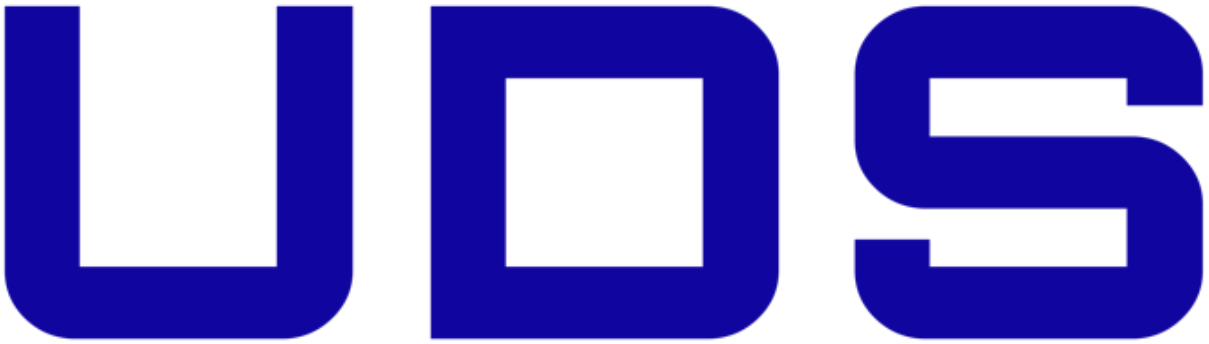


ANTOLOGIA

PROGRAMACIÓN LÓGICA

INGENIERÍA EN SISTEMAS COMPUTACIONALES

OCTAVO CUATRIMESTRE

Marco Estratégico de Referencia

ANTECEDENTES HISTORICOS

Nuestra Universidad tiene sus antecedentes de formación en el año de 1979 con el inicio de actividades de la normal de educadoras "Edgar Robledo Santiago", que en su momento marcó un nuevo rumbo para la educación de Comitán y del estado de Chiapas. Nuestra escuela fue fundada por el Profesor de Primaria Manuel Albores Salazar con la idea de traer Educación a Comitán, ya que esto representaba una forma de apoyar a muchas familias de la región para que siguieran estudiando.

En el año 1984 inicia actividades el CBTiS Moctezuma Ilhuicamina, que fue el primer bachillerato tecnológico particular del estado de Chiapas, manteniendo con esto la visión en grande de traer Educación a nuestro municipio, esta institución fue creada para que la gente que trabajaba por la mañana tuviera la opción de estudiar por las tarde.

La Maestra Martha Ruth Alcázar Mellanes es la madre de los tres integrantes de la familia Albores Alcázar que se fueron integrando poco a poco a la escuela formada por su padre, el Profesor Manuel Albores Salazar; Víctor Manuel Albores Alcázar en septiembre de 1996 como chofer de transporte escolar, Karla Fabiola Albores Alcázar se integró como Profesora en 1998, Martha Patricia Albores Alcázar en el departamento de finanzas en 1999.

En el año 2002, Víctor Manuel Albores Alcázar formó el Grupo Educativo Albores Alcázar S.C. para darle un nuevo rumbo y sentido empresarial al negocio familiar y en el año 2004 funda la Universidad Del Sureste.

La formación de nuestra Universidad se da principalmente porque en Comitán y en toda la región no existía una verdadera oferta Educativa, por lo que se veía urgente la creación de una institución de Educación superior, pero que estuviera a la altura de las exigencias de los jóvenes que tenían intención de seguir estudiando o de los profesionistas para seguir preparándose a través de estudios de posgrado.

Nuestra Universidad inició sus actividades el 18 de agosto del 2004 en las instalaciones de la 4ª avenida oriente sur no. 24, con la licenciatura en Puericultura, contando con dos grupos de cuarenta

alumnos cada uno. En el año 2005 nos trasladamos a nuestras propias instalaciones en la carretera Comitán – Tzimol km. 57 donde actualmente se encuentra el campus Comitán y el Corporativo UDS, este último, es el encargado de estandarizar y controlar todos los procesos operativos y Educativos de los diferentes Campus, Sedes y Centros de Enlace Educativo, así como de crear los diferentes planes estratégicos de expansión de la marca a nivel nacional e internacional.

Nuestra Universidad inició sus actividades el 18 de agosto del 2004 en las instalaciones de la 4ª avenida oriente sur no. 24, con la licenciatura en Puericultura, contando con dos grupos de cuarenta alumnos cada uno. En el año 2005 nos trasladamos a nuestras propias instalaciones en la carretera Comitán – Tzimol km. 57 donde actualmente se encuentra el campus Comitán y el corporativo UDS, este último, es el encargado de estandarizar y controlar todos los procesos operativos y educativos de los diferentes campus, así como de crear los diferentes planes estratégicos de expansión de la marca.

MISIÓN

Satisfacer la necesidad de Educación que promueva el espíritu emprendedor, aplicando altos estándares de calidad Académica, que propicien el desarrollo de nuestros alumnos, Profesores, colaboradores y la sociedad, a través de la incorporación de tecnologías en el proceso de enseñanza-aprendizaje.

VISIÓN

Ser la mejor oferta académica en cada región de influencia, y a través de nuestra Plataforma Virtual tener una cobertura Global, con un crecimiento sostenible y las ofertas académicas innovadoras con pertinencia para la sociedad.

VALORES

- Disciplina
- Honestidad
- Equidad
- Libertad

ESCUDO

El escudo de la UDS, está constituido por tres líneas curvas que nacen de izquierda a derecha formando los escalones al éxito. En la parte superior está situado un cuadro motivo de la abstracción de la forma de un libro abierto.

ESLOGAN

"Mi Universidad"

ALBORES

Es nuestra mascota, un Jaguar. Su piel es negra y se distingue por ser líder, trabaja en equipo y obtiene lo que desea. El ímpetu, extremo valor y fortaleza son los rasgos que distinguen.

PROGRAMACIÓN LÓGICA

Objetivo de la materia:

Conocer los principios lógicos y funcionales de la programación para identificarlos y aplicarlos en la resolución de problemas a través del diseño de agentes inteligentes.

UNIDAD 1 CONCEPTOS FUNDAMENTALES

1.1 Estilos de programación.

1.2 Evaluación de expresiones.

1.3 Definición de funciones.

1.4 Disciplina de tipos.

1.5. Tipos de datos.

1.6 Tipos de datos simples

1.7 Los tipos de datos en un sentido amplio

1.8 La utilización de los tipos de datos

1.9 Ejemplos de tipos de datos

UNIDAD 2 PROGRAMACIÓN FUNCIONAL

2.1 Orígenes

2.2 Qué es la programación funcional

2.3 El tipo de datos.

2.4 Funciones.

2.4.1 Funciones recursivas

2.4.2 Parámetros por valor y por referencia

2.4.3 Parámetros por referencia

2.5 Variables locales y globales

2.6 Intervalos.

2.7 Clasificación

2.8 Operadores.

2.9 Operadores java relacionales

2.10 Aplicaciones de las listas

2.11 Árboles.

UNIDAD 3 EVALUACIÓN PEREZOSA

3.1 Historia

3.2 Aplicaciones

3.3 Estructuras de Control

3.4 Trabajar con estructuras de datos infinitas

3.5 Evitación de condiciones de error

3.6 Otros usos

3.7 Implementación

3.8 Pereza y afán Controlar el entusiasmo en lenguajes perezosos

3.9 Simular la pereza en idiomas ávidos

3.10 La estrategia de evaluación perezosa.

3.11 Técnicas de programación funcional perezosa.

UNIDAD 4 FUNDAMENTOS DE LA PROGRAMACIÓN LÓGICA

4.1.- Repaso de la lógica de primer orden.

4.2.- Unificación y resolución.

4.3.- Cláusulas de Horn. Resolución SLD.

4.4.- Programación lógica con cláusulas de Horn.

4.5.- Semántica de los programas lógicos.

4.6.- Representación causada del conocimiento.

4.7.- Consulta de una base de cláusulas

4.8.- Espacios de búsqueda

4.9.- Programación lógica con números, listas y árboles.

4.10.- Control de búsqueda en programas lógicos.

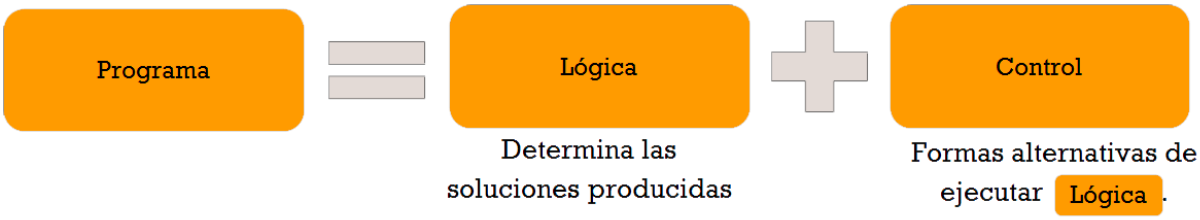
4.11.- Manipulación de términos predicados meta lógicos.

Índice

UNIDAD I

CONCEPTOS FUNDAMENTALES

La programación lógica, junto con la funcional, forma parte de lo que se conoce como programación declarativa. En los lenguajes tradicionales, la programación consiste en indicar cómo resolver un problema mediante sentencias; en la programación lógica, se trabaja de una forma descriptiva, estableciendo relaciones entre entidades, indicando no cómo, sino qué hacer. La ecuación de Robert Kowalski (Universidad de Edimburgo) establece la idea esencial de la programación lógica: algoritmos = lógica + control. Es decir, un algoritmo se construye especificando conocimiento en un lenguaje formal (lógica de primer orden), y el problema se resuelve mediante un mecanismo de inferencia (control) que actúa sobre aquél.



- Se puede ver como una deducción controlada.
- Lógica (programador): hechos y reglas para representar conocimiento
- Control (interprete): deducción lógica para dar respuestas (soluciones)

1.1 ESTILOS DE PROGRAMACIÓN

Estilo de programación (también llamado estándares de código o convención de código) es un término que describe convenciones para escribir código fuente en ciertos lenguajes de programación. El estilo de programación es frecuentemente dependiente del lenguaje de programación que se haya elegido para escribir. Por ejemplo, el estilo del lenguaje de programación C variará con respecto al del lenguaje BASIC. El buen estilo, al tratarse de algo subjetivo, es difícil de categorizar concretamente; con todo, existen un número de características generales. Con el advenimiento de software que da formato al código fuente de forma automática, el foco en cómo éste debe de ser escrito debe evolucionar de forma más amplia a la elección de nombres, lógica y otras técnicas. Desde un punto de vista práctico, el uso de un computador para dar formato al código fuente ahorra tiempo, y es posible forzar el uso de estándares a nivel de una compañía completa sin debates religiosos.

FORTRAN

Hay muchos estilos diferentes de programación, pero se intentará dar algunas guías generales que son de aceptación general.

Portabilidad

Para asegurar la portabilidad del código, se recomienda usar sólo el estándar de Fortran 77. La única excepción que se ha hecho en este manual es usar letras minúsculas.

Estructura del Programa

La estructura total del programa deberá ser modular. Cada subprograma deberá resolver una tarea bien definida. Mucha gente prefiere escribir cada subprograma en un archivo por separado.

Comentarios

Se repite lo que se había indicado previamente: Escriba código legible, pero también agregue comentarios al código fuente para explicar lo que se está haciendo. Es especialmente importante tener una buena cabecera para cada subprograma que explique cada argumento de entrada/salida y que hace el subprograma.

Sangrado

Se debe siempre usar el sangrado apropiado para bloques de ciclos y sentencias if como se mostró en el tutorial.

Variables

Declarar siempre todas las variables. No se recomienda la declaración implícita. Intentar compactar a 6 caracteres como máximo para nombres de variables, o asegurarse que los primeros 6 caracteres son únicos.

Subprogramas

Nunca se debe permitir que las funciones tengan "efectos laterales", por ejemplo no se deben cambiar los valores de los parámetros de entrada. Usar subrutinas en tales casos.
En las declaraciones separar los parámetros, bloques comunes y variables locales.
Minimizar el uso de bloques comunes.

Goto

Minimizar el uso de la sentencia goto. Desafortunadamente se requiere usar goto en algunos ciclos, ya que el ciclo while no es estándar en Fortran.

Arreglos

En muchos casos es mejor declarar todos los arreglos grandes en el programa principal y entonces pasarlos como argumentos a las distintas subrutinas. De esta forma toda la asignación de espacio es hecha en un sólo lugar. Recordar que se deben pasar también las dimensiones principales. Evitar el innecesario "redimensionamiento de matrices".

Asuntos de Eficiencia

Cuando se tenga un ciclo doble que esta accediendo a un arreglo bidimensional, es usualmente mejor tener el primer índice (renglón) dentro del arreglo más interno. Lo anterior por el esquema de almacenamiento en Fortran.

Cuando se tengan sentencias if-then-elseif con condiciones múltiples, intentar colocar primero aquellas condiciones que vayan a ser las más frecuentes que ocurran.

C

No existen un conjunto de reglas fijas para programar con legibilidad, ya que cada programador tiene su modo y sus manías y le gusta escribir de una forma determinada. Lo que sí existen son un conjunto de reglas generales, que, aplicándolas, en mayor o menor medida, se consiguen programas bastante legibles. Aquí intentaremos resumir estas reglas.

Identificadores significativos

Un identificador es un nombre asociado a un objeto de programa, que puede ser una variable, función, constante, tipo de datos... El nombre de cada identificador debe identificar lo más claramente posible al objeto que identifica (valga la redundancia). Normalmente los identificadores deben empezar por una letra, no pueden contener espacios (ni símbolos raros) y suelen tener una longitud máxima que puede variar, pero que no debería superar los 10-20 caracteres para evitar lecturas muy pesadas.

Un identificador debe indicar lo más breve y claramente posible el objeto al que referencia. Por ejemplo, si una variable contiene la nota de un alumno de informática, la variable se puede llamar nota informática. Observe que no ponemos los acentos, los cuales pueden dar problemas de compatibilidad en algunos sistemas. El carácter '_' es muy usado para separar palabras en los identificadores.

Constantes simbólicas

En un programa es muy normal usar constantes (numéricas, cadenas...). Si estas constantes las usamos directamente en el programa, el programa funcionará, pero es más recomendable usar constantes simbólicas, de forma que las definimos al principio del programa y luego las usamos cuando haga falta.

Comentarios, comentarios...

El uso de comentarios en un programa escrito en un lenguaje de alto nivel es una de las ventajas más importantes con respecto a los lenguajes máquina, además de otras más obvias. Los comentarios sirven para aumentar la claridad de un programa, ayudan para la documentación y bien utilizados nos pueden ahorrar mucho tiempo.

No se debe abusar de comentarista, ya que esto puede causar una larga y tediosa lectura del programa, pero en caso de duda es mejor poner comentarios de más.

Estructura del programa

Un programa debe ser claro, estar bien organizado y que sea fácil de leer y entender. Casi todos los lenguajes de programación son de formato libre, de manera que los espacios no importan, y podemos organizar el código del programa como más nos interese.

Para aumentar la claridad no se deben escribir líneas muy largas que se salgan de la pantalla y funciones con muchas líneas de código (especialmente la función principal). Una función demasiado grande demuestra, en general, una programación descuidada y un análisis del

problema poco estudiado. Se deberá, en tal caso, dividir el bloque en varias llamadas a otras funciones más simples, para que su lectura sea más agradable. En general se debe modularizar siempre que se pueda, de forma que el programa principal llame a las funciones más generales, y estas vayan llamando a otras, hasta llegar a las funciones primitivas más simples. Esto sigue el principio de divide y vencerás, mediante el cual es más fácil solucionar un problema dividiéndolo en subproblemas (funciones) más simples.

Indentación o sangrado

La indentación o sangrado consiste en marginar hacia la derecha todas las sentencias de una misma función o bloque, de forma que se vea rápidamente cuales pertenecen al bloque y cuáles no. Algunos estudios indican que el indentado debe hacerse con 2, 3 ó 4 espacios. Usar más espacios no aumenta la claridad y puede originar que las líneas se salgan de la pantalla, complicando su lectura.

La indentación es muy importante para que el lector/programador no pierda la estructura del programa debido a los posibles anidamientos.

Presentación

Al hacer un programa debemos tener en cuenta quien o quienes van a usarlo o pueden llegar a usarlo, de forma que el intercambio de información entre dichos usuarios y el programa sea de la forma más cómoda, clara y eficaz posible.

En general, se debe suponer que el usuario no es un experto en la materia, por lo que se debe implementar un interfaz que sea fácil de usar y de aprender, intuitivo y que permita efectuar la ejecución de la forma más rápida posible.

JAVA

Para lograr la legibilidad de un programa es importante considerar aspectos tales como el nombre de los identificadores, escribir el código con cierta alineación y líneas en blanco en lugares apropiados, así como realizar una buena documentación.

Identificadores

Los identificadores deben ser elegidos de tal manera que el solo nombre describa el uso que se dará dentro del programa, por tanto, no es recomendable usar identificadores de una letra, excepto en el for, ni abreviaturas raras o ambiguas.

Además de eso es recomendable que se escriban:
Empezando con mayúscula si se trata del nombre de una clase o interfaz, y empezando cada palabra en identificador con mayúscula.

CírculoColoreado

Sólo con mayúsculas si es el nombre de una constante. **DIAS_HABILES**
Empezando con minúscula si es el nombre de cualquier otro identificador. De preferencia el nombre de cualquier método debe ser un verbo en infinitivo y el de todo atributo un sustantivo. primerJugador, asignarSueldo().

Archivos fuente

Cada programa en Java es una colección de uno o más archivos. El programa ejecutable se obtiene compilando estos archivos. En cada archivo especifica su contenido como sigue:

- Los paquetes (instrucción package).
- Los archivos de biblioteca (Instrucciones import).

- Un comentario explicando el objetivo del archivo.
- Las clases que defines en ese archivo.

Clases

Cada clase debe ir precedida por un comentario que explique su objetivo. Es recomendable especificar sus elementos como sigue:

- Estructura de los objetos. Primero las variables y luego las constantes.
- Elementos estáticos.
- Constructores.
- Métodos públicos y privados.
- Métodos estáticos.
- Clases internas.
- Deja una línea en blanco después de cada método.
- Todos los elementos deben estar precedidos por public, private o protected. Las variables deben ser privadas. Los métodos y las constantes pueden ser privados o públicos, según se requiera.

Métodos

Todo método excepto main debe empezar con un comentario en formato javadoc.

El cuerpo de un método no debe exceder 30 líneas de código. Esto te obligará a dividir un método complejo en varios más sencillos.

