activation records con la información de variables locales y control links.

```
{int x=2 {int y=3; x=y+2;}}
```

5.3. Diagrame como en el ejercicio anterior.

```
int sum(int n){
    if(n==0)
        return n;
```

```
    else
5       return n+sum(n-1);
}
7 sum(4);
```

Capítulo 6

Control de la ejecución

6.1. Funciones y procedimientos

Mitchell 7.3

6.2. Pasaje de parámetros

Mitchell 7.3.2
https://en.wikipedia.org/wiki/Evaluation_strategy

Los nombres de los parámetros que se usan cuando se declara una función se llaman parámetros formales. Cuando se llama a una función, se usan los llamados parámetros reales, que son los valores efectivos de los parámetros formales en una llamada concreta a la función. Ilustraremos la distinción entre los parámetros formales y los reales se ilustra en el siguiente código:

```
proc p (int x, int y) {
    if (x > y) then . . . else . . . ;
    ...
    x := y*2 + 3;
    ...
}
p (z, 4*z+1);
```

Los identificadores x e y son parámetros formales del procedimiento p. Los parámetros reales en la llamada a p son z y $4 * z + 1$.

La forma en que los parámetros reales se evalúan y se pasan a la función depende del lenguaje de programación y el tipo de mecanismo de pasaje de parámetros que utiliza. Los distintos mecanismos de pasaje de parámetros se diferencian por el momento en el que se evalúa el parámetro real (evaluación estricta, en el momento de pasar el parámetro, o perezosa, cuando se necesita) y por la ubicación de memoria donde se almacena el valor

del parámetro (la misma ubicación del bloque que hace la llamada a función o una nueva ubicación específica de la función).

La mayoría de los lenguajes de programación actuales evalúan los parámetros reales de forma estricta, no perezosa, pero hay algunas excepciones. (Una de las razones para que un lenguaje o optimización de programa podría querer retrasar la evaluación de un parámetro real es que la evaluación podría ser costosa y el parámetro real podría no ser utilizado en algunas llamadas.) Entre los mecanismos que evalúan el parámetro real antes de ejecutar el cuerpo de la función, los más comunes son el pasaje por referencia y el pasaje por valor. Estos dos mecanismos se diferencian entre ellos por la ubicación de memoria donde se almacena el valor del parámetro. El pasaje por referencia pasa el L-valor, es decir, la ubicación en memoria, del parámetro real, mientras que el pasaje por valor pasa el R-valor, es decir, el valor que hay en la ubicación de memoria.

La diferencia entre el pasaje por valor y el pasaje por referencia es importante para el programador por varias razones:

Efectos secundarios Las asignaciones de valor en el interior del cuerpo de la función pueden tener diferentes efectos.

Aliasing el aliasing ocurre cuando dos identificadores de variable se refieren al mismo objeto o ubicación. Esto puede ocurrir cuando dos parámetros se pasan por referencia o un parámetro pasado por referencia tiene la misma ubicación que la variable global del procedimiento.

Eficiencia Pasar por valor puede ser ineficaz para grandes estructuras de datos, si hay que copiar ese valor. Pasar por referencia puede ser menos eficiente que pasar por valor pequeñas estructuras que se adapten bien en la pila, porque cuando los parámetros se pasan por referencia, debemos resolver la referencia de un puntero para obtener su valor.

Hay dos formas de describir la semántica del pasaje de parámetros: con diagramas de la memoria y la pila que muestren si la pila contiene una copia del parámetro real o una referencia. Otra forma es traducir el código a un lenguaje que distinga entre R-valores y L-valores, que es lo que vamos a hacer aquí.

6.2.1. Semántica de pasaje por valor (*call-by-value*)

El pasaje por valor es la estrategia de evaluación más común en los lenguajes de programación. Es una forma estricta de pasaje de parámetros, es decir, que el argumento se evalúa, y el valor que se obtiene (su R-valor) se liga a una variable local de la función, en general copiando el valor a una nueva ubicación de memoria en el activation record de la llamada a función. El valor que se encuentra en el activation record que llama a la función no es modificado en ningún momento.

Por ejemplo, veamos esta definición de función y su llamada:

```
f (x) = {x: = x + 1; volver x};
.... f (y) ...;
```

Si el parámetro se pasa por valor e y es una variable entera, entonces este código tiene el mismo significado que el siguiente código ML:

```
fun f (z: int) = let x = ref z in x:=!x+1;!x end;
...f(!y) ...;
```

Como se puede ver en el tipo, el valor pasado a la función f es un número entero. El entero es el R-valor del parámetro real y, como lo indica la expresión `!y` en la llamada. En el cuerpo de f, se asigna y se inicializa una nueva ubicación para el R-valor de y.

Si el valor de y es 0 antes de la llamada, entonces el valor de $f(!y)$ es 1 porque la función f incrementa el parámetro y devuelve su valor. Sin embargo, el valor de y después de la llamada es aún 0, porque la asignación dentro del cuerpo de f solamente cambia el contenido de una ubicación temporal.

6.2.2. Semántica de pasaje por referencia *call-by-reference*

En pasaje por referencia se liga el L-valor del parámetro real al parámetro formal. En este caso se tiene un único valor referenciado (o apuntado) desde dos puntos diferentes, el programa principal y la función a la que se le pasa el argumento, por lo que cualquier acción sobre el parámetro se realiza sobre la misma posición de memoria (aliasing).

Esto significa que la función que es llamada puede modificar la variable con efecto en el bloque que llama a la función. De esta manera, el pasaje por referencia se puede usar como un canal de comunicación entre la función que llama y la que es llamada, pero a la vez resulta más difícil seguir el rastro de los efectos de una llamada a función, y se pueden introducir errores sutiles. Por esta razón, aunque hay muchos lenguajes que ofrecen pasaje por referencia, pocos lo usan como pasaje por defecto, por ejemplo, Perl. Algunos lenguajes, como C++, PHP, Visual Basic .NET o C# tienen pasaje por valor por defecto pero ofrecen una sintaxis específica para pasar parámetros por referencia.

Veamos el detalle de la semántica de pasaje por referencia.

Considere la misma definición de la función y de llamadas utilizada en la explicación de pasaje por valor:

```
f (x) = {x: = x + 1; volver x};
.... f (y) ...;
```

Si el parámetro se pasa por referencia e y es una variable entera, entonces este código tiene el mismo significado que el siguiente código de ML:

```
fun f (x : int ref) = ( x := !x+1; !x );
...f(y)
```

Como se puede ver en el tipo, el valor pasado a la función f es una referencia, un L-valor. Si el valor de y es 0 antes de la llamada, entonces el valor de $f(!y)$ es 1 porque la función f incrementa el parámetro y devuelve su valor. Sin embargo, a diferencia de la situación para el pasaje por valor, el valor de y es 1 también después de la llamada, debido a que la asignación dentro del cuerpo de f también cambia el valor del parámetro real.

Veamos un ejemplo, escrito en una notación asimilable a Algol, que combina pasaje por referencia y pasaje por valor:

```
fun f(pass-by-ref x : int, pass-by-value y : int)
  begin
    x := 2;
    y := 1;
    if x = 1 then return 1 else return 2;
  end;
var z : int;
z := 0;
print f(z,z);
```

Si traducimos este ejemplo en pseudo-Algol a ML nos da:

```
fun f(x : int ref, y : int) =
  let val yy = ref y in
    x := 2;
    yy := 1;
    if (!x = 1) then 1 else 2
  end;
val z = ref 0;
f(z,!z);
```

Este código, que trata a los R-valores y L-valores explícitamente, muestra que para el pasaje por referencia se pasa un L-valor, la referencia z. Para pasaje por valor, se pasa un R-valor, el contenido $!z$ de z, y se le asigna una nueva ubicación temporal.

Si se pasa y por valor tal como se ve, a z se le asigna el valor 2. En cambio, si y se hubiera pasado por referencia, entonces x e y habrían sido alias de la misma variable y a z se le habría asignado el valor 1.

En este otro ejemplo tenemos una función que comprueba si sus dos parámetros son alias:

```
function (y,z){
  y := 0;
  z :=0;
  y := 1;
  if z=1 then y :=0; return 1 else y :=0; return 0
}
```

Si y y z son alias, entonces asignar 1 a y asignará 1 a z también, y la función devolverá 1. De lo contrario, la función devolverá 0. Por lo tanto, una llamada $f(x, x)$ tendrá diferentes resultados si los parámetros se pasan por valor o por referencia.

6.2.3. Sutilezas entre pasaje por valor y pasaje por referencia

En lenguajes puramente funcionales típicamente no hay diferencia semántica entre pasaje por valor y pasaje por referencia, porque sus estructuras de datos son inmutables, por lo que es imposible que una función pueda modificar sus argumentos. Por esta razón normalmente se describen como de pasaje por valor, a pesar de que las implementaciones utilizan normalmente pasaje por referencia internamente porque es más eficiente.

En algunos casos de pasaje por valor el valor que se pasa no es el habitual de la variable, sino una referencia a una ubicación de memoria. El efecto es que lo que sintácticamente parece llamada por valor puede terminar teniendo los efectos de un pasaje por referencia. Lenguajes como C y ML utilizan esta técnica. No es una estrategia de evaluación independiente, ya que el lenguaje sigue teniendo pasaje por valor pero a veces se llama pasaje por dirección de memoria. En un lenguaje con poca seguridad como C puede causar errores de seguridad de la memoria como desreferencia de puntero nulo, y también puede ser confuso. En cambio, en ML las referencias tienen seguridad de tipo y de memoria.

Aquí vemos un ejemplo de pasaje por dirección que simula pasaje por referencia en C:

```
void Modify(int p, int * q, int * o)
{
    p = 27; // pasado por valor - solo se modifica el
   parametro local
    *q = 27; // pasado por valor o referencia, verificar en
   punto de llamada para comprobar cual
    *o = 27; // pasado por valor o referencia, verificar en
   punto de llamada para comprobar cual
}
int main()
{
    int a = 1;
    int b = 1;
    int x = 1;
    int * c = &x;
    Modify(a, &b, c);   // a se pasa por valor, b se pasa por
   referencia creando un puntero,
                        // c es un puntero pasado por valor
    // b y x han sido cambiados
    return(0);
}
```

Hay varias razones por las que se puede pasar una referencia. Una razón puede ser que el lenguaje no permita el pasaje por valor de estructuras de datos complejas, y se pase una referencia para preservar esa estructura. También se usa de forma sistemática en lenguajes con orientación a objetos, lo que se suele llamar *call-by-sharing*, *call-by-object* o *call-by-object-sharing*.

En *call-by-sharing*, el pasaje de parámetros es por valor, pero se pasa una referencia a un objeto. Por lo tanto, se hace una copia local del argumento que se pasa, pero esa copia es una referencia a un objeto que es visible tanto desde la función llamada como desde la función que llama.

La semántica de *call-by-sharing* difiere del pasaje por referencia en que las asignaciones a los argumentos dentro de la función no son visibles a la función que llama, por lo que, por ejemplo, si se pasa una variable, no es posible simular una asignación en esa variable en el alcance de la función que llama. Por lo tanto, no se pueden hacer cambios a la referencia que se ha pasado por valor.

Sin embargo, a través de esa referencia de la que la función llamada ha hecho una copia, ambas funciones, la llamada y la que llama, tienen acceso al mismo objeto, no se hace ninguna copia. Si ese objeto es mutable, los cambios que se hagan al objeto dentro de la función llamada son visibles para la función que llama, a diferencia del pasaje por valor.

Este es el comportamiento de Java, Python, Ruby, JavaScript o OCaml. Sin embargo, el término *call-by-sharing* no se usa mucho, se usan diferentes términos en los diferentes lenguajes y documentaciones. Por ejemplo, en la comunidad Java dicen que Java tiene pasaje por valor, mientras que en la comunidad Ruby dicen que Ruby tiene pasaje por referencia, a pesar de que los dos lenguajes tienen la misma semántica. El *call-by-sharing* implica que los valores que se pasan por parámetro se basan en objetos en lugar de tipos primitivos, es decir que todos los valores están empaquetados.

Resuelva el ejercicio 6.11 para profundizar sobre este tipo de pasaje de parámetros.

6.2.4. Semántica de pasaje por valor-resultado

Es un tipo de pasaje de parámetros con evaluación estricta (no perezosa) poco usado en los lenguajes de programación actuales. Se basa en que dentro de la función se trabaja como si los argumentos hubieran sido pasados por valor, pero al acabar la función los valores que tengan los argumentos serán copiados a la ubicación de memoria en la que se ubicaba el valor copiado inicialmente.

Este tipo de pasaje de parámetros puede ser simulado en cualquier lenguaje que permita el paso de valores por referencia de la siguiente forma:

```
void EjemploValorRes(int a1, int a2, int a3) {
    int aux1 = a1, aux2 = a2, aux3 = a3;
    // codigo trabajando con aux1, aux2 y aux3
    a1 = aux1; a2 = aux2; a3 = aux3; // Dependiendo del
   compilador la copia se realiza en un sentido o en el otro
}
```

Tal y como indica el ejemplo de simulación de valor-resultado, el orden de copia depende del compilador, lo que implica que la misma función pueda dar resultados diferentes según el compilador usado.

Pregunta 1:
Qué ventajas y qué desventajas tiene pasaje por valor-resultado?

Pregunta 2:
Haga un programa como en la sección anterior, que tenga diferentes resultados si los parámetros se pasan por valor, por referencia o por valor-resultado.

6.2.5. Pasaje de parámetros no estricto (perezoso)

En la evaluación perezosa, los argumentos de una función no se evalúan a menos que se utilicen efectivamente en la evaluación del cuerpo de la función.

Pregunta 3:
¿La evaluación de las alternativas de expresiones condicionales debe considerarse estricta o perezosa?

Pregunta 4:
¿Se le ocurre alguna forma de optimización de la evaluación de expresiones complejas, usando evaluación perezosa y las propiedades del absorbente de un operador?

Pasaje por nombre *call-by-name*

En el pasaje *call-by-name*, los argumentos de una función no se evalúan antes de la llamada a la función, sino que se sustituyen sus nombres directamente en el cuerpo de la función y luego se dejan para ser evaluados cada vez que aparecen en la función. Si un argumento no se utiliza en el cuerpo de la función, el argumento no se evalúa; si se utiliza varias veces, se re-evalúa cada vez que aparece.

En algunas ocasiones, *call-by-name* tiene ventajas sobre pasaje por valor, por ejemplo, si el argumento no siempre se usa, porque nos ahorramos la evaluación del argumento. Si el argumento es un cálculo no termina, la ventaja es enorme. Sin embargo, cuando se utiliza el argumento en la función, el pasaje por nombre es a menudo más lento.

Este tipo de pasaje se usó en ALGOL 60. Scala tiene por defecto pasaje por valor, pero también pueden hacer pasaje por nombre. Eiffel provee agentes, operaciones que serán evaluadas cuando sea necesario.

Pasaje por necesidad *call-by-need*

El pasaje por necesidad *call-by-need* es una versión memoizada de pasaje por nombre donde, si se evalúa el argumento de la función, ese valor se almacena para usos posteriores. En un entorno de "puro"(sin efectos secundarios), esto produce los mismos resultados que el pasaje por nombre; cuando el argumento de la función se utiliza dos o más veces, el pasaje por necesidad es casi siempre más rápido.

Haskell es el lenguaje más conocido que utiliza *call-by-need*. R también utiliza una forma de pasaje por necesidad. Variantes de .NET pueden simular llamada por necesidad utilizando el tipo Lazy¡T¿.

Pregunta 5:
Haga un programa como en la sección anterior, que tenga diferentes resultados si los parámetros se pasan por nombre, por necesidad o por algún otro método.

6.2.6. Ejercicios

6.1. Si el Browse del siguiente programa muestra 4, qué tipo de pasaje de parámetros tiene?

```
declare A=2 B=4 C=1
proc {P X Y Z}
   C=5
   X=Z
   C=4
   Y=Z+A
end
in {P A B A+C}
{Browse B}
```

a) por valor

b) por referencia

c) por valor-resultado

d) por nombre

e) por por necesidad

6.2. Si el Browse del siguiente programa muestra 5, qué tipo de pasaje de parámetros tiene?

```
declare A=2 B=4 C=1
proc {P X Y Z}
   C=5
   X=Z
```

```
  C=4
  Y=Z+A
end
in {P A B A+C}
{Browse B}
```

a) por valor

b) por referencia

c) por valor-resultado

d) por nombre

e) por por necesidad

6.3. La ventaja del pasaje de parámetros por referencia es que...

a) reduce aliasing.

b) ocupa menos lugar en la memoria.

c) no tiene efectos fuera de la función llamada.

d) permite actuar en la función que llama, si el programador lo necesita.

6.4. La ventaja del pasaje de parámetros por valor es que...

a) reduce aliasing.

b) ocupa menos lugar en la memoria.

c) no tiene efectos fuera de la función llamada.

d) permite actuar en la función que llama, si el programador lo necesita.

6.5. La ventaja del pasaje de parámetros por valor-resultado es que...

a) reduce aliasing.

b) ocupa menos lugar en la memoria.

c) no tiene efectos fuera de la función llamada.

d) permite actuar en la función que llama, si el programador lo necesita.

6.6. Los métodos de pasaje por parámetros que siguen la filosofía de evaluación perezosa (*lazy evaluation*) son...

a) pasaje por referencia.

b) pasaje por valor.

c) pasaje por valor-resultado.

d) pasaje por nombre.

e) pasaje por necesidad.

6.7. En el siguiente programa,

```
begin
  integer n;
  procedure p(k: integer);
    begin
      k := k+2;
      print(n);
      n := n+(2*k);
    end;
  n := 4;
  p(n);
  print(n);
end;
```

Qué dos valores se imprimen si `k` se pasa...

a) por valor?

b) por valor-resultado?

c) por referencia?

6.8. En el siguiente programa:

```
begin
  integer a, b, c;
  procedure p(x: integer, y: integer, z: integer);
    begin
      c := 5;
      x := z;
      c := 4;
      y := z+a
    end;
  a := 2;
  b := 4;
  c := 1;
  p(a, b, a+c);
  print(b);
end;
```

Qué dos valores se imprimen si `k` se pasa...

a) por valor?

b) por valor-resultado?

c) por referencia?

