Lenguajes de programación y modelos de computación

Asignatura: Análisis Comparativo de Lenguajes
Responsable: Ariel Gonzalez
e-mail: agonzalez@dc.exa.unrc.edu.ar

Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto - Argentina

2017

Abstract

Este libro es el resultado del dictado del curso *Análisis Comparativo de Lenguajes* para alumnos de pregrado en la Universidad Nacional de Río Cuarto.

Si bien existe una vasta bibliografía en el tema, es difícil encontrar un único libro que cubriese todos los temas y con el enfoque que es buscado en la asignatura.

Los principales objetivos de este trabajo es recopilar contenidos de varias fuentes bibliográficas y compilarlas desde un enfoque de las características de los lenguajes de programación a partir de un punto de vista de modelos de computación y paradigmas (o estilos) de programación, desarrollando los conceptos relevantes de los lenguajes de programación.

En cada capítulo se desarrollan los conceptos a partir de un lenguaje de programación básico, para luego compararlo con las construcciones similares encontradas en algunos lenguajes de programación seleccionados.

Los lenguajes de programación se han seleccionado por su difusión en la industria y por su importancia desde el punto de vista académico, los cuales se analizan en base a los conceptos básicos estudiados.

El enfoque es centrado en la elección de un lenguaje núcleo, para el cual se define su sintaxis y semántica (en base a su máquina abstracta correspondiente). El mismo, es extendido con adornos sintácticos y otras construcciones básicas en función de las características a analizar. La semántica formal permite realizar análisis de correctitud y su complejidad computacional.

Este material está dirigido a alumnos de segundo o tercer año de carreras de ciencias de la computación o ingeniería de software. Sus contenidos permiten desarrollar un curso en cuatro meses de duración con prácticas de aula y talleres. Al final de cada capítulo se proponen ejercicios correspondientes a cada tema.

Los paradigmas estudiados implican el modelo **imperativo**, **funcional**, **orientado a objetos**, **lógico** y el **concurrente**. Este último modelo es transversal a los demás modelos, por lo que se hace un análisis y consideraciones en cada contexto en particular.

El lenguaje *kernel* seleccionado es **Oz**, el cual es un lenguaje académico desarrollado específicamente para el estudio de los diferentes modelos de computación.

Contents

Capítulo 1

Introducción

Los lenguajes de programación son la herramienta de programación fundamental de los desarrolladores de software. Desde los comienzos de la computación, la programación fue evolucionando desde la simple configuración de interruptores, pasando por los primeros lenguajes **assembly**, los cuales permitan escribir las instrucciones de máquina en forma simbólica y la definición de *macros*, hasta llegar a los lenguajes de programación de alto nivel que permiten abstraer al programador de los detalles de la arquitectura y el desarrollo de programas *portables* entre diferentes sistemas de computación[1].

El objetivo de este material es estudiar los conceptos y principios que encontramos en los lenguajes de programación modernos.

Es importante conocer un poco la historia y la evolución de algunos conceptos para poder entender algunas características de algunos lenguajes.

En la actualidad se encuentran catalogados mas de 1500 lenguajes de programación, por lo cual una currícula en ciencias de la computación o de desarrollo de software no puede enfocarse en base al dictado de cursos sobre lenguajes concretos, sino que es necesario que se estudien lenguajes de programación desde el punto de vista de los diferentes modelos o estilos de computación en los cuales se basan.

Estos modelos o estilos permiten clasificar a los lenguajes de programación en familias que generalmente se conocen como *paradigmas*.

El estudio de los lenguajes en base al análisis de cada paradigma permite generalizar conceptos utilizados en grupos de lenguajes mas que en lenguajes particulares.

El enfoque utilizado permite realizar análisis de los conceptos utilizados en todos los lenguajes de programación existentes, permitiendo realizar comparaciones entre lenguajes o familias.

El estudio de los conceptos y principios generales, en lugar de estudiar la sintaxis de lenguajes específicos, permite que el desarrollador pueda estudiar y aprender por sí

[1] Un sistema de computación comprende el *hardware* y el software de base, es decir, sistema operativo, enlazador, compiladores, editores, etc.

mismo, a utilizar correctamente las facilidades provistas por un nuevo lenguaje (o uno desconocido).

Los paradigmas estudiados comprenden el *declarativo*, dentro del cual podemos encontrar el *funcional* y el *lógico*, el *imperativo*, en el cual podemos encontrar una gran cantidad de lenguajes ampliamente utilizados como Pascal, C, Basic, Ada, FORTRAN, COBOL, etc., con sus evoluciones en la *programación orientada a objetos (POO)* y los lenguajes basados en componentes.

Los conceptos y principios de la *concurrencia* son aplicables a todos los demás paradigmas por lo que se estudia como un paradigma en particular analizándose su aplicación en cada modelo de computación en particular.

1.1 Lenguajes como herramientas de programación

Un lenguaje de programación permite al programador definir y usar *abstracciones*. El desarrollo de software se basa fundamentalmente en la utilización de los lenguajes de programación y los procesadores de lenguajes (compiladores, intérpretes y linkers).

Las demás herramientas son auxiliares (como los editores, entornos integrados de desarrollo, generadores de Código, etc.) y su objetivo es sólo hacer más cómoda, automatizable y rápida la tarea de producción de código.

Los métodos de desarrollo de software, los cuales incluyen lenguajes textuales o iconográficos, están basados en los mismos conceptos adoptados en los lenguajes de programación[2].

La afirmación anterior es fácilmente verificable ya que cualquier método de desarrollo deberá permitir la generación de código al menos para algún lenguaje de programación.

1.2 Abstracciones

En la sección anterior se afirma que un lenguaje de programación brinda mecanismos para la definición y utilización de abstracciones.

Estas abstracciones permiten que el programador tome distancia de las características de bajo nivel del hardware para resolver problemas de una manera mas *modular*, y contribuir así a un fácil *mantenimiento* a través de su vida útil.

Aceptando esta definición de lo que es un lenguaje de programación, es mas comprensible que los diseñadores de software a gran escala, generalmente son personas con amplios conocimientos sobre lenguajes (y su implementación), y muestra que es imposible que un (buen) diseñador de software no haya pasado por una etapa de verdadero desarrollo de software, es decir, la escritura de programas concretos en algún lenguaje de programación que incorpore conceptos modernos como abstracciones de

[2] En realidad las características que encontramos en los métodos de desarrollo se pueden encontrar en lenguajes de programación desarrollados con bastante anterioridad.

alto nivel.
Esto nos permite definir el término programación.

Definición 1.2.1 *La programación es la actividad que consiste en definir y usar abstracciones para resolver problemas algorítmicamente.*

Es importante comprender así a la programación, ya que esto muestra el porqué los mejores programadores o diseñadores son aquellos que tienen una buena base en contenidos, en los cuales el concepto de abstracción es indispensable en algunas áreas como la matemática, la lógica y el álgebra.

Un lenguaje de programación generalmente sugiere uno o más *estilos* de programación, por lo que su estudio permite su mejor aprovechamiento en el proceso de desarrollo de sotfware.

1.2.1 Abstracción procedural

Una abstracción procedural permite encapsular en una unidad sintáctica una computación parametrizada.

Es bien conocida la estrategia de solución de problemas conocido como *divide and conquer (divide y vencerás)*, la cual se basa en la descomposición del problema en un conjunto de subproblemas mas simples y una forma de composición de esos subproblemas para obtener la solución final.

La abstracción procedural es la base de la implementación de esta estrategia de resolución de problemas. A modo de ejemplo, la programación funcional se caracteriza por la definición de *funciones* y la composición funcional. En cambio la programación imperativa se caracteriza por definir la evolución de los estados de un sistema basándose en la composición secuencial y en operaciones de cambios de estado (asignación).

1.2.2 Abstracción de datos

Generalmente los programas operan sobre ciertos conjuntos de datos. Es bien conocido que los cambios mas frecuentes producidos en un sistema son los de representación de los datos que se manipulan. Por este motivo es importante poder *ocultar* los detalles de la representación (o implementación) de los datos para facilitar el mantenimiento y la utilización de subprogramas.

Los *tipos abstractos de datos (ADTs)* permiten definir tipos de datos cuyos valores están implícitos o denotados por sus operaciones. Es deseable que los lenguajes de programación permitan la especificación o implementación de ADTs ocultando los detalles de representación.

Es sabido que no todos los lenguajes lo permiten, pero las tendencias actuales han avanzado respecto a las capacidades de modularización y ocultamiento de información, otorgando un mayor control en el encapsulamiento de los componentes de las abstracciones.

1.3 Evaluación de un lenguaje de programación

Un lenguaje de programación puede ser evaluado desde diferentes puntos de vista. En particular, un lenguaje debería tener las siguientes propiedades:

- **Universal**: cada problema *computable* debería ser expresable en el lenguaje.

  Esto deja claro que en el contexto de este libro, a modo de ejemplo, un lenguaje como SQL[3] no es considerado un lenguaje de programación.

- **Natural**: con su dominio de su aplicación.

  Por ejemplo, un lenguaje orientado al procesamiento vectorial debería ser rico en tipos de datos de vectores, matrices y sus operaciones relacionadas.

- **Implementable**: debería ser posible escribir un intérprete o un compilador en algún sistema de computación.

- **Eficiente**: cada característica del lenguaje debería poder implementarse utilizando la menor cantidad de recursos posibles, tanto en espacio (memoria) y número de computaciones (tiempo).

- **Simple**: en cuanto a la cantidad de conceptos en los cuales se basa. A modo de ejemplo, lenguajes como PLI y ADA han recibido muchas críticas por su falta de simplicidad.

- **Uniforme**: los conceptos básicos deberían aplicarse en forma consistente en el lenguaje. Como un contraejemplo, en C el símbolo * se utiliza tanto para las declaraciones de punteros como para los operadores de *referenciación* y multiplicación, lo que a menudo confunde y da lugar a la escritura de programas difíciles de entender.

- **Legible**: Los programas deberían ser fáciles de entender. Una crítica a los lenguajes derivados de C es que son fácilmente confundible los operadores == y =.

- **Seguro**: Los errores deberían ser detectables, preferentemente en forma estática (en tiempo de compilación).

Los lenguajes de programación son las herramientas básicas que el programador tiene en su *caja de herramientas*. El conocimiento de esas herramientas y cómo y en qué contexto debe usarse cada uno de ellos hace la diferencia entre un programador recién iniciado y un experimentado especialista.

Es fundamental que los conceptos sobre lenguajes de programación estén claros para poder aplicar (y entender) las otras áreas del desarrollo de software como lo son las estructuras de datos, el diseño de algoritmos y estructuración (diseño) de programas complejos. En definitiva estas tareas se basan siempre en un mismo concepto: *abstracciones*.

[3] En SQL no se pueden expresar *clausuras*.

1.4 Definición de un lenguaje de programación

Para describir un lenguaje de programación es necesario definir la forma de sus *frases* válidas del lenguaje y de la semántica o significado de cada una de ellas.

1.4.1 Sintaxis

Los mecanismos de definición de sintaxis han sido ampliamente estudiados desde los inicios de la computación. El desarrollo de la teoría de lenguajes y su clasificación[6] ha permitido que se definan formalismos de descripción de lenguajes formales e inclusive, el desarrollo de herramientas automáticas que permiten generar automáticamente programas reconocedores de lenguajes (parsers y lexers) a partir de su especificación[4].

La sintaxis de un lenguaje se especifica por medio de algún formalismo basado en *gramáticas libres de contexto*, las cuales permiten especificar la construcción (o derivación) de las frases de un lenguaje en forma modular.

Las gramáticas libres de contexto contienen un conjunto de *reglas de formación* de las diferentes frases o *categorías sintácticas de un lenguaje.*

Definición 1.4.1 *Una gramática libre de contexto (CFG) es una tupla* (V_N, V_T, S, P)*, donde* V_N *es el conjunto finito de símbolos no terminales,* V_T *es el conjunto finito de símbolos terminales,* $S \in V_N$ *es el símbolo de comienzo y* P *es un conjunto finito de producciones.*

Los conjuntos V_N *y* V_T *deben ser disjuntos (*$(V_N \cap V_T) = \emptyset$*) y denotaremos* $\Sigma = V_N \cup V_T$*.*

P *es un* conjunto de producciones*, donde una producción* $p \in P$ *tiene la forma* (L, R)*, donde* $L \in V_N$ *es la parte izquierda (lhs) de la producción y* $R \in (V_N \cup V_T)^*$ *es la parte derecha (rhs).*

Por claridad, en lugar de describir las producciones como pares, se denotará a una producción rotulada p: $(X_0, (X_1, \ldots, X_{n_p}))$, con $n_p \geq 0$ como:

$$p : X_0 \to X_1 \ldots X_{np} \tag{1.1}$$

y en el caso que $n_p = 0$, se escribirá como:

$$p : X_0 \to \lambda \tag{1.2}$$

De aquí en adelante se asumirá que el símbolo de comienzo S aparece en la parte izquierda de una única producción y no puede aparecer en la parte derecha de ninguna producción[5].

Es común que un conjunto de producciones de la forma $\{X \to \alpha, \ldots, X \to \beta\}$ se abrevie de la forma $X \to \alpha \mid \ldots \mid \beta$.

[4] Como las populares herramientas *lex* y *yacc*.
[5] Esta forma se denomina *gramática extendida*.

Definición 1.4.2 *Sean* $\alpha, \beta \in (V_N \cup V_T)^*$ *y sea* $q : X \to \varphi$ *una producción de* P, *entonces* $\alpha X \beta \underset{G}{\overset{q}{\Longrightarrow}} \alpha\varphi\beta$

La relación $\underset{G}{\Longrightarrow}$ se denomina *relación de derivación* y se dice que la cadena $\alpha X \beta$ deriva directamente (por aplicación de la producción q) a $\alpha\varphi\beta$.

Cuando se desee hacer explícita la producción usada en un paso de derivación se denotará como $\underset{G}{\overset{q}{\Longrightarrow}}$.

Se escribirá $\underset{G}{\overset{*}{\Longrightarrow}}$ a la clausura reflexo-transitiva de la relación de derivación.

Definición 1.4.3 *Sea* $G = (V_N, V_T, S, P)$ *una gramática libre de contexto. Una cadena* α, *obtenida por* $S \underset{G}{\overset{*}{\Longrightarrow}} \alpha$ *que contiene sólo símbolos terminales* ($\alpha \in V_T^*$), *se denomina una sentencia de* G. *Si la cadena* $\alpha \in (V_T \cup V_N)^*$ *(contiene no terminales) se denomina forma sentencial.*

Definición 1.4.4 *El lenguaje generado por* G, *denotado como*

$$L(G) = \{w | w \in V_T^* \mid S \underset{G}{\overset{*}{\Longrightarrow}} w\}$$

Definición 1.4.5 *Sea el grafo dirigido* $ST = (K, D)$ *un árbol, donde* K *es un conjunto de* nodos *y* D *es una relación no simétrica, con* k_0 *como raíz, una función de rotulación* $l : K \to V_T \cup \epsilon$ *y sean* $k_1, \ldots, K_n$, $(n > 0)$, *los sucesores inmediatos de* k_0.

El árbol $ST = (K, D)$ *es un árbol de derivación (o parse tree) correspondiente a* $G =< V_N, V_T, P, S >$ *si cumple con las siguientes propiedades:*

1. $K \subseteq (V_N \cup V_T \cup \epsilon)$
2. $l(K_0) = S$
3. $S \to l(k_1) \ldots l(k_n)$
4. *Si* $l(k_i) \in V_T$, $(1 \leq i \leq n)$, *o si* $n = 1$ *y* $l(k_1) = \epsilon$, *entonces* K_i *es una hoja de* ST.
5. *Si* $l(k_i) \in V_N$, $(1 \leq i \leq n)$, *entonces* k_i *es la raíz del árbol sintáctico para la gramática libre de contexto* $< V_N, V_T, P, l(k_i) >$.

Definición 1.4.6 *Sea ST(G) un árbol de derivación para* $G =< V_N, V_T, S, P >$. *La frontera de ST(G) es la cadena* $l(k_1) \ldots l(k_n)$ *tal que* $k_1 \ldots k_n$ *es la secuencia formada por las hojas de ST(G) visitadas en un recorrido preorden.*

Teorema 1.4.1 *Sea* $G =< V_N, V_T, S, P >$ *una gramática libre de contexto,* $S \underset{G}{\overset{*}{\Longrightarrow}} \alpha$ *si y sólo si existe un árbol de derivación para* G *cuya frontera es* α.

La figura 1.1 muestra una gramática libre de contexto y un árbol de derivación para la cadena *"a + b * c"*.

La gramática dada en la figura 1.1 es *ambigüa* ya que para una misma cadena existen dos (o más) árboles de derivación diferentes. Una gramática puede desambig'uarse introduciendo producciones que definan la *precedencia* entre los diferentes no terminales.

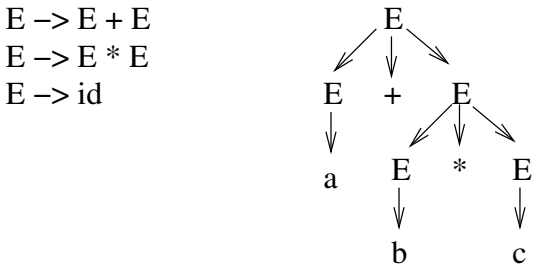


Fig. 1.1: Una CFG y un árbol de derivación.

Definición 1.4.7 *Dos gramáticas g_1 y g_2 son* equivalentes *si generan el mismo lenguaje, es decir que $L(g_1) = L(g_2)$*[6]*.*

Hay gramáticas *inherentemente ambigüas* para las cuales no existe una gramática equivalente no ambigüa.

1.4.1.1 Lenguajes regulares

Las *palabras* que se pueden formar en un lenguaje generalmente se describen con formalismos que no requieren describir estructuras de las frases. Estos formalismos se conocen como las *gramáticas regulares*. Existen otros formalismos equivalentes ampliamente utilizadas, como las *expresiones regulares*.

Definición 1.4.8 *Una* ***gramática regular*** *es una gramática cuyas producciones tienen la forma: $X \to Ya$ y $X \to a$, donde $X, Y \in N$ y $a \in T$.*

Estas gramáticas sólo permiten describir la conformación de las *palabras o tokens* de un lenguaje lenguaje, pero no es posible describir la estructura de frases. A modo de ejemplo se muestra una gramática regular que describe la formación de un valor entero positivo:

```
N --> N '0' | N '1' | ... | N '9' | '0' | '1' | ... | '9'
```

El ejemplo anterior muestra que es extenso definir la forma de construcción de símbolos de un lenguaje por medio de una gramática regular, por lo que es común que se definan por medio de un formalismo, las *expresiones regulares*, cuya expresividad es equivalente y permiten definiciones mas compactas y legibles.

A continuación se da una gramática libre de contexto que describe la sintaxis de una expresión regular:

```
E --> t
| E E            -- secuencia
```

[6] La determinación si dos gramáticas libres de contexto son equivalentes es indecidible, es decir, no existe un algoritmo que lo determine.

```
| (E '|' E)      -- alternativa (choice)
| (E)?           -- opcional (cero o una vez)
| (E)*           -- cero o m\'as veces
```

donde t es un símbolo terminal.

Las *gramáticas regulares extendidas* introducen otras construcciones más cómodas en la práctica como las siguientes:

```
E --> [ E ... E ]     -- set: equivalente a (E | ... | E)
    | (E)+            -- una o m\'as veces: equivalente a (E(E)*)
```

1.4.1.2 EBNFs y diagramas de sintaxis

Una *Extended Backus Naur Form* es una extensión de las gramáticas libres de contexto que permite la descripción de un lenguaje en forma mas compacta.

Informalmente, se puede decir que permiten escribir expresiones regulares extendidas en la parte derecha de las producciones. Las notaciones mas comunmente mas utilizadas son:

- (S): S ocurre una o mas veces.
- $\{S\}$: S ocurre cero o mas veces.
- $[S]$: S es opcional (cero o una vez).

A continuación de muestra un ejemplo de una EBNF.

```
...
var-decl --> var id {',' id} ':' type ';'
type     --> integer | real | ...
...
if-stmt  --> if condition then stmt [ else stmt ]
...
```

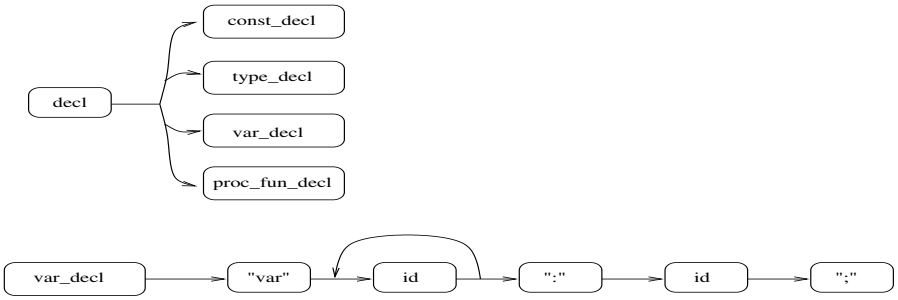


Fig. 1.2: Ejemplo de diagramas de sintaxis.

Los diagramas de sintaxis son una representación gráfica por medio de un grafo dirigido el cual muestra el flujo de aparición de los componentes sintácticos. Los nodos del grafo corresponden a los símbolos terminales y no terminales y los arcos indican el símbolo que puede seguir en una frase. Es común que los nodos correspondientes a los símbolos terminales se denoten con círculos y los nodos que corresponden a no terminales se denoten como óvalos.

La figura 1.4.1.2 muestra un ejemplo de diagramas de sintaxis.

1.4.2 Semántica

La semántica de un lenguaje de programación describe el significado, comportamiento o efectos de las diferentes frases del lenguaje.

Es muy común que en los manuales de los lenguajes de programación la semántica de cada una de las frases de describa de manera informal.

Esta informalidad ha llevado muchas veces a confusiones en los programadores o los implementadores de herramientas como compiladores e intérpretes, causando que los resultados de un programa en una implementación no sean los mismos que en otra[7].

Para dar una definición precisa de la semántica de un lenguaje es necesario utilizar algún formalismo que describa en forma clara y no ambigüa el significado de las frases.

Se han utilizado diferentes estilos de formalismos para dar semántica:

- **Denotacional**: cada construcción del lenguaje se relaciona con alguna entidad matemáticas (ej: conjuntos, funciones, etc) que representa el significado de cada estructura.

  Esta forma de dar semántica es útil desde el punto de vista teórico, pero en general no es cómodo para los implementadores de lenguajes y los desarrolladores.

- **Operacional**: descripción del *efecto o ejecución* de cada construcción del lenguaje en una *máquina abstracta* dada. Una máquina abstracta está basada en algún modelo de computación.

  Esta forma es útil tanto para los implementadores del lenguaje como para los desarrolladores de programas, ya que tienen una visión mas concreta (operacional) del lenguaje.

- **Axiomática**: descripción de cada construcción del lenguaje en términos de cambios de estado. Un ejemplo es la lógica de Hoare, que es muy útil para el desarrollo y verificación formal de programas imperativos.

  Esta técnica es útil para los desarrolladores pero no demasiado buena para los implementadores del lenguaje.

En este libro se utilizará la semántica operacional para dar el significado al lenguaje que se irá desarrollando en cada capítulo, siguiendo la idea de *lenguaje núcleo (kernel)* el cual permite dar una sintaxis y semántica de manera sencilla para luego *adornar* el lenguaje con mejoras sintácticas (syntactic sugars) y abstracciones sintácticas o lingüísticas prácticas, las cuales tendrán un patrón de traducción al lenguaje núcleo.

[7] Esto ha sucedido en C, C++, FORTRAN, y hasta en los lenguajes de reciente aparición.

1.5 Herramientas para la construcción de programas

El programador cuando utiliza un lenguaje de programación, utiliza herramientas que implementan el lenguaje. Estas herramientas son programas que permiten ejecutar en la plataforma de hardware utilizada las construcciones del lenguaje de alto nivel. En general se disponen de las siguientes herramientas:

- **Compilador**: traduce un programa fuente a un programa *assembly* u *objeto* (archivo binario enlazable).
- **Intérprete**: programa que toma como entrada programas fuentes, genera una representación interna adecuada para su ejecución y evalúa esa representación emulando la semántica de las construcciones del programa dado.

  Es posible encontrar intérpretes de bajo nivel, también conocidos como *ejecutores* de programas. Estos ejecutores interpretan lenguajes de bajo nivel (assembly real o hipotético).

  Es común que una implementación de un lenguaje venga acompañado por un compilador a un assembly de una máquina abstracta y un intérprete de ese lenguaje de alto nivel. Ejemplos de esto son algunos compiladores de COBOL, Pascal (se traducía a P-code).

  Actualmente uno de los casos mas conocidos sea Java. Es común que un compilador de Java traduzca los módulos a un assembly sobre una máquina abstracta conocida como la *Java Virtual Machine (JVM)*.

  Este último enfoque permite obtener *portabilidad* binaria, ya que es posible ejecutar un programa en cualquier plataforma que tenga una implementación (intérprete) de la máquina abstracta.

- **Enlazador (linker)**: un archivo objeto puede hacer referencia a símbolos (variables, rutinas, etc) de otros archivos objetos. Estas referencias se denominan *referencias externas*. El linker toma un conjunto de archivos objetos[8], arma una imagen en memoria, resuelve las referencias externas de cada uno (asigna direcciones de memoria concretas a cada referencia externa no resuelta) y genera un archivo binario ejecutable (*programa*).

  En forma más rigurosa, un linker básicamente implementa una función que toma una referencia a un símbolo externo y retorna la dirección de memoria de su definición.

  Generalmente cada archivo objeto se corresponde con un *módulo* del programa fuente. La modularización es útil para dividir grandes programas en unidades lógicas reusables.

[8] Generalmente llamados módulos binarios.

Además, los ambientes de desarrollo generalmente vienen acompañados por módulos básicos para hacerlo mas útil en la práctica (módulos para hacer entrada-salida, funciones matemáticas, implementación de estructuras de datos, etc) lo que comúnmente se conoce como la *biblioteca estándar* del lenguaje.

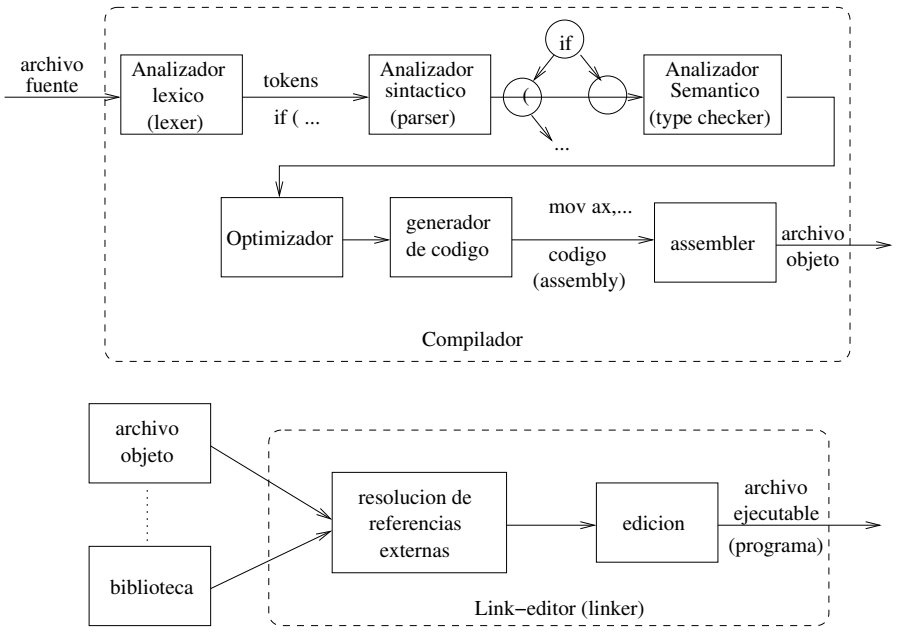


Fig. 1.3: Esquema de compilación de un programa.

La figura 1.3 muestra un esquema del proceso de compilación de un programa.

1.5.1 Bibliotecas estáticas y dinámicas

Una *biblioteca* es un archivo que contiene archivos objeto.

Generalmente un programa de usuario se enlaza con al menos unas cuantas rutinas básicas que comprenden el sistema de tiempo de ejecución (*runtime system*). El runtime system generalmente incluye rutinas de inicio (start-up) de programas[9], y la implementación de otras rutinas básicas del lenguaje.

Cuando en el programa obtenido se incluye el código (y posiblemente datos) de las rutinas de biblioteca utilizadas se denomina enlazado estático (static linking).

Un programa enlazado estáticamente tiene la ventaja que cuando se lo transporta a otra computadora tiene todas sus dependencias resueltas, es decir que todas sus

[9] Una rutina de startup generalmente abre archivos de entrada-salida estándar e invoca a la rutina principal del programa.

referencias (a datos y código) están resueltas y todo está contenido en un único archivo binario.

Los primeros sistemas de computación generalmente soportaban este único tipo de enlazado. De aquí el nombre a estos linkers conocidos como *link-editores*.

A medida que el tamaño de los programas crece, el uso de bibliotecas generales es común. Más aún, en los sistemas multitarea (o multiprogramación), comienzan a aparecer varias desventajas y el mecanismo de enlazado estático se torna prácticamente inviable.

Las principales desventajas son:

- El tamaño de los programas se hace muy grande.
- En un sistema multitarea hay grandes cantidades del mismo código replicado en la memoria y en el sistema de archivos.
- No tiene en cuenta la evolución de las bibliotecas, cuyas nuevas versiones pueden corregir errores o mejorar su implementación.

Por este motivo aparece el enfoque de las *bibliotecas de enlace dinámico*[10] (DLLs).

Este enfoque requiere que el sistema operativo contenga un linker dinámico, es decir que resuelva las referencias externas de un módulo (archivo objeto) en tiempo de ejecución.

Cuando un proceso (instancia de programa en ejecución) hace referencia a una entidad cuya dirección de memoria no haya sido resuelta (referencia externa), ocurre una trampa (trap) o excepción generada por el sistema operativo. Esta trampa dispara una rutina que es la encargada de realizar el enlace dinámico.

Posiblemente se requiera que el código (o al menos la parte requerida) de la biblioteca sea cargada en la memoria (si es que no lo estaba).

Cabe hacer notar que los archivos objetos deben acarrear mas información de utilidad por el linker dinámico. Un programa debe acarrear la lista de bibliotecas requeridas y cada archivo objeto de cada bibliotecas debe contener al menos el conjunto de símbolos que exporta.

Las principales ventajas que tiene este mecanismo son:

- El código de las rutinas de las bibliotecas se encuentra presente una sola vez (no hay múltiples copias).
- El código se carga baja demanda. Es decir que no se cargará el código de una biblioteca que no haya sido utilizada en una instancia de ejecución.

Como desventaja tiene que la ejecución de los programas tiene una sobrecarga adicional (overhead) que es el tiempo insumido por la resolución de referencias externas

[10] En el mundo UNIX son conocidas como *shared libraries*.

y la carga dinámica de código.

Un linker con capacidades de generar bibliotecas dinámicas deberá generar archivos objetos con la información adicional que mencionamos arriba y el sistema operativo deberá permitir ejecutar código reubicable, es decir independiente de su ubicación en la memoria[11].

Una biblioteca compartida no debería tener estado propio, ya que puede ser utilizada por múltiples procesos en forma simultánea, es decir que es un recurso compartido por varios procesos. Por ejemplo, un programador de una biblioteca que pueda utilizarse en forma compartida no podrá utilizar variables globales.

Lo anterior es muy importante a la hora de diseñar bibliotecas. Es bien conocido el caso de la biblioteca estándar de C, la cual define una variable global (`errno`), la cual contiene el código de error de la última llamada al sistema realizada.
Al querer hacer la biblioteca de C compartida, los desarrolladores tuvieron que implementar un atajo para solucionar este problema.

1.6 Ejercicios

Nota: los ejercicios están planteados para ser desarrollados en un sistema que disponga de las herramientas de desarrollo comúnmente encontrados en sistemas tipo UNIX. El práctico se puede desarrollar en cualquier plataforma que tenga instaladas las herramientas básicas de desarrollo del proyecto GNU (software libre) instaladas.

Herramientas necesarias: gcc (GNU Compiler Collection), gpc (GNU Pascal Compiler), ld, grep y wc.

1. Definir una expresión regular que denote un identificador en Pascal.
2. Definir un autómata finito que acepte el lenguaje denotado por la expresión regular del ejercicio anterior.
3. Definir un autómata finito que acepte cadenas de numeros binarios con cantidad par de 0's y cantidad par de 1's.
4. Definir un autómata finito que acepte cadenas de numeros binarios con cantidad par de 0's y cantidad impar de 1'.
5. Usar el comando **grep**[12] que seleccione las líneas del archivo fuente Pascal del ej. 7 los siguientes patrones:
   (a) Las líneas que contengan *Var*
   (b) Las líneas con comentarios

[11] Esto se logra utilizando algún mecanismo de *memoria virtual* (segmentación o paginado)
[12] Uso: grep expresión-regular [file]. Para mas información hacer "man grep".

(c) Comparar la cantidad de begin y la cantidad de end en un programa Pascal. Ayuda: usar grep y wc.

6. Dar una EBNF que defina las declaraciones de constantes de Pascal.

7. Dado el siguiente programa Pascal y el siguiente fragmento de código C. El programa CallToC declara una variable *externa*, le asigna un valor e invoca a un procedimiento *externo*, el cual está implementado en C (en el módulo *inc.c*),

```
Program CallToC;

Var x:integer; external name 'y';
Procedure inc_x; external name 'inc_y';

begin { programa principal }
  x := 1;
  inc_x;
  writeln('x=',x)
end.
```

```
/* file inc.c */
int y;          /* global integer y */

void inc_y(void)
{
    y++;
}
```

   (a) compilar el programa Pascal (usando gpc). En caso de error describir su origen y quién lo genera (compilador o linker).

   (b) compilar el fragmento de programa C para obtener el archivo objeto correspondiente[13] analizando los pasos realizados. Usar el comando *objdump -t inc.o* para ver los símbolos definidos en el archivo objeto.

   (c) generar un archivo ejecutable en base a los dos módulos.

   (d) describir qué pasos se realizaron (compilación, assembly, linking) en el punto anterior.

8. Generar una biblioteca estática (llamada *libmylib.a*) que contenga el archivo objeto *inc.o* (del ejercicio anterior) con la utilidad *ar*.

   Usar el siguiente programa C (el cual invoca a *inc_y*()) para compilarlo enlazarlo con la biblioteca *mylib*.

```
int main(void)
{
  inc_y();
}
```

[13] Usar el comando *gcc -v -c inc.c*.

9. Recompilar el programa Pascal definido arriba usando la biblioteca creada en el ejercicio anterior.

10. El siguiente programa C muestra la carga de una biblioteca dinámica (math), la resolución de una referencia (externa) a la función *cos* (definida en math) y la invocación a *cos(2.0)*.

```
/* File: foo.c */
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int main()
{
    void *handle;
    double (*cosine)(double); /* Pointer to a function */

    /* Load the math library */
    handle = dlopen("libm.so", RTLD_LAZY);

    /* Get (link) the "cos" function: we get a function pointer */
    cosine = (double (*)(double)) dlsym(handle, "cos");
    printf("%f\n", cosine(2.0));
    dlclose(handle);
    exit(EXIT_SUCCESS);
}
```

Compilar el programa (con el comando *gcc -rdynamic -o foo foo.c -ldl*) y ejecutarlo.

Capítulo 2

Lenguajes y modelos de programación

Un lenguaje de programación provee tres elementos principales:

1. un *modelo de computación*, el cual define una sintaxis y la semántica (formal o informal) de sus frases o sentencias.
2. un *conjunto de técnicas de programación*, las cuales definen un *modelo* o estilo de programación.
3. algún *mecanismo para el análisis de programas* (razonamiento, cálculos de eficiencia, etc).

Estos tres puntos definen lo que se conoce como *paradigma* de programación.

Un lenguaje de programación contiene diferentes tipos de constructores con su sintaxis y su semántica propia. Las diferentes construcciones o frases de un lenguaje generalmente se denominan sentencias y pueden clasificarse según su intención o uso en un programa.

En este capítulo, se analizan las diferentes tipos de sentencias y los conceptos fundamentales que podemos encontrar en un lenguaje de programación, independientemente al paradigma o modelo que pertenezca.

2.1 Modelos o paradigmas de programación

Un modelo o paradigma de programación define un estilo de programación. Cada modelo puede hacer que el programador piense los problemas a resolver desde diferentes perspectivas. Los modelos de programación pueden dividirse, en principio, en dos grandes grupos:

- *Modelo declarativo*, en donde no existe la noción de estado (stateless). Es decir que la ejecución de un programa evoluciona generando nuevos valores. Estos

valores nunca cambian. Es decir que la noción de variable (valor mutable) no existe, sino que los identificadores se ligan (asocian) a valores y esa asociación se mantiene inmutable.

- *Modelo con estado*, donde el concepto fundamental es la noción de asignación de valores a variables, es decir celdas con contenido mutable en la memoria.

El modelo sin estado se denomina declarativo porque el estilo de programación permite enfocar en la descripción de las computaciones más que en el detalle de cómo se deben realizar. En el modelo con estado, generalmente se decribe una computación como la evolución temporal del estado del programa, es decir, que se expresa la forma progresiva en que se arriba a una solución.

Cada modelo tiene sus ventajas y desventajas.

En el modelo declarativo las ventajas pueden ser:

a) claridad y simplicidad de los programas, ya que no se expresan en términos de cambios de estado.

b) permite razonamiento modular (se puede analizar cada unidad en forma independiente) y usando técnicas simples como lógica ecuacional e inducción (aritmética y/o estructural).

Como desventaja podemos mencionar:

a) algunos problemas se modelan naturalmente con estado. Por ejemplo, operaciones de entrada-salida, programas cliente-servidor, etc.

b) generalmente se logran rendimientos menores con respecto a programas equivalentes con estado.

En el modelo con estado podemos mencionar las siguientes ventajas:

a) eficiencia.

b) el modelo de programación está íntimamente relacionado con las arquitecturas de las computadoras actuales (arquitecturas Von Newman).

El modelo con estados, sin embargo tiene sus desventajas:

a) pérdida de razonamiento al estilo ecuacional (ya no es posible reemplazar iguales por iguales).

b) pérdida de razonamiento modular, debido a que las diferentes unidades de programa pueden actuar sobre una misma porción del estado del programa.

Definición 2.1.1 *Una expresión es* ***transparente referencialmente*** *si puede ser reemplazada con el valor denotado por su evaluación en cada parte del programa que aparezca. En otro caso diremos que la expresión tiene* ***efectos colaterales****.*

Las funciones matemáticas son transparentes referencialmente, aunque en programación no necesariamente. En el programa de la figura el siguiente programa de ejemplo, suponiendo que *f(v)* retorna el sucesor de *v*:

x := 1;
$\{x = 1\}$
y := f(x) + x;
$\{x = 1 \wedge y = 2 + 1\}$

se muestra un razonamiento ecuacional, el cual es válido sólo si la función f no tiene efectos colaterales. Si f modificara de alguna manera (en un modelo con estado) la variable x del ejemplo, el razonamiento anterior deja de ser válido.

Por esta razón, en el modelo con estado se debe razonar usando alguna lógica que tenga en cuenta cualquier cambio de estado, como por ejemplo la lógica de Hoare.

Esto muestra la necesidad del uso de ciertas técnicas de programación que eviten el uso de abstracciones (ej: funciones) que tengan efectos colaterales, para permitir un razonamiento más modular. Una de las técnicas mas efectivas es la modularización de los programas, definiendo componentes lo más independientes posibles, es decir que no tengan estado compartido.

2.1.1 Lenguajes declarativos

Los lenguajes de programación declarativos pueden clasificarse en alguna de las siguientes categorías:

- Funcionales: un programa se basa en aplicación y composición funcional. Ejemplos: Haskell y subconjuntos de LISP, SCHEME, y los derivados de ML (SML, Ocaml, etc)[1].
- Relacionales o lógicos: un programa opera sobre relaciones. Muy útiles para problemas con restricciones, con soluciones múltiples, etc.
- Memoria con asignación única: se asignan valores a variables una única vez. Las variables están ligadas a algún valor o no. Las variables ligadas permancen en ese estado durante todo su tiempo de vida. Ejemplo: subconjunto de Oz.

2.1.2 Lenguajes con estado

Los lenguajes de programación con estado son los mas comunes de encontrar. Estos lenguajes se caracterizan por poseer operaciones de cambio de estado de variables. Estas operaciones se conocen comúnmente como sentencias de asignación.

Es posible clasificar los lenguajes con estado como:

- Procedurales (o imperativos): se caracterizan por permitir abstracciones funcionales (procedimientos y/o funciones) parametrizadas, sentencias de asignación y de control de flujo de la ejecución y definiciones de variables. Ejemplos: Pascal, C, Fortran, Cobol, Basic, etc.
- Orientados a objetos: permiten definir tipos en clases[2] y permiten organizarlas en forma jerárquica, fundamentalmente usando la relación supertipo-subtipo.

[1] Estos últimos lenguajes no son funcionales puros, ya que también permiten programar con estado.

[2] Una clase define un tipo y es además un módulo que encapsula la representación del conjunto de valores y sus operaciones.

2.2 Elementos de un lenguaje de programación

Un lenguaje de programación ofrece al programador un conjunto de construcciones o sentencias que pueden clasificarse en las siguiente categorías:

- *Valores*: Todo lenguaje permite expresar valores de diferentes tipos, como por ejemplo, valores numéricos, strings, caracteres, listas, etc. Los valores de tipos básicos se denominan *literales* (ej: 123.67E2) y los valores de tipos compuestos (o estructurados) se denominan *agregados* (ej: {'a','b','c'} en C, [0,2,4,8] en Haskell).

- *Declaraciones*: Una declaración introduce una nueva entidad, generalmente identificable. Estas entidades pueden ser:

  - *Constantes simbólicas*: identificadores a valores dados. Ej. en Pascal: Const Pi=3.141516;
  - *Tipos*: introducen nuevos tipos definidos por el programador. Ej. en Pascal: Type Point = Record int x, y end;
  - *Variables*: identificadores que referencian valores almacenados en memoria.
  - *Procedimientos y funciones*: abstracciones parametrizadas de comandos y expresiones, respectivamente.
  - *Externas*: hacen referencia a entidades definidas en otros módulos (o bibliotecas). Ej (en Pascal): **Uses** `<unit>`;

  Aquellas declaraciones de entidades representables en memoria se denominan *definiciones*. Por ejemplo, son definiciones, las declaraciones de variables y de procedimientos y funciones, mientras que una declaración de un tipo no demanda memoria alguna.

- *Comandos*: sentencias de control de flujo de ejecución:

  - Comandos básicos: Como por ejemplo asignación (en Pascal) o invocaciones a procedimientos.
  - Secuencia o bloques.
  - Saltos (o secuenciadores): ejemplos como comandos `goto` L (donde L es un rótulo o punto de programa.

    Otros saltos o secuenciadores son más estructurados como por ejemplo, los comandos `break` y `continue` de C. Generalmente se usan como saltos a puntos específicos en base a la sentencia en que se encuentran. Por ejemplo, en C, la sentencia *break* es un salto al final de la sentencia y puede aparecer en las sentencias *switch* (alternativas por casos) o en una iteración (`for`, `while` o `do-while`.

    En C, la sentencia `continue` (sólo puede aparecer en una iteración) produce un salto en el flujo de ejecución al comienzo de la iteración.

En algunos lenguajes, como en C, la expresión **return** <*expr*>; retorna (el valor de <*expr*>) inmediatamente de la función, haciendo que las sentencias subsiguientes no sean ejecutadas.

  - Sentencias de selección o condicionales: Ej: if-then-else, case, etc.
  - Iteración (o repetición):
    - Definida: el número de ciclos está determinado como por ejemplo la sentencia for de Pascal, for-each, etc.
    - Indefinida: el número de ciclos está determinado por una condición a evaluar en tiempo de ejecución. Ejemplo: sentencias while y repeat-until.

- *Operadores*: Un operador es una función con una sintaxis específica de invocación. Un operador puede verse como una abstracción sintáctica de la invocación a una función y su objetivo es ofrecer una notación más natural al programador. Por ejemplo, es común que un lenguaje contenga operadores aritméticos (sobre enteros, reales, ...), lógicos, de bits (ej: shift), sobre strings, etc.

  Los operadores se pueden usar en diferentes notaciones:

  - *infijos*: operadores binarios donde el operador se escribe entre los operandos. Ejemplos: x * y, a/2, ...
  - *prefijos*: operadores n-arios donde el operador antecede a su operando. Ej: -x, - - y (en C). Generalmente una invocación a una función toma esta forma: $f(e_1, \ldots, e_n)$, donde cada e_i, $(1 \leq i \leq n)$ es una expresión.
  - *posfijos*: operadores donde el operador sucede al operando. Ej: p^ (en Pascal)
  - *midfijos*: los operadores se mezclan con sus operandos. Ej: cond? t: f (en C).

- *Expresiones*: Una expresión representa un valor, es decir que puede ser asignable a una variable o constante, pasado como parámetro en una invocación a un procedimiento o función, etc.

  Una expresión se forma de la siguiente manera:

  i. Un valor de un tipo determinado es una expresión.
  ii. Una referencia (el uso de su identificador) a una variable o constante es una expresión.
  iii. Una invocación a una función, donde cada uno de sus argmentos[3] es una expresión, es una expresión.
  iv. Una aplicación de un operador a sus operandos (expresiones) correspondientes es una expresión.

[3] Un argumento en una invocación a un procedimiento o función se denomina *parámetro actual* o *real*. Un identificador de un parámetro en la declaración de un procedimiento o función se denomina *parámetro formal*.

v. Si `e` es una expresión, entonces `(e)` también lo es.

Una expresión que retorna un valor de verdad (boolean) se denomina *predicado*.

Los operadores generalmente están predefinidos en el lenguaje. Algunos lenguajes permiten al programador definir nuevos o al menos redefinir los existentes.

Cada operador tiene definida una cierta precedencia y asociatividad para permitir al programador evitar el uso excesivo de paréntesis para asociar los diferentes componentes de una expresión.

La precedencia permite establecer el orden de evaluación de una expresión con respecto a los demás operadores que aparecen en la misma. Es común que en el caso de los operadores aritméticos la precedencia habitual sea que se usa comúnmente en matemáticas (ej: `*` tiene mayor precedencia que +). Por ejemplo, la expresión `x+y*z` significa `x+(y*z)`.

La asociatividad define la parentización implícita cuando el operador aparece en secuencia. Por ejemplo, el operador + generalmente asocia a izquierda, mientras que el operador de asignación (=) en C, asocia a derecha. Es decir que la expresión `x+y+z` es equivalente a `(x+y)+z` y `x=y=z` es equivalente a `x=(y=z)`.

2.3 Tipos de datos

Definición 2.3.1 *Diremos que un* ***tipo de datos*** *es una descripción de un conjunto de valores junto con las operaciones que se aplican sobre ellos.*

Cada valor corresponde a un *tipo de datos*. Un valor v es de un tipo T si $v \in T$.

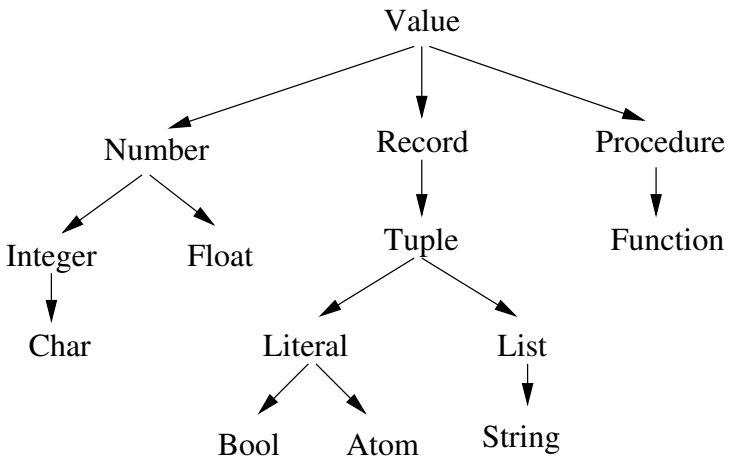


Fig. 2.1: Jerarquía de tipos básicos.

Entre los tipos denominados *básicos* tenemos dos clases, los cuales pueden jerarquizarse según la figura 2.1:

1. **elementales**: (también llamados básicos) tales como los números y literales. Estos valores son indivisibles.

2. **estructurados**: (o compuestos) como lo son los records. Son estructuras compuestos por otros valores.

Cabe aclarar que en otros lenguajes de programación existen otros tipos de datos básicos como las *referencias* y los arreglos.

Todo lenguaje de programación ofrece un conjunto de tipos de datos. La gran mayoría de los lenguajes de programación modernos permiten la declaración de nuevos tipos a partir de otros ya definidos. Esta es una de las principales características en la evolución de los lenguajes de programación. La posibilidad de definir nuevos tipos permite la implementación más elegante de *tipos abstractos de datos*, los cuales definen un tipo de datos en base a sus operaciones.

Los primeros lenguajes (como Fortran, Cobol, Basic), no ofrecían la posibilidad de definir nuevos tipos, por lo que limitaba bastante la claridad de las abstracciones.

A modo de ejemplo, en la figura 2.2, se muestran dos programas que suman números complejos. El programa de la izquierda está en un lenguaje que no permite definir nuevos tipos.

```
double c_sum_r(double r_1, double r_2)
                { ... }
double c_sum_i(double i_1, double i_2)
                { ... }
```

a) Sin nuevos tipos

```
type complex=(double r,i);
complex c_sum(complex c1, complex c2)
                { ... }
```

b) Con nuevos tipos definidos

Fig. 2.2: Ejemplo de programación con definición de nuevos tipos

Las diferentes versiones de la figura 2.2 muestran la diferencia en el estilo y claridad de los programas.

2.3.1 Tipos de datos simples o básicos

Cada lenguaje define un conjunto de tipos básicos como los siguientes:

- *numéricos*:
- *escalares o numerables*: Ej: enteros.
- *no escalares*: Ej: reales.
- caracteres
- lógicos
- punteros y/o referencias

Los enteros pueden representarse en complemento a dos con un número fijo de bits.

Algunos lenguajes permiten definir el número de bits de la representación, como por ejemplo, en C: *short int*, *long int*, etc.

Los reales se representan según algún formato estándar (ej: IEEE) en punto flotante o en punto fijo[4].

Los lógicos se representan comúnmente en un byte (ej: 0=false, 1=true), ya que generalmente es el tamaño mínimo de bloque de memoria direccionable.

Los punteros y referencias se usan para acceder indirectamente otros valores y se representan con una dirección de memoria, por lo que su tamaño de representación generalmente es igual al número de bits necesarios para direccionar una palabra de memoria.

2.3.2 Tipos de datos estructurados

Estos tipos de datos son aquellos que están compuestos por un conjunto de otros valores y pueden clasificarse en base a diferentes caracteríscas.

Por el tipo de sus elementos que contienen:

1. **homogéneos**: todos sus elementos son del mismo tipo, como por ejemplo los arreglos.

2. **heterogéneos**: sus elementos pueden ser de diferente tipo, como los registros.

Por sus características del manejo de sus elementos:

1. **estáticos**: el número de sus elementos está dado por una constante momento de su creación. Ejemplo: arreglos en Fortran o Pascal.

2. **dinámicos**: el número de sus elementos es variable. Ejemplos: listas, arreglos y registros dinámicos, ...

3. **semi-dinámicos**: el número de elementos está determinado por una variable en su creación pero luego el número de sus elementos se mantiene. Ejemplos: arreglos de C o C++.

Las estructuras de datos estáticas permiten una implementación mas eficiente, ya que la cantidad de memoria a asignar para su representación es conocida en tiempo de compilación y por lo tanto se pueden utilizar técnicas de manejo de memoria estática o mediante una pila.

Los arreglos y los registros estáticos o semi-dinámicos permiten una representación contig'ua de sus elementos, lo que permite la implementación simple de sus operadores

[4]Estos últimos son muy adecuados para representar importes monetarios.

de acceso a sus elementos, generalmente conocidas como *selectores*[5].

Generalmente encontramos en los lenguajes a los siguientes tipos:

- Arreglos: contienen a un conjunto de datos almacenados en forma contigüa, permitiendo acceder a cada elemento por medio de un operador de indexación. Un arreglo puede ser multidimensional. Si tienen una dimensión se denominan *vectores*, si tienen dos, *matrices*.

  El operador de indexación (denotado como [index] o (index)) es una función de un tipo índices (escalar) a un valor del tipo base (el tipo de los elementos contenidos).

  Ejemplo en C: float v[N]; ... v[0] = 1.5; ...

  Algunos lenguajes (ej: Pascal) permiten definir el dominio de los índices. Otros (ej: C, Java) el tipo índice es el rango $[0, N-1]$.

  Los arreglos son estructuras homogéneas, ya que todos los elementos que contiene son del mismo tipo.

  En algunos lenguajes son estáticos (como en Pascal) en cuanto a su dimensión, es decir que su dimensión debe ser una constante. En otros lenguajes (como C y Java), son semi-dinámicos, es decir que su dimensión puede determinarse en tiempo de ejecución (aunque luego no puede cambiarse). Algunos lenguajes también permiten que su dimensión varíe en ejecución, por los que se llaman dinámicos o de dimensión variable.

- Registros (o estructuras): sus componentes (campos) pueden ser de cualquier tipo. Sus campos son identificables, por lo que se denominan también tuplas rotuladas. Generalmente se representan en memoria de forma contigüa. El operador de acceso a los campos generalmente se denota con el símbolo . (punto).

  Ejemplo (C):

  struct person { int id, char[N] nombre; };

  person p; /* variable de tipo p */

  p.id = 1; ...