Variables y Constantes

NO definas más de una variable por línea:

```
int horas = 0, minutos = 0;  //Mal
```

es mejor:

```
int horas = 0,
    minutos = 0;
```

Alineación y espacios en blanco

La alineación de instrucciones, se puede hacer de manera automática si se emplea el editor emacs (es recomendable modificar los tabuladores para que dejen sólo tres espacios en blanco).

Usa líneas en blanco para separar partes de un método que son lógicamente distintas.

1.2 EVALUACIÓN DE EXPRESIONES

¿Que son las expresiones?

- Son el método fundamental que tiene el programador de expresar computaciones.
- Las expresiones están compuestas de operadores, operandos, paréntesis y llamadas a funciones. Los operadores pueden ser:
- Unarios: Cuando tan solo tienen un operando. Son operadores prefijos.
- Binarios: 2 Operandos. Son operadores infijos.
- Ternarios: 3 operandos.

Orden de la evaluación de los operadores.

- El orden en que se evalúan los operandos viene dado por unas reglas:
- Reglas de procedencia
- Reglas de asociatividad
- Uso de paréntesis

Evaluación de expresiones

Toda expresión regresa un valor. Si hay más de un operador, se evalúan primero operadores mayor precedencia, en caso de empate, se aplica regla asociatividad

Para evaluar una expresión no hay que hacer nada del otro mundo, pues es bien sencillo. Sólo hay que saber sumar, restar, si un número es mayor que otro

Hay tres reglas de prioridad a seguir para evaluar una expresión:

- Primero, los paréntesis (si tiene)
- Después, seguir el orden de prioridad de operadores
- Por último, si aparecen dos o más operadores iguales, se evalúan de izquierda a derecha.

Las expresiones son secuencias de constantes y/o variables separadas por operadores válidos.

Se puede construir una expresión válida por medio de:

1. Una sola constante o variable, la cual puede estar precedida por un signo + ó – .
2. Una secuencia de términos (constantes, variables, funciones) separados por operadores.

Además, debe considerarse que:

Toda variable utilizada en una expresión debe tener un valor almacenado para que la expresión, al ser evaluada, dé como resultado un valor.

Cualquier constante o variable puede ser reemplazada por una llamada a una función.

Como en las expresiones matemáticas, una expresión en Pascal se evalúa de acuerdo a la precedencia de operadores

Jerarquía de operadores

El orden general de evaluación de los operadores de una expresión va de izquierda a derecha, con la excepción de las asignaciones que lo hacen de derecha a izquierda.

Podemos seguir las siguientes tres reglas de evaluación de expresiones:

- (Regla 1) En todas las expresiones se evalúan primero las expresiones de los paréntesis más anidados (interiores unos a otros); y éstos modifican la prioridad según la cantidad de éstos, los cuales tienen que estar balanceados (el mismo número de paréntesis que abren debe ser igual al número de los paréntesis que cierran).
- (Regla 2) Todas las expresiones se evalúan tomando en cuenta la jerarquía de los operadores.
- (Regla 3) Todas las expresiones se evalúan de izquierda a derecha.

Programación Funcional: es un estilo de programación que enfatiza la evaluación de expresiones, en lugar de la ejecución de comandos. Las expresiones en estos lenguajes se forman utilizando funciones para combinar valores básicos.
Lenguaje Funcional: es un lenguaje que soporta e incentiva la programación en un estilo funcional.

Precedencia de Operadores en Java

operadores sufijo	[] . (params) expr++ expr--
operadores unarios	++expr --expr +expr -expr ~ !
creación o tipo	new (type)expr
multiplicadores	* / %

suma/resta	+ -
desplazamiento	<< >> >>>
relacionales	< > <= >= instanceof
igualdad	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
AND lógico	&&
OR lógico	||
condicional	? :
asignación	= += -= *= /= %= ^= &= |= <<= >>= >>>=

Sentencias de Control de Flujo en Java

Las sentencias de control de flujo determinan el orden en que se ejecutarán las otras sentencias dentro del programa. El lenguaje Java soporta varias sentencias de control de flujo, incluyendo.

Sentencias	**palabras clave**
toma de decisiones	if-else, switch-case
bucles	for, while, do-while

excepciones	try-catch-finally, throw
miscelaneas	break, continue, label:, return

Tipos de expresiones:

Dentro de las expresiones distinguimos dos clases según el tipo de datos que devuelven al evaluarlas.

Aritméticas: las que devuelven un valor numérico

Lógicas: las que devuelven true o false

1.3 DEFINICIÓN DE FUNCIONES

Cuando escribas un nuevo programa o biblioteca, sigue un estilo consistente de ubicación de llaves y de indentación. Si no tienes ninguna preferencia personal de estilo, recomendamos el estilo de programación del núcleo de Linux o el estilo de programación de GNU. Lee el nodo de info (Standards)Writing C en la documentación de GNU. Luego, obtén el código fuente de Linux y lee el archivo linux/Documentation/CodingStyle, e ignora los chistes de Linus. Estos dos documentos te darán una buena idea de nuestras recomendaciones para el código de GNOME.

Estilo de indentación Para el código del núcleo de GNOME preferimos el estilo de indentación del núcleo de Linux. Usa tabuladores de 8 espacios para la indentación. Usar tabuladores de 8 espacios para indentación proporciona un número de beneficios. Permite que el código sea más fácil de leer, ya que la indentación se marca claramente. También ayuda a mantener el código ordenado forzando a dividir funciones en trozos más modulares y bien definidos — si la indentación va más allá del margen derecho, significa que la función está mal diseñada y que debiera dividirse para hacerla más modular o bien, repensarla.

Los tabuladores de 8 espacios para indentación también ayudan al diseño de funciones que encajen bien en la pantalla, lo cual significa que las personas puedan entender el código sin tener que desplazarse atrás y adelante para entenderlo.

Definición de Funciones

Una función es una aplicación que toma uno o más argumentos y devuelve un valor.

Es una correspondencia en la que cada elemento del dominio está relacionado con un único elemento de la imagen.

• Ejemplo de definición de función en Haskell:

- **doble x = x * x**
- **Notación matemática: f(a, b) + cd**
- **Notación Haskell: f a b + c * d**
- **Los paréntesis se utilizan para agrupar expresiones:**
- **Notación matemática: f(x, g (y))**
- **Notación Haskell: f x (g y)**

Las definiciones se incluyen en ficheros de texto. Se acostumbra a identificar dichos ficheros mediante el sufijo. hs.

Los nombres de funciones tienen que empezar por una letra en minúscula.

En Haskell la disposición del texto del programa (el sangrado) delimita las definiciones mediante la siguiente regla:

- Una definición acaba con el primer trozo de código con un margen izquierdo menor o igual que el del comienzo de la definición actual.
- Un comentario simple comienza con -- y se extiende hasta el final de la línea.

- Un comentario anidado comienza con {- y termina en -}
- **máximo x y z=max x(max y z)** {- ejemplo de una definición y un comentario }

La manera más fácil de definir funciones es por combinación de otras funciones:

```
fac n = product [1..n]

impar x = not (even x)

cuadrado x = x*x

suma_de_cuadrados lista = sum (map cuadrado lista)
```

Las funciones pueden tener más de un parámetro:

```
comb n k = fac n / (fac k * fac (n-k))

formulaABC a b c = [ (-b+sqrt(b*b-4.0*a*c)) / (2.0*a)

 , (-b-sqrt(b*b-4.0*a*c)) / (2.0*a)]
```

Las funciones sin parámetros se llaman normalmente constantes:

```
pi = 3.1415926535

e = exp 1.0
```

Toda definición de función tiene por tanto la siguiente forma:

• El nombre de la función

• Los nombres de los parámetros (si existen)

• El símbolo =

• una expresión, que puede contener los parámetros, las funciones estándar y otras funciones definidas.

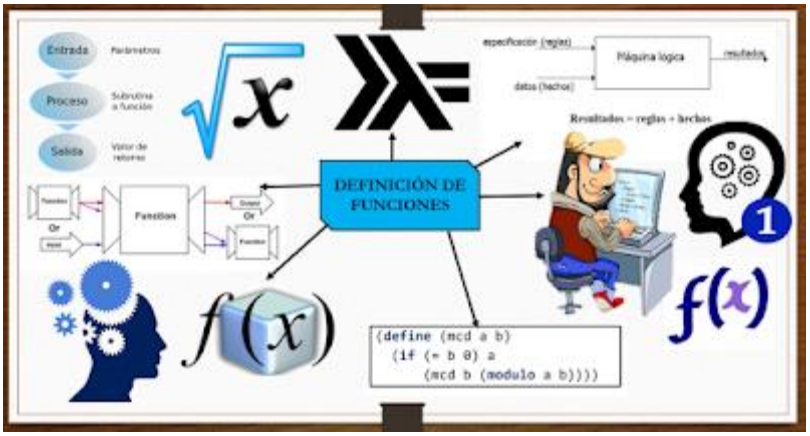
Una función que tiene un valor booleano como resultado, tiene a la derecha del símbolo = una expresión con un valor booleano:

negativo x = x < 0

positivo x = x > 0

esCero x = x == 0

Note la diferencia en el anterior ejemplo entre **=** y **==**. El símbolo = separa la parte izquierda de la parte derecha de la definición. El símbolo == es un operador, como **<** y **>**.



1.4 DISCIPLINA DE TIPOS

"Los tipos se infieren, es decir se comprueban, de forma estática, en tiempo de compilación."

En los lenguajes de programación con disciplina de tipos, cada tipo representa una colección de valores o datos similares. El conocer los tipos de las funciones ayuda a documentar los programas y evitar errores en tiempo de ejecución.

Un lenguaje tiene disciplina de tipos si los errores de tipos se detectan siempre, es necesario determinar los tipos de todos los operandos, ya sea en tiempo de compilación o de ejecución.

Pascal

Cercano a tener disciplina de tipos pero no realiza comprobación de tipos en los registros variantes (incluso puede omitirse la etiqueta discriminatoria en dichos registros).

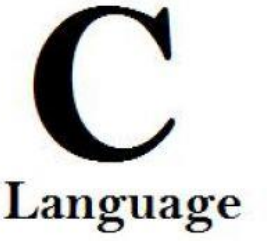
Ada

• Resuelve el problema de los registros variantes realizando comprobación dinámica de tipos (sólo en este caso).

• Tiene una función de biblioteca que permite extraer un valor de una variable de cualquier tipo (como una cadena de bits) y usarlo como un tipo diferente (no es una conversión de tipos) se trata de una suspensión temporal de la comprobación de tipos.

C

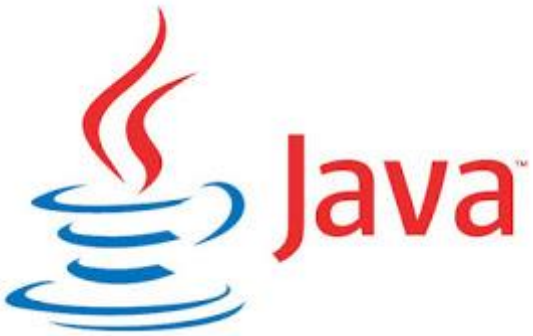
No tiene disciplina de tipos por:



- No se realiza comprobación de tipos sobre las uniones
- Permite funciones con parámetros sobre los que no se realiza comprobación de tipos.

Java

• Tiene disciplina de tipos (no hay uniones)



ML y Haskell

• Poseen disciplina de tipos

• Los tipos de los parámetros de las funciones (y de estas mismas) se conocen en tiempo de compilación (ya sea por declaración del usuario o por inferencia de tipos).

Haskell y otros lenguajes funcionales utilizan el sistema de tipos de Milner, que tiene dos características fundamentales:

- Disciplina estática de tipos: Los programas bien tipados se pueden conocer en tiempo de compilación. Un programa bien tipado se puede utilizar sin efectuar comprobaciones de tipo en tiempo de ejecución, estando garantizado que no se producirán errores de tipo durante el cómputo.

- Polimorfismo: Permite que una misma función se pueda aplicar a parámetros de diferentes tipos, dependiendo del contexto en el que la función se utilice.

1.5 TIPOS DE DATOS

En lenguajes de programación un tipo de dato es un atributo de una parte de los datos que indica al ordenador (y/o al programador) algo sobre la clase de datos sobre los que se va a procesar.

Un tipo de datos define un conjunto de valores y las operaciones sobre estos valores. Casi todos los lenguajes de programación explícitamente incluyen la notación del tipo de datos, aunque lenguajes diferentes pueden usar terminología diferente. La mayor parte de los lenguajes de programación permiten al programador definir tipos de datos adicionales, normalmente combinando múltiples elementos de otros tipos y definiendo las operaciones del nuevo tipo de dato. Por ejemplo, un programador puede crear un nuevo tipo de dato llamado "Persona" que especifica que el dato interpretado como Persona incluirá un nombre y una fecha de nacimiento.

Un tipo de dato entero en computación es un tipo de dato que puede representar un subconjunto finito de los números enteros. El número mayor que puede representar depende del tamaño del espacio usado por el dato y la posibilidad (o no) de representar números negativos. Los tipos de dato entero disponibles y su tamaño dependen del lenguaje de programación usado, así como la arquitectura en cuestión.

Tipos de datos en coma flotante Se usan para representar números con partes fraccionarias. Hay dos tipos de coma flotante: float y double. El primero reserva almacenamiento para un

número de precisión simple de 4 bytes y el segundo lo hace para un número de precisión doble de 8 bytes.

Tipo de dato carácter (Char) Es cualquier signo tipográfico, puede ser una letra, un número, un signo de puntuación o un espacio. Este término se usa mucho en computación. Un valor de tipo carácter es cualquier carácter que se encuentre dentro del conjunto ASCII ampliado, el cual está formado por los 128 caracteres del ASCII más los 128 caracteres especiales que presenta, en este caso, IBM.

Los valores ordinales del código ASCII ampliado se encuentran en el rango de 0 a 255. Dichos valores pueden representarse escribiendo el carácter correspondiente encerrado entre comillas simples (apóstrofos).Así, podemos escribir: 'A' < 'a 'Que significa: "El valor ordinal de A es menor que el de a" o "A está antes que a "Un valor de tipo carácter (char en inglés) se guarda en un byte de memoria. La única operación (además de las relacionales) que podemos hacer con caracteres es la concatenación concatenando dos caracteres, por ejemplo 'a' y 'X' obtendríamos la cadena "aX".

Tipo de dato lógico El tipo de dato lógico o booleano es en computación aquel que puede representar valores de lógica binaria, esto es, valores que representen falso o verdadero. Se utiliza normalmente en programación, estadística, electrónica, matemáticas.

Palabra reservada una palabra reservada es una palabra que tiene un significado Gramatical especial para ese lenguaje y no puede ser utilizada como un identificador en ese lenguaje.

Por ejemplo, en SQL, un usuario no puede ser llamado "group", porque la palabra group es usada para indicar que un identificador se refiere a un grupo, no a un usuario. Al tratarse de una palabra clave su uso queda restringido. Ocasionalmente la especificación de un lenguaje de programación puede tener palabras reservadas que están previstas para un posible uso en futuras versiones. En Java const y goto son palabras reservadas — no tienen significado en Java, pero tampoco pueden ser usadas como identificadores. Al reservar los términos pueden ser implementados en futuras versiones de Java, si se desea, sin que el código fuente más antiguo escrito en Java deje de funcionar.

"*Tipos de Datos En lenguajes de programación un tipo de dato es un atributo de una parte de los datos que indica al ordenador (y/o al programador) algo sobre la clase de datos sobre los que se va a procesar.*"

Todos los datos tienen un tipo asociado con ellos. Un dato puede ser un simple carácter, tal como `b', un valor entero tal como 35. El tipo de dato determina la naturaleza del conjunto de valores que puede tomar una variable.

- Numéricos
- Simples Lógicos
- Alfanuméricos (String)
- Tipos de datos Arreglos (Vectores, Matrices)
- Estructurados Registros (Def. por el Archivos usuario) Apuntadores

1.6 TIPOS DE DATOS SIMPLES

- **Datos Numéricos:** Permiten representar valores escalares de forma numérica, esto incluye a los números enteros y los reales. Este tipo de datos permiten realizar operaciones aritméticas comunes.
- **Datos Lógicos:** Son aquellos que solo pueden tener dos valores (cierto o falso) ya que representan el resultado de una comparación entre otros datos (numéricos o alfanuméricos).
- **Datos Alfanuméricos (String):** Es una secuencia de caracteres alfanuméricos que permiten representar valores identificables de forma descriptiva, esto incluye

nombres de personas, direcciones, etc. Es posible representar números como alfanuméricos, pero estos pierden su propiedad matemática, es decir no es posible hacer operaciones con ellos. Este tipo de datos se representan encerrados entre comillas

1.7 LOS TIPOS DE DATOS EN UN SENTIDO AMPLIO

Un tipo de datos define un conjunto de valores y las operaciones sobre estos valores. Casi todos los lenguajes de programación explícitamente incluyen la notación del tipo de datos, aunque lenguajes diferentes pueden usar terminología diferente.

La mayor parte de los lenguajes de programación permiten al programador definir tipos de datos adicionales, normalmente combinando múltiples elementos de otros tipos y definiendo las operaciones del nuevo tipo de dato. Por ejemplo, un programador puede crear un nuevo tipo de dato llamado "Persona" que especifica que el dato interpretado como Persona incluirá un nombre y una fecha de nacimiento.

1.8 LA UTILIZACIÓN DE LOS TIPOS DE DATOS

1. Entero

Los tipos de datos enteros suelen representar números enteros en la programación. El valor de un número entero se mueve de un número entero a otro sin reconocer los números fraccionarios entre ellos. El número de dígitos puede variar en función del dispositivo, y algunos lenguajes de programación pueden permitir valores negativos.

2. Carácter

En la codificación, las letras del alfabeto denotan caracteres. Los programadores pueden representar estos tipos de datos como (CHAR) o (VARGCHAR), y pueden ser caracteres individuales o una cadena de letras. Los caracteres suelen ser cifras de longitud fija que, por defecto, son de 1 octeto—una unidad de 8 bits de información digital—pero pueden aumentar hasta 65.000 octetos.

3. Fecha

Este tipo de datos almacena una fecha de calendario con otra información de programación. Las fechas suelen ser una combinación de enteros o cifras numéricas. Como suelen ser valores enteros, algunos programas pueden almacenar operaciones matemáticas básicas como los días transcurridos desde ciertos eventos o los días que faltan para un evento próximo.

4. Punto flotante (real)

Los tipos de datos de coma flotante representan los números fraccionarios en la programación. Existen dos tipos de datos de coma flotante principales, que varían en función del número de valores permitidos en la cadena:

Hoja de cálculo: Un tipo de datos que suele permitir hasta siete puntos después de un decimal.

Doble: Un tipo de dato que permite hasta 15 puntos después de un decimal.

5. Largo

Los tipos de datos largos suelen ser enteros de 32 o 64 bits en el código. A veces, éstos pueden representar enteros con 20 dígitos en cualquier dirección, positiva o negativa. Los programadores utilizan un ampersand para indicar que el tipo de datos es una variable larga.