6.9. En los siguientes fragmentos de programa, introduzcan los elementos necesarios para determinar si `C` y `Perl` funcionan por *call-by-value* o *call-by-reference*:

Fragmento en `C`:

```
int main(void) {
    int x=50, y=70;
    ...
    printf("x=%d y=%d", x, y);
    return 0;
}

void interchange(int x1, int y1) {
    int z1;
    ...
    ...
    ...
    printf("x1=%d y1=%d", x1, y1);
}
```

Fragmento en `Perl`:

```
$x=50;
$y=70;
&interchange ($x, $y);
print "x:$x y:$y\n";

sub interchange {
  ($x1, $y1) = @_;
  ...
  ...
  ...
  print "x1:$x1 y1:$y\n";
}
```

6.10. ¿Qué diferencia hay en `Scala` entre pasar una variable por valor, pero *lazy*, a pasar a una variable por nombre? Cree un programa que muestre estas diferencias.

6.11. Describa qué sucede en el siguiente programa en `Java`, teniendo en cuenta el mecanismo de pasaje de parámetros que utiliza el lenguaje.

```
public class Point {
  public int x;
```

```
  public int y;

  public Point(int x, int y){
    this.x = x;
    this.y = y;
  }

  public static void tricky1(Point arg1, Point arg2)
  {
    arg1.x = 100;
    arg1.y = 100;
    Point temp = arg1;
    arg1 = arg2;
    arg2 = temp;
  }

  public static void tricky2(Point arg1, Point arg2)
  {
   arg1 = null;
   arg2 = null;
  }

  public static void main(String [] args)
  {
    Point pnt1 = new Point(0,0);
    Point pnt2 = new Point(0,0);
    System.out.println("pnt1 X: " + pnt1.x + " pnt1 Y: "
   +pnt1.y);
    System.out.println("pnt2 X: " + pnt2.x + " pnt2 Y: "
   +pnt2.y);
    System.out.println("triki1\n");
    tricky1(pnt1,pnt2);
    System.out.println("pnt1 X: " + pnt1.x + " pnt1 Y:" +
   pnt1.y);
    System.out.println("pnt2 X: " + pnt2.x + " pnt2 Y: "
   +pnt2.y);
    System.out.println("\ntriki2");
    tricky2(pnt1,pnt2);
    System.out.println("pnt1 X: " + pnt1.x + " pnt1 Y:" +
   pnt1.y);
    System.out.println("pnt2 X: " + pnt2.x + " pnt2 Y: "
   +pnt2.y);
  }
```

```
40 }
```

6.3. Alcance estático vs. alcance dinámico

Mitchell 7.3.3

Si un identificador `x` aparece en el cuerpo de una función, pero `x` no se declara dentro de la función, entonces el valor de `x` depende de alguna declaración fuera de la función. En esta situación, la ubicación de `x` está fuera del registro de activación para la función y es una variable libre o variable global respecto a esa función. Debido a que `x` ha sido declarada en otro bloque, el acceso a una `x` libre o global consiste en encontrar el registro de activación pertinente en la pila.

Hay dos políticas principales para buscar la declaración adecuada de un identificador de variable libre:

alcance estático (también llamado alcance léxico) Un identificador global se refiere al identificador con ese nombre que se encuentre en el bloque contenedor más cercano *en el texto del programa.*

alcance dinámico Un identificador global se refiere al identificador que se encuentre en el registro de activación más reciente en la pila de ejecución.

Por lo tanto, la búsqueda de una declaración usando la política de alcance estático utiliza la relación estática e inmutable entre bloques en el texto del programa. Por el contrario, el alcance dinámico utiliza la secuencia efectiva de las llamadas que se ejecutan en la ejecución del programa, que es dinámica y puede cambiar.

Aunque la mayoría de los lenguajes actuales de programación de propósito general utilizan alcance estático para las declaraciones de variables y funciones, el alcance dinámico es un concepto importante que se utiliza en lenguajes de propósito específico y en construcciones especializadas como las excepciones. Algunos lenguajes con alcance dinámico son los dialectos antiguos de Lisp, los lenguajes de marcado Tex/LaTex, las excepciones y las macros.

Pregunta 6:
¿En qué contextos le parece que sería útil tener alcance dinámico?

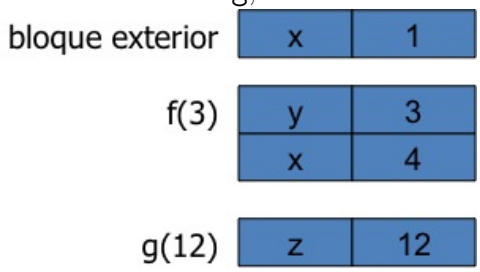
6.3.1. Ejemplo de diferencia entre los dos tipos de alcance

La diferencia entre el alcance estático y dinámico se ilustra mediante el siguiente código, que contiene dos declaraciones de `x`:

```
int x=1;
2 function g(z) = x+z;
function f(y) = {
```

```
4   int x = y+1;
    return g(y*x)
6 };
f(3);
```

La llamada a `f(3)` lleva a una llamada a `g(12)` dentro de la función `f`. Esto provoca que se evalúe la expresión `x + z` en el cuerpo de `g`. Después de la llamada a `g`, la pila de ejecución tendrá registros de activación para la declaración externa de `x`, la invocación de `f`, y la invocación de `g`, como se muestra en la siguiente ilustración:



En este punto se almacenan en la pila dos identificadores de variable con el nombre `x`, uno en el bloque exterior y uno dentro de `f`. Con alcance dinámico, el identificador de `x` en la expresión `x + z` será interpretado como la del registro de activación creado más recientemente, es decir, `x = 4`. Con alcance estático, el identificador de `x` en `x + z` se referirá a la declaración de `x` del bloque del texto de programa más cercano que lo contiene, mirando hacia arriba desde el lugar que aparece `x + z` en el texto del programa. Con alcance estático, la declaración correspondiente de `x` es la del bloque exterior, es decir, `x = 1`.

6.3.2. Naturalidad y overhead del alcance estático

El alcance estático es la semántica más intuitiva para la mayor parte de programadores, por esta razón los lenguajes de programación proveen los mecanismos necesarios para poder mantener esta semántica. Hay diferentes opciones para hacerlo.

La opción más sencilla es prohibir los contextos en los que el alcance estático daría un resultado distinto al alcance dinámico. Esta es la política que implementan las versiones mayoritarias de C, en las que no se puede definir una función dentro de un bloque anidado u otra función, sino que siempre se tienen que definir en el nivel más bajo de anidación del texto del programa.

> **Pregunta 7:**
> *Describa el conflicto en resolución de alcance que se resuelve en C prohibiendo la definición de una función dentro de bloques anidados.*

Sin embargo, en la mayoría de lenguajes se usan *access links* para mantener la semántica de alcance estático.

Un *access link* es una dirección de memoria que se guarda en el activation record de una función y que apunta al activation record del bloque más cercano que lo contiene en el texto

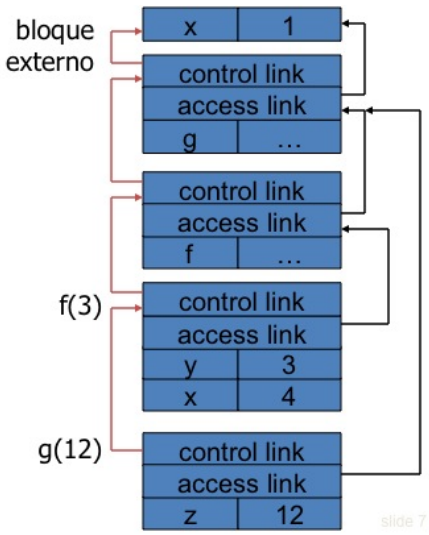


Figura 6.1: Estado de la pila de ejecución después de la llamada a la función `g` dentro del cuerpo de la función `f`.

del programa. Para los bloques anidados in-line esta dirección de memoria es redundante con el control link, ya que el bloque más cercano que lo contiene siempre se corresponde con el activation record que se ha apilado inmediatamente antes.

Para las funciones, sin embargo, el bloque más cercano que la contiene viene determinado por el lugar donde se declara la función. Debido a que el punto de declaración a menudo es diferente del punto en el que se llama a una función, el access link generalmente apunta a un registro de activación diferente al enlace de control.

Echemos un vistazo a los registros de activación y enlaces de acceso para el código del ejemplo anterior. La figura 6.1 muestra la pila de ejecución después de la llamada a la función `g` dentro del cuerpo de `f`.

Como ya sabemos, los control links apuntan al activation record previo en la pila, y los dibujamos a la izquierda para dejar espacio para los access links a la derecha. El access link para cada bloque apunta al activation record del bloque más cercano que lo contiene en el texto del programa.

He aquí algunos puntos importantes acerca de esta ilustración, que sigue nuestra convención de comenzar un nuevo bloque para cada declaración ML:

- La declaración de `g` se produce dentro del alcance de la declaración de `x`. Por lo tanto, el access link para la declaración de `g` apunta al registro de activación para la declaración de `x`.
- La declaración de `f` está dentro del alcance de la declaración de `g`. Por lo tanto, el access link para la declaración de `f` apunta al registro de activación para la declaración de `g`.

- La llamada `f(3)` crea un activation record asociado al alcance del cuerpo de `f`. El cuerpo de `f` ocurre dentro del alcance de la declaración de `f`. Por lo tanto, el access link para `f(3)` apunta al activation record para la declaración de `f`.
- La llamada `g(12)` crea un activation record asociado al alcance del cuerpo de `g`. El cuerpo de `g` ocurre dentro del alcance de la declaración de `g`. Por lo tanto, el access lin para `g(12)` apunta al activation record para la declaración de `g`.
- Evaluamos la expresión `x + z` sumando el valor del parámetro `z`, almacenado localmente en el registro de activación de `g`, y el valor de la variable global `x`. Encontramos la ubicación de la variable global `x` siguiendo el access link de la activación de `g` al activation record asociado a la declaración de `g`. Luego seguimos el access link en ese activation record para encontrar el activation record que contiene la variable `x`.

6.3.3. Ejercicios

6.1. Si el resultado del siguiente programa es 19, qué tipo de alcance tiene el lenguaje de programación en el que está escrito?

```
val x = 4;
    fun f(y) = x*y;
    fun g(x) = let
        f(3) + x;
    g(7);
```

a) estático

b) dinámico

6.2. Diagrame los estados de la pila de ejecución en los diferentes momentos de la ejecución del siguiente programa, mostrando cómo se apilan y desapilan los diferentes activation records a medida que se va ejecutando el programa. Puede representar los activation records con la información de variables locales y control links. Añada en el activation record la información necesaria para poder recuperar la información de la función, con *alcance estático*.

```
int sum(int n){
    if(n==0)
        return n;
    else
        return n+sum(n-1);
}
sum(4);
```

6.3. Diagrame como en el ejercicio anterior.

```
bool isPalindrome(char* s, int len)
{
  if(len < 2)
    return TRUE;
  else
    return s[0] == s[len-1] && isPalindrome(&s[1], len-2);
}
isPalindrome(``ada'')
```

6.4. Diagrame como en el ejercicio anterior.

```
  fact(n) = if n<=1 then 1 else n*fact(n-1); fact(4);
```

6.5. Diagrame como en el ejercicio anterior, pero ahora con *alcance dinámico*.

```
var x=1;
function g(z) {return x+z;}
function f(y) {
  var x = y+1;
  return g(y*x);
}
f(3)
```

6.6. Explique el funcionamiento del siguiente programa en pseudocódigo con alcance estático y con alcance dinámico. Ayúdese de diagramas de active records si le resulta útil.

```
  local D R Div in
      D=5
      fun Div(X)
         X div D
      end
      R = Div(10)
      Print(R)
      local D=0 R2 in
         R2 = Div(10)
         Print(R2)
      end
   end
```

6.7. Diagrame la pila de ejecución del siguiente programa en Bash, incluyendo la estructura de control links y access links, asumiendo que el lenguaje de programación tiene alcance estático.

```
x=1
function g () { echo $x ; x=2 ; }
function f () { local x=3 ; g ; }
f # does this print 1, or 3?
echo $x
```

En realidad este programa está escrito en Bash, que tiene alcance dinámico. Justifique por qué puede ser que Bash se haya diseñado con lenguaje dinámico.

Qué imprimirá este programa, ahora que sabemos que tiene alcance dinámico?

6.8. Reescriba el siguiente programa para que sea un programa en C estándar (sin usar gcc con `-fnested-functions`) con la misma semántica.

```
void f(void)
{
    // Define a function called g
    void g(void)
{
}
    // Call g
    g();
}
```

6.9. Reescriba el siguiente programa en Python para que sea válido en Perl en modo estricto, poniendo atención al lugar de declaración de la variable, para que su alcance sea exactamente el mismo en ambos programas.

```
if c:
    a = 'foo'
else:
    a = ''
```

6.10. Determine si los siguientes lenguajes tienen alcance estático o alcance dinámico: `Python`, `Haskell`, `Java`, `Scala`, `C`, `Javascript` y `Ruby`. Para cada uno de estos lenguajes, cree un programa (simple) que lo demuestre.

6.4. Recursión a la cola

Mitchell 7.3.4

En este apartado nos fijamos en una optimización del compilador que se llama eliminación de la recursión a la cola. Para las funciones recursivas a la cola, que describimos a continuación, se puede reutilizar un activation record para una llamada recursiva a la función. Esto reduce la cantidad de espacio de la pila que usa una función recursiva, y

evita llegar a problemas por límites de hardware como el llamado *stack overflow*, en el que la ejecución de un programa requiere más espacio del que hay disponible en la pila.

El principal concepto de lenguajes de programación que necesitamos entender es el concepto de la llamada a la cola. Supongamos que la función `f` llama a la función `g`. `f` y `g` podrían ser diferentes funciones o la misma, haciendo una llamada recursiva. Una llamada a `f` en el cuerpo de `g` es una llamada de la cola si `g` devuelve el resultado de la llamada `f` sin ningún cálculo adicional. Por ejemplo, en la función

```
fun g(x) = if x=0 then f(x) else f(x)*2
```

la primera llamada a `f` en el cuerpo de `g` es una llamada de la cola, ya que el valor de retorno de `g` es exactamente el valor de retorno de la llamada a `f`. La segunda llamada a `f` en el cuerpo de `g` no es una llamada de la cola porque `g` realiza un cálculo que involucra el valor de retorno de `f` antes de que `g` retorne.

Una función `f` es recursiva de cola si todas las llamadas recursivas en el cuerpo de `f` son llamadas a la cola a `f`.

Veamos como ejemplo la función recursiva a la cola que calcula el factorial:

```
1 fun factcola(n,a) = if n <= 1 then a else factcola(n-1, n *
    a);
```

Para cualquier entero positivo `n`, `factcola (n, a)` devuelve `n!`. Podemos ver que `factcola` es una función recursiva de cola porque la única llamada recursiva en el cuerpo de `factcola` es una llamada a la cola.

La ventaja de la recursión de cola es que podemos utilizar el mismo activation record para todas las llamadas recursivas. Considere la llamada `factcola(3,1)`. La figura 6.2 muestra las partes de cada activation record en el cómputo que son relevantes para la discusión.

6.4.1. Ejercicios

6.1. Diagrame los estados de la pila de ejecución en los diferentes momentos de la ejecución del siguiente programa, mostrando cómo se apilan y desapilan los diferentes activation records a medida que se va ejecutando el programa. Diagrame en dos versiones: una con un compilador que NO optimice las llamadas a la cola y otra con un compilador que sí las optimice, sin una optimización específica para llamadas recursivas a la cola, es decir, sin *overlay*, pero sí consiguiendo que el tamaño de la pila se mantenga constante.

```
1 def fact(n,acc):
    if n==0:
3     return acc
    return fact(n-1, acc*n)
5 fact(4,1)
```

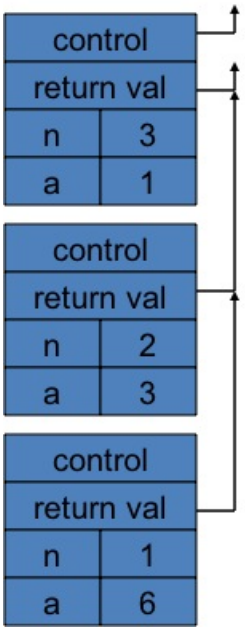


Figura 6.2: Estado de la pila después de tres llamadas a `factcola` sin optimización del compilador.

6.5. Alto orden

Mitchell 7.4

6.5.1. Funciones de primera clase

Un lenguaje tiene funciones de primera clase si las funciones pueden ser

- declaradas dentro de cualquier alcance,
- pasadas como argumentos a otras funciones y,
- devueltas como resultado de funciones.

En un lenguaje con funciones de primera clase y con alcance estático, un valor de función se representa generalmente por una clausura, un par formado por un puntero al código del cuerpo de una función y otro puntero a un activation record, con una semántica prácticamente equivalente a un access link.

Veamos un ejemplo de función en ML que requiere a otra función como argumento:

```
1 fun map (f, nil) = nil
  | map(f, x::xs) = f(x) :: map(f, xs)
```

La función `map` toma una función `f` y una lista `m` como argumentos y aplica `f` a cada elemento de `m` en orden. El resultado de `map(f, m)` es la lista de resultados `f(x)` para elementos `x` de la lista `m`. Esta función es útil en muchos programas en los que se usan listas. Por ejemplo, si tenemos una lista de tiempos de vencimiento para una secuencia de eventos y queremos incrementar cada tiempo de vencimiento, podemos hacerlo pasando una función de incremento a `map`.

Veremos por qué se necesitan las clausuras teniendo en cuenta las interacciones entre el alcance estático y las funciones como argumentos y como valores de retorno. C y C ++ no admiten clausuras porque tratan de mantener el costo computacional del compilador al mínimo, y las clausuras involucran un overhead no despreciable. Sin embargo, la implementación de objetos en C ++ y otros lenguajes se relaciona con la aplicación de valores de función tal como se explican en este capítulo. La razón es que un clausura y un objeto ambos combinan los datos y el código de la función.