- Tuplas: son como los registros pero sus elementos se referencian generalmente usando proyecciones.

- Strings (cadenas de caracteres): generalmente se denotan entre " o '. Algunos lenguajes los representan como arreglos de caracteres (C, Java), mientras que otros los representan como listas (Haskell, ML).

2.3.3 Chequeo de tipos

Un lenguaje de programación generalmente deberá chequear los tipos de los argumentos u operandos de cada operación. El proceso de verficar si los operandos de una

[5] Como los operadores . de los registros y [] de los arreglos.

operación son del tipo correcto se denomina **type checking**. Ese chequeo, en base al momento en que realiza, puede ser:

- *Tipado estático*: el chequeo de tipos se realiza en tiempo de compilación.
- *Tipado dinámico*: el chequeo de tipos se realiza en tiempo de ejecución. Los valores tienen tipo pero las variables no (pueden tomar cualquier valor).

Obviamente el tipado estático requiere que los tipos de las expresiones se determinen (liguen) en tiempo de compilación, por lo que se requiere que el programador tenga que especificar los tipos en las declaraciones o sino el lenguaje tendrá que inferirlos de alguna manera. Algunos lenguajes, como Haskell o ML, las declaraciones de tipos es opcional, salvo en algunos casos que podría dar lugar a ambigüedades. Estos lenguajes tienen un sistema de tipos y algoritmos de inferencia de tipos muy elaborado.
Un sistema de tipos estático es una forma simple de verficación de programas de acuerdo a las reglas que define el sistema de tipos para cada una de sus operaciones. El tipado estático tiene grandes ventajas ya que un programa que *compile*, se podría decir que no tiene errores de tipos (está correctamente tipado).

Un lenguaje con tipado dinámico deberá acarrear información del tipo[6]de cada uno de los valores en tiempo de ejecución.
El tipado dinámico, detectará errores de tipos durante la ejecución, lo cual puede ser un gran inconveniente. La ventaja del tipado dinámico es que permite mayor flexibilidad para definir abstracciones polimórficas[7] con mayor libertad, lo cual da un mayor poder expresivo (relativo).

El sistema de tipos generalmente es conservador ya que el problema general es indecidible (es comparable al problema de la parada). Por ejemplo, en una sentencia de la forma: `if cond then 32 else 1+"hola"` , aún cuando `cond` evalúe siempre a `true` no compilará (`1+"hola"` es una expresión mal tipada) ya que en tiempo de compilación no se podría inferir que esta última expresión nunca se ejecutaría.

2.3.4 Sistemas de tipos fuertes y débiles

Un *sistema de tipos fuerte (strong)* impide que se ejecute una operación sobre argumentos de tipo no esperado.

Un lenguaje con un *sistema de tipos débil (weak)* realiza conversiones de tipos implícitas (casts) o aquellas explícitas. El resultado de la operación puede variar: por ejemplo, el siguiente programa:

```
var x = 5;
var y = ”ab” ;
x+y;
```

en Visual Basic da error de tipos, mientras que en JavaScript produce el string "5ab".

[6] Puede ser un simple rótulo o marca (tag).
[7] Una operación es polimórfica si acepta argumentos de diferentes tipos en diferentes invocaciones.

2.3.5 Polimorfismo y tipos dependientes

Algunos lenguajes permiten que las operaciones operen sobre argumentos de tipos específicos. En este caso se denomina sistema de tipos ***monomórfico***.

Un sistema de tipos que permite operaciones que aceptan argumentos que pueden ser instancias de una familia (relacionada de algún modo) de tipos se denomina un sistema ***polimórfico*** (muchas formas).

Por ejemplo, algunas operaciones en algunos lenguajes son polimórficos (ej: comando `write` o `read` de Pascal, aunque Pascal tiene un sistema de tipos monomórfico).

Otros lenguajes tienen verdaderos sistemas de tipos polimórficos, como Haskell, que soporta polimorfismo ***paramétrico*** (también llamado polimorfismo por instanciación). En este caso, una operación (función) define argumentos con un tipo variable. Cada instanciación (invocación a la función con valores concretos) determina (estáticamente) el tipo de cada argumento en cada invocación y chequea si son válidos.

En la programación orientada a objetos, una operación permite que sus argumentos sean instancias de una familia de tipos relacionados de la forma tipo-subtipo. Esas operaciones se definen como referencias del tipo superior.

Algunos tipos dependen de otros tipos (su tipo base). Por ejemplo, los arreglos son un tipo dependiente del tipo base (el tipo de cada uno de sus elementos). Estos tipos se denominan ***tipos dependientes***.

2.3.6 Seguridad del sistema de tipos

Un sistema de tipos es *seguro (safe)* si no permite operaciones que producirían condiciones inválidas. Por ejemplo, si un lenguaje no chequea que en una operación de indexación en un arreglo, el índice esté en el rango válido, es un lenguaje con un sistema de tipos inseguro.

2.4 Declaraciones, ligadura y ambientes

Una declaración relaciona un identificador con una o más entidades. Esta relación se denomina ***ligadura*** (***binding***, en inglés).

Por ejemplo, la declaración de una constante, como `const Pi = 3.141517;` relaciona al identificador `Pi` con un valor y con un tipo determinado (real en este caso).

Una declaración de un tipo liga un identificador a un tipo.

Una ligadura entre un identificador y alguna de sus propiedades puede determinarse en dos momentos:

i. durante la compilación: En este caso se denomina ***estática***.

ii. en ejecución o ***dinámica***.

En el ejemplo anterior ambas ligaduras ocurren estáticamente, pero en cambio en una declaración de una variable, su tipo puede determinarse estáticamente pero no su valor[8].

Una declaración de alguna entidad (tipo, variable, función, etc) *alcanza* a una cierta porción en un programa. La gran mayoría de los lenguajes de programación permiten que las declaraciones pertenezcan a algún bloque. El alcance de una declaración es similar al alcance de variables cuantificadas en lógica. Por ejemplo, en la siguiente fórmula

$$\forall x.(p(x) \wedge \exists y.q(x,y))$$

El cuantificador universal alcanza a toda la fórmula y el cuantificador existencial alcanza a la subfórmula $q(x,y)$.

Un ***ambiente*** (o ***espacio de nombres***) es un conjunto de ligaduras.

Cada sentencia de un programa se ejecuta en un cierto contexto o ambiente, por lo que su interpretación depende del ambiente en que se ejecuta.

En un lenguaje con ***alcance estático (static scope)*** las sentencias de un bloque (de un procedimiento o función) *se ejecutan en el ambiente de su declaración*. Esto significa que los valores (los tipos, etc.) de los identificadores referenciados en el bloque se determina en tiempo de compilación y corresponden a las declaraciones que contienen textualmete al bloque.

En un lenguaje con ***alcance dinámico (dynamic scope)***, un bloque se *evalúa en el contexto de su invocación*. Es decir que los identificadores referenciados pueden haberse definido en procedimientos o funciones que invocaron (directa o indirectamente) al actual. Esto significa que su contexto no depende del texto del programa, sino que su ambiente se determina sólo en tiempo de ejecución.

En el siguiente programa se muestra la diferencia entre alcance estático y dinámico.

```
var a = 1;
function f()
{
  return a+1;
}
function main()
{
  var a = 2;
  return f();
}
```

Con alcance estático la función f se evalúa en el ambiente de su definición, por lo que la invocación desde *main* retornará 2. Si f se evaluase en el ambiente de su invocación, en éste caso retornaría 3 (ya que haría referencia al valor de la variable `a` declarada en ***main***).

[8] Algunos lenguajes permiten determinar su valor inicial.

El alcance estático permite razonar sobre programas en forma modular, es decir sin tener en cuenta sus posibles contextos de invocación.

Por otro lado, el alcance dinámico provee mayor flexibilidad, pero la mayor desventaja es que no permite analizar un bloque de código en forma modular, ya que los identificadores a los que se hace referencia dependen de una determinada traza de ejecución.

Además, el alcance dinámico tiene otro problema. Suponga que la declaración de `a` en *main* se define con tipo *string*. El programa ahora tendría un error de tipos ya que la función *f* trataría de sumar un valor de tipo *string* con un entero.

Por esta razón, generalmente los lenguajes con alcance dinámico también tienen tipado dinámico.

Lenguajes como Clipper, Scheme y Perl permiten tener ambos tipos de alcance.

2.5 Excepciones

Una excepción es un evento que ocurre durante la ejecución de un programa, como producto de la ejecución de alguna operación inválida, y require la ejecución de código fuera del flujo normal de control. El manejo de excepciones (*exception handling*), es una característica de algunos lenguajes de programación, que permite manejar o controlar los errores en tiempo de ejecución. Proveen una forma estructurada de atrapar las situaciones completamente inesperadas, como también errores predecibles o resultados inusuales. Todas esas situaciones son llamadas *excepciones*.

Â¿Cómo manejamos situaciones excepcionales dentro de un programa?, como por ejemplo una división por cero, tratar de abrir un archivo inexistente, o seleccionar un campo inexistente de un registro. Debería ser posible que los programas las manejaran en una forma sencilla. Los lenguajes deberian proveer a los desarrolladores las herramientas necesarias para detectar errores y manejarlos dentro de un programa en ejecución. El programa no debería detenerse cuando esto pasa, mas bien, debería transferir la ejecución, en una forma controlada, a otra parte, llamada el manejador de excepciones, y pasarle a éste un valor que describa el error.

Algunos lenguajes de programación (como, por ejemplo, Lisp, Ada, C++, C#, Delphi, Objective C, Java, VB.NET, PHP , Python ,Eiffel y Ocaml) incluyen soporte para el manejo de excepciones. En esos lenguajes, al producirse una excepción se desciende en la pila de ejecución hasta encontrar un manejador para la excepción, el cual toma el control en ese momento.

Ejemplo de manejo de excepción en C:

```
#include <stdlib.h>
#include <stdio.h>
#include <windows.h>

int main()
```

```
{
    __try
    {
            int x,y = 0;
            int z;
            z = x / y;
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        MessageBoxA(0,"Capturamos la excepcion","SEH Activo",0);
    }

    return 0;
}
```

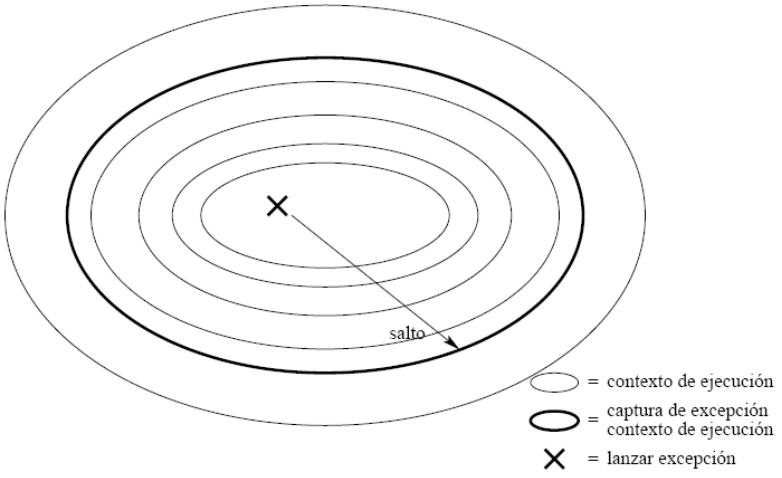


Fig. 2.3: Manejo de Excepciones.

Cómo debería ser el mecanismo de manejo de excepciones?.
Podemos realizar dos observaciones. La primera, el mecanismo debería ser capaz de confinar el error, i.e., colocarlo en cuarentena de manera que no contamine todo el programa. LLamamos a esto el principio de confinamiento del error. Suponga que el programa está hecho de componentes que interactúan, organizados en una forma jerárquica. Cada componente se construye a partir de componentes mas pequeños. El principio del confinamiento del error afirma que un error en un componente debe poderse capturar en la frontera del componente. Por fuera del componente, el error debe ser invisible o ser reportado en una forma adecuada.

Por lo tanto, el mecanismo produce un "salto" desde el interior del componente hasta su frontera. La segunda observación es que este salto debe ser una sola operación.

El mecanismo deber ser capaz, en una sola operación, de salirse de tantos niveles como sea necesario a partir del contexto de anidamiento (ver figura 2.3).

2.6 Qué es programar?

Analizando los elementos de un lenguaje de programación, se nota que cuando se programa, es decir, se soluciona un problema dado por medio de un programa de computadora, en realidad se termina definiendo un conjunto de tipos.

Por lo que podemos decir que programar es definir tipos y/o usar los valores y operaciones de un conjunto de tipos definidos.

Como se verá mas adelante, los diferentes modelos, paradigmas de programación proveen diferentes estilos en la definiciones de tipos.

2.7 Ejercicios

1. En las siguientes declaraciones Pascal, marque cuáles son definiciones:

   (a) `const` Pi = 3.1415;

   (b) `var` x:`integer`;

   (c) `External Procedure` P(x:`integer`);

   (d) `Function` Square(x:`integer`);
   ```
   begin
         Square = x*x
   end
   ```

   (e) `type` Person = `record` name:string; age:integer `end`;

2. Dar un ejemplo en un lenguaje imperativo de una expresión que no tenga transparencia referencial.

3. Muestre un ejemplo de programa en Haskell y en C que no compile por un error de tipos aún cuando nunca se produciría en ejecución.

4. Muestre un ejemplo de un programa Pascal que compile pero que tenga un error de tipos en ejecución.

5. Realice un experimento para determinar qué sistema de equivalencia de tipos usa C. Justifique su respuesta.

6. Experimentar la equivalencia de tipos con Haskell usando `newtype` y `type`. Cual es su conclusión?

7. Dar un programa C que muestre que no tiene un sistema de tipos seguro.

8. El principio de completitud de tipos determina que ninguna operación debería restringir *arbitrariamente* el tipo de sus operandos.
   Dar al menos tres ejemplos de operaciones de Pascal o C que violan este principio.

9. Hacer un análisis comparativo entre sistemas de tipos estáticos vs. dinámicos.

10. Dado el siguiente programa Pascal, determinar el ambiente de ejecución de las sentencias del cuerpo del procedimiento P.

```
Program Example;
Var x:integer;

  Function F(a:integer):integer;
    begin
      F := x*a
    end;


  Procedure P(y:integer)
    var x:integer;
        z:bool;
  begin
    x := 1;
    z := (y mod 2 == 0);
    if z then
      x := F(y+1)
    else
      x := F(y)
  end;

begin { main }
  x := 2;
  P(x)
end.
```

11. Experimentar con Perl: sistema de tipos y *scope* de variables (usar la palabra reservada *my*).

12. De un ejemplo de uso de excepciones en C++.

Capítulo 3

El modelo declarativo

Lo agradable de la programación declarativa es que uno escribe una especificación y la ejecuta como un programa. Lo desagradable de la programación declarativa es que algunas especificaciones claras resultan en programas increíblemente malos. La esperanza de la programación declarativa es que se pueda pasar de una especificación a un programa razonable sin tener que dejar el lenguaje.
Adaptación libre de The Craft of Prolog, Richard O'Keefe (1990)

La programación comprende tres cosas:

- Un modelo de computación, el cual define formalmente la sintaxis y la semántica de las frases (sentencias) del lenguaje.
- Un conjunto de técnicas de programación y principios de diseño. Generalmente ésto se conoce como el *modelo (o estilo) de programación*.
- Un conjunto de técnicas para razonar sobre los programas y calcular su eficiencia.

Otros autores llaman a los tres puntos que caracterizan a familias de lenguajes como que definen un *paradigma* de programación.

El modelo que se presentará es el modelo *declarativo*, el cual define mecanismos básicos para evaluar funciones parciales (o funciones sobre estructuras de datos parciales).

Este modelo se conoce comunmente como *programación sin estado (stateless)*, opuesto a la *programación con estados (statefull)* o *programación imperativa*.

El término *imperativo* se refiere a que el programador ve a los programas como una secuencia de *comandos* que cambian el estado (valores de un conjunto de variables) para llegar a obtener el efecto del programa deseado. En caso que los programas arrojen resultados, el estado final se correspondería con el resultado del programa.

La programación declarativa comprende básicamente dos paradigmas declarativos, el paradigma *funcional* y el *lógico*.

Estos paradigmas comprenden la programación en base a funciones sobre valores completos como Sheme y ML, como también a la programación no determinística sobre relaciones, como Prolog.

Cualquier operación computacional (un fragmento de programa con entradas y salidas) es declarativa si, cada vez que se invoca con los mismos argumentos, devuelve los mismos resultados independientemente de cualquier otro estado de computación (*transparencia referencial*. 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).

Los programas declarativos 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). Además, razonar sobre programas declarativos es sencillo, como sólo pueden calcular valores, se pueden usar técnicas sencillas de razonamiento algebraico y lógico.

Definimos un *componente* como un fragmento de programa, delimitado precisamente, con entradas y salidas bien definidas. Un componente se puede definir en términos de un conjunto de componentes más simples. Por ejemplo, en el modelo declarativo un procedimiento es una especie de componente. El componente que ejecuta programas es el más alto en la jerarquía. En la parte más baja se encuentran los componentes primitivos los cuales son provistos por el sistema. La interacción entre componentes se determina únicamente por las entradas y salidas de cada componente.

Una de las grandes ventajas de la programación declarativa es que es mas simple el razonamiento sobre los programas ya que al no tener estado es posible hacer razonamiento en forma modular.

En estos modelos, es común que la programación esté basada en definiciones recursivas (tanto de funciones como de datos), lo que permite razonar en términos matemáticos utilizando el principio de inducción (natural y estructural).

Como desventaja es necesario mencionar que hay algunos problemas que parecen ser modelados mas naturalmente con la noción de estado, como lo son la entrada-salida, computaciones usando evaluación parcial, sistemas reactivos, etc.

Si bien lo anterior es posible modelarlo sin estado, las implementaciones son mas complejas y difíciles de entender.

3.1 Un lenguaje declarativo

Como se vió en el capítulo 1, aquí se utilizará el enfoque del lenguaje *núcleo* para describir la sintaxis y la semántica del lenguaje a utilizar.

El modelo declarativo requiere que las variables no sean modificables, sólo inicializables, es decir que una vez que tomaron un valor éstas no pueden tomar otro.

3.1.1 Memoria de asignación única

Las variables en ésta memoria están inicialmente no ligadas y pueden ser ligadas sólo una vez.

Una *ligadura* es una asociación entre una variable y un valor.

Una variable ligada es indistiguible de su valor.

La memoria contendrá almacenados valores, esto es, estructuras de datos que representan valores del lenguaje y variables declarativas. Estas variables se representarán como *referencias* a valores, cuando estén ligadas. Las variables no ligadas no hacen referencia a ningún valor.

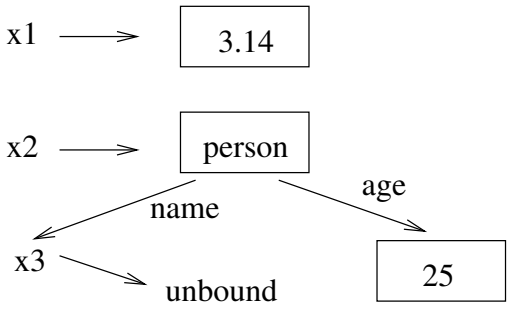


Fig. 3.1: Ejemplo de la memoria conteniendo variables y valores.

El lenguaje podrá determinar si una variable está ligada o no, lo cual requiere que la representación de las variables o, que tengan un tag indicando si están ligadas o no, o que las variables no ligadas hagan referencia a un valor especial que lo distinga de los demás.

La figura 3.1 muestra un ejemplo de la memoria de variables y valores en el modelo declarativo. La figura ilustra la representación de un valor *Integer* y del *Record* `person(name:Y age:25)`, donde el identificador `Y` corresponde a la variable $x3$ la cual no está ligada.

No se deben confundir las variables con los *identificadores* ya que es posible que algunas variables se identifiquen en el programa por medio de indentificadores, mientras que otras pueden no estar nombradas en el programa (pero sí tal vez denotada por alguna expresión). Generalmente esas variables se denominan *mudas*.

Además un mismo identificador puede estar asociado a una variable en un contexto (ambiente) y a otra variable en otro, como comúnmente ocurre en lenguajes que permiten definir múltiples ambientes.

Se debe notar que los valores pueden ser valores parciales, esto es una estructura puede contener componentes no ligados (unbounded).

En este sentido el lenguaje que se está introduciendo es más general que las variables de algunos lenguajes funcionales como por ejemplo ML o Haskell. Los lenguajes

funcionales se caracterizan por el hecho que todas sus variables están ligadas desde el momento de su introducción. Esto quiere decir que la creación de una variable va acompañada de su valor.

3.1.2 Creación de valores

La operación básica sobre la memoria de valores y variables es la creación de valores.

Una sentencia de la forma

```
X = 25
```

crea la estructura de datos para representar el valor 25 (ej: un bloque de memoria de 4 bytes) y liga la variable identificada por X al valor.

El lenguaje núcleo del modelo declarativo permite la creación de los siguientes tipos de valores:

1. Números (enteros y reales).

   Los caracteres serán representados como números (su código correspondiente según la codificación utilizada, ej: ASCII o Unicode).

2. Registros. Ejemplo: `person(name:"George" age:25)`.

3. Procedimientos. En el lenguaje núcleo los procedimientos son valores a diferencia del paradigma imperativo. El uso de procedimientos como valores otorga una gran flexibilidad para definir construcciones de mas alto nivel.

En los programas de ejemplo usaremos denotaciones de la forma *[x1 x2 ... xn]* para representar *listas*. Las listas se representarán como registros de la forma *list(X1:x1 X2:x2 ... XN:xn)*.

También denotaremos *tuplas* de la forma *tuple(x1 x2 ... xn)*, las cuales se representarán como registros de la forma *tuple(1:x1 2:x2 ... n:xn)*.

3.1.3 Un programa de ejemplo

El siguiente programa define un procedimiento que calcula el factorial de un número entero positivo.

Asumiremos que el procedimiento `Browse` es un procedimiento predefinido (built-in) en la biblioteca estándar del lenguaje, el cual muestra por consola (salida estándar) el valor pasado como argumento.

3.1.4 Identificadores de variables

Las variables mencionadas en el texto de un programa son identificadores de variables. En el ejemplo anterior, el identificador X sirve de nombre para una variable (ej: x_1) la cual está ligada al valor 25.

```
local Factorial in
    Factorial = proc {$ N ?R}
       if N == 0 then R = 1 else R = N * { Factorial N - 1 } end
    end
end
local F in
    F = {Factorial 5} – invocación a Factorial(5)
    {Browse F}
end
```

Fig. 3.2: Programa de ejemplo en el lenguaje núcleo declarativo.

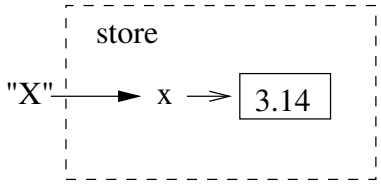


Fig. 3.3: Identificadores y variables.

La distinción entre variables e identificadores es importante para poder comprender la semántica que se dará mas adelante.

Los identificadores se encuentran fuera de la memoria de variables y valores[1]. En el lenguaje núcleo, los identificadores se denotarán en mayúsculas y utilizaremos minúsculas para referirnos a las variables dentro de la memoria de asignación simple.

Definición 3.1.1 *Un **ambiente** es un conjunto de asociaciones (pares) de identificadores y variables.*

Los identificadores de variables estarán contenidos en un área separada de memoria. De hecho en los lenguajes compilados, los identificadores (es decir, el ambiente) no existen en tiempo de ejecución. Estos son sólo identificadores que utiliza el compilador para *ligar* nombres con variables[2].

La figura 3.3 muestra la diferencia entre variables e identificadores.

3.1.5 Valores parciales, estructuras cíclicas y aliasing

La figura 3.1 muestra un ejemplo de un valor construido parcialmente. El registro (record) `person(name:X age:25)` es un valor parcial, ya que la variable asociada al identificador X (es decir, $x3$) no está ligada a ningún valor.

[1] Se encuentran en el *ambiente*, lo cual es un mapping de identificadores en variables del store.

[2] En realidad las variables se corresponden con direcciones de memoria o índices de una tabla. Un ambiente comúnmente se denomina tabla de símbolos.

El lenguaje también permite describir estructuras cíclicas, tal como el siguiente ejemplo:

```
X = node(value:345 tail:X)
```

Dos variables pueden referirse al mismo valor, por lo tanto en un programa los identificadores correspondientes también lo harán. Esto se denomina *aliasing* ya que puede haber varias variables denotando el mismo valor.

El concepto de aliasing cobrará mayor importancia durante el estudio del paradigma imperativo ya que en general se convierte en algo no deseable a la hora de razonar sobre los programas.

El problema es que cuando se cambia el valor de una variable, ese cambio se reflejará automáticamente en las demás variables ligadas entre ellas.

En el modelo declarativo el aliasing no genera problemas ya que las variables ligadas son *inmutables*.

El lenguaje declarativo permite unificar (igualar) dos variables. La sentencia

```
X = Y
```

liga las variables X e Y. Esta sentencia se denomina *variable-variable binding*.

3.2 Sintaxis del lenguaje núcleo declarativo

$< s >$	::=		sentencias
	|	**skip**	sentencia vacía (sin efecto)
	|	$< s >_1 < s >_2$	composición secuencial
	|	**local** $< x >$ **in** $< s >$ **end**	creación de variable
	|	$< x >_1 =< x >_2$	ligadura variable-variable
	|	$< x >=< v >$	creación de valor
	|	**if** $< x >$ **then** $< s >_1$ **else** $< s >_2$ **end**	Condicional
	|	**case** $< x >$ **of** $< pattern >$ **then** $< s >_1$ **else** $< s >_2$ **end**	Pattern matching
	|	$\{ < x > < y >_1 \ldots < y >_n \}$	Aplicación procedural
$< v >$	::=	$< number > \mid < record > \mid < procedure >$	valores
$< number >$	::=	$< integer > \mid < float >$	números
$< record >$	::=	$< pattern >$	records
$< pattern >$	::=	$< literal >$	patrones
	|	$< literal > (< field >_1 : x_1 \ldots < field >_n : x_n)$	
$< literal >$	::=	$< atom > \mid < bool >$	
$< field >$	::=	$< literal > \mid < integer >$	
$< bool >$	::=	**false** | **true**	booleans

Fig. 3.4: Sintaxis del lenguaje núcleo declarativo.

La figura 3.4 describe por medio de una EBNF la sintaxis del lenguaje núcleo declarativo.

Las denotaciones para las listas y tuplas serán adornos sintácticos para hacer más legibles los programas, pero se representerán como registros como se mencionó anteriormente.

Las cadenas de caracteres (strings) serán reprentados como listas de enteros pero se denotarán entre comillas dobles, tal como es habitual en la mayoría de los lenguajes de programación.

3.2.1 Porqué registros y procedimientos?

Los registros son una estructura muy útil para representar otros tipos de datos como las tuplas y listas. Además los registros permiten describir datos como por ejemplo componentes gráficos de un sistema de generación de interfaces de usuario.

Los procedimientos son más generales que las funciones en el sentido que es posible simular funciones asumiento que son procedimientos con un argumento adicional al final el cual se utilizaría para retornar el resultado.

Se utilizará la notación `?Arg` para denotar un parámetro formal que será utilizado como valor de salida en un procedimiento. En éste modelo se requerirá que el parámetro actual (o real) no esté ligado en el momento de la invocación.

El símbolo *?* permite dar claridad sintáctica a la definición de un procedimiento pero no tiene ninguna semántica particular.

3.2.2 Adornos sintácticos y abstracciones lingüísticas

Un *adorno sintáctico (syntactic sugar)* es una notación mas compacta (shortcut) de frases del lenguaje que se utilizan frecuentemente. Un adrono sintáctico no provee una nueva abstracción sino hace una notación mas conveniente o compacta.

Se permitirán varios adornos sintácticos.

- Sentencia **local**: en lugar de escribir algo como
  if N == 1 **then** s
  else
  local L **in**
  ...
  end
  end

  podemos escribir **if** N == 1 **then** s
  else L **in**
  ...
  end

  Otro adorno en una sentencia **local** es permitir la declaración de varios identificadores de la forma **local** `A B ...` **in** ... **end**, lo cual será equivalente al siguiente esquema:

local A **in**
    local B **in**
        ...
    end
end

- **Expresiones**: Las expresiones representan valores y se forman en base a valores y operadores. La evaluación de una expresión arroja un valor de un tipo determinado. Un operador es una función sobre valores de un tipo determinado. Se permitirá en el lenguaje que se escriban sentencias de la forma `A = B * (C + D)` la cual será traducida al lenguaje núcleo como
  local T1 T2 **in**
      {Number.'+' C D T1}
      {Number.'*' B T1 T2}
      A = T2
  end

  donde los procedimientos `Number.'+'` y `Number.'*'` pertenecen al módulo `Number`[3].

  En las expresiones es posible utilizar paréntesis para asociar operaciones.

  El lenguaje núcleo tendrá en cuenta la precedencia y asociatividad entre los operadores básicos del lenguaje de la manera habitual en otros lenguajes de programación. Por ejemplo la expresión `A + B * C` significa `A + (B * C)` (precedencia) y la expresión `A+B+C` significa `(A+B)+C` (asociatividad).

- **Definición de procedimientos**: se podrá utilizar la notación
  proc { P $X_1 \ldots X_n$ } ... **end** en lugar de
  P = **proc** { $ $X_1 \ldots X_n$ } ... **end**

Una *abstracción linguística* introduce una nueva abstracción para hacer más cómodo el lenguaje pero similarmente a los adornos sintácticos tienen un patrón de traducción al lenguaje núcleo.

Un ejemplo de abstracción sintáctica que se permitirá en el lenguaje declarativo es la definición de funciones. Una definición de la forma
fun { $ A B ... } ... < *expr* > **end**
se traducirá como
proc { $ A B ... ?R } ... R = < *expr* > **end**
donde < *expr* > será una expresión que deberá ser ejecutada al final de la función.

Las invocaciones a funciones podrán formar parte de las expresiones.

Una función se podrá definir como los procedimientos descriptos arriba. Es decir que en el ejemplo 3.2, la función `Factorial` se podría definir de la forma
fun { Factorial N }
    if N == 0 **then** R = 1

[3] Los módulos se describirán en capítulos subsiguientes.

```
        else R = N * { Factorial N - 1 }
        end
end
```

Otra abstracción lingüística definida será la *inicialización* de variables en la sentencia **local**. De este modo, por ejemplo, podremos escribir:

local X=1 Y=2*X **in** ... **end**

cuya traducción al lenguaje núcleo es natural y se deja como ejercicio.

3.2.3 Operaciones básicas del lenguaje

Operador	Descripción	Operandos
A==B	Comparación (igualdad)	Value
A\=B	Comparación (desigualdad)	Value
{IsProcedure P }	Test si P es Procedure	Procedure
A<=B	Comparación (menor o igual)	Number o átomo
A<B	Comparación (menor)	Number o átomo
A>=B	Comparación (mayor o igual)	Number o átomo
A>B	Comparación (mayor)	Number o átomo
A+B	Suma	Number
A-B	Diferencia	Number
A*B	Producto	Number
A/B	División	Number
A **div** B	División entera	Integer
A **mod** B	Módulo (resto)	Integer
{ Arity R }	Aridad	Record
{ Label R }	Rótulo	Record
R.F	Selección de campo	Record

Fig. 3.5: Operadores básicos del lenguaje núcleo declarativo.

La tabla de la figura 3.5 se listan los operadores básicos del lenguaje núcleo declarativo.

Se debe notar que las operaciones listadas en la figura 3.5 se muestran en notación infija, en realidad es un adorno sintáctico para hacer más legibles las expresiones tal como se describió anteriormente.

3.3 Semántica

La semántica se definirá en términos de un modelo operacional simple. Básicamente se definirá una máquina abstracta adecuada para la evaluación de funciones sobre valores parciales.

El modelo permitirá al programador razonar en forma simple sobre los programas sobre su correctitud y complejidad computacional. La máquina definida es una máquina de alto nivel la cual elimina los detalles encontrados en otras máquinas abstractas como registros del procesador y direcciones de memoria.

Si bien la máquina es de alto nivel, su implementación no tiene grandes dificultades.

Un programa en el lenguaje núcleo declarativo es simplemente una sentencia. A modo de ejemplo, dado el siguiente programa:

local A **in**
    A = 20
    {Browse A * 2}
end

Para dar una noción informal de su funcionamiento, la máquina abstracta es una máquina pila en la cual inicialmente se encuentra la sentencia inicial. La máquina entra en un ciclo en el cual realiza los siguientes pasos:

1. Tomar (Pop) la sentencia s del tope de la pila.

2. Ejecutar las operaciones básicas de la máquina (operación `Push, Pop, Bind y operaciones sobre ambientes`) según la semántica de s. Esto puede hacer que se apilen nuevos elementos.

La ejecución finaliza cuando la pila queda vacía.

3.3.1 La máquina abstracta

La ejecución de un programa se define en términos de pasos de computaciones sobre la máquina abstracta, lo cual es una secuencia de estados de ejecución.

Definición 3.3.1 *La **memoria de asignación única** σ es un conjunto de variables y valores. Las variables en la memoria forman una partición de conjuntos que son iguales pero no ligadas a valores y variables ligadas (a números, registros o procedimientos).*

Por ejemplo, la memoria de asignación única se denota como $\{x_1, x_2 = x_3, x_4 = 25\}$. En este ejemplo, la variable x_1 no está ligada (y no está igualada a ninguna otra), las variables x_2 y x_3 están igualdas entre sí y no están ligadas, finalmente la variable x_4 está ligada al valor 25 (tipo *integer*).

Definición 3.3.2 *Un **ambiente** E es un mapping de identificadores a variables en σ.*

Denotaremos un ambiente E como un conjunto de pares. Por ejemplo, $\{X \rightarrow x_1, Y \rightarrow x_2\}$.

Definición 3.3.3 *Una **sentencia semántica** es un par de la forma $(< s >, E)$, donde $< s >$ es una sentencia y E es un ambiente.*

Intuitivamente, una sentencia semántica asocia una sentencia con su ambiente de ejecución (conjunto de identificadores visibles por la sentencia).

Definición 3.3.4 *Un **estado de ejecución** es un par de la forma (ST, σ), donde ST es una pila (stack) de sentencias semánticas y σ es una memoria de asignación única.*

Denotaremos una pila como $[(< s >_1, E_1), \ldots, (< s >_n, E_n)]$, donde el elemento de mas a la izquierda corresponde al tope.

Definición 3.3.5 *Una **computación** es una secuencia de estados de ejecución comenzando desde el estado inicial:*
$(ST_0, \sigma_0) \rightarrow (ST_1, \sigma_1) \rightarrow \ldots$

Cada transición en una computación es un *paso de computación.*

3.3.2 Ejecución de un programa

Dado un programa o sentencia $< s >$, el estado de ejecución inicial es denotado como
$([(< s >, \emptyset)], \emptyset)$
es decir, la pila tiene la sentencia correspondiente al programa con un ambiente vacío y la memoria está vacía.

La máquina puede estar en tres estados de ejecución posibles:

- **Runnable** (o activa), es decir que puede realizar un próximo paso de ejecución.
- **Terminated** cuando la pila está vacía.
- **Suspended** la pila no está vacía pero no puede realizar un próximo paso de ejecución[4]. En el modelo secuencial, si la máquina entra al estado **suspended**, el proceso (programa en ejecución) quedará congelado sin poder progresar.

3.3.3 Operaciones sobre ambientes

La máquina abstracta tiene que realizar ciertas operaciones sobre ambientes.

- **Adjunction**: define un nuevo ambiente en base de uno existente adicionando un nuevo par (mapping). Ejemplo: $E + \{X \rightarrow x\}$ denota un nuevo ambiente E' construido a partir de E con un par adicional.
- **Restriction**: define un nuevo ambiente el cual es un subconjunto de uno existente.

  La notación $E \mid_{\{x_1,\ldots,x_n\}}$ denota un nuevo ambiente $E' = dom(E) \cap \{x_1, \ldots, x_n\}$ y $E'(x) = E(x)$ pata todo $x \in dom(E')$.

  Es decir que el nuevo ambiente no contiene otros identificadores más que $\{x_1, \ldots, x_n\}$.

Se utilizará la notación $E(< X >)$ para referirnos al valor asociado al identificador $< X >$ en el ambiente E.

[4] Cuando se introduzca la noción de *dataflow variables* y *concurrencia* se verá el porqué éste estado tiene sentido.

3.3.4 Semántica de las sentencias

En esta sección se define la semántica operacional de la máquina abstracta. Recordar que la máquina ejecuta el siguiente ciclo:

```
state = running
while stack not empty and state = running do
    s := top stack
    pop
    exec s
end
```

A coontinuación se define la semántica de cada sentencia del lenguaje, es decir qué deberá implementar `exec`.

- (**skip**, E): Pop sobre el stack.
- ($s_1 s_2, E$): (composición secuencial).
  1. Push (s_2, E).
  2. Push (s_1, E).
- (**local** X **in** s **end**, E): (declaración de variable).
  1. Crear una nueva variable x en la memoria σ.
  2. Push ($s, E + \{X \rightarrow x\}$).
- ($X = Y, E$): (ligadura variable-variable).
  1. $Bind(E(X), E(Y))$ en la memoria σ.
     El operador $Bind$ se explicará en detalle mas abajo. Informalmente, asocia (*unifica*) las variables X y Y para que $E(X) == E(Y)$.
- ($X = v, E$): (creación de un valor). v es un valor parcial de tipo Number, Record o Procedure.
  1. Crear la representación del valor v en la memoria σ. Todos los identificadores de v tienen que referenciar a entidades determinadas por E.

     Ya se ha visto cómo se crean valores numéricos o registros, pero los procedimientos (también llamados *clausuras*) se crean de la siguiente manera:
     Un valor **proc** { \$ $y_1 \ldots y_n$ } s **end**, declarado en el ambiente E, hace que se cree un estado de ejecución
     (**proc** { \$ $y_1 \ldots y_n$ } s **end**,CE)
     donde $CE = E\,|_{\{z_1,\ldots,z_1\}}$ y $\{z_1, \ldots, z_1\}$ son los identificadores que aparecen *libres* en el procedimiento, es decir que no son uno de sus parámetros formales o son variables locales definidos en s.
     Las variables en $\{z_1, \ldots, z_1\}$ se denominan *referencias externas del procedimiento*.

2. Hacer que x se refiera a v en σ.

- (**if** X **then** s_1 **else** s_2 **end** , E): (condicional).

  1. Si X no está ligada ($E(X)$ es indeterminado), `state := Suspended`.
  2. Sino, si X está ligada a un valor que no es de tipo *Boolean*, disparar un error (tipo inválido).
  3. En otro caso, si X es **true**, hacer Push (s_1,E), sino Push (s_2,E).

- (**case** X **of** $lit(feat_1 : X_1 \ \ldots \ feat_n : X_n)$ **then** s_1 **else** s_2 **end** , E): (pattern matching).

  1. Si X no está ligada ($E(X)$ es indeterminada), entonces `state := Suspended`.
  2. Sino, si $Label(E(X)) = lit$ y $Arity(E(X)) = [feat_1 \ \ldots feat_1]$, entonces hacer
     Push $(s_1, E + \{X_1 \rightarrow E(X).feat_1, \ \ldots \ , X_n \rightarrow E(X).feat_n\})$
  3. en otro caso, Push (s_2, E).

- $(\{X \ Y_1 \ \ldots Y_n\}, E)$: (aplicación procedural).

  1. Si X no está ligada ($E(X)$ es indeterminada), entonces `state := Suspended`.
  2. Sino, si no $IsProcedure(E(X))$ o $Arity(E(X) <> n$, disparar un error de tipo.
  3. Si $E(X)$ tiene la forma (**proc** { \$ $z_1 \ \ldots z_n$ } s **end**,CE), hacer
     Push (s,$CE + \{z_1 \rightarrow E(Y_1), \ \ldots \ z_n \rightarrow E(Y_n)\}$)

3.3.5 Ejemplo de Ejecución

A continuación se analizará el comportamiento del siguiente programa:

$$s \equiv \left\{ \begin{array}{l} \textbf{local X in} \\ \qquad X = 1 \\ s_1 \equiv \left\{ \begin{array}{l} \textbf{local X in} \\ \qquad X = 2 \\ \qquad \{\text{Browse X}\} \\ \textbf{end} \end{array} \right. \\ s_2 \equiv \{\text{Browse X}\} \\ \textbf{end} \end{array} \right.$$

1. El estado de ejecución inicial (único elemento en la pila) es
   $([(s, \emptyset)], \emptyset)$
   es decir, la sentencia s con un ambiente vacío y la memoria está vacía.

2. La ejecución de la primer sentencia **local** de s, obtenemos
   $([(s_1 \ s_2, \{X \rightarrow x_1)\}], \{x_1 = 1\})$

3. La ejecución de la composición secuencial obtenemos
   $([(s_1, \{X \rightarrow x_1)\}), (s_2, \{X \rightarrow x_1\})], \{x_1 = 1\})$
   cada sentencia en la pila tiene su propio ambiente (el mismo en este caso).

4. La primer sentencia de s_1 es una sentencia **local** y su ejecución deja
   $([(X = 2\,\{BrowseX\}, \{X \rightarrow x_2\}), (s_2, \{X \rightarrow x_1\})], \{x_1 = 1, x_2\})$
   esto es, la sentencia s_1 con el ambiente $\{X \rightarrow x_2\}$ y la sentencia s_2 con el ambiente $\{X \rightarrow x_1\}$.
   Se debe notar que el identificador X se refiere a dos variables distintas en cada sentencia.

5. Después de ejecutar la sentencia $X = 2$ (binding) obtenemos
   $([(\{BrowseX\}, \{X \rightarrow x_2\}), (s_2, \{X \rightarrow x_1\})], \{x_1 = 1, x_2 = 2\})$
   La invocación al procedimiento `{Browse X }` muestra el valor de la variable `X` (en este caso 2, el valor de la variable x_2).

6. Finalmente la máquina queda en la siguiente configuración
   $([(\{BrowseX\}, \{X \rightarrow x_1\})], \{x_1 = 1, x_2 = 2\})$
   el cual imprime el valor 1.

   Luego de la ejecución de ésta última sentencia la pila queda vacía por lo que finaliza la ejecución del programa.

Lo anterior muestra que la semántica de la sentencia **local** introduce un nuevo ambiente y las sentencias hacen referencia a las variables definidas en el contexto de su declaración.

Como vimos en la sección 2.4 del capítulo 2 esto se denomina ***alcance estático (static scope)*** ya que el ambiente de referenciación de cada sentencia se puede determinar sin necesidad de ejecutar el programa.

Lo anterior tiene un mayor impacto en la definición de procedimientos y funciones, ya que en una invocación, el procedimiento se ejecuta en el ambiente de su declaración. Es decir que cualquier referencia no local en el cuerpo del procedimiento (variables locales o parámetros formales), se refiere a identificadores ligados a variables en su ambiente de declaración y no en el ambiente de su invocante.

3.3.6 Sistema de Tipos del lenguaje núcleo declarativo

Cabe aclarar que es posible realizar el chequeo de tipos estáticamente en el lenguaje núcleo declarativo definido (aún sin cambiar su sintaxis, cómo?).

Por otro lado, el lenguaje núcleo declarativo es seguro ya que tal como se describió en la semántica, antes de realizar una operación se verifica que los datos sobre los que opera sean del tipo esperado.

3.3.7 Manejo de la memoria

Como se puede apreciar en la ejecución del programa de ejemplo de la sección 3.3.5, el lenguaje núcleo tiene sentencias de creación de valores pero no para su destrucción.

Las variables y valores continúan estando en la memoria aún cuando ya no exista la posibilidad que el programa haga una referencia a ellas (porque no existen en el ambiente identificadores que las referencien).

Si bien esto no es un inconveniente, ya que por ahora sólo nos interesa definir su semántica formal, una implementación real debería tomar esto en cuenta.

Una implementación debería incluir un mecanismo para podel eliminar de la memoria (para que en ésta no se extinga su capacidad de almacenamiento) aquellas variables y valores que ya no podrán ser referenciadas por el programa.

Este mecanismo se conoce generalmente como ***recolección de basura (garbage collection)***. El recolector de basura realiza un barrido de la pila y la memoria para detectar y eliminar variables y valores inalcanzables.

En el capítulo de manejo de la memoria se verá en mas detalle el funcionamiento de un recolector de basura.

3.3.8 Unificación (operador '=')

El operador = produce asociaciones (o bindings) en la memoria de variables y valores (*value store*).

Definición 3.3.6 *Un **término** es un átomo, un número, un registro o un identificador.*

Definición 3.3.7 *Una **sustitución** de una variable por un término es una función que toma una variable X, un término t y una expresión E y retorna una expresión E' la cual es obtenida a partir de E en la cual se han reemplazado todas las ocurrencias de X por T.*

Definición 3.3.8 *Se dice que dos expresiones E_1 y E_2 **unifican** si existe una composición de **sustituciones** tal que aplicada E_1 y E_2, las expresiones resultantes son iguales (sintácticamente).*

Es posible ver a la unificación como una operación que produce información adicional en la memoria σ.

El operador = tiene las siguientes propiedades:

1. *Simétrico*: por ejemplo `X = person(name:X1 age:25)` es igual que `person(name:X1 age:25) = X`.

2. *Opera sobre valores parciales*: por ejemplo, (si X1 y X2 no están ligadas)

   `person(name:X1 age:25) = person(name:"George" age:X2`

   produce los *bindings* $X1 \rightarrow$ "George" y $X2 \rightarrow 25$.

3. *Puede no causar cambios*: en el caso que ambos valores (parciales o no) sean iguales (unifiquen).

4. *Puede causar error*: en el caso que ambos valores sean incompatibles.

   Por ejemplo: `person(name:X1 age:25)=person(name:X1 age:26)`

5. *Puede crear estructuras cíclicas*: como en el siguiente caso:

   `L=node(value:X rest:L)`

6. *Puede ligar estructuras cíclicas*: es posible unificar las siguientes expresiones:

   `X=f(a:X b:_)` y `Y=f(a:_ b:Y)`

   (b:_) significa que el campo B no está ligado (unbound).

   La sentencia `X=Y` crea una estructura con dos ciclos, la cual puede expresarse como `X=f(a:X b:X)`.

   La figura 3.6 muestra su representación en la memoria.

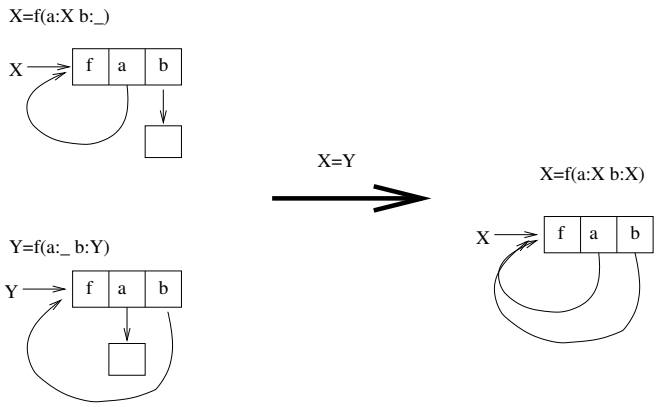


Fig. 3.6: Ejemplo de estructuras cíclicas.

3.3.9 El algoritmo de unificación

Para dar una definición precisa de la unificación, se definirá la operación *unify(x,y)* que unifica dos valores parciales x e y en la memoria σ.

Es necesario definir algunos conceptos previos.

Tal como se vio anteriormente, la memoria de asignación única σ, está particionada en los siguientes conjuntos:

- Conjuntos de variables no ligadas iguales (han sido igualadas por variable-variable binding). Estos conjuntos se denominan *conjuntos de equivalencia*.

- Variables ligadas a valores de tipo Number, Record o Procedure (variables determinadas).

Un ejemplo de la memoria particionada es: $\{x_1 = foo(a : x_2), x_2 = 25, x_3 = x_4 = x_5, x_6, x_7 = x_8\}$.

El algoritmo de unificación se basa en las operaciones primitivas *bind* y *merge*, las cuales operan sobre la memoria σ.

1. $bind(ES, v)$: liga las variables en el conjunto de equivalencia ES con el valor v. Por ejemplo, $bind(\{x_7, x_8\}, foo(a : x_2))$ modifica la memoria dada en el ejemplo anterior resultando la memoria

   $\{x_1 = foo(a : x_2), x_2 = 25, x_3 = x_4 = x_5, x_6, x_7 = foo(a : x_2), x_8 = foo(a : x_2)\}$.

2. $merge(ES_1, ES_2)$: hace que en la memoria los conjuntos de equivalencia ES_1 y ES_2 se fusionen (unión) en un conjunto de equivalencia.

   Nuevamente con la memoria de ejemplo de arriba, $merge(\{x_3 = x_4 = x_5\}, \{x_6\})$ modifica la memoria σ como

   $\{x_1 = foo(a : x_2), x_2 = 25, x_3 = x_4 = x_5 = x_6, x_7 = x_8\}$.

$unify'(x, y, L)$

1. Si $(x, y) \in WL$ o $(y, x) \in WL$, terminar.
2. Si $x \in ES_x$ e $y \in ES_y$, entonces hacer $merge(ES_x, ES_y)$.
3. Si $x \in ES_x$ e y es determinada, hacer $bind(ES_x, y)$.
4. Si $y \in ES_y$ y x es determinada, hacer $bind(ES_y, x)$.
5. Si x está ligada a $l(l_1 : x_1 \ \ldots \ l_n : x_n)$ e y está ligada a $l'(l'_1 : y_1 \ \ldots \ l'_m : y_m)$, con $l \neq l'$ o $\{l_1, \ldots, l_n\} \neq \{l'_1, \ldots, l'_m\}$, disparar un error.
6. Si x está ligada a $l(l_1 : x_1 \ \ldots \ l_n : x_n)$ e y está ligada a $l(l_1 : y_1 \ \ldots \ l_n : y_n)$, hacer $unify'(x_i, y_i, WL + \{(x, y)\})$, para todo i, $1 \leq i \leq n$.

$unify(x, y) = unify'(x, y, \emptyset)$

Fig. 3.7: El algoritmo de unificación.

La figura 3.7 muestra el algoritmo de unificación. Se debe notar que el algoritmo funciona aún con estructuras cíclicas ya que en WL se recuerda la lista de variables ya unificadas.

Esto impide que el algoritmo entre en un ciclo infinito, ya que a lo sumo se invoca a lo sumo una vez a $unify'(x, y)$ por cada par de variables x e y. Como la cantidad de variables en la memoria es finita, el algoritmo termina.

3.3.10 Igualdad (operador ==)

La operación ==, también llamada *entailment check*[5], es una función lógica (retorna **true** o **false**) que chequea si x e y son iguales o no.

El operador sigue el siguiente algoritmo:

1. Retorna **true** si los grafos cuyos vértices parte de x e y son iguales, es decir, tienen la misma estructura.

2. Retorna **false** si los grafos cuyos vértices parte de x e y son diferentes, o sea que no tienen la misma estructura.

3. Pone la máquina abstracta en modo **Suspended** cuando encuentra algún componente una de estructuras que no está ligado y en la otra sí.

Este tipo de igualdad (o equivalencia) se conoce como **emphequivalencia estructural**.

Tiene como ventaja que es posible comparar estructuras complejas pero el algoritmo requiere tiempo lineal sobre el tamaño de la estructura. El algoritmo deberá tener cuidado con las estructuras cíclicas, pero el problema ha sido ampliamente estudiado en el campo de los algoritmos sobre grafos.

Otros lenguajes realizan sólo comparaciones sobre valores de los tipos básicos no estructurados, como es común en lenguajes imperativos como Pascal o C.

La igualdad de valores estructurados generalmente la tiene que definir el programador.

3.4 El modelo declarativo con Excepciones

Extendemos el modelo de computación declarativa con excepciones. En la figura 3.8 se presenta la sintaxis del lenguaje núcleo extendido. Los programas pueden utilizar dos declaraciones nuevas, `try` y `raise`.

$< s >::=$		sentencias
	skip	sentencia vacía (sin efecto)
	$< s >_1 < s >_2$	composición secuencial
	local $< x >$ **in** $< s >$ **end**	creación de variable
	$< x >_1 =< x >_2$	ligadura variable-variable
	$< x >=< v >$	creación de valor
	if $< x >$ **then** $< s >_1$ **else** $< s >_2$ **end**	Condicional
	case $< x >$ **of** $<$ *pattern* $>$ **then** $< s >_1$ **else** $< s >_2$ **end**	Pattern matching
	$\{ < x > < y >_1 \ldots < y >_n \}$	Aplicación procedural
	try $< s >_1$ `catch` $< x >$ **then** $< s >_2$ **end**	Contexto de la excepción
	raise $< x >$ **end**	Lanzamiento de la excepción

Fig. 3.8: El lenguaje núcleo declarativo con excepciones.

[5] El término *entailment* viene de la lógica, ya que puede verse como $\sigma \models (x = y)$ (desde *sigma* puede inferirse que $x = y$.

Una declaración `try` puede especificar una cláusula `finally` que siempre se ejecutará, ya sea que la ejecución de la declaración lance una excepción o no.
`try` $< s >_1$ `finally` $< s >_2$ `end`

La cláusula `finally` es útil cuando se trabaja con entidades externas al modelo de computación. Con `finally` podemos garantizar que se realice alguna acción de "limpieza" sobre la entidad, ocurra o no una excepción.