6. Corto

Al igual que el tipo de datos long, un short es un número entero variable. Los programadores los representan como números enteros, y pueden ser positivos o negativos. A veces, un tipo de datos short es un único número entero.

7. Cadena

Un tipo de datos de cadena es una combinación de caracteres que puede ser constante o variable. A menudo incorpora una secuencia de tipos de datos de caracteres que dan lugar a comandos específicos dependiendo del lenguaje de programación. Las cadenas pueden incluir letras mayúsculas y minúsculas, números y signos de puntuación.

8. Booleano

Los datos booleanos son los que utilizan los programadores para mostrar la lógica en el código. Suele ser uno de dos valores—verdadero o falso—para aclarar las declaraciones condicionales. Pueden ser respuestas a escenarios «si/cuando», donde el código indica si un usuario realiza una determinada acción. Cuando esto ocurre, los datos booleanos dirigen la respuesta del programa' que determina el siguiente código en la secuencia.

9. Nada

El tipo de dato nada muestra que un código no tiene valor. Esto puede indicar que falta un código, que el programador inició el código de forma incorrecta o que hubo valores que desafían la lógica prevista. También se denomina «tipo anulable»;

10. Anular

Al igual que el tipo nothing, el tipo void contiene un valor que el código no puede procesar. Los tipos de datos void indican al usuario que el código no puede devolver una respuesta.

Los programadores pueden utilizar o encontrar el tipo de datos void en las primeras pruebas del sistema cuando aún no hay respuestas programadas para los pasos futuros.

1.9 Ejemplos de tipos de datos

Los tipos de datos pueden variar en función del tamaño, la longitud y el uso dependiendo del lenguaje de codificación. Aquí hay algunos ejemplos de los tipos de datos enumerados anteriormente que puede encontrar al programar:

Entero

Los enteros son dígitos que representan sólo números enteros. Algunos ejemplos de números enteros son:

425

65

9

Personajes

Los caracteres son letras u otras cifras que los programadores pueden combinar en una cadena. Algunos ejemplos de caracteres son

a

^

¡!

Fecha

Los programadores pueden incluir fechas individuales, rangos o diferencias en su código. Algunos ejemplos podrían ser:

2009-09-15

1998-11-30 09:45:87

SYSDATETIME ()

Largo

Los tipos de datos largos son números enteros, tanto positivos como negativos, que tienen muchos valores de posición. Algunos ejemplos son:

-398,741,129,664,271

9,000,000,125,356,546

Breve

Los tipos de datos cortos pueden ser hasta varios enteros, pero siempre son menores que los datos largos. Algunos ejemplos son:

-27,400

5,428

17

Punto flotante (real)

Los tipos de datos flotantes podrían tener este aspecto:

float num1 = 1.45E2

float num2 = 9.34567

Un ejemplo similar, pero a menudo más largo, podría ser el doble de punto flotante:

double num2 = 1.87358497267482791E+222

double num2 = 3.198728764857268945

El tipo doble de punto flotante puede proporcionar valores más precisos, pero también puede requerir memoria adicional para procesar.

Cadena

Las cadenas son una combinación de cifras que incluye letras y signos de puntuación. En algún código, esto podría tener el siguiente aspecto:

String a = new String(«Open»)

String b = new String(«La puerta»)

String c = new String(«¡Diga Hola!»)

Estos pueden ser mandatos independientes, o pueden trabajar juntos.

Booleano

Los datos booleanos pueden ayudar a guiar la lógica de un código. He aquí algunos ejemplos de su uso:

bool béisbolEsMejor = falso;

bool footballIsBest = true;

Dependiendo del programa, el código puede dirigir al usuario final a diferentes pantallas en función de su selección.

Nada

Nada significa que un código no tiene valor, pero el programador codificó algo distinto al dígito 0. Esto es a menudo «Null,» «NaN» o «Nada» en el código. Un ejemplo de esto es:

Dim opción = Nada

Program.WriteWords(x Is Nothing)

Vacío

El tipo de dato void en la codificación funciona como un indicador de que el código puede no tener todavía una función o una respuesta. Esto podría aparecer como:

int nombre_función (void).

UNIDAD 2

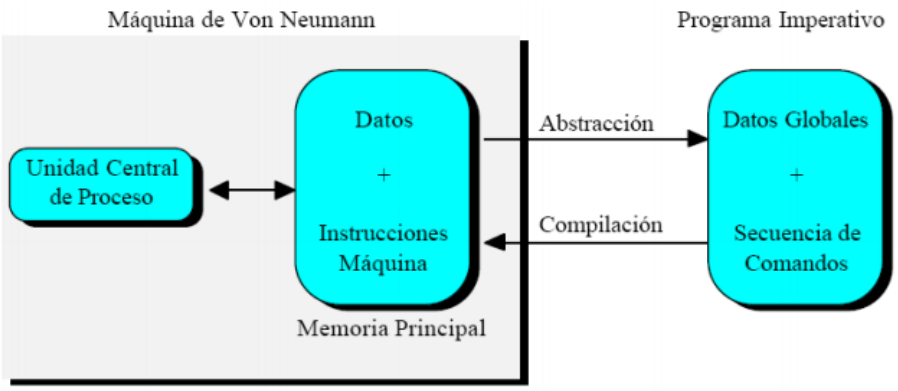
PROGRAMACIÓN FUNCIONAL

2.1 ORÍGENES

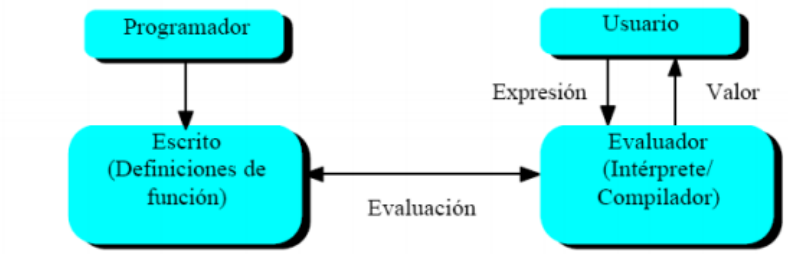
Los orígenes de la programación funcional pueden rastrearse al matemático Alonzo Church, que trabajaba en la Universidad de Princeton, y, al igual que otros matemáticos de allí, estaba interesado en la matemática abstracta, particularmente en el poder computacional de ciertas máquinas abstractas. Las preguntas que se hacía eran, por ejemplo: si dispusiésemos de máquinas de un ilimitado poder de cómputo, ¿qué tipos de problemas se podrían solucionar?, o ¿se pueden resolver todos los problemas? Para contestar este tipo de preguntas, Church desarrolló un lenguaje abstracto, denominado Cálculo Lambda, que el cual sólo realizaba evaluación de expresiones usando funciones como mecanismo de cómputo. Este lenguaje abstracto no tenía en cuenta limitaciones concretas de implementación de ningún tipo. Al mismo tiempo que Church, otro matemático, Alan Turing, desarrolló una máquina abstracta para intentar resolver el mismo tiempo de problemas planteados por Church. Después se demostró que ambos enfoques son equivalentes.

Las primeras computadoras digitales se construyeron siguiendo un esquema de arquitectura denominado de Von Neumann, que es básicamente una implementación de la máquina de Turing a una máquina real. Esta máquina forzó de alguna manera el lenguaje en el cual se escriben sus programas, justamente el paradigma procedimental, el cual, como menciona Backus en un muy famoso artículo que escribió al recibir el premio Turing en 1978 (Backus 1978), tiene tantísimos defectos, que muchos programadores padecemos aun hoy. La programación funcional se aparta de esta concepción de máquina, y trata de ajustarse más a la forma de resolver el problema, que a las construcciones del lenguaje necesarias para cumplir con la ejecución en esta máquina. Por ejemplo, un condicionamiento de la máquina de Von-Neumann es la memoria, por lo cual los programas procedimentales poseen variables. Sin embargo, en la programación funcional pura, las variables no son necesarias,

ya que no se considera a la memoria necesaria, pudiéndose entender un programa como una evaluación continua de funciones sobre funciones. Es decir, la programación funcional posee un estilo de computación que sigue la evaluación de funciones matemáticas y evita los estados intermedios y la modificación de datos durante la misma. Hoy en día existen diversos lenguajes funcionales. Se podría considerar como uno de los primeros lenguajes funcionales al LISP, que actualmente sigue en uso, sobre todo en áreas de la inteligencia artificial. También un pionero de este paradigma es APL desarrollado en los años 60 (Iverson 1962). El linaje funcional se enriqueció en los años 70, con el aporte de Robin Milner de la Universidad de Edimburgo al crear el lenguaje ML. Éste se subdividió posteriormente en varios dialectos tales como Objective Caml y Standard ML. A fines de los años 80, a partir de un comité, se creó el lenguaje Haskell, en un intento de reunir varias ideas dispersas en los diferentes lenguajes funcionales (un intento de estandarizar el paradigma). Este año Microsoft Research ha incluido un nuevo lenguaje (funcional), denominado F#, a su plataforma .NET.



Modelo imperativo (de Labra 1998)



2.2 QUÉ ES LA PROGRAMACIÓN FUNCIONAL

En este breve artículo, intentaré explicar la utilidad y potencia de la programación funcional, por medio de pequeños ejemplos, para comprender más rápidamente esta filosofía de programación. Dado el nombre del paradigma, sabemos básicamente que lo central en el mismo es la idea de función, que podríamos decir es análoga a lo que conocemos de funciones de la matemática. Por ejemplo, podemos escribir en el lenguaje funcional Haskell: Factorial: int -> int factorial 0 = 1 factorial n = n * factorial (n-1) Es decir, la última es una función sencilla, parecida a la que conocemos de las matemáticas de la secundaria, que permite calcular la factorial de un número entero (ver definición de la función factorial más abajo).

Principales características

Para demostrar las principales características de los lenguajes de programación funcionales modernos, vamos a utilizar el lenguaje Haskell. Este es un lenguaje funcional puro, de propósito general, que incluye muchas de las últimas innovaciones en el desarrollo de los lenguajes de programación funcional, como son las funciones de orden superior, evaluación perezosa (lazy evaluation), tipos polimórficos estáticos, tipos definidos por el usuario, encaje por patrones (pattern matching), y definiciones de listas por comprensión. Tiene además otras características interesantes como el tratamiento sistemático de la sobrecarga, la facilidad en la definición de tipos abstractos de datos, el sistema de entrada/salida puramente funcional y la posibilidad de utilización de módulos. Aunque seria imposible explicar con detalle cada una de estas características en este artículo, daremos una breve explicación de alguna de ellas. El lector interesado puede recurrir a un excelente libro en castellano (Ruiz 2004) y otros dos en inglés (Thompson 1999; Doets & van Eijck 2004) para profundizar en el tema.

2.3 EL TIPO DE DATOS.

Tipos de datos, tuplas y listas

Los lenguajes funcionales, en particular Haskell, tienen un rico conjunto de datos atómicos predefinidos, tales como los numéricos int, integer (de mayor precisión que el anterior), float, double, etc., y además los tipos char y bool.

El sistema de tipos de Haskell es uno de los más sofisticados que existen. Es un sistema polimórfico, que permite una gran flexibilidad de programación, pero a la vez mantiene la correctitud de los programas. Contrariamente a la mayoría de los lenguajes de programación procedimentales actuales, Haskell utiliza un sistema de inferencias de tipos, es decir sabe el tipo resultante de una expresión, por lo que las anotaciones de tipo en un programa son opcionales.

La parte más interesante de Haskell en relación con los tipos son los constructores, las tuplas y las listas. Una tupla es un dato compuesto donde el tipo de cada componente puede ser distinto. Una de las utilidades de este tipo de datos es cuando una función tiene que devolver más de un valor:

```
predSuc :: Integer → (Integer,Integer)
predSuc x = (x-1,x+1)
```

Las listas son colecciones de cero o más elementos de un mismo tipo (a diferencia de las tuplas que pueden tenerlos de diferentes). Los operadores utilizados son el [] y (:). El primero representa una lista vacía, y el segundo denominado cons o constructor, permite añadir un elemento al principio de una lista, construyendo la lista en función de agregar elementos a la misma, por ejemplo [Thompson99]:

```
4 : 2 : 3 : []
```

da lugar a la lista [4, 2, 3]. Su asociatividad es hacia la derecha. Un tipo particular de lista son las cadenas de caracteres. Para terminar, diremos que el constructor utilizado para declarar el tipo correspondiente a las distintas funciones es el símbolo →.

Patrones

Como vimos anteriormente, una función puede ser definida como:

```
f <pat1> <pat2> . . . <patn> = <expresión>
```

donde cada una de las expresiones <pat1> <pat2> . . . <patn> representa un argumento de la función, al que también podemos denominar como patrón. Cuando una función está definida mediante más de una ecuación, será necesario evaluar uno o más argumentos de la función para determinar cuál de las ecuaciones aplicar. Este proceso se llama en castellano encaje de patrones. En los ejemplos anteriores se utilizó el patrón más trivial: una sola variable. Como ejemplo, considérese la definición de factorial:

```
fact n = product [1..n]
```

Si se desea evaluar la expresión "fact 3" es necesario hacer coincidir la expresión "3" con el patrón "n" y luego evaluar la expresión obtenida a partir de "product [1..n]" substituyendo la "n" con el "3". Una de los usos más útiles es la aplicación de los patrones a las listas. Por ejemplo, la siguiente función toma una lista de valores enteros y los suma:

```
suma :: [Integer] → Integer
suma [] = 0 -- caso base
suma (x : xs) = x + suma xs -- caso recursivo
```

Las dos ecuaciones hacen que la función esté definida para cualquier lista. La primera será utilizada si la lista está vacía, mientras que la segunda se usará en otro caso. Tenemos la siguiente reducción (evaluación):

```
     suma [1,2,3]
⇒ { sintaxis de listas}
     suma (1 : (2: (3: [])))
⇒ { segunda ecuación de suma (x ← 1, xs ← 2 : (3 : [])) }
     1 + suma (2 : (3 : []))
⇒{ segunda ecuación de suma (x ←2, xs ← 3 : []) }
     1 + (2 + suma (3 : []))
⇒{ segunda ecuación de suma (x ← 3, xs ← []) }
     1 + (2 + (3 + suma []))
⇒{ primera ecuación de suma}
     1 + (2 + (3 + 0))
⇒{ definición de (+) tres veces}
     6
```

Existen otros tipos de patrones

Patrones Anónimos:

Se representan por el carácter (_) y encajan con cualquier valor, pero no es posible referirse posteriormente a dicho valor.

Patrones con nombre:

Para poder referirnos al valor que está encajando.

Patrones n+k:

encajan con un valor entero mayor o igual que k.

En Haskell, el nombre de una variable no puede utilizarse más de una vez en la parte izquierda de cada ecuación en una definición de función.

2.4 FUNCIONES.

En programación, una **función** es una sección de un programa que calcula un valor de manera independiente al resto del programa.

Una función tiene tres componentes importantes:

- los **parámetros**, que son los valores que recibe la función como entrada;
- el **código de la función**, que son las operaciones que hace la función; y
- el **resultado** (o **valor de retorno**), que es el valor final que entrega la función.

En esencia, una función es un mini programa. Sus tres componentes son análogos a la entrada, el proceso y la salida de un programa.

En programación a los procedimientos y funciones también se les conoce por el nombre de rutinas, subrutinas o subprogramas. Son bloques de instrucciones que realizan tareas específicas. Las rutinas se declaran una sola vez, pero pueden ser utilizadas, mediante llamadas, todas las veces que se quiera en un programa. Una rutina es independiente del resto del programa por lo que, en principio, facilita el diseño, el seguimiento y la corrección de un programa. Pueden además almacenarse independientemente en colecciones llamadas librerías o unidades, lo cual permite que sean utilizadas en cualquier programa. De hecho, existen funciones y procedimientos que vienen ya construidos para el lenguaje de programación TurboPascal (Cos, Sin, Exp, ReadLn, WriteLn, ClrScr...), que están almacenados en distintas unidades (System, Crt, Graph...) y que el programador puede emplear en sus programas. Además, el programador puede construir sus propias unidades con las constantes, tipos de datos, variables, funciones y procedimientos, que desee incluir.

En el seudo lenguaje una función se declara de la siguiente manera:

```
funcion <nombre> ( param1 : tipo1 , ..., paramn : tipon ) : tipo
variables
    <declaraciones>
inicio
    <instrucciones>
    retornar <expresión>
fin_funcion
```

Donde,

- <nombre>: representa el nombre de la función
- $param_i$: representa el parámetro i-ésimo de la función.
- $tipo_i$: representa el tipo del i-ésimo parámetro de la función.

2.4.1 FUNCIONES RECURSIVAS

Una función recursiva es una función que se define en términos de si misma, es decir, que el resultado de la función depende de resultados obtenidos de evaluar la misma función con otros valores.

Se debe tener mucho cuidado en la definición de funciones recursivas, pues si no se hace bien, la función podría requerir de un cálculo infinito o no ser calculable.

Procedimientos

En muchos casos existen porciones de código similares que no calculan un valor si no que por ejemplo, presentan información al usuario, leen una colección de datos o calculan más de un valor. Como una función debe retornar un único valor 2 este tipo de porciones de código no se podrían codificar como funciones. Para superar este inconveniente se creó el concepto de procedimiento. Un procedimiento se puede asimilar a una función que puede retornar más de un valor mediante el uso de parámetros por referencia. Una función puede retornar más de un valor si ella usa parámetros por referencia. En este texto los parámetros por referencia sólo se usarán en los procedimientos ya que, es una muy mala técnica de programación el uso de parámetros por referencia en funciones. Esta consideración se hace pues, desde el punto de vista matemático, una función no puede modificar los valores de los parámetros.

Los procedimientos se usan para evitar duplicación de código y conseguir programas más cortos. Son también una herramienta conceptual para dividir un problema en subproblemas logrando de esta forma escribir más fácilmente programas grandes y complejos.

2.4.2 PARAMETROS POR VALOR Y POR REFERENCIA

Parámetros por valor

Los parámetros convencionales son por valor, es decir, a la función o procedimiento se le envía un valor que almacena en la variable correspondiente al parámetro, la cual es local, de manera que su modificación no tiene efecto en el resto del programa.

2.4.3 PARÁMETROS POR REFERENCIA

Si un procedimiento tiene un parámetro por referencia quiere decir que no está recibiendo un valor sino una referencia a una variable, es decir la misma variable (posición en memoria y valor) que envía el algoritmo que hace el llamado al procedimiento con un alias (el nombre de la variable del parámetro que se recibe por referencia). Por lo tanto, cualquier modificación al parámetro que se haga dentro del procedimiento, tiene efectos en el algoritmo que realizó el llamado al procedimiento. Cuando se ejecuta un procedimiento con uno o varios parámetros por referencia, ni un literal ni una constante se pueden poner en la posición de alguno de estos parámetros, es decir, ni las constantes ni los literales pueden ser pasados por referencia a un procedimiento. Un parámetro por referencia se especifica en seudo-lenguaje, anteponiendo la palabra ref a su definición.