6.5.2. Pasar Funciones a otras Funciones

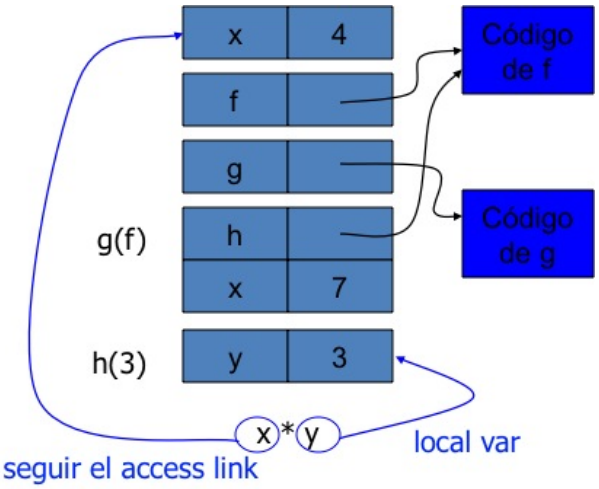
Vamos a ver que cuando una función `f` se pasa a una función `g`, es posible que tengamos que pasar también la clausura de `f`, que contiene un puntero a su activation record. Cuando `f` se llama dentro del cuerpo de `g`, el puntero de entorno de la clausura se utiliza para determinar cuál es el access link que tiene que guardarse en el activation record de la llamada a `f`. La necesidad de clausuras en este contexto se conoce como el problema de *downward funarg*, porque sucede cuando se pasan funciones como argumentos hacia abajo en alcances anidados.

Vamos a ilustrar los principales aspectos de este problema con un programa de ejemplo con dos declaraciones de una variable `x` y una función `f` que se pasa a otra función `g`:

```
val x = 4;
fun f(y) = x*y;
fun g(h) = let val x=7 in h(3) + x;
g(f);
```

En este programa, el cuerpo de `f` contiene una variable global `x` que se declara fuera del cuerpo de `f`. Cuando `f` se llama dentro `g`, el valor de `x` se tiene que recuperar desde el activation record asociado al bloque exterior. De lo contrario el cuerpo de `f` se evalúa con la variable `x` local declarada dentro de `g`, lo cual es inconsistente con la semántica de alcance estático.

Podemos ver el problema de la búsqueda de variables en la pila de ejecución después de la llamada a `f` desde la invocación de `g`.



Esta ilustración simplificada muestra sólo los datos contenidos en cada activation record. En esta ilustración, la expresión `x * y` del cuerpo de `f` se muestra en la parte inferior, el activation record asociado a la invocación de `f` (a través del parámetro formal `h` de `g`). Como muestra la ilustración, la variable `y` es local para la función y por lo tanto se puede encontrar en el activation record actual. Sin embargo, la variable `x` es global, y se encuentra varios registros de activación por encima del actual. Para encontrar las variables globales tenemos que seguir los access links, por lo tanto el access link del activation record inferior debe permitirnos alcanzar el activation record de la parte superior de la ilustración.

Cuando las funciones se pasan a otras funciones, hay que guardar el access link para el registro de la activación de cada función para que podamos encontrar las variables globales de esa función según la semántica del alcance estático. Y para poder cubrir este requisito necesitamos extender alguna estructura de datos de tiempo de ejecución. Esa estructura de datos serán las clausuras.

La solución estándar para mantener el alcance estático cuando las funciones se pasan como argumentos a otras funciones o se devuelven como resultados es utilizar una estructura de datos llamada clausura. La clausura es un par formado por un puntero a la parte de la memoria donde se guarda el código de la función y otro puntero a un activation record. Debido a que cada activation record contiene un access link señalando al activation record del bloque del texto del programa más cercano que lo contiene, un puntero al alcance en el que se declara una función también proporciona enlaces al activation record del bloque que la contiene.

Cuando se pasa como parámetro una función a otra función, el valor real que se pasa es un puntero a una clausura. A partir de una clausura, una función se llama de la siguiente forma:

6.1. Asignar un activation record para la función que se llama, como es habitual.

6.2. Establecer el access link en el activation record utilizando el puntero del activation record de la clausura.

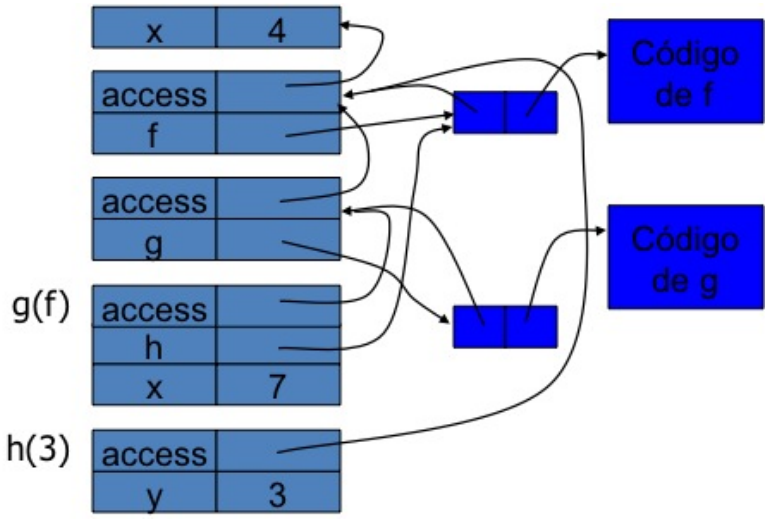


Figura 6.3: Cómo se establecen los access links desde la clausura.

Podemos ver cómo esto resuelve el problema de acceso a variables para funciones que se pasan a otras funciones como argumentos. Para ello vamos a diagramar el estado de la pila, con los diferentes activation records, cuando se ejecuta el programa anterior, tal como se muestra en la figura 6.3.

Podemos entender la figura 6.3 atravesando la secuencia de pasos de ejecución que conducen a la configuración que se muestra en la figura:

6.1. Declaración de `x`: se apila un activation record para el bloque en el que se declara `x`. El activation record contiene un valor para `x` y un control link que no se muestra.

6.2. Declaración de `f`: se apila un activation record para el bloque en ue se declara `f`. Este activation record contiene un puntero a la representación en tiempo de ejecución de `f`, que es una clausura con dos punteros. El primer puntero apunta al activation record para el alcance estático de `f`, que es el activation record para la declaración de `f`. El segundo puntero apunta al código de verb—f—, que fue producido durante la compilación y se coloca en alguna ubicación de memoria que conoce el compilador.

6.3. Declaración de `g`: Al igual que con la declaración de `f`, se apila un activation record para el bloque en el que se declaró `g`. Este activation record contiene un puntero a la representación en tiempo de ejecución de `g`, que es una clausura.

6.4. Llamada a `g(f)`: La llamada hace que se apile un activation record para la función `g`. El tamaño y estructura de este record los determina el código para `g`. El access link se establece hacia el activation record para el alcance donde se declara `g`; el access link apunta al mismo activation record que el activation record que hay en la clausura de `g`. El activation record tiene espacio para el parámetro `h` y la variable local `x`. Debido a que el parámetro real es la clausura de `f`, el valor del parámetro `h` es un puntero a

la clausura de `f`. La variable local `x` tiene un valor de 7, que viene dado por el código fuente.

6.5. Llamada a `h(3)`: El mecanismo para la ejecución de esta llamada es el punto principal de este ejemplo. Debido a que `h` es un parámetro formal de `g`, el código de `g` se compila sin saber dónde se declara la función `h`. Como resultado, el compilador no puede establecer el access link para el activation record para la llamada `h(3)`. Sin embargo, la clausura provee esa información: el access link para este activation record se establece mediante el puntero de activation record de la clausura de `h`. Debido a que el parámetro real es `f`, el acces link apunta al activation record para el alcance en el que se declaró `f`. Cuando se ejecuta el código de `f`, el access link se utiliza para encontrar `x`. En concreto, el código seguirá el access link hasta el segundo activation record de la ilustración, seguirá un access link adicional porque el compilador sabía, cuando generó el código para `f`, que la declaración de `x` se encuentra un alcance por encima de la declaración de `f`, y encontrará el valor 4 para la variable global `x` en el cuerpo de `f`.

Como se describe en el paso 5, la clausura de `f` hace posible que el código que se ejecuta encuentre el activation record que contiene la declaración global de `x`.

Cuando podemos pasar funciones como argumentos, los access links dentro de la pila forman un árbol. La estructura no es lineal, porque el activation record correspondiente a la llamada a la función `h(3)` tiene que saltar el activation record para `g(f)` para encontrar el `x` adecuado. Sin embargo, todos los access links apuntan hacia arriba. Por lo tanto, sigue siendo posible asignar y desasignar activation records mediante el uso de la disciplina de pila (último apilado, primero desapilado).

6.6. Excepciones

Mitchell 8.2

Las excepciones proporcionan una forma estructurada de salto que se puede utilizar para salir de una construcción tal como un bloque o llamada a función. El nombre de excepción sugiere que las excepciones deben ser utilizados en circunstancias excepcionales. Sin embargo, es muy habitual usarlas de formas muy poco excepcionales, como por ejemplo:

- salir de un bloque o llamada de función
- pasar datos como parte de un salto
- volver a un punto del programa fijado para continuar la computación

Además de saltar de un punto a otro del programa, también se asocia con las excepciones parte del manejo de memoria. Más en concreto, se pueden desapilar activation records que ya resulten innecesarios como resultado del salto.

Todo mecanismo de excepción incluye dos construcciones lingüísticas:

- levantar, *rise* o *throw*, una construcción para ejecutar una excepción, que corta parte del cómputo actual y provoca un salto (transferencia de control),
- un controlador, *handle* o *catch*, un mecanismo de control, que permite ciertas declaraciones, expresiones o llamadas a función provistos de código para responder a las excepciones que se lanzan durante la ejecución.

Hay varias razones por las que las excepciones se han convertido en construcciones muy aceptadas en la mayor parte de lenguajes. Muchos lenguajes no tienen ningún otro mecanismo limpio para saltar fuera de una llamada de función, por ejemplo, abortando la llamada. En lugar de utilizar instrucciones del tipo `go to`, que permiten crear código totalmente desestructurado (código *spaghetti*), muchos programadores prefieren usar excepciones, que se pueden utilizar para saltar sólo hacia la parte del programa que ya se ha cargado en la pila, no hacia alguna parte del programa que no se ha ejecutado todavía.

Las excepciones también permiten a un programador pasar datos como parte del salto, lo cual resulta muy útil si el programa trata de recuperarse de algún tipo de condición de error. Las excepciones proporcionan un método dinámico y útil de determinar hasta donde llega un salto. Si se declara más de un mecanismo de control, se establece cuál es el controlador adecuado en cada caso siguiendo una semántica de alcance dinámico, lo cual no sería fácil de lograr con otras formas de saltos de control.

La utilidad del alcance dinámico para estos casos se ilustra en el siguiente ejemplo, el cálculo de la inversa de una matriz de números reales. Calculamos la multiplicación de matrices, que se utiliza en el álgebra lineal, multiplicando las entradas de dos matrices. Si A es una matriz e I es la matriz identidad (con unos a lo largo de la diagonal y ceros en todas las otras posiciones), entonces la inversa A_{-1} de A es una matriz tal que el producto $AA_{-1} = I$. Una matriz cuadrada tiene una inversa si y sólo si algo que se llama el determinante de A no es igual a cero. Encontrar el determinante de una matriz requiere aproximadamente el mismo coste computacional que invertir una matriz.

Supongamos que escribimos una función para invertir matrices que tienen entradas de números reales. Como sólo se pueden invertir matrices con determinantes distintos de cero, no podemos calcular correctamente la inversa de una matriz si el determinante resulta ser cero. Una forma de manejar esta dificultad podría ser la de comprobar el determinante antes de llamar a la función de inversión. Pero esto no es una buena idea, porque calcular el determinante de una matriz es aproximadamente tanto trabajo como invertirla. Como podemos comprobar fácilmente el determinante cuando estamos tratando de convertir a la inversa, tiene más sentido tratar de invertir la matriz y lanzar una excepción en el caso de que detectemos un determinante cero. Esto lleva a un código estructurado de la siguiente manera:

```
exception Determinante; (* Declarar excepcion Determinante *)
fun invertir(aMatriz) =
  ...
  if ...
```

```
  then raise Determinante (* si el determinante de la matriz
   es cero *)
    else ...
  end;
invertir (miMatriz) handle Determinante ...;
```

En este ejemplo, la función `invertir` genera una excepción si el determinante de `aMatrix` resulta ser cero. Si la función levanta esta excepción como resultado de la llamada `invertir(myMatrix)`, entonces, debido a la declaración de `handle` asociada, la llamada a `invertir` se aborta y se transfiere el control al código que sigue inmediatamente al `handle Determinante`.

En este ejemplo, el mecanismo de excepción se utiliza para manejar una condición que hace que sea imposible continuar el cálculo. En concreto, si el determinante es cero, la matriz no tiene inversa, y es imposible para la función de inversión devolver la inversa de la matriz.

> **Pregunta 8:**
> *Si lo piensas bien, este ejemplo ilustra la necesidad de algún tipo de mecanismo de alcance dinámico. Más específicamente, supongamos que hay varias llamadas para invertir en un programa que hace cálculos matriciales. Cada una de estas llamadas podrían levantar la excepción* `Determinante`*. Cuando una llamada levanta esta excepción, donde nos gustaría manejar la excepción?*

La motivación principal para una semántica de alcance dinámico de los *handles* de excepción es que el código en el que se llama a la función `invertir` es el mejor lugar para decidir qué hacer si el determinante es cero. Si las excepciones tuvieran alcance estático, entonces al lanzar una excepción se saltaría al controlador de excepción que se declaró léxicamente (en el texto del programa) antes de la definición de la función `invert`. Si `invert` está en una librería de programas, entonces el controlador tendría que estar también en la librería. Sin embargo, la persona que escribió la librería no tiene idea de qué hacer en los casos en que se produce la excepción. Por este motivo, las excepciones están en el alcance de forma dinámica: cuando se produce una excepción, el control salta al controlador que se estableció por última vez en la historia dinámica de llamadas del programa. Vamos a ver cómo funciona esto con más detalle en algunos mecanismos de excepción específicos.

6.6.1. Ejercicios

6.1. En las siguientes funciones en ML:

```
exception Excpt of int;
fun twice(f,x) = f(f(x)) handle Excpt(x) => x;
fun pred(x) = if x = 0 then raise Excpt(x) else x-1;
fun dumb(x) = raise Excpt(x);
fun smart(x) = 1 + pred(x) handle Excpt(x) => 1;
```

Cuál es el resultado de evaluar cada una de las siguientes expresiones?

a) `twice(pred,1)`

b) `twice(dumb,1)`

c) `twice(smart,1)`

Explique qué excepción se levanta en cada caso y dónde se levanta.

6.2. Java tiene una construcción lingüística llamada "`try... catch... finally...`" cuya semántica consiste en que el bloque de código bajo el alcance de `finally` se ejecuta siempre, tanto si se lanza una excepción bajo el alcance de `try` como si no se lanzó. Si se lanza una excepción, la excepción funciona exactamente igual que si no existiera el bloque "`finally`" (aunque el bloque `finally` se ejecuta en cualquier caso), si se puede capturar mediante el `catch` se captura, y si no sigue buscando hasta que encuentra un `catch` que la pueda capturar. Sabiendo esto, explique qué se imprime si ejecutamos los siguientes tres programas y por qué.

```
class Ejemplo1 {
   public static void main(String args[]){
     try{
        System.out.println("primera sentencia del bloque
   try");
        int num=45/0;
        System.out.println(num);
     }
     catch(ArrayIndexOutOfBoundsException e){

   System.out.println("ArrayIndexOutOfBoundsException");
     }
     finally{
        System.out.println("bloque finally");
     }
     System.out.println("fuera del bloque
   try-catch-finally");
   }
}
```

```
class Ejemplo2 {
   public static void main(String args[]){
      try{
         System.out.println("primera sentencia del bloque
   try");
         int num=45/0;
         System.out.println(num);
```

```
        }
        catch(ArithmeticException e){
            System.out.println("ArithmeticException");
        }
        finally{
            System.out.println("bloque finally");
        }
        System.out.println("fuera del bloque
    try-catch-finally");
    }
}
```

```
class Ejemplo3 {
  public static void main(String args[]){
    try{
        System.out.println("primera sentencia del bloque
    try");
        int num=45/3;
        System.out.println(num);
    }
    catch(ArrayIndexOutOfBoundsException e){

    System.out.println("ArrayIndexOutOfBoundsException");
    }
    finally{
        System.out.println("bloque finally");
    }
    System.out.println("fuera del bloque
    try-catch-finally");
  }
}
```

6.3. En este fragmento de código se captura una excepción y se vuelve a lanzar la misma excepción. Explique qué sucede en este fragmento de código, ayudándose de un diagrama de una secuencia de estados de la pila de ejecución, mostrando cómo se apilan y desapilan los diferentes *activation records* a medida que se va ejecutando el programa. Para mayor claridad, puede acompañarlo de una descripción verbal. Describa también verbalmente cuál sería el objetivo de este programa. ¿Qué sentido tiene capturar una excepción para volver a lanzarla? ¿Es eso lo único que se hace?

```
ITransaccion transaccion = null;
try
{
 transaccion = sesion.EmpiezaTransaccion();
```

```
 // hacer algo
 transaccion.Commit();
}
catch
{
 if (transaccion != null) {
   transaccion.RestaurarEstadoPrevio(); }
 throw;
}
```

6.4. En el siguiente código java, inserte en el `main` los mecanismos de manejo de excepciones necesarios para capturar la excepción de forma adecuada.

```
public class RepartirComensales
{
   public static void main(String [] args)
   {
      Mesa mesa = new Mesa(1);
      System.out.println("vamos a llenar la mesa 1...");
      mesa.aniadirComensal("Ana");
      mesa.aniadirComensal("Juan");
      mesa.aniadirComensal("Maria");
      mesa.aniadirComensal("Pedro");
      mesa.aniadirComensal("Juana");
      mesa.aniadirComensal("Esteban");
      mesa.aniadirComensal("Lola");
    }
}

public class Mesa
{
   private int numeroDeComensales;
   private int numeroDeMesa;
   public Mesa(int numeroDeMesa)
   {
      this.numeroDeMesa = numeroDeMesa;
   }
   public void aniadirComensal(string comensal) throws
   ExcepcionMesaLlena
   {
      if(numeroDeComensales > 5)
      {
         throw new ExcepcionMesaLlena(numeroDeComensales)
```

```
        }
        else
        {
            numeroDeComensales += 1;
        }
    }
}
```