3.4.1 Semántica del *try* y *raise*

Como vimos en la sección 2.5 del capítulo 2, el mecanismo de control de excepciones debe producir un "salto" desde el interior de un componente hasta su frontera. El mecanismo deber ser capaz, en una sola operación, de saltearse tantos niveles como sea necesario a partir del contexto de anidamiento. En la semántica, se debe definir un contexto como una entrada en la pila semántica, es decir, una instrucción que tiene que ser ejecutada mas adelante. Los contextos anidados se crean por invocaciones a procedimientos y composiciones secuenciales.
El modelo declarativo no puede dar ese salto en una sola operación. El salto debe ser codificado explícitamente en saltos pequeños, uno por contexto, utilizando variables booleanas y condicionales. Esto vuelve a los programas mas voluminosos, especialmente si debe añadirse código adicional en cada sitio donde puede ocurrir un error.
Se deja como ejercicio proponer una extensión sencilla del modelo que satisfaga las siguientes condiciones (ver la sección *ejercicios* del capítulo). La declaración `try` crea un contexto para capturar excepciones junto con un manejador de excepciones. La declaración `raise` salta a la frontera del contexto para capturar la excepción mas interna e invoca al manejador de excepciones allí. Declaraciones `try` anidadas crean contextos anidados. Ejecutar `try` $< s >_1$ `catch` $< x >$ `then` $< s >_2$ `end` es equivalente a ejecutar $< s >_1$, si $< s >_1$ no lanza una excepción. Por otro lado, si $< s >_1$ lanza una excepción, i.e., ejecutando una declaración `raise`, entonces se aborta la ejecución (aún en curso) de $< s >_1$. Toda la información relacionada con $< s >_1$ es sacada de la pila semántica. El control se transfiere a $< s >_2$, pasándole una referencia a la excepción mencionada en $< x >$.

A modo de ayuda, considere una tercera declaración, `catch` $< x >$ `then` $< s >$ `end`, que se necesita internamente para definir la semántica pero no es permitida en los programas. La declaración `catch` es una "marca" sobre la pila semántica que define la frontera del contexto que captura la excepción.

3.5 Técnicas de Programación Declarativa

Uno de los factores que determinan un estilo o paradigma de programación es un conjunto de técnicas de programación y principios de diseño.

La técnica básica para escribir programas declarativos es considerar el programa como un conjunto de definiciones de funciones recursivas, utilizando programación de alto orden para simplificar la estructura del programa. Programación de alto orden

significa que las funciones pueden tener otras funciones como argumentos o como resultados. Esta capacidad es el fundamento de todas las técnicas para construir abstracciones.

3.5.1 Lenguajes de Especificación

Los proponentes de la programación declarativa afirman algunas veces que ésta les permite prescindir de la implementación, pues la especificación lo es todo. Esto es verdad en un sentido formal, pero no en un sentido práctico. Lenguajes declarativos sólo pueden usar las matemáticas que se puedan implementar eficientemente.

Es posible definir un lenguaje declarativo mucho más expresivo que el que usamos en el libro. Tal lenguaje se llama un lenguaje de especificación. Normalmente es imposible implementar eficientemente lenguajes de especificación. Esto no significa que sean poco prácticos, por el contrario, son una herramienta importante para pensar sobre los programas. Ellos pueden ser usados junto con un probador de teoremas, es decir, un programa que puede realizar cierto tipo de razonamientos matemáticos. Con la ayuda del probador de teoremas, un desarrollador puede probar propiedades muy fuertes sobre su programa.

3.5.2 Computación Iterativa

Una computación iterativa es un ciclo que durante su ejecución mantiene el tamaño de la pila acotado por una constante, independientemente del número de iteraciones del ciclo. Este tipo de computación es una herramienta básica de programación. Hay muchas formas de escribir programas iterativos, y no siempre es obvio determinar cuando un programa es iterativo. En general una computación iterativa es un transformador de estados:

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots S_{final}$$

Se presenta un esquema general para construir muchas computaciones iterativas interesantes en el modelo declarativo.

```
fun { Iterar S_i }
    if { Es_Final S_i } then S_i
    else S_{i+1} in
            S_{i+1} = { Transformar S_i }
            { Iterar S_{i+1} }
    end
end
```

las funciones *EsFinal* y *Transformar* son dependientes del problema. Cualquier programa que sigue este esquema es iterativo. El tamaño de la pila de la máquina abstracta no crece cuando se ejecuta, mas bien es acotado.

Un ejemplo de computación iterativa es el método de Newton para calcular la raíz cuadrada de un número positivo real x. la idea es comienzar con una adivinanza a de la raíz cuadrada, y mejorar esta adivinanza iterativamente hasta que sea suficientemente

buena (cercana a la raíz cuadrada real) (ver implementación en la sección de ejercicios del actual capítulo).

3.5.3 Del esquema general a una abstracción de control

El esquema implementa un ciclo **while** general con un resultado calculado. Para que el esquema se vuelva una abstracción de control, tenemos que parametrizarlo extrayendo las partes que varían de uno a otro uso. Hay dos partes de esas: las funciones *EsFinal* y *Transformar*. Colocamos estas dos partes como parámetros de *Iterar*:

fun { Iterar S_i Es_Final Transformar }
    if { Es_Final S_i } **then** S_i
    else S_{i+1} **in**
        S_{i+1} = { Transformar S_i }
        { Iterar S_{i+1} Es_Final Transformar}
    end
end

Para utilizar esta abstracción de control, los argumentos EsFinal y Transformar se presentan como funciones de un argumento. Pasar funciones como argumentos de funciones es parte de un rango de técnicas de programación llamadas programación de alto orden.

3.5.4 Computación Recursiva

La recursión es más general que esto. Una función recursiva puede invocarse a sí misma en cualquier lugar en el cuerpo y puede hacerlo más de una vez. En programación, la recursión se presenta de dos maneras: en funciones y en tipos de datos. Las dos formas de recursión están fuertemente relacionadas, pues las funciones recursivas se suelen usar para calcular con tipos de datos recursivos. Una computación iterativa (recursiva a la cola) tiene un tamaño de pila constante, como consecuencia de la optimización de última invocación. Este no es siempre el caso en computación recursiva. El tamaño de la pila puede crecer a medida que el tamaño de la entrada lo hace. Algunas veces es inevitable, por ejemplo cuando se realizan cálculos con árboles. Una parte importante de la programación declarativa es evitar que el tamaño de la pila crezca, siempre que sea posible hacerlo. En esta sección se presenta un ejemplo de cómo hacerlo. Empezamos con un caso típico de computación recursiva que no es iterativa, la definición ingenua de la función factorial.

```
fun {Fact N}
  if N==0 then 1
  elseif N>0 then N*{Fact N-1}
  else raise errorDeDominio end
  end
end
```

Notar que factorial es una función parcial. No está definida para N negativo. Así definida el tamaño de la pila es creciente cuyo máximo tamaóo es proporcional al argumento N de la función.

Podríamos convertir ésta versión en una versión recursiva a la cola de la siguiente manera. En la versión anterior los cálculos se realizan:

$$(5 * (4 * (3 * (2 * (1 * 1)))))$$

Si reorganizamos los números así:

$$(((((1 * 5) * 4) * 3) * 2) * 1)$$

entonces los cálculos se podríaan realizar incrementalmente, comenzando con 1*5. Esto da 5, luego 20, luego 60, luego 120, y finalmente 120. La definición iterativa que realiza los cálculos de esta manera es:

```
fun {Fact N}
  fun {FactIter N A}
    if N==0 then A
    elseif N>0 then {FactIter N-1 A*N}
    else raise errorDeDominio end
    end
  end
in
  {FactIter N 1}
end
```

3.5.5 Programación de Alto Orden

La programación de alto orden es la colección de técnicas de programación disponibles cuando se usan valores de tipo procedimiento en los programas. Los valores de tipo procedimiento se conocen también como clausuras de alcance léxico.

El término alto orden viene del concepto de orden de un procedimiento. Un procedimiento en el que ningún argumento es de tipo procedimiento se dice de primer orden. Un lenguaje que sólo permite esta clase de procedimientos se llama un lenguaje de primer orden. Un procedimiento que tiene al menos un procedimiento de primer orden como argumento se llama de segundo orden. Y así sucesivamente, un procedimiento es de orden $n+1$ si tiene al menos un argumento de orden n y ninguno de orden más alto. La programación de alto orden significa, entonces, que los procedimientos pueden ser de cualquier orden. Un lenguaje que permite esto se llama un lenguaje de alto orden.

Existen cuatro operaciones básicas que subyacen a todas las técnicas de programación de alto orden:

- Abstracción procedimental: la capacidad de convertir cualquier declaración en un valor de tipo procedimiento.
- Genericidad: la capacidad de pasar un valor de tipo procedimiento como argumento en una invocación a un procedimiento.

- Instanciación: la capacidad de devolver valores de tipo procedimiento como resultado de una invocación a un procedimiento.
- Embebimiento: la capacidad de colocar valores de tipo procedimiento dentro de estructuras de datos.

3.5.5.1 Abstracción procedimiental

Ya hemos introducido la abstracción procedimental. Cualquier declaración $< d >$ puede ser "empaquetada" dentro de un procedimiento como **proc** $ <d> **end**. Esto no ejecuta la declaración, pero en su lugar crea un valor de tipo procedimiento (una clausura).

Los valores de tipo procedimietno pueden tener argumentos, los cuales permiten que algo de su comportamiento sea influenciado por la invocación. La abstracción procedimental es enormememente poderosa. Ella subyace a la programación de alto orden y a la programación orientada a objetos, y es la principal herramienta para construir abstracciones.

3.5.5.2 Genericidad

Ya hemos visto un ejemplo de programación de alto orden en una sección 3.5.3. Fue con la abstracción de control *Iterar*, la cual usa dos argumentos de tipo procedimiento, *Transformar* y *Es_Final*. Hacer una función genérica es convertir una entidad específica (una operación o un valor) en el cuerpo de la función, en un argumento de la misma.

Considere la función *SumList*:

```
fun {SumList L}
  case L of nil then 0
  [] X|L1 then X+{SumList L1}
  end
end
```

Esta función tiene dos entidades específicas: el número cero (0) y la operación de adición (+). El cero es el elemento neutro de la operación de adición. Estas dos entidades puede ser abstraidas hacia afuera. Cualquier elemento neutro y cualquier operación son posibles. Los pasamos como parámetros. Esto lleva a la siguiente función genérica:

```
fun {FoldR L F U}
  case L of nil then U
  [] X|L1 then {F X {FoldR L1 F U}}
  end
end
```

Esta función asocia a la derecha. Podemos definir *SumList* como un caso especial de *FoldR*:

```
fun {SumList L}
  {FoldR L fun {$ X Y} X+Y end 0}
end
```

Podemos usar *FoldR* para definir otras funciones sobre listas. Por ejemplo, la función para calcular el producto de todos los elementos de una lista (se deja como ejercicio).

3.5.5.3 Instanciación

Un ejemplo de instanciación es una función *CrearOrdenamiento* que devuelve una función de ordenamiento. Funciones como éstas se suelen llamar "fábricas" o "generadoras". *CrearOrdenamiento* toma una función booleana de comparación F y devuelve una función de ordenamiento que utiliza a F como función de comparación. Por ejemplo:

```
fun {CrearOrdenamiento F}
  fun {$ L}
   {Ordenar L F}
  end
end
```

Podemos ver a *CrearOrdenamiento* como la especificación de un conjunto de posibles rutinas de ordenamiento. Invocar *CrearOrdenamiento* instancia la especificación, es decir, devuelve un elemento de ese conjunto, el cual decimos que es una instancia de la especificación.

3.5.5.4 Embebimiento

Los valores de tipo procedimiento se pueden colocar en estructuras de datos. Esto tiene bastantes usos:

- *Evaluación perezosa explícita.* La idea es no construir una estructura de datos en un solo paso, sino construirla por demanda. Se construye sólo una pequeña estructura de datos con procedimientos en los extremos que puedan ser invocados para producir más pedazos de la estructura. Por ejemplo, al consumidor de la estructura de datos se le entrega una pareja: una parte de la estructura de datos y una función nueva para calcular otra pareja. Esto significa que el consumidor puede controlar explícitamente qué cantidad de la estrcutura de datos se evalúa.

- *Módulos.* Un módulo es un registro que agrupa un conjunto de operaciones relacionadas.

- *Componentes* de Software. Un componente de software es un procedmiento genérico que recibe un conjunto de módulos como argumentos de entrada y devuelve un nuevo módulo.

3.5.5.5 Currificación

La currificación es una técnica que puede simplificar programas que usan intensamente la programación de alto orden. La idea consiste en escribir funciones de n argumentos como n funciones anidadas de un solo argumento. Por ejemplo, la función que calcula el máximo:

```
fun {Max X Y}
  if X>=Y then X else Y end
end
```

se reescribe así:

```
fun {Max X}
  fun {$ Y}
    if X>=Y then X else Y end
  end
end
```

Se conserva el mismo cuerpo de la función original, pero se invoca teniendo en cuenta la nueva definición: {{Max 10} 20}, devuelve 20. La ventaja de usar la currificación es que las funciones intermedias pueden ser útiles en sí mismas. Por ejemplo, la función {Max 10} devuelve un resultado que nunca es menor que 10. A la función {Max 10} se le llama función parcialmente aplicada. Incluso, podemos darle el nombre *CotaInferior10*:
$CotaInferior10 = \{Max 10\}$

En muchos lenguajes de programación funcional, particularmente en Standard ML y Haskell, todas las funciones están implícitamente currificadas.

3.6 Ejercicios

1. Mostrar a salida del siguiente programa:

```
local X in
   local Y in
      local Z in
       X = person(name:"George" age:Y)
           Z = 26
       Z = Y
       {Browse Y}
      end
   end
   {Browse X}
end
```

   Nota: el procedimiento *{Browse Arg}* muestra el valor asociado a *Arg*.

2. Ejecutar paso a paso el siguiente programa:

```
local X in
   local Y in
    X = person(name:"George" age:25)
    Y = person(name:"George" age:26)
    X = Y
   end
   {Browse Y}
end
```

3. Dado el siguiente programa, mostrar su ejecución paso a paso según la máquina abstracta definida.

```
local X in
   X = 1
   local P in
      P = proc {\$}
          {Browse X}
      end
      local X in
         X = 2
         {P}
      end
   end
end
```

4. Ejecutar el programa paso a paso según la máquina abstracta definida. Cual es la salida de {Browse X}?

```
local X Y in
   Y = 1
   local F P in
      F = proc {$ Y} {P Y} end
      P = proc {$ Z}  Z = Y end
      {F X}
      {Browse X}
    end
end
```

5. Una expresión es una abreviación sintáctica de una secuencia de operaciones que arrojan un valor. Suponiendo que extendemos la sintaxis de nuestro lenguaje para permitir expresiones en las operaciones de *binding* y en la condición de la sentencia `if`.

   A continuación se muestra un ejemplo, asumiendo que también se ha extendido la sentencia **local** para permitir la introducción de una lista de variables.

```
local X Y in
   X = 1
   Y = 2 + X
   if X > Y then
      {Browse X}
   else
      {Browse Y}
   end
end
```

Traducir el programa al lenguaje kernel.

6. Dado el siguiente algoritmo:

```
local Max A B C in
  fun {Max X Y}
     if X>=Y then X else Y end
  end
  A = 3
  B = 2
  {Browse {Max A B}}
end
```

   - Traducir el programa al lenguaje kernel.
   - Mostrar su ejecución paso a paso según la máquina abstracta definida.

7. Ejecutar el programa paso a paso según la máquina abstracta definida. Cual es la salida de {Browse A+Y}?

```
local X in
  X = 1
  local P in
    P = proc {$ Y}
          local P, A in
            P = proc {$ Z} Z=10 end
            {P A}
            {Browse A+Y}
          end
        end
    {P X}
  end
end
```

8. Mostrar que en el siguiente programa recursivo a la cola, el tamaño de la pila se mantiene limitada o acotada.

```
proc {Loop5 I}
   local C in
      C = I == 5
      if C then skip
      else
         local J in
            J = I + 5
            {Loop5 J}
         end
   end
end
```

Nota: ejecutar al menos dos invocaciones recursivas.

9. Convertir las siguientes funciones recursivas en funciones recursivas a la cola.

   (a)
   ```
   fun {Length Ls}
      case Ls
      of nil then 0
      [] _|Lr then 1+{Length Lr}
      end
   end
   ```

   (b) Definir en Haskell una versión recursiva a la cola de la siguiente función.

   ```
   ejemplo: [a]-->[a]
     | [] =[]
     | x:xs =  (ejemplo xs) ++ [map x]
   ```

   donde *map* es una función de

   ```
    a --> a
   ```

   (c) Definir en Haskell una versión recursiva a la cola de la siguiente función inversa de una lista.

   ```
   inversa: [a]-->[a]
     | [] =[]
     | x:xs = concat (inversa xs) [x]
   ```

   donde *concat* es la función que concatena 2 listas:

   ```
    [a] --> [a] --> [a]
   ```

10. GENERECIDAD y ABSTRACCIONES SINTáCTICAS. Diseñe un algoritmo (*FoldR*) que generalice, mediante la incorporación de parámetros, el comportamiento de las siguientes funciones.

    (a)
    ```
    fun {SumList L}
       case L
       of nil then 0
    ```

```
    [] X|L1 then X+{SumList L1}
  end
end

fun {ProdList L}
  case L
  of nil then 1
    [] X|L1 then X*{SumList L1}
  end
end
```

(b) QUE devuelve la siguiente función?

```
fun {Some L}
    {FoldR L fun {$ X Y} X orelse Y end false}
end
```

(c) Utilizar un esquema general de iteración para implementar un algoritmo que calcule la Raiz Cuadrada de un número real positivo, mediante el método de Newton. A continuación se detalla una posible implementación del método sin utilizar abstracción.

```
fun {Raiz X}
  fun {RaizIter Adiv}
    fun {Mejorar}
      (Adiv + X/Adiv) / 2.0
    end
    fun {Buena}
      {Abs X-Adiv*Adiv}/X < 0.00001
    end
  in
    if {Buena} then Adiv
    else
      {RaizIter {Mejorar}}
    end
  end
  Adiv=1.0
in
  {RaizIter Adiv}
end
```

11. De un ejemplo de una función perezosa y una aplicación interesante de ella. La sintaxis para definir una función perezosa es: *fun lazy* $\{FX_1...X_n\}$.

Ejercicios Adicionales

12. Definir un procedimiento que calcule el factorial de su argumento en el lenguaje kernel.

13. Traducir a una versión recursiva a la cola.

```
fun {Sum1 N}
    if N==0 then 0 else N+{Sum1 N-1} end
end

proc {Fact N ?R}
  if N==0 then R=1
  elseif N>0 then N1 R1 in
       N1=N-1
       {Fact N1 R1}
       R=N*R1
    else raise errorDeDominio end
  end
end
```

14. Suponiendo que *{Sum2 N X}* es la nueva versión recursiva a la cola de *Sum1*, que debería suceder en el sistema Mozart si las invocamos de la siguiente manera: {Sum1 100000000} y {Sum2 100000000 0}. Verificar.

15. El programa del ejercicio anterior muestra que el alcance (scope) es estático, es decir un procedimiento se evalúa en el ambiente de su definición.
    Modificar la semántica del lenguaje kernel para que tenga alcance dinámico, es decir, una invocación a un procedimiento se evalúa en el ambiente de su invocante.
    Ejecutar el programa con la nueva semántica para verificar que la salida del programa sería

16. *PATTERN MATCHING*. Desarrolle la operación INSERT(valor, TreeIn, ?TreeOut) que inserta un *valor* en un árbol binario de búsqueda. Considere la siguiente representación de arboles binarios:

```
tree(raiz TIzq TDer)
```

17. Definir la semántica de las declaraciones `try` y `catch`

Capítulo 4

Lenguajes funcionales

Como ya se mencionó en el capítulo anterior, el modelo declarativo comprende dos submodelos: la programación funcional y la programación lógica. En este capítulo se desarrollarán en mayor detalle los conceptos subyacentes a la programación funcional y en particular a las características y conceptos propios de los lenguajes funcionales.

En este capítulo se describen las características propias de la programación funcional, sus modelos de computación fundamentales y finalmente se describen algunas características de algunos lenguajes funcionales modernos como Haskell y ML.

4.1 Programación funcional

La programación funcional tiene sus orígenes con el desarrollo de LISP, desarrollado por John McCarty a finales de 1950. Si bien LISP no es un lenguaje funcional puro, tiene muchos conceptos utilizados en lenguajes funcionales modernos.

Los modelos abstractos de computación (o fundamentos teóricos) que pueden tomarse como la maquinaria necesaria para la implementación de lenguajes funcionales tienen sus raíces en el *cálculo lambda* y en la lógica combinatoria.

Lenguajes como Scheme y Dylan son derivados de LISP. En 1977, John Backus[1] publicó el memorable artículo *Can Programming Be Liberated From the Von Newmann Style?*, por el cual se le otorgó el premio Turing Award Lecture. En ese artículo Backus propone el lenguaje *FP* (*function-level programming language*), el cual introduce los principales conceptos de la programación funcional moderna.

La programación funcional tiene sus propias características y conceptos, que se describen a continuación.

[1] John Backus, entre otras cosas, desarrolló FORTRAN y fue el creador de la BNF (Backus Naur Form).

4.2 Características principales

Algunos conceptos son propios de la programación funcional y a menudo ausentes en lenguajes imperativos. Sin embargo, el estudio de estos conceptos son útiles para el programador aún cuando utilice lenguajes imperativos (y orientados a objetos) ya que muchos lenguajes imperativos modernos introducen algunos mecanismos para poder implementar algunas de las características propias de los lenguajes funcionales.

Algunos de los conceptos que caracterizan a la programación funcional son:

- **Funciones (o expresiones) puras**: no tienen estado propio (memoria) y no tienen *efectos colaterales*. Es decir que tienen las siguientes propiedades (algunas de las cuales pueden ser usadas por el intérprete o compilador para optimizar código):
  - Si el resultado de una expresión no se utiliza, puede descartarse en forma segura (no afectará a las otras funciones o expresiones.
  - Cada vez que se llame a una función con los mismos argumentos, el resultado será el mismo. Esto se conoce como *transparencia referencial*.
  - No existe dependencia de datos en las funciones puras, lo que dos o más funciones (que no dependan entre ellas) pueden ser evaluadas concurrentemente sin ninguna necesidad de sincronización.
  - Se puede utilizar cualquier orden o estrategia de evaluación.
- *Recursión*: A diferencia de los lenguajes imperativos, la iteración se consigue por medio de recursión, es decir, una función puede invocarse a sí misma. La recursión puede requerir espacio en la pila de ejecución, aunque como ya se vio anteriormente, una función (o procedimiento) recursivo a la cola (tail recursion) puede implementarse como una iteración (while) en un lenguaje imperativo.
- *Funciones de alto orden*: Una función es de alto orden cuando puede tomar funciones como argumentos o retornar funciones como resultado. Las funciones de alto orden son naturales en la programación funcional y son la base de la técnica de programación conocida como *programación genérica*.
- *Orden de evaluación*: En general se pueden utilizar dos estrategias de evaluación de las expresiones: *estricta o ansiosa (eager)* y *no estricta (normal)*.

A modo de ejemplo del estilo de la programación funcional se muestra el problema de calcular el producto interior de dos vectores a y b de dimensión N.

Al estilo imperativo, se podrá escribir un fragmento de programa de la forma:

```
...
p := 0;
for i := 1 to N do
    p := p + a[i] * b[i]
```

El resultado quedará en la variable p.

Una solución al estilo funcional podría escribirse como (al estilo de FP):[2]
$InnerProduct \equiv (Insert+) \circ (ApplyToAll\times) \circ Transpose$
La definición anterior dice que el producto entre dos vectores representados de la forma $<< a_1, \ldots, a_n >, < b_1, \ldots, b_n >>$ (una matriz de dos filas, representada como una lista de listas) puede ser evaluada mediante la composición de tres pasos:

1. Obtener la transpuesta de matriz de entrada.
2. Aplicar el operador $\times$ a cada elemento (pares) de la lista obtenida.
3. Aplicar el operador $+$ (recursivamente a los elementos de la lista obtenida).

Un ejemplo de evaluación de la función $InnerProduct$ aplicada al argumento $<< 1, 2, 3 >, < 6, 5, 4 >>$ sería:

1. La aplicación de *Transpose* retorna $<< 1, 6 >, < 2, 5 >, < 3, 4 >>$.
2. La aplicación de $ApplyToAll\times$ retorna $< \times :< 1, 6 >, \times :< 2, 5 >, \times :< 3, 4 >>$, cuya evaluación arrojaría el valor $< 6, 10, 12 >$
3. La aplicación de *Insert* $+$ a $< 6, 10, 12 >$, retorna $+ :< 6, + :< 10, 12 >>$, cuya evaluación retornaría $+ :< 6, 22 >$ y la evaluación final sería el valor 28.

4.3 Ventajas y desventajas con respecto a la programación imperativa

De un análisis inmediato del programa imperativo, podemos hacer las siguientes observaciones:

- Sus sentencias operan sobre un estado *poco visible*.
- No es modular (jerárquico), salvo por la parte derecha de la asignación (expresión). Es decir, no construye entidades desde otras mas simples (lo cual sucede generalmente con grandes programas).
- Es dinámico y repetitivo. Se debe ejecutar mentalmente para comprender qué hace.
- Computa un valor a la vez en cada repetición (mediante la asignación) y por modificación (de la variable i).
- Funciona sólo para vectores de longitud N.

[2] Extraido del artículo *Can Programming Be Liberated From the Von Newmann Style?* de John Backus, disponible en http://www.stanford.edu/class/cs242/readings/backus.pdf

- Sus argumentos están nombrados. Si se desea mayor generalidad (reusabilidad) se deberá definir una abstracción (procedimiento o función) parametrizada, lo cual requiere de otros mecanismos complejos.

- Las operaciones básicas están dispersas por todo el programa (sentencia **for**, ::=, ...).

El programa funcional tiene las siguientes propiedades:

- Opera sólo sobre sus argumentos.

- Es jerárquico. La función se define en términos de otras.

- Es estática y no repetitiva.

- Opera sobre valores conceptuales, no en pequeños pasos.

- Es completamente general (opera sobre dos vectores cualesquiera).

- No nombra a sus argumentos.

Si bien del análisis anterior es posible ver las ventajas de la programación funcional sobre la imperativa, es justo nombrar algunos problemas en los cuales la noción de estado es importante y natural para su solución.

- El modelado de la entrada-salida es natural si se piensa en términos de estado.

- Algunos sistemas (ej: sistemas reactivos) requieren estado. Ejemplos: sistemas operativos, sistemas de control, etc.

- Sistemas cuya información no se computa sino que se mantiene (almacena) en memoria y es modificada o recuperada por algunas operaciones (ej: colecciones, cuentas bancarias, etc).

Es posible modelar estado en la programación funcional utilizando *mónadas*. En la sección 4.6.2 se hace una breve introducción a mónadas.

4.4 Fundamentos teóricos

Los fundamentos o teoría subyacente de la programación funcional tiene sus raíces en el cálculo lambda, el cual ha sido utilizado no sólo para dar semántica de lenguajes funcionales sino también en algunos fragmentos de lenguajes imperativos y orientados a objetos.

4.4.1 Cálculo lambda

El cálculo lambda ($\lambda - calculus$), desarrollado por Alonzo Church y Stephen Cole Kleen en 1930, es un sistema formal diseñado para investigar las definiciones de funciones y sus aplicaciones.

El cálculo lambda se puede ver como un lenguaje minimalista de programación, sobre el cual se define un modelo de computación, el cual tiene el mismo poder computacional de otros lenguajes de programación (turing computable).

Además de sus aplicaciones en ciencias de la computación tiene un rol fundamental en el área de fundamentos de la matemática, gracias a la correspondencia Curry-Howard.[3]

En esta sección se analizará el cálculo lambda no tipado. Muchas de sus aplicaciones utilizan una de sus extensiones conocida como cálculo lambda tipado.

Informalmente, en el cálculo lambda cada expresión es una función unaria. Es decir que cada función toma un argumento y retorna un valor.

Definición 4.4.1 *Una expresión en el cálculo lambda se puede definir inductivamente como:*

1. *Una variable (denotadas como $x, y, \ldots$) es una expresión lambda.*
2. *Abstracción lambda: si x es una variable y M es una expresión lambda, entonces también lo es $(\lambda x.M)$.*
3. *Aplicación: si M, N son expresiones lambda, entonces también lo es $(M\ N)$.*

Para evitar el uso excesivo de paréntesis, comúnmente se aplican las siguientes convenciones:

- Los paréntesis exteriores se omiten, esto es, $M\ N$ en lugar de $(M\ N)$.
- La aplicación se asume asociativa a izquierda. Esto es, $M\,N\,P$ significa $(M\,N)\,P$.
- El cuerpo de una abstracción se extiende todo lo posible a la derecha: λxMN significa (λxMN) y no $(\lambda xM)N$.
- Una secuencia de abstracciones se pueden contraer: $\lambda x \lambda y.N$ se puede abreviar como $\lambda xy.N$.

Definición 4.4.2 *El conjunto de variables libres de una expresión lambda M, denotada como $FV(M)$, se define recursivamente como:*

- $FV(x) = \{x\}$ *(x es una variable).*
- $FV(\lambda x.M) = FV(M) - \{x\}$.

[3] Relación directa entre programas y desmostraciones matemáticas.

- $FV(MN) = FV(M) \cup FV(N)$.

Una variable no libre en una expresión M se dice que está *ligada*.

Definición 4.4.3 *Una expresión M con $FV(M) = \emptyset$ se denomina cerrada, también conocidos como combiadores.*

4.4.1.1 Reducción

La relación de reducción define los pasos de computación que se pueden aplicar a una expresión lambda.

- **Conversión** *alpha*: permite renombrar a las variables ligadas. Cabe aclarar que la aplicación en una expresión de esta regla puede cambiar su significado. Por ejemplo, si en la expresión $\lambda x \lambda y.x$ se reemplaza x por y, se obtiene $\lambda y \lambda y.y$.

- **Reducción** *beta* (aplicación de función): $((\lambda x.E_1)E_2$ es $E_1[x \leftarrow E_2]$, donde $E_1[x \leftarrow E_2]$ significa el reemplazo simultáneo de E_2 por cada variable libre x en la expresión E_1. Esto se denomina *sustitución*.

- **Conversión** *eta* (extensionalidad): dos funciones son iguales si dan los mismos resultados en cada uno de sus posibles argumentos. Por ejemplo, $\lambda x.fx$ puede convertir a f, siempre que x no aparezca libre en f.

  Esta conversión no es siempre conveniente cuando las expresiones lambda son interpretadas como programas (la evaluación de $\lambda x.fx$ puede terminar, aún cuando la evaluación de f no).

4.4.1.2 Computación y cálculo lambda

A continuación se mostrará que es posible definir valores y funciones básicas lo que muestra que el cálculo lambda tiene la misma expresividad que cualquier lenguaje de programación.

- **Numerales de Church y aritmética**:

  $0 \equiv \lambda fx.x$
  $1 \equiv \lambda fx.fx$
  $2 \equiv \lambda fx.f(fx)$
  ...

  El numeral de Church n es una función que toma una función f como argumento y retorna la enésima composición de f (f compuesta consigo misma n veces).

  Función sucesor: $SUCC \equiv \lambda n\ f\ x.f(n\ f\ x)$
  Suma: $PLUS \equiv \lambda m\ n\ f\ x.n\ f(m\ f\ x)$ o $PLUS \equiv \lambda m\ n.m\ SUCC\ n$.
  Multiplicación: $MULT \equiv \lambda m\ n\ f.m(n\ f)$ o $PLUS \equiv \lambda m\ n.m(PLUS\ n)0$

La función predecesor ($PREDn = n - 1$ y $PRED0 = 0$) es mas dificultosa para describir. La siguiente definición se puede validar por inducción: sea $T = \lambda g\, h.h(g\, f)$, entonces $T^n(\lambda ux) = (\lambda h.h(f^{n-1}(x)))$, para $n > 0$.

$PRED \equiv \lambda n\, f\, x.n(\lambda g\, h.h(g\, f))(\lambda u.x)(\lambda u.u)$

Mas abajo de define $PRED$ de dos maneras alternativas adicionales usando condicionales y pares.

- **Operadores lógicos**

  $FALSE \equiv \lambda t\, f.f$ (notar que es igual al cero).
  $TRUE \equiv \lambda t\, f.t$
  $AND \equiv \lambda p\, q.p\, q\, p$
  $OR \equiv \lambda p\, q.p\, p\, q$
  $NOT \equiv \lambda pab.pba$

- **Predicados** (funciones que retornan valores lógicos):

  $ISZERO \equiv \lambda n.n(\lambda x.FALSE)TRUE$
  $LEQ \equiv \lambda mn.ISZERO(SUBmn)$ (donde $SUBmn$ es $m - n$)

  Es posible redefnir $PRED$ usando predicados:

  $PRED \equiv \lambda n.n(\lambda g\, k.ISZERO(g\, 1)k\, PLUS(g\, k)1)(\lambda v.0)0$

- **Pares**: (f representaría el *constructor* del par)

  $PAIR \equiv \lambda xyf.f\, x\, y$
  $FST \equiv \lambda p.pTRUE$
  $SND \equiv \lambda p.pFALSE$

  También es posible redefinir $PRED$ usando pares:

  $PRED \equiv \lambda n.FIRST(n\, SHIFTANDINC(PAIR\, 0\, 0)$, donde

  $SHIFTANDINC \equiv \lambda x.PAIR(SECOND\, x)(SUCC(SECOND\, x))$

  $(SHIFTANDINCPAIR(m\, n) = PAIR(n\, n + 1))$

- **Recursión**

  El combinador

  $Y \equiv \lambda g.(\lambda x.g(xx))(\lambda x.g(xx))$

  y se conoce como el *operador de punto fijo* ya que Yg es un punto fijo de g (expande a $g(Yg)$).

  Por ejemplo, sea $FACT \equiv \lambda fn.(1, ifn = 0 and n.f(n.1), ifn > 0)$, si se aplica $FACT\, (Y\, FACT)\, n$, es posible computar el factorial de n.

El concepto de *computabilidad* se puede definir de la siguiente forma:

Definición 4.4.4 *Una función $F : \mathbb{N} \to \mathbb{N}$ es computable si y sólo si existe una expresión lambda tal que para cada par de $x, y \in \mathbb{N}$, $F(x) = y$ si y sólo si $f\ a = b$, donde a y b son los numerales de Church correspondientes a x e y, respectivamente.*

No existe un algoritmo (una función computable) que tome dos expresiones lambda y retorne $TRUE$ o $FALSE$ en base si las expresiones son equivalentes. Este resultado se conoce como la *tesis de Church*.

Muchos lenguajes de programación permiten definir funciones lambda (python, ...) las cuales pueden pasarse como argumentos a otras funciones (o procedimientos). Otros lenguajes permiten implementar alto orden usando punteros o referencias (ej: C, C++). Algunos lenguajes orientados a objetos permiten definir objetos que representan funciones, con lo cual se logra alto orden (ej: C++, Smalltalk, Ruby, Eiffel).

Algunos mecanismos de pasajes de parámetros en lenguajes imperativos permiten implementar alto orden (ej: pasaje por nombre y las referencias o punteros a funciones).

4.4.1.3 Estrategias de reducción

En esta sección se analizan diferentes estrategias de aplicación de las reglas de reducción. Las estrategias se conocen también como *órdenes de evaluación*.

Definición 4.4.5 *Una expresión que no puede ser reducida se denomina* forma normal.

Definición 4.4.6 *Un **redex** (expresion reducible) es o un $\beta - redex$ o un $\eta - redex$.*

- *Una expresión E es un $\beta - redex$ si E tiene la forma $((\lambda x.M)\ N)$.*
- *Una expresión E es un $\eta - redex$ si E tiene la forma $\lambda x.M\ x$ y x no está libre en M.*

*La expresión a la cual se reduce por aplicación de una regla se denomina **reducto**. Los reductos de los redexes son $M[x \to N]$ y M, respectivamente.*

A continuación se describen las diferentes estrategias posibles:

- **Full beta reduction**: cualquier redex puede reducirse en cada paso.
- **Orden aplicativo**: en cada paso se elige el redex más interno y más a la derecha. Intituivamente, esto significa que los argumentos siempre se reducen antes que la aplicación de la función, lo que implica que las funciones siempre operan sobre formas normales.

  Muchos lenguajes de programación utilizan esta estrategia de evaluación (C, C++, Java, ML, LISP, ...). Esta estrategia se conocen también como *estricta*.
- **Orden normal**: en cada paso se elige el redex de más a la izquierda y más externo. Los argumentos se sustituyen dentro del cuerpo de una abstracción antes que sean reducidos.

- **Por nombre (call-by-name)**: igual que con orden normal, pero no se realizan reducciones dentro de las abstraciones.

  Por ejemplo, la expresión $\lambda x.(\lambda x.x)x$, ya está en forma nomal usando esta estrategia, aunque contiene el redex $(\lambda x.x)x$

- **Por valor (call-by-value)**: en cada paso se reduce el redex más externo. Esto implica que un redex puede ser reducido después que su parte derecha (right hand side) se ha reducido a un valor (variable o abstracción lambda).

- **Por necesidad (call-by-need)**: igual que en orden normal, pero aquellas aplicaciones que podrían generar términos duplicados en lugar de nombres de argumentos, se reducen sólo cuando sean necesitados. Esta estrategia se conoce generalmente como *evaluación lazy* y en la práctica comúnmente se implementa usando punteros (el cual representa el *nombre*) y el redex se representa con una estructura de datos (usualmente un árbol o un DAG).

El orden aplicativo no es una estrategia normalizante, es decir que no siempre permite reducir a formas normales. Por ejemplo, la expresión $E_1 = (\lambda x.xx\ \lambda x.xx)$, contiene sólo un redex (la expresión completa) y su reducto es E_1. Esto significa que E no tiene forma normal (bajo ninguna estrategia). La expresión $E_2 = (\lambda xy.x)(\lambda x.x)E_1$, usando orden aplicativo, se debe aplicar en E_1 primero, pero como E_1 no tiene forma normal, el orden aplicativo falla para encontrar una forma nomrmal de E_2.

Con orden normal siempre es posible reducir a una forma normal.[4] En el ejemplo anterior, utilizando orden normal, E_2 reduce a $I = \lambda x.x$.

Una de las ventajas del orden aplicativo sobre el orden normal, es que el primero no produce computaciones innecesarias si se utilizan todos los argumentos en las abstraciones, porque nunca sustituye argumentos conteniendo redexes (porque ya han sido reducidos previamente).

4.5 LISP

LISP (LISt Processing)fue inventado por John McCarty en 1958 en el MIT. La sintaxis del lenguaje se basa en la noción de *S-expresiones* (symbolic expressions), como por ejemplo *(car(cons A B))*.[5]

El lenguaje tiene una sintaxis sorprendentemente simple, tipado dinámico y manejo de la memoria automático ya que incluye un recolector de basura (garbage collector). LISP generalmente es interpretado y tiene varios lenguajes derivados como Common-LISP y Sheme.

[4] De allí el nombre de la estrategia.
[5] Lo cual evalúa a A, la cabeza (*car*) de la lista [a,b] (*cons* es el constructor de listas.

4.5.1 Sintaxis

LISP es un lenguaje orientado a expresiones. La evaluación de una expresión arroja un valor (o lista de valores). Los valores pueden ser básicos (ej: números, caracteres y átomos) o estructurados (listas).

Los tipos de datos en LISP pueden ser dos: átomos o listas.

Un átomo es un identificador (símbolo) o una constante numérica o caracter. Las expresiones se escriben como listas en forma prefija.

Una lista en LISP es una lista simplemente enlazada. Cada celda de una lista se denomina un `cons` el cual tiene dos componentes: la cabeza (`car`) y el resto (`cdr`).[6]

Por ejemplo, la función *list* retorna sus argumentos como una lista, así la expresión `(list '1 'foo)` retorna la lista `(1 foo)`. El apóstrofe (quote) que precede los argumentos es un operador especial que previene su evalación.

La expresión predefinida `(if c` E_1 E_2`)` retorna E_1 si `c` evalúa a `nil` (un átomo que representa la lista vacía), sino retorna E_2.

Las *s-expresiones* representan listas.

LISP soporta la definición de `expresiones lambda`, es decir funciones anónimas. La forma de una definición `lambda` es `(lambda args body)`, donde `args` es la lista de argumentos y `body` es la lista que representa la definición de la función.

A modo de ejemplo, la expresión `(lambda (n) (+ n 1))` evalúa a una función que, cuando sea aplicada, evaluará al sucesor del argumento dado.

4.5.2 Semántica

LISP evalúa las expresiones a menos que estén precedidas por el operador `quote` ('). La evaluación finaliza cuando una expresión es un valor básico (número, átomo o lista que evalúa a sí misma). Cada elemento de una lista que esté ligado a una definición de una función podrá ser evaluada, sino es considerada un valor (evalúa a sí mismo).

El orden de evaluación utilizado por LISP es estricto, específicamente orden aplicativo.

El intérprete de LISP (función `eval`) implementa un ciclo de la forma *leer, evaluar, mostrar*, es decir que se lee una expresión, se evalúa y muestra el resultado.

La función `defun` define una nueva función en una sesión la cual puede ser invocada posteriormente. A continuación se muestra la definición de la función factorial.

```
(defun factorial (n)
     (if (<= n 1) 1 (* n (factorial (- n 1))))
)
```

[6] Los nombres `car` y `cdr` se deben a *contents of address register* y *contents of decrement register*, respectivamente y tienen relación directa a la primera implementación de LISP en la IBM 704.

Ahora es posible evaluar el factorial de 5 como: `(factorial 5)`.

El alcance (scope o clausuras) en LISP original era dinámico. Algunos lenguajes derivados de LISP como CommonLisp o Sheme tienen alcance estático.

Otras implementaciones, como los dialectos de LISP que soportan aplicaciones como Emacs o AutoCAD mantienen el alcance dinámico.

4.5.3 Estado

LISP no es un lenguaje funcional puro ya que permite compartir y modificar estructuras definidas.

Por ejemplo, la función `setf`, permite ligar (bind) una referencia a un valor.

Si se hace `(setf s (cons 'a 'b))` y luego `(setf (car s) 1)`, al evaluar `s` arroja el valor $(1.b)$.[7]

4.5.4 Aplicaciones

El lenguaje LISP y sus derivados se utilizan ampliamente en campos de inteligancia artificial, como procesamiento simbólica, representación de conocimiento o otros. Además, la simplicidad de su implementación lo hace adecuado como lenguaje de scripting, los cuales generalmente algunas aplicaciones incorporan para permitir realizar extensiones (plugins), como Emacs y AutoCAD.

4.6 Lenguajes funcionales modernos

Los lenguajes funcionales han evolucionado principalmente en sus sistemas de tipos, con polimorfismo paramétrico, inferencia de tipos, tipos de datos algebraicos, definiciones por patrones (pattern matching), listas por comprensión y manejo de excepciones, entre otras características.

En esta sección de analizan dos lenguajes funcionales modernos ampliamente utilizados. El primero (ML) no es un lenguaje funcional puro, ya que permite definir y manipular estado, es decir, variables mutables. El segundo (Haskell) es un lenguaje funcional puro con características propias como evaluación perezosa (lazy).

4.6.1 ML

ML fue desarrollado por Robin Milner y otros en los comienzos de la década de 1970 en la Universidad de Edimburgo. Uno de los principales objetivos fue la implementación de tácticas de prueba para demostradores de teoremas.

Su nombre proviene de *metalenguaje*. ML permite introducir *efectos colaterales*, lo que lo hace un lenguaje funcional impuro, por lo que se considera un lenguaje funcional

[7] La notación `(a . b)` es equivalente a `(cons 'a 'b)`.

con características imperativas.

ML utiliza orden de evaluación ansiosa (eager), en particular la estrategia denominada *call by value* y tiene alcance (scope) estático. Actualmente existen varios derivados de ML, como Standard ML (SML) y Ocaml.

A continuación se muestran algunos ejemplos en Ocaml.

```
let average a b = (a +. b) /. 2.0 ;;
```

En este primer ejemplo se define una función que calcula el valor medio entre dos valores dados. La sentencia *let* introduce un nuevo indentificador dentro del ambiente en que se aplica.
Esta función tiene tipo

```
val average : float -> float -> float = <fun>
```

lo que indica que *average* es una función que toma dos argumentos *float* y retorna un *float*. Los operadores (funciones) $+.$ y $/.$ operan sobre *floats*.

La siguiente función muestra que es posible definir *procedimientos* (o funciones que retornan *void* en C, C++ o Java).

```
let sum a b =
              let s = a + b in
                  Print.printf "Sum=%d\n" s ;;

val sum : int -> int -> unit = <fun>
```

Cabe notar que no es obligatorio denotar los tipos en las definiciones. ML infiere los tipos de las expresiones desde sus argumentos y operadores. Esto hace que en ML los operadores no estén sobrecargados, es decir que cada operador (o función) debe operar sobre tipos concretos.

Es posible definir funciones recursivos.

```
let rec factorial n =
                      if (n < 0) then (raise Exit) else
                         match n with
                              0 -> 1
                            | n -> (n * (factorial (n-1)));;
```

La función *factorial* muestra que es posible disparar excepciones (los cuales son valores de un tipo específico) y reconocimiento de patrones (pattern matching).
La definición de funciones recursivas deben denotarse explícitamente por las reglas de alcance (scope) del lenguaje. Si se omitiese, la invocación de *factorial* en el cuerpo de su definición trataría de resolver a un símbolo definido previamente, mas que el símbolo que se está definiendo. *factorial*.

Los tipos básicos de Ocaml son: int, bool, char, string, unit.

4.6.1.1 Tipos de datos estructurados

En Ocaml es posible definir nuevos tipos de datos a partir de otros tipos definidos previamente. La sentencia `type` introduce un nuevo nombre de tipo.

- **tuplas**: un valor de una tupla se define de la forma `(`v_1`,...,`v_n`)`, donde v_i, $1 \leq i \leq n$ son valores.

  Ejemplo de definición de un tipo basado en tuplas:

  ```
  type pair = int * int
  ```

- **registros (records)**: son como las tuplas, pero con sus elementos rotulados. Son equivalentes a las estructuras (struct) de C o registros de Pascal.

  Ejemplo:

  ```
  type person = {name:string ; id:int};;
  ...
  let p = {name="Haskell Curry", id=1};;
  p.name;;
  ...
  ```

- **Variantes (uniones) y enumeraciones**: los registros variantes o uniones (como **union** de C) y las enumeraciones (tipos enumerados de Pascal, **enum** de C, etc), se definen de manera concisa y elegante como se muestra el siguiente ejemplo:

  ```
  type DiaSemana = Dom | Lun | Mar | Mie | Jue | Vie | Sab;;
  ```

  Los tipos variantes pueden usarse para definir estructuras de datos como en el siguiente ejemplo:

  ```
  type bin_tree = Leaf of int | Tree of bin_tree * bin_tree;;
  ```

  Ejemplos de expresiones (valores) de tipo *bin_tree* son:

  ```
  Leaf 1
  Tree (Leaf 3, Leaf 4)
  Tree (Tree (Leaf 3, Leaf 4), Leaf 5)
  ```

- **Listas**: las listas son las estructuras de datos mas comunmente utilizadas en la programación funcional. Las listas en ML son homogéneas, es decir que todos sus elementos deben ser del mismo tipo.

  Ejemplos de listas:

  ```
  [1;2;3]
  []       (lista vac\'ia)
  1 :: [2;3]
  1 :: 2 :: 3 :: []
  ```

El operador *::* (cons) construye una lista tomando como su primer argumento la cabeza (head) y su segundo argumento el resto o cola (la cual es una lista).

Las notaciones de listas usando corchetes son un ejemplo de *syntactic sugar*.

Las listas son un ejemplo de un tipo de datos parametrizado.

Un tipo de datos (o función) parametrizado (con parámetros de tipo) define una familia de tipos (o funciones), llamados comúnmente *politipos*.

El tipo *list* se puede definir de la siguiente manera:

```
type 'a list = Nil | :: of 'a * 'a list;;
```

La evaluación de la expresión `1 :: Nil` resulta en

```
- : int list = :: (1,Nil)
```

o sea una lista de enteros, en particular la lista `[1]`.

Es posible definir la función `length` (longitud de una lista)

```
let rec length a' l =
    match l with
        Nil       -> 0
      | _ :: rest -> 1 + length rest;;
```

4.6.1.2 Referencias (variables)

El siguiente programa

```
let my_ref = ref 0;;
ref := 100;;
let value = !my_ref;;
```

define una referencia a una celda en memoria que inicialmente contiene el valor 0. Es posible acceder al valor de una variable v por medio de la expresión `!v`. El operador `:=` permite modificar (asignar) un nuevo valor a una referencia.

Lo anterior muestra que Ocaml permite el estilo de programación imperativa, por lo que no es un lenguaje funcional puro.

4.6.1.3 Otras características imperativas

La familia de lenguajes derivados de ML generalmente comparten otras características imperativas como arreglos y sentencias de iteración como *while* y *for*.

La *denotación* de valores de tipo `array` (en realidad define vectores o arreglos unidimensionales) tienen la forma

$[|e_0; \ldots; e_n|]$

El siguiente ejemplo muestra el uso de arreglos con una sentencia de control *for*.

```
let a = Array.create 10 0;; (* create a array of 10 elements, initialized with 0 *)
 for i = 0 to Array.length a - 1 do
    a.(i) <- i
  done;;
```

El tipo `array` es un ejemplo de un tipo de dato `mutable`.

Un campo de un registro puede declararse como `mutable`. Por ejemplo, el tipo *ref* está definido como un registro con un único campo mutable:

```
type 'a ref = { mutable contents:'a };;
```

4.6.2 Haskell

Haskell es un lenguaje funcional puro, con evaluación lazy, alto orden y pattern matching. Tiene un avanzado sistema de tipos que permite polimorfismo paramétrico y sobrecarga de constructores de clases.

De igual forma que ML, Haskell es un lenguaje basado en expresiones, cuya evaluación arroja *valores*, de un determinado tipo, como resultado. Los tipos de datos básicos son `Char`, `Integer`, `Float`, entre otros. Los tipos de datos básicos estructurados son las listas (ej: `[1,2,3]`) y las tuplas (ej: `('b',4)`).

Los tipos básicos no difieren de los tipos definibles por el programador, ya que es posible definirlos en Haskell, pero el lenguaje usa notaciones convenientes para esos valores. Por ejemplo, el tipo `Char` puede definir como una (gran) enumeración compuesta de constructores constantes.

Si bien los tipos básicos no difieren de los demás tipos, Haskell provee algunas construcciones sintácticas que favorecen la legibilidad de los programas. Por ejemplo pueden definirse listas por comprensión como en los siguientes ejemplos:

```
 [ f x | x <- l ] -- lista de f(x) tal que x recorre
la lista l [ (x,y) | x <- l1, y <- l2] -- producto cartesiano de l1
con l2 [ x | x <- l, x > 0 ] -- lista de valores positivos de la lista
l
```

En el último ejemplo se puede ver que en una lista por comprensión pueden incluirse condiciones, llamadas *guardas*.

Otra conveniencia sintáctica son las cadenas (strings), las cuales se pueden denotar como `"Hola"` cuando en realidad es una lista de caracteres (`['h','o','l','a']`).

Las funciones se definen en forma similar a ML, es decir que consisten en un conjunto de ecuaciones. Por ejemplo la función `quicksort` puede definirse como

```
quicksort []    = []
quicksort (x:xs)= quicksort [y | y <- xs, y<x]
```

```
                ++ [x] ++
                quicksort [y | y <- xs, y<=x]
```

El operador : es el constructor de listas (similar al operador cons de LISP) y el operador ++ es la concatenación de listas.

La notación `[1..n]` son abreviaturas sintácticas para denotar secuencias aritméticas. Ejemplos:

```
 [1..5] => [1,2,3,4,5] [5,2..4] => [5,2,3,4] [1,3..]
=> [1,3,5,7,9,...] (lista infinita)
```

Mas adelante se tratarán en detalle las listas infinitas.

Las funciones son currificadas, es decir que toman un solo argumento, permitiendo definir funciones parciales. A modo de ejemplo, la función

```
add x y = x + y
```

define una función con el tipo `add ::  Integer -> Integer -> Integer`.

La función `inc = add 1` tiene tipo `inc ::  Integer -> Integer`, o sea que su valor de retorno es una función. Cuando se aplica a un argumento entero se obtendrá un valor concreto. Por ejemplo, `inc 4` retorna 5.

Si se desea descurrificar uan función se pueden agrupar los argumentos en una tupla. Por ejemplo, `add (x,y) = x + y` es una función de un único argumento.

La aplicación de funciones es asociativa a izquierda.

En Haskell también es posible definir funciones lambda, como por ejemplo

```
inc = \x -> x + 1
```

Los operadores infijos se pueden definir por medio de ecuaciones como cualquier otra función. Por ejemplo, el operador de concatenación de listas está definido como

```
 [] ++ ys = ys (x:xs) ++ ys = x : (xs ++ ys)
```

Para cualquier operador infijo, es posible definir su precedencia (del 0 al 9, donde 9 es la precedencia mas alta) y asociatividad. Por ejemplo, el operador ++ se define asociativo a derecha con precedencia 5.

```
infixr 5 ++
```

Los operadores infijos asociativos a izquierda se definen con `infixl` y los no asociativos con `infix`.