2.5 VARIABLES LOCALES Y GLOBALES

Variables Globales

Son variables definidas al comienzo del programa (antes de cualquier función), que se pueden usar a lo largo de todo el programa, es decir, dentro del algoritmo principal y en cada función definida en el programa.

Variables Locales

Son variables definidas dentro de cada función y/o procedimiento, y que solo se pueden usar en la función y/o procedimiento, en la que son declaradas.

Una buena técnica de programación es no usar, o usar la menor cantidad de variables globales, de tal forma que las funciones y/o procedimientos que se creen no dependan de elementos externos, en este caso las variables globales, para realizar su proceso. El no usar variables globales dentro de una función y/o procedimiento garantiza su fácil depuración y seguimiento.

2.6 INTERVALOS

En numerosos trabajos de planificación y programación de tareas, éstas poseen restricciones de tiempo motivadas por aspectos físicos de los materiales involucrados en la fabricación, por normas de proceso, por horarios de trabajo, o por otras imposiciones en los intervalos de procesamiento. En estos casos, la capacidad del sistema, es decir, la cantidad de recursos disponibles, juega un papel fundamental.

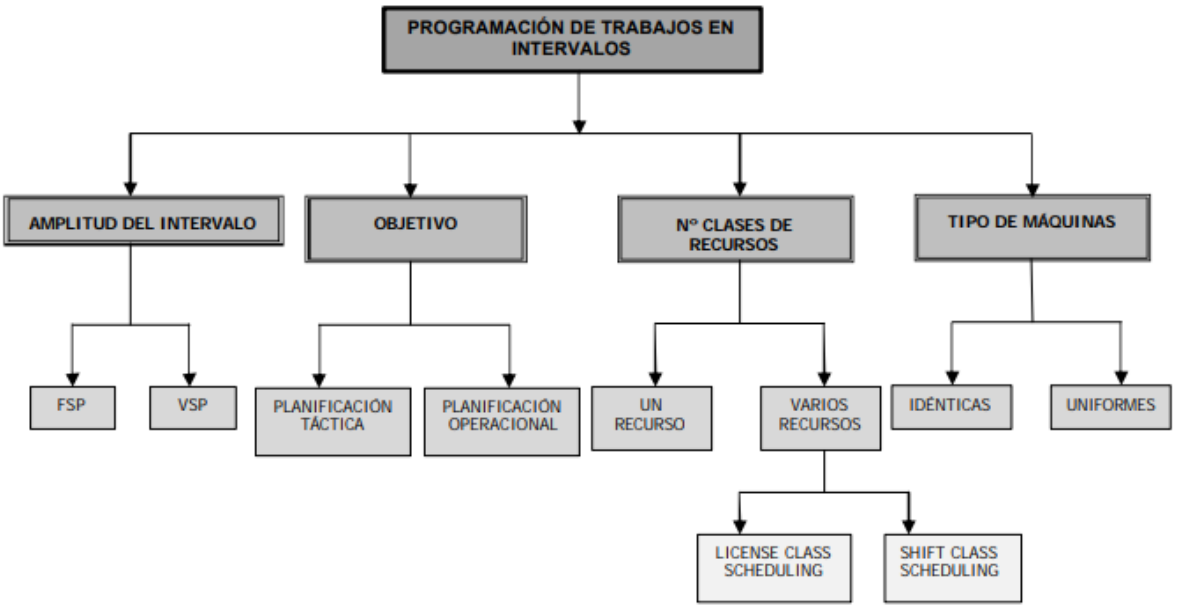
En este tipo de sistemas, la planificación de la capacidad trata tanto de la asignación de los recursos a las tareas atendiendo a un criterio de optimización (planificación operacional) como al cálculo de la capacidad necesaria para completar todas las tareas (planificación táctica). El enfoque del problema como de una sola etapa para la realización de las tareas se encuentra en multitud de disciplinas donde existen actividades programadas de antemano, es decir, donde se parte de un conjunto de tareas planificadas previamente y el análisis se centra en gestionar los recursos necesarios convenientemente para realizar dichas tareas. La programación de trabajos en intervalos con una sola etapa de procesamiento, puede ser definida como sigue:

Sea un conjunto de n trabajos J1,…,Jn cada uno con un intervalo [ai,bi] en el que debe ser procesado, un tiempo de proceso ti, un peso o prioridad wi y perteneciente a una determinada clase de trabajo di dentro de un conjunto de D diferentes clases de trabajos (Figura 1). Para la realización de los trabajos se dispone de un conjunto de máquinas M={M1,…, Mm} cada una con un intervalo de disponibilidad [ij, fj], un coste de utilización uj, y perteneciente a una determinada clase de máquina cj dentro de un conjunto de C clases diferentes de máquinas. La compatibilidad entre clases de trabajos y máquinas se expresa a través de una matriz de compatibilidad LDxC definida como sigue:

$$L_{DXC} = L(d, c) = \begin{cases} 1 & \text{si un trabajo de clase d puede ser procesado por una máquina de clase c} \\ 0 & \text{en otro caso} \end{cases}$$

2.7 CLASIFICACIÓN

La programación de trabajos en intervalos aparece en la literatura con diferentes variaciones, dependiendo de los intervalos de tiempo que se tenga para el procesamiento de las tareas, del número de recursos (máquinas) de distintas clases existente, de las características que motivan las diferentes clases de trabajos y recursos y, evidentemente, del objetivo definido. La Figura resume la clasificación del problema que se desarrolla más adelante:



FSP FRENTE A VSP

Una primera clasificación de estos sistemas se produce según la amplitud del intervalo para el procesamiento de las tareas. Según ello, existe un primer tipo de problema donde el tiempo de proceso de las tareas coincide exactamente con la amplitud del intervalo, recibiendo este grupo el nombre de Fixed Job Scheduling Problem (FSP). Si el tiempo de proceso de alguna de las tareas es menor que la amplitud del intervalo, nos encontramos ante un Variable Job Scheduling Problem (VSP).

Calcular con enteros

Con una división de enteros (Int), se trunca a la parte entera: 10/3 es 3. Sin embargo, no siempre es necesario usar números del tipo Float con divisiones. Por el contrario, muchas veces el resto de la división es más interesante que la fracción decimal. El resto de la división 10/3 es 1.

Funciones devuelven siempre el mismo valor

- Los lenguajes funcionales puros tienen la propiedad de transparencia referencial
- Como consecuencia, en programación funcional, una función siempre devuelve el mismo valor cuando se le llama con los mismos parámetros
- Las funciones no modifican ningún estado, no acceden a ninguna variable ni objeto global y modifican su valor

Diferencia entre declaración y modificación de variables

- En programación funcional pura una vez declarada una variable no se puede modificar su valor
- En algunos lenguajes de programación (como Scala) este concepto se refuerza definiendo la variable como inmutable (con la directiva val).
- En programación imperativa es habitual modificar el valor de una variable en distintos pasos de ejecución.

2.8 OPERADORES.

Un operador es una función de dos parámetros, que se escribe entre estos en vez de delante. Los nombres de funciones se forman con letras y cifras; los nombres de operadores se forman con símbolo.

OPERADORES JAVA ARITMÉTICOS

Los operadores aritméticos en java son:

- **+** Suma. Los operandos pueden ser enteros o reales
- **-** Resta. Los operandos pueden ser enteros o reales
- Multiplicación. Los operandos pueden ser enteros o reales
- **/** División. Los operandos pueden ser enteros o reales. Si ambos son enteros el resultado es entero. En cualquier otro caso el resultado es real.
- **%** Resto de la división. Los operandos pueden ser de tipo entero o real.

Ejemplo de operaciones aritméticas:

```
int a = 10, b = 3;
double v1 = 12.5, v2 = 2.0;
char c1='P', c2='T';
```

Operación	Valor	Operación	Valor	Operación	Valor
a+b	13	v1+v2	14.5	c1	80
a-b	7	v1-v2	10.5	c1 + c2	164
a*b	30	v1*v2	25.0	c1 + c2 + 5	169
a/b	3	v1/v2	6.25	c1 + c2 + '5'	217
a%b	1	v1%v2	0.5		

En aquellas operaciones en las que aparecen operandos de distinto tipo, java convierte los valores al tipo de dato de mayor precisión de todos los datos que intervienen. Esta conversión es de forma temporal, solamente para realizar la operación. Los tipos de datos originales permanecen igual después de la operación.

Los tipos short y byte se convierten automáticamente a int.

Por ejemplo:

```
int i = 7;
double f = 5.5;
char c = 'w';
```

Operación	Valor	Tipo
i + f	12.5	double
i + c	126	int
i + c – '0'	78	int
(i + c) – (2 * f / 5)	123.8	double

2.9 OPERADORES JAVA RELACIONALES

Los operadores relacionales comparan dos operandos y dan como resultado de la comparación verdadero o falso.

Los operadores relacionales en java son:

< Menor que

> Mayor que

<= Menor o igual

>= Mayor o igual

!= Distinto

== Igual

Los **operandos** tienen que ser de **tipo primitivo**.

Por ejemplo:

int a = 7, b = 9, c = 7;

Operación	Resultado
a==b	false
a >=c	true
b < c	false
a != c	false

OPERADORES JAVA LÓGICOS

Los operadores lógicos se utilizan con operandos de tipo boolean. Se utilizan para construir expresiones lógicas, cuyo resultado es de tipo true o false.

Los operadores lógicos en Java son:

&& AND. El resultado es verdadero si los dos operandos son verdaderos. El resultado es falso en caso contrario. Si el primer operando es falso no se evalúa el segundo, ya que el resultado será falso.

|| OR. El resultado es falso si los dos operandos son falsos. Si uno es verdadero el resultado es verdadero. Si el primer operando es verdadero nose evalúa el segundo.

! NOT. Se aplica sobre un solo operando. Cambia el valor del operando de verdadero a falso y viceversa.

Las definiciones de las operaciones OR, AND y NOT se recogen en unas tablas conocidas como **tablas de verdad.**

OPERADORES JAVA UNITARIOS.

Los operadores unitarios en java son:

– + signos negativo y positivo

++ -- incremento y decremento

~ complemento a 1

! NOT. Negación

Estos operadores **afectan a un solo operando**.

El **operador ++** (operador incremento) incrementa en 1 el valor de la variable.

OPERADORES JAVA A NIVEL DE BITS

Realizan la manipulación de los bits de los datos con los que operan.

Los datos deben ser de tipo entero.

Los operadores a nivel de bits en java son:

& and a nivel de bits

| or a nivel de bits

^ xor a nivel de bits

<< desplazamiento a la izquierda, rellenando con ceros a la derecha

>> desplazamiento a la derecha, rellenando con el bit de signo por la izquierda

>>> desplazamiento a la derecha rellenando con ceros por la izquierda

OPERADORES JAVA DE ASIGNACIÓN.

Se utilizan para asignar el valor de una expresión a una variable.

Los operandos deben ser de tipo primitivo.

Los operadores de asignación en java son:

= Asignación
+= Suma y asignación
–= Resta y asignación
*= Producto y asignación
/= División y asignación
%= Resto de la división entera y asignación
<<= Desplazamiento a la izquierda y asignación
>>= Desplazamiento a la derecha y asignación
>>>= Desplazamiento a la derecha y asignación rellenando con ceros
&= AND sobre bits y asignación
|= OR sobre bits y asignación
^= XOR sobre bits y asignación

Si los dos operandos de una expresión de asignación (el de la izquierda y el de la derecha) son de distinto tipo de datos, **el valor de la expresión de la derecha se convertirá al tipo del operando de la izquierda**.

Por ejemplo, una expresión de tipo real (float, double) se truncará si se asigna a un entero, o una expresión en de tipo double se redondeará si se asigna a una variable de tipo float.

En Java están permitidas las asignaciones múltiples.

Ejemplo: x = y = z = 3; equivale a x = 3; y = 3; z = 3;

Ejemplo de asignaciones en Java:

a += 3; equivale a a = a + 3;

a *= 3; equivale a a = a * 3;

OPERADOR JAVA CONDICIONAL

Se puede utilizar en sustitución de la sentencia de control if-else, pero hace las instrucciones menos claras.

- El operador condicional java se forman con los caracteres **?** y **:**
- Se utiliza de la forma siguiente:
- expresión1 **?** expresión2 **:** expresión3
- Si expresión1 es cierta entonces se evalúa expresión2 y éste será el valor de la expresión condicional. Si expresión1 es falsa, se evalúa expresión3 y éste será el valor de la expresión condicional.

Ejemplo de operador condicional:

```
int i = 10, j;
j = (i < 0) ? 0 : 100;
```

PRIORIDAD Y ORDEN DE EVALUACIÓN DE LOS OPERADORES EN JAVA

La siguiente tabla muestra todos los operadores Java ordenados de mayor a menor prioridad. La primera línea de la tabla contiene los operadores de mayor prioridad y la última los de menor prioridad. Los operadores que aparecen en la misma línea tienen la misma prioridad.

Una expresión entre paréntesis siempre se evalúa primero y si están anidados se evalúan de más internos a más externos.

2.10 APLICACIONES DE LAS LISTAS

Una lista es una estructura de datos lineal que se puede representar simbólicamente como un conjunto de nodos enlazados entre sí. Las listas permiten modelar diversas entidades del mundo real como, por ejemplo, los datos de los alumnos de un grupo académico, los datos del personal de una empresa, los programas informáticos almacenados en un disco magnético, etc.

REPRESENTACIÓN EN MEMORIA

La Lista Lineal es una estructura dinámica, donde el número de nodos en una lista puede variar a medida que los elementos son insertados y removidos, el orden entre estos se establece por medio de un tipo de datos denominado punteros, apuntadores, direcciones o referencias a otros nodos, es por esto que la naturaleza dinámica de una lista contrasta con un arreglo que permanece en forma constante.

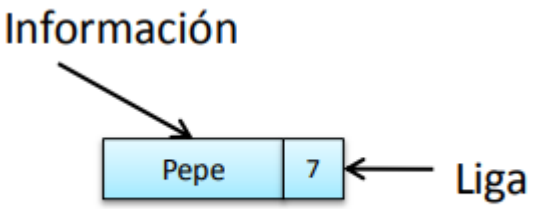


Fig. 4.1. Representación gráfica de un nodo.

Los nodos, en forma general, constan de dos partes: el campo información y el campo liga. El primero contendrá los datos a almacenar en la lista; el segundo será un puntero empleado para enlazar hacia el otro nodo de una lista.

Las operaciones más importantes que se realizan en las estructuras de datos Lista son las siguientes:

- Búsqueda
- Inserción
- Eliminación
- Recorrido

LISTAS ENLAZADAS

Una lista enlazada se puede definir como una colección de nodos o elementos. "El orden entre estos se establece por medio de punteros; esto es, direcciones o referencias a otros nodos. Un tipo especial de lista simplemente ligada es la lista vacía.

El apuntador al inicio de la lista es importante porque permite posicionarnos en el primer nodo de la misma y tener acceso al resto de los elementos. Si, por alguna razón, este apuntador se extraviara, entonces perderemos toda la información almacenada en la lista. Por otra parte, si la lista simplemente ligada estuviera vacía, entonces el apuntador tendrá el valor NULO."

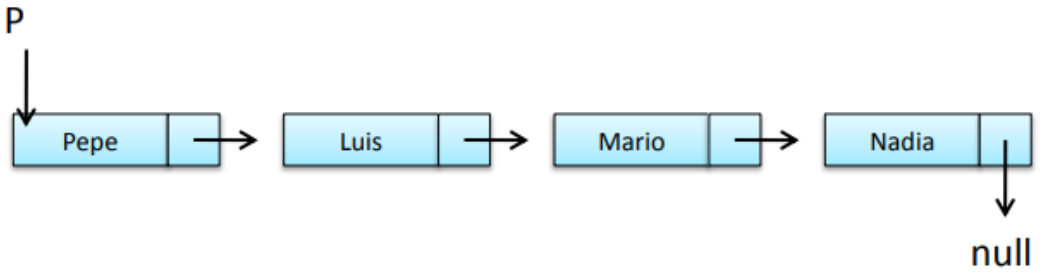


Fig. 4.2. Representación gráfica de una lista enlazada.

Dentro de las listas se pueden mencionar a las listas con cabecera, las cuales emplean a su primer nodo como contenedor de un tipo de valor (*, - , +, etc.). Un ejemplo de lista con nodo de cabecera es el siguiente:

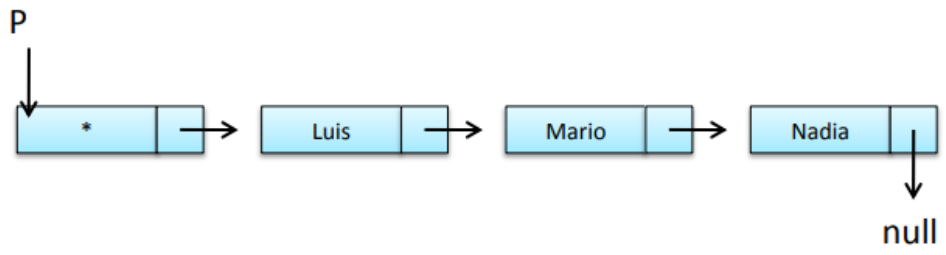


Fig. 4.2. Lista enlazada con nodo de cabecera.

LISTAS DOBLEMENTE ENLAZADAS

Se puede referir a una lista doble o doblemente ligada, a una colección de nodos que emplean además de su dato, dos elementos llamados punteros, los cuales se utilizan para especificar cuál es el elemento anterior y sucesor. Estos punteros se denominan Li (anterior) y Ld (sucesor). Tales punteros permiten moverse dentro de las listas un registro adelante o un registro atrás, según tomen las direcciones de uno u otro puntero. La estructura de un nodo en una lista doble es la siguiente:

Fig. 4.3. Liga doblemente enlazada.

Podemos mencionar a dos tipos de listas del tipo doblemente ligadas, las cuales se mencionan a continuación:

- Listas dobles lineales. En este tipo de lista el puntero Li del primer elemento apunta a NULL y el último elemento indica en su puntero Ld a NULL.
- Listas dobles circulares. Este otro tipo de lista tiene la particularidad que en su primer elemento el puntero, Li apunta al último elemento de la lista. Y el último elemento indica, en su puntero Ld, a el primer elemento de la lista.