6.5. Escriba el siguiente programa con excepciones en lugar de tratar manejo de errores mediante estructuras de control.
Nota: no es necesario que el programa compile, puede usar pseudocódigo siempre que su semántica sea inambigua.

```
codigoErrorType readFile {
    initialize codigoError = 0;

    // abrir el archivo;
    if (elArchivoEstaAbierto) {
        // determinar la longitud del archivo;
        if (obtenemosLongitudArchivo) {
            // alojar esa cantidad de memoria;
            if (tenemosSuficienteMemoria) {
                // leemos el archivo a memoria;
                if (falloLectura) {
                    codigoError = -1;
                }
            } else {
                codigoError = -2;
            }
        } else {
            codigoError = -3;
        }
        // cerrar el archivo;
        if (elArchivoNoSeCerro && codigoError == 0) {
            codigoError = -4;
        } else {
            codigoError = codigoError and -4;
        }
    } else {
        codigoError = -5;
    }
    return codigoError;
}
```

6.6. En el siguiente código java, inserte en el `main` los mecanismos de manejo de excepciones necesarios para capturar la excepción de forma adecuada.

```
public class BankDemo
{
   public static void main(String [] args)
   {
      CuentaBancaria c = new CuentaBancaria(101);
      System.out.println("Depositando $500...");
      c.depositar(500.00);
      System.out.println("\nRetirando $100...");
      c.retirar(100.00);
      System.out.println("\nRetirando $600...");
      c.retirar(600.00);
    }
}
```

```
public class CuentaBancaria
{
   private double balance;
   private int numero;
   public CuentaBancaria(int numero)
   {
      this.numero = numero;
   }
   public void depositar(double cantidad)
   {
      balance += cantidad;
   }
   public void retirar(double cantidad) throws
   ExcepcionFondosInsuficientes
   {
      if(cantidad <= balance)
      {
         balance -= cantidad;
      }
      else
      {
         double necesita = cantidad - balance;
         throw new ExcepcionFondosInsuficientes(necesita);
      }
   }
```

```
25      public double getBalance()
        {
27          return balance;
        }
29      public int getNumber()
        {
31          return numero;
        }
33  }
```

6.7. La siguiente función de Python añade un elemento a una lista dentro de un diccionario de listas. Modifíquelo usando excepciones para manejar un posible error de llaves (`KeyError`) si la lista con el nombre no existe todavía en el diccionario, en lugar de comprobar por adelantado si ya existe. Incluya una cláusula `finally` (que se ejecutará independientemente de si se lanza una excepción o no).

```
1  def aniadir_a_lista_en_diccionario(diccionario,
       nombrelista, elemento):
     if nombrelista in diccionario:
3        l = diccionario[nombrelista]
         print("%s ya tiene %d elementos." % (nombrelista,
       len(l)))
5    else:
         diccionario[nombrelista] = []
7        print("Creamos %s." % nombrelista)

9    diccionario[nombrelista].append(elemento)

11   print("Aniadimos %s a %s." % (elemento, nombrelista))
```

6.7. Recolección de basura (*garbage collection*)

6.7.1. Ejercicios

6.1. Para los puntos en el código marcados con `(A)`, `(B)` y `(D)` reporte qué variables pueden ser recolectadas por el recolector de basura.

```
1  local X Y Z B R
       fun {M X}
3         proc {R Y} Y=X end
       end
5      P={M X}
       Q={M Y}                                  % (A)
```

```
    R=P
in
    X=2
    local A B C R=Q Z=foo(A f1:B C) in
       {R Z}                                % (B)
       {Browse Z}                           % (C)
    end
    {R Z}                                   % (D)
    Y=3
    {Browse Z}                              % (E)
end
```

6.2. En los puntos indicandos con `<----`, indique cuáles son las variables que pueden ser recolectadas si el recolector de basura (garbage collector) se ejecuta.

```
fun {Mergesort Xs}
 case Xs
      of nil then nil
         [] [X] then [X]
       else Ys Zs in
           {Split Xs Ys Zs}
                        <----
           {Merge {Mergesort Ys}
                       {Mergesort Zs}}
  end
end

Z = [1 2 3 4 5]
XS = {Mergesort  1 | 2 | 3 | Z }
{Browse Z}
          <----
{Browse XS}
          <----
```

6.3. Escriba en pseudocódigo un ejemplo de programa donde hay variables que se pueden recoletar semánicamente pero no sintácticamente.

6.4. Escriba en pseudocódigo un ejemplo de programa donde hay variables que, aunque por la estructura de bloques se podrían recolectar, no se pueden recolectar porque tienen una referencia de otra variable. Luego, modifique el programa anterior insertando una sentencia "`dereference`" que elimine todas las referencias a una variable. Ahora, la variable quedará disponible para recolección de basura? Qué problema puede acarrear el uso de una sentencia como `dereference`? Compare ese problema con el problema de los `memory leaks`, y opine sobre la gravedad de cada uno de ellos.

6.5. En el siguiente código en Ruby, cuándo se puede recolectar la memoria asociada a `obj`?

```
module Base
 def click(element_hash)
  begin
   obj =
   @browser.element(element_hash.keys[0],element_hash.values[0])
   obj.when_present.click
   report_pass("Element successfully clicked")
  rescue=>exception
   report_fail("Failed to click on object. #{exception})
 end
end
```

6.6. En el siguiente programa en Java, qué variables pueden ser recolectadas por el recolector de basura, y cuándo? En su explicación, use el concepto de *alcanzabilidad*. **NOTA:** En Java, `obj = null` elimina exactamente una referencia a un objeto.

```
class A { int i = 5; }
class B { A a = new A(); }
class C {
   B b;
   public static void main(String args[]) {
      C c = new C();
      c.b = new B();
      c.b = null;
   }
```

6.7. Compare el overhead que se da en Garbage Collection con el que se da en Orientación a Objetos. En un caso, el overhead se encuentra en el acceso a estructuras de datos en memoria, en otro caso el overhead se dá por cambios de contexto y procesos de sistema. Cuál cree que proporciona un mayor beneficio en eficacia del programador, cuál cree que contribuye a la naturalidad para expresar soluciones informáticas, cuál cree que evita errores de programación, y por qué?

6.8. El método `finalize()` en Java contiene código que se ejecuta cuando el recolector de basura reclama a un objeto. La acción `obj = null;` elimina una referencia a un objeto. Cuál de estas dos acciones contribuye de forma más efectiva a la liberación de memoria del programa?

6.9. Comente el siguiente texto, explicando primero cuál es la función del *garbage collector* en un lenguaje de programación, y describiendo los problemas que se pueden encontrar con el uso de `finalize()`.

Many GC systems have a notion of "finalization..An object may be registered with the GC system so that when it is about to reclaim the object, it runs a function on the object that can perform necessary cleanups. Finalization is tricky. Some of the issues are:
When can an object actually be finalized? This is trickier than it first appears in the presence of some normally-desirable optimizing transformations.
In what thread, resource, or security context does a finalization function run?
What happens when registered objects reference each other?
What happens if a finalization function makes an object not be garbage any more?

6.10. El método `finalize()` en Java se ejecuta cuando el recolector de basura reclama un objeto. En el siguiente pedazo de código tenemos el uso del método `cleanup`. Imaginemos que tanto `cleanup()` como `finalize()` se usan para liberar los recursos asociados a `myObj`, por ejemplo, *file handlers*. Entonces, en cuál de los dos casos sabemos con exactitud en qué momento se liberan los recursos asociados a `myObj`?

```
1 MyClass myObj;
2
3 myObj = new MyClass();
4 // ...
5 myObj.finalize();
```

```
1  MyClass myObj;
2
3  try {
4      myObj = new MyClass();
5
6      // ...
7  } finally {
8
9      if (null == myObj) { // no hay ninguna referencia a
       myObj
10         myObj.cleanup();
11     }
12 }
```

Capítulo 7

Orientación a objetos

Mitchell 10., 11., 12.1, 12.2, 12.3, 13.1, 13.2

7.1. Ejercicios

7.1. En las siguientes declaraciones de clase, indique qué partes son implementación y qué partes son interfaz, marcando la interfaz.

```
public class URLExpSimple {

    public static void main(String[] args) {
        try {
            URL mySite = new
   URL("http://www.cs.utexas.edu/~scottm");
            URLConnection yc = mySite.openConnection();
            Scanner in = new Scanner(new
   InputStreamReader(yc.getInputStream()));
            int count = 0;
            while (in.hasNext()) {
                System.out.println(in.next());
                count++;
            }
            System.out.println("Number of tokens: " +
   count);
            in.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
public class Stopwatch
{
    private long startTime;
    private long stopTime;

    public static final double NANOS_PER_SEC =
   1000000000.0;

  public void start()
  { startTime = System.nanoTime();  }

  public void stop()
  { stopTime = System.nanoTime(); }

  public double time()
  { return (stopTime - startTime) / NANOS_PER_SEC;  }

  public String toString(){
      return "elapsed time: " + time() + " seconds.";
  }

  public long timeInNanoseconds()
  { return (stopTime - startTime);  }
}
```

```
public class MineSweeper
{ private int[][] myTruth;
  private boolean[][] myShow;

  public void cellPicked(int row, int col)
  { if( inBounds(row, col) && !myShow[row][col] )
    { myShow[row][col] = true;

      if( myTruth[row][col] == 0)
      { for(int r = -1; r <= 1; r++)
          for(int c = -1; c <= 1; c++)
            cellPicked(row + r, col + c);
      }
    }
  }

  public boolean inBounds(int row, int col)
```

```
  { return 0 <= row && row < myTruth.length && 0 <= col
   && col < myTruth[0].length;
  }
}
```

```
public class PrimeEx {

public static void main(String[] args) {
  printTest(10, 4);
  printTest(2, 2);
  printTest(54161329, 4);
  printTest(1882341361, 2);
  printTest(36, 9);

  System.out.println(isPrime(54161329) + " expect false");
  System.out.println(isPrime(1882341361) + " expect
   true");
  System.out.println(isPrime(2) + " expect true");
  int numPrimes = 0;
  Stopwatch s = new Stopwatch();
  s.start();
  for(int i = 2; i < 10000000; i++) {
    if(isPrime(i)) {
      numPrimes++;
    }
  }
  s.stop();
  System.out.println(numPrimes + " " + s);
  s.start();
  boolean[] primes = getPrimes(10000000);
  int np = 0;
  for(boolean b : primes)
    if(b)
      np++;
  s.stop();
  System.out.println(np + " " + s);

  System.out.println(new BigInteger(1024, 10, new
   Random()));
}

public static boolean[] getPrimes(int max) {
  boolean[] result = new boolean[max + 1];
```

```
  for(int i = 2; i < result.length; i++)
38    result[i] = true;
  final double LIMIT = Math.sqrt(max);
40  for(int i = 2; i <= LIMIT; i++) {
    if(result[i]) {
42    // cross out all multiples;
      int index = 2 * i;
44    while(index < result.length){
        result[index] = false;
46       index += i;
      }
48  }
  }
50  return result;
}
52

54public static void printTest(int num, int
   expectedFactors) {
  Stopwatch st = new Stopwatch();
56  st.start();
  int actualFactors = numFactors(num);
58  st.stop();
  System.out.println("Testing " + num + " expect " +
   expectedFactors + ", " +
60      "actual " + actualFactors);
  if(actualFactors == expectedFactors)
62    System.out.println("PASSED");
  else
64    System.out.println("FAILED");
  System.out.println(st.time());
66}

68// pre: num >= 2
public static boolean isPrime(int num) {
70  assert num >= 2 : "failed precondition. num must be >=
   2. num: " + num;
  final double LIMIT = Math.sqrt(num);
72  boolean isPrime = (num == 2) ? true : num % 2 != 0;
  int div = 3;
74  while(div <= LIMIT && isPrime) {
    isPrime = num % div != 0;
76    div += 2;
```

```
  }
78  return isPrime;
}
80
// pre: num >= 2
82 public static int numFactors(int num) {
  assert num >= 2 : "failed precondition. num must be >=
   2. num: " + num;
84  int result = 0;
  final double SQRT = Math.sqrt(num);
86  for(int i = 1; i < SQRT; i++) {
    if(num % i == 0) {
88    result += 2;
    }
90  }
  if(num % SQRT == 0)
92    result++;
  return result;
94 }
```

7.2. Dibuje la jerarquía de clases en que se basa el siguiente código.

```
 class Vehicle {
2 public:
     explicit
4    Vehicle( int topSpeed )
     : m_topSpeed( topSpeed )
6    {}
     int TopSpeed() const {
8       return m_topSpeed;
     }
10
     virtual void Save( std::ostream& ) const = 0;
12
 private:
14   int m_topSpeed;
 };
16
 class WheeledLandVehicle : public Vehicle {
18 public:
     WheeledLandVehicle( int topSpeed, int numberOfWheels
   )
20   : Vehicle( topSpeed ), m_numberOfWheels(
```

```
    numberOfWheels )
      {}
      int NumberOfWheels() const {
        return m_numberOfWheels;
      }

      void Save( std::ostream& ) const; // is implicitly
    virtual

  private:
      int m_numberOfWheels;
  };

  class TrackedLandVehicle : public Vehicle {
  public:
     TrackedLandVehicle ( int topSpeed, int numberOfTracks
    )
      : Vehicle( topSpeed ), m_numberOfTracks (
    numberOfTracks )
      {}
      int NumberOfTracks() const {
         return m_numberOfTracks;
      }
      void Save( std::ostream& ) const; // is implicitly
    virtual

   private:
      int m_numberOfTracks;
   };
```

```
class DrawableObject
 {
  public:
    virtual void Draw(GraphicalDrawingBoard&) const = 0;
   //draw to GraphicalDrawingBoard
 };

 class Triangle : public DrawableObject
 {
 public:
   void Draw(GraphicalDrawingBoard&) const; //draw a
   triangle
 };
```

```
12
   class Rectangle : public DrawableObject
14 {
   public:
16   void Draw(GraphicalDrawingBoard&) const; //draw a
     rectangle
   };
18
   class Circle : public DrawableObject
20 {
   public:
22   void Draw(GraphicalDrawingBoard&) const; //draw a
     circle
   };
24
   typedef std::list<DrawableObject*> DrawableList;
26
   DrawableList drawableList;
28 GraphicalDrawingBoard drawingBoard;

30 drawableList.pushback(new Triangle());
   drawableList.pushback(new Rectangle());
32 drawableList.pushback(new Circle());

34 for(DrawableList::const_iterator iter =
     drawableList.begin(),
      endIter = drawableList.end();
36    iter != endIter;
      ++iter)
38 {
     DrawableObject *object = *iter;
40   object->Draw(drawingBoard);
   }
```

7.3. Identifique en el siguiente código en C++ un problema con la herencia del miembro `meow`.

```
1 class Felino {
  public:
3  void meow() = 0;
  };
5
  class Gato : public Felino {
```

```
public:
 void meow() { std::cout << "miau\n"; }
};

class Tigre : public Felino {
public:
 void meow() { std::cout << "ROARRRRRR\n"; }
};

class Ocelote : public Felino {
public:
 void meow() { std::cout << "roarrrrr\n"; }
};
```

7.4. A partir del siguiente código Ruby hemos tratado de escribir un código Java con la misma semántica, pero los resultados no son iguales. Cuál es la diferencia y por qué? Cómo deberíamos modificar el programa en Java para que haga lo mismo que el programa en Ruby?

```
#!/usr/bin/ruby

class Being

    @@count = 0

    def initialize
        @@count += 1
        puts "creamos un ser"
    end

    def show_count
        "Hay #{@@count} seres"
    end

end

class Human < Being

   def initialize
       super
       puts "creamos un humano"
   end
end
```

```
25
class Animal < Being
27
    def initialize
29      super
        puts "creamos un animal"
31  end
end
33
class Dog < Animal
35
    def initialize
37      super
        puts "creamos un perro"
39  end
end
41
Human.new
43 d = Dog.new
puts d.show_count
```

```
class Being {
2
    private int count = 0;
4
    public Being() {
6       count++;
        System.out.println("creamos un ser");
8   }

10  public void getCount() {
        System.out.format("hay %d seres%n", count);
12  }
}
14 class Human extends Being {

16  public Human() {
        System.out.println("creamos un humano");
18  }
}
20 class Animal extends Being {

22  public Animal() {
```

```
        System.out.println("creamos un animal");
    }
}
class Dog extends Animal {

    public Dog() {
        System.out.println("creamos un perro");
    }
}

public class Inheritance2 {

    @SuppressWarnings("ResultOfObjectAllocationIgnored")
    public static void main(String[] args) {
        new Human();
        Dog dog = new Dog();
        dog.getCount();
    }
}
```

7.5. A partir de la siguiente `template` en C++, escriba una clase de C++ con la misma semántica pero específica para `int`.

```
template <class A_Type> class calc
{
  public:
    A_Type multiply(A_Type x, A_Type y);
    A_Type add(A_Type x, A_Type y);
};
template <class A_Type> A_Type
   calc<A_Type>::multiply(A_Type x,A_Type y)
{
  return x*y;
}
template <class A_Type> A_Type calc<A_Type>::add(A_Type
   x, A_Type y)
{
  return x+y;
}
```

7.6. Los `mixins` son una construcción de Ruby que permite incorporar algunas de las funcionalidades de la herencia múltiple, ya que Ruby es un lenguaje con herencia simple. Con un `mixin` se pueden incluir en una clase miembros de otra clase, con

la palabra clave `include`. Los *name clashes*, si los hay, se resuelven por el orden de los `include`, de forma que la última clase añadida prevalece, y sus definiciones son las que se imponen en el caso de conflicto. Teniendo esto en cuenta, describa el comportamiento del siguiente pedazo de código.

```
module EmailReporter
    def send_report
      # Send an email
    end
  end

  module PDFReporter
    def send_report
      # Write a PDF file
    end
  end

  class Person
  end

  class Employee < Person
    include EmailReporter
    include PDFReporter
  end

  class Vehicle
  end

  class Car < Vehicle
    include PDFReporter
    include EmailReporter
  end
```