4.6.2.1 Tipos

En Haskell se define un nuevo tipo mediante la palabra reservada `data`.

```
data Bool = False | True
data Complex = C Float Float   -- tupla (par) de dos reales
```

Haskell también permite definiciones polimórficas.

```
data Pair a = Pr a a
data BinTree a = Leaf a | Branch (Tree a) (Tree a)
```

Es posible definir sinónimos (alias) de tipos, usando `type`.

```
type String = [Char]
type Name = String
type Address = String
type Person = (Name,Address)
```

También es posible definir tuplas con campos rotulados o *registros*.

```
data Point = Pt { coordx, coordy ::Float }
```

Los nombres de los campos pueden usarse como selectores de componentes de un registro:

```
coordx :: Point -> Float }
coordy :: Point -> Float }
```

4.6.2.2 Casos y patrones

Como ya se vio anteriormente, en las definiciones de funciones se usan patrones para poder hacer definiciones por casos. En los patrones se pueden usar variables *mudas (_)* (también llamados comodines o wildcards), cuando una determinada parte de un patrón no se necesita en la parte derecha de la ecuación.

```
head (x:_) = x
tail (_:xs)= xs
```

Existen patrones *irrefutables*, como por ejemplo, las variables y otros patrones *refutables*.

El algoritmo utilizado de comparación de patrones (pattern matching) puede tener éxito, puede fallar o puede divergir.

Cuando la comparación de dos patrones tiene éxito (unifican) se producen ligaduras (bindings) entre variables y valores. Dos patrones pueden no unificar, como por ejemplo `[1,_],[2,x]`, ya que dos constantes diferentes no unifican.

Dos patrones pueden diverger, es decir, producir el valor `bot` ($\perp$), como por ejemplo `[1,2]` con `[bot,0]`.[8]

Los patrones pueden tener una *guarda*. Por ejemplo

[8] La compraración de `bot` con 0 diverge.

```
sign x | x > 0 = 1
sign x | x == 0 = 0
sign x | x < 0 = -1
```

En Haskell los patrones se procesan de arriba hacia abajo y de derecha a izquierda.

Las expresiones `case` permiten reconocer patrones dentro de expresiones. Una expresión `case` tiene la forma

```
case (x1 ... xk) of
   (p11 ... p1k) -> e1
   ...
   (pn1 ... pnk) -> en
```

Por ejemplo, las expresiones condicionales `if c then e1 else e2` es equivalente a

```
case (c) of
   True -> e1
   Else -> e2
```

4.6.2.3 Evaluación perezosa y sus consecuencias

Las funciones en Haskell no son estrictas. Es decir que la función

```
bot = bot
```

no termina. Esto quiere decir que si se define una función constante de la forma

```
uno x = 1
```

la aplicación de `uno bot` retorna 1, ya que la función `uno` no usa su argumento y por lo tanto no es necesario evaluarlo.

Esto se debe a que Haskell utiliza evaluación peresoza (lazy), la cual es un método de evaluación *call by need*.

Una de las ventajas del uso de esta estrategia de evaluación es que, entre otras cosas, permite, por ejemplo, la definición de estructuras de datos infinitas como en los siguientes ejemplos.

```
ones = 1 : ones
from n = n : from (n+1)
squares = map (^2) (from 0)
```

Como desventaja de la evaluación lazy se puede mencionar el consumo de memoria adicional requerido, ya que es necesario mantener estructuras de datos aún no evaluadas (reducidas) en memoria.

4.6.2.4 Ambientes

A menudo es deseable generar un ámbito local en una expresión o definición. Esto permite hacer definiciones que ocultan detalles de implementación.

- Las expresiones **let** son útiles en casos que se definan múltiples declaraciones. Por ejemplo:

```
let y = a * b
    f x = (x+y)/y
in f c + f d
```

- La cláusula **where** permite realizar varias definiciones en una ecuación con varias guardas. Por ejemplo:

```
f x y | y > z  = ...
      | y <= z = ...
where z = x * x
```

- Expresiones *lambda* (abstracciones):

```
\x -> E
```

  donde E es una expresión es equivalente a $\lambda x.E$

4.6.2.5 Clases y sobrecarga de operadores

Las clases permiten definir operaciones comunes a una determinada familia de tipos de datos.

Por ejemplo, la clase `Eq` definida en preludio estándar[9] define las operación de igualdad y desigualdad para que pueda ser instanciada por un conjunto de tipos concretos.

```
class Eq a where
   (==), (/=) :: a -> a -> Bool
   x == y = not (x /= y)
   x /= y = not (x == y)
```

Las definiciones de las operaciones `==` y `/=` en la clase `Eq` son operaciones por omisión (default).

En la definición de tipos de datos concretos, las operación de igualdad podrá ser una instancia de la clase `Eq`:

```
instance Eq Integer where
   x == y = integerEq x y


...
instance Eq Float where
   x == y = floatEq x y
```

[9] El preludio estándar es como la blioteca estándar en otros lenguaje de programación, que contiene las definiciones de tipos de datos básicos y funciones de uso común.

donde las funciones `integerEq` y `floatEq` son funciones de igualdad primitivas para cada tipo de datos, respectivamente.

La definición del operador `/=` queda definido (por omisión) en la clase `Eq`.

Las definiciones en las instancias se denominan *métodos*.

Las clases pueden derivarse, es decir que una clase puede heredar de otra, extendiéndola. Por ejemplo, la clase `Ord` deriva de `Eq`.

```
clase (Eq a) => Ord a where
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min :: a -> a -> a
```

Cabe aclarar que las clases de Haskell no son equivalentes al concepto de clase de la programación orientada a objetos. A continuación se describen algunas diferencias:

- Haskell separa la definición de un tipo de la definición de los métodos asociados a dicho tipo.
- Los métodos de Haskell se corresponden con las funciones virtuales de C++.
- Las clases de Haskell son similares a las interfaces de Java, es decir que definen un *protocolo* para el uso de objetos en lugar de un objeto en sí.
- Los tipos de Haskell no pueden ser *coercionados*.[10]
- Haskell no necesita de *virtual tables* en tiempo de ejecución.
- El sistema de clases de Haskell no contempla el control de acceso a métodos. El sistema de módulos de Haskell contempla el control de acceso.

4.7 Ejercicios

1. Aplicar la función $PLUS$ a 1 y 2.
2. Definir la función *IFTHENELSE*, la cual tendrá la semántica de una expresión condicional de la forma **if** *condición* e_1 **else** e_2.
3. Dada la expresión lambda $(\lambda x.x\ x)((\lambda x.x)y)$:
   (a) Reducir a forma normal usando orden normal.
   (b) Reducir a forma normal usando orden aplicativo.
   (c) Cuál estrategia es la mas eficiente, en este caso?
4. Reducir la expresión $(\lambda u.\ u\ u)\ ((\lambda x.\ x)\ (\lambda x.\ x))$ usando orden:
   (a) Normal.
   (b) Aplicativo.

[10] Una coerción es una conversión (cast) de tipos implícito.

5. Definir en LISP la función *length*, que retorna la longitud de una lista.

6. Definir en LISP una función que compute las raíces de una función utilizando el método de bisección (dado un error máximo permitido). Evaluar la longitud del programa usando *length*.

7. Implementar una función en Haskell que retorne una lista de los enteros pares positivos. Escribir un programa que muestre su uso.

8. Proponer cambios en la sintaxis y semántica del lenguaje kernel para convertirlo en un lenguaje funcional puro.

Ejercicios Adicionales

9. Dada la función $NIL \equiv \lambda x.TRUE$ que representa la lista vacía:

   (a) Definir el predicado $NULL$ (retorna $TRUE$ si su argumento es NIL).

   (b) Definir la lista $[1, 2, 3]$.

   Nota: Tratar las listas como Pares (head,tail) (ver definición de *PAIR*)

10. Implementar en Haskell una función `reverse` que tome un árbol binario (genérico) y retorne su espejado, es decir que en cada nodo tiene intercambiados sus sub-árboles izquierdos y derechos.

11. Implementar en Ocaml y/o Haskell uns función que tome una lista ordenada de valores de un tipo genérico y retorne un árbol binario balanceado con los elementos de la lista.

Capítulo 5

Programación Relacional

Un procedimiento en el modelo declarativo usa sus argumentos de entrada para calcular los valores de los argumentos de salida. Desde un punto de vista matemático se podría ver esto como un calculo funcional, en el sentido de que para un set de valores de argumentos de entrada solo hay un set de valores de argumentos de salida.

El procedimiento relación es mas general y flexible, dado que es posible dar cualquier numero de valores de salida (cero si no hay resultado). Esta flexibilidad resulta apropiada para el desarrollo de aplicaciones en áreas como Sistemas Expertos [1], Procesamiento del lenguaje humano, entre otras.

La programación relacional extiende la programación declarativa con una nueva sentencia llamada *choice*, la cual selecciona nodeterministicamente una de un set de alternativas posibles.

La flexibilidad de la programación relacional tiene algunas desventajas. Puede conducir a programas ineficientes si no se la utiliza apropiadamente. Cada nueva operación *choice* produce un aumento exponencial del espacio de búsqueda en busca de soluciones. Es muy práctica cuando el espacio de búsqueda es pequeño, no obstante suelen utilizarse técnicas apropiadas de optimización, búsqueda utilizando heurísticas o introduciendo restricciones cuando el espacio de búsqueda es considerable.

5.1 El modelo de Computación Relacional

5.1.1 Las sentencias *choice* y *fail*

El modelo computacional relacional extiende el modelo declarativo introduciendo las sentencias *choice* y *fail*.

- *choice* $< s >_1 [] \ldots [] < s >_n$ *end*. Elige nodeterministicamente una de un set de alternativas. Si la elegida llega a fallar entonces elige otra del conjunto. Cada vez que hace una elección determina lo que se llama un punto de elección. Un punto de elección es una parte de la máquina abstracta que encapsula la

[1] Un sistema experto es un programa que imita el comportamiento de un experto humano.

información necesaria para poder retroceder (*backtracking*) en la ejecución y elegir otra alternativa.

- *Fail.* Indica que la alternativa es erronea. Un fail es ejecutado implícitamente cuando se intenta ligar dos valores incompatibles (ej. 3=4).

Ejemplo:

```
fun {Soft} choice beige [] coral end end
fun {Hard} choice mauve [] ochre end end

proc {Contrast C1 C2}
    choice C1 = {Soft} C2 = {Hard} [] C1 = {Hard} C2 = {Soft} end
end
fun {Suit}
    Shirt Pants Socks
in
    {Contrast Shirt Pants}
    {Contrast Pants Socks}
    if Shirt==Socks then fail end
    suit(Shirt Pants Socks)
end
```

El siguiente programa ayuda a un diseñador de ropa a elegir los colores colores para el traje de un hombre. *Soft* elige un color suave y *Hard* uno fuerte. *Contrast* escoge un par de colores contrastantes (uno suave y uno fuerte). *Suit* retorna un juego completo de combinación de colores para un traje excepto aquellos en los que coincida la camisa con los calcetines.

5.1.2 Arbol de Búsqueda

Un programa relacional es ejecutado secuencialmente. En particular la sentencia *choice* va eligiendo las alternativas en el orden que ellas son encontradas (de izquierda a derecha). Cuando un *fail* es ejecutado, se produce un retroceso de la ejecución a la última alternativa elegida del *choice*, el cual elige la proxima alternativa, y asi sucesivamente. Esta estrategia puede ser ilustrada con un *arbol de busqueda*(ver figura 5.1).

5.1.3 Búsqueda Encapsulada

Un programa relacional es interesante porque este puede ejecutarse potencialmente de muchas maneras, dependiendo de las elecciones del *choice*. Por lo tanto se debe dar un control sobre cuales alternativas son elegidas y cuando. Por ejemplo, cual estrategia de búsqueda podría aplicar: *depth-first search*, *breadth-first search*, o alguna otra. También se debe especificar sobre como y cuantas soluciones son retornadas: *una solución, todas las soluciones, soluciones bajo demanda.*

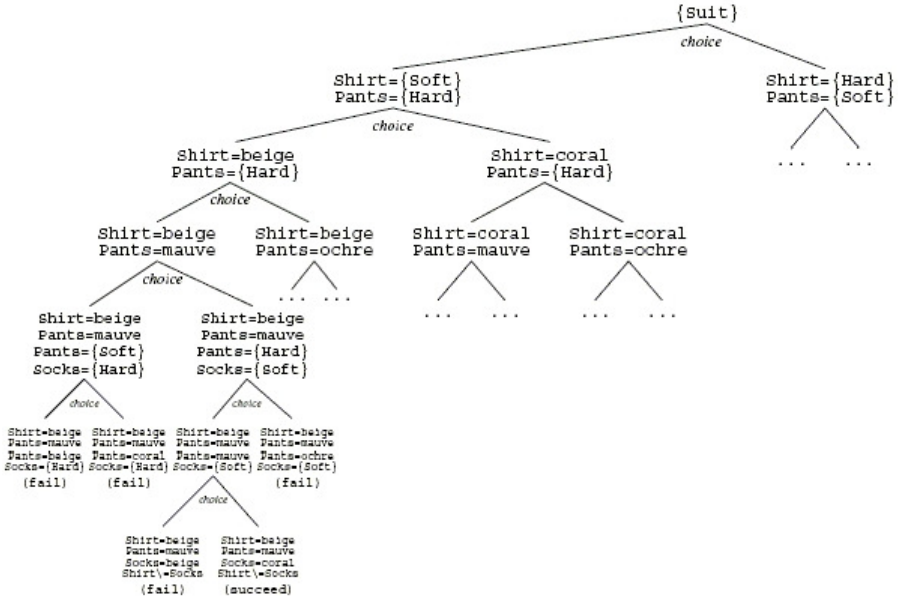


Fig. 5.1: árbol de Búsqueda para el ejemplo del diseñador de ropa.

Una manera de llevar a cabo este control es mediante búsqueda encapsulada. La encapsulación significa que el programa relacional se ejecuta dentro de una clase de 'environment'. Un programa relacional puede producir múltiples *binding* de la misma variable cuando diferentes elecciones son efectuadas. Estos múltiples *bindings* no deben ser visibles al resto de la aplicación. La busqueda encapsulada tambien es muy importante para la modularidad y composicionalidad:

- *Modularidad*: permite que se puedan ejecutar mas de un programa relacional concurrentemente.
- *Composicionalidad*: una búsqueda encapsulada puede ejecutarse dentro de otra búsqueda encapsulada.

Lenguajes mas viejos como Prolog tienen backtracking global, multiples *binding* son visibles al resto de la aplicación, lo cual no es bueno para los conceptos mencionados anteriormente.

5.1.4 La función *Solve*

La función *Solve* toma una función sin argumentos y retorna una solución a un programa relacional como una lista *lazy* de todas las soluciones, ordenadas de acuerdo a la estrategia depth-first search. Por ejemplo la llamada:

```
L={Solve fun {\$} choice 1 [] 2 [] 3 end end}
```

retorna la lista lazy [1,2,3], dado que *Solve* es *lazy*, es decir solo calcula las soluciones que son necesitadas. Solve es composicional, es decir, puede ser anidada (la función F puede contener llamadas a *Solve*). Se puede definir ahora las operaciones: *one-solution search* y *all-solutions search*.

```
fun {SolveOne F}
    L={Solve F}
in
    if L==nil then nil else [L.1] end
end

fun {SolveAll F}
    L={Solve F}
    proc {TouchAll L}
        if L==nil then skip else {TouchAll L.2} end
    end
in
    {TouchAll L}
    L
end
```

Ejemplos:

```
fun {Digit}
    choice 0 [] 1 [] 2 [] 3 [] 4 [] 5 [] 6 [] 7 [] 8 [] 9 end
end
{Browse {SolveAll Digit}}

Devuelve: [0 1 2 3 4 5 6 7 8 9]
```

5.2 Programación Relacional a Lógica

Tanto el modelo computacional declarativo como el relacional están muy ligados a la programación lógica. En la programación lógica los programas tienen dos semánticas, una lógica y una operacional. Sin embargo la programación lógica no puede ser usada para todos los modelos computacionales.
El paradigma lógico, que resultó una gran novedad en la década del 70, tiene como característica diferenciadora el hecho de manejarse de manera declarativa y con la aplicación de las reglas de la lógica. Esto significa que en lugar de basarse, como en el caso de los paradigmas procedurales, en el planteo del algoritmo para la resolución del problema, se basa en expresar todas las condiciones del problema y luego buscar un objetivo dentro de las declaraciones realizadas. Esta forma novedosa de tratamiento de la información llevó a pensar en un determinado momento en la revolución que significaría la existencia de 'programas inteligentes' que pudieran responder, no por tener en la base de datos determinados conocimientos, sino por poder inferirlos a través de

la deducción.
Existe toda una gama de lenguajes que siguen este paradigma, aunque la mayoría de ellos están basados en el lenguaje PROLOG.
Para nuestro propósito un programa lógico consiste de un set de axiomas en el calculo de predicados de primer orden, una sentencia llamada consulta (*query*) y un probador de teoremas, es decir, un sistema que puede realizar deducciones usando los axiomas a fin de probar o desaprobar una consulta (ejecución de un programa lógico).

Un sistema de estas características debe abordar los siguientes problemas:

- Un probador de teoremas es limitado en lo que éste puede hacer. No garantiza encontrar una prueba o una desaprobación de un *query* en cualquier modelo.
- Incluso en aquellos casos en donde es posible encontrar pruebas, puede ser ineficiente. La búsqueda para una prueba puede tomar un tiempo exponencial.
- La deducción efectuada por el probador de teoremas debe ser constructiva. En los casos en que un *query* es satisfecho por la existencia de algún x que cumple la propiedad, entonces el sistema debe construir tal testigo.

Dos posibles acercamientos para solucionar estos problemas son incorporando restricciones sobre los axiomas y dando al programador la posibilidad de proveer ayuda al probador de teoremas con conocimiento operacional. El primer lenguaje en incorporar estos acercamientos fue Prolog, luego fue subsecuentemente seguido por otros lenguajes.

5.2.1 Semántica Operacional y Lógica

Hay dos maneras de mirar un programa lógico: mediante una vista lógica y una operacional. La vista lógica es simplemente una sentencia de la lógica. La vista operacional define una ejecución sobre una computadora.
Los programas en el modelo declarativo tienen tanto una semántica lógica como operacional. Es directo traducir un programa declarativo en una sentencia lógica. Si el programa termina correctamente, es decir, no se bloquea, no entra en un loop infinito, o no surge una excepción, entonces todos los *bindings* son deducciones correctas desde los axiomas.

La tabla de la figura 5.2 define un esquema de traducción T el cual traduce cualquier sentencia $< s >$ en el lenguaje kernel relacional en una forma lógica $T(< s >)$. La definición de procedimientos son traducidas en definiciones de predicados. Las excepciones no son traducidas.

Una semantica lógica dada puede corresponder a muchas semánticas operacional. Por ejemplo las siguientes 3 sentencias:

```
X=Y <s>
<s> X=Y
if X==Y then s else fail end
```

Relational statement	Logical formula
skip	true
fail	false
$\langle s \rangle_1\ \langle s \rangle_2$	$T(\langle s \rangle_1) \wedge T(\langle s \rangle_2)$
local X **in** $\langle s \rangle$ **end**	$\exists x.T(\langle s \rangle)$
X=Y	$x = y$
X=f(l1:X1 ... ln:Xn)	$x = f(l_1 : x_1, ..., l_n : x_n)$
if X **then** $\langle s \rangle_1$ **else** $\langle s \rangle_2$ **end**	$(x = \text{true} \wedge T(\langle s \rangle_1)) \vee (x = \text{false} \wedge T(\langle s \rangle_2))$
case X **of** f(l1:X1 ... ln:Xn) **then** $\langle s \rangle_1$ **else** $\langle s \rangle_2$ **end**	$(\exists x_1, \ldots, x_n.x = f(l_1 : x_1, \ldots, l_n : x_n) \wedge T(\langle s \rangle_1)) \vee ((\neg\exists x_1, \ldots, x_n.x = f(l_1 : x_1, \ldots, l_n : x_n)) \wedge T(\langle s \rangle_2))$
proc {P X1 ... Xn} $\langle s \rangle$ **end**	$\forall x_1, \ldots, x_n.p(x_1, \ldots, x_n) \leftrightarrow T(\langle s \rangle)$
{P Y1 ... Yn}	$p(y_1, \ldots, y_n)$
choice $\langle s \rangle_1$ [] $\cdots$ [] $\langle s \rangle_n$ **end**	$T(\langle s \rangle_1) \vee \cdots \vee T(\langle s \rangle_n)$

Fig. 5.2: Traducción de un Programa Relacional a Fórmula Lógica.

tienen exactamente la misma semántica lógica: $X = Y \wedge T(s)$, pero sus semanticas operacionales son muy diferentes. Escribir un programa lógico consiste de 2 partes: escribir la semantica lógica y entonces elegir una semántica operacional para esta. El arte de la programación lógica es hacer un buen balanceo entre una simple semántica lógica y una eficiente semántica operacional.

Ejemplo: *Deterministic Append*

```
fun {Append A B}
    case A
    of nil then B
    [] X|As then X|{Append As B}
    end
end
```

Traducido a un procedimiento:

```
proc {Append A B ?C}
    case A of nil then C=B
        [] X|As then Cs in
            C=X|Cs
            {Append As B Cs}
    end
end
```

De acuerdo a la tabla de la figura 5.2 , este procedimiento tiene la siguiente semántica lógica:

$\forall a, b, c : append(a, b, c) \Leftrightarrow (a = nil \wedge c = b) \vee (\exists\, x, a', c' : a = x|a' \wedge c = x|c' \wedge append(a', b, c'))$

La semántica operacional esta dada por la semántica del modelo declarativo.
La llamada {Append [1 2 3] [4 5] X} se ejecuta satisfactoriamente y retorna X=[1,2,3,4,5]. La fórmula lógica de una llamada sería la tupla *append([1,2,3],[4,5],x)*. Luego de la ejecución la tupla queda *append([1,2,3],[4,5],[1,2,3,4,5])*. Esta tupla es un miembro de la relación *append* . Vemos que el procedimiento Append puede ser visto como un programa lógico.

El modelo relacional provee una forma de programación lógica nodeterministica muy parecida a la que provee Prolog. Mas precisamente es un subconjunto de Prolog llamado 'pure Prolog'. Los programas escritos en el modelo de computación relacional y pure Prolog tiene una traducción directa, con las siguientes diferencias:

- Prolog usa *clausulas de Horn* como sintaxis con una semántica operacional basada en el principio de resolución.
- Programación Higher-order no es soportada en pure Prolog.
- El modelo relacional distingue entre operaciones deterministicas (sin usar choice) y operaciones nodeterministicas (usando choice). En *pure Prolog* ambas tienen la misma sintaxis, las operaciones determinísticas realizan cálculos funcionales, mientras que las no deterministicas cálculos relacionales.

5.3 Prolog

Es un lenguaje de programación ideado a principios de los años 70 en la universidad de Aix-Marseille por los profesores Alain Colmerauer y Phillipe Roussel. Inicialmente se trataba de un lenguaje totalmente interpretado hasta que, a mediados de los 70, David Warren desarrolló un compilador capaz de traducir Prolog en un conjunto de instrucciones de una máquina abstracta denominada Warren Abstract Machine (WAM). Desde entonces Prolog es un lenguaje semi-interpretado.

Los programas en Prolog se componen de *cláusulas de Horn* que constituyen reglas del tipo 'modus ponens'. No obstante, la forma de escribir las cláusulas de Horn es al contrario de lo habitual. Primero se escribe el consecuente y luego el antecedente. El antecedente puede ser una conjunción de condiciones que se denomina *secuencia de objetivos*. Cada objetivo se separa con una coma y podría considerarse similar a una llamada a procedimiento de los lenguajes imperativos. En Prolog no existen instrucciones de control.

Su ejecución se basa en dos conceptos: la *unificación* y el *backtracking*.

Gracias a la unificación, cada *objetivo* determina un subconjunto de cláusulas susceptibles de ser ejecutadas. Cada una de ellas se denomina *punto de elección*. Prolog selecciona el primer punto de elección y sigue ejecutando el programa hasta determinar

si el objetivo es verdadero o falso. En caso de ser falso entra en juego el *backtracking*, que consiste en deshacer todo lo ejecutado situando el programa en el mismo estado en el que estaba justo antes de llegar al punto de elección. Entonces se toma el siguiente punto de elección que estaba pendiente y se repite de nuevo el proceso. Todos los objetivos terminan su ejecución bien en éxito ('verdadero'), bien en fracaso ('falso').

5.3.1 Elementos Básicos

Un programa Prolog podría verse como la representación de un universo finito en forma de hechos y reglas (hipotesis y axiomas), con el objetivo de encontrar soluciones en dicho universo. Para ello debemos contar con una gramática adecuada, cuyos elementos (términos) son los siguientes:

- *Símbolos de Variables*:
  representan objetos cualesquiera del universo. Comienzan siempre con mayúsculas. Ejemplos de variables son: *L*, *L2*, *Persona*, _ (variable anónima). Una variable puede estar ligada cuando existe algún objeto representado por ella, o no ligada en caso contrario.

- *Predicados y funciones*:
  La sintaxis de ambos es la misma. Ejemplos de ellos son:

  ```
  padre_de(juan, berta), ama(maria, X), suma(secesor(0), X, 2),...
  ```

  Sin embargo aunque idénticos en su sintaxis difieren conceptualmente, ya que una función nos devuelve un valor: sucesor(0) tiene como referente el valor 1, en cambio un predicado describe una propiedad o una relación entre sus elementos, ya que pueden ser verdaderas o falsas.
  Una característica fundamental del Prolog es que en el fondo todos los términos pueden reducirse a predicados: en Prolog todo son predicados.

- *Atomos*:
  Las estructuras de tipo predicado también se denominan átomos. Se dice que un átomo es cerrado cuando sus argumentos son variables ligadas, y que no lo es cuando tiene un argumento sin ligar.

- *Listas*:
  Una lista está formada por un primer elemento (*head*) mas una lista (*tail*). Ejemplos:

  ```
  [] (lista vac\'ia, tambi\'en llamada 'nil'), [A, B, C], [Cabeza|Resto]
  ```

5.3.2 Cláusulas Prolog

Las clausulas son estructuras que permiten representar *hechos*, *preguntas o consultas* y *reglas*.
Un hecho es algo que siempre es verdadero en un determinado universo. La sintaxis de los *hechos* y las *preguntas* es la misma. Ejemplo:

Programa 1:

```
1.es_padre(terach, abraham).        9.es_hombre(terach).
2.es_padre(terach, nachor).         10.es_hombre(abraham).
3.es_padre(terach, haran).          11.es_hombre(nachor).
4.es_padre(abraham, isaac).         12.es_hombre(haran).
5.es_padre(haran, lot).             13.es_hombre(isaac).
6.es_padre(haran, milcah).          14.es_hombre(lot).
7.es_padre(haran, yiscah).          15.es_mujer(sarah).
8.es_madre(sarah, isaac).           16.es_mujer(milcah).
                                    17.es_mujer(yiscah).
```

Toda clausula de prolog ha de terminar con un punto.
Todos los hechos anteriores son hechos de base (sin variables), pero también se pueden introducir hechos con variables como axiomas, por ejemplo: suma(0, X, X). En ellos, las variables se consideran cuantificadas universalmente. Es decir, $\forall x : suma(0, x, x)$.

Al igual que el hecho es_mujer(sarah) establece la verdad de la sentencia 'Sarah es mujer', el hecho suma(0, X, X) establece la verdad para cualquier valor que pueda tomar la variable, es decir, nos dice que 'para todo término x, la suma de 0 con x es x' . Equivale a un conjunto de hechos de base como serían: suma(0, 1, 1), suma(0, 2, 2), etc.

A un programa PROLOG se le hacen preguntas. Una pregunta se escribe como: ?A1, A2, . . . , Am. siendo $m > 0$. Informalmente, dicha pregunta se leerá: 'Son ciertos los hechos A1 y A2 y . . . y Am?'.

Por ejemplo, se puede hacer consultas al Programa 1 del siguiente tipo:

PREGUNTA	SIGNIFICADO	RESPUESTA
?es_padre(abraham,isaac).	¿es padre Abraham de Isaac?	**Sí**, pues encuentra un hecho que lo satisface.
?es_padre(abraham,lot).	¿es padre Abraham de Lot?	**No**, pues no hay ningún hecho que lo afirme.
? es_padre(haran,X).	¿existe un X tal que es padre Haran de X?	**Sí**, dando todas las soluciones posibles: X=lot ; X=milcah ;X=yiscah.

Las *reglas* son estructuras complejas compuestas por varios átomos. Constan de dos pares: una *cabeza*, formada por sólo átomo; y un *cuerpo*, en el que pueden aparecer varios átomos. Además de las reglas entran en juego los operadores lógicos que son la conjunción (representada por las comas que separan los átomos del *cuerpo*), y

la disyunción, que puede escribirse de 2 maneras: separando con punto y coma los átomos del *cuerpo*, o poniendo cada miembro de la disyunción en una cláusula aparte.

Ejemplo:

Una regla que expresa la relación de ser hijo es: es_hijo(X,Y) :- es_padre(Y,X), es_hombre(X). que se leería de la forma:'para todo X e Y, X es hijo de Y si Y es padre de X y X es hombre', con lo que sirve para definir una nueva relación a partir de otras.

NOTA: Esta regla representa la fórmula: $\forall xy((es\_padre(y,x) \wedge es\_hombre(x)) \rightarrow es\_hijo(x,y))$. De igual forma se definirían otras relaciones mediante reglas: es_hija(X,Y):- es_padre(Y,X),es_mujer(X). es_abuelo(X,Z):- es_padre(X,U), es_padre(U,Z).

El símbolo (:-) se lee como una implicación lógica, pero al revés, es decir, si el cuerpo de la cláusula es verdadero entonces la cabeza es verdadera. Podríamos entonces escribir los hechos como:
$madre(gea, cronos) : -true.$

Debemos reafirmar que en Prolog todo es predicado, ya que los operadores lógicos lo son. Así podríamos escribir en notación prefija: :-(A,B), en lugar de A:-B.

Con estas tres nuevas relaciones entre objetos y los hechos de base del Programa 1 se puede crear el siguiente Programa 2.

```
Programa 2:
18.es_hijo(X,Y):- es_padre(Y,X), es_hombre(X).
19.es_hija(X,Y):- es_padre(Y,X), es_mujer(X).
20.es_abuelo(X,Z):- es_padre(X,Y), es_padre(Y,Z).
```

Algunas consultas que se pueden efectuar al programa 2 son:

```
? es_hijo(lot,haran).

    Respuesta: S\'i, este hecho puede deducirse, ya que seg\'un la regla
    18 es equivalente a preguntar:
    ?es_padre(haran,lot),es_hombre(lot).

? es_hija(X,haran).     (Â¿existe un X tal que es hija X de Haran?)

    Respuesta: Seg\'un la regla 19 es equivalente a preguntar
    ? es_padre(haran,X), es_mujer(X). por lo que el programa
    contestar\'a que S\'i, dando las soluciones: X=milcah ; X=yiscah.
```

5.3.3 Fundamentos Lógicos de Prolog

Hay dos elementos esenciales a la base del Prolog: *las cláusulas de Horn* y el *principio de resolución* de Robinson.

5.3.3.1 La forma Clausal y las cláusulas de Horn

Prolog se sustenta en la lógica de predicados de primer orden, sin embargo no utiliza esta lógica en la forma que la conocemos, sino en lo que se denomina la forma clausal. En este formalismo contamos con los siguientes elementos:

- Constantes.
- Variables. De forma implícita todas las variables están cuantificadas universalmente.
- Predicados.
- Funciones.

Los predicados y funciones se denominan *literales* y pueden ser de dos tipos: *literales afirmados* (PERSONA(x)), y *literales negados* ($\neg$CASADO(juan)). Los literales se combinan para formar expresiones mas complejas denominadas cláusulas. Pero para llevar esto a cabo se necesitan las siguientes conectivas lógicas: *negación ($\neg$), disyunción ($|$), y conjunción*. Esta última no se representa explícitamente pero cuando tenemos un conjunto de cláusulas, éstas están relacionadas implícitamente mediante conjunción.
No se necesitan mas conectivas lógicas dado que el implica y coimplica pueden defnirse en términos de los anteriores. $A \rightarrow B$ puede definirse como $\neg A|B$.

$ABUELO(X,Y) : -PADRE(X,Z), PADRE(Z,Y)$ es lógicamente equivalente a (por la regla anterior y por ley de morgan) a $\neg PADRE(z,y)|\neg PADRE(x,z)|ABUELO(x,y)$

Hay un tipo de cláusulas que resulta especialmente adecuado para la prueba de teoremas con el principio de resolución. Son las *clausulas de Horn*. Se definen como aquellas cláusulas que tiene como máximo un literal no negado. De ello se desprende que hay dos tipos de *cláusulas de Horn*:

- Encabezadas: tienen un literal no negado. En prolog son las cláusulas que usamos para expresar las reglas y los hechos.
- No encabezadas: todos sus literales son negados. En prolog las utilizamos para formular las preguntas, o dicho de otra forma, son los teoremas que queremos probar ($: -es\_hija(X, haran).$).

Para resolver problemas de prueba de teoremas se consideran conjuntos de clausulas de Horn tales que una de ellas esté sin encabezar y el resto encabezadas, debido a la forma en que trabaja el principio de resolución que veremos a continuación.

5.3.3.2 El Principio de Resolución

Este principio de inferencia puede formularse como: de $\neg A \vee B$ y $A \vee C$ se deriva la cláusula $B \vee C$, donde las letras son literales. Las dos primeras cláusulas son los axiomas o hipótesis y las que se van obteniendo de ellas, teoremas o *resolventes*, de los cuales pueden ser utilizados para obtener nuevos resolventes. Hay dos casos particulares de esta ley:

- $\neg A \vee B$ y de A se deriva B (modus ponens).
- de $\neg A$ y de A se deriva la clausula vacía ($[]$).

Esto hace que el principio de resolución funcione bien como método de reducción al absurdo. Así, dado un conjunto consistente de axiomas, se puede probar un teorema (nuestro objetivo, que es una cláusula sin encabezar) negándolo en el punto de partida y procediendo a llevar a cabo la resolución de dos en dos cláusulas, eliminando de ambas los literales que 'coincidan', con la única diferencia de que en una de las cláusulas el literal ha de estar afirmado y en la otra negado, o tal como se encontraría en prolog, que en una cláusula se encuentre a la izquierda del :- (*cabeza*), y en la otra a la derecha (*cuerpo*). Esta 'coincidencia' de dos literales uno afirmado y el otro negado, se denomina **unificación** (ver próxima sección), y dará lugar a una nueva cláusula resolvente que podrá utilizarse en un paso posterior. Si se consigue llegar a la cláusula vacía, se habrá probado el objetivo.

Ejemplo:

1.$PADRE(jorge, juan)$
2.$HERMANA(rita, juan)$
3.$SOLTERA(rita)$

Estos hechos contienen gran cantidad de información implícita que se usaría para resolver este problema intuitivamente, por ejemplo, que 'juan' es varón, pero un probador de teoremas necesita que toda la información aparezca de forma explícita, por lo que debe especificarse:

4.$\neg SOLTERO(x) | \neg PADRE(y, x) | MISMOAPELLIDO(y, x)$
5.$\neg HERMANA(x, y) | \neg PADRE(z, y) | PADRE(z, x)$
6.$\neg MISMOAPELLIDO(x, y) | \neg MISMOAPELLIDO(y, z) | MISMOAPELLIDO(x, z)$
7.$\neg PADRE(x, y) | \neg PADRE(y, x) | VARON(y) | HIJO(y, x)$
8.$\neg HIJO(x, y) | MISMOAPELLIDO(x, z)$
9.$VARON(juan)$
Por ultimo se niega lo que queremos probar:
10.$\neg MISMOAPELLIDO(juan, rita)$

Se podría empezar a resolver esto por cualquier cláusula, pero hay caminos mas cortos que otros para llegar a la conclusión. Aquí se elige el mas corto. Por la cláusula 5 y teniendo en cuenta que las variables esta cuantificadas universalmente, seguirá

siendo verdad que si *rita* es hermana de *juan* y *jorge* es el padre de *juan*, entonces *jorge* es el padre de *rita*. Lo único que se ha hecho es sustituir las variables por valores concretos. Este resultado es el que obtenemos con las cláusulas 1, 2 y 5, sustituyendo las variables de 5 por lo valores que aparecen en 1 y 2, aplicando modus ponens, y efectuando la **unificación**.

11.$PADRE(jorge, rita)$
Siguiendo el mismo proceso de 3, 4 y 11 se deriva:
12.$MISMOAPELLIDO(jorge, rita)$
13.$HIJO(juan, jorge)$ //de 1, 7 y 9
14.$MISMOAPELLIDO(juan, jorge)$// de 8 y 13
15.$\neg MISMOAPELLIDO(jorge, rita)$ // de 6, 10 y 14
Pero en 12 decía justamente lo contrario, por lo que se ha llegado a una contradicción.
16.[]
Por lo tanto se concluye que juan y rita tienen el mismo apellido.

Este razonamiento puede resumirse de la siguiente manera: para probar la existencia de algo, suponer lo contrario y usar modus ponens y la regla de eliminación del cuantificador universal, para encontrar un contra ejemplo al supuesto. El objetivo es convertido en un conjunto de átomos a ser probados. Para ello, se selecciona un literal del objetivo $p(s1, ..., sn)$ y una cláusula de la forma $p(t1, ..., tn) : -A1, ...An$ para encontrar una instancia común de $p(s1, ..., sn)$ y $p(t1, ..., tn)$, es decir, una **sustitución** que hace que $p(s1, ..., sn)$ y $p(t1, ..., tn)$ sean idénticos. Tal sustitución se conoce como **unificador**. El nuevo objetivo se construye remplazando el átomo seleccionado en el original, por los átomos de la cláusula seleccionada, aplicando a todos los átomos obtenidos de esta manera.

El paso de computación básico, puede verse como una regla de inferencia puesto que transforma fórmulas lógicas y se denomina principio de resolución SLD.

Observar que generalmente, la computación de estos pasos de razonamiento no es determinista: cualquier átomo de la meta puede ser seleccionado y pueden haber varias cláusulas del programa que unifiquen con el átomo seleccionado. Otra fuente de indeterminismo es la existencia de unificadores alternativos para dos átomos. Esto sugiere que es posible construir muchas soluciones (algunas veces, una cantidad infinita de ellas). Por otra parte, es posible también que el átomo seleccionado no unifique con ninguna cláusula en el programa. Esto indica que no es posible construir un contra ejemplo.

5.3.3.3 Unificación y Regla de Resolución

El cómputo que PROLOG realiza para contestar a una pregunta (a un programa dado) se basa en los conceptos de unificación y resolución.

Unificación

Para definir la unificación formalmente son necesarias algunas definiciones previas:

Definición 5.3.1 *Una* ***sustitución*** *es un conjunto finito de pares de la forma* $\{v1 \rightarrow t1, v2 \rightarrow t2, \ldots, vn \rightarrow tn\}$*, donde cada* vi *es una variable, cada* ti *es un término (distinto de* vi*) y las variables* vi *son todas distintas entre sí.*

Cuando se aplica una sustitución a una expresión (regla, hecho o pregunta) se obtiene una nueva expresión, reemplazado en la expresión original cada aparición de la variable vi por el término ti $(1 \leq i \leq n)$.

Por ejemplo, dada la pregunta $c = es\_padre(Y, X), es\_hombre(X)$, y la sustitución $s = \{Y \rightarrow haran, X \rightarrow lot\}$, la pregunta obtenida al aplicar s a c será:
$c\ s = es\_padre(haran, lot), es\_hombre(lot)$.

Definición 5.3.2 *Dadas dos sustituciones* s *y* s'*, se define la operación de composición* $s \bullet s'$ *de la siguiente forma:*
Si $s = \{v1 \rightarrow t1, v2 \rightarrow t2, \ldots, vn \rightarrow tn\}$ *y* $s' = \{w1 \rightarrow t'1, w2 \rightarrow t'2, \ldots, wn \rightarrow t'n\}$*, entonces*
$s \bullet s' = \{v1 \rightarrow t1s', v2 \rightarrow t2s', \ldots, vn \rightarrow tns', w1 \rightarrow t'1, w2 \rightarrow t'2, \ldots, wn \rightarrow t'n\} - \{vi \rightarrow tis' | vi = tis'\} - \{wj \rightarrow t'j | wj \in \{v1, \ldots, vn\}\}$

La operación de composición es asociativa y tiene como elemento neutro la sustitución vacía que se denota ε. Además, dada una expresión c y dadas dos sustituciones s y s' se verifica: $(c\ s)\ s' = c\ (s \bullet s')$.

Por ejemplo, dada la pregunta $c = es\_padre(Y, X), es\_hombre(X)$ y las sustituciones $s = \{Y \rightarrow Z, X \rightarrow lot\}$ y $s' = \{Z \rightarrow haran, X \rightarrow juan\}$, la pregunta obtenida al aplicar primero s a c y después s' al resultado será:
$(c\ s)\ s' = (es\_padre(Z, lot), es\_hombre(lot))s' = es\_padre(haran, lot), es\_hombre(lot)$
que puede ser obtenida como $c\ (s \bullet s')$ ya que $s \bullet s' = \{Y \rightarrow haran, X \rightarrow lot, Z \rightarrow haran\}$

Un *unificador* de dos átomos $c1$ y $c2$, es una sustitución s tal que $c1\ s = c2\ s$, es decir, 's unifica o iguala $c1$ y $c2$'. Por ejemplo, dado $c1 = p(a, Y)$ y dado $c2 = p(X, f(Z))$. Los siguientes son unificadores de $c1$ y $c2$:

- $s1 = \{X \rightarrow a, Y \rightarrow f(a), Z \rightarrow a\}$ ya que $c1\ s1 = c2\ s1 = p(a, f(a))$.
- $s2 = \{X \rightarrow a, Y \rightarrow f(f(a)), Z \rightarrow f(a)\}$ ya que $c1\ s2 = c2\ s2 = p(a, f(f(a)))$.

Un unificador más general (umg) de dos átomos $c1$ y $c2$ es el unificador g de ambos tal que, cualquier unificador s de $c1$ y $c2$ se puede obtener como la composición de g con alguna otra sustitución u, es decir $s = g \bullet u$. El unificador más general de dos átomos es único (salvo renombramiento de variables).

En el ejemplo anterior el umg de $c1$ y $c2$ sería: $g = \{X \rightarrow a, Y \rightarrow f(Z)\}$ con
$c1\ g = c2\ g = p(a, f(Z))$.
El resultado $p(a, f(Z))$ obtenido tras aplicar g, es una expresión más general que la

obtenida aplicando $s1$, $p(a, f(a))$, o la obtenida aplicando $s2$, $p(a, f(f(a)))$.

Existe un algoritmo para decidir si dos átomos son unificables (esto es, si existe para ellos algún unificador). Este algoritmo siempre termina. Si los átomos son unificables el algoritmo devuelve el *umg* de ellos, en otro caso, devuelve 'fracaso'.

En el algoritmo se utiliza la noción de 'conjunto de disparidad' de dos átomos, definido como el conjunto formado por los dos términos que se extraen de dichos átomos a partir de la primera posición en que difieren en un símbolo. Por ejemplo, el conjunto de disparidad de $p(X, f(Y, Z))$ y $p(X, a)$ es $D = f(Y, Z), a$, mientras que el conjunto de disparidad de $p(X, f(Y, Z))$ y $p(X, f(a, b))$ es $D = Y, a$.

Algoritmo de Unificación Entrada: dos átomos $c1$ y $c2$ Salida: el umg g de $c1$ y $c2$ o $fracaso$

```
Algoritmo:
    g := set vacio
    fracaso := falso
    mientras c1 g <> c2 g  y no (fracaso)
        calcular conjunto de disparidad D de c1 g y c2 g
        si existe en D una variable V y un t\'ermino t tal que V no aparece en t
            g := g * {V --> t}.
        sino
            fracaso := cierto
        fin-si
    fin-mientras

    if fracaso
        devolver fracaso
    else
        devolver g
    fin-si
end.
```

Ejercicio: Unificar $\{f(X, h(Z), h(X)), f(g(a, Y), h(b), h(Y))\}$

Estrategia de Resolución usada en PROLOG

Dado un programa PROLOG como conjunto de hechos y reglas, y dada una pregunta a dicho programa, el cómputo que se realiza es aplicar paso a paso la regla de resolución con los hechos y/o reglas del programa hasta conseguir que en alguno de esos pasos queden eliminados todos los fines de la pregunta (es decir, hasta llegar a la pregunta vacía). En concreto, el procedimiento a seguir es el siguiente:
Dada una pregunta de la forma $?A1, A2, \ldots, An.$ se buscará la primera sentencia del programa $B :- B1, \ldots, Bm.$ tal que $A1$ y B sean unificables por un umg g. PROLOG

elige (de entre las posibles sentencias cuya cabeza B se unifica con $A1$) la primera introducida en el programa.
Por tanto, el orden en que se introducen las sentencias de un programa es significativo, entre otras cosas determinará el orden de las respuestas. Entonces la nueva pregunta (el resolvente) será $?(B1, \ldots, Bm, A2, \ldots, An)\ g$.

El proceso continuará de igual forma hasta obtener la pregunta vacía (lo que corresponde a un *éxito*), o una pregunta para la cual no exista resolvente con ninguna sentencia del programa (lo que corresponderá a un *fracaso*). Esta estrategia de resolución que sigue PROLOG se denomina **estrategia de resolución SLD**. La secuencia de pasos desde la pregunta original hasta la pregunta vacía se llama refutación SLD.

Ejemplo, considerando el Programa 2.

Pregunta : $?es\_hijo(lot, haran)$.

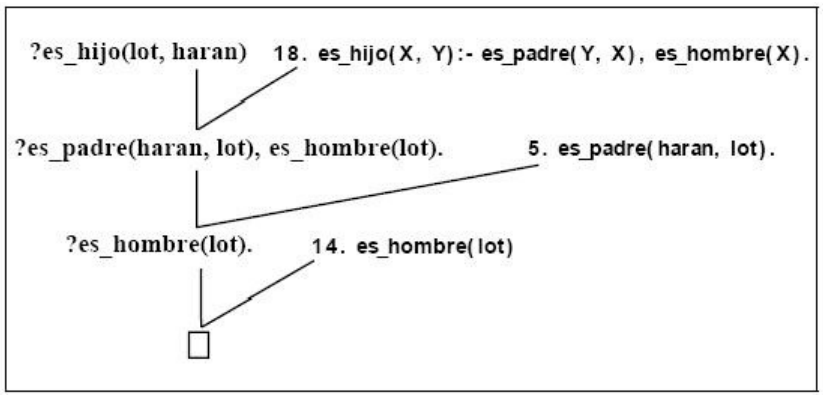


Fig. 5.3: Estrategia de Resolución SLD.

Como se ha llegado a la pregunta vacía (se ha encontrado una refutación), el hecho $es\_hijo(lot, haran)$ se deduce del programa, por lo que PROLOG contesta **sí**.
En cada paso del proceso de resolución se obtiene un unificador más general (umg). La composición de todos estos unificadores da lugar a las ligaduras que han tomado las variables finalmente para llegar a la respuesta. En el ejemplo:

$g1 = umg(es\_hijo(lot, Z), es\_hijo(X, Y)) = \{X \to lot, Y \to Z\}$
$g2 = umg(es_padre(Z, lot), es_padre(haran, lot)) = \{Z \to haran\}$
$g3 = umg(es_hombre(lot), es_hombre(lot)) = \varepsilon$

Las ligaduras de las variables finalizado el proceso, vienen dadas por la composición de $g1$, $g2$ y $g3$: $g = g1 \bullet g2 \bullet g3 = \{X \to lot, Y \to haran, Z \to haran\}$. Si de este conjunto de ligaduras nos quedamos sólo con las correspondientes a las variables de la

pregunta original, obtenemos la respuesta dada por PROLOG: $Z = haran$.

Recorrido en el Espacio de Búsqueda

PROLOG es capaz de dar todas las posibles respuestas a una pregunta, es decir, buscará todas las posibles refutaciones SLD. El espacio de búsqueda de soluciones se puede representar mediante un árbol, donde cada rama representa una posible refutación SLD. En lugar de expresar en cada paso de resolución la pregunta a tratar y la sentencia con la que se resuelve, tal y como vimos en el apartado anterior, se indicará únicamente el número de la sentencia con la que se resuelve dicha pregunta, tal y como se muestra en la figura 5.4.

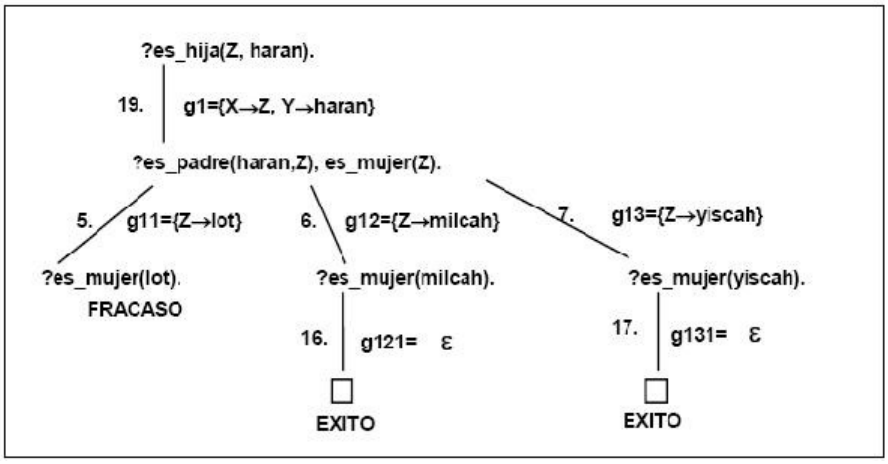


Fig. 5.4: Recorrido en el espacio de búsqueda.

Las respuestas a las dos refutaciones vienen dadas por las ligaduras de las variables en cada rama.

- $g = g1 \bullet g12 \bullet g121 = \{X \to milcah, Y \to haran, Z \to milcah\}$ que restringido a las variables de la pregunta da como respuesta $Z = milcah$.
- $g = g1 \bullet g13 \bullet g131 = \{X \to yiscah, Y \to haran, Z \to yiscah\}$ que restringido a las variables de la pregunta da como respuesta $Z = yiscah$.

Por tanto, el sistema contesta: $Z = milcah; Z = yiscah$

El recorrido en el árbol es en *profundidad* y con 'vuelta atras' (*backtracking*) a la última elección hecha (ver figura 5.5 para el ejemplo anterior).

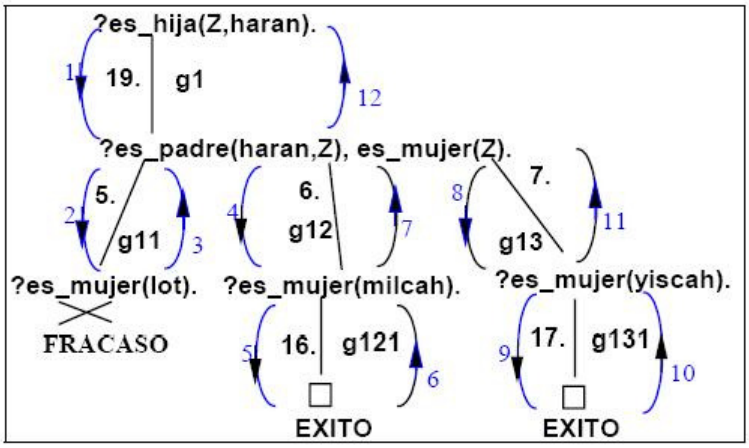


Fig. 5.5: Recorrido en el espacio de búsqueda con Backtracking.

5.3.4 Predicado cut (!)

Prolog dispone del corte (!) como método para podar la búsqueda y aumentar la eficiencia de los programas, es decir, se lo utiliza para cortar el backtracking.
Ejemplo:

```
rel :- a, b, !, c.
rel:- d.
```

Se indica al interprete que solo busque la primera solución que encuentre para las subconsultas *a* y *b*. En cambio para *c* podrá buscar múltiples soluciones. Cuando encuentra un *cut* inhibe también la evaluación de otra cláusula que defina la misma relación. En este caso la segunda cláusula que define a *rel* solo será evaluada en caso de que no se llegue a evaluar el *cut* en la primera.

5.3.5 Problema de la Negación

Los programas expresan conocimiento positivo y no dicen nada acerca del resto. De esta forma únicamente es cierto aquello que está en el programa o que sea consecuencia lógica del mismo.
Bajo esta hipótesis todo aquello que no está en el programa y que no pueda ser derivado a través de sus reglas es FALSO y por lo tanto su negación (metapredicado *not*)es cierta. Esta afirmación implica que el uso incorrecto puede resultar peligroso.

Ejemplo: 'Un piloto suicida es todo aquella persona que conduce y es irresponsable' e 'irresponsable es toda persona que no es responsable'.
Una posible traducción a un programa sería:

```
piloto_suicida(X):- conduce(X), irresponsable(X).
irresponsable(X) :- not( responsable(X) ).
```

Si el programa no contiene el predicado responsable nos encontraremos con que toda persona que conduce es un piloto suicida, lo que en la realidad no es cierto pero bajo esta hipótesis sí. La solución más correcta sería evitar la negación y definir el predicado *irresponsable*.

5.3.6 Predicado fail

Se utiliza para forzar el backtracking, dado que es un predicado que siempre falla. Se podría definir la negación utilizando los predicados *cut* y *fail*.

```
no(P) :- P, !, fail.
no(P).
```

5.4 Ejercicios

1. Teniendo en cuenta el ejemplo *Deterministic Append*, que sucede con la llamada *Append X [3] [1 2 3]* la cual lógicamente debería devolver *X=[1 2]*.

2. Considere las siguientes funciones:

```
fun {Digit}
  choice 0 [] 1  end
end

fun {TwoDigit}
  10*{Digit}+{Digit}
end

fun {StrangeTwoDigit}
  {Digit}+10*{Digit}
end
```

   Que muestran las siguientes sentencias:

   (a) *{Browse {SolveAll Digit}}*

   (b) *{Browse {SolveAll TwoDigit}}*

   (c) *{Browse {SolveAll StrangeTwoDigit}}*

   (d) Utilizando *Digit* generar todos los números *palindromos* de 4 dígitos.