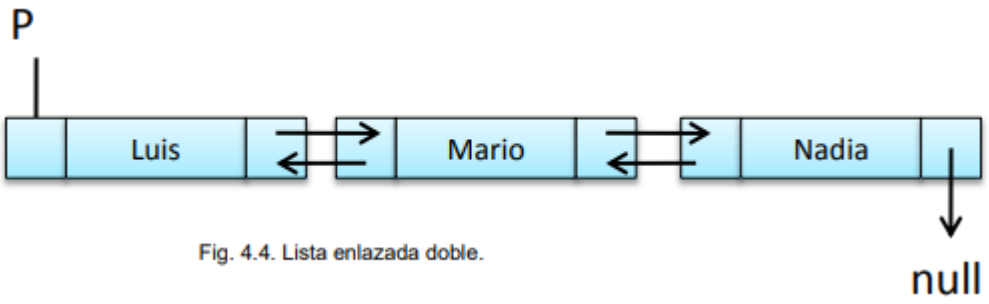


Fig. 4.4. Lista enlazada doble.

En la ilustración siguiente, se ejemplifica una lista doblemente ligada circular, la cual apunta a sus respectivos elementos.

Fig. 4.4.- Lista circular enlazada doble.

2.11 ÁRBOLES.

Un árbol es una estructura de datos ramificada (no lineal) que puede representarse como un conjunto de nodos enlazados entre sí por medio de ramas. La información contenida en un nodo puede ser de cualquier tipo simple o estructura de datos.

Los árboles permiten modelar diversas entidades del mundo real tales como, por ejemplo, el índice de un libro, la clasificación del reino animal, el árbol genealógico de un apellido, etc.

Una definición formal es la siguiente: Un árbol es una estructura de datos base que cumple una de estas dos condiciones:

- Es una estructura vacía
- Es un nodo de tipo base que tiene de 0 a N subárboles disjuntos entre sí.

Al nodo base, que debe ser único, se le denomina raíz y se establece el convenio de representarlo gráficamente en la parte superior.

Árboles binarios.

Se definen como árboles de grado 2. Esto es, cada nodo puede tener dos, uno o ningún hijo. Al tratarse como mucho de dos hijos, cada uno de ellos puede identificarse como hijo izquierdo o hijo derecho.

Implementación física.

El gráfico de un árbol es una representación conceptual cuya implementación física admite diversas posibilidades condicionadas, en primer lugar, por el dispositivo de almacenamiento del mismo (memoria principal o memoria externa). A los efectos del curso nos ocuparemos exclusivamente de la memoria principal en donde puede optarse por dos filosofías principales:

- Estructuras de datos estáticas, normalmente matrices.
- Estructuras de datos dinámicas

En cualquier caso, la representación física de un árbol requiere contemplar tres componentes:

- La clave (simple o compuesta).
- Dos punteros, indicando las ubicaciones respectivas de los nodos hijo izquierdo e hijo derecho.

Recorridos.

Se entiende por recorrido el tratamiento realizado para acceder a los diferentes nodos de un árbol. El recorrido puede afectar a la totalidad de los nodos del árbol (recorrido completo), por ejemplo, si se desea conocer el número de nodos, o finalizar anticipadamente en función de determinada/s circunstancia/s, por ejemplo, al encontrar el valor de una clave determinada.

UNIDAD 3

EVALUACIÓN PEREZOSA

3.1 HISTORIA

La evaluación perezosa fue introducida para el cálculo lambda por Christopher Wadsworth [8] y empleada por Plessey System 250 como una parte crítica de una metamáquina Lambda-Calculus, reduciendo la sobrecarga de resolución para el acceso a objetos en un espacio de direcciones de capacidad limitada. [9] Para los lenguajes de programación, fue introducido de forma independiente por Peter Henderson y James H. Morris [10] y por Daniel P. Friedman y David S. Wise.

3.2 APLICACIONES

La evaluación retrasada se usa particularmente en lenguajes de programación funcionales. Cuando se usa la evaluación retrasada, una expresión no se evalúa tan pronto como se vincula a una variable, sino cuando el evaluador se ve obligado a producir el valor de la expresión. Es decir, una declaración como x = expression;(es decir, la asignación del resultado de una expresión a una variable) claramente exige que se evalúe la expresión y se coloque el resultado x, pero lo que realmente está incluido xes irrelevante hasta que se necesite su valor. a través de una referencia xen alguna expresión posterior cuya evaluación podría aplazarse, aunque eventualmente el árbol de dependencias de rápido crecimiento se podaría para producir algún símbolo en lugar de otro para que el mundo exterior lo viera.

La evaluación retrasada tiene la ventaja de poder crear listas infinitas calculables sin bucles infinitos o cuestiones de tamaño que interfieran en el cálculo. Por ejemplo, se podría crear una función que cree una lista infinita (a menudo llamada flujo) de números de Fibonacci . El cálculo del n -ésimo número de Fibonacci sería simplemente la extracción de ese elemento de la lista infinita, lo que obligaría a evaluar solo los primeros n miembros de la lista.

Por ejemplo, en el lenguaje de programación Haskell , la lista de todos los números de Fibonacci se puede escribir como:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

En la sintaxis de Haskell, " :" antepone un elemento a una lista, taildevuelve una lista sin su primer elemento y zipWithusa una función específica (en este caso, la adición) para combinar los elementos correspondientes de dos listas para producir una tercera.

Siempre que el programador sea cuidadoso, solo se evalúan los valores que se requieren para producir un resultado en particular. Sin embargo, ciertos cálculos pueden hacer que el programa intente evaluar un número infinito de elementos; por ejemplo, solicitar la longitud de la lista o intentar sumar los elementos de la lista con una operación de plegado daría como resultado que el programa no terminara o se quedara sin memoria.

3.3 ESTRUCTURAS DE CONTROL

En casi todos [*cuantifique*] los lenguajes "ansiosos" comunes, *si las* declaraciones se evalúan de manera perezosa.

si a entonces b si no c
evalúa (a), entonces si y solo si (a) se evalúa como verdadero, evalúa (b), de lo contrario evalúa (c). Es decir, no se evaluarán ni (b) ni (c). Por el contrario, en un lenguaje ansioso, el comportamiento esperado es que

definir f (x, y) = 2 * xestablecer k = f (d, e)
seguirá evaluando (e) al calcular el valor de f (d, e) aunque (e) no se utilice en la función f. Sin embargo, las estructuras de control definidas por el usuario dependen de la sintaxis exacta, por ejemplo

definir g (a, b, c) = si a entonces b si no cl = g (h, i, j)
(i) y (j) se evaluarían en un lenguaje ávido. Mientras que, en un idioma perezoso,

l '= si h entonces yo si no j
Se evaluarían (i) o (j), pero nunca ambos.

La evaluación diferida permite que las estructuras de control se definan normalmente y no como primitivas o técnicas en tiempo de compilación. Si (i) o (j) tienen efectos secundarios o introducen errores de tiempo de ejecución, las diferencias sutiles entre (l) y (l ') pueden ser complejas. Por lo general, es posible introducir estructuras de control perezosas definidas por el usuario en lenguajes ansiosos como funciones, aunque pueden apartarse de la sintaxis del lenguaje para una evaluación ansiosa: a menudo, los cuerpos de código involucrados (como (i) y (j)) deben estar envueltos en un valor de función, de modo que se ejecuten solo cuando se llaman.

La evaluación de cortocircuito de las estructuras de control booleanas a veces se denomina *perezosa*.

3.4 TRABAJAR CON ESTRUCTURAS DE DATOS INFINITAS

Muchos lenguajes ofrecen la noción de estructuras de datos infinitas. Estos permiten que las definiciones de los datos se den en términos de rangos infinitos o recursividad sin fin, pero los valores reales solo se calculan cuando es necesario. Tomemos, por ejemplo, este programa trivial en Haskell:

numberFromInfiniteList :: Int -> Int numberFromInfiniteList n = infinito !! n - l donde infinito = [l ..]main = imprimir $ numberFromInfiniteList 4
En la función numberFromInfiniteList , el valor de infinito es un rango infinito, pero hasta que se necesite un valor real (o más específicamente, un valor específico en un índice determinado), la lista no se evalúa, e incluso entonces solo se evalúa según sea necesario (es decir, hasta que el índice deseado .)

Patrón de lista de éxitos Evitando un esfuerzo excesivo

Una expresión compuesta puede tener el formato EasilyComputed **o** LotsOfWork, de modo que, si la parte fácil da **verdadero, se** podría evitar una gran cantidad de trabajo.

Por ejemplo, suponga que se debe verificar un gran número N para determinar si es un número primo y una función IsPrime (N) está disponible, pero lamentablemente, puede requerir muchos cálculos para evaluar. Quizás N = 2 **o** [Mod (N, 2) ≠ 0 **e** IsPrime (N)] ayudarán si va a haber muchas evaluaciones con valores arbitrarios para N.

3.5 EVITACIÓN DE CONDICIONES DE ERROR

Una expresión compuesta puede tener el formato SafeTo Try **y** Expression, por lo que, si SafeTo Try es **falso,** no se debe intentar evaluar la expresión para que no se señale un error en tiempo de ejecución, como dividir por cero o indexar fuera de los límites, etc. Por ejemplo, el siguiente pseudocódigo ubica el último elemento distinto de cero de una matriz:

L: = Longitud (A); **Mientras que** L> 0 **y** A (L) = 0 **hacen** L: = L - 1;
Si todos los elementos de la matriz son cero, el ciclo funcionará hasta L = 0 y, en este caso, el ciclo debe terminarse sin intentar hacer referencia al elemento cero de la matriz, que no existe.

3.6 OTROS USOS

En los sistemas de ventanas de computadora, la pintura de información en la pantalla es impulsada por eventos de exposición que impulsan el código de pantalla en el último momento posible. Al hacer esto, los sistemas de ventanas evitan el cálculo de actualizaciones de contenido de pantalla innecesarias. [dieciséis]

Otro ejemplo de pereza en los sistemas informáticos modernos es la asignación de páginas de copia en escritura o la paginación por demanda, donde la memoria se asigna solo cuando se cambia un valor almacenado en esa memoria.

La pereza puede ser útil para escenarios de alto rendimiento. Un ejemplo es la función mmap de Unix, que proporciona la carga de páginas desde la disco impulsada por la demanda, de modo que solo las páginas realmente tocadas se cargan en la memoria y no se asigna la memoria innecesaria.

MATLAB implementa copiar al editar, donde las matrices que se copian tienen su almacenamiento de memoria real replicado solo cuando se cambia su contenido, lo que posiblemente conduce a un error de memoria insuficiente al actualizar un elemento posteriormente en lugar de durante la operación de copia.

3.7 IMPLEMENTACIÓN

Algunos lenguajes de programación retrasan la evaluación de expresiones de forma predeterminada, y otros proporcionan funciones o sintaxis especial para retrasar la evaluación. En Miranda y Haskell , la evaluación de los argumentos de función se retrasa de forma predeterminada. En muchos otros idiomas, la evaluación puede ser retrasado mediante la suspensión de forma explícita el cálculo utilizando la sintaxis especial (como ocurre con el esquema de " delay" y " force" y OCaml 's ' lazy' y " Lazy.force") o, más generalmente, envolviendo la expresión en un golpe seco . El objeto que representa una evaluación tan explícitamente retrasada se llama *futuro perezoso* . Raku usa la evaluación perezosa de listas, por lo que uno puede asignar listas infinitas a variables y usarlas como argumentos para funciones, pero a diferencia de Haskell y Miranda, Raku no usa la evaluación perezosa de operadores aritméticos y funciones por defecto.

3.8 PEREZA Y AFÁN CONTROLAR EL ENTUSIASMO EN LENGUAJES PEREZOSOS

En los lenguajes de programación perezosos como Haskell, aunque el valor predeterminado es evaluar las expresiones sólo cuando se exigen, en algunos casos es posible hacer que el código sea más ansioso o, por el contrario, hacerlo más perezoso nuevamente después de que se haya hecho más ansioso. Esto se puede hacer codificando explícitamente algo que fuerce la evaluación (lo que puede hacer que el código sea más ansioso) o evitando dicho código (lo que puede hacer que el código sea más perezoso). *La* evaluación *estricta* suele implicar entusiasmo, pero son conceptos técnicamente diferentes.

Sin embargo, hay una optimización implementada en algunos compiladores llamada análisis de rigurosidad, que, en algunos casos, permite al compilador inferir que siempre se usará un valor. En tales casos, esto puede hacer que la elección del programador de forzar o no ese valor en particular sea irrelevante, porque el análisis de rigor forzará una evaluación estricta.

En Haskell, marcar los campos del constructor como estrictos significa que sus valores siempre se exigirán de inmediato. La seqfunción también se puede usar para exigir un valor inmediatamente y luego pasarlo, lo cual es útil si un campo de constructor generalmente debe ser perezoso. Sin embargo, ninguna de estas técnicas implementa un rigor *recursivo*; para eso, deepSeqse inventó una función llamada.

3.9 SIMULAR LA PEREZA EN IDIOMAS ÁVIDOS

Java

En Java , la evaluación diferida se puede realizar mediante el uso de objetos que tienen un método para evaluarlos cuando se necesita el valor. El cuerpo de este método debe contener el código necesario para realizar esta evaluación. Desde la introducción de expresiones lambda en Java SE8, Java ha admitido una notación compacta para esto. El siguiente ejemplo de interfaz genérica proporciona un marco para la evaluación diferida:

```
interfaz  Lazy < T >  {  T  eval (); }
```

La Lazyinterfaz con su eval()método es equivalente a la Supplierinterfaz con su get()método en la java.util.functionbiblioteca.

Cada clase que implementa la Lazyinterfaz debe proporcionar un evalmétodo, y las instancias de la clase pueden tener los valores que el método necesite para realizar una evaluación perezosa. Por ejemplo, considere el siguiente código para calcular e imprimir perezosamente 2:

Perezoso < Entero > a = () -> 1 ; for (int i = 1 ; i <= 10 ; i ++) { final Lazy < Integer > b = a ; a = () -> b . eval () + b . eval (); } Sistema . fuera . println ("a =" + a . eval ());

En lo anterior, la variable a inicialmente se refiere a un objeto entero diferido creado por la expresión lambda ()->1. Evaluar esta expresión lambda equivale a construir una nueva instancia de una clase anónima que se implementa Lazycon un método eval que regresa 1.

Cada iteración de los enlaces de bucle a a un nuevo objeto creado mediante la evaluación de la expresión lambda dentro del bucle. Cada uno de estos objetos tiene una referencia a otro objeto perezoso, b, y tiene un eval método que llama b.eval()dos veces y devuelve la suma. La variable b es necesario aquí para cumplir con el requisito de Java de que las variables a las que se hace referencia desde dentro de una expresión lambda sean finales.

Este es un programa ineficiente porque esta implementación de enteros perezosos no memoriza el resultado de llamadas anteriores a eval. También implica una considerable cantidad de autoboxing y unboxing. Lo que puede no ser obvio es que, al final del ciclo, el programa ha construido una lista enlazada de 11 objetos y que todas las adiciones reales involucradas en el cálculo del resultado se realizan en respuesta a la llamada a a.eval()en la línea final de código. Esta llamada recorre recursivamente la lista para realizar las adiciones necesarias.

Podemos construir una clase Java que memorice objetos perezosos de la siguiente manera:

class Memo < T > implementa Lazy < T > { privado Lazy < T > lazy ; // una expresión perezosa, eval la establece en nulo private T memo = null ; // el memorando del valor anterior public Memo (Lazy < T > lazy) { // constructor this . perezoso = perezoso ; } public T eval () { if (lazy ! = null) { memo = lazy . eval (); perezoso = nulo ; } nota de devolución ; } }

Esto permite reescribir el ejemplo anterior para que sea mucho más eficiente. Donde el original se ejecutó en tiempo exponencial en el número de iteraciones, la versión memorizada se ejecuta en tiempo lineal:

```
Perezoso < Entero > a = () -> 1 ; for ( int i = 1 ; i <= 10 ; i ++ ) { final Lazy < Integer > b = a ; a = nuevo Memo < Entero > ( () -> b . eval () + b . eval () ); } Sistema . fuera . println ( "a =" + a . eval () );
```

Tenga en cuenta que las expresiones lambda de Java son simplemente azúcar sintáctico. Todo lo que pueda escribir con una expresión lambda se puede reescribir como una llamada para construir una instancia de una clase interna anónima que implemente la interfaz, y cualquier uso de una clase interna anónima se puede reescribir usando una clase interna con nombre, y cualquier clase interna con nombre puede ser movido al nivel de anidación más externo.

JavaScript

En JavaScript, la evaluación diferida se puede simular utilizando un generador. Por ejemplo, el flujo de todos los números de Fibonacci se puede escribir, usando memorización, como:

```
/ ** * Las funciones generadoras devuelven objetos generadores, que reifican la evaluación perezosa. * @return {! Generator } Un generador de enteros no nulo. * / function * fibonacciNumbers () { dejar memo = [ 1n , - 1n ]; // crea el estado inicial (por ejemplo, un vector de números "negafibonacci") while ( verdadero ) { // repite indefinidamente memo = [ memo [ 0 ] + memo [ 1 ], memo [ 0 ]]; // actualiza el estado de cada nota de rendimiento de evaluación [ 0 ]; // producir el siguiente valor y suspender la ejecución hasta que se reanude } } let stream = fibonacciNumbers (); // crea un flujo de números evaluados de forma perezosa let first10 = Array . from ( new Array ( 10 ), () => stream . next (). value ); // evalúa solo los primeros 10 números de la consola . log ( primeros 10 ); // la salida es [0n, 1n, 1n, 2n, 3n, 5n, 8n, 13n, 21n, 34n]
```

Pitón

En Python 2.x, la range()función calcula una lista de enteros. La lista completa se almacena en la memoria cuando se evalúa la primera declaración de asignación, por lo que este es un ejemplo de evaluación ansiosa o inmediata:

>>> r = rango (10) >>> imprimir r [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] >>> imprimir r [3] 3

En Python 3.x, la range()función [23] devuelve un objeto de rango especial que calcula elementos de la lista bajo demanda. Los elementos del objeto de rango solo se generan cuando se necesitan (por ejemplo, cuando print(r[3])se evalúa en el siguiente ejemplo), por lo que este es un ejemplo de evaluación diferida o diferida:

>>> r = rango (10) >>> imprimir (r) rango (0, 10) >>> imprimir (r [3]) 3

Este cambio a la evaluación diferida ahorra tiempo de ejecución para rangos grandes a los que nunca se puede hacer referencia por completo y el uso de memoria para rangos grandes donde solo se necesitan uno o unos pocos elementos en cualquier momento.

En Python 2.x es posible usar una función llamada xrange()que devuelve un objeto que genera los números en el rango a pedido. La ventaja de xrangees que el objeto generado siempre ocupará la misma cantidad de memoria.