7.7. Reescriba el siguiente código en Java en Haskell, de forma que su semántica denotacional sea idéntica.

```
import core.List;
import static core.List.*;

import java.util.NoSuchElementException;

class P01 {

    <T> T f1(List<T> list) {
```

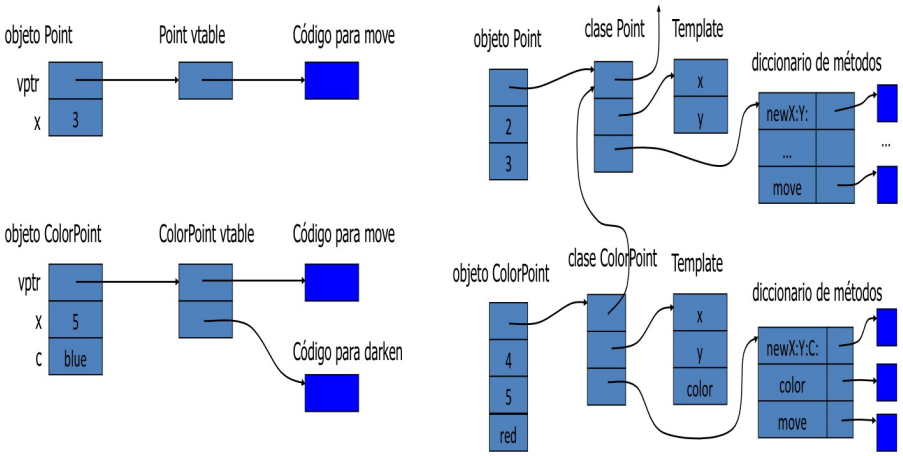
```
        if (list.isEmpty()) throw new
   NoSuchElementException("List is empty");
        List<T> elements = list.tail();
        List<T> result = list;
        while (elements.nonEmpty()) {
            result = elements;
            elements = elements.tail();
        }
        return result.head();
    }
```

7.8. El siguiente ejemplo en C++ causa un error de compilación. Por qué? cómo puede solucionarse?

```
class trabajador
{
  public:
     void hora_de_levantarse( )
     { .... ... .... }
};
class estudiante
{
    void hora_de_levantarse( )
     { .... ... .... }
};
class ayudante_alumno : public trabajador, public
   estudiante
{

};

int main()
{
    ayudante_alumno obj;

    obj.hora_de_levantarse( )
}
```

7.9. Explique qué tests realizaría para saber cuál es la visibilidad de un método *protected* en Ruby.

7.10. Explique por qué Smalltalk es menos eficiente que C++ utilizando el concepto de *overhead*. Puede ayudarse de los siguientes gráficos, en los que se muestra el proceso de lookup de información asociada a los objetos en uno y otro lenguaje.



7.11. En el siguiente código el compilador está creando una instancia de una subclase anónima de la clase abstracta, y luego se está invocando al método de la clase abstracta. Explique por qué es necesario que el compilador haga este proceso.

```
abstract class my {
    public void mymethod() {
        System.out.print("Abstract");
    }
}

class poly {
    public static void main(String a[]) {
        my m = new my() {};
        m.mymethod();
    }
}
```

7.12. En el siguiente programa en C++ indique si hay *name clashes*. Si los hay, identifíquelos y mencione alguna política para solucionarlos. Si no los hay, introdúzcalos y añada los mecanismos para solucionarlos (no es necesario que sea el mecanismo que efectivamente implementa C++, puede ser otra política). ¿En qué contexto se dan los *name clashes*?

```
class Persona
{
private:
```

```
4       std::string m_strNombre;
        int m_nEdad;
6       bool m_bEsVaron;

8   public:
        Persona(std::string strNombre, int nEdad, bool
    bEsVaron)
10          : m_strNombre(strNombre), m_nEdad(nEdad),
    m_bEsVaron(bEsVaron)
        {
12      }

14      std::string GetNombre() { return m_strNombre; }
        int GetEdad() { return m_nEdad; }
16      bool EsVaron() { return m_bEsVaron; }
    };
18
    class Empleado
20  {
    private:
22      std::string m_strEmpleador;
        double m_dSalario;
24
    public:
26      Empleado(std::string strEmpleador, double dSalario)
            : m_strEmpleador(strEmpleador),
    m_dSalario(dSalario)
28      {
        }
30
        std::string GetEmpleador() { return m_strEmpleador; }
32      double GetSalario() { return m_dSalario; }
    };
34
    class Profesor: public Persona, public Empleado
36  {
    private:
38       int m_nDictaGrado;

40  public:
        Profesor(std::string strNombre, int nEdad, bool
    bEsVaron,
42      std::string strEmpleador, double dSalario, int
```

```
    nDictaGrado)
          : Persona(strNombre, nEdad, bEsVaron),
44        Empleado(strEmpleador, dSalario),
          m_nDictaGrado(nDictaGrado)
46    {
      }
48 };
```

7.13. Indique si el siguiente programa en C++ se podría escribir en Java con una semántica equivalente. Si no se puede, indique por qué. Si se puede, indique con qué recursos del lenguaje se puede.

```
class Persona
2 {
private:
4     std::string m_strNombre;
      int m_nEdad;
6     bool m_bEsVaron;

8 public:
      Persona(std::string strNombre, int nEdad, bool
    bEsVaron)
10          : m_strNombre(strNombre), m_nEdad(nEdad),
    m_bEsVaron(bEsVaron)
      {
12    }

14    std::string GetNombre() { return m_strNombre; }
      int GetEdad() { return m_nEdad; }
16    bool EsVaron() { return m_bEsVaron; }
};
18
class Empleado
20 {
private:
22    std::string m_strEmpleador;
      double m_dSalario;
24
public:
26    Empleado(std::string strEmpleador, double dSalario)
          : m_strEmpleador(strEmpleador),
    m_dSalario(dSalario)
28    {
```

```
    }

    std::string GetEmpleador() { return m_strEmpleador; }
    double GetSalario() { return m_dSalario; }
};

class Profesor: public Persona, public Empleado
{
private:
     int m_nDictaGrado;

public:
    Profesor(std::string strNombre, int nEdad, bool
   bEsVaron,
     std::string strEmpleador, double dSalario, int
   nDictaGrado)
         : Persona(strNombre, nEdad, bEsVaron),
         Empleado(strEmpleador, dSalario),
         m_nDictaGrado(nDictaGrado)
    {
    }
};
```

7.14. En Java existen dos formas de crear un nuevo *thread* explícitamente: creando una subclase de la clase `Thread` o bien implementando la interfaz `Runnable`. En ambos casos el nuevo *thread* debe implementar el método `run`, que contendrá el código ejecutado por el *thread*. Cuál de estas dos opciones crea un objeto más versátil y por qué?

7.15. Estos dos programas, en dos lenguajes distintos, tienen la misma semántica. ¿Cuáles son las diferencias entre estos dos lenguajes? Refiérase a los elementos de los programas que ejemplifican las diferencias entre estos dos lenguajes. Cuando sea relevante, argumente las ventajas y desventajas de la forma de expresión de cada lenguaje.

```
interface Iterable<T>
  def each(&block : T ->)
end

interface Addable<T>
  def +(other : T)
end

def sum(values : Iterable<T>) where T : Addable<T>
  count = 0
```

```
  values.each do |value|
    count += value
  end
  count
end
```

```
def sum(values)
  count = 0
  values.each do |value|
    count += value
  end
  count
end
```

7.16. El lenguaje de programación Scala implementa como parte del lenguaje una heurística llamada "linearización", que permite determinar qué implementación se usará en un objeto que hereda una componente con el mismo nombre de diferentes clases. Explique por qué es necesario que el lenguaje implemente esta heurística (qué problema resuelve) y dé un ejemplo donde se muestre una configuración donde entraría en juego esta heurística para evitar el problema, por ejemplo, la configuración conocida como el problema diamante.

7.17. Java implementa un tipo de pasaje de parámetros que consiste en pasar por valor la referencia a un objeto. Mediante este tipo de pasaje de parámetros, ¿Se puede implementar la función *swap* para que intercambie el objeto A por el objeto B? ¿Se puede implementar la función *swap* para intercambiar algún elemento, algún pedazo de software? ¿Cuál?

7.18. En muchos lenguajes de programación no se implementa herencia múltiple, sino que se usan mecanismos alternativos. En este ejemplo de Scala, explique cómo se usa la construcción `trait` de una forma parecida a las interfaces de Java o los mixins de Ruby para conseguir una expresividad parecida a la herencia múltiple de C++.

```
trait Cantante{
  def cantar { println(" cantando ... ") }
  //mas metodos
}

class Persona{
  def explicar {  println (" humano ") }
  //mas metodos
}

class Actor extends Persona with Cantante
```

```
class Actor extends Cantante with Performer
```

7.19. El enlace dinámico (o *dynamic dispatch*) es un mecanismo de la programación orientada a objetos por el cual se selecciona con qué método o función se responderá a un determinado mensaje en tiempo de ejecución. Es necesario hacer esto en tiempo de ejecución porque es imposible saber en tiempo de compilación cuál será el tipo del mensaje recibido. En C++ el tipo de *dispatch* por defecto es estático, y para que se pueda realizar *dynamic dispatch* hay que declarar un método como `virtual`. Los métodos virtuales se implementan mediante una estructura de datos que se llama `vtable` o `virtual table`, que define la correspondencia entre mensajes y métodos para cada clase. Comente la utilidad y coste del *dynamic dispatch* en términos de overhead y flexibilidad. Establezca un paralelismo con el polimorfismo de tipos mediante un ejemplo.

7.20. En el siguiente ejemplo, explique la diferencia entre `extends` e `implements` en Java, y explique cómo se pueden usar las interfaces para aproximar la expresividad de herencia múltiple evitando *name clashes*.

```
2  public class ExtendsAndImplementsDemo{
       public static void main(String args[]){
4
           Dog dog = new Dog("Tiger",16);
6          Cat cat = new Cat("July",20);

8          System.out.println("Dog:"+dog);
           System.out.println("Cat:"+cat);
10
           dog.remember();
12         dog.protectOwner();
           Learn dl = dog;
14         dl.learn();

16         cat.remember();
           cat.protectOwner();
18
           Climb c = cat;
20         c.climb();

22         Man man = new Man("Ravindra",40);
           System.out.println(man);
24
           Climb cm = man;
26         cm.climb();
```

```
        Think t = man;
        t.think();
        Learn l = man;
        l.learn();
        Apply a = man;
        a.apply();

    }
}

abstract class Animal{
    String name;
    int lifeExpentency;
    public Animal(String name,int lifeExpentency ){
        this.name = name;
        this.lifeExpentency=lifeExpentency;
    }
    public void remember(){
        System.out.println("Define your own remember");
    }
    public void protectOwner(){
        System.out.println("Define your own
   protectOwner");
    }

    public String toString(){
        return
   this.getClass().getSimpleName()+":"+name+":"+lifeExpentency;
    }
}
class Dog extends Animal implements Learn{

    public Dog(String name,int age){
        super(name,age);
    }
    public void remember(){

   System.out.println(this.getClass().getSimpleName()+"
   can remember for 5 minutes");
    }
    public void protectOwner(){

   System.out.println(this.getClass().getSimpleName()+ "
```

```
    will protect owner");
     }
66     public void learn(){

    System.out.println(this.getClass().getSimpleName()+ "
    can learn:");
68     }
}
70 class Cat extends Animal implements Climb {
     public Cat(String name,int age){
72         super(name,age);
     }
74     public void remember(){

    System.out.println(this.getClass().getSimpleName() + "
    can remember for 16 hours");
76     }
     public void protectOwner(){
78

    System.out.println(this.getClass().getSimpleName()+ "
    won't protect owner");
     }
80     public void climb(){

    System.out.println(this.getClass().getSimpleName()+ "
    can climb");
82     }
}
84 interface Climb{
     public void climb();
86 }
interface Think {
88     public void think();
}
90

interface Learn {
92     public void learn();
}
94 interface Apply{
     public void apply();
96 }

98 class Man implements Think,Learn,Apply,Climb{
```

```
    String name;
100     int age;

102     public Man(String name,int age){
            this.name = name;
104         this.age = age;
        }
106     public void think(){
            System.out.println("I can
    think:"+this.getClass().getSimpleName());
108     }
        public void learn(){
110         System.out.println("I can
    learn:"+this.getClass().getSimpleName());
        }
112     public void apply(){
            System.out.println("I can
    apply:"+this.getClass().getSimpleName());
114     }
        public void climb(){
116         System.out.println("I can
    climb:"+this.getClass().getSimpleName());
        }
118     public String toString(){
            return "Man :"+name+":Age:"+age;
120     }
}
```

7.21. En Go existen interfaces con características semejantes a las de Java, pero con una diferencia:

> *With how Go interfaces work, you don't need to declare an interface implementation. If you implement the proper methods, you implement the interface.*

Relacione esta propiedad con las políticas de subtipado de diferentes lenguajes orientados a objetos, que pueden estar implementadas en herencia o en inclusión entre interfaces.

7.22. omente el siguiente texto en el que se compara subtipado e interfaces, y explique por qué un la herencia y el subtipado son distintos y qué ventajas o desventajas puede tener separarlos o usarlos juntos.

Inheritance is about gaining attributes (and/or functionality) of super types. For example:

```
class Base {
    //interface with included definitions

}

class Derived inherits Base {
    //Add some additional functionality.
    //Reuse Base without having to explicitly
   forward
    //the functions in Base
}
```

Here, a Derived cannot be used where a Base is expected, but is able to act similarly to a Base, while adding behaviour or changing some aspect of Bases behaviour. Typically, Base would be a small helper class that provides both an interface and an implementation for some commonly desired functionality.

Subtype-polymorphism is about implementing an interface, and so being able to substitute different implementations of that interface at run-time:

```
class Interface {
    //some abstract interface, no definitions
   included
}

class Implementation implements Interface {
    //provide all the operations
    //required by the interface
}
```

Here, an Implementation can be used wherever an Interface is required, and different implementations can be substituted at run-time. The purpose is to allow code that uses Interface to be more widely useful.

Your confusion is justified. Java, C#, and C++ all conflate these two ideas into a single class hierarchy. However, the two concepts are not identical, and there do exist languages which separate the two.

Capítulo 8

Frameworks de programación

Un *framework de software* es una abstracción en la cual un software que provee una funcionalidad genérica se puede modificar en algunos puntos mediante código escrito por el usuario. De esta forma se instancia un software genérico en uno específico para una aplicación determinada.

Un framework de software provee una forma estándar de construir y desplegar aplicaciones. Es un entorno de software reusableq ue provee una funcionalidad particular como parte de una plataforma de software más grande, facilitando el desarrollo de una familia de aplicaciones de software. Pueden incluir programas de soporte, compiladores, librerías de código, herramientas y APIs, en resumen, los diferentes componentes necesarios para desarrollar un proyecto o sistema.

Los frameworks tienen algunas características clave que los distinguen de las librerías normales:

- inversión de control: en un framework, a diferencia de las librerías o aplicaciones estándar, el flujo de control del programa no está dirigido por el programa que lo llama, sino por el framework.
- extensibilidad: un usuario puede extender el framework - normalmente mediante sobreescritura de algunas partes seleccionadas, o los programadores pueden añadir código especializado para proveer alguna funcionalidad específica.
- el código del framework es no modificable: el código del framework, en general, no se puede modificar, sólo extender.

8.1. Inversión de Control

También conocida como *el principio de Howllywood*: no nos llame, nosotros lo llamaremos a usted.

La inversión de control es un principio de diseño por el cual un programador escribe código que recibe el flujo de control de un framework genérico. Una arquitectura de software

con este diseño invierte el control en comparación a la programación procedural tradicional. En la programación tradicional, el código que escribe el programador llama a librerías que se encargan de tareas genéricas. En cambio, con la inversión de control es el framework el que llama al código escrito por el programador.

La inversión de control se usa para aumentar la modularidad del programa y hacerlo extensible. Este término está relacionado al principo de *inversión de dependencia*, que se encarga de desacoplar dependencias entre niveles altos y niveles bajos del código a través de abstracciones compartidas.

El concepto general también se relaciona con la *programación dirigida por eventos* (*event-driven programming*), ya que ésta se implementa normalmente usando inversión de control. Efectivamente, en la programación dirigida por eventos el código escrito por el programador sólo se encarga de tratar con los eventos, mientras que el framework se encarga de manejar la ocurrencia de los eventos, el envío de eventos o mensajes o el entorno de ejecución en general.

En pocas palabras, la inversión de control convierte un código escrito secuencialmente en una estructura de delegación. En lugar de que tu programa controle todo explícitamente, el programa configura una clase o librería con algunas funciones específicas para tu aplicación, que se llamarán cuando sucedan algunos eventos específicos de tu aplicación.

Esta aproximación reduce la duplicación de código. Por ejemplo, con programación tradicional un programador tenía que escribir el código para manejar la ocurrencia de eventos, o buscar las librerías de sistema para nuevos eventos. Hoy en día, la mayor parte de APIs modernas permiten que el programador simplemente declare qué eventos le interesan y qué librerías necesita, y la API te notifica cuándo suceden los eventos.

La inversión de control se ha simplificado en muchos lenguajes a través del concepto de delegados, interfaces o incluso punteros de función crudos.

Como todos los paradigmas, la inversión de control no es adecuada para todos los problemas. Puede resultar difícil seguir el flujo de un programa escrito en inversión de control. Sin embargo, es una forma útil de diseñar métodos cuando se escribe una librería que se va a reusar.

8.1.1. Desacoplamiento

Sin desacoplamiento:

```
public class ServerFacade {
    public <K, V> V respondToRequest(K request) {
        if (businessLayer.validateRequest(request)) {
            DAO.getData(request);
            return Aspect.convertData(request);
        }
        return null;
    }
}
```

Este esquema básico en Java da un ejemplo de código con inversión de control. Pero vemos que en `ServerFacade` se hacen muchas asunciones sobre cómo son los datos que devuelve el objeto de acceso a los datos (`DAO`). Estas asunciones son probablmente ciertas en el momento de creación del código, pero en la evolución del código `ServerFacade` y `DAO` pueden cambiar de forma independiente, y perder ese acoplamiento.

