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.

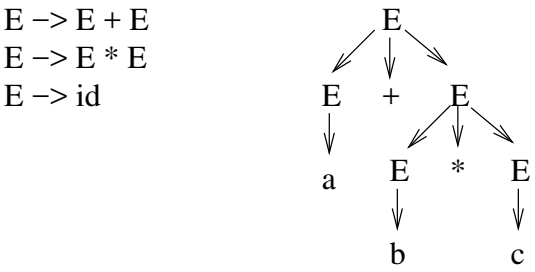

Fig. 1.1: Una CFG y un árbol de derivación.

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 ]
...
```

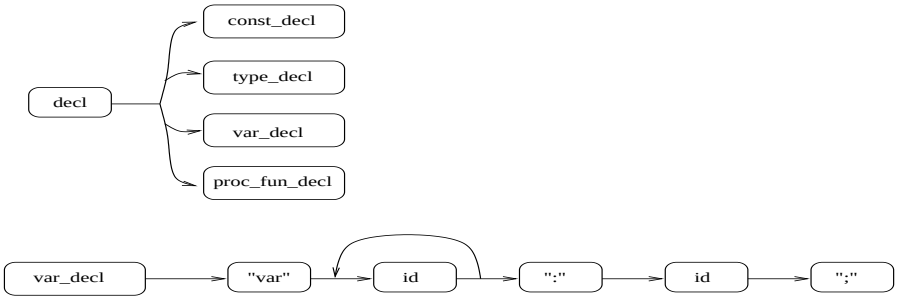

Fig. 1.2: Ejemplo de diagramas de sintaxis.

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.

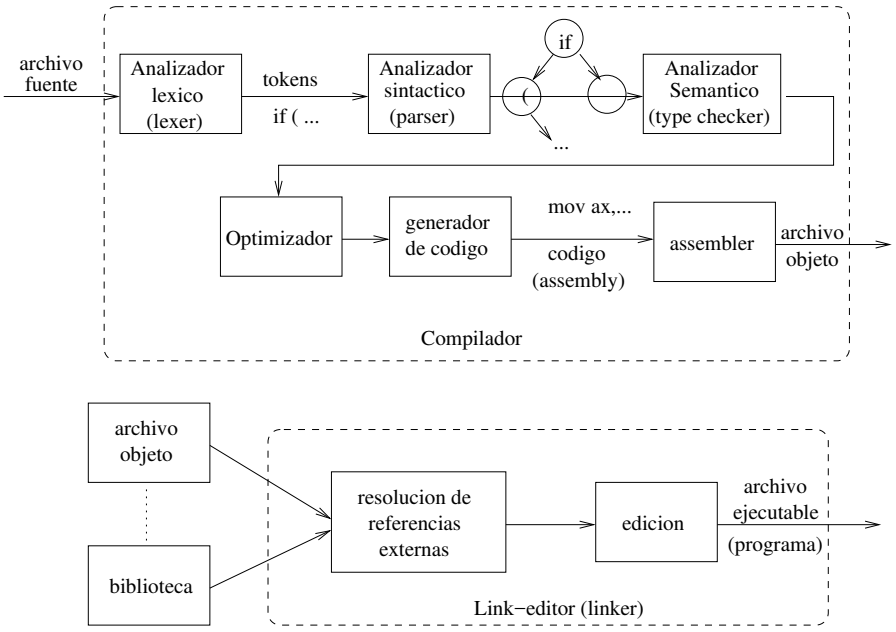

Fig. 1.3: Esquema de compilación de un programa.

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.

Ejercicios Adicionales

11. Implementar un programa que reconozca frases según la siguiente EBNF:

$E \rightarrow T['+'E]$
$T \rightarrow F['*'T]$
$F \rightarrow V \mid '('E')'$
$V \rightarrow ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')+$

Ayuda: Por cada regla de la gramática de la forma $X \rightarrow' a'Y'b'$ se puede definir un procedimiento con la forma:

```
Procedure X;
begin
  if current_token = 'a' then begin
```

```
        next_token;
        Y
    end
    else
        error;
    if current_token = 'b' then
        next_token
    else
        error
end;
```

donde *next_token* es un procedimiento que obtiene el próximo (símbolo) token de la entrada.

Para ésta gramática *next_token* debería reconocer (y obtener) valores numéricos y los símbolos + y * (e ignorar espacios, tabs y new-lines).

Generar patrones de código para reglas que contengan componentes opcionales (0 o una vez) y repeticiones (0 o mas y 1 o mas).

12. Extender el programa anterior para que evalúe la expresión.
Ayuda: utilizar una pila de operandos y una pila de operadores.

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*:

$$
\begin{array}{l}
\text{x} := 1; \\
\{x = 1\} \\
\text{y} := \text{f(x)} + \text{x}; \\
\{x = 1 \wedge y = 2 + 1\}
\end{array}
$$

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 $v \in T$.

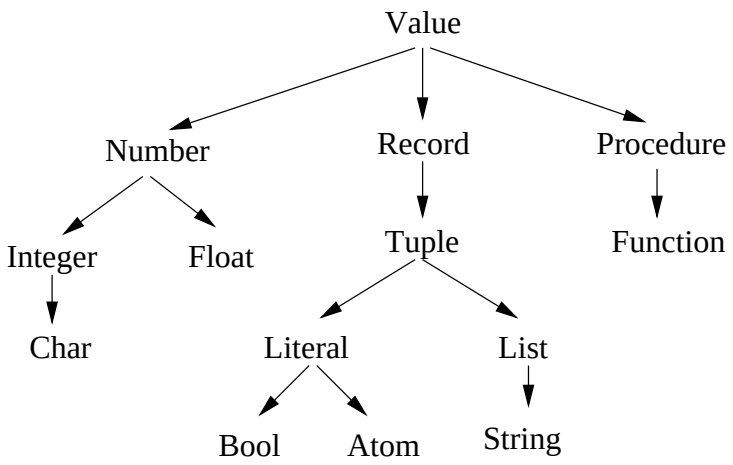

Fig. 2.1: Jerarquía de tipos básicos.

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*.

£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;
}
```