3. Dada las siguientes premisas :

   1 $arc(a, b)$.

   2 $arc(a, c)$.

3 $arc(a, f)$.

4 $arc(c, b)$.

5 $arc(b, d)$.

6 $arc(X, Y) \rightarrow path(X, Y)$

7 $arc(X, Y) \wedge path(Y, Z) \rightarrow path(X, Z)$

Verificar mediante el método del absurdo la validéz de $? - path(a, A)$.
Muestre claramente las sustituciones en cada paso. En los casos de éxitos mostrar cual es la ligadura de A.

4. Dada las siguientes premisas :

   1 $hijo(luisPerez, josePerez)$.

   2 $hijo(mariaPerez, josePerez)$.

   3 $hijo(miguelPerez, josePerez)$.

   4 $hijo(luisPerez, lolaPerez)$.

   5 $hombre(lolaPerez)$.

   6 $hijo(X, Y) \wedge hombre(Y) \rightarrow padre(Y, X)$

   Verificar mediante el método del absurdo la validéz de $? - padre(X, Y)$.
   Muestre claramente las sustituciones en cada paso. En los casos de éxitos mostrar cual es la ligadura de X e Y.

5. Definir en Prolog los siguientes predicados:

   (a) $pertenece(L, L1)$. L está en la lista $L1$.

   (b) $length(L, N)$. Donde N es la longitud de la lista L.

   (c) $elimina(X, YS, ZS)$. ZS es la lista resultante de eliminar X de YS.

   (d) *Intersección* de conjuntos.

6. Definir la relación $MultipliacaLista(L1, F, L2)$ de forma que la lista $L2$ la lista resultado de multiplicar cada elemento de $L1$ por F
   Ejemplo de uso:
   ?- **MultiplicaLista([3,2,5], 2, [6,4,10]).**
   yes;
   ?- **MultiplicaLista(X, 3, [6,9,3]).**
   X = [2,3,1]
   Yes;

7. Definir la relación $sub\_set(L_1, L_2)$, donde L_2 es un subconjunto de L_1.
   No puede utilizar la relación $pertenece(X, L)$. Ejemplo:

```
?- sub_set([1 , 2, 3],L).
L = [1,2,3] ? ;
L = [1,2] ? ;
L = [1,3] ? ;
L = [1] ? ;
L = [2,3] ? ;
L = [2] ? ;
L = [3] ? ;
L = []
yes
```

8. Dado el siguiente programa Prolog:

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

   (a) Es ineficiente? porque?
   (b) Proponer una mejora de su eficiencia.
   (c) Construir el arbol SLD de ambas versiones con la siguiente consulta: $?max(2, 2, Z)$.

9. Dar un ejemplo del problema de la negación en Prolog.

10. Considerando el siguiente programa, que devuelven las siguientes consultas? justificar en cada caso.

```
hombre(juan).
hombre(pedro).
mujer(X) :- not(hombre(X)).
```

   (a) $?mujer(juan)$.
   (b) $?mujer(X)$.
   (c) $?not(not(hombre(X)))$. Además, $not(not(hombre(X)))$ es equivalente a $hombre(X)$?.

11. Teniendo en cuenta el predicado $pertenece(L, L1)$ del ejercicio 5 en su versión eficiente, usando el $cut(!)$, y en su versión ineficiente, sin uso del $cut(!)$, construir el correspondiente árbol SLD para la siguiente consulta: $?pertenece(X, [1, 2, 1, 3])$.

Ejercicios Adicionales

12. Definir en Prolog los siguientes predicados:

   (a) $length(L, N)$. Donde N es la longitud de la lista L.
   (b) $append(A, B, C)$. Verifica si C es la lista obtenida concatenando los elementos de la lista B a continuación de los elementos de la lista A.
   (c) *unión* y *diferencia* de conjunstos.

Capítulo 6

El modelo con estado (statefull)

El modelo o estilo de programación con estado explícito permite que un componente (un procedimiento por ejemplo) depende de parámetros internos (su estado) además de sus argumentos. Estos parámetros internos permiten que se *memoricen* valores entre sus activaciones, lo cual, permite simplificar algunas tareas que naturalmente requieren la noción de estado.

El concepto de estado explícito hace perder declaratividad en el sentido que las definiciones ahora se basan en la noción de cambios de estado para llegar a una solución final lo cual hace perder el purismo del modelo *aplicativo*, centrándose mas en la utilización de comandos de cambios de estado.

Los cambios de estado de las variables se realizan por medio de la operación de *asignación*, la cual se torna en una operación fundamental en el modelo imperativo. Este modelo se conoce como imperativo ya que se enfoca más en el cómo que en el qué de un problema. Un programa imperativo describe una secuencia de cambios de estado para llegar al resultado esperado (estado final).

En el modelo declarativo, el concepto de estado está implícito en los valores intermedios utilizados para arribar a la solución final.

Los parámetros internos brindan un mecanismo de memoria de largo plazo, permitiendo definir programas que evolucionan, mientras que el estilo de programación funcional brinda un mecanismo de memoria de corto plazo, es decir que los estados intermedios se descartan (olvidan).

La idea básica es que en este modelo disponemos de *celdas*, las cuales se comportan como *referencias* a valores.

De manera mas rigurosa, una celda es un par (id, ref) donde id es su identificador (nombre) y ref es una referencia a la memoria de datos.

Es posible modificar la referencia de una celda, por eso se dice que las celdas son mutables.

Una celda representa una *referencia* o *puntero* en un lenguaje de programación imperativa como Pascal, C, C++ o Java.

En estos lenguajes una variable es mutable, es decir puede ser modificada. Una variable puede ser de un tipo básico (entero, caracter, real, ...), estructurados (arreglos, registros, etc) o una referencia a un valor.

Las variables de los lenguajes mencionados arriba generalmente son mutables (excepto las constantes) y con un alcance y tiempo de vida dependiente del contexto (bloque) de su declaración.

Las referencias, en estos lenguajes, son variables (tipadas) cuyos valores son direcciones de memoria (de la celda que contiene el valor). En algunos lenguajes (como Java) sus valores referenciados se deben crear dinámicamente mediante operadores especiales (ej: *new*).

Una diferencia entre las referencias de C++ y Java es que en C++ son inmutables y deben estar inicializadas, es decir no se pueden hacer apuntar a otra celda (o a *null*), por lo que en C++ son realmente alias de de otras variables.

Generalmente, en los lenguajes con referencias, el valor referenciado se accede de manera implícita (es decir que el identificador de la referencia denota el valor referenciado). Esto se conoce como *referenciación implícita*. Ejemplos de estos tipos de referencias encontramos en Pascal, C++ y Java.

Un *puntero* es similar a una referencia pero en general pueden asignarse y su inicialización es opcional.

Los punteros generalmente tienen asociados operadores de *referenciación* (acceder al valor del puntero) y eventualmente de *desreferenciación* (obtener la dirección de memoria de una variable).

El lenguaje C tiene ambos operadores (* y &), respectivamente.

Los punteros no son *alias* de la variable (ya que pueden cambiar de variable apuntada), aunque sí es posible construir distintas expresiones que pueden denotar una misma entidad (valor) en un momento dado.

De esta manera es posible definir una referencia como una abstracción de un puntero, en el sentido que un puntero puede verse como un valor cuando una referencia es un tipo abstracto de datos (TAD) que encapsula y oculta un puntero, el cual sólo puede manipulado por sus operaciones (asignación, referenciación y desreferenciación).

En general ambos tipos de datos son tipos de datos parametrizados sobre el tipo de valores al que hacen referencia.

Existen lenguajes, como Pascal y C++ que contiene ambos tipos de datos: referencias y punteros. Otros, como Java sólo tienen referencias.

6.1 Semántica de celdas

Existen operaciones básicas sobre celdas:

- `{NewCell X C}`: crea una nueva celda `C` con valor inicial `X`.
- `{Exchange C X Y}`: liga (atómicamente) el contenido de `X` con el contenido de `C` y `C` toma el nuevo valor `Y`.
- `C := X` (asignación): el nuevo contenido de `C` es `X`.
- `@C`: retorna el valor referenciado por `C`.

Se permitirá usar la asignación `C := X` como una expresión la cual arroja el valor anterior de C. La sentencia `X = C := Y` es equivalente a `{Exchange C X Y}`.

Antes de poder definir la semántica de las nuevas operaciones es necesario ampliar el modelo de la máquina abstracta. Es necesario una nueva memoria, denominada *memoria mutable (mutable store)* μ la cual contendrá las celdas (inicialmente vacía).

Esta memoria contiene un conjunto de pares, denotados como $x : y$, donde el identificador x está ligado al nombre de la celda (identificador interno) e y puede ser cualquier valor parcial. El identificador interno juega el rol de *dirección de memoria del valor refereciado*.

Un estado de ejecución de la máquina abstracta ahora se convierte en una tripla (ST, σ, μ).

A continuación se describe la semántica de las operaciones sobre celdas, describiendo el comportamiento de la máquina con cada estado de ejecución involucrando las operaciones básicas sobre celdas.

- $(\{Newcell < x >< y >\}, E)$
  1. Crear un nombre (valor) único (n) en σ para la celda.
  2. $bind(E(< y >), n)$ en σ.
  3. Si la ligadura tuvo éxito, agregar el par $(E(< y >) : E(< x >))$ a la memoria μ.
  4. Si la ligadura falló, generar un error.
- $(\{Exchange < x >< y >< z >\}, E)$
  1. Si $E(< x >)$ es determinado, hacer:
     (a) Si $E(< x >)$ no está ligado a un nombre de una celda, error.
     (b) Sea $(E(< x >) : w) \in \mu$:
        i. Actualizar μ tal que contenga el par $(E(< x >) : E(< z >))$.
        ii. $bind(E(< y >), w)$ en σ.
  2. Sino, la máquina pasa al estado **Suspended**.

Se debe notar que las demás operaciones sobre celdas `@C` y `C:=X` se pueden definir en términos de `Exchange` y se dejan como ejercicio.

La semántica dada, simula el cambio de valores de las celdas por medio de la memoria μ, aunque es importante notar que no se realizan cambios en los valores básicos del programa, ya que siempre se encuentran en la memoria inmutable σ.

Las implementaciones características de los lenguajes imperativos sólo mantienen la representación de la memoria μ, donde cada celda se puede representar con un par (id, ref) si el lenguaje es interpretado.

En el caso de un lenguaje compilado, los identificadores se reemplazan con la dirección de memoria del valor referenciado y un puntero o referencia es un valor que contiene una dirección de memoria (la del valor referenciado o una dirección especial usada para denotar que no se hace referencia a algún valor en particular, como el valor *null* de Java).

En un programa compilado generalmente la memoria μ contiene directamente los valores de los tipos de datos definidos en el programa, es decir que los identificadores están implícitamente asociados a las direcciones de memoria en que se almacenarán sus valores.

La inclusión en la máquina abstracta de la memoria mutable μ, requiere que se redefinan los conceptos de alcanzabilidad y del reclamo de celdas para el recolector de basura.

Definición 6.1.1 *Una variable* $\boldsymbol{y}$ *es alcanzable si* μ *contiene* $\boldsymbol{x:y}$ *y* $\boldsymbol{x}$ *es alcanzable.*

Definición 6.1.2 *Si la variable* $\boldsymbol{x}$ *no es alcanzable y* μ *contiene el par* $\boldsymbol{x:y}$ *(para cualquier* $\boldsymbol{y}$*) es posible eliminarlo con seguridad.*

6.2 Aliasing

Dos variables son *alias (sinónimos)* si se refieren a la misma entidad (celda).

Es obvio que en el modelo con estado explícito es muy común que se produzca aliasing. El problema del aliasing es que complica el razonamiento sobre los programas, ya que el cambio a una variable puede afectar a la otra (alias) ya que denotan la misma entidad.

En programas grandes es difícil para el programador llevar la pista de las variables que pueden ser alias en alguna instancia de ejecución.

Este es el principal motivo del porqué la abstracciones de datos son adecuadas en el modelo con estado explícito.

La idea básica de la abstracción de datos es el *encapsulamiento del estado*, esto es, el ocultamiento del estado dentro de componentes. El estado sólo puede ser modificado a través de sus operaciones. Estas son las ideas básicas de los tipos abstractos de datos (ADTs) y su objetivo es minimizar los efectos colaterales de las operaciones, ya que cada valor sólo se puede manipular por medio de sus operaciones correspondientes.

6.3 Igualdad

En el modelo declarativo se vio que la la igualdad requerida es la estructural ya que tiene sentido que dos valores son iguales si poseen la misma representación.

En el modelo imperativo podemos tener el siguiente caso:

```
X=\{NewCell 10}
Y=\{NewCell 10}
```

La comparación `X==Y` debería arrojar **true**?

La respuesta es claramente no, porque estamos comparando *celdas*, cada una de las cuales tiene identidad propia. Si se modifica el contenido de una celda, no se afecta a la otra.

Este tipo de igualdad se conoce como *igualdad de tokens* en contraposición de la igualdad estructural. Se puede notar que se corresponde a la comparación de punteros o referencias en lenguajes como Pascal, C y C++, lo que significa comparación de valores que representan direcciones de memoria.

El modelo presentado aquí es muy similar a las referencias de la familia de lenguajes funcionales ML.

Obviamente en una expresión `@X==@Y` se realiza la comparación habitual de valores (entailment check).

6.4 Construcción de sistemas con estado

En la construcción de grandes programas (sistemas) es fundamental tener el control sobre su complejidad para permitir su análisis y su mantenibilidad (corrección, adaptación y extensión).

Los grandes sistemas requieren que puedan verse como varias partes o subprogramas combinados entre sí.

Los sistemas deben tener las siguientes propiedades, las cuales son diferentes clases de un principio mas general: *abstracción*.

- **Encapsulamiento**: debería ser posible ocultar los detalles internos de cada parte (information hidding).
- **Composicionalidad**: se deberían poder combinar las partes para formar otras.
- **Instanciación/invocación**: debería ser posible crear varias instancias a partir de una misma definición. Este mecanismo permite la reutilización de partes.

6.4.1 Razonando con estado

Partes de un programa imperativo puede tener *efectos colaterales*: modificación de variables que pueden ser usados en otras partes del programa. Esto, junto al problema de aliasing, hace extremadamente difícil el razonamiento de los programas imperativos.

El encapsulamiento permite reducir la complejidad de los estados.

El razonamiento sobre estados requiere que cada construcción o sentencia del lenguaje tenga definida su semántica en base a cómo transforma el estado previo a su ejecución. Esta semántica se conoce como semántica axiomática, ya que cada regla de transformación de estado se define como un axioma de una lógica.[1]

Los métodos de razonamiento se basan en que es posible definir *invariantes* de componentes (principalmente en iteraciones).

Un invariante es un predicado (función booleana) que establece que en ese punto del programa ese estado debería valer (ser verdadero).

Es posible expresar o especificar el comportamiento de una parte de un programa en base a su estado de entrada o *precondición*, su estado de salida o *postcondición* y su invariante, el cual especifica una condición de su evolución o relación entre sus (principales) variables.

El invariante da una idea de la evolución de una parte de un programa pero no de su comportamiento o finalización. Para esto es preciso dar una idea de *progreso*.

Lo anterior muestra que la construcción de programas imperativos no es tan simple como en la programación declarativa, por lo que es conveniente mantener los componentes de un programa tan declarativos como se pueda, ya que el razonamiento ecuacional es mas fácil y composicional.

6.4.2 Programación basada en componentes

Las tres propiedades, encapsulamiento, composicionalidad e instanciación, definen la programación basada en componentes.

Un componente es una parte de un programa que define dos cosas: una hacia el exterior, su *interface* y otra hacia adentro, su *implementación*.

Los componentes de pueden definir a diferentes niveles.

- **Abstracción procedural**: el componente se denomina la definición de procedimientos y sus instancias son las invocaciones o llamados a procedimientos.
- **Functores (unidades de compilación)**: pueden ser compilados en forma separada. Sus definiciones son functores y sus instancias módulos.
- **Componentes concurrentes**: un sistema con entidades independientes, interactuando entre sí por medio de mensajes.

[1] Es bien conocido la aplicación de la lógica de Hoare para razonar sobre programas con estado explícito.

Los sistemas basados en componentes se construyen principalmente en base a composición: un nuevo componente combina instancias de otros.

La programación orientada a objetos provee mecanismos para la definición y uso de componentes, con un mecanismo adicional para la definición de nuevos componentes: la *herencia*.

La herencia permite definir un nuevo componente mediante la extensión de otros. Estas extensiones pueden ser especializaciones, alternativas de implementación, entre otras. Este mecanismo permite otra dimensión adicional para la definición incremental y jerárquica de componentes.

Se analizarán los detalles de la programación orientada a objetos en el capítulo 9.

6.5 Abstracción procedural

Un procedimiento es una abstracción de un comando, es decir una acción parametrizada. Una función es una abstracción de una expresión (denota un valor).

Para que una abstracción pueda reutilizarse deben poder parametrizarse, es decir que se pueda definir en términos de datos de entrada los cuales serán instanciados en cada invocación.

Los parámetros usados en la definición de una abstracción procedural (procedimiento o función) se denominan *parámetros formales*.

Los valores aplicados en una invocación a un procedimiento o función se denominan *parámetros actuales o reales*.

La mayoría de los lenguajes de programación permiten uno o varios mecanismos de pasajes de parámetros. Estos mecanismos de pasajes de parámetros definen diferentes tipos de ligaduras entre los parámetros actuales y formales.

A continuación se da una clasificación de los diferentes mecanismos de pasajes de parámetros.

- **Por copia**
  - **por valor** (o de entrada): el parámetro formal se liga a una copia del parámetro actual. Cualquier modificación al parámetro formal no afecta al parámetro actual.
  - **por resultado** (o salida): el parámetro formal se copia al parámetro actual durante el retorno. En la entrada al procedimiento el parámetro formal no está ligado a ningún valor concreto.
  - **por valor-resultado** (o entrada-salida): combinación de los dos mecanismos anteriores.
- **Denotacionales**

– **por referencia**: el parámetro formal se convierte en una nueva identidad del parámetro actual. El parámetro puede usarse libremente. Este es el mecanismo usado por el lenguaje kernel utilizado.

  Algunos lenguajes imperativos usan una pequeña variante de este método, pasando la identidad (ej: dirección de memoria) del parámetro actual. El parámetro formal se accede (indirectamente) usando esa identidad. En la práctica, se accede al valor del parámetro usando direccionamiento indirecto.

  Este mecanismo se conoce como pasaje *variable* o *por dirección*.

– **por nombre**: la expresión (sin evaluar) que denota el parámetro actual se liga al parámetro formal. Cada referencia al parámetro formal, dentro de la subrutina, lanza la evaluación de la expresión asociada. En general se implementa asociando al parámetro formal una función, la cual es evaluada en cada referencia al parámetro formal.

  El mecanismo conocido como **por necesidad** (by need) es una modificación del mecanismo anterior, donde la expresión (o función) se evalúa una sola vez. Las próximas referencias al parámetro formal retornarán el resultado que se *recuerda*.

6.6 Ejercicios

1. Describiendo los cambios que sufren tanto la memoria inmutable como la mutable, decir que muestra cada uno de los siguientes algoritmos:

   (a)
   ```
   local X Y in
      X=1
      {NewCell X Y}
      {Browse @Y}
   end
   ```

   (b)
   ```
   local Z X C1 C2 Temp1 Temp2 in
      X=10
      {NewCell X C1}
      {NewCell C1  C2}
      {Browse @C1}
      {Exchange C1 Temp1 C2}
      {Exchange C2 Temp2 Temp1}
      {Browse @C2}
      {Browse @C1}
    end
   ```

   (c)
   ```
   local X Y Z in
      X=1
      {NewCell X Y}
   ```

```
      {NewCell 1 Z}
      {Browse @Y==@Z}
   end
```

2. Definir las siguientes expresiones en base a la operación primitiva *Exchange*.
   (a) X = C := Y
   (b) C := X
   (c) X = @C

3. Traducir a lenguaje núcleo y ejecutar según la máquina abstracta.

```
local X C Y in
   Y=1
   {NewCell 0 C}
   {Exchange C X Y}
   {Browse @C}
   {Browse X}
end
```

4. Implementar la función `Reverse L`, la cual retorna la reversa de una lista utilizando estado.

5. Implementar en el lenguaje kernel con estado el tipo de datos Stack. Escribir un ejemplo de su uso.

6. Determinar el tipo de pasaje de parámetros que usan los siguientes lenguajes:
   (a) Pascal
   (b) C
   (c) C++
   (d) Java
   (e) Haskell

Ejercicios Adicionales

7. Dar esquemas en el lenguaje kernel con estado para simular cada uno de los mecanismos de pasajes de parámetros descriptos.

8. Describiendo los cambios que sufren tanto la memoria inmutable como la mutable, decir que muestra cada uno de los siguientes algoritmos:
   (a)
```
local X Y Z in
   {NewCell 2 Z}
   {NewCell Z X}
   {NewCell X Y}
```

```
    {Browse @Y}
    {Browse @X}
  end
```

(b)
```
local X Y Z in
   X=1
   {NewCell X Y}
   Z=Y
   {Browse Y==Z}
end
```

9. Traducir a lenguaje núcleo y ejecutar según la máquina abstracta.

```
local X C in
  X=1
  C={NewCell X}
  {Exchange C @C @C+1}
  {Browse @C}
  {Browse X}
end
```

Capítulo 7

Lenguajes de programación imperativos

En este capítulo se analizan algunos conceptos comunes a varios lenguajes imperativos, en particular sentencias comunes de control y tipos de datos generalmente utilizados.

En la sección 7.4 se hace una descripción general del lenguaje C, el cual es un punto de partida para el estudio de varios lenguajes derivados como C++, Objective C, Java y C#.

Los lenguajes de programación imperativos generalmente tienen construcciones sintácticas para realizar declaraciones (y particularmente definiciones) de entidades, como variables, constantes, tipos de datos y procedimientos y funciones. Los lenguajes modernos, los cuales se basan en los conceptos de la programación estructurada[1] , contienen sentencias de control y computaciones (asignación) y generalmente proveen operadores básicos y mecanismos para construir expresiones.

Generalmente los mecanismos de abstracción que proveen son la abstracción procedural y funcional y las definiciones de nuevos tipos de datos.

7.1 Declaraciones

Generalmente un programa es un conjunto de declaraciones. Algunas declaraciones introducen nuevas entidades, por lo que se denominan definiciones, como lo son las definiciones de variables, de nuevos tipos de datos y de procedimientos y funciones.

Algunas declaraciones simplemente hacen referencia a entidades existentes, como por ejemplo una declaración de una referencia externa, típica declaración de las entidades en la interface de un módulo o una declaración por adelantado (forward), es decir la declaración de una entidad que se encuentra definida más adelante en el programa.

[1] La programación estructurada se basa en el uso de sentencias de control de flujo con una semántica bien definida y la prohibición del uso de control de flujo basado en instrucciones de saltos (goto).

7.2 Expresiones y comandos

Algunas sentencias en un lenguaje imperativo toman la forma de expresiones o comandos de control de flujo de ejecución.

Una expresión representa un valor (de algún tipo de datos). Los comandos no denotan valores, sino que representan construcciones sintácticas de control estructurado del flujo de ejecución de las sentencias que contienen.

La operación fundamental de cómputo es la asignación y la evaluación de expresiones.

En algunos lenguajes la sentencia de asignación es un comando (ej: Pascal, Eiffel), mientras que en otros es una expresión.

Cuando la asignación toma la forma de una expresión permite realizar asignaciones múltiples en secuencia, como sucede, por ejemplo, en C.

Las expresiones se forman a partir de valores constantes (literales o agregados, dependiendo si se corresponden con valores de tipos básicos o estructurados, respectivamente) y operadores. Los operadores son abstracciones lingüísticas adecuadas las cuales se traducen o implementan con operaciones de máquina (como suma, multiplicación, etc) sobre algunos tipos de datos básicos (ej: enteros y reales en punto flotante) o bien como invocaciones a funciones de biblioteca, en el caso que no sean soportadas directamente por el hardware (ej: funciones trigonométricas, generación de números aleatorios, etc).

Los operadores predefinidos en el lenguaje generalmente respetan algún orden de precedencia y tienen alguna regla de asociatividad (a izquierda, derecha o no son asociativos).

Los lenguajes imperativos como Pascal o C, son lenguajes monomórficos, es decir que no soportan polimorfismo. Algunos operadores son polimórficos (ej: operador + de Pascal, el cual suma enteros, reales, concatena strings y además hace unión de conjuntos) pero generalmente se implementan por sobrecarga, es decir mediante la implementación de varias funciones diferenetes con argumentos de diferentes tipos. El compilador o intérprete invoca a la función correspondiente en base a los tipos de sus operandos.

Algunos lenguajes de programación modernos, como por ejemplo C++, permite que el programador sobrecargue los operadores del lenguaje (con algunas pocas excepciones).

```
Sintaxis:
tipo  operator + (lista de par\'ametros);

Ejemplo:
Caja operator+(Caja a) {
  Caja temp;
  temp.longitud = longitud + a.longitud;
```

```
  temp.anchura = anchura + a.anchura;
  temp.altura = altura + a.altura;
  return temp;
}
```

Los comandos o sentencias de control se pueden clasificar en las siguientes categorías:

- **Bloque o secuencia**: grupos de sentencias **begin** $s_0; s_1; \ldots s_n$ **end** las cuales se ejecutará en secuencia. Los delimitadores de bloque mas usados son los pares **begin,end**, { , }, etc. En algunos lenguajes los bloques no están delimitados por palabras claves (keywords), sino por reglas de indentación, como por ejemplo en Python y Haskell.

- **Selección**: sentencias de la forma `if <cond> then <s1> else <s2> end` y sus variantes (cláusulas `elseif <cond2 ...>`). Estas últimas cláusulas son usualmente abreviaciones sintácticas de varias sentencias de selección **if** anidadas.

  Otra sentencia de selección común es la de selección por casos, como la sentencia **case** de Pascal o la sentencia **switch** de C, C++ o Java.

  Estas sentencias permiten implementar una especie de tablas de despacho para diferentes grupos (bloques) de sentencias. Muchos lenguajes requieren que los valores de cada caso sea una constante, aunque algunos permiten que sean expresiones evaluables en tiempo de ejecución.

- **Iteración**: permiten repetir la ejecución de un bloque un cierto número de veces. Existen diferentes tipos de sentencias de iteración:

  - *Indefinida*: como los clásicos comandos **while**, o *repeat-until*. Estas sentencias pueden repetir la ejecución de su cuerpo cero (como en caso del **while**), una o un número indefinido de veces, es decir que pueden no terminar.
  - *Definida*: como el clásico comando **for**, cuya variable de control permite que el cuerpo se ejecute un cierto número de veces[2].

- **Secuenciadores**: saltos (ej: `goto label`) y saltos computados (donde `label` puede ser una expresión la cual deberá ser evaluada dinámicamente. Estos últimos tipos de saltos se encuentran en Fortran, algunas versiones de BASIC y como extensión al lenguaje C en el compilador C de GNU (gcc).

  Las sentencias de salto generalmente restringen el destino a la abstracción procedural que las contienen, lo cual prohibe que se generen saltos intraprocedurales.

  Los saltos, aunque generalmente son sentencias prohibidas en la programación estructurada, son útiles en ciertos casos cuando se aplica programación defensiva (chequeos de argumentos y verificaciones de condiciones esperables) lo cual

[2] En el lenguaje C (y muchos de sus derivados), el comando `for` puede tomar la forma de un `while`.

produciría código difícil de leer (por los múltiples anidamientos de chequeos y posiblemente repetición de código) y poco eficiente.

Es común encontrar el uso del `goto` en código de sistemas operativos u otro software de base.

El uso más apropiado de sentencias tipo `goto` es como vía de escape a código de recuperación de errores.

Estas técnicas caen en desuso si el lenguaje provee excepciones.

Algunos lenguajes, como por ejemplo C y sus derivados, proveen sentencias de saltos estructurados, generalmente conocidos como *escapes*. Ejemplos de este tipo se sentencias son `break` y `continue`, los cuales sólo pueden aparecer dentro de determinados comandos, como `switch` (para salir de un caso) y en las iteraciones `while, do-while, for`. El comando `break` salta al final de la sentencia y el comando `continue` salta al comienzo (sólo en iteraciones).

Otra sentencia que puede considerarse un escape es `return` en varios lenguajes como C, Fortran, etc.

7.3 Excepciones

Las excepciones son sentencias estructuradas para atrapar (controlar el flujo de ejecución) en la ocurrencia de situaciones excepcionales, como excepciones generadas por el hardware (ej: división por cero, o error de escritura en un archivo) o por software (ej: por incumplimiento de una condición esperada o por el vencimiento de una alarma o timer).

Una forma muy común de las excepciones en muchos lenguajes de programación modernos es la siguiente:
try <*bloque*> **catch**(e1:T1) <h1> ... **catch**(en:Tn) <hn>
con el significado informal de que si ocurre una excepción en <*bloque*> de tipo `Ti`, se ejecuta el bloque (exception handler) `hi`, con $1 \leq i \leq n$.

Es posible que la sentencia soporte la definición de un manejador de excepciones por omisión (default), el cual generalmente toma la forma **finally** <*hd*>.

Las excepciones permiten una mejor estructuración del código en la programación defensiva, lo cual permite que el código sea mas legible sin perder eficiencia en la ejecución.

Si se produce una excepción durante la ejecución de un bloque que no atrapa la excepción, su ejecución se interrumpe y se retorna a su invocante, para luego repetirse el proceso. Esto se conoce como propagación de la excepción.
Como se puede apreciar, la propagación de una excepción sigue la cadena dinámica de invocaciones.

Si la propagación de una excepción alcanza al bloque inicial del programa, la excepción es atrapada por el manejador por omisión generado automáticamente por el intérprete o compilador, el cual generalmente, imprime un mensaje correspondiente a la excepción ocurrida y luego finaliza el proceso.

7.4 Introducción al lenguaje C

El lenguaje C fue muy popular desde su aparición en 1970, desarrollado por Dennis Ritchie en los laboratorios de AT&T.

Es un lenguaje que tiene características de bajo y alto nivel, por lo que se considera un lenguaje de nivel intermedio. El lenguaje C ha sido ampliamente utilizado para el desarrollo de software de base como sistemas operativos, compiladores e intérpretes de lenguajes de programación, y muchísimos utilitarios.

Generalmente es la base de muchos sistemas de computación, como por ejemplo en los sistemas tipo UNIX.

El nombre del lenguaje viene del hecho que Ritchie lo desarrolló como una extensión al lenguaje B.

AT&T desarrolló UNIX escrito casi completamente en C. Era la primera vez que se un sistema operativo se escribía en un lenguaje de programación de alto nivel. Hasta entonces, los sistemas operativos se escribían en assembly. La posibilidad de contar con un sistema operativo escrito en un lenguaje de alto nivel, tenía grandes ventajas ya que facilitaba su comprensión, mantenibilidad y portabilidad a diferentes arquitecturas.

En los comienzos de los 80 se formó el comité para la estandarización del lenguaje C, el cual se publicó el Standard ANSI C en 1989, el cual incluye los contenidos mínimos de su biblioteca estándar.

En 1990 la International Organization for Standarization, publicó el estándar ISO 9899-1990, el cual es prácticamente un duplicado del ANSI 89.

C ha influenciado a muchos lenguajes de programación desarrollados posteriormente, los cuales lo extienden incluyendo conceptos de programación orientada a objetos (POO) y en algunos casos eliminando algunas características no deseadas. Estos lenguajes se conocen como derivados de C y podemos mencionar *Objective C*, *C++*, *Java*, *C#* y otros.

7.5 Estructura de un programa C

Un programa C es una secuencia de declaraciones, definiciones y directivas del preprocesador.

Las declaraciones y definiciones pueden ser de variables, constantes, tipos o funciones.

```c
#include <stdio.h>

#define PI 3.141516

float circle_area(float radius)
{
    return PI * (radius * radius);
}

int main(void)
{
    float result;

    result = circle_area(10.0);
    printf("Area:%f\n",result);
    return 0;                   /* return process exit code */
}
```

Fig. 7.1: Ejemplo de un programa C

Las directivas del procesador pueden ser utilizados para incluir otros archivos fuente, definir constantes simbólicas y macros. Mas adelante se profundizará en el uso de las directivas del preprocesador.

En la figura 7.1 se puede ver la estructura de un programa C.

Este contiene la definición de una constante simbólica por medio de la directiva *define* del preprocesador y la definición de dos funciones: la función

circle_area : $float \rightarrow float$

la cual computa la superficie de un círculo, tomando su radio como argumento, y la función *main* (principal), que invoca (llama) a la función *circle_area* y muestra su valor retornado por la salida estándar[3]

La función principal (`main`), por donde el programa inicia, está definida como que retorna un *integer*, tal como se sugiere en los sistemas *POSIX*[4] compatibles. El valor retornado por *main* es tomado por el sistema como el código de finalización del proceso (el cual puede ser consultado por el proceso que lo creó).

La palabra *void* en el argumento de la función *main* indica que la función no tiene argumentos. La palabra *void* usada como tipo de retorno en una función significa que no retorna valor alguno (es un *procedimiento*).

Como se puede apreciar en la figura 7.1, una función tiene asociada una sentencia de bloque delimitada por llaves ({,}).

Dentro de un bloque es posible realizar declaraciones (de variables, constantes, etc) y a continuación se definen las acciones (sentencias) a ejecutar en el bloque.

[3] La salida estándar se corresponde comúnmente a la consola, aunque se puede redirigir.

[4] Estándar que define una interface (system calls) de un sistema operativo. Esta interface está definida en C

Cada sentencia debe finalizar con un punto y coma (;).

La directiva del preprocesador *include* utilizada incluye el archivo de cabecera *stdio.h*, el cual es provista por cualquier compilador ANSI C, contiene las declaraciones necesarias para realizar entrada-salida.
La función *printf* permite imprimir (con formato) valores por la salida estándar. Mas adelante se entrará en mayor detalles con respecto a la entrada-salida en C.

El lenguaje C no contiene primitivas de entrada-salida. Todas las operaciones de entrada-salida se realizan por medio de funciones de biblioteca.

7.6 El compilador C

El conjunto de herramientas en un entorno de desarrollo C básicamente contiene:

1. el pre-procesador: toma las directivas en el archivo fuente y las reemplaza por su evaluación (o expansión). Por ejemplo, en el programa de la figura 7.1 cada referencia a *PI* es reemplazada (expandida) a 3.141516.

2. el compilador C: genera un archivo con código assembly a partir de un archivo fuente C.

3. el *assembler*: genera un archivo *objeto* a partir de un archivo fuente en lenguaje *assembly*. Un archivo objeto es un archivo binario no directamente ejecutable. Puede ser utilizado por el *linker* para generar un programa (archivo ejecutable). Mas adelante se entrarán en mayores detalles sobre estos conceptos.

4. el *linker*: combina archivos objetos y bibliotecas para generar un archivo ejecutable (programa).

5. el *debugger*: permite ejecutar un programa para depuración, esto es, permite definir puntos de parada (breakpoints), ver valores de variables durante la ejecución, modificarlos, etc.

6. otras (make, libtool, ...): son herramientas para el control de la compilación (ej:make) y generación de bibliotecas (ej:libtool). La herramienta `make` dirige el proceso de compilación en base a una especificación (Makefile), en la cual se describen las dependencias entre los diferentes módulos de un programa y las acciones a tomar cuando algunas de esas dependencias requiera actualizarse (ej: un archivo objecto con respecto a su archivo fuente).

7.7 Compilación de un programa

Si bien esto depende del compilador y del entorno de desarrollo utilizado, aquí se asumirá que se compilará desde una consola o terminal corriendo un intérprete de comandos (shell) del sistema operativo.

Todos los ejemplos de compilación asumirán que se dispone del compilador C libre de la colección de compiladores de GNU[5] el cual incluye un compilador C.

El programa de la figura 7.1, asumiendo que se encuentra en el archivo fuente *example1.c*, se puede compilar mediante el comando

```
gcc example1.c
```

Esto generará (en caso que no existan errores en el programa) un programa (archivo ejecutable) *a.out*, el cual se podrá ejecutar introduciendo el comando

```
./a.out
```

lo que indicará al shell que ejecute el programa *a.out* que se encuentra en el directorio corriente (./).

La opción del compilador (flag) *-o <filename>* permite que el programa de salida tome otro nombre que *a.out*. El siguiente ejemplo muestra la compilación del programa anterior que genere el ejecutable con el nombre de programa *example1*

```
gcc example1.c -o example1
```

Si se encuentra en la plataforma MS-WINDOWS el programa de salida deberá tener extensión *.exe*.

En los sistemas tipo UNIX (ej: GNU/Linux) se puede obtener mas información sobre las opciones del compilador mediante el comando

```
man gcc
```

(man es la abreviatura de *manual*).

7.8 El pre-procesador

El compilador **C** tiene un componente auxiliar llamado pre-procesador, que actúa en la primera etapa del proceso de compilación. Su misión es buscar, en el texto del programa fuente entregado al compilador, ciertas directivas que le indican realizar alguna tarea a nivel de texto. Por ejemplo, inclusión de otros archivos, o sustitución de ciertas cadenas de caracteres (símbolos o macros) por otras. El preprocesador cumple estas directivas en forma similar a como podrían ser hechas manualmente por el usuario, utilizando los comandos de un editor de texto ("incluir archivo" y "reemplazar texto"), pero en forma automática. Una vez cumplidas todas estas directivas, el preprocesador entrega el texto resultante al resto de las etapas de compilación, que terminarán dando por resultado un módulo objeto.

El pre-procesador procesa (expande o evalúa) directivas que se distinguen de otras sentencias del lenguaje C porque están precedidas por el símbolo # (numeral).

En la tabla 7.2 describen algunas directivas del pre-procesador.

[5] La fundación GNU tiene como objetivo el desarrollo de un sistema operativo y aplicaciones *libres*, es decir sin restricciones en las licencias sobre su uso (http://www.gnu.org/).

Directiva	Descripción
#include file	Inserta el contenido de *file*.
#define *def*	Define un símbolo o una macro.
#ifdef *symbol* b_1 #else b_2 #endif	Expande b_1 si *symbol* está definido, b_2 en otro caso.

Fig. 7.2: Algunas directivas del pre-procesador.

En la directiva `#include`, *file* puede encerrarse entre ángulos (< y >), lo cual hace que el archivo se busque en los caminos (paths) pre-definidos o dados en el comando de compilación (opción -I en el compilador gcc). Si *file* se encierra entre comillas dobles ("), se busca en el directorio corriente.

La directiva `#define` permite definir símbolos o macros. Una macro es una expresión parametrizada a cual será expandida en cada ocurrencia en el programa por su definición, realizando una sustitución textual de los parámetros formales por los actuales.

A modo de ejemplo, el siguiente fragmento de programa, muestra su utilización.

```
...
#define MAX_SIZE 1024
...
#define max(A,B) ((A>B)?A:B)
...
  int x,y,z;
  float delta,f,g;
  ...
  x = max(y,z);      /* x = ((y>z)?y:z) */
  delta = max(f,g)   /* delta = ((f>g)?f:g) */
...
```

Cabe hacer notar que una macro no es equivalente a una función. Una función es tipada (tipo de su resultado y sus argumentos), mientras que una macro no. A menudo los programadores C utilizan macros para definir expresiones las cuales serán utilizadas sobre diferentes tipos de datos, lo cual permite implementar una forma de polimorfismo paramétrico ad-hoc (basado en sustitución textual).

7.9 Tipos de datos básicos

El lenguaje C es un lenguaje estáticamente tipado, es decir que se realiza el chequeo de los tipos de los valores involucrados en cualquier operación en tiempo de compilación.

C ofrece un conjunto de *tipos de datos básicos* que se muestran en la tabla 7.3. Mas adelante verá el tipo de datos *puntero*, el cual también es un tipo básico, pero al que se dedicará una sección específica.

Tipo	Descripción
char	caracteres (generalmente representados en un byte)
int	enteros (16, 32 o 64 bits, depende de la plataforma)
short int	entero corto
long int	entero largo
float	real (representación en punto flotante)
double	real de doble precisión

Fig. 7.3: Tipos de datos básicos

Además de los nombres de los tipos básicos mencionados, éstos pueden ser precedidos por la palabra reservada **unsigned**, lo cual significa que no pueden tener valores negativos.

Cabe hacer notar que el tipo *char* en C es considerado un tipo numérico (generalmente se representa en un byte) y sus valores generalmente se corresponden a algún código de caracteres estándar (ej: ASCII).

7.10 Declaraciones y definiciones

En el lenguaje C pueden existir diferentes tipos de declaraciones y definiciones.

Una *definición* introduce una nueva entidad, mientras que una *declaración* es un concepto más general que puede involucrar una definición.

7.11 Definiciones de variables

Una declaración puede introducir (definir) variables de un determinado tipo. Una declaración de variables tiene la forma:

```
<type> id1 [=value], id2 [=value], ... ;
```

donde *id1*, *id2*, ..., son los identificadores (nombres) de las variables, <type> es su tipo (ej: *int, char, ...*)[6]. Se puede dar un valor inicial a una variable.

A modo de ejemplo, las siguientes declaraciones definen dos variables de tipo *entero*, una de tipo *char* y otra de tipo *float*.

```
int x,y;
char c;
float f = 3.14;
```

Se debe notar que sólo la variable f está inicializada.

[6] La notación <x> denota que x es una categoría sintáctica (los símbolos menor y mayor no deben escribirse). Lo que aparecen entre corchetes, significa que es opcional.

7.12 Definiciones de constantes

Se introduce una constante antecediendo a una declaración como la de la sección anterior la palabra reservada **const**.

Ejemplo:

```
const float Pi = 3.14;
```

Algunos programadores prefieren declarar valores constantes por medio de la directiva del preprocesador *#define*. La diferencia fundamental es que una constante definida como una variable **const** ocupa memoria mientras que una directiva no.

El modificador *const* es mas utilizado en la declaración de parámetros formales para indicar que el cuerpo de la función no debe modificar el argumento. El compilador puede detectar un intento de modificación del argumento y en ese caso arrojará un error de compilación.

7.13 Definiciones de tipos

C permite definir nuevos tipos a partir de los tipos básicos. Es posible definir *enumeraciones* las cuales son constantes simbólicas (de tipo entero).

Ej: `enum` dia_sem = { dom, lun, mar, mie, jue, vie, sab };

La declaración anterior define un nuevo tipo de dato, aunque C toma sus valores como enteros. Las constantes simbólicas de una enumeración toman valores por omisión (default) comenzando desde cero (0). Es posible definir el valor de cada constante o dar el valor de la primera y así las demás toman valores subsiguientes.

Ej: `enum` dia_sem = { dom=1, lun, mar, mie, jue, vie, sab };

Las definiciones de estructuras pueden introducir nuevos nombres de tipos. A modo de ejemplo, la siguiente declaración

```
struct person {
  int id;
  char name[31];
  unsigned short age;
};
```

define el tipo `struct Person`.

La sentencia `typedef type name;` define un nuevo nombre (sinónimo) de un tipo.

A modo de ejemplo, `typedef struct person Person;` define el tipo `Person` como un sinónimo o alias del tipo `struct person`[7].

[7] Los símbolos `person` y `Person` son diferentes, ya que C distingue mayúsculas de minúsculas.

7.14 Funciones

La definición de una función tiene la forma

`return_type function_name(`*`arguments`*`)` *`body`*

donde *`arguments`* es una lista de argumentos de la forma `type id`, separados por coma o `void` en el caso de una función sin argumentos. Los procedimientos deben usar `void` como tipo de retorno.

El cuerpo de una función es una sentencia de bloque.

En las expresiones pueden aparecer invocaciones a funciones, las cuales toman la forma

`f(`arg_1`, ..., `arg_n`)`, donde `f` es el nombre de una función.

En el caso que la función no tenga argumentos se invocará como `f()`.

La sentencia `return` *`expression`* en el cuerpo de una función, finaliza su activación, retornando el valor denotado por *expression*. En el caso de procedemientos (funciones con tipo de retorno `void`), la expresión se omite.

Cuando se alcanza el final del bloque que constituye el cuerpo de una función se ejecuta un `return` implícito.

Las funciones pueden tener un número variable de argumentos, los cuales se denotan con ... (tres puntos seguidos). Esto permite definir funciones como por ejemplo `printf`, cuyo perfil es

```
int printf(const char * format, ...)
```

La biblioteca estándar de C provee tipos de datos (`va_list`) y funciones (`va_start`,`va_arg`,`va_end`) para acceder a los argumentos.

Las funciones como `printf` se basan en el primer argumento (`format`) para acceder y manipular los restantes argumentos.

7.15 Alcance de las declaraciones

Una declaración en el ambiente global se denomina una *declaración global* y las entidades (variables, constantes, ...) se denominan *globales*.

Un bloque en la definición de una función C consta de dos partes: al comienzo puede contener declaraciones y luego puede contener sentencias y/o comandos. No se puede pueden mezclar declaraciones y otras sentencias.

Las declaraciones contenidas en un bloque *sentencia { ... }* , tienen alcance *local al bloque*, es decir que fuera del bloque las entidades declaradas no pueden ser referenciadas.

El ejemplo 7.4 muestra declaraciones locales y globales en un programa. La variable g es global, es decir que puede referenciarse en cualquier parte del programa. La

```
int g;           /* variable global */

int f(float p)
{
    char c;      /* c es una variable local a la funcion f */

    ...
    {
        int x;   /* x es local a este bloque */
                 /* aqu\'i pueden referenciarse todas las variables */
    }
    g++;
    x = 1;       /* ERROR, x no existe en este ambito */
}
```

Fig. 7.4: Ejemplo de declaraciones locales y globales

variable local *c* y el *parámetro formal p* son locales en la función *f*. La variable *x* es sólo visible dentro del bloque mas interno.

Esta separación de *ámbitos o ambientes* es de gran ayuda para modularizar programas, permitiendo que sólo se declaren variables en los ámbitos donde son necesarios, sin permitir el acceso en ámbitos que no deberían conocer de su existencia.

Las declaraciones globales (variables, constantes y funciones) pueden ser usadas en un módulo externo (cliente) mediante declaraciones *extern*.

Una declaración externa hace referencia a una entidad que ha sido definida en otro módulo. Ejemplo: `extern void stack_push(Stack * s, int item);` .

Una declaración global que se quiera hacer visible sólo en el módulo corriente (archivo fuente), deberá ser precedida por *static*.

Las declaraciones de tipos no se pueden hacer externas, es decir que en un archivo que se quiera referenciar a un tipo de dato, se deberá incluir su definición, ya sea escribiendo la definición o incluyendo un archivo de cabecera (.h) que la contenga.

En la sección 7.22 se describe la modularización de programas C.

7.16 Tiempo de vida de las entidades

Las variables, como entidades que requieren almacenarse en la memoria principal de la computadora, tienen asociado un tiempo de vida, es decir un intervalo de tiempo entre su creación y su destrucción.

Es razonable que el tiempo de vida de una variable global sea desde el momento en que el programa se inicia (se convierte en un proceso del sistema) hasta que el mismo

finalice. Es decir que una variable global tiene un tiempo de vida que se corresponde con el tiempo de ejecución del programa.

En general, los valores de las variables globales se almacenan en una área de memoria estática la cual se reserva al comienzo del programa y al finalizar se libera.

Por el contrario, las variables locales a un bloque no tendría sentido que estén presentes en la memoria cuando la función a la cual pertencen no está activa, es decir no está siendo ejecutada (ha sido *invocada*) actualmente.

Cuando una función retorna (finaliza su activación) sería razonable que todas las entidades declaradas localmente sean destruidas, es decir liberada la memoria en donde se almacenaban sus valores, lo cual permitiría reutilizar esas áreas de memoria para contener valores de otra posible función a activarse.

Este es el comportamiento normal de las variables locales y de los parámetros formales de una función. Esta es la razón principal por lo que los lenguajes de este tipo (con diferentes ambientes), manejen a la memoria donde se almacenan los valores para las variables locales y parámetros formales como una estructura de pila (*stack*)[8].

Cuando una función es invocada, se crea (reserva) en la memoria (tope de la pila) un *registro de activación*, el cual contiene suficiente espacio para contener los valores de sus parámetros formales y sus variables locales[9].

Cuando una función retorna (se ejecutó una sentencia **return** o se alcanzó el final del bloque principal de la función), el registro se destruye por lo que las variables locales y los parámetros formales "desaparecen".

7.16.1 Cambiando el tiempo de vida de variables locales

En C, la palabra reservada **static** indica, en una declaración local que las entidades apareciendo en la declaración permanezcan "vivas" luego del retorno de la función.

De todos modos dichas entidades no pueden ser referenciadas fuera de su bloque correspondiente, es decir que su alcance no es modificado, pero sus valores permanecen en la memoria para poder reutilizarse en la próxima activación de ese bloque.

Estas variables generalmente no se almacenan en la pila (stack), sino que se reserva espacio en el área de memoria estática en donde se almacenan las entidades globales.

Cabe hacer notar que *static* tiene dos significados diferentes:

- Cuando se aplica a declaraciones globales afecta a su visibilidad, es decir que significa que no se exporta (no es visible fuera del módulo).
- Cuando se aplica a variables locales, afecta a su tiempo de vida.

[8] Una estructura de pila tiene una política LIFO (Last In First Out, el último en entrar es el primero en salir).

[9] Además el mismo registro de activación puede usarse para almacenar valores temporarios durante la evaluación de alguna expresión.

7.17 Operadores

Los denominados *operadores* de un lenguaje de programación son la base de la formación de las expresiones. Entre ellos generalmente encontramos operadores aritméticos, relacionales, lógicos, etc.

Un operador puede estar definido como *infijo* (ej: $x < y$), *prefijo* (ej: $-x$) o *sufijo* (ej: $x++$).

La siguiente tabla muestra algunos de los operadores de C:

Aritméticos	**Descripción**
+	adición de enteros y reales (infijo)
-	substracción de enteros y reales (infijo)
*	multiplicación de enteros y reales (infijo)
/	división de enteros y reales (infijo)
%	resto (modulo) de enteros (infijo)
Manipuladores de bits	**Descripción**
<<	dezplazamiento a izquierda de bits (infijo)
>>	dezplazamiento a derecha de bits (infijo)
|	or (bit a bit, infijo)
&	and (bit a bit, infijo)
Relacionales	**Descripción**
==	igual (infijo)
!=	distinto (infijo)
<	menor (infijo)
>	mayor (infijo)
<=	menor o igual (infijo)
>=	mayor o igual (infijo)
Lógicos	**Descripción**
||	or (infijo)
&&	or (infijo)
!	not (prefijo)

7.17.1 Asignación

C es un lenguaje con un conjunto de sentencias de asignación muy rico, a diferencia de otros lenguajes que sólo contienen unas pocas o una única sentencia de asignación.

En el modelo de la programación imperativa la asignación es generalmente la única sentencia que reamente produce una computación efectiva, ya que es la única sentencia que realiza un cambio de estado durante la ejecución de un programa.

Otro caso en donde se produce un cambio de estado es en el pasaje de parámetros, lo que se verá mas adelante.

En C las sentencias de asignación son expresiones (operadores), es decir que a su vez denotan un valor, lo que permite al programador realizar composición de asignaciones.

A continuación se muestra una tabla con algunos de los operadores de asignación en C:

Operador	Descripción
=	asignación (ej: x = y + 1)
+=	acumulación (ej: x += y es equivalente a x = x + y)
-=	(ej: x -= y es equivalente a x = x - y)
*=	(ej: x *= y es equivalente a x = x * y)
/=	(ej: x *= y es equivalente a x = x / y)
%=	(ej: x %= y es equivalente a x = x % y)
x++	pos-incremento (ej: x = y++; asigna y a x y luego incrementa y)
x–	pos-decremento
++x	pre-incremento (ej: x = y++; incrementa y, luego asigna y a x)
–x	pre-decremento

7.17.2 Expresiones condicionales

Una expresión condicional permite denotar un valor el cual será computado dependiendo de una condición.

En C una *expresión condicional* sigue el siguiente esquema sintáctico:

```
( <condici\'on> )? <expr1> : <expr2>
```

la cual significa que si la condición evalúa a verdadero, arroja el resultado de la evaluación de *expr1*, sino arroja el resultado de la evaluación de *expr2*.

Ejemplo:

```
x = ( y > 0 )? y+1 : z;
```

7.17.3 Otras expresiones

En C las expresiones pueden contener constantes, variables, operadores y llamados a funciones, estas últimas, ya sean que se encuentren en la biblioteca estándar o que sean definidas en el mismo programa.

Ejemplo de una expresión:

```
a = x = ( y > 0 )? y+1 : f(z) * 2;
```

7.18 Sentencias de control: comandos

Los comandos afectan al flujo de ejecución del programa. El lenguaje C contiene sentencias de control estructurados, es decir que las sentencias tienen una semántica de control especificado, por lo que el control de flujo (saltos) están implícitos[10].

Las sentencias de control control de flujo, generalmente se clasifican en sentencias secuenciales, condicionales, iterativas y otras.

[10] Aunque C mantiene el comando **goto L;** el cual realiza un salto explícito.

7.18.1 Secuencia

La sentencia que establece que sus componentes deben ejecutarse en secuencia es la sentencia de bloque que ya se ha visto con anterioridad.

Tanto las declaraciones como los comandos que deben ir a continuación se ejecutan en forma secuencial de arriba hacia abajo.

Existen sentencias las cuales pueden contener otras sentencias. Generalmente, las reglas sintácticas son que contienen una sentencia (la cual puede ser un bloque).

7.18.2 Sentencias condicionales

Una sentencia condicional permite ejecutar diferentes grupos de sentencias en base a una condición dada.