Si se usa inversión de control de forma total, el control lo tiene totalmente el objeto `DAO`, y el código se convierte en:

```
public class ServerFacade {
    public <K, V> V respondToRequest(K request, DAO dao) {
        return dao.getData(request);
    }
}
```

8.1.2. Boilerplate

Boilerplate es un término que se usa para nombrar porciones del código que se usan en muchos lugares con poca o ninguna alteración. Se usa más frecuentemente para referirse a lenguajes que se consideran verbosos.

En programas orientados a objetos, en general hay clases que ya vienen con métodos para obtener y configurar variables de instancia. Las definiciones de estos métodos en generalse consideran boilerplate. Aunque el código puede variar de una clase a otra, es lo suficientemente predecible en su estructura como para ser generado automáticamente, mejor que escrito a mano. Por ejemplo, en la siguiente clase Java que representa a una mascota, casi todo el código es boilerplate excepto por las declaraciones de Mascota, nombre y propietario.

```
public class Pet {
    private PetName name;
    private Person owner;

    public Pet(PetName name, Person owner) {
        this.name = name;
        this.owner = owner;
    }

    public PetName getName() {
        return name;
    }

    public void setName(PetName name) {
        this.name = name;
```

```
    }
17
    public Person getOwner() {
19      return owner;
    }
21
    public void setOwner(Person owner) {
23      this.owner = owner;
    }
25  }
```

8.2. Ejercicios

8.1. Argumente cuál de los dos fragmentos de código que siguen es un ejemplo de inversión de control.

```
class PaymentsController < ApplicationController
  def accept_payment
    if Rails.env.development? || Rails.env.test?
      @credit_card_validator = BogusCardValidator.new
    else
      @credit_card_validator = RealCardValidator.new
    end
    if Rails.env.production?
      @gateway = RealPaymentGateway.new
    elsif Rails.env.staging?
      @gateway = RealPaymentGateway.new(use_testing_url:
   true)
    else
      @gateway = BogusPaymentGateway.new
    end
    card = @credit_card_validator.validate(params[:card])
    @gateway.process(card)
  end
end
```

```
class PaymentsController < ApplicationController
  def accept_payment
    card = @credit_card_validator.validate(params[:card])
    @gateway.process(card)
  end
end
```

```
# config/dependencies/production.rb:
RailsENV::Dependencies.define do
  prototype :payment_gateway,       RealPaymentGateway
  prototype :credit_card_validator, RealCardValidator
  controller PaymentsController, {
    gateway:                ref(:payment_gateway)
    credit_card_validator: ref(:credit_card_validator)
  }
end

# config/dependencies/staging.rb:
RailsENV::Dependencies.define do
  inherit_environment(:production)
  prototype :payment_gateway, RealPaymentGateway,
   {use_testing_url: true}
end

# config/dependencies/development.rb:
RailsENV::Dependencies.define do
  inherit_environment(:production)
  singleton :payment_gateway, BogusPaymentGateway
  singleton :credit_card_validator, BogusCardValidator
end
```

8.2. En el código que sigue, explique por qué este es un ejemplo de *boilerplate* y desarrolle (implemente) cómo el programador podría usarlo para instanciar un caso particular.

```
<!DOCTYPE html>
<html>
   <head>
      <title></title>
   </head>
   <body> </body>
</html>
```

Capítulo 9

Paradigma funcional

Mitchell 4.4.
Van Roy y Haridi. 2004. Concepts, Techniques, and Models of Computer Programming. MIT Press. Capítulo 3: introducción

9.1. Expresiones imperativas vs. expresiones funcionales

En muchos lenguajes de programación, las construcciones básicas son imperativas, del mismo modo que lo sería una oración imperativa en lenguaje natural, como por ejemplo "*Traeme esa manzana.*". Por ejemplo, esta instrucción de asignación:

```
x: = 5
```

es una orden que el programador le da a la computadora para guardar el valor 5 en un lugar determinado.

Los lenguajes de programación también contienen construcciones declarativas, como la declaración de la función

```
function f(int x) { return x+1; }
```

que describe un hecho, de la misma forma que en lenguaje natural podríamos decir "*La tierra es redonda.*". En este ejemplo, lo que describimos es que "*f es una función cuyo valor de retorno es 1 mayor que su argumento.*".

La distinción entre construcciones imperativas y declarativas se basa en que las imperativas cambian un valor y las declarativas declaran un nuevo valor. Por ejemplo, en el siguiente fragmento de programa:

```
{ int x = 1;
  x = x+1;
  { int y = x+1;
    { int x = y+1;
}}}
```

sólo la segunda línea es una operación imperativa, las otras líneas contienen declaraciones de nuevas variables.

Un punto sutil es que la última línea en el código anterior declara una nueva variable con el mismo nombre que el de una variable declarada anteriormente. La forma más sencilla de entender la diferencia entre declarar una nueva variable y cambiar el valor de una variable ya existente es cambiando el nombre de la variable. Las variables ligadas, que no son libres en una expresión (que están definidas dentro del alcance de la expresión) pueden cambiar de nombre sin cambiar el significado de la expresión. En particular, podemos cambiar el nombre de las variables ligadas en el fragmento de programa anterior de la siguiente manera:

```
{ int x = 1;
  x = x+1;
  { int y = x+1;
    { int z = y+1;
}}}
```

(Si hubiera más ocurrencias de x dentro del bloque interior, también les cambiaríamos el nombre a z.) Después de volver a escribir el programa a esta forma equivalente, podemos ver fácilmente que la declaración de una nueva variable z no cambia el valor de cualquier variable ya existente.

La asignación imperativa puede introducir **efectos secundarios** porque puede destruir el valor anterior de una variable, sustituyéndolo por uno nuevo, de forma que éste no esté disponible más adelante. En programación funcional la asignación imperativa se conoce como asignación o actualización destructiva.

Decimos que una operación computacional es declarativa si, cada vez que se invoca con los mismos argumentos, devuelve los mismos resultados independientemente de cualquier otro estado de computación. Una operación declarativa es:

independiente no depende de ningún estado de la ejecución por fuera de sí misma,

sin estado no tiene estados de ejecución internos que sean recordados entre invocaciones, y

determinística siempre produce los mismos resultados para los mismos argumentos y, por extensión, ejecutar la operación no tendrá efectos secundarios.

Una consecuencia muy valiosa de estas propiedades es la conocida como *transparencia referencial*. Una expresión es transparente referencialmente si se puede sustituir por su valor sin cambiar la semántica del programa. Esto hace que todas las componentes declarativas, incluso las más complejas, se puedan usar como valores en un programa, por ejemplo, como argumentos de función, como resultados de función o como partes de estructuras de datos. Esto aporta una gran flexibilidad a los lenguajes funcionales, ya que se pueden tratar de forma homogénea expresiones de estructura muy variable, como por ejemplo en el siguiente programa:

```
HayAlgunExceso xs n = foldr ( filter (>n) ( map
   convertirSistemaMetrico xs) ) False xs
```

Veamos un ejemplo de dos funciones, una referencialmente transparente y otra referencialmente opaca:

```
globalValue = 0;

integer function rq(integer x)
begin
  globalValue = globalValue + 1;
  return x + globalValue;
end

integer function rt(integer x)
begin
  return x + 1;
end
```

La función *rt* es referencialmente transparente, lo cual significa que `rt(x) = rt(y)` si `x = y`. Sin embargo, no podemos decir lo mismo de `rq` porque usa y modifica una variable global. Por ejemplo, si queremos razonar sobre la siguiente aserción:

```
integer p = rq(x) + rq(y) * (rq(x) - rq(x));
```

Podríamos querer simplificarla de la siguiente forma:

```
integer p = rq(x) + rq(y) * (0);
integer p = rq(x) + 0;
integer p = rq(x);
```

Pero esto no es válido para `rq()` porque cada ocurrencia de `rq(x)` se evalúa a un valor distinto. Recordemos que el valor de retorno de `rq` está basado en una variable global que no se pasa como parámetro de la función y que se modifica en cada llamada a `rq`. Esto implica que no podemos asegurar la veracidad de aserciones como $x - x = 0$.

Por lo tanto, la transparencia referencial nos permite razonar sobre nuestro código de forma que podemos construir programas más robustos, podemos encontrar errores que no podríamos haber encontrado mediante testing y podemos encontrar posibles optimizaciones que podemos trasladar al compilador.

9.2. Propiedades valiosas de los lenguajes funcionales

Los lenguajes funcionales, como Lisp o ML, realizan la mayor parte del cómputo mediante expresiones con declaraciones de funciones. Todos los lenguajes funcionales tienen también construcciones imperativas, pero en esos lenguajes se pueden escribir programas muy completos sin usarlas.

En algunos casos se usa el término "lenguaje funcional" para referirse a lenguajes que no tienen expresiones con efectos secundarios o cualquier otra forma de construcción imperativa. Para evitar ambigüedades entre estos y lenguajes como Lisp o ML, llamaremos a estos últimos "lenguajes funcionales puros". Los lenguajes funcionales puros pueden pasar el siguiente test:

> Dentro del alcance de las declaraciones $x_1, ..., x_n$, todas las ocurrencias de una expresión e que contenga sólo las variables $x_1, ..., x_n$ tendrán el mismo valor.

Como consecuencia de esta propiedad, los lenguajes funcionales puros tienen una propiedad muy útil: si la expresión e ocurre en varios lugares dentro de un alcance específico, entonces la expresión sólo necesita evaluarse una vez. Esto permite que el compilador pueda hacer optimizaciones de cálculo y de espacio en memoria (aunque no necesariamente todos los compiladores de todos los lenguajes funcionales lo van a hacer).

Otra propiedad interesante de los programas declarativos es que son **composicionales**. Un programa declarativo consiste de componentes que pueden ser escritos, comprobados, y probados correctos independientemente de otros componentes y de su propia historia pasada (invocaciones previas).

Otra propiedad interesante es que **razonar** sobre programas declarativos es sencillo. Es más fácil razonar sobre programas escritos en el modelo declarativo que sobre programas escritos en modelos más expresivos. Como los programas declarativos sólo pueden calcular valores, se pueden usar técnicas sencillas de razonamiento algebraico y lógico.

En un programa declarativo, la interacción entre componentes se determina únicamente por las entradas y salidas de cada componente. Considere un programa con un componente declarativo. Este componente puede ser mirado en sí mismo, sin tener que entender el resto del programa. El esfuerzo que se necesita para entender el programa completo es la suma de los esfuerzos que se necesitan para entender el componente declarativo y para entender el resto.

Si existiera una interacción más estrecha entre el componente y el resto del programa no se podrían entender independientemente. Tendrían que entenderse juntos, y el esfuerzo requerido sería mucho más grande. Por ejemplo, podría ser (aproximadamente) proporcional al producto de los esfuerzos requeridos para entender cada parte. En un programa con muchos componentes que interactuán estrechamente, esto explota muy rápido, difcultando o haciendo imposible entenderlo. Un ejemplo de interacción estrecha es un programa concurrente con estado compartido.

Pero para algunos problemas, es necesario trabajar con interacciones estrechas. Sin embargo, podemos adoptar por principio la directiva de tratar de hacer código lo más modular posible, con interacciones estrechas sólo cuando sea necesario y no en cualquier caso. Para soportar este principio, el total de componentes declarativos debería ser el mayor número posible.

Estas dos propiedades son importantes tanto para programar en grande como en pequeño. Sería muy agradable si todos los programas pudieran escribirse en el modelo declarativo. Desafortunadamente, este no es el caso. El modelo declarativo encaja bien con ciertas clases de programas y mal con otras.

9.3. Problemas naturalmente no declarativos

La forma más elegante y natural (compacta e intuitiva) de representar algunos problemas es mediante estado explícito. Es el caso, por ejemplo, de las aplicaciones cliente-servidor y de las aplicaciones que muestran video. En general, todo programa que realice algún tipo de Input-Output lo hará de forma más natural mediante estado explícito. También es el caso en el que estamos tratando de modelar algún tipo de comportamiento (por ejemplo, agentes inteligentes, juegos) en el que representamos a una componente de software como una entidad que tiene *memoria*, porque inherentemente la memoria cambia a lo largo del tiempo.

> **Pregunta 1:**
> *Piense algunos ejemplos de problemas en los que el estado debería estar representado en la solución de software de forma explícita.*

También hay tipos de problemas que se pueden programar de forma mucho más eficiente si se hace de forma imperativa. En esos casos podemos llegar a convertir un problema que es **intratable** con programación declarativa en un problema **tratable**. Este es el caso de un programa que realiza modificaciones incrementales de estructuras de datos grandes, e.g., una simulación que modifica grafos grandes, que en general no puede ser compilado eficientemente. Sin embargo, si el estado está guardado en un acumulador y el programa nunca requiere acceder a un estado ya pasado, entonces el acumulador se puede implementar con asignación destructiva (ver Van Roy y Haridi 6.8.4.).

Por esta razón muchos lenguajes funcionales proveen algún tipo de construcción lingüística para poder expresar instrucciones imperativas. Incluso en lenguajes funcionales puros existen estas construcciones, que en general se conocen como mónadas. Las mónadas son una construcción de un lenguaje que permite crear un alcance aislado del resto del programa. Dentro de ese alcance, se permiten ciertas operaciones con efectos secundarios, por ejemplo, el uso de variables globales (globales al alcance) o asignación destructiva. Está garantizado que estos efectos secundarios no van a afectar a la parte del programa que queda fuera del alcance de la mónada. Las mónadas han sido descritas como un "punto y coma programable", que transportan datos entre unidades funcionales que los van transformando un paso a la vez.

Sabiendo todo esto, ¿podemos considerar que la programación declarativa es eficiente? Existe una distancia muy grande entre el modelo declarativo y la arquitectura de una computadora. La computadora está optimizada para modificar datos en su lugar, mientras que el modelo declarativo nunca modifica los datos sino que siempre crea datos nuevos. Pero sin embargo esto no es un problema tan grave como podría parecer, porque el compilador puede convertir partes importantes de los programas funcionales en instrucciones imperativas.

> **Pregunta 2:**
> *Piense dos ejemplos de operación declarativa y dos ejemplos de operación no declarativa*

Pregunta 3:
Piense por lo menos una operación que no se pueda llevar a cabo sin estado

9.4. Concurrencia declarativa

Una consecuencia muy valiosa de los programas escritos de forma funcional pura, sin expresiones con efectos secundarios, es que se pueden ejecutar de forma concurrente con otros programas con la garantía de que su semántica permanecerá inalterada. De esta forma, paralelizar programas funcionales puros es trivial. En cambio, para paralelizar programas imperativos, es necesario detectar primero las posibles regiones del programa donde puede haber condiciones de carrera, y, si es necesario, establecer condiciones de exclusión mútua.

Backus acuñó el término "*el cuello de botella von Neumann*" (*von Neumann bottleneck*) para referirse al hecho de que cuando ejecutamos un programa imperativo, la computación funciona de a un paso por vez. Como es posible que cada paso en un programa dependa del previo, tenemos que pasar el estado de la memoria (asignación de valores a variables) de la memoria a la CPU cada vez que ejecutamos un paso de la computación. Este canal secuencial entre la CPU y la memoria es el cuello de botella von Neumann.

Sin embargo, aunque paralelizar programas funcionales sea trivial, es difícil aprovechar este potencial en la práctica. Efectivamente, en algunos puntos de un programa tiene sentido introducir paralelismo y en otros no. El paralelismo tiene un pequeño coste (*overhead*) en la creación de nuevos procesos, incluso si son ligeros, y también en el posible cambio de contexto si hay un solo procesador. Por otro lado, si la ejecución de un proceso depende del resultado de otro, no se obtiene ningún beneficio al paralelizarlos. Es necesario entonces hacer algún tipo de análisis previo para que la paralelización de los programas declarativos sea realmente efectiva.

9.5. Ejercicios

9.1. Cuál de estas dos funciones, `rq` o `rt`, es transparente referencialmente?

```
globalValue = 0;

integer function rq(integer x)
begin
  globalValue = globalValue + 1;
  return x + globalValue;
end

integer function rt(integer x)
begin
  return x + 1;
end
```

9.2. El siguiente pedazo de código es declarativo? Por qué?

```
{ int x = 1;
  x = x+1;
  { int y = x+1;
    { int x = y+z;
}}}
```

9.3. Si una componente de software usa una variable global en una guarda (como parte de una condición con "if"), entonces no es independiente de contexto. Si, en cambio, una componente de software modifica una variable global pero ésta no afecta a su ejecución, conserva la transparencia referencial pero puede tener efectos secundarios.

Escriba dos programas en pseudocódigo, uno en el que se de el primer fenómeno (una variable global que hace que un programa no sea independiente de contexto, es decir, que los resultados de su ejecución no dependan solamente de sus parámetros de entrada), y otro en el que se de el segundo fenómeno (un programa que tiene efectos secundarios a través de una variable global).

9.4. De la misma forma que las variables globales, el pasaje de variables por referencia puede producir efectos secundarios. Sin embargo, en componentes de software declarativas no hay efectos secundarios, y no hay diferencia entre pasaje de parámetros por valor o por referencia. Explique cómo se puede dar este fenómeno, y cuál es la restricción lingüística que imponen los lenguajes que se inscriben en el paradigma funcional para forzar esta propiedad en los programas que se escriben en esos lenguajes, es decir, para forzar que el pasaje de parámetros no pueda ser una vía para propagar efectos secundarios.