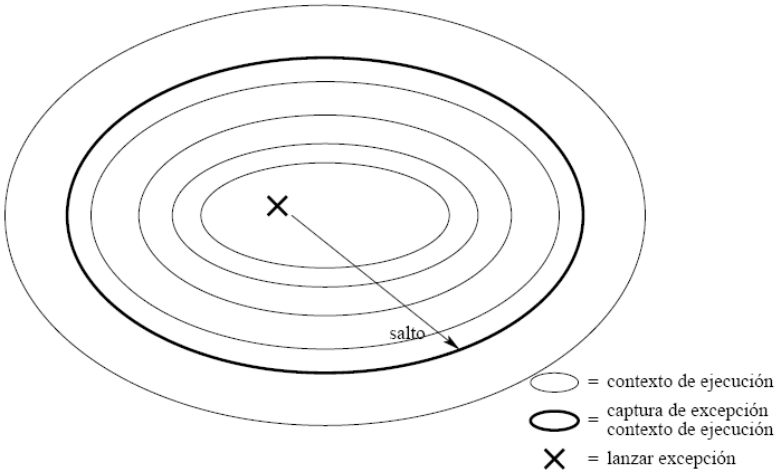

Fig. 2.3: Manejo de Excepciones.

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 Pascal 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 Pascal. Justifique su respuesta.

6. Dar un programa Pascal que muestre que no tiene un sistema de tipos seguro.

7. 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 que violan este principio.

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

9. 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.
```

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

Ejercicios Adicionales

11. Determine qué tipo de alcance tiene el lenguaje LOGO (usado ampliamente en la enseñanza de la programación).

12. De un ejemplo de uso de excepciones en Java.

13. Determinar (mediante un experimento) el número de bits de los enteros en el compilador Pascal que utilice.

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.

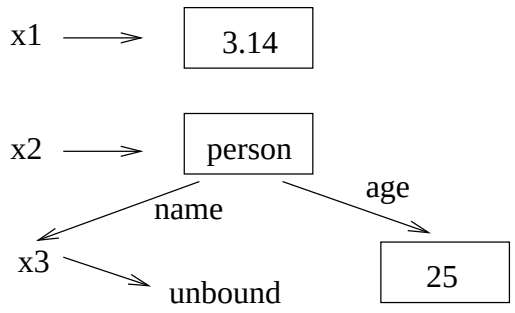

Fig. 3.1: Ejemplo de la memoria conteniendo variables y valores.

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.

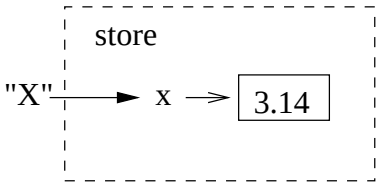

Fig. 3.3: Identificadores y variables.

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* $>$|$<$ *record* $>$|$<$ *procedure* $>$	valores
$<$ *number* $>$	::=	$<$ *integer* $>$|$<$ *float* $>$	números
$<$ *record* $>$	::=	$<$ *pattern* $>$	records
$<$ *pattern* $>$	::=	$<$ *literal* $>$	patrones
	|	$<$ *literal* $>$ $(< \mathit{field} >_1 : x_1 \ldots < \mathit{field} >_n : x_n)$	
$<$ *literal* $>$	::=	$<$ *atom* $>$|$<$ *bool* $>$	
$<$ *field* $>$	::=	$<$ *literal* $>$|$<$ *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.

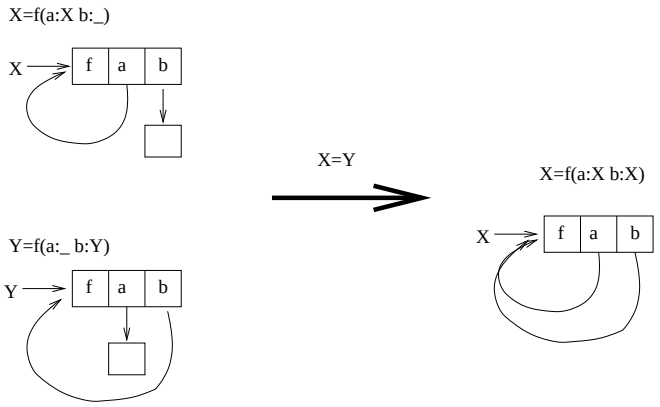

Fig. 3.6: Ejemplo de estructuras cíclicas.

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 Si }
    if { Es_Final Si } then Si
    else Si+1 in
            Si+1 = { Transformar Si }
            { Iterar Si+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*:
$CotaInferior10 = \{Max10\}$

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]
   ```

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`