>>> r = xrange (10) >>> print (r) xrange (10) >>> lst = [x para x en r] >>> print (lst) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Desde la versión 2.2 en adelante, Python manifiesta una evaluación perezosa mediante la implementación de iteradores (secuencias perezosas) a diferencia de las secuencias de tuplas o listas. Por ejemplo (Python 2):

>>> números = rango (10) >>> iterador = iter (números) >>> imprimir números [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] >>> imprimir iterador < objeto listiterator en 0xf7e8dd4c> >>> iterador de impresión . siguiente ()

El ejemplo anterior muestra que las listas se evalúan cuando se llaman, pero en el caso del iterador, el primer elemento '0' se imprime cuando surge la necesidad.

.NET Framework

En .NET Framework es posible realizar una evaluación diferida usando la clase . La clase se puede explotar fácilmente en F # usando la palabra clave, mientras que el método forzará la evaluación. También hay colecciones especializadas como las que brindan soporte integrado para la evaluación

diferida.System.Lazy<T>lazyforceMicrosoft.FSharp.Collections.Seq

sea fibonacci = Seq . desplegar (diversión (x , y) -> Algunos (x , (y , x + y))) (0l , 1l) fibonacci |> Seq . nth 1000
En C # y VB.NET, la clase se usa directamente.System.Lazy<T>

public int Sum () { int a = 0 ; int b = 0 ; Lazy < int > x = new Lazy < int > (() => a + b); a = 3 ; b = 5 ; devuelve x . Valor ; // devuelve 8 }

O con un ejemplo más práctico:

// cálculo recursivo del n-ésimo número de fibonacci public int Fib (int n) { return (n == 1)? 1 : (n == 2)? 1 : Fib (n - 1) + Fib (n - 2); }public void Main () { Console . WriteLine ("¿Qué número de Fibonacci desea calcular?"); int n = Int32 . Analizar (Consola . ReadLine ()); Lazy < int > fib = new Lazy < int > (() => Fib (n)); // la función está preparada, pero no se ejecuta bool execute ; si (n > 100) { Consola . WriteLine

("Esto puede llevar algo de tiempo. ¿Realmente desea calcular este gran número? [S / n]"); ejecutar = (Consola . ReadLine () == "y"); } más ejecutar = verdadero ; si (ejecutar) Console . WriteLine (FIB . Valor); // el número solo se calcula si es necesario }

Otra forma es usar la yieldpalabra clave:

// evaluación ansiosa pública IEnumerable < int > Fibonacci (int x) { IList < int > fibs = new List < int > (); int prev = - 1 ; int siguiente = 1 ; para (int i = 0 ;

3.10 LA ESTRATEGIA DE EVALUACIÓN PEREZOSA.

Programar en funcional es pensar en funciones. Cualquier algoritmo puede - y debe bajo esta perspectiva - pensarse como un encadenamiento de expresiones funcionales. Sin embargo, la programación funcional no es sólo eso. Sobre todo, y ante todo, cualquier lenguaje de programación que se precie dentro de este paradigma se caracteriza por un comportamiento específico acerca de cómo evalúa estas construcciones. Entender bien este hecho es muy importante de cara a controlar el paradigma ya que como desarrolladores es importante saber qué espera nuestro lenguaje de nosotros y de nuestro código.

La Programación Funcional y El Tiempo

1. Optimización Por Recursión En La Cola
2. Las 3 Evaluaciones de la Programación Funcional
3. Estructuras Recursivamente Infinitas

Evaluación Impaciente vs Evaluación Perezosa

Comencemos por el más sencillo de los 3 tipos de evaluación, aquel que prescribe el momento exacto del tiempo en el que se deben evaluar las expresiones que aparecen en el código fuente. Encontrar una definición formal que permita diferenciar los dos modelos que se distinguen dentro de esta categoría es sencillo.

La **Evaluación impaciente** evalúa cada expresión en el momento exacto del tiempo en que ésta es encontrada dentro del código fuente. La **evaluación perezosa**, por el contrario, posterga la evaluación de la expresión hasta que su valor es realmente demandado por el programa en ejecución.

El mejor ejemplo para ilustrar esta diferencia lo encontramos en las expresiones lógicas. Imaginemos una construcción de la forma x && y, que como sabemos corresponde a la conjunción lógica de las sub-expresiones en x e y. El modelo de evaluación impaciente evaluaría vorazmente las partes izquierda y derecha de la expresión y sólo cuando se obtuviera el valor de cada una de ellas se computaría el resultado final. La evaluación perezosa actúa de una manera más inteligente aplicando una estrategia de evaluación que, como hemos descrito, depende de la semántica de la operación posterior. En este ejemplo, primero se computa el valor en x y sólo si éste es cierto el compilador da continuidad a la evaluación de la sub-expresión en y dado que se sabe que si x fuera falso la operación de conjunción lógica arrojaría un valor lógico falso, con independencia del valor de la y. Un razonamiento similar puede encontrarse para el resto de operadores lógicos.

```
// A. Codificación

x > 0 && f (x)              // if (x>0) f(x)
xs && ys && xs.push(ys)     // if (xs && ys) xs.push (ys)
x && x.y && (x.y.z=1)       // if (x && x.y) x.y.z = 1
xs = [...(ys || []), ...zs] // if (ys) xs = [...ys, ...zs] else xs=[...zs]
(fn || x => x)(3)           // if (fn) fn(3) else 3

// B. Recursión

function search (xs, x) {
  return (function has (p) {
    return (
      p < xs.length && (
        xs[p] === x ||
        has (p+1)
      )
    )
  })(0);
}

// C. Optimización

function search (xs, x) {
  return (function has (p, q) {
    let m = Math.floor ((p + q) / 2)
    return p === q ? xs[p] === x :
      has (p, m) ||
      has (m+1, q)
  })(0, xs.length-1);
}
```

Listado 1. Ejemplos de Evaluación Perezosa.

- **Codificación**. La evaluación perezosa permite alcanzar un modelo de programación defensivo basado en aserciones que resulta generalmente más sinóptico y sencillo de leer y entender. En lugar de comprobar las precondiciones de una operación a base de un abuso de sentencias y expresiones condicionales se utilizan expresiones lógicas como prefijo de las operaciones a realizar para que operen de guarda que dispara la ejecución de la operación a la derecha sólo si se satisface cierta condición ambiental. En el código (A) del Listado 1 pueden verse algunos ejemplos de este uso canónico de construcciones.
- **Recursión**. De forma similar al caso anterior, los esquemas de diseño funcional recursivo también se prestan mucho a hacer uso de este tipo de construcciones. En el código (B) del Listado 1 aparece un ejemplo de función de búsqueda recursiva que determina si un elemento x se encuentra dentro de un vector xs. Tal función

consiste en una sola expresión que indica que el vector contendrá al elemento, si éste se encuentra en la posición de análisis en curso p o está en alguna posición posterior a este índice, comenzando desde el 0. Y todo ello siempre que p no supere la longitud del vector. En este caso, la evaluación perezosa nos servirá para que se pare la recursión cuando esta última condición sobre p deje de ser cierta o el elemento se haya encontrado.

- **Rendimiento**. La evaluación perezosa también puede ser relevante en temas de rendimiento. El código (C) del Listado 1 muestra una función similar a la anterior, pero en este caso aplicando búsqueda binaria. Básicamente, en cada iteración recursiva, el vector se parte - de forma imaginaria - en 2 subvectores iguales con ayuda de dos índices posicionales p y q. Buscar el elemento x consiste en recurrir con la búsqueda en cada subvector. El proceso terminará cuando se llegue a alguna situación unitaria donde py q coincidan y pueda comprobarse que en ese punto reside el elemento buscado. La ventaja de la evaluación perezosa en este escenario consiste en que todo el proceso de encadenamiento compositivo por operaciones de disyunción lógica se para automáticamente en cuanto se encuentra la primera coincidencia ahorrando muchos ciclos de cómputo. Recuerde que el modelo de ejecución es secuencial y que la operación has no se lanzan en paralelo.

De todo lo anterior podemos colegir que el modelo de evaluación perezosa resulta bastante más conveniente que su contrapartida impaciente. No obstante, debemos ser conscientes de que en este modelo el orden de escritura de las sub-expresiones es relevante. Defender ante la matemática que computacionalmente no es equivalente la construcción x && y a su simétrica y && x es algo que, cuando menos, resulta comprometido.

La buena noticia es que este tipo de evaluación perezosa - la que opera sobre el espacio de expresiones lógicas, también llamada **evaluación en cortocircuito** - está en la mayoría de lenguajes actuales. Es el caso de nuestro lenguaje JavaScript. No obstante, hay otros escenarios donde resultaría importante disponer de evaluación perezosa y que no quedan cubiertos por JavaScript. Me estoy refiriendo a la evaluación en las llamadas a funciones.

Cuando evaluamos una función en JavaScript, todas las expresiones que se corresponden con sus argumentos actuales se evalúan primero y sólo después se invoca la función con dichos valores resueltos. Es decir, en este caso JavaScript aplica una estrategia de evaluación impaciente. Esta estrategia, conocida como **evaluación en modo estricto**, contrasta con aquélla que se encuentra en los lenguajes de programación funcional más puros en los que la evaluación de los argumentos se posterga hasta que su valor es necesario. En este contexto, este modo de evaluación perezosa recibe el nombre de **evaluación no estricta**.

```
function Ones () {
  return [1, ...Ones()]
}
let ones = Ones ()
```

Listado 2. Error de Evaluación Estricta.

Evaluación Completa vs Evaluación Parcial

La segunda categoría de evaluación refiere a la forma en que las funciones pueden ser aplicadas cuando se demanda su ejecución. En este sentido, también existen 2 modelos de evaluación contrapuestos.

La **evaluación completa**, requiere disponer de todos los argumentos actuales antes de invocar a una función. La **evaluación parcial**, por el contrario, permite proporcionar sólo parte de los argumentos para obtener otra función con aridad reducida.

El modelo de evaluación completa es característico de los paradigmas centrados en procesos, típicamente la programación estructurada y sus derivados. En este tipo de enfoques una función se interpreta de forma similar a un procedimiento que arroja un resultado cuando es evaluado contra unos argumentos determinados.

El modelo de evaluación parcial, sin embargo - que naturalmente puede entenderse como una generalización del anterior - permite ir reduciendo progresivamente la aridad de la función proporcionando los argumentos en el orden requerido por ésta a lo largo del

tiempo y sólo cuando todos los argumentos se han proporcionado se alcanza el resultado final.

Lamentablemente, JavaScript no ofrece mecanismos de evaluación parcial de forma nativa. Sin embargo, aplicando las capacidades de orden superior de nuestro lenguaje podemos diseñar las funciones para que reciban los parámetros de acuerdo a un esquema de **diseño por fases**. En efecto, como puede verse en el código (A) del Listado 3, una sencilla función de suma puede reformularse en términos tales que requiera un esquema de invocación parcial. Esto es, recibiendo en 2 etapas de evaluación primero la x y después la y.

```
function add (x) {
  return function (y) {
    return x + y
  }
}
let inc = add (1)
let dec = add (-1)

// B. Currificación

function curry (fn) {
  return function aux (...args) {
    return args.length >= fn.length ?
      fn (...args) :
      function (...rest) {
        return aux (...args, ...rest)
      }
  }
}

let add = curry (function (x, y) { return x + y })
let mul = curry (function (x, y) { return x * y })

let inc    = add (1)
let dec    = add (-1)
let double = mul (2)
let half   = mul (1/2)
```

Listado 3. Diseño por Fases & Currificación.

El gran inconveniente de esta solución por rediseño es precisamente ese, que todas las funciones que queramos utilizar en evaluación parcial deberían reescribirse de acuerdo a una estrategia de diseño por fases. Esto complica la escritura de funciones y genera problemas de versionado más aún cuando las diferentes formas de uso en que nos puede interesar proporcionar los parámetros actuales de una función de forma progresiva conducen a una explosión combinatoria.

Para aliviar este problema podemos crear una función de orden superior que realice este proceso de transformación de forma automática para cualquier función y número de parámetros formales. El código (B) del Listado 3 muestra una función curry que se encarga precisamente de hacer esto. Parece un esquema prolijo, pero en realidad es muy sencillo. En esencia, estamos devolviendo otra función aux para solicitar los primeros argumentos

actuales. Si éstos resultan suficientes para invocar a la función original fnprocederemos con su invocación. Si no, devolveremos otra nueva función que solicite más parámetros e invocaremos recursivamente a aux con la concatenación de toda la información paramétrica recibida. Un esquema similar, recogiendo los parámetros desde la derecha también resulta de interés. Aquí lo dejamos como ejercicio al ávido lector. A esta técnica se la llama currificación y no debe confundirse con la evaluación parcial.

Esta estrategia de transformación es interesante puesto que convierte a las funciones ordinarias en factorías de otras funciones más específicas con un subconjunto original de los parámetros formales ya resueltos y cuyo valor queda retenido léxicamente. Cada vez que aplicamos evaluación parcial convertimos en estado inmutable parte de los parámetros y dejamos el resto como grados de libertad para el usuario. En el Listado 3 pudo verse cómo sendas estrategias de resolución - el diseño por fases y la currificación - permitían generar funciones cuya semántica se expresaba en términos de un sólo parámetro. Desde la suma add se generaron inc y dec y desde el producto mul, se obtuvieron double y half.

- **Evaluación por Fases**. Con independencia de que se desarrolle un diseño por fases o por currificación, el primer uso de la evaluación parcial consiste en resolver las dimensiones paramétricas de una función en distintas fases del tiempo. El código (A) del Listado 4 presenta un sencillo ejemplo que tiene por objeto construir distintos tipos de direcciones de red. Atendiendo a lo que estamos explicando, este problema podría atacarse en dos etapas sucesivas. Primero se crean funciones específicas para crear direcciones de red de tipo A, B y C. Después, podrán usarse esas funciones, en una segunda fase de evaluación para obtener direcciones específicas de estos tipos por medio de un ejercicio consistente en especificar el resto de la información paramétrica.
- **Evaluación por Pares**. Otro uso prototípico de la evaluación parcial es la evaluación por pares. En este caso, existen 2 agentes que articulan una colaboración con roles diferentes. El agente cliente pide una función al agente proveedor y éste la resuelve de forma parcial antes de devolverla. De esta manera se fija parcialmente el comportamiento de dicha función para adaptarlo a las condiciones ambientales

requeridas por el cliente. El código (B) del Listado 4 muestra un ejemplo de este escenario sobre funciones de traza.

Evaluación Inmediata vs Evaluación Diferida

El último tipo de evaluación refiere al momento exacto del tiempo en que una función se ejecuta tras su invocación. Nuevamente, en esta dimensión del discurso, distinguimos dos modelos contrapuestos.

La **Evaluación Inmediata** lanza a ejecución una función en el mismo momento en que se procesa su invocación. La **Evaluación Diferida**, por el contrario, permite trabajar con funciones cuya invocación puede postergarse en el tiempo hasta un momento más conveniente.

En sentido estricto, esta diferenciación no responde a una característica de los lenguajes de programación. Diferir la ejecución de una función es más bien un recurso algorítmico dentro del estilo de diseño funcional. No obstante, no es menos cierto que para que esta práctica pueda articularse el lenguaje debe proporcionar las herramientas necesarias. Aunque existen varias formas de dar soporte a esta idea todas giran en torno a las capacidades de orden superior.

Listado 5. Evaluación Diferida

```
function defer (fn) {
  return function (...args) {
    return function () {
      return fn (...args)
    }
  }
}

let add  = function (x, y) { console.log (x + y) }
let mul  = function (x, y) { console.log (x * y) }
let dAdd = defer (add)
let dMul = defer (mul)
let five = dAdd (2, 3)
let six  = dMul (2, 3)
```

- **Comandos**. Las técnicas de diferimiento se utilizan a menudo para generar comandos que se ejecutan en el momento adecuado. Un comando es una operación que al ejecutarse transforma el estado de la arquitectura. Dado que la ejecución de tal operación debe postergarse hasta el momento en que sea explícitamente

requerido, los comandos difieren la ejecución. El ejemplo por antonomasia de este tipo de soluciones se encuentra en los sistemas de UI. Sin embargo, como ejemplo sencillo nosotros hemos implementado en el código (A) del Listado 6 una abstracción Stack que incorpora la operación undopara deshacer operaciones anteriores. Quizá este código está un poco traído por los pelos para que sirva de ejemplo a nuestros propósitos. Pero lo cierto es que en lugar de implementar la operación de deshacer trabajando sobre el estado de la pila nosotros hemos seguido una aproximación basada en comandos. Primero definimos 2 factorías de comandos que revierten las operaciones de la abstracción: cPush deshace el push y cPop deshace el pop. Después, definimos las operaciones clásicas push y pop. Pero en este caso, ambas, además de transformar el estado interno en state, almacenan el comando asociado en una pila de comandos cmds. En estos términos, la operación undo puede formularse como extraer el último comando de la estructura cmds e invocarlo.

- **Thunks**. Si los comandos representan operaciones diferidas que alteran el estado de una arquitectura, los thunks son funciones diferidas que devuelven un valor de interés al invocarlas. Este tipo de artefactos es clásico de las arquitecturas de integración. Un thunk sirve de vehículo para comunicar información entre un origen y un destino. El último ejemplo más sonado que hemos visto dentro de la comunidad de JS es el uso de thunks dentro de la programación asíncrona resuelta con generadores. No está en nuestro ánimo explicar aquí todo el proceso ejemplificado en el código (B) del Listado 6, ya que cae fuera de nuestro alcance. Pero baste comprobar cómo en ese código, la función Async se encarga de transformar funciones bloqueantes en no bloqueantes que arrojan el resultado cuando son invocadas proporcionando un manejador. El contexto de la solución de asincronía con generadores ilustra que es una función Go la encargada de proporcionar este manejador cuya lógica es meramente la de dar continuidad al algoritmo ocultando dentro del código de la función generadora la perdida de secuencialidad. Pero insisto, esto es ya asunto de otro post sobre asincronía.

- **Breakpoints**. Finalmente, merece la pena destacar el uso de técnicas de diferimiento funcional para establecer puntos de parada dentro del código. En general cualquier lógica que se encapsule dentro de una función quedará automáticamente retenida hasta que dicha función se invoque explícitamente. Este uso pragmático del diferimiento lo encontramos, entre otros casos, en la técnica de trampolín, que sirve para resolver el problema del desbordamiento de pila en grandes recursiones. Como explicamos en el artículo anterior, después de convertir las funciones recursivas no finales en recursivas finales, esperábamos que un buen compilador aplicara, de forma automática y transparente, técnicas de optimización por recursión en la cola. Pues bien, hasta que ese sea el caso de JS nosotros podemos hacer uso de una función de trampolín que reaproveche el espacio de memoria. En el código (C) del Listado 6 puede verse como la función factorial fno solamente se reformula en términos de un esquema de recursivida final, sino que además encapsula el caso recursivo para poder diferir su ejecución. Al pasar esta función a la función trampoline obtenemos una función equivalente tf que arroja los mismos resultados que f. La función trampolineopera como un entorno de ejecución que reutiliza la variable local rpara la obtención de resultados. Todo este reaprovechamiento funciona en base a la hipótesis de que f es una función, en efecto, de recursividad final. En esencia, la función trampoline invoca la función que se devuelve como resultado (caso recursivo diferido) hasta que se obtiene un valor real (caso base final).