9.5. En los siguientes programas imperativos, identifique porciones de código que no sean declarativas.

```
    int A;
  int B;

  int Add()
  {
    return A + B;
  }

  int main()
  {
    int answer;
    A = 5;
```

```
    B = 7;
    answer = Add();
    printf("%d\n",answer);
    return 0;
  }
```

```
int glob = 0;

int square(int x)
{
  glob = 1;
  return x*x;
}

int main()
{
  int res;
  glob = 0;

  res = square(5);
  res += glob;

  return res;
}
```

9.6. Reescriba el siguiente código en Java en un lenguaje declarativo, por ejemplo Haskell o pseudocódigo declarativo, de forma que su semántica denotacional sea idéntica.

```
class P01 {

    <T> T f1(List<T> list) {
        if (list.isEmpty()) throw new
   NoHayElemento("lista vacia");
        List<T> elements = list.tail();
        List<T> result = list;
        while (elements.nonEmpty()) {
            result = elements;
            elements = elements.tail();
        }
        return result.head();
    }
```

9.7. En el siguiente programa, identifique las porciones del programa que no son funcionales, y mencione por qué no lo son. Escriba en pseudocódigo un programa funcional con la misma semántica que este programa.

```
Module Modulo1
    Dim texto As String = "textoUno"

    Public Sub ConcatenaConGuion(ByVal textoAConcatenar
   As String)
        texto = texto & "-" & textoAConcatenar
    End Sub

    Sub Main()
        ConcatenaConGuion("textoDos")
    End Sub
End Module
```

9.8. Estos dos programas están escritos en Perl. ¿Cuál de los dos es funcional? Identifique en el programa no funcional las partes del código que no son funcionales y explique cómo se expresa la misma semántica en la versión funcional para que sea funcional.

```
my @numbers = qw(1 3 5 6 7 8);

my $sum = 0;
for my $num (@numbers) { $sum += $num;}

say $sum;
```

```
my @numbers = qw(1 3 5 6 7 8);

sub reduce (&@) {
    my ($code, $first, @rest) = @_;
    return $first unless @rest;

    return $code->(
        $first,
        reduce($code, @rest),
    );
}

sub sum {
    return reduce { $_[0] + $_[1] } @_;
}

```

```
say sum @numbers;$
```

9.9. Escriba un programa (puede ser en pseudocódigo) en el que una variable sea el medio por el cual se propagan efectos secundarios más allá del alcance de una función. Relacione este fenómeno con los paradigmas funcional e imperativo, incluyendo en su explicación los conceptos de determinismo en las componentes de software. Explique como este comportamiento que puede originar tantos comportamientos inesperados puede llegar a utilizarse de forma ventajosa en un esquema productor-consumidor.

9.10. En un programa imperativo cualquiera, identifique porciones de código que no sean declarativas.

9.11. En un programa imperativo cualquiera, identifique posibles canales por los que eventualmente propagarse efectos secundarios.

Capítulo 10

Programación lógica

Mitchell 15.

10.1. Ejercicios

10.1. Por qué el cut (`!`) no pertenece al paradigma declarativo?

a) porque no es un objetivo en prolog

b) porque tiene efectos secundarios

10.2. Cuáles de los siguientes pares de términos unifican? En el caso de que unifiquen, indique cómo se instancian las variables.

a) `pan = pan`

b) `'Pan' = pan`

c) `'pan' = pan`

d) `Pan = pan`

e) `pan = salchicha`

f) `comida(pan) = pan`

g) `comida(pan) = X`

h) `comida(X) = comida(pan)`

i) `comida(pan,X) = comida(Y,salchicha)`

j) `comida(pan,X,cerveza) = comida(Y,salchicha,X)`

k) `comida(pan,X,cerveza) = comida(Y,hamburguesa)`

l) `comida(X) = X`

m) `meal(comida(pan),bebida(cerveza)) = meal(X,Y)`

n) `meal(comida(pan),X) = meal(X,bebida(cerveza))`

10.3. A partir de la siguiente base de datos:

```
elfo_domestico(dobby).
bruja(hermione).
bruja('McGonagall').
bruja(rita_skeeter).
puede_hacer_magia(X):-  elfo_domestico(X).
puede_hacer_magia(X):-  hechicero(X).
puede_hacer_magia(X):-  bruja(X).
```

Cuáles de las siguientes consultas se satisfacen? Con qué instanciaciones de variables?

```
?-  puede_hacer_magia(elfo_domestico).
?-  hechicero(harry).
?-  puede_hacer_magia(hechicero).
?-  puede_hacer_magia('McGonagall').
?-  puede_hacer_magia(Hermione).
```

Dibuje el árbol de búsqueda para la consulta `puede_hacer_magia(Hermione).`

10.4. La búsqueda en Prolog es exhaustiva, mientras que en otros lenguajes se ejecuta la sentencia controlada por el primer patrón que hace pattern matching. Sin embargo, existe una forma en prolog para cortar una búsqueda exhaustiva, usando el operador `cut`, como se ve en el siguiente ejemplo:

```
max(X,Y,Y)  :-  X  =<  Y,!.
max(X,Y,X)  :-  X>Y.
```

Escriba un pseudocódigo en imperativo (usando `if ... then ... else ...`) o funcional (usando guardas) con la misma semántica.

10.5. Defina en Prolog el problema de recomendar "amigos" en una red social, especificado como sigue: para un usuario P con un conjunto de amigos $CA = [A_1, A_2, ...A_n]$, los cuales a su vez tienen cada uno asociado un conjunto de amigos C_1, C_2... C_n le recomendaremos a P todos los miembros de la unión de todos los conjuntos de amigos de sus propios amigos $CCA = C_1 \cup C_2 ... \cup C_n$ que NO están en la lista de sus propios amigos.

Pueden usar la siguiente base de conocimiento.

```
amigo(pedro,maría).      amigo(pedro,juan).
amigo(maría,pedro).      amigo(maría,clara).
amigo(maría,romina).     amigo(juan,pedro).
amigo(juan,clara).       amigo(clara,maría).
amigo(clara,juan).       amigo(romina,maría).
```

10.6. Defina en Prolog un sistema de ayuda a una central de turnos. Este sistema dirá si se le puede asignar un turno a un paciente con un determinado médico para un determinado día, siguiendo las siguientes premisas:

- el paciente no tiene ningún turno asignado para el mismo día a la misma hora.
- el paciente no tiene ningún turno asignado con ningún especialista de la misma especialidad para la que pide turno.
- el médico para el que pide turno no tiene turno asignado con ningún otro paciente para el mismo día a la misma hora.

Se puede definir con una sola regla!

La base de conocimiento puede ser como sigue:

```
turno(celia,rivas,(6,30,8)).
turno(celia,zilvetti,(7,14,11)).
turno(tomás,rivas,(7,11,10)).
turno(tomás,pérez,(8,11,10)).
turno(tomás,schuster,(9,11,10)).
turno(lidia,zilvetti,(7,14,10)).
turno(lidia,schuster,(9,11,11)).
turno(esteban,rivas,(7,1,9)).

especialidad(rivas,oftalmologia).
especialidad(smith,oftalmologia).
especialidad(zilvetti,ginecología).
especialidad(román,ginecología).
especialidad(pérez,endocrinología).
especialidad(schuster,clínico).
```

Capítulo 11

Programación concurrente

Mitchell 14.

11.1. Event-driven concurrency

```
function main
    initialize()
    while message != quit
        message := get_next_message()
        process_message(message)
    end while
end function
```

11.2. Ejercicios

11.1. En el siguiente ejemplo se está usando la variable común `lock` como lock. Describa una ejecución del código que lleve a un estado inconsistente, describiendo en orden las asignaciones y evaluaciones que ocurren al ejecutar el código. Explique cómo se podría evitar este funcionamiento no deseado con implementaciones atómicas de `wait` y `signal`.

```
lock := 0;
cobegin
  begin
    while lock=1 do end;
    lock:=1;
    sign up(fred);
    lock:=0;
  end;
  begin
```

```
    while lock=1 do end;    // loop que no hace nada hasta
   que el lock sea 0
    lock:=1;                // setear el lock para entrar
   a la seccion critica
    sign up(bill);          // seccion critica
    lock := 0;              // soltar el lock
  end;
end;
```

11.2. En java existen dos formas de crear un nuevo *thread* explícitamente: creando una subclase de la clase `Thread` o bien implementando la interfaz `Runnable`. En ambos casos el nuevo *thread* debe implementar el método `run`, que contendrá el código ejecutado por el *thread*. Cuál de estas dos opciones crea un objeto más versátil y por qué?

11.3. El siguiente es un programa concurrente en Erlang. Escriba su versión secuencial.

```
f1(0) ->
    0;
f1(1) ->
    1;
f1(N) ->
    Self = self(),
    spawn(fun() ->
      Self ! f0(N-1)
    end),
    spawn(fun() ->
      Self ! f0(N-2)
    end),
    receive
  F1 ->
      receive
    F2 ->
        F1 + F2
      end
    end.
   }
 }
```

11.4. Explique por qué el esquema productor - consumidor no tiene condiciones de carrera.

11.5. En lenguajes con evaluación perezosa la concurrencia sin condiciones de ca rrera puede implementarse mediante estructuras de datos. Explique cómo, a partir del siguiente ejemplo:

```
lista_dobles = map (\x -> 2 * x) lista_todos_los_naturales
```

11.6. Hay ocho combinaciones posibles en el pasaje de mensajes entre procesos concurrentes, combinando sincronicidad / asincronicidad, ordenado / no ordenado y con buffer / sin buffer. Explique cuáles tienen ventajas y cuáles no tienen sentido o tienen desventajas.

11.7. Los actores se comunican mediante mensajes desordenados. Explique qué información y métodos debería añadir a un modelo basado en actores para poder procesar los mensajes recibidos en el orden en el que fueron enviados.

11.8. Escriba en pseudocódigo dos versiones secuenciales (no concurrentes) del siguiente programa, una en paradigma funcional y otra en paradigma imperativo. Argumente las ventajas y desventajas de las tres versiones del programa usando los conceptos de *velocidad*, *overhead* y *determinismo*.

```
class Ejemplo extends RecursiveTask<Integer> {
  final int n;
  Ejemplo(int n) { this.n = n; }
  Integer compute() {
    if (n <= 1)
       return n;
    Ejemplo f1 = new Ejemplo(n - 1);
    f1.fork();
    Ejemplo f2 = new Ejemplo(n - 2);
    return f2.compute() + f1.join();
  }
}
```

11.9. En el código del ejercicio anterior, identifique las expresiones que expresan semántica exclusiva de un programa concurrente y describa su semántica informalmente.

11.10. En las siguientes interfaces y clases de Java, subraye las partes de la descripción que describen semántica concurrente.

BlockingQueue<E>	A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
Future<E>	A Future represents the result of an asynchronous computation.
PriorityBlockingQueue<E>	An unbounded blocking queue that uses the same ordering rules as class PriorityQueue and supplies blocking retrieval operations.
CyclicBarrier	A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.
SynchronousQueue<E>	A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa.

11.11. En el siguiente código Java, subraye las partes del programa que hacen referencia a semántica concurrente.

```
protected class ReadLock{
    WriteLock lock;

    public ReadLock(WriteLock lock) {
        super();
        this.lock = lock;
    }

    public void lock() throws InterruptedException { //If
  write() then wait
        synchronized (lock) {
            System.out.println("Waiting to read:
  "+Thread.currentThread());
            Thread callingThread = Thread.currentThread();
            while (lock.isLocked && lock.lockedBy !=
  callingThread) {
                lock.wait();
            }
            lock.readers++; //Increment writelocks readers
            System.out.println("Reading:
  "+Thread.currentThread());
        }
    }
```

```
20
    public void unlock() {
22      synchronized (lock) {
            lock.readers--; //Subtract from writelocks
   readers
24          lock.notify();
        }
26  }

28 }
```

11.12. En Java se puede escribir sincronización de métodos, pero esta funcionalidad es una forma más cómoda de escribir (el llamado *azúcar sintáctico*), porque en realidad el alcance de una sincronización es siempre el objeto entero. Sabiendo esto, en el siguiente código, ¿pueden dos threads acceder los métodos `addA()` y `addB()` simultáneamente? ¿Por qué? ¿Podemos decir que las variables `a` y `b` son regiones críticas o canales de comunicación entre dos threads?

```
class X {
2
    private int a;
4   private int b;

6   public synchronized void addA(){
        a++;
8   }

10  public synchronized void addB(){
        b++;
12  }

14 }
```

11.13. En Quake-C se usa un modelo de pasaje de mensajes. Acá mostramos algunos tipos de mensaje, su especificación y algunos ejemplos. Para los mensajes de tipo MSG_BROADCAST, MSG_ONE, MSG_ALL, diga cuáles de las tres características del pasaje de mensajes del modelo de actores tiene sentido usar: sincronización, buffer, orden.

```
MSG_BROADCAST = 0;  unreliable message, sent to all
2 MSG_ONE = 1;       reliable message, sent to msg_entity
MSG_ALL = 2;        reliable message, sent to all
4 MSG_INIT = 3; write to the init string
```

```
// Use unreliable (but fast) messages, when it's of no
   importance that a client misses the message.
// examples: sound, explosions, monster deaths, taunts....
// Use reliable messages when it's very important that
   every client sees the message, or a game incoherency
   might happen.
// examples: shots, player deaths, door moves, game ends
   ... and CD track changes!.

Message: Killed Monster

WriteByte (MSG_ALL, SVC_KILLEDMONSTER);
Increase by one the count of killed monsters, as
   available to the client. Can be displayed with
   showscores.

Message: Set View Angles

  msg_entity = player
  WriteByte (MSG_ONE, SVC_SETVIEWANGLES);
  WriteAngle( MSG_ONE, camera.angles_x);
  WriteAngle( MSG_ONE, camera.angles_y);
  WriteAngle( MSG_ONE, camera.angles_z);

Message: Final Message

  WriteByte (MSG_ALL, SVC_FINALE);
  WriteString (MSG_ALL, "any text you like\n");
```

11.14. En el siguiente código en React, tenemos un ejemplo de programación declarativa? Relacione este código y el framework React en general con el paradigma de actores y con otros frameworks.

```
import React, { PropTypes } from 'react'

const Todo = ({ onClick, completed, text }) => (
  <li
    onClick={onClick}
    style={{
      textDecoration: completed ? 'line-through' : 'none'
    }}
  >
    {text}
```

```
    </li>
)

Todo.propTypes = {
  onClick: PropTypes.func.isRequired,
  completed: PropTypes.bool.isRequired,
  text: PropTypes.string.isRequired
}

export default Todo
```

11.15. El siguiente código está dentro del modelo de actores. Señale en el código los mecanismos propios de actores y sírvase de ellos para explicar cómo los actores implementan concurrencia declarativa.

```
import akka.actor._

case object PingMessage
case object PongMessage
case object StartMessage
case object StopMessage

class Ping(pong: ActorRef) extends Actor {
  var count = 0
  def incrementAndPrint { count += 1; println("ping") }
  def receive = {
    case StartMessage =>
        incrementAndPrint
        pong ! PingMessage
    case PongMessage =>
        incrementAndPrint
        if (count > 99) {
          sender ! StopMessage
          println("ping stopped")
          context.stop(self)
        } else {
          sender ! PingMessage
        }
  }
}

class Pong extends Actor {
  def receive = {
```

```
    case PingMessage =>
      println("  pong")
      sender ! PongMessage
    case StopMessage =>
      println("pong stopped")
      context.stop(self)
  }
}

object PingPongTest extends App {
  val system = ActorSystem("PingPongSystem")
  val pong = system.actorOf(Props[Pong], name = "pong")
  val ping = system.actorOf(Props(new Ping(pong)), name =
   "ping")
  // start them going
  ping ! StartMessage
}
```

11.16. [**11 pt.**] En el siguiente lenguaje de programación, la palabra "`local`" se usa para declarar variables en un alcance, y el operador "$\rightarrow$" se usa para ligar la variable de la derecha a la de la izquierda, de forma que el valor de la variable de la izquierda será una referencia a la de la derecha. Todas las variables se representan en la pila como punteros a una estructura de datos en el heap.

Diagrame los diferentes estados por los que pasa la pila de ejecución y muestre en qué momento se puede recolectar cada variable. Señale también si hay casos de variables que podrían ser recolectadas antes de que queden sintácticamente disponibles para que el recolector de basura las recolecte.

```
{ local bli
  local bla
  { local ble
    local blu
    local bla
    bla <- ble
    { local blo
      ble <- blo
    }
    bli = 3
  }
}
```

Capítulo 12

Paradigma de *scripting*

capítulo 13. Michael Scott. Programming Language Pragmatics (2nd ed.). Morgan Kaufmann. 2016.

Los lenguajes de programación tradicionales están diseñados principalmente para crear aplicaciones autónomas: programas que aceptan algún tipo de entrada, la manipulan de alguna manera que se entiende bien, y generan la salida esperada. Pero la mayoría de los usos reales de las computadoras requieren la coordinación de múltiples programas. Un sistema de nóminas institucional, por ejemplo, debe procesar los datos de informes de los lectores de tarjetas, formularios en papel escaneados, y la entrada manual (teclado); ejecutar miles de consultas de bases de datos; cumplir cientos de normas jurídicas e institucionales; crear un extenso rastro de papel"para el mantenimiento de registros, auditoría, y los propósitos de preparación de impuestos; cheques de impresión; y comunicarse con los servidores de todo el mundo para el depósito directo *on-line*, la retención de impuestos, la acumulación de jubilación, seguro médico, y así sucesivamente. Estas tareas suelen involucrar docenas o cientos de programas ejecutables por separado. La coordinación entre estos programas requiere pruebas y condicionales, bucles, variables y tipos, subrutinas y abstracciones: el mismo tipo de herramientas lógicas que un lenguaje convencional proporciona dentro de una aplicación.

En principio, cualquier lenguaje puede ser utilizado como un lenguaje de scripting, siempre que cuente con las librerías o bindings para un entorno específico. Sin embargo, los lenguajes específicamente de scripting están pensados para ser muy rápidos de aprender y escribir, ya sea como archivos ejecutables o de forma interactiva en un bucle de lectura–evaluación–impresión (REPL). En general, esto implica una sintaxis y semántica relativamente simples; típicamente un "script"se ejecuta de principio a fin, como un "guión", sin un punto de entrada explícito.

Por ejemplo, no es común utilizar Java como lenguaje de scripting debido a su sintaxis verbosa y reglas sobre las clases que se encuentran en diferentes archivos, y directamente es imposible ejecutar Java de forma interactiva, ya que los archivos fuente sólo pueden tener las definiciones que deben ser invocados externamente por una aplicación lanzador de aplicaciones. Por el contrario, en Python se pueden definir algunas funciones en un solo archivo, evitar por completo las funciones y escribir en un estilo de programación

imperativo, o incluso utilizarlo de forma interactiva.