En C encontramos dos tipos de sentencias: la sentencia **if** y la sentencia **switch**. La primera permite ejecutar sentencias en base al valor de verdad de una condición dada (expresión booleana), mientras que la segunda permite ejecutar diferentes sentencias dados diferentes casos determinados por el valor de una expresión.

El esquema sintáctico de la sentencia **if** es la siguiente:

```
if ( <condici\'on> )
    <sentencias por verdadero>
else
    <sentencias por falso>
```

donde la cláusula *else* es opcional.

Un ejemplo:

```
if ( x == y ){
    x = y * z;
    ...
}
else
    x = y;
```

El esquema sintáctico de la sentencia **switch** es la siguiente:

```
switch ( <condici\'on> ) {
    case <valor1>: <sentencias caso 1>
                   break;
    case <valor2>: <sentencias caso 2>
                   break;
    default:
        <sentencias por ninguno de los valores anteriores>
}
```

donde la cláusula *default* es opcional y los valores *valor1*, *valor2*, ..., deben ser constantes de tipo entero.

Si en cada caso se omite la sentencia **break**, la ejecución continuará con las sentencias del caso siguiente, si hubiera.

Ejemplo:

```
switch ( x-y ) {
    case 5: x = y - 1;
            break;
    case 8: y = z;
            break;
}
```

7.18.3 Sentencias de iteración

Las sentencias e iteración permiten que un conjunto de sentencias sean ejecutadas en forma repetitiva.

7.18.3.1 Iteración definida

Una sentencia de iteración definida expresa que se debe repetir la ejecución de su cuerpo un número determinado de veces. La única sentencia de este tipo en C es el **for**.

A continuación se da el esquema sintáctico del **for**:

```
for ( <inicializaci\'on> ; <condici\'on>; <paso> )
    <sentencia>
```

La semántica del **for** es: se ejecuta la inicialización, luego se verifica la condición, si es verdadera, se ejecuta su cuerpo (*sentencia*), finalmente se ejecuta el paso y se repite el ciclo a partir de la verificación de la condición.

La sentencia **for** de C en una de las mas complicadas de describir ya que tanto en la inicialización como en la condición y en el paso, pueden escribirse una secuencia de expresiones arbitrarias separadas por coma, lo que la convierte, en realidad, en una sentencia de iteración indefinida, rigurosamente hablando.

También es posible que los tres componentes estén vacíos. Una condición vacía es evaluada a verdadero.

Se aconseja, por razones de legibilidad y para evitar errores, utilizarla al estilo de sentencias **for** como se encuentran en otros lenguajes, como por ejemplo, Pascal.

Ejemplo:

```
for (i=0; i<N; i++)
    x += i;
```

7.18.3.2 Iteración indefinida

C ofrece dos sentencias de iteración indefinida.

La sentencia **while** y la sentencia **do ... while**.

A continuación se muestran los esquemas de su sintaxis:

```
while ( <condicion> )
    <sentencia>
```

y

```
do {
    <sentencia>
} while ( <condicion> );
```

La primera (**while**) evalúa la condición y si arroja verdadero, se ejecuta su cuerpo.

La segunda **do ... while**, primero ejecuta su cuerpo para luego evaluar su condición. Mientras la condición evalúe a verdadero, se continuará ejecutando el cuerpo.

Se debe notar que la primera permite ejecutar su cuerpo cero o más veces, mientras que la segunda permite ejecutarlo uno o más veces.

ejemplos:

```
while ( x > 0 ) {
    a = 1;
    b = 2 * x;
}
...
do {
    x++;
    y--;
} while ( x < 100 && y > 10);
```

7.19 Tipos de datos estructurados

Además de los tipos básicos vistos en el capítulo anterior, el lenguaje C permite definir tipos de valores *estructurados*, es decir que son colecciones o agrupaciones de otros datos.

7.19.1 Arreglos

Un arreglo es una estructura homogénea[11] del elementos almacenados en forma contig'ua en la memoria.

Si bien algunos lenguajes permiten definir arreglos de más de una dimensión, C sólo permite definir *vectores* (arreglos de una dimensión).

[11] Es decir que todos sus componentes don del mismo tipo.

Esto en general no es un problema ya que un arreglo multidimensional se puede simular con vectores de vectores, es decir definiendo un vector donde cada uno de sus elementos es otro vector.

Un vector en C se define de la siguiente manera:

```
<type> vec[N];
```

donde <*type*> es algún tipo básico o estructurado y N es una constante o variable de tipo entero, la cual define su *dimensión* (el número de componentes). El siguiente ejemplo define una vector de 10 reales en punto flotante de doble precisión:

```
double vec[N];
```

Cuando se declara un vector en un parámetro formal de una función, su dimensión se omite.

Se debe notar que la dimensión de un vector puede definirse en tiempo de ejecución, ya que C permite que sea una variable y su valor en un momento específico sólo puede ser determinado durante la ejecución. Otros lenguajes (como FORTRAN 77 o Pascal estándar) requieren que su dimensión sea determinada en tiempo de compilación por lo que requieren que sea una constante.

Si bien su dimensión puede determinarse dinámicamente (tiempo de ejecución), una vez creado, un vector no puede cambiar su dimensión. Los vectores de este tipo se conocen como *semidinámicos*.

Cada elemento de un vector se puede acceder por medio de su *índice* en el vector. El índice es una especie de identificador (o nombre) unívoco de cada elemento. El primer elemento de un vector tiene índice 0 (cero) y el útimo $N-1$, (donde N es si dimensión).

Es responsabilidad del programador no intentar acceder a elementos con un índice inválido. Si este fuese el caso, estaríamos ante la presencia de un acceso fuera de rango (out of bound). C no detecta durante la ejecución los accesos fuera de rango, por lo que si esto sucediera el programa puede continuar su ejecución aunque seguramente realizará computaciones no esperadas.

El siguiente ejemplo muestra la definición y uso de un vector:

```
#include <stdio.h>
#define N 100

/* returns the sum of elements of vector v */
double sum_vector(const double v[])
{
    int i;
    double sum = 0.0;

    for (i=0; i<N; i++)
        sum += v[i];
```

```
    return sum;
}

void main(void)
{
    double vector[N];       /* definition of vector v */

    /* initialization of vector ... TO DO */

    printf("Suma=%10.2d\n", sum_vector(vector));
}
```

Es posible declarar una matriz (arreglo de dos dimensiones) adicionando la otra dimensión en la declaración. Ejemplo:

```
double v[N][M];
```

lo cual declara un vector de dimensión N, en el cual, cada elemento es un vector de M elementos. Esto se puede ver como una matriz de NxM (N filas por M columnas).

Lo anterior se extiende para definir arreglos multidimensionales.

El siguiente ejemplo muestra la definición y la inicialización de una matriz:

```
#include <stdio.h>
#define N 10
#define M 20

void main(void)
{
    double vec[N][M];
    int i,j;

    for (i=0; i<N; i++)
        for(j=0; j<M; j++)
            vec[N][M] = 0.0;
}
```

7.19.2 Estructuras

Las estructuras (o registros) se caracterizan por ser contenedores heterogéneos, es decir sus componentes pueden ser de distinto tipo. Generalmente se utilizan para describir propiedades de entidades.

Sus componentes de denominan *campos (fields)* y se accede a cada uno de ellos por su identificador (nombre del campo).

El siguiente ejemplo muestra la definición y uso de una estructura:

```
...
struct persona {            /* declaracion del tipo struct persona */
    unsigned long dni;
```

```
    char nombre[31];
    unsigned short edad;
};

struct persona p;           /* p es una variable de tipo struct persona */

...

    p.dni = 24456789;       /* asignacion de valores a campos ... */
    p.edad = 31;
    ...
```

7.19.3 Uniones disjuntas

Una unión disjunta (o variante) permite definir campos que sólo uno de ellos es válido en un determinado momento en la ejecución de un programa.

Son útiles para acceder o ver un dato de diferentes formas.

Si uno de sus campos se modifica, los demás también, ya que en realidad se está refiriendo al mismo valor, es decir en la memoria se ha reservado tanto espacio como requiere su componente de mayor tamaño en su representación.

A modo de ejemplo, a continuación se muestra un caso en que es posible acceder a cada byte de un entero de 32 bits, lo cual serviría para ver cómo es su representación en memoria:

```
union u {
    int value;
    unsigned char bytes[4];
};

u rep_int;
...

    rep_int.value = -1;
    for (i=0; i<4; i++)
        printf("byte[i]=%d%n", rep_int.bytes[i]);
...
```

Como se puede apreciar en el ejemplo, los campos de una unión se acceden de la misma manera que los de una estructura.

7.20 Punteros

Los punteros son datos básicos del lenguaje. Sus valores representan direcciones de memoria de otras entidades (variables o valores). El tipo de datos puntero es un tipo parametrizado ya que son tipados, esto es, un puntero apunta a entidades de un tipo

determinado.

La sintaxis de la declaración de un puntero tiene la forma `T * ptr`, donde `T` es el tipo de los valores apuntados por el puntero `ptr`.

Los punteros tienen dos operadores asociados:

- *referenciación (operador &)*: el operador prefijo & aplicado a una variable, retorna su dirección de memoria, la cual puede ser asignado a pun puntero de un tipo compatible con dicha variable.
- *desreferenciación (operador *)*: el operador (prefijo) `*` aplicado a un puntero `p` retorna el valor apuntado por `p`.

El siguiente ejemplo muestra el uso de punteros:

```
...
{
   int v = 5; int * p1 = &v;
   int *p2;

   p2 = p1;      /* asignacion (copia) entre punteros */
   (*p1)++;        /* el valor de v es incrementado (6) */
   printf("El valor de v es: %d\n", *p2);
}
...
```

7.20.1 Vectores y punteros

Como ya se vio anteriormente, C permite definir vectores semidinámicos, es decir que su dimensión puede determinarse en tiempo de ejecución. Esto trae algunos problemas de representación ya que el compilador no podría generar código para reservar espacio en la memoria ya que no conoce la cantidad de memoria requerida en tiempo de compilación.

Una implementación posible es hacer que un vector está descripto en tiempo de ejecución por un puntero a su dirección base (su primer elemento). Con el fin de conseguir eficiencia y simplicidad, dicho descriptor no contiene información en runtime sobre su dimensión, por lo cual el manejo de los vectores (y no acceder a elementos fuera de rango), queda bajo la total responsabilidad del progamador.

A continuación se muestra un fragmento de programa de ejemplo ilustrando una situación con vectores semidinámicos.

```
...
int avg(float v[], int dim)
{
  int i; float acum = 0.0;
```

```
  for (i=0; i<dim; i++)
    acum += v[i];
  return acum / dim;
}
...
{
  float vec[N];

  a = avg(vec,N);
  ...
}
```

El programa de arriba merece algunas aclaraciones.

- la declaración del parámetro formal `float v[]` es equivalente a `float *v`.
- la dimensión del parámetro actual tiene que ser pasado como parámetro, ya que el parámetro formal no tiene información sobre su dimensión.

La figura 7.5 muestra una forma posible de representación de la memoria de la declaración de la forma `T v[N]`.

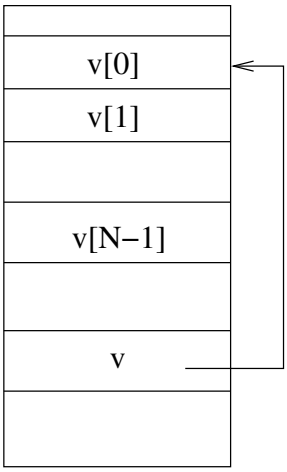


Fig. 7.5: Representación en memoria de un vector (en el stack).

El programa siguiente es equivalente al programa anterior.

```
...
int avg(float *v, int dim)
{
  int i; float acum = 0.0;
```

```
  for (i=0; i<dim; i++)
    acum += *(v+i);
  return acum / dim;
}
...
{
  float vec[N];

  a = avg(vec,N);
  ...
}
```

Un análisis de este último programa muestra que:

- la expresión `v[i]` es equivalente a `*(v+i)`.
- es posible realizar operaciones aritméticas sobre punteros.

La aritmética sobre punteros no es equivalente a la aritmética sobre enteros. Los arreglos tienen en la memoria una representación contig'ua. El puntero `v` apunta a su dirección base. Podría pensarse que la expresión `v+1` suma uno a la dirección almacenada en `v`. Si fuera así, la expresión `*(v+i)` se refereriría a un valor con la dirección base del vector mas uno, lo cual no sería una referencia a un valor del vector, ya que un entero generalmente se representa en más de un byte (es común que en una arquitectura de 32 bits se represente en 4 bytes).

Este ejemplo muestra que la expresión `v+i` debería significar, en aritmética de enteros, `address(v)+i*sizeof(int)`, es decir, la suma de la dirección base de `v` con el valor de `i` multiplicado por el tamaño (en bytes) de la representación del tipo base del puntero.

Efectivamente la aritmética de punteros tiene en cuenta el tamaño de su tipo base. El programa anterior podría recorrer el vector utilizando solamente punteros, tal como se muestra a continuación.

```
...
int avg(float *v, int dim)
{
  int *p; float acum = 0.0;

  for (p=v; p < v+dim; p++)
    acum += *p;
  return acum / dim;
}
...
{
  float vec[N];

  a = avg(vec,N);
  ...
```

```
}
```

Esto muestra que hay una relación muy estrecha con los vectores y punteros.

- Una variable de tipo vector de elementos de tipo T se representa como un bloque contig'uo de elementos de tipo T mas un puntero (constante) a la dirección base del bloque.
- La diferencia entre un puntero y una variable de tipo vector es que la variable es un puntero constante, es decir, que no puedo modificar su dirección base, cuando un puntero común es modificable.

Esta decisión de representación tiene sus ventajas y desventajas. Como ventajas podemos mencionar su simplicidad y eficiencia tanto en representación como en acceso a componentes. Como desventajas es fácil de ver que al no disponer de la dimensión en el descriptor de un vector, su manejo queda bajo la total responsabilidad del programador.

Esto es inconveniente, ya que las funciones que toman como parámetros vectores, su declaración en sí no es clara (ej: f(int *v) /* es un puntero o un vector? */). Los strings, que son vectores de tipo `char`, sufren estas mismas consecuencias.

Los strings, al ser vectores, no son tratados como los tipos básicos, esto es, no pueden asignarse (se tiene que hacer elemento a elemento) y generalmente se manipulan mediante funciones de biblioteca (ver la interface del archivo de cabecera `string.h`.

Otra desventaja de este manejo de los vectores es que los programadores novatos en el lenguaje a menudo se confunden que la declaración de un puntero define un vector.

La representación de los vectores de C es muchas veces la causa de problemas de seguridad en muchas aplicaciones de red, como lo son los ataques por desborde de buffers (buffer-overflow) que se basan en hacer que cierto código del programa copia datos fuera del rango de algún arreglo.

Esto permite desarrollar técnicas de *inyección de código*.

7.20.2 Punteros a funciones

Una función `f` de la forma `T f(args)` tiene el tipo `T (*f)(args)`.

Esto significa que el nombre de una función es un puntero (constante), lo que permite usar a los nombres de las funciones como valores en expresiones (involucrando punteros).

A continuación se muestra un ejemplo del uso de punteros a funciones.

```
...
int square(int n)
{
  return n*n;
}
```

```
int main(void)
{

  int (*ptr_square)(int) = square;
  int sq;

  sq = ptr_square(5);
  ...
}
```

7.21 Manejo de memoria dinámica

C no tiene incorporado nativamente en el lenguaje operadores para el manejo de memoria dinámica (heap). El heap es manipulado por funciones de la biblioteca estándar, cuya interface está definida en el archivo de cabecera `malloc.h`.

La biblioteca estándar del lenguaje provee dos funciones para manipulación de variables heap.

- *Reserva de bloque (allocation)*: la función `void * malloc(size_t s)` reserva un bloque y retorna un puntero a su dirección base (el tipo `size_t` generalmente está definido como `unsigned int`).
- *Liberación de un bloque (deallocation)*: la función `void free(void p)` libera el bloque apuntado por el puntero `p`.

El siguiente programa muestra un ejemplo de su uso.

```
{
  int *p = (int *) malloc(sizeof(int));
  float *vec = (float *) malloc(sizeof(float));
  ...
  vec[i] = ...;
  free(vec);
  free(p);
}
```

Cabe notar que, como la función `malloc` retorna un puntero a `void`, es necesario hacer un cambio de tipo (cast) para que el compilador, al chequear los tipos no reporte un error. [12]

7.22 Estructuración de programas: módulos

Cuando se escriben programas grandes, éstos generalmente se descomponen estructuralmente en módulos. Un módulo generalmente define una interface, la cual contiene

[12] Notar que no es necesario convertir un puntero de tipo T a un puntero a `void`, pero sí a la inversa. Esto muestra que el tipo `void` representa un tipo unión de todos los tipos definibles.

declaraciones de sus componentes y los exporta al exterior para otros módulos que lo usen (clientes).

Un módulo contiene una descripción de su interface y su implementación. Algunos lenguajes ofrecen construcciones sintácticas para la definición de módulos (clases de Haskell, Units de object Pascal o Delphi, etc).

El lenguaje C no ofrece ninguna sintaxis para la definición de módulos. Un módulo se corresponde con una unidad de compilación (archivo objeto o biblioteca). Un módulo que requiera acceder a componentes de otro módulo, deberá declararlos antes de su uso. Esas declaraciones deberán ser precedidas por la palabra reservada `extern`, lo cual le dice al enlazador (linker) que las definiciones de esos componentes (tipos, variables, constantes o funciones) no están en el módulo corriente, sino que están en otro módulo (el cual deberá ser includo en los respectivos comandos de compilación o enlazado).

La convención usada en programas C es que las declaraciones de los componentes que exporta cada módulo se incluyen en un archivo de cabecera, con sufijo `.h`. Estas declaraciones se definen como externas ya que su finalidad es la de ser incluidas en los módulos clientes.

La implementación de un módulo se escribe en un archivo con sufijo `.c`.

Uno de los inconvenientes de este esquema, el cual se basa simplemente en la inclusión textual de código, es que un archivo de cabecera podría incluirse más de una vez en un módulo[13] y podrían producirse errores de complilación por múltiples declaraciones o definiciones de los mismos identificadores. Para evitar esto, la práctica habitual es proteger a cada archivo de cabecera con directivas del preprocesador para detectar y evitar declaraciones repetidas.

La directiva `#ifndef symbol <body> #endif`, expande su cuerpo sólo si `symbol` está definido. Por lo tanto un patrón comúnmente utilizado en los achivos de cabecera es

```
#ifndef S
#define S
/* declarations */
#endif
```

A continuación se muestra un ejemplo de un archivo de cabecera (interface).

```
/* File: list.h */

#ifndef LIST_H
#define LIST_H
```

[13] El archivo `f1.h` podría incluir a `f2.h` y el programador podría incluir explícitamente ambos en otro archivo de cabecera.

```
typedef struct n {
    int value;
    struct n * next;
} list_node;

typedef struct {
    list_node * first, * last;
    int items;
} list;

extern list * list_new(void);
extern void * list_insert(list *l, int value, int pos);
...
#endif
```

La implemenación se muestra a continuación.

```
/* File: list.c */
#include <malloc.h>

typedef struct n {
    int value;
    struct n * next;
} list_node;

typedef struct {
    list_node * first, * last;
    int items;
} list;

list * list_new(void)
{
    list * result = (list *) malloc(sizeof(list));
    result->first = result->last = NULL;
    result->items = 0;
    return result;
}

void * list_insert(list *l, int value, int pos)
{
    ...
}

/* Auxiliar function (see static modifier) */
static list_node * goto_pos(list *l, int pos)
{ ...
```

7.23 Ejercicios

Diseñar un algoritmo utilizando el lenguaje C para cada uno de los siguientes problemas.

1. Leer un carácter y dos números enteros. Si el carácter leído es un operador aritmético calcular la operación correspondiente, si es cualquier otro mostrar error. Hacer el programa utilizando la instrucción *switch()*.

2. - Leer un entero positivo y averiguar si es perfecto. Un nÂ° es perfecto cuando es igual a la suma de sus divisores excepto el mismo.

3. - Leer por teclado un número entero largo e indicar si el número leído es o no capicúa. Para ello debe utilizarse un array unidimensional para almacenar cada una de las cifras del número leído. Se deben implementar dos funciones, una para descomponer el número en cifras y cargar el array, y otra para comparar las posiciones del array y determinar si el número es capicúa.

4. Calcular las raíces de una función utilizando el Método de la Bisección.

5. Cargar un array bidimensional de $p*q$ y devolver un puntero apuntando a la fila que mas suma.

6. Cargar un array bidimensional de $p * q$ y devolver un arreglo A de longitud p donde *A[i]* contiene la suma de la fila i de la matriz. Puede utilizar únicamente punteros tanto para recorrer la matriz como para cargar los resultados en el arreglo.

Capítulo 8

Manejo de la memoria

En este capítulo se describirán los mecanismos mas utilizados en la implementación eficiente del manejo de la memoria de los lenguajes de programación.

8.1 Manejo de la memoria eficiente

El modelo de celdas presentado es adecuado para dar una semántica abstracta de un modelo con estado mutable (statefull) el cual contiene tres tipos de memoria diferentes: una pila, la cual sirve como estructura de control para el flujo de ejecucuón de los programas, la memoria de valores σ y la memoria de celdas μ.

La arquitectura dominante de hardware sigue el modelo Von Newmann, el cual permite que tanto los datos como las instrucciones del programa se almacenen en la misma memoria de acceso directo (RAM).

La mayoría de los compiladores e intérpretes de lenguajes de programación modernos realizan un manejo de la memoria de manera muy eficiente. El manejo de la memoria que se describe a continuación se adapta a lenguajes de distintos paradigmas y estructurados a bloques[1].

Un proceso (programa en ejecución) tiene los siguientes bloques de memoria asignados. Estos bloques no están necesariamente contig'uos.

La figura 8.1 muestra la representación de un proceso en memoria. Un proceso consta de varios segmentos o áreas.

- Segmento de código (text segment): contiene las instrucciones del programa, es decir, el código del programa principal y de sus subrutinas.
- Segmento de datos: área de almacenamiento de las variables globales (y variables locales estáticas en C o C++).

[1] Un lenguaje estructurado a bloques contiene construcciones sintácticas (bloques) con sus propios ambientes. Los bloques pueden ser anidadas y cada bloque interno *hereda* el ambiente de su bloque contenedor.

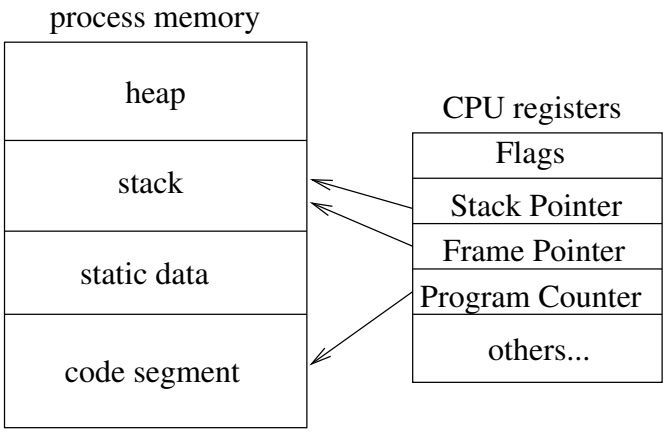


Fig. 8.1: Representación en memoria de un proceso.

- **Stack**: área de almancenamiento en las que se lleva el control de las activaciones de las subrutinas (direcciones de retorno) y en la mayoría de los lenguajes de programación modernos se lleva información de control adicional para acceso a ambientes locales y no locales y además se almacenan las variables locales y valores de los parámetros actuales de las subrutinas activas.

  En cada invocación a una subrutina se construye un *registro de activación*. En cada retorno de una función el registro de activación del tope se elimina dejando en el tope en el estado anterior a la invocación (su invocante).

  Mas abajo de describe en mas detalle el manejo del stack.

- **Heap**: mantiene bloques de memoria que representan datos del programa creados dinámicamente, esto es, creados por operaciones del tipo *new* (de Pascal, C++, Java, etc), *malloc* de C o valores creados implícitamente como el lenguaje utilizado aquí o lenguajes funcionales como Haskell o ML.

  Los bloques del heap son liberados explícitamente (operaciones como *delete*, *free*, *dispose*, ...) o implícitamente por un mecanismo de recolección de basura, como se describen más adelante.

8.2 Manejo del stack

En los primeros lenguajes de programación, como FORTRAN 77 y COBOL, el stack sólo se utilizó para llevar el control de invocaciones y retornos de subrutinas. El stack sólo contenía direcciones de retorno que eran salvadas por una instrucción de salto a una subrutina (instrucciones assembly *JSR S* o *CALL S*), la cual contiene un operando que corresponde a la dirección de memoria de la primera instrucción de la subrutina invocada. La instrucción apila (push) automáticamente el contenido del

program counter[2] en el stack y lo modifica con la dirección del operando. De esta forma el control pasa a la subrutina destino.

Cuando en la subrutina se ejecuta una instrucción de retorno (*RET*), ésta automáticamente desapila (pop) el contenido del tope del stack y lo copia al *program counter*, por lo que la ejecución del programa continúa con la instrucción siguiente a la invocación.

La estructura de datos de pila permite que los retornos se produzcan de una manera *last call- first returned*.

En la jerga de arquitecturas de computadoras la pila se denomina *system stack*.

Los lenguajes mencionados arriba almacenaban los valores de las variables del programa en un área estática (cuyo tamaño se calculaba en tiempo de compilación). En esa área estática se mantenían tanto los valores de las variables globales y las variables locales y parámetros de las subrutinas.

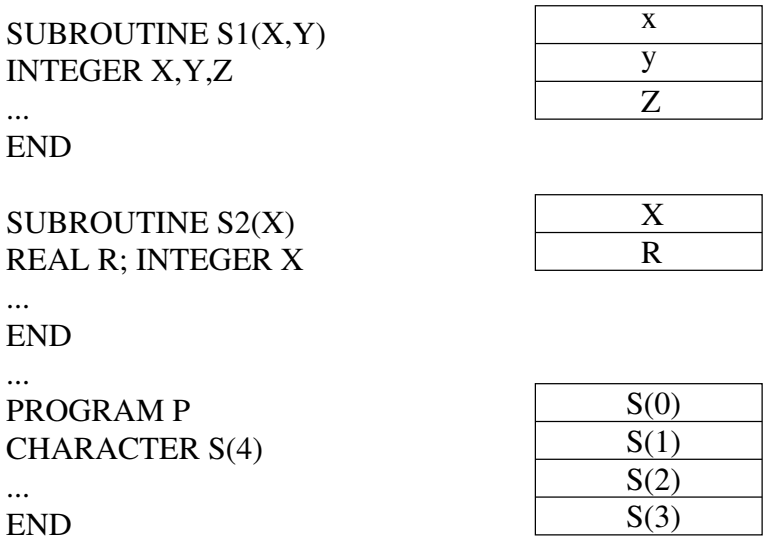


Fig. 8.2: Manejo de la memoria estática en FORTRAN 77.

La figura 8.2 muestra un ejemplo del manejo de la memoria estática implementada en los compiladores de FORTRAN 77 y anteriores.

El manejo de memoria en COBOL era aún mas simple ya que las subrutinas no soportan parámetros.

Este manejo, si bien es muy simple y eficiente, no permite recursión, lo cual es prohibitivo en la implementación de lenguajes de programación modernos.

Como ventaja se puede mencionar que las variables locales de las subrutinas mantenían su valor entre las invocaciones, por lo que los programadores muchas veces

[2] El cual contiene la dirección de la siguiente instrucción a la corriente en ejecución.

hacían uso de esa característica.

En la máquina abstracta para el lenguaje utilizado aquí el uso del stack permite realizar la tarea descripta arriba y además llevar el flujo de ejecución de cada instrucción, lo cual abstrae el *program counter*, ya que la máquina no contiene registros como las arquitecturas convencionales. Esta abstracción es adecuada para describir la semántica de las sentencias del lenguaje pero es obvia una implementación eficiente utilizando una arquitectura convencional moderna.

Los lenguajes modernos, muchos de los cuales son estructurados a bloques y permiten recursión, sólo hacen manejo estático de la memoria para las variables globales y hacen uso del stack no sólo para el control de invocación y retornos de subrutinas, sino que además se aprovecha para almacenar los ambientes locales (variables locales y valores de los parámetros actuales).

Esto hace que el stack se convierta en una estructura heterogénea, es decir que sus elementos sean *records* de tamaño variable por lo que es necesario llevar un puntero (link) en cada registro que apunte a una dirección base del registro inmediato anterior. Esto implementa un stack como una lista enlazada.

Este puntero generalmente se denomina *dynamic link*. También es necesario disponer de un puntero a una dirección base del registro de activación corriente (registro del tope del stack) para el acceso a los diferentes componentes dentro del registro, como las variables locales o parámetros.

La dirección base al registro de activación corriente generalmente se mantiene en un registro de la CPU reservado con ese fin. La mayoría de las CPUs corrientes contienen un registro especial para este propósito, generalmente denominado *frame pointer* o *base pointer*[3].

La figura 8.3 se muestra el formato de un registro de activación.

Se puede notar que el registro también se puede utilizar para almacenar valores retornados por las rutinas que representan funciones, aunque en algunos lenguajes se asume que los valores se retornan en un registro de la CPU. Esto último impone que las funciones sólo puedan retornar valores de tipos básicos (numéricos, caracteres, etc) o punteros a valores estructurados. Esto se da en lenguajes como C o Java.

Un registro de activación se forma en una secuencia de pasos durante en una invocación en la cual cooperan el invocante y la rutina invocada:

1. El invocante reserva lugar para el valor retornado por la rutina (sólo en caso que sea una función).

2. El invocante apila los valores de los parámetros actuales.

3. El invocante ejecuta la invocación (ej: instrucción CALL S). El control pasa a la primera instrucción de la subrutina.

4. La subrutina salva en el stack (apila) el valor del *frame pointer* (también denominado *dynamic link*) para poder restaurarlo en el retorno. El *frame pointer* se

[3] En la arquitectura Intel IA32, utilizada en la mayoría de las PCs actuales el registro usado es el *Base Pointer (EBP)*.

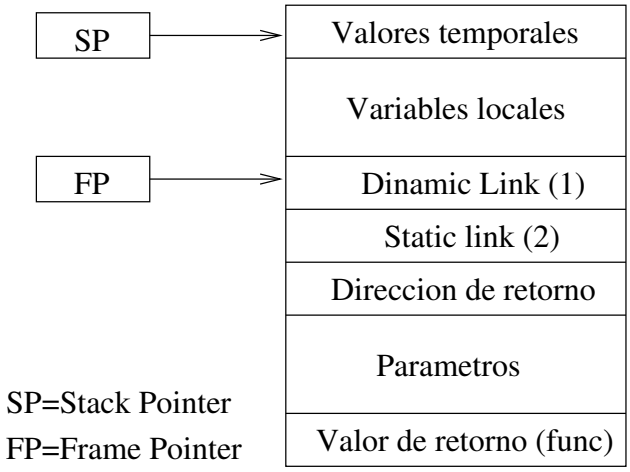


Fig. 8.3: Formato de un registro de activación.

hace apuntar al tope del stack. Esta dirección será usada como dirección base dentro del registro de activación.

Los valores de las variables locales se acceden con desplazamientos negativos y los parámetros con desplazamientos positivos (si el stack crece hacia direcciones decrecientes).

Cabe aclarar que todos los accesos a variables o parámetros se hacen usando el modo de direccionamiento basado, el cual es provisto por la mayoría de las CPUs modernas.

5. Si el lenguaje usa alcance estático (*static scope*), apila el *static link*, el cual apunta al registro de activación más próximo del bloque que lo contiene estáticamente en el programa. Cabe aclarar que si el lenguaje no es estructurado a bloques (como en el caso de C, el cual tiene sólo dos ambientes, el local y el global), este puntero no es necesario pues el ambiente no local o es global o corresponde al ambiente del invocante, el cual puede accederse mediante el *frame pointer*.

   En la siguiente sección se analizará en detalle el manejo del alcance estático.

6. La subrutina reserva lugar en el stack para los valores de las variables locales. Esto se realiza simplemente modificando el valor del stack pointer (decrementándolo o incrementándolo, dependiendo hacia adónde crece el stack, si hacia direcciones crecientes o decrecientes).

   El espacio requerido para las variables locales generalmente es determinado estáticamente (tiempo de compilación).

7. Se continúa con la ejecución del resto de las instrucciones del cuerpo de la rutina. Los valores temporarios producidos por la evaluación de expresiones complejas pueden apilarse en el tope del stack.

8. Comienzo de las acciones de retorno de la subrutina: se eliminan las variables locales. Nuevamente esto se realiza simplemente moviendo el tope del stack (stack pointer).

9. Se desapila y recupera el *frame pointer* del registro de activación anterior (invocante).

10. Se ejecuta el retorno (ej: instrucción RET), la cual desapila el contenido del tope del stack y se lo asigna al *program counter*.

11. El control vuelve al invocante, el cual debe eliminar los valores de los parámetros actuales (modificando el tope del stack). En el caso que se haya invocado a una función, se desapilará el valor retornado.

12. El invocante continúa con su ejecución.

Es importante notar que cada subrutina tiene que realizar ciertas acciones en la entrada (acciones 1 a 6) y durante la salida o retorno (acciones 8 a 10). Estas acciones se implementan con dos o tres instrucciones de máquina y se denominan preámbulo y epílogo, respectivamente.

Algunas arquitecturas (ej: Intel IA32[7]) proveen instrucciones de máquina para realizar estas operaciones[4].

8.2.1 Implementación del manejo de alcance de ambientes.

Un lenguaje con alcance dinámico (*dinamic binding*), el acceso a una variable o es local al bloque corriente, o puede accederse siguiendo el *dynamic link* ya que la búsqueda debe seguir la cadena de invocaciones.

Los lenguajes con alcance estático, requieren de un link adicional ya que el acceso a una variable no local deberá seguir la cadena de declaraciones de los ambientes estáticos. Este link se conoce como el *static link* y se almacena en el mismo registro de activación y apunta al registro de activación correspondiente a la activación mas próximo del bloque que lo contiene estáticamente en el programa.

El link estático implementa en forma eficiente la unión de ambientes[5], ya que representa el conjunto de ambientes de un bloque como una lista enlazada.

El acceso a una variable no local deberá seguir la cadena estática hasta acceder al primer registro de activación que corresponda al bloque que declaró tal variable.

En los lenguajes estructurados a bloques, como Pascal y Oz, un acceso a una entidad no local requiere que se realicen n pasos siguiendo el static link, donde n es la

[4] Instrucciones ENTER y LEAVE en [7]

[5] Recordar la semántica de la sentencia **local** de nuestro lenguaje kernel.

diferencia de niveles entre el bloque corriente y el bloque mas inmediato que declaró la entidad. El costo es lineal.

Para evitar que los programas recorran parte de una lista en cada acceso a entidades no locales, generalmente se utiliza un *display*.

El display es un arreglo el cual contiene un puntero a la dirección base de un registro de activación por cada ámbito (nivel) estático del programa.

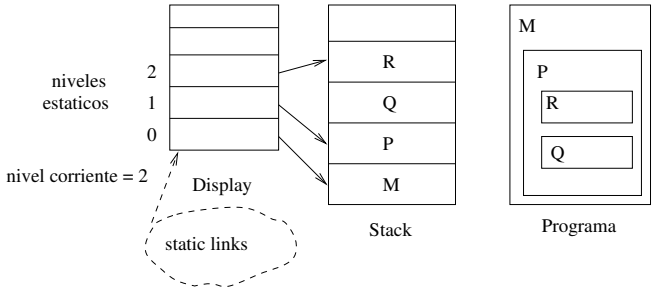


Fig. 8.4: Implementación de static scope con un display.

La figura 8.4 muestra el uso de un display para un programa estructurado a bloques con alcance estático. La figura muestra que el programa se encuentra ejecutando el bloque `R`, el cual fue invocado por `Q`.

Se debe notar que, en este ejemplo, la cadena dinámica (invocaciones) no se corresponde con la cadena estática. El display permite el acceso a entidades no locales con costo constante, ya que se accede a la dirección base del registro de activación correspondiente, mediante un desplazamiento dentro del display.

Este desplazamiento puede computarse estáticamente, es decir, en tiempo de compilación.

El display se actualiza en la entrada de cada subrutina, salvando en el registro de activación la dirección del tope del display para luego hacer que el display apunte (en el nivel correspondiente) al tope del stack (registro de activación corriente).

En el caso que la subrutina corresponda a un nivel mayor que el corriente, se incrementa el nivel corriente el el display (se realiza un push sobre el display). Si la subrutina invocada se corresponde al mismo nivel estático que el invocante, el valor del tope del display se reemplaza por el nuevo (se mantiene la altura del display).

Durante el retorno, se actualiza el nivel corriente del display. Si el nivel del bloque al que se retorna es igual al corriente (como en el caso del retorno de `R` a `Q`, en la figura 8.4) se restaura el puntero del nivel corriente del display con el salvado en el registro de activación de la rutina que está retornando.

Los demás casos se dejan como ejercicios al lector.

8.3 Valores creados dinámicamente. Manejo del heap.

La gran mayoría de los lenguajes modernos permiten la creación de valores dinámicamente, es decir en tiempo de ejecución y que no están asociados a un ámbito local en particular.

Esos valores generalmente son referenciados por *referencias*, como por ejemplo en Java o Eiffel, o se pueden acceder mediante *punteros*, como en Pascal, C o C++.

A modo de ejemplo, el siguiente programa Pascal muestra el uso de variables dinámicas:

```
...
Var p,q:^integer;
begin
  new(p);
  p^ := 100;
  q := p;
  ...
end
...
  dispose(q);
```

En el ejemplo, un valor numérico es creado (*allocated*) en el heap mediante el procedimiento nativo **new**, el cual toma su argumento (un puntero) por resultado, haciendo que apunte al bloque reservado. El operdor postfijo ^ se aplica a un puntero

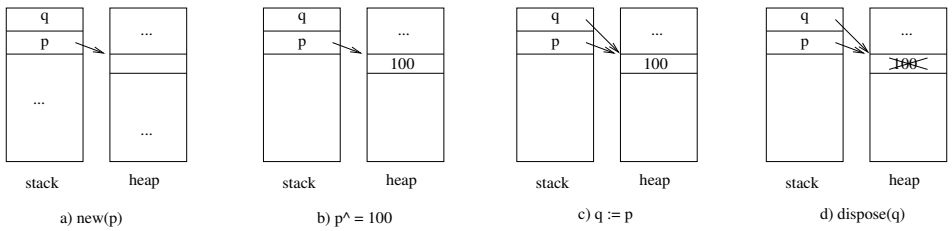


Fig. 8.5: Creación y manejo de valores dinámicos.

y retorna el valor apuntado. El procedimiento nativo **delete** destruye (*deallocate*) el valor apuntado por su argumento. La figura 8.5 muestra los estados en la memoria.

Pascal y C son lenguajes en los cuales el programador es responsable del manejo del heap. Tanto la creación como la destrucción de valores son explícitas.

Esto puede acarrear dos problemas en los programas:

- ***Basura***: se produce cuando un valor en el heap no puede ser referenciado porque no hay forma de referenciarlos (las referencias o punteros a él ya no existen).

  Por ejemplo, si el ejemplo anterior se omitiera la sentencia ***dispose(q)***, luego de que se hubiesen destruido los punteros p y q sería imposible de referenciar al

bloque de memoria en heap y por lo tanto nunca podría ser reutilizado.

Este tipo de errores es bastante difícil de detectar porque el programa puede tener el comportamiento esperado, pero a medida que avance su ejecución irá consumiendo el heap hasta que en algún momento la operación de *allocation* falle.

Se dice que un programa que genera basura tiene *memory leaks* (lagunas de memoria).

- ***Referencias colgadas***: se produce cuando se intenta acceder a un valor que ha sido destruido (*deallocated*).

  Por ejemplo, si en el ejemplo anterior, luego de ejecutar ***dispose***(*q*), hubiese la sentencia `write p^;` se produciría una referencia colgada. Otro error común es referenciar punteros o referencias no inicializadas.

  Cuando un programa tiene referencias colgadas, su comportamiento será errático y a menudo se produce el error *segmentation fault* (falla de segmentación)[6].

8.3.1 Manejo del heap

El manejo del heap está implementado en las operaciones de *allocation* y *deallocation* de bloques en el heap. El manejador del heap contiene dos listas, una que memoriza los bloques asignados y otra que registra los libres. La figura 8.7 muestra una representación del heap.

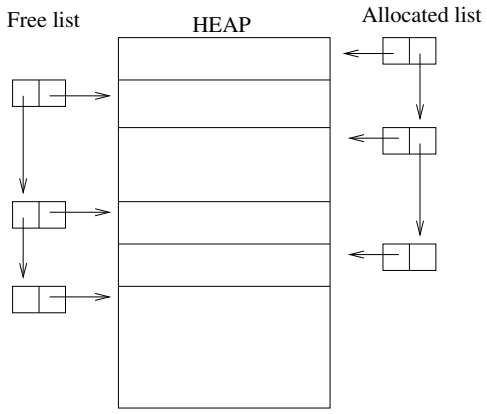


Fig. 8.6: Estructuras de datos del heap.

[6] El error es generado por el sistema operativo ya que el proceso está tratando de acceder a un segmento (bloque) de memoria sin permiso.

Hay lenguajes que permiten que los bloques del heap tengan tamaño uniforme, como por ejemplo en Lisp y Haskell. Esto permite que el manejo del heap sea muy simple.

El heap se divide en n bloques y en realidad no importa qué bloque se asigne en un requerimiento pues son todos iguales.

Otros lenguajes, donde los valores pueden tener representaciones de diferentes tamaños (como es el caso de registros o arreglos), el heap tendrá que ser manejado como una estructura heterogénea, es decir puede tener bloques de diferentes tamaños. Esto accarrea una complicación extra en el manejo, ya que cada bloque deberá tener al menos un descriptor de su tamaño (además de su dirección).

En este esquema, inicialmente el heap tiene un único gran bloque disponible.

El manejo del heap con bloques de diferentes tamaños acarrea el problema de la *fragmentación del heap*. Esto quiere decir que a medida que se van realizando requerimientos y liberación de bloques de diferentes tamaños en el heap, van quedando *huecos* entre los bloques en uso que podrían ser menor que el tamaño de un próximo bloque requerido, por lo que este último requerimiento no podría ser satisfecho.

Existen dos posibles soluciones a este problema:

1. **Compactación total**: este proceso movería los bloques en el heap para reconvertir los múltiples bloques libres en uno solo. El problema de la compactación total es que hay que modificar las referencias ya que los bloques fueron movidos.

2. **Compactación parcial**: esto se realiza en la operación de liberación de un bloque. Al momento de liberar un bloque, si éste tiene bloques adyacentes libres, se fusionan en un solo bloque. Esto es mas eficiente y generalmente en la práctica funciona muy bien.

8.3.2 Manejo automático del heap

Los lenguajes de programación modernos proveen mecanismos para el manejo de la memoria en forma automática. Por ejemplo los lenguajes funcionales de la familia ML y Haskell proveen tanto la creación de valores como su destrucción de manera implícita. Algunos lenguajes, como los lenguajes orientados a objetos y a clases, tienen operadores para la creación de objetos pero no para su destrucción. La destrucción es implícita.

El manejo de la memoria en forma automática evita que los programas contengan los errores mencionados anteriormente con algún costo en la performance de la ejecución de los programas.

Hay básicamente dos métodos para el manejo de la memoria que se han implementado en los lenguajes de programación modernos.

1. **Contadores de referencias**: cada bloque de memoria que representa un valor del programa tiene asociado un contador de referencias al bloque.

En cada copia de referencias, esto es en las operaciones de asignación y pasaje de parámetros, los contadores correspondientes se actualizan.

A modo de ejemplo, en una asignación de la forma `p := q` (donde `p` y `q` son punteros o referencias a bloque del heap), se deberá decrementar el contador de referencias del bloque previamente referenciado por `p`, para luego incrementar el contador de referencias del bloque referenciado (o apuntado) por `q`.

Cada vez que una referencia finaliza su tiempo de vida (porque finaliza la ejecución del ambiente en que fue declarado), el contador de referencias se decrementa. Si el contador llegó a cero, significa que el bloque ya no puede ser referenciado, por lo tanto puede liberarse en forma segura.

Se debe notar que en el lenguaje kernel utilizado en el actual apunte el incremento del contador se debería implementar en la operación primitiva *Bind*. Se deja como ejercicio la redefinición de la semántica de la máquina abstracta para el uso de contadores de referencias.

Este mecanismo es simple de implementar pero tiene la desventaja de tener una sobrecarga (overhead) constante en la ejecución de un programa. Cabe recordar que las principales operaciones de computación es el *binding*.

2. **Recolectores de basura**: en este enfoque el mecanismo permite que el programa vaya generando basura y en algún momento de la ejecución[7] se dispare un proceso, el *recolector de basura* al cual se encargará de detectar los bloques basura para luego recuperarlos.

8.3.3 Algoritmos de recolección de basura

Existen muchos algoritmos de recolección de basura. Esta ha sido y sigue siéndolo en la actualidad un área de activa investigación y hay libros de texto completos únicamente destinados a cubrir esta área.

Se describirán tres algoritmos, los cuales son la base de la gran mayoría de los demás.

- **Mark and sweep**: la idea se basa en un algoritmo de dos etapas. La primera de ellas (mark) se encarga de *marcar* los bloques ocupados, es decir los bloques alcanzables desde las referencias (o punteros) del programa. Este algoritmo generalmente comienza explorando la memoria estática y el stack, para luego realizar una clausura de los bloques alcanzados directamente[8]. Este algoritmo deberá tener en cuenta que dicha relación de alcanzabilidad puede ser cíclica.

[7] Por ejemplo cuando no se pueda satisfacer un requerimiento de *allocation* en el heap.
[8] Un bloque podría contener referencias a otros bloques

El segundo paso (sweep) se encarga de tomar los bloques no marcados y adicionarlos a la lista de bloques libres. Obviamente los bloques no marcados son inalcanzables desde las referencias del programa, por lo que son basura.

- **Generacionales**: la idea consiste en dividir al heap en *generaciones*. El manejador va categorizando los bloques en generaciones en base a la duración de su tiempo de vida activa. Los bloques de mayor tiempo de vida se almacenan en su propia partición (generación). El recolector procesa las generaciones mas jóvenes primero. Cada cierto tiempo el recolector *mueve* bloques a generaciones *mas viejas*. Esto permite que se haga una recolección parcial (o incremental) de la basura lo que permite que el tiempo de ejecución del recolector no sea larga y el programa no sufra largas *pausas*.

- **Copia**: los recolectores de basura basadas en copia generalmente particionan el heap en dos partes. Una parte es la región activa. Cuando una solicitud no puede ser satisfecha, el recolector copia todos los bloques alcanzables de la región activa a la otra, la cual pasa a ser la nueva región activa. Durante el proceso de copia se puede hacer la compactación de los bloques. La región anterior queda completamente disponible para la próxima copia.

  Esta técnica tiene la ventaja que realiza una sola pasada pero como desventaja, aprovecha la mitad del heap.

Un problema adicional a resolver con el manejo automático de la memoria es el uso de programas externos. Por ejemplo, un programa podría tener una estructura de datos que puede ser accedida de una biblioteca externa al lenguaje. En este escenario, si la estructura fuese reclamada por el manejador de la memoria, sería un error. Otro ejemplo sería que un programa tenga referencias a datos externos. Esos datos no podrían ser reclamados cuando ya no estén en uso ya que el lenguaje no puede manejar el heap del programa (o biblioteca) externo.

Estos casos generalmente se solucionan de dos maneras posibles. Las bloques de memoria local referenciados externamente se marcan y no se reclaman. Los datos externos referenciados por el programa tienen asociados *proxies*, los cuales realizan el reclamo al programa externo cuando el proxi en sí sea reclamado. Esta operación generalmente se conoce como *finalización*.

Muchos lenguajes de programación brindan mecanismos para la finalización. Es el caso de los destructores de C++ (los cuales son invocados automáticamente en la destrucción de un objeto, ya sea automáticamente cuando éste reside en el stack o por la operación `delete`) y el método *finalize()* de Java, el cual es invocado por el recolector de basura.

8.4 Ejercicios

1. Determinar en Pascal un experimento para ver la representación de la variable s:

```
Type DiaSemana = (Dom,Lun,Mar,Mie,Jue,Vie,Sab);
Var s:Set of DiaSemana;
```

2. La siguiente expresión calcula la dirección base del elemento i-ésimo de un vector (a lo C) `T A[N]`, asumiendo que el tipo índice es el subrango [0..N-1].

$$dir(i) = dir\_base(A) + i * sizeof(T)$$

donde *sizeof(T)* es el tamaño de la representación de un valor de tipo T.

   - Dar la expresión para una declaración de un vector a lo Pascal: `A[li..ls] of T`.
   - Idem al anterior pero para una matriz de la forma `A[li0..ls0,li0..ls0] of T`.

3. Dado el siguiente programa Pascal, mostrar la pila de registros de activación hasta el punto *, mostrando los link estáticos y dinámicos.

```
Program P;
Var g:boolean;

  Function f2(Var b:boolean; y:integer):integer;
  begin
    if b then f2 := y + 1 else f2 := y - 1;
    b := not b
  end

  Procedure p1;
  Var x:integer;

    Function f1(y:integer):integer;
    begin
      f1 := y + f2(g,y)
    end

  begin {p1 body}
    x := 1
    x := f1(x) {*}
  end {p1}

begin
  g := True;
  p1
end.
```

4. Dado el siguiente programa Pascal, mostrar la pila de registros de activación hasta el punto *, mostrando los link estáticos y dinámicos.

```
Program P;
Var a:integer;

  Procedure P2(e:Char; Var f:integer; g:integer);

    Function fun(c:char; a:integer):integer;
      begin
        fun := ord(c) + a
      end;

  begin
    f := f + 1;
    g:= fun(e,f);
    writeln(g) {*}
  end;

  Procedure P1(b:integer);
  Var c:Char;
  begin
    c := chr(b);
    P2(c,b,a)
  end;

begin
  a := 64;
  P1(a);
end.
```

5. Dar un ejemplo en Pascal y en C que genere *basura*.

6. Dar un ejemplo en Pascal y en C el cual tenga una referencia colgada.

7. Diseñar un algoritmo en JAVA, que fuerce la ejecución del recolector de basuras. La creación de objetos debe detenerse en cuanto el recolector comience a funcionar. Se debe visualizar:

   - la cantidad de objetos credos hasta la ejecución del recoletor de basura.
   - el orden en que los objetos van finalizando (ver método *finalize()*).
   - un mensaje `'Todos los objetos finalizaron'` cuando todos los objetos hayan finalizado.

8. Utilize la herramienta *VisualVM*[9] para observar el comportamiento en memoria del programa del ejercicio 7.

9. *Garbage collector* del lenguaje JAVA:

   - Analizar la estrategia por Generaciones que usa Java 11 (o posterior).

- Estudie los parámetros de la MV Java que permite modificar (adaptar) el heap a las necesidades de una aplicación.
  Ejemplo: `java -Xmx4g programaJava`, indica que el Heap tendrá un tamaño máximo de 4Gb.
- Utilizando los parámetros adecuados realize experimentos de tal forma que deje registros (.log) del comportamiento del GC. Chequear con un editor de textos el *.log*. También visualizar el log con la herramienta *GcViewer*[10]. Use los siguientes parámetros al lanzar el programa java: $-verbose : gc - XX : +PrintGCDetails - XX : +UseConcMarkSweepGC - Xloggc : /directorio..../garbage.log$,
- Usando el comando `jmap`[9][11] realize un volcado de memoria de un programa Java en ejección. Luego con la herramienta *Eclipse Memory Analyzer*[13] (u otra para los mismos fines como *VisualVM*) analize el uso de la memoria corriente.

10. Manejo de Memoria. La figura 8.7 muestra un estado del HEAP de un proceso. Mostrar gráficamente como queda el mismo luego de la ejecución del recolector de basura el cual se basa en la estrategia por *copia*.

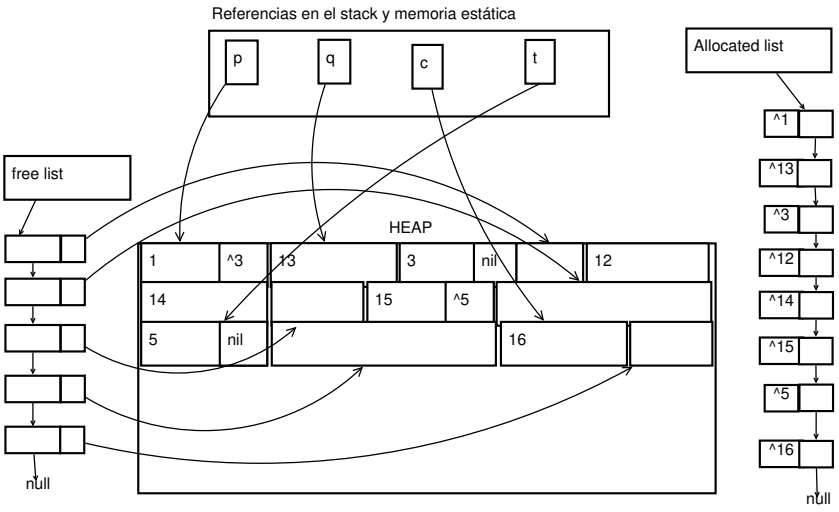


Fig. 8.7: Estado del Heap.

Ejercicios Adicionales

11. Realizar un experimento en Pascal para determinar como se almacenan los valores de una matriz (arreglo bidimensional).

[9] jmap -dump:file=heap.hprof PID

12. Escriba en Pascal o en C un programa que contenga una función recursiva $factorial(n)$ y mostrar la pila de ejecución para $n = 3$.