La aplicación de las técnicas de diferimiento funcional que acabamos de describir ayudaría a resolver los inconvenientes de ejecución que encontramos en la evaluación del código en el Listado 2. No obstante, dado que esa no es la única solución posible y que nos estamos quedando sin espacio, abordaremos este tema en el próximo artículo para cerrar esta serie.

En fin, que evaluación perezosa, parcial y diferida son el incienso, oro y mira de la programación funcional. Si quieres saber cuan bueno es un lenguaje en relación al soporte que ofrece a la programación funcional mírale las cosquillas en estos tres puntos.

3.11 TÉCNICAS DE PROGRAMACIÓN FUNCIONAL PEREZOSA.

Los beneficios de la evaluación perezosa son:

- El incremento en el rendimiento al evitar cálculos innecesarios, y en tratar condiciones de error al evaluar expresiones compuestas.
- La capacidad de construir estructuras de datos potencialmente infinitas.
- La capacidad de definir estructuras de control como abstracciones, en lugar de operaciones primitivas.

La evaluación perezosa puede también reducir el consumo de memoria de una aplicación, ya que los valores se crean solo cuando se necesitan. Sin embargo, es difícil de combinar con las operaciones típicas de programación imperativa, como el manejo de excepciones o las operaciones de entrada/salida, porque el orden de las operaciones puede quedar indeterminado. Además, la evaluación perezosa puede conducir a fragmentar la memoria. Lo contrario de la evaluación perezosa sería la evaluación acaparadora, o evaluación estricta, que es el modo de evaluación por defecto en la mayoría de los lenguajes de programación.

Técnica de Backtracking

Si una alternativa falla, el flujo retrocede hasta la última posición o intenta de nuevo.

¿Qué es backtracking?

Cuando un problema no tiene un método algorítmico para resolverse, en general la única forma posible de resolverlo es la búsqueda exhaustiva de soluciones entre todas las posibilidades del problema; este método se conoce como fuerza bruta: se generan todos los casos posibles y se testean uno a uno hasta encontrar las soluciones necesarias (a veces basta con encontrar una, en otras ocasiones hay que encontrar todas ellas, o quedarse con la mejor).

Sin embargo, en muchos de estos problemas no es necesario crear completamente un caso para ver si es una solución o no. Cuando resolver un problema puede hacerse por etapas se puede comprobar paso a paso si se está creando una solución o si se han tomado

decisiones que no conseguirán resolver el problema: en cada etapa se estudian las propiedades cuya validez ha de examinarse con objeto de seleccionar las adecuadas para proseguir con el siguiente paso.

La gestión de las etapas, las posibles decisiones que se toman y las relaciones entre ellas, suponen la generación de un árbol de decisiones. En los nodos se encuentran las soluciones parciales, y los enlaces entre ellos son las decisiones tomadas para cambiar de etapa. El avance a lo largo del árbol se detiene cuando se llega a una situación en que no puede tomarse ninguna decisión que permita obtener una solución, o cuando efectivamente se llega a resolver el problema. En estos casos se recupera una solución parcial anterior y se continúa buscando si es necesario.

Backtracking (o Vuelta Atrás) es una técnica de recursión intensiva para resolver problemas por etapas, que utiliza como árbol de decisiones la propia organización de la recursión. Cuando se "avanza" de etapa se realiza una llamada recursiva, y cuando se "retrocede" lo que se hace es terminar el correspondiente proceso recursivo, con lo que efectivamente se vuelve al estado anterior por la pila de entornos creada en la recursión. Como el árbol de decisiones no es una estructura almacenada en memoria, se dice que Backtracking utiliza un árbol implícito y que se habitualmente se denomina árbol de expansión o espacio de búsqueda.

Básicamente, Backtracking es un método recursivo para buscar de soluciones por etapas de manera exhaustiva, usando prueba y error para obtener la solución completa del problema añadiendo decisiones a las soluciones parciales.

¿Cómo funciona backtracking?

En su forma básica Backtracking se asemeja a un recorrido en profundidad del árbol de expansión; se recorre en preorden: primero se evalúa el nodo raíz o actual, y después todos sus hijos de izquierda a derecha. Por tanto, hasta que no se termina de generar una solución parcial (sea válida o no) no se evalúa otra solución distinta.

En los nodos del nivel k del árbol se encuentran las soluciones parciales formadas por k etapas o decisiones. Hay que recordar que el árbol no está almacenado realmente en memoria, solo se recorre a la vez que se genera, por lo que todo lo que quiera conservarse (decisiones tomadas, soluciones ya encontradas al problema, etc.) debe ser guardado en alguna estructura adicional.

Finalmente, debido a que la cantidad de decisiones a tomar y evaluar puede ser muy elevada, si es posible es conveniente tener una función que determine si a partir de un nodo se puede llegar a una solución completa, de manera que utilizando esta función se puede evitar el recorrido de algunos nodos y por tanto reducir el tiempo de ejecución.

Costes y eficiencia en backtracking

La forma de generar y escoger entre las distintas posibilidades determina la forma del árbol, la cantidad de descendientes de un nodo, la profundidad del árbol, etc., y determina la cantidad de nodos del árbol y por tanto posiblemente la eficiencia del algoritmo, pues el tiempo de ejecución depende en gran medida del número de nodos que se generen y se visiten. Habitualmente, tendremos como máximo tantos niveles como valores tenga una secuencia solución

Suponiendo que hay n etapas y m posibles decisiones en cada una de ellas (normalmente m es mayor que n) el orden de eficiencia de Backtracking está en O(mn); aun cuando se pueda tener m = n se consigue el orden potencial O(nn).

En general, la eficiencia depende de:

- el tiempo necesario para generar las decisiones posibles en el punto (2)
- la cantidad de decisiones posibles que se obtienen en (2)
- incorporar una decisión a una solución parcial en el punto (3)
- el número de soluciones parciales que satisfacen es_completable

• y en mucha menor medida, del tiempo que se tarde en comprobar es solución y tratar solución.

Técnica de Guardias

Si más de una es cierta, se escoge cualquiera de ellas.

Técnica de Aprendizaje Reforzado

Recordar decisiones exitosas y aumentar su prioridad, así como considerar las decisiones en el contexto del estado mutable.

Métodos

Variante de flujo regular o flow shop scheduling

Dentro de la teoría de scheduling se pueden distinguir un gran número de problemas. En estos se tiene un conjunto de ***N*** trabajos que han de ser procesados sobre un conjunto de ***M*** recursos o máquinas físicas siguiendo un patrón de flujo o ruta tecnológica. Una secuenciación consiste en encontrar para cada trabajo un tiempo o un intervalo de tiempos en los que este pueda procesarse en una o varias máquinas [4, 27]. El objetivo es encontrar una secuencia sujeta a una serie de restricciones que optimice una o varias funciones objetivo [20, 24, 27].

En general, en un problema de *scheduling* intervienen los siguientes elementos: trabajos, actividades, máquinas, tipo de *scheduling* (patrón de flujo) y objetivos. El problema que se analiza en este artículo está sujeto a las siguientes restricciones:

- Solo se cuenta con una máquina-herramienta de cada tipo por etapa.
- Las restricciones tecnológicas están bien definidas y son previamente conocidas, además de que son inviolables.

- No está permitido que dos operaciones del mismo trabajo se procesen simultáneamente.
- Un trabajo puede procesarse o no en una o varias etapas.
- Ningún trabajo puede ser procesado más de una vez en la misma máquina.
- Cada trabajo es procesado hasta concluirse, una vez que se inicia una operación esta se interrumpe solamente cuando se concluye.
- Ninguna máquina puede procesar más de un trabajo a la vez.
- Los tiempos de configuración se conocen de antemano.
- Existe precedencia entre trabajos.
- El tiempo de transportación entre etapas no es considerado.

Como se mencionó anteriormente, el objetivo es encontrar una secuencia de trabajos por etapas bajo la restricción de que el procesamiento de cada trabajo tiene que ser continuo con respecto al objetivo de minimizar el *makespano* C_{max} como también se le conoce.

Bajo estas condiciones, el tiempo de procesamiento total corresponde al tiempo de culminación de procesamiento del último trabajo. En otras palabras, es el tiempo necesario para completar todos los trabajos.

Aprendizaje reforzado

El AR es un enfoque de la IA en el que los agentes aprenden a partir de su interacción con el ambiente. Es aprender qué acción tomar dada una situación determinada con el objetivo de maximizar una señal numérica de recompensa que da la medida de cuán buena fue la acción elegida por el agente.

En el paradigma del AR un agente se conecta a su ambiente mediante la percepción y acción, como lo descrito en la Figura 1. En cada paso de la interacción, el agente percibe el estado actual "s" de su ambiente y selecciona una acción "a" para cambiar este estado. Esta transición genera una señal de refuerzo "r", que es recibida por el agente. La tarea del agente es aprender una política de elección de acciones en cada estado, que le posibilite

recibir un número máximo de recompensas acumuladas. Los métodos del aprendizaje reforzado exploran el ambiente todo el tiempo para obtener una política deseada.

Fig. 1. Modelo estándar de Aprendizaje Reforzado

De manera formal, el modelo básico de AR consiste en:

- Un conjunto de estados del ambiente *S*.
- Un conjunto de acciones *A*.
- Un conjunto de "recompensas" en $\mathbb{R}$.
- Una función de transición *T*.

En cada instante *t,* el agente observa el estado $s_t \in S$ y el conjunto de posibles acciones $A(s_t)$. Escoge una acción $a \in A(s_t)$ y recibe del ambiente el nuevo estado s_{t+1} y una recompensa r_{t+1}, esto significa que el agente implementa una asociación entre estados y probabilidades de seleccionar cada posible acción. Esta asociación es la política del agente y se denota π_t, donde $\pi_t(s,a)$ es la probabilidad de seleccionar la acción *a* en el estado *s* en el instante *t*.

La función de recompensa define el objetivo en un problema de AR. Asocia cada estado observado (o par estado-acción) del ambiente con un valor numérico, indicando la deseabilidad de esa acción en ese estado. El objetivo de un agente usando Aprendizaje Reforzado es maximizar la cantidad total de recompensa que recibe a largo plazo. La función de recompensa define cuáles son los eventos buenos y malos para el agente.

De la misma forma en que la función de recompensa indica lo que es "inmediatamente" bueno, la función de valor especifica lo que es bueno a largo plazo. En otras palabras, el valor de un estado es la cantidad total de recompensa descontada que un agente puede

esperar acumular en el futuro, comenzando en ese estado. Por ejemplo, un estado puede siempre conducir a bajas recompensas inmediatas, pero aún tener un alto valor porque es normalmente seguido por otros estados que conducen a recompensas altas.

Adaptación del algoritmo q-learning para la solución del FSSP

QL es un algoritmo del AR que se basa en la capacidad de aprendizaje de los agentes asociados al método. Este algoritmo trabaja aprendiendo de una función acción-valor que da la utilidad esperada de tomar una acción en un estado determinado. El núcleo del algoritmo es la actualización de un q-valor en cada iteración. Cada par estado-acción (*s,a*) tiene un q-valor asociado. Cuando la acción es seleccionada por el agente que se encuentra en un estado determinado, el q-valor para ese par estado-acción es actualizado en base a la recompensa recibida cuando se seleccionó esa acción y el mejor q-valor para el subsiguiente estado. La regla de actualización para cada par estado-acción es la siguiente:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\,[r + \gamma\, max_{a'}\, Q(s',a') - Q(s,a))] \quad (1)$$

En esta expresión $\alpha \hat{I}$ [0, 1] es la proporción de aprendizaje y *r* la recompensa o penalidad resultante de tomar una acción *a* en el estado *s*. La proporción de aprendizaje determina el grado por el cual el viejo valor es actualizado. Por ejemplo, si la proporción de aprendizaje es α = 0, entonces no habrá actualización. Por otro lado, si α = 1, el viejo valor es remplazado por el nuevo estimado. Usualmente es utilizado un valor pequeño, por ejemploα = 0.1. El factor de descuento (parámetro γ) tiene el rango de valores desde cero hasta uno. Si γ es cercano a cero el agente tiende a considerar solamente la recompensa inmediata. Si γ es cercana a uno el agente considerará la recompensa futura en mayor medida. En el algoritmo QL interviene además otro parámetro que permite el balance entre la exploración y explotación denotado por ε (épsilon). Todas las acciones a realizar tienen asociadas una probabilidad generada aleatoriamente. Si esta probabilidad está por debajo de ε se selecciona una acción aleatoria, de lo contrario se selecciona una acción de acuerdo con la política del agente.

El algoritmo 1 es usado por los agentes para aprender a partir de la experiencia y el entrenamiento. Cada episodio es equivalente a una sesión de entrenamiento. En cada uno de ellos el agente explora el ambiente y toma la recompensa hasta alcanzar el estado objetivo. El propósito del entrenamiento es aumentar el conocimiento del agente representado por los q-valores. Entre mayor sea la cantidad de episodios se alcanzarán mejores resultados.

Los principales elementos que intervienen en el desempeño del método de solución propuesto son detallados a continuación:

Estados y acciones: Existe un agente asociado a cada etapa (en nuestro caso se asocia un agente por máquina pues solo existe una por etapa) y este agente tomará decisiones sobre futuras acciones. Para un agente el tomar una acción significa que debe decidir cuál va a ser el próximo trabajo a procesar del conjunto de trabajos posibles (estos son los que están esperando en la cola asociada al recurso correspondiente teniendo siempre en cuenta la precedencia entre ellos). El agente además de tener una visión local con la información asociada a su recurso (como los trabajos que están esperando por él), tiene conocimiento acerca de las colas de espera de los demás agentes así como la información asociada a estos.

Q-Valores: una matriz de ***m*** filas y ***n*** columnas es construida para almacenar los q-valores ya que para cada una de las ***m*** máquinas existen, al menos, ***n*** posibles trabajos a procesar.

Resultados

Al no existir instancias disponibles de problemas que contemplen las restricciones presentes en este artículo, los datos para los experimentos computacionales fueron generados aleatoriamente. Para comprobar la calidad de los resultados obtenidos se usaron 10 instancias de diferentes tamaños y complejidad que sirvieron para ejecutar el algoritmo propuesto. El tamaño de las instancias fue de 5x3, 5x4, 5x5, 7x6, 7x7, 8x8, 9x4, 9x9, 10x8 y 10x10 respectivamente. Se generaron números aleatorios para crear los tiempos iniciales de preparación de las máquinas y los tiempos de configuración entre trabajos. La tabla

1 muestra los tiempos de procesamiento de cada trabajo por máquina. La tabla 2, por su parte, detalla los tiempos de preparación inicial de las máquinas. La tabla 3muestra los tiempos de configuración entre trabajos en cada una de las máquinas. Por último, la tabla 4 muestra la precedencia entre trabajos. Todos los datos mostrados en las siguientes tablas corresponden a la instancia 5x5.

Tabla 1. Tiempo de procesamiento

	Tiempo de Procesamiento				
Máquina	J_0	J_1	J_2	J_3	J_4
M_0	10	6	11	0	11
M_1	15	9	14	10	0
M_2	12	11	9	10	6
M_3	8	4	8	9	12
M_4	6	6	8	6	3

Tabla 2. Tiempo de preparación de las máquinas

Máquina	Tiempo de Preparación
M_0	9
M_1	3
M_2	8
M_3	16
M_4	23

Tabla 4. Precedencia entre trabajos

Trabajo	Precedencia	
J_0	-1	
J_1	3	4
J_2	1	
J_3	4	
J_4	-1	

Inicialmente se realizaron numerosos experimentos para analizar el proceso de aprendizaje utilizando típicas combinaciones de valores para los parámetros que intervienen en el algoritmo [25]. Las combinaciones propuestas para el algoritmo QL son las siguientes:

- Combinación 1: episodios = $n*m$, $\alpha = 0.1$, $\gamma = 0.8$, $\varepsilon = 0.2$
- Combinación 2: episodios = $n*m$, $\alpha = 0.1$, $\gamma = 0.9$, $\varepsilon = 0.1$
- Combinación 3: episodios = $n*m$, $\alpha = 0.1$, $\gamma = 0.8$, $\varepsilon = 0.1$

- Combinación 4: episodios = $n*m$, $\alpha = 0.1$, $\gamma = 0.9$, $\varepsilon = 0.2$

  El comportamiento no determinístico de los algoritmos sobre múltiples conjuntos de datos es una razón por la cual no existe un procedimiento establecido para poder compararlos. Distintas técnicas estadísticas son utilizadas para tratar de determinar si las diferencias encontradas entre dos algoritmos son significativas. Debido a la no disponibilidad de problemas que contemplen las restricciones presentadas en el problema que se analiza en este artículo solo se realizó una comparación entre los rendimientos de cada combinación para las instancias propuestas y así determinar cuál de ellas tiene mejor desempeño. Para esto se tuvo en cuenta la moda de C_{max}obtenida después de 10 ejecuciones para 4 instancias escogidas aleatoriamente. La tabla 5 muestra estos resultados.

Luego de los experimentos realizados se decidió mantener la Combinación 3 para todas las instancias de problemas teniendo en cuenta la calidad de las soluciones obtenidas.

La figura 2 muestra el proceso de aprendizaje en la solución de la instancia 5x5 y 10x8 con la combinación de parámetros escogida. Las soluciones para dichas instancias son C_{max} = 172 y C_{max} = 241 respectivamente.

El algoritmo QL fue implementado en Java, se ejecutó en una PC con un Corei3 a 3.4 GHz y 2 GB de memoria RAM. La Figura 3 muestra la solución de la instancia 5x5 donde el C_{max}= 172 a través un diagrama de Gantt. La Tabla 6muestra los valores de C_{max}para las 10 instancias.

Tabla 6. Resultados del algoritmo Q-Learning para las intancias de problemas propuestas.

Instancia	C_{max}	Instancia	C_{max}
5x3	106	8x8	343
5x4	116	9x4	209
5x5	172	9x9	460
7x6	195	10x8	412
7x7	183	10x10	261

UNIDAD 4

FUNDAMENTOS DE LA PROGRAMACIÓN LÓGICA

Los primeros sistemas lógicos se remontan a Aristóteles, sin embargo, la lógica de primer orden, tal y como la conocemos, fue originalmente propuesta por Gottlob Frege (Conceptografía) en la segunda mitad del siglo XIX. Posteriormente publicaría un trabajo titulado Los fundamentos de la aritmética en el cual refinaría sus ideas.