Un lenguaje de scripting suele ser interpretado a partir del código fuente o el bytecode.

Los lenguajes de scripting pueden estar diseñados para ser usados por usuarios finales de una aplicación o sólo para uso interno de los desarrolladores. Los lenguajes de scripting suelen utilizar abstracción para hacer transparente a los usuarios los detalles de los tipos internos variables, almacenamiento de datos y la gestión de memoria.

Los scripts son a menudo creados o modificados por la persona que los ejecuta, pero a menudo también se distribuyen, por ejemplo, buena parte de los juegos están escritos en un lenguaje de scripting.

12.1. Tipos de lenguajes de scripting

Podemos distinguir dos grandes familias de lenguajes de scripting: los de propósito general y los específicos de dominio o de extensión.

Los lenguajes de scripting de propósito general, como Perl y Python, se suelen llamar lenguajes pegamento (*glue languages*), porque fueron diseñados originalmente para "pegar"las salidas y entradas de otros programas, para construir sistemas más grandes. Con el crecimiento de la World Wide Web, los lenguajes de scripting han adquirido una nueva importancia en la generación de contenido dinámico.

Los lenguajes específicos de dominio están pensados para extender las capacidades de una aplicación o entorno, ya que permiten al usuario personalizar o extender las funcionalidades mediante la automatización de secuencias de comandos.

Lenguajes como TCL o Lua se han diseñado específicamente como lenguajes de scripting de propósito general, para ser incorporados en cualquier aplicación. Otros lenguajes como Visual Basic para Aplicaciones (VBA) está pensado para facilitar la integración en un sistema operativo específico.

Implementar las secuencias de comandos en estos lenguajes independientes de dominio en lugar de desarrollar un nuevo lenguaje para cada aplicación hace que los desarrolladores no tengan que aprender nuevos lenguajes o traducir de un lenguaje a otro.

Si comparamos el paradigma de scripting con el resto, podemos pensarlo como comparar secuencias de comandos con programación del sistema. Los scripts son código bisagra, su función es conectar componentes de software. Los lenguajes "pegamento" (*glue languages*) son los que se especializan para conectar componentes de software. Algunos ejemplos clásicos son los *pipes* y las secuencias de comandos del *shell*. También en el desarrollo web se requiere esa funcionalidad, por ejemplo, para conectar un servidor web con diferentes servicios web, como una base de datos.

Perl fue desarrollado para funcionar principalmente como lenguaje pegamento. Sin embargo, pronto empezaron a expresarse partes más grandes de la lógica de los programas en los scripts en perl, por lo que dejaron de cumplir su función de pegamento para pasar a cumplir función de componentes de software en sí, programas.

Los lenguajes pegamento son especialmente útiles para escribir y mantener:

- comandos personalizados para una consola de comandos;

- programas más pequeños que los que están mejor implementados en un lenguaje compilado;
- programas de çontenedor"para archivos ejecutables, como manipular archivos y hacer otras cosas con el sistema operativo antes o después de ejecutar una aplicación como un procesador de textos, hoja de cálculo, base de datos, ensamblador, compilador, etc .;
- secuencias de comandos que pueden cambiar;
- prototipos rápidos de una solución.

Los lenguajes de macros que manipulan componentes de un sistema o aplicación específicos pueden funcionar también como lenguajes de pegamento. Otras herramientas como AWK también pueden ser considerados lenguajes pegamento, al igual que cualquier lenguaje implementado por un motor de Windows Script Host (VBScript, JScript y VBA por defecto en Windows y motores de otros fabricantes, incluyendo implementaciones de Rexx, Perl, Tcl, Python, XSLT, Ruby, Delphi, etc.). La mayoría de las aplicaciones pueden acceder y utilizar los componentes del sistema operativo a través de los modelos de objetos o de sus propias funciones.

Otros dispositivos, como las calculadoras programables, también pueden tener lenguajes pegamento. Por ejemplo, la Texas Instruments TI-92, se puede programar de fábrica con un lenguaje de secuencias de comandos. En la serie TI-Nspire de calculadoras se incluyeron, además de las secuencias de comandos, el lenguaje Lua. Los lenguajes de alto nivel de la mayoría de las calculadoras gráficas (frecuentemente variantes de Basic, a veces derivados de Lisp, y, menos frecuentemente, derivados C), en muchos casos pueden pegar funciones, tales como gráficos de la calculadora, listas, matrices, etc. Para algunas de las máquinas de TI y HP se usan conversores de C y Perl, REXX AWK, así como secuencias de comandos shell a Perl, VBScript hacia y desde Perl. De esta forma se puede escribir un programa en un lenguaje de pegamento para la eventual aplicación (como un programa compilado) en la calculadora.

Una clase importante de lenguajes de scripting ha crecido a partir de la automatización del control de tareas, principalmente la puesta en marcha y control del comportamiento de los programas del sistema. (En este sentido, se podría pensar que las shells son descendientes del JCL de IBM, o Job Control Language, que se utilizaba para este propósito.) Muchos de los intérpretes de lenguajes doblan como intérpretes de línea de comandos, como el shell de Unix o el MS-DOS COMMAND.COM.

Con el advenimiento de interfaces gráficas de usuario, surgió un tipo especializado de lenguaje de scripting. Estos lenguajes interactúan con los gráficos de ventanas, menús, botones, etc., al mismo nivel de abstracción que resulta natural para un usuario humano. Esto se hace facilitando que se puedan reproducir las acciones de un usuario, ya que estos lenguajes se usan normalmente para automatizar las acciones del usuario. A estos lenguajes también se les llama "macros"si el control es a través de la simulación de las acciones de presionar teclas o clics del ratón, así como tocar o pulsar en una pantalla táctil activada.

Estos lenguajes podrían, en principio, utilizarse para controlar cualquier aplicación de interfaz gráfica de usuario; pero, en la práctica, su uso es limitado porque están muy ligados a una aplicación específica o un sistema operativo particular. Hay algunas excepciones a esta limitación. Algunos lenguajes de scripting de GUI se basan en el reconocimiento de objetos gráficos a través de sus píxeles de la pantalla de visualización. Estos lenguajes de scripting de GUI no dependen del apoyo del sistema operativo o aplicación.

Muchos programas de aplicación grandes incluyen un pequeño lenguaje de programación adaptado a las necesidades del usuario de la aplicación. Del mismo modo, muchos juegos de computadora utilizan un lenguaje de programación específico para expresar las acciones de personajes no jugadores y del entorno del juego. Los lenguajes de este tipo están diseñados para una sola aplicación; y, si bien pueden parecerse superficialmente un lenguaje específico de propósito general (por ejemplo QuakeC se parece a C), tienen características específicas que los distinguen, principalmente abstracciones que acortan la expresión de los conceptos propios del juego. El Emacs Lisp, aunque un dialecto totalmente formada y capaz de Lisp, contiene muchas características especiales que hacen que sea más útil para ampliar las funciones de edición de Emacs. Un lenguaje de programación específico de la aplicación puede ser visto como un lenguaje de programación específico del dominio especializado para una sola aplicación.

Varios lenguajes han sido diseñados para reemplazar lenguajes de scripting específicos una aplicación, como estándares para tareas de scripting. De esta forma, los programadores de aplicaciones (que trabajan en C o en otro lenguaje de programación no de scripting) incluye "ganchosçon los que el lenguaje de scripting puede controlar la aplicación. Estos lenguajes pueden ser técnicamente equivalentes a un lenguaje de extensión específico de la aplicación, pero cuando una aplicación incorpora un lenguaje çomún", el usuario puede programar con el mismo lenguaje para diferentes aplicaciones. Una alternativa más genérica es simplemente proporcionar una librería que un lenguaje de propósito general puede utilizar para controlar la aplicación, sin modificar el lenguaje para el dominio específico.

JavaScript comenzó como un lenguaje de secuencias de comandos en el interior de los navegadores web, y ésa sigue siendo su función principal. Sin embargo, la estandarización del lenguaje como ECMAScript lo ha popularizado como lenguaje integrable de propósito general. En particular, la aplicación SpiderMonkey Mozilla está incrustada en varios ambientes.

TCL fue creado como un lenguaje de extensión, pero ahora se usa principalmente como un lenguaje de propósito general, parecido a Python, Perl o Ruby. Por otro lado, Rexx fue creado originalmente como un lenguaje de control de trabajos, pero se usa como un lenguaje de extensión, así como un lenguaje de propósito general. Perl es un lenguaje de propósito general, pero tenía en 1990 el dialecto Oraperl que consiste en un binario de Perl 4 con Oracle Call Interface compilado. Sin embargo, desde entonces se ha sustituido por una biblioteca (Perl Module), DBD :: Oracle.

Otras aplicaciones complejas y orientadas a tareas pueden incorporar y un lenguaje de programación para permitir a sus usuarios un mayor control y más funcionalidades. Por ejemplo, las herramientas de autoría Autodesk Maya 3D incrustan el lenguaje de script MEL, o Blender, que utiliza Python para desempeñar este papel.

12.2. Ejercicios

12.1. Si quiero un control fuerte de la seguridad de una aplicación, debo renunciar a usar un lenguaje de scripting como por ejemplo JavaScript? Dé por lo menos un argumento que muestre las vulnerabilidades de un lenguaje de scripting y por lo menos dos argumentos o estrategias con las que podría tratar de reducir los riesgos de un lenguaje de scripting. Argumente qué ventajas aportan las características del lenguaje que implican vulnerabilidades.

12.2. En el siguiente código en Quake-C, identifique por lo menos 3 características que lo identifican como un lenguaje de scripting, y argumente por qué, si es posible, comparándolo con C, lenguaje en el que está basado Quake-C.

```
// Example of an alternate fiend jumping AI

// Here's a jumping algorithm I came up with for the fiends in Quake.
// It's a pretty good monster cheat that has them jumping for you and
// making it half the time no matter where you are.  Even from underwater.

// You can also change
//   setsize (self, VEC_HULL2_MIN, VEC_HULL2_MAX);
// to
//   setsize (self, VEC_HULL_MIN, VEC_HULL_MAX);
// in void() monster_demon1

// Quake only has three clipping hulls for moving entities, point-size,
// man-size, and shambler-size.  Making the hull man-sized gives the
// fiend a BIG edge in tracking you down, (it's a definite hack :)

DEMON.QC
(Change demon1_run6 and demon1_jump4)

void()   demon1_jump1;
void()   demon1_run6 =[ $run6, demon1_run1 ]
{
   ai_run(36);
   // if underwater, give a chance to switch to jump AI anyway
   if (self.waterlevel == 1)
   {
      if ( random() > 0.8 )
         self.think = demon1_jump1;
   }
```

```
};

void() demon1_jump4 =[ $leap4,demon1_jump5 ]
{
   local vector dir;
   local float dist;

   ai_face();

self.touch = Demon_JumpTouch;
makevectors (self.angles);
self.origin_z = self.origin_z + 1;
   if (self.enemy.origin_z > self.origin_z + 40)
   {
      if ( infront(self.enemy) || self.waterlevel == 1 )
      {
         dir = normalize(self.enemy.origin - self.origin);
         dist = vlen(self.enemy.origin - self.origin);
         self.velocity = dir * dist;
      }
      else
         self.velocity = v_forward * 150;

      dist = self.enemy.origin_z - self.origin_z;

      // a vertical velocity to jump just above player's head
      // for any given distance between z origins.
      // (hack numbers that work for Quake)
      dist = dist - 237;
      self.velocity_z = 714 + dist;
   }
   else
   {
      // slow forward to fall off ledge if player down
      // else normal horizontal velocity
      if (self.enemy.origin_z < self.origin_z - 40)
         self.velocity = v_forward * 150;
      else
         self.velocity = v_forward * 600;

      // check for wall in front
      traceline( self.origin - '0 0 16',
                 (self.origin - '0 0 16') + v_forward*150, TRUE, self);
```

```
      // attempt to jump wall or normal vel.
      if (trace_fraction < 1)
      {
         self.velocity = v_forward * 110;
         self.velocity_z = 250 * (random() + 1);
      }
      else
         self.velocity_z = 250;
   }

   if (self.flags & FL_ONGROUND)
self.flags = self.flags - FL_ONGROUND;
};

// (Add random statement to CheckDemonJump() to avoid hang-ups and
// enable going for it no matter what)

float() CheckDemonJump =
{
local vector dist;
local float d;

   if (random() < 0.4)
      return TRUE;
```

12.3. Ordene los siguientes fragmentos de código de más de scripting a menos de scripting, y explique cuáles son los principios que han contribuído a su ordenamiento.

```
class HELLO_WORLD

creation
        make
feature
        make is
        local
                io:BASIC_IO
        do
                !!io
                io.put_string("%N Hello World!!!!")
        end --make

end -- class HELLO_WORLD
```

```
program hello_prog

root
    HELLO_WORLD: "make"


cluster

    "./"

    end

    include  "library/lib.pdl"


end -- hello_prog
```

```
while 1,
 disp('hello world')
end
```

```
\font\HW=cmr10 scaled 3000
\leftline{\HW Hello World}
\bye
```

```
PROC write.string(CHAN output, VALUE string[])=
  SEQ character.number = [1 FOR string[BYTE 0]]
    output ! string[BYTE character.number]

write.string(terminal.screen, "Hello World!")
```

12.4. Ordene los siguientes fragmentos de código de más de scripting a menos de scripting, siempre justificando por qué.

```
#!/bin/bash
for jpg; do
    png="${jpg%.jpg}.png"
    echo converting "$jpg" ...
    if convert "$jpg" jpg.to.png ; then
        mv jpg.to.png "$png"
    else
        echo 'jpg2png: error: failed output saved in
   "jpg.to.png".' >&2
        exit 1
```

```
    fi
done
echo all conversions successful
exit 0
```

```
proc for {initCmd testExpr advanceCmd bodyScript} {
    uplevel 1 $initCmd
    set testCmd [list expr $testExpr]
    while {[uplevel 1 $testCmd]} {
        uplevel 1 $bodyScript
        uplevel 1 $advanceCmd
    }
}
```

```
#!/usr/bin/perl
use strict;
use warnings;
use IO::Handle;

my ( $remaining, $total );

$remaining = $total = shift(@ARGV);

STDOUT->autoflush(1);

while ( $remaining ) {
    printf ( "Remaining %s/%s \r", $remaining--, $total );
    sleep 1;
}

print "\n";
```

```
class classname : public superclassname {
 protected:
 // instance variables

 public:
 // Class (static) functions
 static void * classMethod1();
 static return_type classMethod2();
 static return_type classMethod3(param1_type
   param1_varName);
```

```
11  // Instance (member) functions
    return_type instanceMethod1With1Parameter (param1_type
      param1_varName);
13  return_type instanceMethod2With2Parameters (param1_type
      param1_varName, param2_type param2_varName=default);
};
```

Capítulo 13

Seguridad basada en lenguajes de programación

Mitchell 13.5

13.1. Ejercicios

13.1. En Java, hay varios mecanismos de seguridad a través del lenguaje, y muchos de ellos interactúan. Los *class loaders* son parte del *Java Runtime Environment (JRE)* y sirven para cargar dinámicamente, por demanda, clases en tiempo de ejecución. No se puede cargar ninguna clase sin un class loader. Sabiendo el rol del *Security Manager* en Java, explique cómo imagina que debe ser la relación entre el *Security Manager* y un *class loader* para preservar seguridad.

13.2. Describa la intención del *Proof-Carrying-Code* y cómo provee seguridad.

13.3. En lenguajes inseguros, se pueden implementar políticas de seguridad llamadas "ejecución defensiva". Describa algunas de estas políticas, por ejemplo, para pervenirse de vulnerabilidades por debilidad en el sistema de tipos de un lenguaje de scripting.

13.4. La *no interferencia* es uno de los principios básicos de seguridad. Explique cómo los niveles de visibilidad de un lenguaje orientado a objetos pueden contribuir a la no interferencia. Ahora, explique cómo la visibilidad ortogonal (por ejemplo, el `package` en Java) pueden contribuir a la implementación de políticas de acceso a información más complejas que las basadas únicamente en herencia.

13.5. Explique las vulnerabilidades asociadas a un sistema de tipos como el de un lenguaje de scripting, que provee robustez y facilita casteos.

13.6. Ante lenguajes de programación inseguros podemos tomar estrategias de programación defensiva o de programación ofensiva. De los dos códigos que encontramos abajo, explique cuál de los dos estaría aplicando una estrategia ofensiva, cuál una estrategia

defensiva. Explique qué tipo de vulnerabiliad encontramos en estos códigos, si vulnerabilidad en el sistema de tipos o vulnerabilidad de memoria, y en qué consiste esta vulnerabilidad de forma genérica.

```
const char* trafficlight_colorname(enum
   trafficlight_color c) {
    switch (c) {
        case TRAFFICLIGHT_RED:    return "red";
        case TRAFFICLIGHT_YELLOW: return "yellow";
        case TRAFFICLIGHT_GREEN:  return "green";
    }
    return "black";
}
```

```
const char* trafficlight_colorname(enum
   trafficlight_color c) {
    switch (c) {
        case TRAFFICLIGHT_RED:    return "red";
        case TRAFFICLIGHT_YELLOW: return "yellow";
        case TRAFFICLIGHT_GREEN:  return "green";
    }
}
```

13.7. El siguiente es un ejemplo de un operador que busca aumentar la seguridad de un lenguaje. Explique qué aporta este operador en términos de seguridad, y qué otros mecanismos se pueden implementar en lenguajes inseguros para aumentar su seguridad.

> Kotlin makes a distinction between nullable and non-nullable datatypes. All nullable objects must be declared with a ¿"postfix after the type name. Operations on nullable objects need special care from developers: null-check must be performed before using the value. Kotlin provides null-safe operators to help developers:

?. (safe navigation operator) can be used to safely access a method or property of a possibly null object. If the object is null, the method will not be called and the expression evaluates to null.

?: (null coalescing operator) often referred to as the Elvis operator:

```
fun sayHello(maybe : String?, neverNull : Int) {
    // use of elvis operator
    val name : String = maybe ?: "stranger"
    println("Hello $name")
}
```

Un ejemplo de uso:

```
// returns null if (and only if) foo is null, or bar() returns null,
// or baz() returns null
foo ?. bar() ?. baz()
```