Capítulo 9

Programación orientada a objetos

En este capítulo se describirán los principales conceptos de la programación orientada a objetos (OOP). Se estudiarán conceptos tales como objetos, clases, herencia, dynamic binding, polimorfismo y generecidad.

Se extenderá el lenguaje kernel para soportar los conceptos mencionados y se dará su semántica formal en la máquina abstracta.

9.1 Objetos

Una función (o varias) con una memoria interna (estado) se denomina un *objeto*. Se verá que los objetos son muy útiles. A modo de ejemplo, la figura 9.1 muestra la implementación de un objeto *Counter*.

```
local C in
    C={NewCell 0}
    fun {Inc}
        C := @C + 1
        @C
    end
    fun {Read}
        @C
    end
end
```

Fig. 9.1: Implementación de un objeto Counter

La sentencia **local** define la variable `C`, la cual es visible sólo dentro del bloque. El contador sólo es accesible por medio de las funciones **Inc** y **Read**.

Esto se llama *encapsulamiento*, ya que la variable está oculta al resto del programa y sólo puede ser manipulado por una *interface*, las funciones **Inc** y **Read**.

La separación entre interface e implementación es fundamental para realizar abstracciones de datos. Esto permite que los programas se abstraigan de la implementación de los objetos y sólo tengan que respetar su interface. Este mecanismo es escencial para una adecuada modularización en la programación en gran escala.

9.2 Clases

En la sección anterior se ha definido un objeto. Cómo es posible crear más objetos como `Counter`?

Para ello es necesario una fábrica (*factory*) de objetos. Las clases en la OOP juegan el papel de fábricas de objetos. Una clase también define un módulo: *una implementación de un tipo abstracto de datos que define una interface y oculta su implementación (y su estado).*

La figura 9.2 muestra una posible implementación de una clase que crea objetos de tipo `Counter`. La función `NewCounter` retorna un registro con las funciones `Inc` y `Read` que operan sobre la variable `C`, la cual permanece oculta.

```
declare fun {NewCounter}
    local C Inc Read in
        C={NewCell 0}
        fun {Inc}
            C := @C + 1
            @C
        end
        fun {Read}
            @C
        end
        counter(inc:Inc read:Read)
    end
end
```

Fig. 9.2: Una posible implementación de una clase.

Un programa podrá usar la función `NewCounter` para crear diferentes instancias (objetos):

```
C1 = NewCounter
C2 = NewCounter
{Browse {C1.inc}}
{Browse {C2.read}}
```

Se extenderá el lenguaje núcleo con estado para dar soporte sintáctico a la programación orientada a objetos, como se muestra en la figura 9.3.

```
<statement>  ::= class <variable> { <class-desc> } { <method> }
               | ...
<class-desc> ::= from { <expr> }+ | prop { <expr> }+ | attr { <attr_init> }+
<method>     ::= meth <meth_head> [ '=' <var> ] ( <in_expr> | <in_stmt> ) end
<expr>       ::= class '$' { <class_desc> } { <method> }
               | self
               | ...
<attr_init>  ::= ( [!] <var> | <atom> | unit | true | false ) [ ':' <expr> ]
<meth_head>  ::= ( [!] <var> | <atom> | unit | true | false ) [ ':' <expr> ]
                 [ '(' { <arg> } ['...'] ')' ] [ '=' <var> ]
<arg>        ::= [ <feature> ':' ] ( <var> | '_' | '$' ) [ '<=' <expr> ]
```

Fig. 9.3: Sintaxis extendida para soportar OOP.

Las extensiones al lenguaje núcleo será una abstracción lingüística ya que se verá como se puede representar una clase y sus características específicas.

La figura 9.4 muestra un ejemplo de la clase `Counter` con la sintaxis extendida.

```
class Counter
    attr val;
    meth init(Value)
        val := Value
    end
    meth inc(Value)
        val := val + Value
    end
    meth get
        @val
    end
end
```

Fig. 9.4: Ejemplo de una clase.

La figura 9.5 muestra una posible implementación de la clase **Counter** en el lenguaje núcleo con estado.

Como se puede apreciar en la figura 9.5, la clase `Counter` se representa como un registro con dos campos: una lista de atributos y otro registro cuyos campos se ligan a referencias de los procedimientos correspondientes. Este último registra los métodos.

Los objetos (instancias de clases) se crean mediante el operador `New`. A continuación se muestra la creación de un objeto (instancia) de la clase `Counter` y su utilización.

```
C = {New Counter init(0)}
{C inc(6)}
```

```
local
    proc {Init M S}
        init(Value)=M in (S.val) := Value
    end
    proc {Inc M S}
        inc(Value)=M in (S.val) := @(S.val) + Value
    end
    proc {Get M S}
        get = M in @(S.val)
    end
in
    Counter = c(attrs:[val] methods:m(init:Init inc:Inc get:Get))
end
```

Fig. 9.5: Ejemplo de una clase.

```
{Browse {C get}}
```

Una sentencia de la forma `{C inc(x)}` se denomina una aplicación sobre el objeto `C` del método `inc(x)`, invocación de un método. El objeto sobre el cual se realiza la aplicación se denomina el objeto destino (target). En otros lenguajes populares orientados a objetos como C++, Java, Eiffel, etc, es común la notación `object.method`.

Es importante notar que en esta forma de representar clases y objetos, una aplicación se implementa por medio de un mecanismo de mensajes codificados en forma de registros. La decodificación del mensaje se logra por medio de pattern matching en cada procedimiento.

Para poder entender la semántica completa de este ejemplo, a continuación se muestra la definición del operador `New`.

```
fun {New Class Init}
    Fs = {Map Class.attrs fun {$ X} X#{NewCell _} end}
    S = {List.toRecord state Fs}
    proc {Obj M}
        {Class.methods.{Label M} M S}
    end
in
    {Obj Init}
    Obj
end
```

Esta función crea el estado del objeto y lo inicializa (creando para cada uno de ellos una nueva celda). Además define un procedimiento `Obj` con un argumento. Este procedimiento representa el objeto y su argumento el método (mensaje) a ser invocado. Cuando se invoca `Obj` con un mensaje como argumento, redirige (invoca) al

procedimiento correspondiente al mensaje pasándoles como parámetros el registro que codifica el mensaje y el estado del objeto.

Es posible ver al procedimiento (objeto) `Obj` como un despachador (dispatcher) de mensajes, esto es, relaciona un mensaje con un procedimiento que implementa el método correspondiente. El método destino es el rótulo del registro que codifica el mensaje.

Se debe notar que la función `New` antes de retornar el objeto, invoca al método de incialización (`Init`).

Esta implementación de objetos es un ejemplo de programación con estado y alto orden (la función `New` retorna un procedimiento).

El estado del objeto está oculto por las reglas de alcance lexicográfico.

9.3 Clases y objetos

Una clase define los componentes que tendrán los objetos (instancias) correspondientes a la clase. En el lenguaje definido en este capítulo existen tres tipos de partes que un objeto puede tener:

- **Atributos**: definidos por la palabra **attr**. Un atributo es una celda que almacena parte del estado del objeto.

  En la terminología de la OOP, un atributo se conoce generalmente como una variable de instancia o campo (field member en C++).

  Cada instancia (objeto) tiene su propio conjunto de atributos, aunque algunos lenguajes proveen mecanimos para que todas las instancias de una clase compartan algunos de sus atributos, como por ejemplo con el modificador `static` de Java o C++.

  Los atributos podrán ser accedidos y modificados por los métodos de la clase. Algunos lenguajes permiten que se accedan a los atributos desde otros objetos, si es que son visibles. Algunos lenguajes, como Eiffel, ven a los atributos como funciones de consulta (query), por lo que los atributos pueden ser sólo leídos desde otros objetos, si la visibilidad lo permite.

- **Métodos**: declarados por la palabra **meth**. Un método representa un procedimiento o función que en una invocación se ejecuta en el contexto de un objeto en particular, el cual es el objeto *target*.

- **Propiedades**: declarados con la palabra **prop**. Una propiedad modifica el comportamiento de los objetos. Una propiedad puede crear un cerrojo (lock) en cada objeto creado, útil en concurrencia. La propiedad `final` afecta a la herencia (prohíbe su extensibilidad). La herencia se verá mas adelante en este capítulo.

Otros lenguajes de POO tienen algunos de estos mecanismos como modificadores de la declaración de cada clase, método o atributo.

9.3.1 Inicialización de atributos

Los atributos se pueden inicializar de dos formas posibles:

- *por instancia*: un atributo puede tener un valor inicial en cada instancia, lo que requiere que deberán ser inicializados con valores pasados como parámetros del método de inicialización.
- *por clase*: un atributo podrá tener el misno valor inicial para todas las instancias de la clase. En su declaración se debe dar su valor incicial, luego del símbolo : (dos puntos).

  Por ejemplo: attr city : "Río Cuarto"

En la mayoría de los lenguajes OO, la inicialización de objetos se realiza por medio de métodos u operaciones especiales denominados *constructores*.

9.3.2 Métodos y mensajes

Los métodos de una clase son invocados por medio de mensajes codificados como registros. El encabezado de cada mensaje es un patrón que encaja con un record. Como consecuencia de este mecanismo, una aplicación (invocación) de la forma `{Obj M}` puede hacerse de dos formas posibles:

- *M es un record estático*: es decir, es conocido en tiempo de compilación, como por ejemplo: `{Counter inc(6)}`
- *M es un record dinámico*: es posible una expresión `{Obj M}` donde `M` es una variable. Esto da una gran flexibilidad, ya que los registros pueden ser creados dinámicamente, es posible crear mensajes de la misma forma.

También es posible definir métodos con un número fijo o variable de argumentos. Un método con un número variable de argumentos (usando el símbolo `...`) es aceptado si concuerdan los argumentos listados explícitamente.

El encabezado de un método puede ligarse a una variable. Por ejemplo, la declaración

```
meth foo(a:A b:B ...) = M
   % body
end
```

En este caso la variable `M` referencia al mensaje completo al momento de una invocación.

Un argumento puede ser usado opcionalmente en una invocación, ya que es posible definir un valor por omisión (default)por cada argumento en el encabezado de un método, tal como lo muestra el siguiente ejemplo.

```
meth foo(a:A b:B<=5)
   % body
end
```

Un método puede ser privado (oculto fuera de la clase) si se utiliza una variable protegida por las reglas lexicográficas (dentro de la clase).

También es posible que el nombre (label) de un método pueda definirse o computarse dinámicamente, utilizando el operador `!` precediendo a la variable usada como rótulo del método. Por ejemplo:

```
meth !A(x)
   % body
end
```

Esto es posible ya que las clases se construyen dinámicamente.

Finalmente, es posible definir el método `otherwise`, el cual funciona como un método por defecto en el caso que el mensaje falle en el matching con los demás. Si este método existe en una clase, sus instancias aceptarán cualquier mensaje.

El compilador de Oz, trata de optimizar las invocaciones. En el caso de que el mensaje sea estático, crea un llamado tan rápido como una invocación a un procedimiento, sino compila a una llamada general sobre el objeto. Esta última instrucción usa una técnica llamada memorización o catching. La primera invocación es lenta, pero las siguientes se realizan muy rápido porque el método ya se encuentra en la caché.

9.3.3 Atributos de primera clase

Los nombres de los atributos de un objeto pueden determinarse dinámicamente. Por ejemmplo, la clase

```
class Inspector
   meth get(A ?X)
      X = @A
   end
   meth set(A X)
      A = X
   end
end
```

Cualquier clase que tenga métodos de ese tipo dejará sus atributos expuestos al mundo exterior. Esto es peligroso como técnica de programación pero puede ser muy útil para hacer introspeción (acceder a información sobre un objeto dinámicamente) y depuración (debugging).

9.4 Herencia

Las clases son una unidad de modularización y también definen nuevos tipos de datos. Esta dualidad es fundamental para definir abstracciones de datos en forma incremental.

La herencia es un mecanismo para definir nuevas clases a partir de otras existentes. Una clase `B` puede heredarse desde una clase base `A`, lo que significa que `B` automáticamente contiene (hereda) las definiciones de `A`. Además en la definición de `B`, es posible incluir nuevas características (atributos, métodos y propiedades) y redefinir características heredadas de `A`.

Se dice que la clase `B` es la *subclase* de `A` y esta última es su *superclase* (inmediata).

La relación de herencia es *transitiva* y *no reflexiva*. La herencia define una relación entre clases que puede modelarse como un árbol.

Una clase puede heredar de una o más clases, lo cual se conoce como *herencia múltiple*. Las clases de las cuales se hereda, aparecen en la cláusula **from** en la definición de una clase.

En presencia de herencia múltiple la relación de herencia entre clases puede modelarse como un grafo dirigido acíclico.

Definición 9.4.1 *Un método en una clase* `B` ***sobreescribe*** *(overrides) cualquier método con el mismo rótulo definido en sus superclases.*

El concepto de sobreescritura (o sobrecarga) es mas general que la *redefinición*, la cual es una sobreescritura respetando el perfil del método (tipo de valores de retorno y número y tipos de los argumentos).

9.4.1 Control de acceso a métodos (ligadura estática y dinámica)

La sobreescritura (o sobrecarga) de métodos en clases heredadas complica la desición de a qué método invocar ya que puede haber varios candidatos. El siguiente ejemplo muestra tal situación.

```
class A {                class B from A
   ...                      ...
   meth m                   meth m1
      {self m1 ...}            ...
   end                         {A,m ...}
                            end
   meth m1                  ...
      ...                 end
   end
   ...
end
```

Dada una sentencia de la forma `O = {New B ...}`, la cual crea una instancia de tipo `B`, y la siguiente aplicación `{O m}`, lo cual invoca al método `m`. En el cuerpo de *m*

se invoca al método `m1`, cl cual fue sobreescrito en la clase `B`. Esta situación presenta una pregunta: a qué versión de `m1` se debe invocar en el cuerpo de `m`?

Claramente, la respuesta es que, en este caso, debería invocar a la versión de `m1` de la clase `B`.

Esto trae un problema de compilación. Al compilarse la clase `A` el compilador no puede asumir a qué método invocar específicamente ya que puede ser sobreescrito en alguna subclase, por lo que deberá generar código para que la determinación del método a invocar se realice dinámicamente (en tiempo de ejecución). Esto se conoce como *ligadura dinámica (dynamic binding)* o *ligadura tardía (late binding)*.

Ahora supongamos que en el método `m1` de la clase `B` se desea invocar al método `m1` de la clase `A`. Si se realiza la invocación de la forma `{m1 ...}`, sería interpretada por el compilador o intérprete como una llamada recursiva. Para realizar la invocación deseada, se deberá indicar explícitamente al método `m1` de la clase `A`. En Oz esto se denota como `{A,m1 ...}`.

Esta última forma de invocación se puede determinar estáticamente, ya que explícitamente se denota a qué método específico se desea invocar. En esta forma de invocación se utiliza *ligadura estática (static binding)*.

En Java se utiliza la palabra reservada `super` para especificar que el destino es un método específico de la superclase. En C++, la sintaxis es similar a Oz, ya que existe un operador de alcance de la forma *`clase`*`::`*`member`*, similar al operador de Oz `,`.

En otros lenguajes de programación, una invocación puede omitir el objeto destino (target), en cuyo caso se interpreta como que el objeto destino de la invocación es `self`. El operador `self` se conoce en otros lenguajes como `this` (en C++, Java, etc), o `Current` como en Eiffel.

La determinación en tiempo de ejecución del método al cual se debe invocar acarrea desafíos de eficiencia de su implementación.

Para lograr una mayor eficiencia en el mecanismo de invocaciones (dispatching) muchos lenguajes orientados a objetos, principalmente aquellos con sistemas de tipos estáticos, como C++, Java o Eiffel, implementan las invocaciones como llamadas a procedimientos (y no como mensajes), los cuales tienen un argumento adicional (**self**), que generalmente tiene la forma de puntero o referencia al estado del objeto target. Todas las invocaciones con destino implícito se realizan sobre el parámetro **self**.

En los comienzos de 1980, Bjarne Stroustrup[1], propone una implementación de lo que él llamó *métodos virtuales*, es decir aquellos métodos cuyas invocaciones deben determinarse en tiempo de ejecución, con complejidad temporal constante.

El método consiste en que cada clase tiene asociada un vector de punteros a sus

[1] Creador e implementador original de C++.

métodos que requieran binding dinámico. Además cada objeto debería contener un campo adicional: un puntero a dicha tabla. Cada invocación dinámica se resuelve con un acceso extra a la tabla de punteros. El compilador computa estáticamente el desplazamiento (offset) constante para cada método dentro de la tabla.

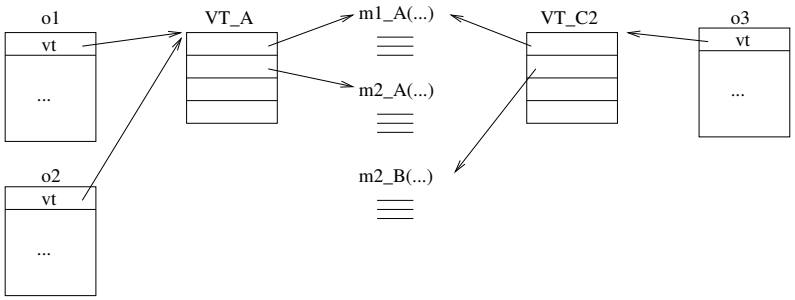


Fig. 9.6: Ejemplo de uso de tablas virtuales

La figura 9.6 muestra un escenario de esta implementación, donde los objetos `o1` y `o2` son instancias de una clase `A` y `o3` es una instancia de una clase `B` en la cual se ha redefinido el método `m2`.

Este método es uno de los más usados en lenguajes que soportan objetos y herencia, como Java, C#, Eiffel y otros.

Uno de los principales problemas con esta implementación es el tamaño de las tablas, ya que en sistemas grandes, con cientos de clases, el tamaño de las tablas puede alcanzar tamaños considerables en memoria. En presencia de herencia múltiple, el tamaño es un problema aún mayor.

Muchos lenguajes permiten controlar al programador controlar el tamaño de las tablas. C++ deja en manos del programador la decisión sobre qué métodos tendrán binding dinámico y cuáles no. Por omisión los métodos de C++ utilizan binding estático (la semántica no deseada en OOP). Un método definido `virtual` utilizará binding dinámico en sus invocaciones.

En otros lenguajes, por el contrario, su semántica por omisión es binding dinámico (como Java y Eiffel). Estos lenguajes permiten explicitar que sobre algunos métodos no será necesario aplicar binding dinámico usando las palabras reservadas `final` (en Java) o `frozen` (en Eiffel).

9.5 Control de acceso a elementos de una clase

La gran mayoría de los lenguajes que soportan objetos, proveen mecanimos para controlar el acceso a los componentes de una clase. Cada miembro de una clase se define dentro un alcance específico.

Generalmente cada miembro tiene un alcance por omisión y se puede alterar por medio de ciertas palabras reservadas, como por ejemplo, `public`, `protected` y `private` el cual define su visibilidad al resto del programa.

Un miembro `public` es visible para el resto del programa. Un miembro `private` es visible sólo dentro de la clase (o instancia). Un miembro `protected` es visible dentro de la clase y de sus clases derivadas.

Oz no tiene palabras reservadas para modificar su visibildad. Un método puede hacerse privado usando como rótulo una variable (ya que el compilador crea un identificador para la variable) o usando una variable precedida del operador ! (escape), lo que significa que la variable se ha definido fuera de la clase.

Eiffel tiene un mecanismo muy general, ya que permite agrupar los miembros (features) de una clase indicando su visibilidad especificando la lista de clases (tipos) que pueden acceder a esos miembros.

La especificación de una clase en una cláusula `feature` permite que cualquier instancia de esa clase o de sus clases derivadas, puede acceder al miembro.

En Eiffel, al igual que en Java, todas las clases derivan (por omisión) de una clase raíz. En Eiffel, esa clase raíz se denomina `ANY`, mientras que en Java se denomina `Object`.

C++ no tiene esa semántica, por lo cual permite al programador diseñar su propia jerarquía de clases partiendo desde su propia clase raíz.

9.6 Clases: módulos, estructuras, tipos

Como ya dijo anteriormente, una clase define un módulo y un tipo. Esta dualidad en la visión de una clase a menudo confunde a los programadores poco experimentados al momento de diseñar jerarquías de clases ya que es posible diseñar jerarquías con una visión estructural en lugar de tener una visión con respecto a una jerarquía de tipos de datos.

Es común encontrar en cierta bibliografía introductoria en OOP, ejemplos tales como que una clase `Circulo` hereda de una clase `Punto`. Desde un punto de vista estructural puede tener sentido, ya que es razonable pensar que un círculo contiene un punto (su centro). Desde el punto de vista de tipos de datos, justamente lo inverso es razonable: es posible pensar que un punto es un círculo de radio cero.

El ejemplo anterior muestra la dificultad de diseñar correctamente jeraquías de clases. La visión mas adecuada es la visión de tipos, ya que la herencia es una relación *es-un*, es decir que una instancia de una clase derivada, también es compatible (es del tipo) de su superclase. El tipo de datos de una clase incluye los subtipos correspodientes a sus subclases, por lo que la superclase es un tipo mas general, mientras que sus subclases definen subtipos.

La visión de clases como tipos satisface la *propiedad de sustitución*, es decir que cada operación que actúa sobre objetos de un tipo dado, también actúa sobre instancias de sus subtipos (subclases). Muchos lenguajes de POO, como C++, Java, Smalltalk, etc, están diseñados para esta visión.

El lenguaje Eiffel, está diseñado para soportar ambas visiones.

9.7 Polimorfismo

Es posible definir la herencia en términos de un sistema de tipos en base a una relación tipo-subtipo. Esto permite implementar abstracciones polimórficas - procedimientos, funciones y estructuras de datos (contenedores) - ya que una abstracción que se basa en una clase `A`, acepta instancias de `A` o de sus clases derivadas.

El polimorfismo basado en herencia requiere que las operaciones tomen referencias o punteros a objetos, en lugar de objectos por copia. Esto permite implementar fácilmente el concepto que una referencia o puntero a un objeto de tipo `A` toma el significado que es una referencia a on objeto de tipo `A` o alguno de sus derivados. Este tipo de polimorfismo se conoce como polimorfismo por inclusión, ya que se basa en la relación tipo-subtipo. Para poder implementar este mecanismo, es necesario que las invocaciones a métodos tengan binding dinámico.

9.8 Clases y métodos abstractos

Las clases y la herencia permiten la construcción en forma incremental de jerarquías de clases que representarán una jeraquía de tipos de datos.

Esta construcción incremental permite definir métodos generales que confían en otros métodos que podrán (o deberán) ser sobrecargados (especializados) en subclases.

Es frecuente que algunos métodos en una clase no puedan ser definidos, ya que su implementación dependerá de cada subclase en particular.

Definición 9.8.1 *Un método es abstracto si no tiene una implementación.*

Definición 9.8.2 *Una clase es abstracta si no permite tener instancias de ella. Esto es lo mismo que decir que tiene una implementación parcial.*

Es fácil de ver que si una clase contiene al menos un método abstracto, no podrá tener instancias, ya que esos objetos tendrían una implementación parcial, por lo que la clase deberá ser abstracta.

Las clases abstractas permiten definir *moldes* de las subclases y delega en ellas la responsabilidad de completar su implementación.

Uno de los principales objetivos de las técnicas de programación orientada a objetos es la reutilización de código, lo cual se logra maximizando las abstracciones (aún la procedural).

Una clase abstracta puede tener implementados muchos de sus métodos, los cuales serán reutilizados en las sublcases.

A continuación se muestra el uso de clases abstractas.

```
abstract class Persistent {
   ...
   abstract state get_state();
   save() { medium.save(get_state()); }
   ...
   storage medium;
   ...
};

class Person extends Persistent {
   ...
   state get_state() { return id+name+address; }
   ...
};
```

La clase abstracta `Persistent` define el método `save` el cual invoca a `get_state()`, el cual es abstracto. Cualquier clase concreta derivada de `Persistent` deberá implementar `get_state()`.

9.9 Delegación y redirección

La herencia no es el único mecanismo para definir abstracciones en forma incremental. La herencia requiere de especificaciones adecuadas de jerarquías de clases, lo que no siempre es fácil. Además requiere de binding dinámica.

Es posible lograr efectos similares con la herencia utilizando métodos mas simples como lo son la redirección (forwarding) y la delegación. Estos mecanismos se definen a nivel de objetos: si un objeto `o1` no comprende un mensaje `m`, éste se reenvía, en forma transparente, a un objeto `o2`.

La diferencia entre redirección y delegación es la forma cómo tratan a **self**.

- En redirección, `o1` y `o2` mantienen sus entidades separadas. Una invocación sobre **self** en `o2` se realiza sobre `o2`.

  La redirección puede ser implementada en Oz por medio del método `otherwise`.

- En delegación, `o1` y `o2` existe referencian a una misma entidad, por ejemplo, `o1`. Una invocación sobre **self** en `o2` se hará sobre `o1`.

  La delegación permite definir jerarquías a nivel de objetos, no clases. En lugar de hacer una jerarquía de clases, se definen objetos que delegan en otros (en tiempo de creación de los objetos).

Dados dos objetos, `o1` y `o2`, suponiendo que tienen un método `setDelagate`, tal que `{o2 setDelegate(o1)}` hace que el objeto `o2` delega sus invocaciones a `o1`. En otras palabras, `o1` se comporta como una superclase de `o2`.

Una propiedad importante de la delegación es que **self** se preserva, por lo que los métodos actúan sobre el estado del objeto que ha iniciado la delegación.

9.10 Reflexión

Un sistema (u objeto) es reflectivo si éste puede inspeccionar parte de su estado de ejecución dinámicamente. La reflexión puede ser puramente instrospectiva, es decir sólo podrá leer su estado interno, o intrusiva, es decir que puede modificar su estado.

La refleción puede hacerse a bajo o alto nivel. En el primer caso, podría ser que es posible ver los elementos de la pila semántica como clausuras (ambientes). Un ejemplo del segundo caso podría ser que es posible ver la memoria como un vector de enteros.

Algunos lenguajes modernos permiten instropección, como en Java y Eiffel. Otros lenguajes mas dinámicos ofrecen reflexión intrusiva, ya que es posible agregar, modificar y eliminar componentes de objetos dinámicamente, como por ejemplo en Python y Ruby.

Otros lenguajes como C++, permiten una reflexión muy limitada. Sólo permite inspeccionar el tipo de un objeto si se utiliza RTTI (*Run Time Type Information*).

9.11 Meta objetos y meta clases

La OOP es un área muy apropiada para aplicar reflexión a diferentes niveles. Por ejemplo, el sistema podría examinar (y posiblemente cambiar) la jerarquía de clases dinámicamente (por ejemplo en Smalltalk y Ruby), o cambiar la semántica de algunos mecanismos básicos, como por ejemplo la herencia (cómo se realiza la determinación de qué método invocar) y cómo se realizan las invocaciones.

La descripción de cómo se comporta un sistema de objetos en su nivel básico se conoce como un meta-protocolo.

La posibilidad de modificar el meta-protocolo brinda un gran poder descriptivo que puede tener aplicaciones muy interesantes, como depuración (debugging), personalización (customizing) y separación de conceptos (como por ejemplo, encriptar los mensajes de las invocaciones, introducir mecanismos de tolerancias a fallas o introducir aspectos[2].

[2]La programación orientada a aspectos permite modificar el sistema de ejecución para introducir computaciones en puntos específicos de un programa dado sin modificarlo, es decir *hacerlo por fuera*.

Las metaclases, en lenguajes como Smalltalk, Ruby o CLOS (Common Lisp Object System) dan una gran flexibilidad para implementar sistemas que son muy flexibles en tiempo de ejecución ya que permiten prácticamente la construcción y modificación de objetos y de sus jerarquías en tiempo de ejecución.

Los lenguajes que no disponen de metaclases, se denominan orientados a clases (y tal vez no necesariamente a objetos). Lenguajes como C++, Java o Eiffel, no proveen metaclases, aunque las características y mecanismos provistos por metaclases pueden ser provistos por métodos de clases específicas (como `INTERNALS` y `MEMORY` de Eiffel) o algunos métodos de la clase `Object` de Java.

El proyecto *Open C++*[8] tiene como objetivo proveer de mecanismos de metaclases para C++ por medio de un pre-compilador de metaclases (que genera clases C++).

9.12 Constructores y destructores

En la terminología de la OOP, se conoce como *constructor* a un método que se invocará automáticamente por el sistema en el momento de la creación de un objeto (tal como la operación `New` descripta anteriormente).

En la mayoría de los lenguajes orientados a objetos es posible definir varios constructores. Alguno de ellos se invocará (explícitamente o implícitamente) durante la creación de un objeto.

En algunos lenguajes, como C++ y Java, los constructores deben tener el mismo nombre de la clase y no retornan ningún valor (ni siquiera `void`). En ese caso, un constructor es sobrecargado.

Es común en la práctica de la programación que cuando un objeto esté por destruirse, se desee realizar algunas operaciones antes de su destrucción, tal como liberar recursos (cerrar archivos, conexiones, etc).

En C++ esto se logra por medio de los *destructores*. Un destructor es un método que tiene el mismo nombre que la clase sin parámetros (y sin valor de retorno). Es invocado automáticamente por el sistema inmediatamente antes que el objeto sea destruido (automáticamente por la reglas de alcance o por una operación `delete`).

Un destructor en C++ puede definirse como `virtual` ya que puede ser invocado en un contexto polimórfico donde se requiere determinar el destructor (método) correspondiente en forma dinámica.

Los lenguajes que proveen amnejo automático de la memoria, como Eiffel o Java, permiten que las clases redefinan un método provisto en la clase base del lenguaje. Por ejemplo, la clase base `Object` de Java provee un método `finalize()` el cual se invoca cuando el recolector de basura reclama el objeto.

9.13 Herencia múltiple

La herencia múltiple acarrea algunos problemas adicionales a la herencia simple. Cuando una clase `C` hereda de más de una clase, es posible que ocurran los siguientes inconvenientes:

1. *conflictos de nombres de miembros de clases*: en el caso que las superclases tengan algún miembro con nombre común.

2. *problema del rombo*: si dos (o más) clases, sean `B` y `C` heredan de una clase común (`A`) y una clase (`D`) hereda de `B` y `C`, se da el problema que `D` hereda por dos caminos diferentes de la clase `A`.

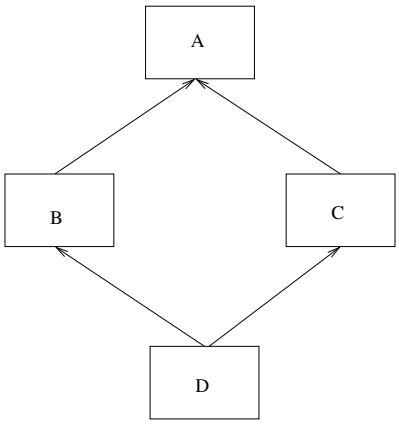


Fig. 9.7: Problema del rombo con herencia múltiple

Esto trae dos problemas: el primero es un ovbio conflicto de nombres y el segundo es la repetición de atributos. La figura 9.7 muestra tal situación.

Cualquier lenguaje de programación que soporte herencia múltiple deberá ofrecer mecanismos para resolver estos problemas. En el caso de C++, el problema de nombres no existe ya que es posible referenciar a un miembro específico de una clase utilizando el operador de alcance (símbolo ::).

El problema de la repetición de atributos se resuelve en C++ definiendo en una de las ramas de la herencia (en el ejemplo, podría ser en la clase `B`) que se hereda de la superclase (`A`) en forma virtual. Se usa la palabra reservada **virtual**, tal como se muestra a continuación.

```
class A {...};
class B : virtual public A {...};
class C : public A {...};
class D : public B, public C {...};
```

Eiffel soporta múltiples cláusulas de herencia y en cada una se pueden renombrar e indefinir *features* (attributos o métodos). Por ejemplo:

```
class A
feature {ANY}
  attr:T
  m(x:INTEGER)
    ...
end
class B
  inherith A
    ...
  end
  ...
end
class C
  inherith A
    rename m as m_of_A
  end
  ...
end
class C
  inherith B
    indefine attr1
  end
  inherith C
  end
end
```

9.14 El lenguaje Java (parte secuencial)

Java es un lenguaje orientado a objetos derivado de C++. Java se acerca mucho a un lenguaje orientado a objetos puro (casi todas las cosas son objetos, excepto las sentencias y los tipos de datos primitivos).

Los objetivos de Java y C++ son diferentes. Mientras que C++ permite la representación directa de los datos en hardware y una traducción de sus sentencias simple y directa a lenguaje de máquina, Java da una representación abstracta de los datos y realiza manejo de la memoria en forma automática.

Java soporta concurrencia nativa (threads) en un modelo de memoria compartida (shared memory) y computación distribuida sobre múltiples plataformas. Tiene un sistema de objetos pre-elaborado.

Para lograr los objetivos mencionados, un sistema Java tiene dos componentes:

- **el compilador** (javac): que traduce las definiciones de clases en los programas

fuentes (archivos con subfijo `.java`) a archivos objetos en formato *Java bytecode* (archivos con subfijo `.class`).

- **la máquina virtual** (java virtual machine o *jvm*): que se encarga de cargar, enlazar y ejecutar clases. La máquina virtual es un intérprete del lenguaje assembly virtual *java bytecode*. Cada plataforma con soporte para Java debe implementar su propia *jvm*.

Java es un lenguaje estáticamente tipado (al igual que C++), con clases, objetos pasivos y threads. A diferencia de C++, los valores en java pueden ser enteros, reales en punto flotante, caracteres (unicode), lógicos (booleans) y referencias a objetos.

No es posible declarar objetos directamente, sino referencias a objetos, lo que hace que el mecanismo de pasaje de mensajes sea por valor (las referencias se pasan por valor).

Una variable declarada tiene un valor inicial predefinido para cada tipo de dato (a diferencia de C++).

Soporta asignación simple de atributos utilizando la palabra `final`, lo cual es equivalente a los objetos constantes de C++. La misma palabra puede utilizarse para expresar que un método no puede sobrecargarse en clases derivadas[3].

El concepto `self` se denomina `this`, el cual es una referencia al objeto activo del thread corriente.

Las clases pueden contener atributos (llamados *fields* en la terminología Java) y métodos.

Cada definición de atributos o métodos puede tener propiedades (las cuales preceden a la declaración) de las siguientes categorías:

- **visibilidad**: como `public`, `protected` o `private` con el significado que se vio con anterioridad.
- **modificabilidad**: con `final` como se explicó mas arriba.
- **tiempo de vida**: con la palabra `static`, la que convierte el elemento (atributo o método) en *elemento de clase* y no de objetos. En el caso de atributos, esto permite definir campos compartidos (shared) entre todos los objetos. Cuando se aplica a un método, es posible invocarlo sin crear una instancia de su clase. Un ejemplo de su aplicación es la función `main`, la cual hace de función de arranque de un programa Java y es invocada por la jvm.

  Cualquier clase puede contener una función `main`. Esto quiere decir que un programa Java podría tener muchos puntos de inicio, dependiendo de la clase por la cual se de inicio al programa.

.

Una diferencia con C++, es que este permite al programador sobrecargar casi la totalidad de los operadores del lenguaje[4].

[3] Esto pone algún límite en las tablas de despacho.

[4] Salvo los operadores `::`, `->` y `..`

```
class Int {
   private int value;
   public Int(int value) { this.value = value; }
   public int get_value() { return value; }
   public set_value(int v) { value = v; }
   public set_value(Int other) { value = other.get_value(); }
}

class Example {
   public void sqrt(Int n)
    {
       n.set_value( n.get_value() * n.get_value(); )
    }

   public static void main(String[] args)
    {
      int x = 5;
      Int v = new Int(x+10);
      Example.sqrt(v);
      System.out.println("v=" + v.get_value());
    }
}
```

Fig. 9.8: Ejemplo de un programa Java.

La figura 9.8 muestra un ejemplo de un programa Java.

9.14.1 Herencia

Java soporta herencia simple, aunque es posible simular la herencia múltiple usando *interfaces*.

La herencia se expresa por la palabra `extends` (equivalente al `from`).

Una *interface* es una descripción de un protocolo. Contiene declaraciones de métodos (no atributos) sin definir (sin cuerpo).

Las interfaces se pueden heredar (inclusión textual) y una clase puede implementar una interface.

Esto último se refleja en la declaración de una clase utilizando la palabra `implements`.

No se deben confundir a las interfaces con clases abstractas. Una clase abstracta generalmente tiene una implementación parcial (de sus métodos) y puede tener atributos. Una interface no tiene estado.

A diferencia de C++, la semántica del sistema de objetos es ligadura dinámica (la adecuada), por lo que no hace falta declarar a qué métodos se les puede aplicar un

tipo de ligadura.

En el capítulo 10 se mostrarán ejemplos de herencia ya que analizarán los mecanismos de soporte a la concurrencia en Java.

9.15 Generecidad

Se conoce como generecidad a los mecanismos para definir algoritmos y tipos de datos en forma abstracta, en el sentido que pueden operar sobre cualquier tipo de datos. El (nombre) de los tipos de datos de las abstracciones son parámetros de la definición.

Se diferencian con definiciones polimórficas al estilo del polimorfismo paramétrico en que las definiciones genéricas son *esqueletos* de funciones y tipos de datos, a partir de los cuales el comilador generará versiones específicas en base a los tipos de datos con que se instancien.

Este mecanismo de polimorfismo también se conoce como *polimorfismo por instanciación*.

También se conoce a la generecidad como *polimorfismo estático* contrastando con el polimorfismo basado en herencia, también llamado *polimorfismo dinámico*.

Mas allá de sus diferencias, lo interesante es que ambos mecanismos se pueden combinar, lo que da como resultado un polimorfismo mas general.

Si bien la generacidad no es exclusiva a la OOP, se describe aquí ya que varios lenguajes de programación de este paradigma lo soportan, como por ejemplo, C++, Java, Eiffel y otros.

A modo de ejemplo, a continuación se muestra el uso de la clase `ARRAY` de la biblioteca estándar de Eiffel:

```
class A                        class ARRAY [T]
...                              ...
  vec1:ARRAY[INTEGER]            rep:NATIVE_ARRAY[T]
  vec2:ARRAY[STRING]             ...
...                            end
end
```

En C++ las definiciones genéricas se denominan *templates*. Estos lenguajes proveen en su biblioteca estándar un conjunto de definiciones genéricas tanto como algoritmos (funciones) como clases, las cuales generalmente representan estructuras de datos comúnmente usadas como pilas, listas, conjuntos, etc. Estas estructuras de datos en la jerga de la OOP se conocen como *contenedores (containers)*.

A continuación se describe el mecanismo de *templates* de C++.

9.15.1 Templates (plantillas) de C++

El mecanismo de plantillas de C++ puede aplicarse tanto a funciones como a clases y estructuras. La sintaxis de una plantilla sigue el siguiente esquema:

`template <generic-args> <definition>`

donde `<definition>` puede ser una definición de una función, clase o estructura. En la definición se pueden mencionar los argumentos de la plantilla.

Los argumentos de una plantilla se debe separar por coma. Un argumento genérico tiene la forma `typename T` donde `T` es en parámetro formal de la plantilla.

```
template <typename T>
T add(T const & v1, T const & v2)
{
   return v1 + v2;
}
```

```
template <typename T1, typename T2>
   struct pair {
       T1 first;
       T2 second;
   };
```

Fig. 9.9: Ejemplos de plantillas (templates) en C++.

La figura 9.9 muestra un ejemplos de una plantillas.

Se crea una instancia concreta (de una función o clase) cuando en una declaración o en una expresión se hace referencia parámetros de tipo con tipos concretos. Por ejemplo, la expresión

`add(5,x)`

crea una instancia de la función `int add(int const &v1, int const &v2) ...`, asumiendo que `x` es de tipo entero. La expresión

`add(s,string("..."))`

crea una instancia de `add` para el tipo string, la cual retornará la concatenación de sus argumentos, ya que el operador `+` está definido como la concatenación en la clase `string`.

La declaración

`pair<string,int> address;`

genera una instancia de la clase (o estructura) `pair` con `first` de tipo `string` y `second` de tipo `int`.

C++ provee en su biblioteca estándar un conjunto de plantillas, tanto de funciones como estructuras de datos (*abstract containers*). Este conjunto de plantillas se conoce como la `STL (Standard Template Library)`.

Muchas de las funciones definen algoritmos abstractos, como `sort`, `accumulate`, `find`, etc.

Entre los contenedores abstractos se encuentran `vector`, `list`, `stack`, `map`, `set` y otros.

Estos contenedores utilizan *iteradores (iterators)* para el acceso a los elementos que contienen.

Un iterador es una abstracción de un puntero, están definidos como clases genéricas y definen al menos operadores como `operator==()`, `operator++()` y `operator*()`

(igualdad, incremento y referenciación, respectivamente).

Otras clases abstractas definen *objectos-función*, objetos que representan funciones, lo que puede hacerse en C++ con aquellos objetos cuyas clases tienen sobrecargado el operador de invocación a función (operador `()`)

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

class case_less
{
    public:
        bool operator()(string const &left, string const &right) const
        {
            return strcasecmp(left.c_str(), right.c_str()) < 0;
        }
};

void print(string s)
{
    cout << s << endl;
}

int main(int argc, char *[]argv)
{
    vector<string> v(argv,argv + argc);
    sort(v.begin(), v.end(), case_less());
    for_each(v.begin(), v.end(), print)
}
```

Fig. 9.10: Ejemplos de uso de la STL.

La figura 9.10 muestra un ejemplo de uso de la STL.

Java ofrece *abstract containers* al estilo de Eiffel, con una sintaxis similar al de C++ aunque no tan poderoso en su expresividad.

Los templates de C++ definen una especie de *meta-lenguaje* en cual puede ser utilizado para *instruir* al compilador de cómo generar (y aún evaluar) código C++. Con este mecanismo es posible realizar evaluación en tiempo de compilación, lo que permite implementar una especie de evaluación parcial estáticamente.

9.16 Ejercicios

1. Dada la siguiente clase en Oz, mostrar su uso extendiendo el programa con la creación de objetos y sus invocaciones a métodos. Compilar y ejecutar.

```
class Counter
   attr val
   meth init(V)
     val := V
   end
   meth inc(V)
     val := @val + V
   end
end
```

2. En lenguajes como C++ o Java, podemos hablar de mensajes o son en realidad invocaciones a procedimientos? Justificar la respuesta.

3. Describa las ventajas y desventajas de manejar las invocaciones a métodos como mensajes versus invocaciones a procedimientos.

4. Mostrar con un ejemplo que en Java utiliza *dynamic dispatching* y C++ utiliza *static dispatching* por default. Como hacer en C++ para que un método trabaje con *dynamic dispatching*.

5. Determinar si el destino (target) de la siguiente expresión puede ser determinada estáticamente o no:

```
...
super.m();
...
```

   Justificar la respuesta.

6. Explicar como se puede resolver el problema del rombo (con herencia múltiple) en C++ y Python. Haga un experimento en ambos lenguajes.

7. Dado el siguiente programa, indicar que muestra por la salida estándar en cada uno de los casos. Corroborar mediante la ejecución del mismo.

```
public class B extends A2 {

  String m1 (B x) { return "BA2" ; }
  String m2 (B x) { return "Paso por B - m2(B) "+ this.m1(x) ; }
  String m2 (A2 x) { return "Paso por B - m2(A2) "+ this.m1(this) ; }
```

```
  public static void main(String[] args) {
    A2 a = new A2 ();
    A2 b2 = new B ();
    B b3 = new B();
    System.out.println (a.m2 (b3));      (1)
    System.out.println (b2.m2 (b2));     (2)
    System.out.println (b3.m2 ((B)b2));  (3)
    System.out.println (b3.m2 (a));      (4)

  }
}

class A2 {
  String m1 (A2 x) { return "AA2" ; }
  String m2 (A2 x) { return "Paso por A - m2(A2) "+ this.m1(x) ; }
}
```

8. Dado el siguiente programa en java, indicar que muestra por salida estándar en cada uno de los casos.

```
class A{
 public void type(){
  System.out.println("Soy A");
 }

 public void type(A _a) {
   _a.type();
 }

}


class B extends A{
 public void type(){
  System.out.println("Soy B");
 }
}

class C extends B{
 public void type() {
   System.out.println("Soy C");
  }

public void type(B _b) {
```

```
    _b.type();
   }
  }

  class Test{

   public static void main(String[] args){
     A a = new A();
     B b= new B();
     B c = new C();

     a.type(b);
     b.type(a);
     b.type(b);
     c.type((A) c);
     c.type((C) c);
   }

  }
```

9. Dé un ejemplo de una función en Java con número de parámetros variable.

10. Introspección. Utilizando las funciones del módulo `inspect`, experimente en Pyhton como verificar si un argumento es una clase, si un argumento es un método de una clase, mostrar la jerarquia de clases dada una clase, etc.

11. De un ejemplo en Python de una clase genérica.

12. Escribir un programa en C++ que ordene los argumentos que pueda recibir desde la línea de comandos durante su invocación. Deberá usar los algoritmos de la STL (Standard Template Library) `sort` y alguno de los objetos-funciones (objetos que tienen sobrecargado el operador `()`) de comparación (greater-equal, less, etc).

13. De un ejemplo en Python de un algoritmo generico usando variables de tipo.

Ejercicios Adicionales

14. Modificar la clase lista de C++ para almacenar valores de un tipo T concreto (no de sus derivados).

15. Implementar en Java el TAD Pila de un tipo T. Muestre un ejemplo de uso.

16. Dar un ejemplo de cómo se puede implementar un diseño con herencia múltiple en Eiffel.

17. Escribir un programa que sume y multiplique los elementos de un vector. Los elementos deberán estar almacenados en un objeto de tipo `vector<int>`. Deberá utilizar el algoritmo genérico `accumulate`.

18. Implementar en Java y C++ el TAD *lista* (heterogénea). Definir una clase abstracta y al menos dos implementaciones: sobre arreglos y nodos enlazados.

    Mostrar ejemplos de uso.

Capítulo 10

Concurrencia

En este capítulo se abordan conceptos sobre ejecución concurrente de varias unidades de ejecución.

Algunos programas, se pueden escribir más fácilmente como un conjunto de actividades que ejecutan independientemente. Tales programas se denominan concurrentes.

Programas que interactúan con el ambiente como agentes (o servicios), interfaces gráficas de usuario (GUIs), sistemas operativos, etc, se escriben naturalmente como un sistema concurrente.

Cada una de esas actividades ejecutándose en forma concurrente puede interactuar, en algún punto del programa, con otras. Por ejemplo, pueden compartir ciertos recursos (variables, archivos, etc) y en muchos casos es necesario un mecanismo de sincronización para el acceso a los mismos.

Los programas se escriben siguiendo alguna lógica de comportamiento de sus partes, como por ejemplo, siguiendo diseños cliente-servidor o productor-consumidor.

Los modelos de ejecución concurrentes pueden clasificarse en dos esquemas:

1. *memoria compartida (shared memory)*: cada actividad o módulo concurrente del sistema accede a un área de memoria común. El mecanismo natural de comunicación entre las actividades es el uso de variables compartidas. En este modelo, cada actividad se denomina comúnmente un *hilo (thread) de ejecución.*

2. *pasaje de mensajes (message passing)*: las actividades tienen su propia área de memoria y se comunican por medio de *mensajes*, los cuales se envían por *canales de comunicación (channels).* En este modelo, es común que las actividades se denominen *procesos* o *tareas (tasks).* Este modelo es comúnmente utilizado en sistemas distribuidos débilmente acoplados como las redes. Las reglas que definen formatos de los mensajes y las políticas de interacción entre los procesos, definen un *protocolo* de comunicación.

Como se verá mas adelante, ambos modelos son equivalentes en poder expresivo, ya que cada uno de ellos puede definirse en términos del otro.

10.1 Concurrencia declarativa

En ésta sección se extenderá el lenguaje núcleo del modelo declarativo con concurrencia y permancerá declarativo, es decir que se mantienen los métodos de razonamiento del modelo declarativo.

El mecanismo confía en el concepto de *variables data-flow (flujo de datos)* las cuales pueden ser ligadas a un único valor. Esto arroja dos resultados importantes:

- Los resultados del programa serán los mismos, independientemente si se utiliza concurrencia o no.
- Lo nuevo es que un programa puede ser computado en forma incremental, a medida que arriban los datos de entrada. Por ejemplo, un thread puede leer datos de la entrada y otro thread procesar esos datos. Es un ejemplo de lo que se conoce como un *stream*. Esto implica que un programa puede no terminar, ya que si se dejan de ingresar datos los threads quedarán suspendidos hasta que se ingresen nuevos datos.

El lenguaje será extendido incluyendo

- una nueva sentencia: **thread <s> end**. Un thread es una sentencia de ejecución en una nueva pila (stack) de la máquina. La máquina se extiende para tener varias pilas semánticas en lugar de una como en el modelo secuencial.
- incluir un nuevo orden de ejecución: por medio de disparadores (triggers) y una nueva instrucción primitiva `{ByNeed P X}`, la cual permitirá realizar computaciones bajo demanda, como ejecución perezosa (lazy).

10.1.1 Semántica de los threads

En la máquina extendida se mantiene la memoria de asignación única σ, los ambientes y los formatos de una pila semántica $(< s >, E)$.

Ahora la máquina soportará múltiples pilas semánticas, por lo que un estado de ejecución será un par (MST, σ), donde MST es un multiset (conjunto con elementos con ocurrencias múltiples). Es necesario un multiset porque es posible tener, en un momento dado, múltiples pilas semánticas con el mismo contenido.

Una computación sigue siendo una secuencia de estados de ejecución.

El estado inicial de la máquina sigue conteniendo la sentencia principal del programa, es decir es un estado de la forma $([< s >, \emptyset], \emptyset)$ (el multiset tiene un único stack, con la memoria vacía y es stack tiene el estado de ejecución formado por la sentencia principal del programa con un ambiente vacío).

La máquina, en cada ciclo de ejecución, selecciona una pila semántica y ejecuta la sentencia del tope tal como lo hacía en el modelo secuencial[1].

[1] La única acción adicional es la selección de la pila semántica al inicio del ciclo.

Esta selección la realiza una función interna de la máquina denominado *planificador (scheduler)*.

La máquina termina su ejecución si todas las pilas semánticas están en estado *finalizadas (terminated)*.

La semántica de la sentencia *thread* se define como

$$(\{[\text{thread } <\text{s}> \text{ end,E}] + ST'\} \biguplus MST', \sigma) \rightarrow (\{[(<\text{s}>,\text{E})]\} \biguplus ST \biguplus MST, \sigma)$$

donde el operador $\biguplus$ es la unión de multisets y el operador + es la concatenación de elementos del stack.

La ejecución de una sentencia thread crea un estado donde el cuerpo del thread se ejecuta en el contexto de una nueva pila semántica.

El administrador de la memoria requiere ser extendido:

- Una pila semántica *terminada (terminated)* puede ser eliminada.
- Una pila semántica bloqueada puede reclamarse si es que su condición de activación depende de una variable inalcanzable. Es fácil de ver que una pila semántica en estas condiciones nunca podrá pasar al estado *activo (runnable)*.

En este modelo toman importancia las sentencias *bloqueantes o suspendibles*. En el modelo secuencial, una sentencia al bloquearse tiene como efecto la suspención de la ejecución de la máquina, ya que disponía de una única pila semántica.

En el modelo concurrente, una pila semántica puede pasar a estado suspendido (por ejemplo en una sentencia **if** la variable de condición puede no estar ligada), pero en el futuro la máquina podrá ejecutar otra sentencia (de otra pila semántica) que ligue tal variable, lo que hará que la pila anterior pase al estado *runnable* nuevamente.

Este es un mecanismo de sincronización entre threads y se conoce como *variables data flow* o *data driven* ya que se basa en el control de flujo de datos de las variables.

10.1.2 Orden de ejecución

Dado un programa, en el modelo concurrente podemos obtener diferentes secuencias de estados de ejecución (dependiendo del thread elejido en cada paso de ejecución).

Cada secuencia se denomina una *traza* de ejecución. Estamos asumiendo que nuestra máquina básicamente sigue operando en forma secuencial, pero desde el punto de vista del programador o usuario, es posible ver que los diferentes threads evolucionan simultáneamente.

La idea de simultaneidad depende de la visión del pasaje del tiempo. En una máquina como la que se ha definido, la idea de simultaneidad está dada por la ejecución de sentencias de diferentes threads de ejecución en cada paso. Estas operaciones

realizadas a gran velocidad, tal como lo pueden hacer las computadoras actuales, pueden dar la ilusión de ejecución simultánea de los diferentes threads.

Esta forma de ejecución secuencial de sentencias de diferentes threads en cada ciclo, se denomina *intercalación (interleaving)*.

Un sistema de computación con varias máquinas, operando simultáneamente, permitirá ejecución *realmente* simultánea o en paralelo, por lo que estos sistemas se denominan sistemas paralelos.

En ambos sistemas de computación (única o múltiples máquinas), el orden de las computaciones de los diferentes threads o programas está controlada por el sistema, no por el programador o usuario, por lo que lleva a que la ejecución sea *no determinística*.

El no determinismo en el orden de ejecución de las sentencias, puede hacer que los resultados de un programa sean diferentes en dos trazas de ejecución distintas. En el modelo concurrente declarativo, ésto no puede suceder, ya que las variables toman un único valor (no pueden ser modificadas), por lo que se conoce como *determinismo no observable*.

Dado un programa concurrente, sus pasos de ejecución forman un *orden causal*, el cual es un orden parcial (a diferencia de un programa secuencial que forma un orden total).

Definición 10.1.1 *En un orden causal una sentencia s_i precede a otra s_j si ambas pertenecen al mismo thread y en él s_i se debe ejecutar antes que s_j.*

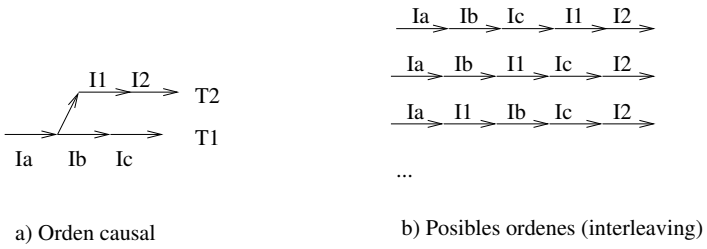


Fig. 10.1: Orden causal e intercalación (interleaving)

La figura 10.1 muestra la relación de un orden causal de los threads `T1` y `T2` y algunas posibles secuencias dadas por interleaving. La relación de orden causal del ejemplo, puede describirse por extensión como $\{(I_a, I_b), (I_a, I_1), (I_b, I_c), (I_1, I_2)\}$.

Definición 10.1.2 *Un conjunto de ligaduras en la memoria se denomina una* ***restricción (constraint)****.*

Definición 10.1.3 *Para cada variable x y cada restricción c, $values(x, c)$ contiene el conjunto de todos los valores posibles de x tal que se verifique c.*

Definición 10.1.4 *Dos restricciones c_1 y c_2 son* ***lógicamente equivalentes*** *si*

1. *contienen las mismas variables, y*
2. *para cada variable x, $values(x, c_1) = values(x, c_2)$.*

En el modelo concurrente declarativo pueden ocurrir dos cosas para todas las ejecuciones posibles de un programa:

1. puede no terminar, o
2. puede terminar parcialmente (computar valores parciales) y dichos valores son lógicamente equivalentes.

Esto implica que el no determinismo en el orden de las ejecuciones de las sentencias del programa concurrente, no es visible, ya que siempre se obtienen resultados equivalentes.

10.2 Planificación de threads (scheduling)

Un plainificador o menejador de threads debería ser justo (fair), en el sentido que cada thread eventualmente deberá progresar. Un scheduler en la práctica debe tener otras características deseables. Sheduling, actualmente es un campo muy activo de investigación.

Un scheduler debería ser parametrizado para poder implementar políticas de planificación con prioridades, contabilidad (tiempos de ejecución de cada thread) e implementar el cambio de threads (context switch) muy eficientemente.

Algunos sistemas permiten asignar prioridades a los threads, ya sea tanto estática como dinámicamente.
A modo de ejemplo, muchos sistemas interactivos van priorizando los threads o procesos orientados a entrada-salida sobre los orientados a uso de cpu intensivos, brindando así mejores tiempos de respuesta a las sesiones interactivas.

Si bien en la máquina abstracta definida en este documento se establece que en cada ciclo de ejecución se selecciona una de las pilas semánticas, en la práctica hacer un cambio de contexto por cada instrucción haría que la sobrecarga (overhead) del scheduler sea mayor que el requerido para procesar los threads en sí.
Por esto, en general un scheduler selecciona otro thread (o pila semántica) por dos posibles motivos:

1. el thread corriente se suspende, ya sea por una sentencia suspendible o por una operación de entrada-salida la cual puede tomar un largo tiempo (o indefinido) o
2. el thread ya ha tenido el control por demasiado tiempo, por lo que es momento de darle el control a otro.

La segunda causa establece que el scheduler debe determinar un intervalo de tiempo máximo de ejecución para cada thread. Este intervalo se denomina *time slice o quantum*. El intervalo de tiempo puede determinarse estáticamente o dinámicamente. Muchos schedulers modernos calculan el intervalo en forma dinámica, generalmente es una función que depende de varios parámetros como velocidad de la CPU, prioridad del thread, etc.

10.3 Control de ejecución

Los threads se conocen como un mecanismo de *concurrencia cooperativa*. Los programas que utilizan threads se programan de tal forma donde cada thread aporta a un fin común con los demás.

En un sistema operativo, en cambio, cada programa tiene un fin individual, por lo que los diferentes programas en ejecución *compiten* por los recursos del sistema. En un sistema operativo, cada programa en ejecución se denomina *proceso* o *tarea (task)*.

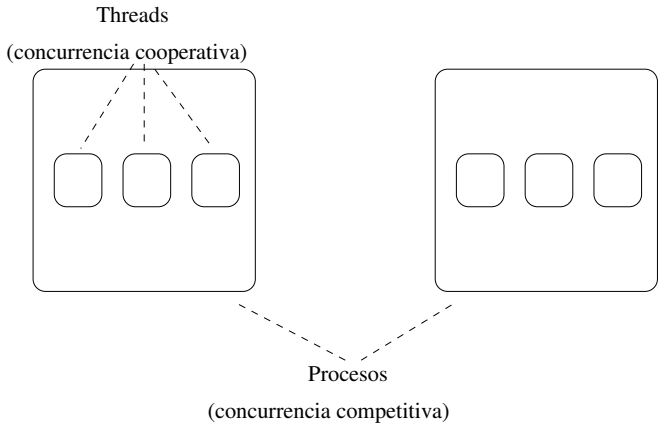


Fig. 10.2: Procesos o tareas y threads

Para dar algún grado de control sobre la ejecución de los threads en un programa, generalmente los sistemas de threads ofrecen algunas funciones sobre threads. Oz ofrece el módulo `Thread` que contiene las operaciones sobre threads (entre otras) que se listan en la tabla 10.3.

10.3.1 Corrutinas

Una corrutina es un thread no interrumpible. Una corrutina abandona la cpu en forma voluntaria.

Las corrutinas tienen dos operaciones fundamentales:

Operación	Descripción
{Thread.this}	Retorna el nombre del thread corriente
{Thread.state T}	Retorna el estado del thread T
{Thread.suspend T}	Suspende al thread T
{Thread.resume T}	Activa al thread T
{Thread.preempt T}	Interrumpe (quita el control) al thread T
{Thread.terminate T}	Finaliza (inmediatamente) al thread T
{Thread.wait T}	Espera por la terminación del thread T

Fig. 10.3: Algunas operaciones sobre threads.

- **Spawn P**: crea una nueva corrutina con `P` como procedimiento principal de la corrutina y retorna un identificador de la corrutina creada.

- **Resume C**: transfiere el control a la corrutina `C`.

Cada corrutina tiene la responsabilidad de transferir el control a las demás corrutinas que cooperan con un fin común.

Se puede apreciar que un sistema de corrutinas es simplemente un mecanismo explícito de transferencia de control a diferentes unidades de ejecución.

10.3.2 Barreras

Una barrera (barrier) es una operación de control de ejecución concurrente que permite establecer un punto de encuentro (punto de sincronización) de diferentes componentes concurrentes (threads o procesos). Una barrera permite establecer un punto de ejecución que requiere que otros componentes alcancen antes de poder continuar.

Una barrera también se conoce como *rendezvous*.

```
...
{Barrier [X1 X2 ... Xn]}
```

Fig. 10.4: Ejemplo de una barrera (rendezvous).

La figura 10.4 muestra un ejemplo de una barrera usando variables data-flow, donde el procedimiento `Barrier` suspende al thread corriente hasta que las variables `X1, X2, ...Xn` estén ligadas.

Las barreras generalmente definen puntos de sincronozación en base a requerimientos del computación del programa: dada una barrera, el thread no puede continuar hasta que los demás threads (cooperativos) no hayan finalizado la computación de ciertos valores.

10.3.3 Ejecución perezosa (lazy)

En el contexto de evaluación perezosa, una expresión se evalúa cuando su resultado es requerido en alguna parte del programa.

En esta sección se desarrolla una estrategia de control de ejecución mas general, llamada *ejecución lazy*,que la evaluación lazy, ya que ésta última estrategia se aplica a un contexto de ejecución secuencial.

El modelo de ejecución lazy extiende el modelo concurrente declarativo con un nuevo concepto: disparadores por necesidad (by need triggers).

Para poder dar soporte a los disparadores por necesidad, se incluirá una nueva operación primitiva a la máquina abstracta concurrente: `{ByNeed P Y}`, la cual tiene el mismo efecto que la sentencia {**thread** `P Y` }, excepto para scheduling, en el sentido que el procedimiento `P` será ejecutado (planificado) solamente si el valor `Y` es necesitado.

A continuación se describe la semántica de la operación `{ByNeed P Y}`.

Se extiende la memoria para disponer de una *memoria de triggers*, τ.

La semántica de la sentencia $(\{ByNeed\ <x><y>\}, E)$ es

1. si $E(<y>)$ no es determinado, crear el trigger $trig(E(<x>), E(<y>))$ a la memoria de triggers τ.

2. si $E(<y>)$ es determinado, crear un nuevo thread con la sentencia inicial (o cuerpo) $(\{<x><y>\})$.

Un trigger se activa cuando existe un valor $trig(x, y)$ y se detecta una necesidad de y, es decir, existe un thread que está suspendido por y o se está ligando a y.

En la activación de un trigger $trig(x, y)$ se ejecutan las siguientes acciones:

1. Eliminar $trig(x, y)$ de la memoria de triggers τ.

2. Crear un nuevo thread cuya sentencia inicial sea $(\{<x><y>\}, \{<x> \rightarrow x, <y> \rightarrow y\})$. O sea que produce la invocación al procedimiento ligado a x con el argumento ligado a y.

Esta extensión de la máquina requiere que se extienda el manejador de la memoria (garbage collector):

- Una variable x es alcanzable si $trig(x, y) \in \tau$ e y es alcanzable.

- Si la variable y se torna inalcanzable y el $trig(x, y) \in \tau$, el trigger $trig(x, y)$ se puede eliminar de la memoria τ.

Es posible implementar ejecución perezosa (lazy) usando `ByNeed`. Una función lazy es evaluada sólo si se requiere su resultado.

La abstracción lingüística `fun lazy {F args} <s> end` define a la función `F` en términos de `ByNeed`.

Es bueno aclarar que la *evaluación lazy* se diferencia de la *ejecución lazy* presentada aquí en el sentido que la primera se basa en un ambiente de ejecución secuencial, mientras que la última en un ambiente de ejecución concurrente.

Algunas aplicaciones de la ejecución lazy son:

- **Lazy streams**: un *stream* es una lista de mensajes o valores, potencialmente infinita, (una lista cuya cola está dada por una variable no ligada). Las tuberías (pipes) de los sistemas tipo Unix son un ejemplo de streams. Es común en algunos lenguajes funcionales que se definan listas por comprensión, las cuales se evaluarán en forma perezosa.

- **Operadores de alto orden**: como iteradores o filtros que aplican una función a cada elemento de una lista, la cual se va construyendo bajo demanda.

- **Enlazado dinámico (dynamic linking)**: utilizado comúnmente para aplicaciones basadas en el concepto de *componentes*. El código fuente de un programa consiste de un conjunto de especificaciones (atributos, métodos y propiedades) denominados funtores. Una aplicación en ejecución consiste de componentes instanciados, denominados módulos. Un módulo se puede representar como un registro que agrupa las operaciones del módulo (en cada campo). Los componentes se ligan bajo demanda, es decir cuando un módulo se carga en memoria y se instancia con un funtor. En el primer intento del programa de acceder a un campo (operación) de un módulo, se realiza el enlace.

10.4 Aplicaciones de tiempo real

Un programa de tiempo real (real time) define un conjunto de operaciones que tienen que completarse antes de determinados instantes de tiempo (vencimientos).

Una aplicación se dice que es de tiempo real blando (soft real time) cuando las operaciones deberán completarse antes de los vencimientos correspondientes en la mayoría de los casos.

Por el contrarrio, una aplicación de tiempo real duro (hard real time), requiere que *siempre* deberán completarse las operaciones antes de los vencimientos correspondientes.

Las aplicaciones de tiempo real duro, como por ejemplo, aplicaciones de control en vehículos, equipos médicos, sistemas de comunicaciones, etc, requiere soporte tanto de hardware como de software (sistema oeprativo).

Las aplicaciones de tiempo real blando no tienen vencimientos estrictos y pueden implementarse usando las siguientes operaciones (definidas en el módulo `Time`):

- `{Delay I}`: suspende la ejecución del thread que la ejecuta por al menos `I` microsegundos.

- `{Alarm I U}`: crea un nuevo thread que ligará `U` (a **unit** después que hayan transcurido al menos `I` microsegundos.

- `{Time.time}`: retorna el número de segundos desde una fecha dada (ej: el comienzo del año corriente).

10.5 Concurrencia y excepciones

La concurrencia tiene que estar relacionado al mecanismo de excepciones. A modo de ejemplo, supongamos que un thread ejecuta la sentencia `X=1` y otro thread ejecuta `X=2` (sobre la misma variable `X`).

En este caso una de las dos operaciones de ligadura deberá fallar (la que se ejecute después). Por esto el thread en el cual falle la operación de *bind* deberá disparar una excepción, dándole la opertunidad al thread de recuperarse, si es que atrapa la excepción.

Un problema adicional surge con ejecución por necesidad. Cuál debería ser el comportamiento de un disparo de un trigger y la computación de la función involucrada falla?

Si la variable a ligar con el resultado de la función no se liga, el thread que demanda el valor de la función no logrará su objetivo. Si la ejecución de la función asociada al trigger falla, Â¿a qué valor debería ligarse la variable a asociar al resultado de la función?

Una solución posible es ligar la variable a ligar con el retorno de la función a un valor especial.

El valor especial podría ser `cannotCalculate`. Ese valor debería ser retornado por alguna función primitiva de la máquina abstracta. Esa función se denominará `FailedValue`.

Ahora, con esta función es posible definir una función más robusta de ejecución por necesidad, como se muestra a continuación:

```
proc {ByNeed2 P X}
   {ByNeed
       proc {$ X}
          try
             Y in {P Y} X=Y
          catch E then X={FailedValue E}
          end
       end
       X
    }
end
```

El procedimiento `ByNeed2` se invoca de la misma manera que `ByNeed`, pero encapsula la excepción en un valor `FailedValue`.

10.6 Sincronización

Cuando un thread necesita un valor calculado por otro thread, el primer deberá esperar hasta que el resultado esté disponible. Se dice que los threads se sincronizan según la disponibilidad de resultados.

La sincronización es uno de los principales conceptos de la programación concurrente.

Definición 10.6.1 *Sean dos threads T1 y T2, con sentencias* $\alpha_1, \alpha_2 \ldots$ *y* $\beta_1, \beta_2 \ldots$, *respectivamente, un* ***punto de sincronización*** *entre* α_i *y* β_j, *si en cada traza de ejecución posible (interleaving),* α_i *se ejecuta antes que* β_j.

La sincronización puede hacerse de dos maneras:

- *dataflow*: (con evaluación estricta). Las operaciones que requieren un valor, deberán esperar hasta que el valor esté disponible.
- *bajo demanda*: (ejecución peresoza). El intento de ejecución de una operación, causa la evaluación de sus argumentos. El cálculo de los argumentos, causa un punto de sincronización entre las operaciones.

En un programa, la sicronización puede ser

- *implícita*: los puntos de sincronización no son visibles en el texto del programa, como en el modelo de concurrencia declarativa descripto hasta ahora.
- *explícita*: los puntos de sincronización son visibles en el programa y generalmente toman la forma de operaciones que implementan cerrojos (locks) o monitores, como se describen en la sección 10.7.

10.7 Concurrencia con estado compartido

La concurrencia en el modelo con estado (variables mutables), crea problemas para el desarrollo de programas. Si bien desde el punto de vista computacional este modelo es equivalente al de concurrencia con pasaje de mensajes, ya que éste último requiere el concepto de estado para representar los puertos, el estilo de programación es muy diferente, por lo que es lógico considerarlos modelos diferentes.

A modo de ejemplo, la figura 10.5 muestra un ejemplo en donde dos threads actualizan una variable compartida.

La salidas posibles del programa pueden ser: 0, 1 o 2. Se deja al lector el ejercicio de verificar esta afirmación.

Esto muestra que la salida depende del orden de ejecución (trazas) de las sentencias, lo cual lleva a que se produzca no determinismo observable.

Sucede que ya no es posible lograr que dos restricciones sean lógicamente equivalentes para una variable x.

```
local X in
   X := 0
   thread X := X + 1 end
   thread X := X * 2 end
   {Browse X}
end
```

Fig. 10.5: Ejemplo una condición de carrera.

El problema es que varios threads pueden estar actualizando una misma variable y, como en el ejemplo mostrado, pueden ocurrir *condiciones de carrera (race conditions)*.

La situación mostrada en el programa hace que no sea posible escribir un invariante con precisión ya que en un punto de programa dado, el valor de una variable puede ser incierto.

```
class Stack
   attr S
   attr top
   meth Push(V) S := V|S top := top + 1 end
   meth Pop(?V) in local S1 in S = V|S1 S := S1 top := top - 1 end
end
```

Fig. 10.6: Ejemplo una condición de carrera.

En el caso de la ejecución concurrente de operaciones sobre estructuras de datos, como por ejemplo operaciones sobre una pila, como se muestra en la figura 10.6, el invariante (ej: $pushes - pops = top$) puede invalidarse en una traza de ejecución determinada. Nuevamente, se deja al lector el ejercicio, dar una secuencia de ejecución que invalide dado.

Para poder especificar invariantes mas precisos en un programa, se requiere que se introduzcan puntos de sincronización, es decir que se secuencializen ciertas operaciones o que se garanticen que ciertas secuencias de sentencias, sobre ciertos datos compartidos, no puedan ser interrumpidas por el scheduler. Estos bloques de secuencias que deben ejecutarse sin interrupción se conocen como *regiones críticas* y se dice que deben ejecutarse en forma *atómica* (con respecto a los demás threads que acceden a los mismos recursos o variables).

La programación concurrente con estado compartido consiste principalmente en el reconocimiento y sincronización de las regiones críticas.

El reconocimiento de regiones críticas requiere que se indentifiquen puntos del programa en el que podrían ocurrir condiciones de carrera. Esta actividad es realmente difícil, como se puede apreciar en las listas de errores reportandos en proyectos de software como sistemas operativos, servidores de bases de datos, etc.

10.7.1 Primitivas de sincronización

Este modelo requiere de operaciones que permitan sincronizar los threads. Es posible clasificarlos en dos grandes grupos:

- **Cerrojos (locks)**: permiten agrupar secuencias de sentencias o instruciones con acceso exclusivo.
  Las funciones de los cerrojos en general son tres: *init()* (o *newlock()*), *lock()* y *unlock()*. El cerrojo se inicializa (o crea) con la función *init()* (o *newlock()*). Luego cada proceso/hilo debe llamar a la función *lock()* antes de acceder a los datos protegidos por el cierre. Al finalizar su sección crítica, el dueño del cerrojo debe desbloquearlo mediante la función *unlock()*.

- **Semáforos**: un semáforo, inventados por Dijkstra, es una variable (de tipo entera) especial protegida (ADT), el cual puede ser manipulada por tres operaciones:

  - `init(s,n)`: inicializa el semáforo con el valor `n`.
  - `P(s)`: intenta decrementar en uno el valor del semánforo, si es que su valor es positivo. En otro caso el thread invocante deberá esperar (bloquearse) hasta que sea desbloqueado por otro thread.
  - `V(s)`: incrementa en uno el valor del semáforo y despierta a posibles procesos bloqueados por el semáforo.

  Las operaciones `P` y `V` se corresponden con los nombres originales en holandés que les dio Dijkstra, por *Proberen* (probar) y *Verhogen* (incrementar), respectivamente.

  Intuitivamente, un semáforo representa el número de recursos disponibles y por lo tanto permite hasta n threads acceder a la región crítica simultáneamente.

  Un semáforo debe mantener el siguiente invariante:

  $$Vc(s) \leq s \leq Pc(s) + initial\_value(s)$$

  donde $Vc(s)$ es la cantidad de operaciones V realizadas sobre el semáforo s y $Pc(s)$ es el número de operaciones P realizadas sobre s.

- **Monitores**: La característica principal es que sus métodos son ejecutados con exclusión mutua. Lo que significa, que en cada momento, un hilo como máximo puede estar ejecutando cualquiera de sus métodos.
  Para que resulten útiles en un entorno de concurrencia, los monitores deben incluir algún tipo de forma de sincronización. Por ejemplo, supóngase un thread que está dentro del monitor y necesita que se cumpla una condición para poder continuar la ejecución. En ese caso, se debe contar con un mecanismo de bloqueo del thread, a la vez que se debe liberar el monitor para ser usado por otro hilo. Más tarde, cuando la condición permita al thread bloqueado continuar ejecutando, debe poder ingresar en el monitor en el mismo lugar donde fue

suspendido. Para esto los monitores poseen variables de condición que son accesibles sólo desde adentro. Generalmente existen dos funciones para operar con las variables de condición:

  - *cond_ wait(c)*: suspende la ejecución del proceso que la llama con la condición *c*. El monitor se convierte en el dueño del lock y queda disponible para que otro proceso pueda entrar.
  - *cond_ signal(c)*: reanuda la ejecución de algún proceso suspendido con *cond_ wait* bajo la misma condición *c*. Si hay varios procesos con esas características elige uno. Si no hay ninguno, no hace nada.

- **Regiones críticas condicionales**: son abstracciones de uso de monitores en forma de sentencias de un lenguaje. Un región crítica está protegida con un lock asociado a una condición. Un thread intentando ingresar a la región crica deberá esperar hasta que la condición sea verdadera.

  Generalmente tienen la forma de `region ... when <cond> <s> end`.

- **Transacciones**: una transacción es una secuencia de operaciones que pueden ejecutarse con éxito (`commit`) o ser abortadas. En el caso que se produzca una salida anormal (`abort`) el estado se retrotrae al estado previo del inicio de la transacción.

El lenguaje Oz ofrece las siguientes primitivas que implementan cerrojos reentrantes[2].

- `{NewLock L}`: crea un nuevo lock `L`.
- `{IsLock L}`: retorna **true** si `L` referncia un lock.
- `lock X then <s> end`: protege a la sentencia `<s>` con el lock `X`. Un thread ejecutando una sentencia protegida por un lock `X` impide que otro thread acceda (debe esperar) al cuerpo de la sentencia lock sobre la misma variable.

La implementación de primitivas de sincronización, como locks y semáforos, requiere de mecanismos mas primitivos aún para garantizar la atomicidad de sus implementaciones. Cabe hacer notar que una implementación de las operaciones de lock y unlock o P y V de semáforos, tienen condiciones de carrera sobre sus propias variables de estado.

En muchos sistemas o lenguajes se implementan como primitivas en donde la atomicidad de sus operaciones está garantizada, por ejemplo, desabilitando momentáneamente el scheduler y a veces también deshabilitando interrupciones (de hardware), ya que estas últimas podrían crear una condición de carrera sobre el acceso a algún recurso.

Con primitivas como `lock` y `unlock` y algunas operaciones de control sobre threads (como `Thread.wait` y `Thread.resume`) es posible implementar todos los demás mecanismos de más alto nivel, como monitores y regiones críticas condicionales.

[2] Un lock reentrante permite que se aniden locks por parte del mismo thread, sin bloquearlo.

10.8 Concurrencia con pasaje de mensajes

Un sistema concurrente puede definirse como un conjunto de componentes (generalmente denominados procesos) separados, cada uno con su memoria local (independiente) y que se comunican entre sí por medio de mensajes.

Este enfoque permite una mayor modularización de los programas y aún una mayor independencia en la defínción de cada uno de las partes del sistema. Cada componente podría implementarse en un lenguaje de programación diferente siempre que todos usen el mismo mecanismo de transmisión de mensajes.

Estos sistemas también se conocen como *sistemas distribuidos* ya que cada componente podría ejecutarse en su propio ambiente o sistema de computación.

El mecanismo de transmisión de mensajes generalmente consta de *canales* que permiten interconectar procesos.

Un canal se conecta a un proceso por medio de un *puerto (port)*.

Un canal puede ser

- **Asincrónico**: un proceso puede enviar mensajes por un canal y continuar con su ejecución. Generalmente un proceso que lee datos del canal deberá esperar (bloquear) hasta que el canal contenga al menos un mensaje.

  Este mecanismo generalmente requiere que el canal disponga de un *buffer*. En esta sección se describirá un mecanismo de pasaje de mensajes extendiendo el modelo declarativo con dos operaciones primitivas adicionales:

- **Sincrónico**: un proceso que desee enviar un mesaje deberá esperar a que otro proceso realice una lectura por el otro extremo. Este mecanismo de pasaje de mensajes actúa también como mecanismo de sincronización.

- `{NewPort S P}`: crea un nuevo puerto `P` con un stream (lista lazy) `S`.

- `{Send X P}`: envía un mensaje (valor ligado a) `X` por el puerto `P`.

La lectura de mensajes se hace por el mecanismo habitual de ligadura de variables.

```
local P S in
   {NewPort S P}
   thread for I in {Generator} do {Send I P} end end
   thread for M in S do {Browse M} end end
end
```

Fig. 10.7: Ejemplo de productor-consumidor con mensajes y puertos.

La figura 10.7 muestra un ejemplo de una implementación de un productor-consumidor con puertos.

10.8.1 Semántica de los puertos

Para poder implementar los puertos es necesario representarlos con la noción de estado, ya que un puerto requiere de un identificador. Se utilizará la memoria mutable μ.

- La sentencia `({NewPort <x> <y>},E)` realiza lo siguiente:
  1. crea un nuevo identificador (nombre) n para el puerto,
  2. `Bind(E(<y>),n)` y
  3. si la ligadura tuvo éxito, agregar el par `E(<y>):E(<x>)` a μ,
  4. en otro caso, generar una condición de error (excepción).
- La sentencia `({Send <x> <y>},E)` realiza los siguientes pasos:
  1. si la condición de activación de `E(<x>)` (`E(<x>)` es determinada) es falsa suspender la ejecución del thread, sino hacer los siguientes pasos:
  2. si `E(<x>)` no está ligado a un nombre de un puerto, generar un error, sino
  3. si μ contiene el par `E(<x>):z`, entonces hacer
     (a) crear una nueva variable `z'` en la memoria σ,
     (b) actualizar el par `E(<x>):z` en μ con `E(<x>):z'`,
     (c) ligar `z` con la nueva lista `E(<y>)|z'`,

10.8.2 Protocolos de comunicación entre procesos

Procesos (port objects) genralmente requieren comunicarse entre sí de alguna forma coordinada. Además deberán respetar formatos de mensajes para que cada uno de los intervinientes en el sistema puedan reconocerlos y actuar en consecuencia.

Un *protocolo de comunicación* define dicha coordinación y los formatos de los mensajes. La coordinación se expresa generalmente como secuencias de mensajes en los diferentes posibles escenarios del sistema. Generalmente esto se expresa por sistemas de transición de estados o diagramas de mensajes (o secuencias).

En los sistemas distribuidos como las redes, existen protocolos a diferentes niveles del sistema. Por ejemplo, la familia de protocolos TCP/IP, define 4 capas (niveles) de protocolos en un sistema de computación. Al más bajo nivel se encuentra el nivel de enlace (data link layer), el cual define la interface entre mensajes (datagramas) IP y el hardware encargado de su transmisión y recepción (ethernet, point-to-point, etc). El nivel de red (network layer) se encarga de definir el mecanismo de ruteo de envío y recepción de mensajes, definiendo identificadores (direcciones IP)para cada interface del sistema y otros mecanismos de control (como el protocolo ICMP). El nivel de transporte (transport layer) define dos protocolos de comunicación donde los extremos ya consideran el concepto de puerto, lo que permite representar conexiones entre dos procesos (locales o remotos). Uno de los protocolos es UDP (User Datagram Protocol), el cual define un modelo de envío y recepción sin conexión y no brinda ningún servicio de garantías de envío y recepción. El segundo es TCP (Transmission Control Protocol), el cual permite establecer una conexión (virtual) entre dos procesos y garantiza

secuencialidad en la transmisión de los mensajes (paquetes) y realiza retransmisiones en caso de fallas.

Al mas alto nivel se encuentran definidas interfaces (APIs) para las aplicaciones que definen formatos para algunos tipos de datos, como orden de bytes uniforme, y brindan operaciones de establecimiento y cierre de conexiones y transisión y recepción de mensajes. Una de las APIs mas populares en TCP/IP es *Berkeley sockets*.

TCP/IP es la familia de protocolos usado en Internet.

La Organización Internacional de Estándares (ISO) ha definido un modelo de referencia para los protocolos de comunicación que consiste en siete capas.

Otros protocolos muy populares son RPC (Remote Procedure Call) y RMI(Remote Method Invocation). RMI fue desarrollado por Sun en la década de 1980 y su objetivo es permitir desarrollar aplicaciones en las cuales el envío de mensajes tuviesen el mismo significado que las invocaciones o llamadas a procedimientos. Esto permite ejecutar servicios de procesos remotos como si se tratara de una invocación común a procedimientos.

En estos últimos protocolos, el protocolo encapsula las llamadas a procedimientos en mensajes que al ser recibidos por el destinatario, se decodifican y se invoca al procedimiento asociado.

RMI es la extensión del concepto con la noción de objeto.

Los protocolos llamados *web services* son protocolos al estilo RPC o RMI, generalmente implementados sobre el protocolo *HTTP*, el cual fue diseñado para la *World Web Wide* en Internet.

Entre los protocolos que se usan ampliamente para implementar servicios en internet, se pueden mencionar XML-RPC, SOAP, Java Servlets, Java Beans, Java RMI, etc.

Muchas empresas y comunidades de software libre ofrecen familias de bibliotecas (frameworks) para el desarrollo de aplicaciones distribuidas, como CORBA, OLE-COM, .Net, J2EE, etc.

10.9 Deadlock

Es posible escribir un programa en el cual varios componentes accediendo, en forma controlada por mecanismos de sincronización, a ciertos recursos podría alcanzar un estado en el cual dos o más threads o procesos podrían quedar esperándose mutuamente para acceder a los recursos. El uso de mecanismos de sincronización en el modelo concurrente con estado, trae problemas adicionales en el diseño e implementación de programas concurrentes.

Este estado que imposibilita el progreso de componentes de un sistema concurrente se conoce como *deadlock* (abrazo mortal).

La figura 10.8 muestra un esquema de un programa en el que podría ocurrir *deadlock*.

```
T1 = thread              T2 = thread
...                      ...
lock X in                lock Y in
   ... (1)                  ... (2)
   lock Y in                lock X in
         ...                      ...
   end                      end
   ...                      ...
end                      end
```

Fig. 10.8: Ejemplo de un programa con un potencial deadlock.

El thread `T1` puede estar en `(1)`, es decir que ya tiene acceso exclusivo a `X` y el thread `T2` puede estar en `(2)`, es decir que ya obtuvo acceso exclusivo a `Y`. A partir de allí, ambos threads esperarán por siempre.

Para que exista la posibilidad de que ocurra un deadlock, se deben cumplir las siguientes condiciones en forma simultánea:

- **exclusión mutua**: los threads o procesos acceden en forma excusiva a los recursos.
- **espera y retención**: los threads esperan por un recurso mantienen el uso exclusivo de otros que ya han adquirido con anterioridad.
- **no quita de recursos**: ningún thread o proceso puede quitar en forma compulsiva un recurso a otro thread.
- **espera circular**: debe darse la situación en que hay un ciclo en la relación de requerimientos de recursos que tratan de acceder los threads y los recursos que están reteniendo. Esta relación puede modelarse como un grafo.

El programador de aplicaciones, generalmente no tiene control sobre las tres primeras condiciones, ya que la primera es un requermiento para una solución correcta del problema y las dos siguientes son condiciones impuestas por el sistema.

Por lo tanto, generalmente, deberá escribir los programas de tal forma para evitar la espera circular.

Esto se puede lograr si cada thread accede a los recursos en un determinado orden, perdiendo tal vez, algún grado de concurrencia por el bloqueo temprano de recursos con respecto a su uso concreto.

10.10 Concurrencia en Java

El lenguaje de programación Java contiene facilidades de concurrencia ofreciendo threads y algunos mecanismos de sincronización como locks y monitores.

Se puede definir un thread como una clase derivada de `Thread` o implementando la interface `Runnable`. Ambas técnicas requieren que se redefina el método `run()`, el cual hará de sentencia inicial a ejecutarse cuando se inicie el nuevo thread.

Cada objeto que sea un thread puede ser (o comportarse como) un monitor.

Una sentencia (bloque) puede hacerse atómica usando la palabra reservada `synchronized`. Esta keyword también puede ser aplicada en la definición de un método, lo que hace que el método se ejecute dentro del monitor del objeto. Esto no ha sido una decisión muy acertada en el diseño del lenguaje, ya que código (ej: un método) no sincronizado puede acceder a los atributos del objeto sin control del monitor.

Un monitor en un objeto Java soporta las operaciones `wait()` y `notify()`[3].

Estas operaciones sólo son posibles dentro de un monitor, es decir en código protegido por `synchronize`.

Las operaciones mencionadas hacen lo siguiente:

- `wait()`:
  1. El thread corriente se suspende.
  2. El thread es colocado en la cola de threads suspendidos del objeto (wait set).
  3. Se libera el lock sobre el objeto.
- `notify()`:
  1. Un thread `T` se saca de la cola de threads suspendidos (si hay alguno).
  2. `T` intentará obtener el lock. En este momento, `T` puede competir por el lock con otros threads.
  3. `T` continúa (resume) su ejecución normalmente.

La operación `notifyAll()` hace lo mismo que `notify()`, sólo que para todos los threads en la *wait set* del objeto.

10.11 Concurrencia en Erlang

Erlang es un lenguaje funcional de alto nivel que fue desarrollado por Ericsson para aplicaciones en telecomunicaciones (telefonía) y redes de alta velocidad. Su implementación, Ericsson OPT (Open Telecom Platform), cuenta con una eficiente concurrencia, fiabilidad extrema (alto rendimiento en la tolerancía a fallas) y capacidad de sustitución de código mientras se ejecuta. Permite ocultar la representación interna de los datos y hace un manejo automático de la memoria. Una de las ideas principales del lenguaje, es aprovechar la independencia de las entidades del pasaje de mensajes para permitir que otros procesos continuen su ejecución incluso si alguno ha fallado, por lo que en un sistema disenado adecuadamente los procesos restantes pueden reorganizarse a si mismos y continuar brindando el servicio.

[3] También existe la operación `notifyAll()`, la cual desbloquea a todos los threads en ese monitor.

10.11.1 Características del Lenguaje

Variables

Las variables en Erlang son de una sola asignación, como en el lenguaje kernel. Una vez que se le asignó ún valor a una variable, permanece ligada a ese valor durante toda la ejecución del programa:

```
1>X = 10.
10
```

El scope de una variable es la unidad lexica en la cual fue definida. Si X es utilizada dentro de una simple función f, no puede ser accedida desde afuera de f. Si X ocurre en diferentes funciones, entonces todos los valores de X en las distintas funciones son diferentes.

Átomos

Son utilizados para representar diferentes valores constantes no numéricos. Son globales y pueden ser creados sin el uso de definiciones externas. Por ejemplo, para represenar días de una semana simplemente se pueden utilizar los átomos *lunes*, *martes*, etc. El valor de un átomo es exactamente el átomo.

Tuplas

Permiten agrupar un número fijo de items en una misma entidad. Por ejemplo, para representar un punto de un plano:

```
1>P = { 10, 45 }.
```

Cómo Erlang no tiene declaraciones de tipos, no es necesario espeficiar el tipo de cada campo de la tupla. Las tuplas pueden ser anidadas, es decir, se pueden crear tuplas donde alguno de los campos sea también una tupla. Para extraer los valores de una tupla debemos ligar el identificador de la tupla con una entidad que sea estructuralmente igual pero con variables no ligadas. Es decir, si queremos extraer los puntos de la coordenada P:

```
2>{ X, Y } = P.
{ 10, 45}
3>X.
10
4> Y.
45
```

donde X e Y deben ser variables no ligadas. Al final X tendrá el valor 10 e Y tendrá el valor 45.

Listas

Los elementos individuales de una lista pueden ser de cualquier tipo:

```
1>list = [1, 2*2, {10, 45}, atom].
[1, 2*2, {10, 45}, atom]
```

Se pueden agregar más elementos facilmente al principio de la lista:

```
2>list1 = [2, 3, |list].
[2, 3, 1, 4, {10, 45}, atom]
```

Notemos que debemos utilizar un nuevo identificador para la lista resultante, ya que list está ligado por la expresión previa. Para extraer los elementos de una lista, al igual que con las tuplas, podemos ligar una parte de la lista con una variable no ligada, es decir:

```
3>[X1 | list2] = list1.
[2, 3, 1, 4, {10, 45}, atom]
4>X1.
2
5>list2.
[3, 1, 4, {10, 45}, atom]
```

Strings

Estrictamente hablando, no hay strings en Erlang. Son representadas como listas de enteros. Por ejemplo:

```
1>Name = ``Hello''.
``Hello''
```

"Hello" es solo una representación de la lista de enteros que representa los caracteres individuales del string.

Módulos

Los módulos son la unidad básica de código en Erlang. Todas las funciones son almacenadas en módulos. Son definidos en archivos con la extensión *.erl*. Por ejemplo:

```
-module(geometry).
-export([area/1]).
area({regtangle, Width, Height}) -> Width * Height;
area({circle, Radio}) ->3.14159 * Radio * Radio.
```

Este módulo contiene una función *area*, la cuál consiste de dos cláusulas separadas por punto y coma. Cada una tiene una cabeza y un cuerpo. La cabeza consiste del nombre de una función, *area* en este caso, seguida de un patrón (entre paréntesis). El cuerpo consiste de una secuencia de expresiones que serán evaluadas si el patrón de la cabeza puede ser ligado con los argumentos de una invocación. Para ejecutar las funciones de un módulo, debe ser compilado:

```
1>c(geometry).
{ok, geometry}
2>geometry:area({rectangle, 10, 5}).
50
3>geometry:area({circle, 1.4}).
6.15752
```

Las funciones de un módulo pueden ser invocadas de la misma manera en el cuerpo de funciones de otros módulos.

Funciones

Cómo vimos, todas las funciones son definidas dentro de módulos. En Erlang, dos funciones con el mismo nombre y diferente aridad en el mismo módulo, representan funciones diferentes. Ejemplo:

```
sum(L) ->sum(L, 0).

sum([], N) ->N;
sum([H,T], N) ->sum(T, H+N).
```

La función *sum(L)* suma todos los elementos de la lista L utilizando una función diferente *sum* que toma dos argumentos y calcula la suma con recursividad a la cola. Es posible definir funciones anónimas:

```
1>Double = fun(X) -> 2*X end.
```

Y luego acceder a ellas mediante el identificador:

```
2>Double(2).
4
```

Cómo en los lenguajes funcionales, las funciones en Erlang pueden ser utilizadas cómo argumentos de otras funciones. Por ejemplo, utilizando el módulo standard *lists*:

```
1>L = [1,2,3,4].
[1,2,3,4]
2>lists:map(Double, L).
[2,4,6,8]
```

10.11.2 Modelo de Computación

El modelo de computación de Erlang tiene una elegante estructura dividida en capas. Básicamente consiste de entidades concurrentes llamadas "procesos", los cuáles tienen un identificador PID (constante única que identifica el proceso), un puerto y una casilla de mensajes. El lenguaje puede ser dividio en dos capas:

- Núcleo funcional: los procesos son programados en un lenguaje estrictamente funcional y dinámicamente tipado. El puerto de cada proceso es definido mediante una función recursiva. Un proceso que genera un nuevo proceso especifica que función debería ser ejecutada inicialmente dentro del nuevo proceso.

- Extensión de pasaje de mensajes: los procesos se comunican entre sí mediante el envío de mensajes de manera asíncrona. Los mensajes pueden contener cualquier valor, incluyendo funciones, y se van ubicando en la casilla de mensajes del proceso destino, el cuál utiliza "pattern mathing" para ir quitando los mensajes de la casilla sin perturbar los otros mensajes. Esto significa que los mensajes no son necesariamente tratados en el orden en que son enviados.

Una importante propiedad de los procesos Erlang es que son independientes por defecto. Es decir, lo que pase en un proceso no tiene efecto en ninguno de los otros procesos, al menos que esté programado explícitamente. Esto implica que los mensajes son siempre copiados cuando se envían entre procesos. Nunca hay referencias compartidas entre los procesos. Esta es también la profunda razón por la que la comunicación tiene como primitiva un envío asíncrono. La comunicación síncrona crea una dependencía, ya que el proceso que envía el mensaje debe esperar una respuesta del proceso destino. La independencia de procesos hace simple crear sistemas altamente fiables.

El modelo central puede ser extendido para distribución y tolerancía a fallas:

- Distribución transparente: los procesos puede estar en la misma máquina o en diferentes máquinas. En un programa, la comunicación entre procesos locales o remotos es escrita exactamente de la misma manera. El PID encapsula el destino y permite al sistema decidir cuando hacer una operación local o remota. Los procesos son estacionaros, esto significa que una vez que un proceso es creado en un nodo (entorno de una máquina) permanece allí durante todo su tiempo de vida. Enviar un mensaje a un proceso remoto requiere exactamene una operación de red, es decir, no hay nodos intermedios involucrados. Los programas son transparentes de acuerdo a la red, es decir, dan el mismo resultado sin importar en qué nodos los procesos están almacenados. El programador tiene un control completo del lugar de los procesos y puede optimizarlos de acuerdo a las características de la red.

- Detección de fallas: un proceso puede ser configurado para realizar alguna acción cuando otro proceso falla, esto es llamado "linkear" dos procesos. Una posible acción es que cuando el segundo proceso falla, un mensaje es enviado al primer proceso. La habilidad de la detección de fallas permite que muchos mecanismos de tolerancia a fallas puedan ser programados completamente en Erlang.

- Persistencia: el sistema Erlang incluye una base de datos, llamada Mnesia.

10.11.3 Programación

A continuación se muestra como escribir funciones y programas concurrentes con pasaje de mensajes en Erlang. Para más información sobre la programación en Erlang, se puede consultar el libro Concurrent Programming in Erlang.

Una función simple

El núcleo de Erlang es un lenguaje estrictamente fucional con tipado dinámico. Por ejemplo, una definiciín de la función factorial es la siguiente:

```
factorial(0) -> 1;
factorial(N) when N > 0 -> N*factorial(N-1);
```

Las funciones son definidas mediante cláusulas, donde cada una tiene una cabeza (con un patrón y una guarda opcional) y un cuerpo. Los patrones son chequeados en orden comenzando con la primer cláusula. Si un patrón coincide, las variables son ligadas y si no tiene guarda o bien la guarda opcional retorna true, el cuerpo es ejecutado. Si un patrón no coincide, entonces se intenta con la próxima cláusula, y así sucesivamente.

Pattern matching con tuplas

Erlang también permite pattern matching utilizando tuplas:

```
area({square, Side}) -> Side*Side;
area({rectanfle, X, Y}) -> X*Y;
area({circle, Radius}) -> 3.14159*Radius*Radius;
area({triangle, A, B, C}) ->
    S=(A+B+C)/2;
    math:sqrt(S*(S-A)*(S-B)*(S-C)).
```

La función calcula el área de una figura plana, la cuál está representada por una tupla con el nombre de la figura y su tamano.

Concurrencia y pasaje de mensajes

Algunos hechos básicos de cómo funcionan los programas Erlang:

- Los programas están hechos de muchos procesos, cada uno de los cuáles es independiente al resto y contiene una memoria privada, es decir, no existe el concepto de memoria compartida. La única manera de que los procesos interactuen entre sí es mediante el envío de mensajes.
- Estos mensajes pueden o no ser recibidos y comprendidos por el proceso destino. La única manera de saber si otro proceso ha recibido y comprendido el mensaje es esperar por una respuesta luego del envío.
- Pares de procesos pueden ser enlazados entre sí. Si uno de los dos procesos muere, el otro proceso en el par recibirá un mensaje conteniendo la razón por la que el primer proceso se detuvo.

En Erlang, cada proceso es creado con un puerto y una casilla de mensajes. Para la programación de procesos, solo se necesitan tres operaciones primitivas del lenguaje:

1. La operación **spawn** (escrita como **spawn(M,F,A)**) crea un nuevo proceso y retorna un valor (llamado "identificador del proceso") que puede ser utilizado para enviarle mensajes. Los argumentos de **spawn** dan la función que inicia el proceso, identificada por el módulo M, la función F, y la lista de argumentos A.

2. La operación **send** (escrita como **PID!Msg**) envía de forma asíncrona el mensaje Msg al proceso con identificador PID. Los mensajes son almacenados en la casilla de mensajes.

3. La operación **receive** es utilizada para remover mensajes de la casilla de mensajes mediante pattern matching.

A modo de ejemplo, podemos crear un proceso que contenga la función area en un servidor que puede ser llamado desde cualquier otro proceso:

```
-module(areaserver).
-export([start/0, loop/0]).

start() -> spawn(areaserver, loop, []).

loop() ->
    receive
        {From, Shape} ->
            From!area(Shape),
            loop()
        end.
```

El código define las operaciones **start** y **loop** dentro del módulo **areaserver**. Ambas son exportadas fuera del módulo. Es necesario definirlas dentro un módulo ya que la operación **spawn** (función que crea un nuevo proceso) requiere el nombre del módulo como argumento. La operación **loop** lee repetidamente un mensaje (una tupla de dos argumentos Form, Shape) y responde llamando la función **area** y enviando la respuesta al proceso **From**. Cómo mencionamos, podemos invocarlo desde otro proceso:

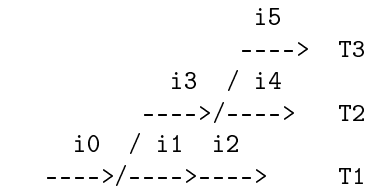
```
Pid=areaserver:start(),
Pid!{self(), {square, 3.4}},
receive
    Ans -> ...
end,
```

La operación **self** retorna el identificador del proceso del proceso actual, de modo que cuando el proceso anterior retorna la respuesta a **From** lo está haciendo al proceso que lo invocó.

Erlang hace la programación paralela fácil modelando el mundo como un conjunto de procesos paralelos que sólo pueden interactuar mediante el intercambio de mensajes. No existen métodos sincronizados entre los procesos ni posibilidad de memoria compartida.

10.12 Ejercicios

1. Dado el siguiente esquema de ejecución de threads:



   (a) Dar el orden causal como un conjunto por extensión (orden parcial de ejecución de las instrucciones).

   (b) Dar todos las posibles trazas de ejecución dadas por el interleaving.

2. Ejecutar manualmente, siguiendo la semántica del lenguaje concurrente declarativo del siguiente programa:

```
local X,Y in
   X = 1
   thread fun {$}
            Y = X
            X+1
          end
   end
   thread fun {$} Y end end
   {Browse X Y}
end
```

3. Implementar en Oz un programa con un thread que genere la sucesión de los números de Fibonacci (hasta un cierto N) y otro thread que los vaya mostrando por salida estándar (*Browse*). Este problema es una instancia del productor-consumidor donde la lista es un *stream*.

4. Dado el siguiente programa C

```
#include <stdio.h>
#include <unistd.h>/* por getpid() */
#define N (2000)
int main(void)
{
   FILE * f ;
   int i, value;
   for (i=0; i<N; i++) {
      f = fopen("seqno.txt","r+");
      fscanf(f,"%d", &value);
```

```
            value++;
            rewind(f);
            fprintf(f,"%6d\n", value);
            fflush(f);
            printf("Process id: %d, value: %d\n", getpid(), value);
            fflush(stdout); /* forzar la escritura al archivo */
            fclose(f);
//          sleep(2);
        }

    }
```

(a) Crear el archivo `seqno.txt` conteniendo una única línea con un cero.

(b) Correr dos (o más) instancias del programa en forma secuencial (ej: `prog ; prog`) y verificar que la salida es la esperada.

(c) Correr dos (o más) instancias del programa en forma concurrente (ej: `prog & prog`) y verificar el resultado final en el archivo.

(d) Existe una *race condition*? Justificar.

5. Modificar el programa anterior para que logre su objetivo cuando se ejecuta en forma concurrente.

   Ayuda: usar la función de biblioteca `lockf()`.

6. Dado el siguiente programa.

```
local Y1 Y2 C in
   C={NewCell nil}
   thread X=1 C:=X|@C Y1='listo' end
   thread C:=2|@C Y2='listo' end
   {Barrier Y1 Y2}
   {Browse @C}
end
```

   - Defina una traza de ejecución por la cual la lista correspondiente a la Celda C termina con un solo elemento.
   - Cuales son los posibles valores que muestra *{Browse @C}*?
   - Usando algún mecanismo de sincronización, sincronize la región crítica para que la lista siempre termine con 2 elementos.

7. Implementar en Java una clase `Semaphore` el cual se comporte como los semáforos definidos en este capítulo.

8. Considere la siguiente clase JAVA. Escriba un programa que genere dos o mas thread que actúen sobre un objeto de dicha clase.

```
public class Cuenta {
  private int saldo;
  public Cuenta(int saldoInicial) {
    saldo = saldoInicial;
  }
  public void deposita(int monto) {
    saldo += monto;
  }
  public void retira(int monto) {
    saldo -= monto;
  }
  public int getSaldo() { return saldo; }
}
```

(a) Verificar si existen condiciones de carrera.

(b) Que mecanismos ofrece Java para sincronizar los procesos?

(c) Utilice los semáforos definidos en el ejercicio 7 para garantizar la exclusión mutua a los métodos de la clase *Cuenta*.

9. El problema de los filósofos comensales es un problema clásico para el estudio de deadlocks. Existen n filósofos, los cuales, repetitivamente, piensan por un momento y luego toman dos tenedores que se encuentran al lado del plato y comen, para finalmente dejar los tenedores nuevamente en la mesa.

   Cada filósofo tiene un plato y hay n tenedores: uno entre cada plato.

   (a) Describir un estado en que puede ocurrir deadlock.

   (b) Escribir un programa que modele el problema libre de deadlock.

10. Implementar y ejecutar en Erlang el módulo *areaserver* definido en el cápitulo.

11. Analizar como soporta Erlang (en caso de que lo cubra) los siguientes conceptos:

    (a) Paradigma.

    (b) Sistema de Tipos.

    (c) Polimorfismo.

    (d) Modelo Concurrente. Comunicación sincrónica o asincrónica.

    (e) Técnicas de Alto Orden.

    (f) Ejemplos de aplicaciones implementadas parcial o totalmente en Erlang.

Ejercicios Adicionales

12. Implementar un programa concurrente en Oz con estado que contenga al menos dos threads realizando acciones sobre una instancia de la clase `Stack` definida en este capítulo. Utilizar locks para sincronizar las operaciones sobre el stack.

13. Implementar en Oz un procedimiento `{Wait X}` el cual debe hacer esperar (bloquear) al thread que lo ejecuta hasta que `X` se ligue.

14. Implementar en Oz un procedimiento `{Barrier [X1 X2 ... Xn]}` que se comporte como una barrera.

15. Dada la siguiente clase Java, implementar el problema del productor-consumidor (o bounded buffer). Definir las clases `Producer` y `Consumer` que representen threads produciendo y consumiendo valores a/desde una instancia de `Buffer`.

```
class Buffer
   int[] buf;
   int first, last, n, i;
   public void init(int size) {
       buf=new int[size];
       n=size; i=0; first=0; last=0;
   }
   public synchronized void put(int x) {
      while (i<n) wait();
      buf[last]=x;
      last=(last+1)%n;
      i=i+1;
      notifyAll();
   }
   public synchronized int get() {
      int x;
      while (i==0) wait();
      x=buf[first];
      first=(first+1)%n;
      i=i-1;
      notifyAll();
      return x;
   }
}
```

16. Qué sucedería si en la clase `Buffer` del ejercicio anterior se reemplaza cada `notifyAll()` por `notify()`?

17. Modificar el programa Java del productor-consumidor usando los semáforos definidos anteriormente.

18. Un *port object* (o agente) es un objeto que combina múltiples puertos con un único stream. Esto permite la comunicación entre procesos de la forma muchos a uno. Los *port objects* son generalmente usuados para modelar e implementar sistemas distribuidos. Un *port object* procesa los mensajes que arriban por sus puertos.

Implementar una aplicación del tipo *múltiples escritores, un lector*, el cual es una instancia del problema general de productores y consumidores, usando *port objects*.

Referencias

[1] Peter Van Roy and Seif Haridi. *Concepts, Techniques and Models of Computer Programming.* MIT Press, Cambridge, Massachusetts. ISBN: 0-262-22069-5. 2004.

[2] John C. Reynolds. *Theories of Programming Languages.* Cambridge University Press. 1998. ISBN: 0-521-59414-6.

[3] David Watt. *Programming Language Concepts and Paradigms.* Prentice Hall. 1990. ISBN: 0-13-728874-3.

[4] Terrence Pratt, Marvin Zelkowitz. *Programming Languages, Design and Implementation (Third Edition).* Prentice Hall. 1996. ISBN: 0-13-678012-1.

[5] Richard Bird, Philip Wadler. *Introductio to Functional Programming.* Prentice Hall. 1988. ISBN: 0-13-484189-1.

[6] Noam Chomsky. *Grammars.*

[7] IA-32 Intel®Architecture Software Developer's Manual. Vol 1, 2, 3.

[8] *Open C++.* URL: `http://www.openc++.org`.

[9] *VisualVM.* URL: `https://visualvm.github.io/index.html`.

[10] *GCViewer.* URL: `https://sourceforge.net/projects/gcviewer/`.

[11] *Jmap.* URL: `https://docs.oracle.com/en/java/javase/11/tools/jmap.html#GUID-D2340719-82BA-4077-`

[12] *Jstack.* URL: `https://docs.oracle.com/en/java/javase/11/tools/jstack.html#GUID-721096FC-237B-473`

[13] *Eclipse Memory Analyzer.* Java Heap Analyzer. URL: `https://www.eclipse.org/mat/`.

List of Figures