No obstante, no fue hasta que Giuseppe Peano y Bertrand Russell modificaron la notación de Frege para que las ideas de este alcanzaran una mayor audiencia.

En la década de los años treinta, Kurt Göedel y Jacques Herbrand estudiaron la noción de lo computable basada en derivaciones. Su trabajo puede verse como el origen de la computación como deducción.

Origen

Además, Herbrand discutió un conjunto de reglas para manipular ecuaciones algebraicas en términos que pueden verse ahora como un bosquejo de la unificación.

Treinta años más tarde, Robinson publicó su artículo sobre la demostración automática. En este trabajo se introduce el principio de resolución, la noción de unificación y un algoritmo para su cómputo.

Durante los años siguientes la lógica de primer orden ha recibido aportes como demostraciones y teoremas de personajes importantes como Göedel y Turing, lo cual le ha permitido establecerse como un sistema formal sólido dentro de la metalógica.

4.1 Repaso de la lógica de primer orden.

Un predicado es lo que se afirma de un sujeto.

En lógica, también se conoce como predicado a las relaciones entre sujetos.

La Tierra es un planeta
(expresión) (predicado)

Júpiter posee más masa que la Tierra
(expresión) (relación) (expresión)

Objetivo principal

Describir los objetos que conforman un universo de discurso, así como las relaciones que existen entre ellos con el para obtener conclusiones nuevas a partir de lo anterior.

Para entender cómo se logra, es necesario analizar la lógica de primer orden como un sistema formal considerando:

Sintaxis: definir cuáles expresiones pertenecen a este conjunto.

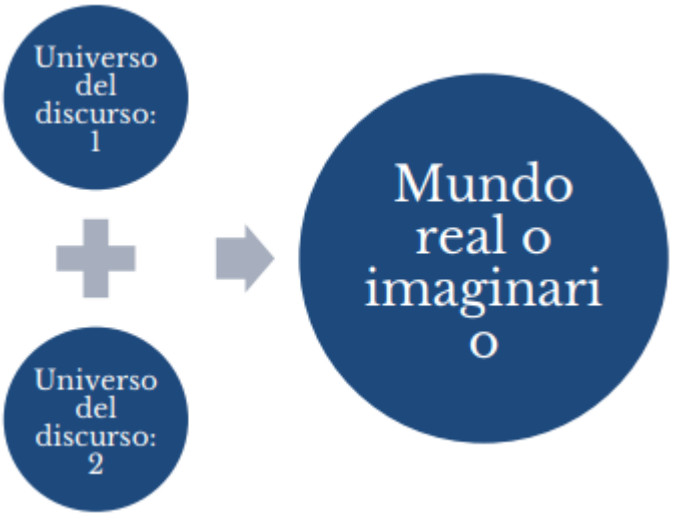
Semántica: determinar qué hace verdadera o falsa una cláusula

Reglas de razonamiento: obtener nuevas conjeturas.

Lenguaje

El lenguaje de la lógica de primer orden está representado por un conjunto de símbolos que nos permite expresarnos sobre los objetos de un determinado dominio.

El conjunto de todos esos objetos se conoce como universo de discurso. Un objeto es algo sobre lo cual queremos expresarnos sin importar su naturaleza.



Funciones

Una función es un tipo especial de relación entre los objetos del dominio de discurso que mapea un conjunto de objetos de entrada a un objeto único de salida.

El conjunto de todas las funciones del universo del discurso se conoce como base funcional.

Predicados

Un predicado puede utilizarse para definir hechos o verdades aceptadas dentro de un universo de discurso. El conjunto de todos los predicados usados en la conceptuación se conoce como base relacional.

Variables, constantes y cuantificadores

Las variables se utilizan para representar los objetos del universo de discurso y se representan normalmente por cualquier secuencia de caracteres que inicie con una letra mayúscula.

Las constantes son representaciones que referencian siempre la misma entidad. Podría verse como una función que recibe 0 argumentos.

Cuantificadores:

Para todo (∀): nos permite expresar hechos acerca de todos los objetos en el universo del discurso, sin enumerarlos.

Existe (∃): nos permite expresar la existencia de un objeto en el universo de discurso con cierta propiedad en particular.

Conectivas

La lógica de primer orden incorpora las conectivas de la lógica proposicional para combinar los predicados, constantes, variables y cuantificadores.

La Tierra y Marte poseen menos masa que Júpiter

Conectiva	Lenguaje natural	Símbolo
Negación	no	$\neg, \sim$
Conjunción	y	$\wedge, \ \& \ , \cdot$
Disyunción	o	$\vee$
Condicional	si... entonces	$\rightarrow, \supset$
Bicondicional	si y sólo sí	$\leftrightarrow, \equiv$
Disyunción excluyente	o bien... o bien	$\nleftrightarrow, \oplus, \not\equiv, W, \veebar$

4.2 Unificación y resolución.

Unificar consiste en atribuir provisionalmente valores a variables. La palabra «provisional» diferencia a la unificación de la asignación en lenguajes procedimentales (operación ésta que es «destructiva»). En los ejemplos que hemos seguido siempre se ha tratado de unificar variables con constantes. Pero la unificación es más general. Así, por ejemplo, p(a,X,b) puede unificarse con p(a,Y,Z) mediante la **sustitución** de Y por X y Z por b (lo que se puede indicar escribiendo «X/Y, b/Z» , con una notación que ya hemos utilizado en las figuras anteriores). Sin embargo, p(Z,X,b) y p(a,Y,Z) no pueden unificarse, ya que Z no se puede sustituir al mismo tiempo por la constante a y por la b.

La unificación de una variable con una constante (o de un conjunto de variables con un conjunto de constantes) se llama «**ejemplarización**» : «sócrates» es un **ejemplar** de X en «hombre(X)» .

Retroceso

Como ya se ha visto en los ejemplos, en el momento en que fracasa la unificación de un predicado el procesador retrocede y deshace la última unificación que tuvo lugar (se deshacen todas las sustituciones) y prueba con otra unificación. Para hacer esto posible, el procesador maneja una estructura de datos de tipo pila en la que va almacenado las

sustituciones y los «puntos de retroceso» . Cuando llega a un fracaso, recupera de la pila la información necesaria para restaurar el estado anterior al de la última unificación (si la pila está vacía, entonces responde «NO»).

Procesador

El procesador de Prolog consta de dos procedimientos: un **procesador de objetivos** y un **procesador de reglas y hechos**. En un intérprete son programas independientes que se llaman de manera recursiva; en un compilador están imbricados, pero el resultado de la ejecución es el mismo.

He aquí una descripción muy resumida de ambos procedimientos:

El procesador de objetivos llama sucesivamente al procesador de reglas y hechos para cada uno de los subobjetivos pendientes («de izquierda a derecha»). Si el procesador de reglas y hechos devuelve un fracaso, el de objetivos retrocede al subobjetivo anterior (si lo hay) y deshace la última unificación, llamando de nuevo al procesador de reglas con ese subobjetivo.

El procesador de hechos y reglas explora las cláusulas («de arriba abajo») buscando posibles unificaciones. Si encuentra una unificación con un hecho la introduce en la pila e informa del éxito al procesador de objetivos. Si encuentra unificación con la cabeza de una regla llama al procesador de objetivos para el conjunto de subobjetivos del cuerpo de la regla.

Ante una consulta, se llama al procesador de objetivos para el conjunto de subobjetivos que constituye la consulta. En caso de éxito para todos los subobjetivos, responde «YES» (si la consulta no contenía variables) o bien los valores unificados con las variables, y en este último caso continúa la búsqueda de más unificaciones hasta agotar las posibilidades. En caso de fracaso de alguno de los subobjetivos iniciales el procesador da la respuesta «NO».

Influencia del modelo procesal en el modelo funcional

El ideal de la programación lógica es que la máquina que ejecuta el programa interprete exclusivamente su significado lógico, y que, por tanto, dé los mismos resultados para dos programas que, aunque distintos, sean lógicamente equivalentes. Pero cuando lo que tenemos como «máquina lógica» es un intérprete (o compilador) de Prolog programado sobre una máquina convencional las cosas dejan de ser ideales, y es preciso tener conocimiento del modelo procesal para escribir programas correctos. Veámoslo en el ejemplo de los jefes y los superiores, pero ahora, para simplificar la presentación de los procesos, supondremos que sólo hay tres hechos:

H1: jefe(ana,eva).

H2: jefe(eva,pepe).

H3: jefe(eva,paco).

Consideremos cuatro maneras diferentes de escribir las reglas para definir superior:

Versión 1:

R1: superior(X,Y) :- jefe(X,Y).

R2: superior(X,Y) :- jefe(X,Z),superior(Z,Y).

Versión 2:

R1: superior(X,Y) :- jefe(X,Z),superior(Z,Y).

R2: superior(X,Y) :- jefe(X,Y).

Versión 3:

R1: superior(X,Y) :- jefe(X,Y).

R2: superior(X,Y) :- superior(X,Z),jefe(Z,Y).

Versión 4:

R1: superior(X,Y) :- superior(X,Z),jefe(Z,Y).

R2: superior(X,Y) :- jefe(X,Y).

La primera de las versiones es la que ya habíamos utilizado, pero obsérvese que las cuatro son lógicamente equivalentes. Analicemos, en cada caso, el proceso que seguiría la máquina para la consulta:

?- superior(ana,paco).

cuya respuesta debe ser «YES».

En el caso de la versión 1, la Figura A.7 resume el proceso. Como la regla R1 fracasa para el objetivo inicial, se prueba con la R2, que lo descompone en los subobjetivos «jefe(ana,Z)» y «superior(Z,paco)» . H1 satisface al primero con la sustitución eva/Z, lo que convierte al segundo en «superior(eva,paco)» , que se satisface por aplicación sucesiva de R1 y H3.

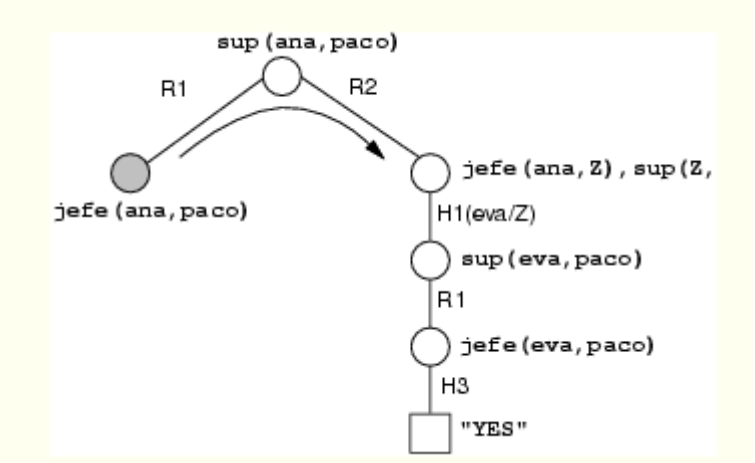


Figura A.7: Proceso de búsqueda para superior(ana,paco) con la versión 1 de las reglas para definir superior(X,Y)

4.3 Cláusulas de Horn. Resolución SLD.

El acrónimo SLD La S indica que se está usando una regla fija para seleccionar el literal sobre el que operara el paso de resolución (una regla de computación). La L indica que el proceso es lineal puesto que siempre se usa como padre en el paso de resolución al más reciente resolvente (esta es la cláusula central, la cláusula lateral es precisamente la que se toma del programa usando la regla de computación). La D indica que los programas son conjuntos de cláusulas definidas. La nueva regla restringida (Resolución SLD) sigue siendo (como Resolución) completa con respecto a refutación (¿ Qué significa esto?). Observe que

cuando se habló de una regla fija para seleccionar el literal no se dijo cuál regla. resolución sigue siendo completa con la estrategia SLD sin importar cual regla de computación se usa (a esto se le llama independencia de la regla de computación).

La resolución general es un mecanismo muy potente de demostración...

pero tiene un alto grado de indeterminismo: en la selección de las cláusulas con las que hacer resolución y en la selección de los literales a utilizar en la resolución. Desde el punto de vista computacional es muy ineficiente. Desde el punto de vista práctico puede sacrificarse algo de expresividad y obtener un mecanismo más eficiente que sustente un lenguaje de programación más "realista": restringimos la forma de las cláusulas de modo que a lo sumo tengan un literal positivo. En notación de Kowalski esto quiere decir que a lo sumo tienen un ´átomo en el lado izquierdo de ← estudiaremos un método de resolución específico para este tipo de cláusulas.

Cláusulas de Horn

Una cláusula de Horn es una secuencia de literales que contiene a lo sumo un literal positivo. Al escribirla en notación de Kowalski tendrá una de estas cuatro formas:

1. Hecho: $p \leftarrow$
2. Regla: $\underbrace{p}_{cabeza} \leftarrow \underbrace{q_1, \ldots, q_n}_{cuerpo}$
3. Objetivo: $\leftarrow q_1, \ldots, q_n$
4. Éxito: $\leftarrow$

Los hechos y las reglas se denominan cláusulas definidas:

- Los hechos representan "hechos acerca de los objetos" (de nuestro universo de discurso), relaciones elementales entre estos objetos.
- Las reglas expresan relaciones condicionales entre los objetos, dependencias.

Las reglas engloban todos los casos en el siguiente sentido:

- un hecho es una regla con cuerpo vacío
- un objetivo es una regla con cabeza vacía
- y el éxito es una regla con cabeza y cuerpo vacíos

Nótese que en las cláusulas de Horn trabajamos con secuencias de literales en vez de conjuntos (como veníamos haciendo con las cláusulas generales). Esto implica dos cosas:

- los literales pueden aparecer repetidos en el cuerpo
- hay un orden en los literales del cuerpo (podemos hablar del primer literal, segundo literal, etc).
- Un predicado p queda definido por el conjunto de cláusulas (hechos y reglas) cuyas cabezas tienen ese símbolo de predicado. Así pues, la definición de un predicado en general tendrá el aspecto:

$$
\begin{aligned}
&p(t_1, \ldots, t_n) \quad \leftarrow \\
&p(s_1, \ldots, s_n) \quad \leftarrow \\
&\ldots \\
&p(u_1, \ldots, u_n) \quad \leftarrow q_1(\ldots), \ldots, q_m(\ldots) \\
&p(u_1, \ldots, u_n) \quad \leftarrow r_1(\ldots), \ldots, r_k(\ldots) \\
&\ldots
\end{aligned}
$$

4.4 Programación lógica con cláusulas de Horn.

Antes de explicar lo que son las cláusulas de Horn nos detendremos un momento en la escritura de cláusulas en forma de condicional, que ayuda a comprender su significado.

La forma clausulada es interesante, como hemos visto, para la implementación de sistemas deductivos, pero su interpretación por la mente humana es difícil. Sin embargo, podemos

hacer una transformación inversa muy sencilla que ayuda a interpretar las cláusulas. Dada una cláusula cualquiera, $L_1 \vee L_2 \vee ...$, podemos transformarla en una sentencia equivalente con forma de condicional. Para ello, escribimos primero los literales negativos y luego los positivos:

$$\neg p_1 \vee \neg p_2 \vee ... \vee \neg p_k \vee q_1 \vee q_2 \vee ... \vee q_m$$

Por una de las leyes de de Morgan, esta sentencia es equivalente a:

$$\neg(p_1 \wedge p_2 \wedge ... \wedge p_k) \vee q_1 \vee q_2 \vee ... \vee q_m$$

y por la equivalencia (9) esta otra sentencia también es equivalente:

$$(p_1 \wedge p_2 \wedge ... \wedge p_k) \Rightarrow (q_1 \vee q_2 \vee ... \vee q_m)$$

Se llaman **cláusulas de Horn** aquellas que tienen como máximo un literal positivo. Hay dos tipos:

Las **cláusulas determinadas** (definite clauses), o «cláusulas de Horn con cabeza» son las que sólo tienen un literal positivo:

$$(\neg p_1 \vee \neg p_2 \vee ... \vee \neg p_k \vee q) \equiv (p_1 \wedge p_2 \wedge ... \wedge p_k \Rightarrow q)$$

Caso particular son las no tienen más que ese literal positivo, que representan «hechos» , es decir, conocimiento factual.

Los **objetivos determinados** (definite goals), o «cláusulas de Horn sin cabeza» son las que no tienen ningún literal positivo:

$$(\neg p_1 \vee \neg p_2 \vee ... \vee \neg p_k) \equiv \neg(p_1 \wedge p_2 \wedge ... \wedge p_k)$$

En este caso, las fórmulas atómicas $p_1,p_2,...,p_k$ son inconsistentes, es decir no es posible que sean todas verdaderas. Provienen de la negación de conclusiones que el sistema deductivo debe obtener mediante resolución y refutación.

Además de permitir la implementación de sistemas deductivos completos, una propiedad interesante de las cláusulas de Horn es que la asunción del mundo cerrado no conduce a una teoría inconsistente, según este teorema:

«Si Δ_0 está formada por cláusulas de Horn que no son inconsistentes entonces la teoría que resulta de la asunción del mundo cerrado, T_D, es consistente»

El interés de las cláusulas de Horn (como el de los demás conceptos introducidos en este Capítulo) está en su versión de lógica de predicados, donde también es aplicable el teorema anterior, y donde son el fundamento de la programación lógica.

Sistemas deductivos con resolución y refutación

El modelo procesal de un sistema deductivo con resolución y refutación tendrá como estado inicial $\Delta_0 = \{\{P\}, \neg C\}$, es decir, el conjunto formado por las premisas (en forma clausulada) y la negación de la conclusión (de esta negación resultará, en general, un conjunto de cláusulas). El proceso seguirá su curso, de acuerdo con la estrategia de control adoptada, aplicando siempre la única regla: la de resolución. Si C es una conclusión correcta y el sistema deductivo es completo el proceso terminará encontrando la cláusula vacía, Φ. Tal cosa ocurrirá cuando como resolvente aparezca un literal L y se observe que $\neg L$ ya estaba incluido en Δ: la resolución de los dos produce como resolvente la cláusula vacía, que corresponde a la contradicción.

Como decíamos la estrategia más sencilla es la de la búsqueda en extensión. Se demuestra que con esta estrategia un sistema deductivo para la lógica de proposiciones con resolución y refutación:

es consistente y completo: siempre que $\Delta_0 \vdash_{SI} \Phi$ se cumple que $\Delta_0 \models \Phi$ (es decir, si el sistema refuta C, Δ_0 es inconsistente, lo que significa que las premisas $\{P\}$implican C), y siempre que $\Delta_0 \models \Phi$ resulta que $\Delta_0 \vdash_{SI} \Phi$ (es decir, si las premisas implican la conclusión el sistema encuentra la cláusula vacía).

es decidible: si $\Delta_0 \not\models \Phi$ el proceso termina sin generar la cláusula vacía

Pero tanto la complejidad espacial como la temporal de una búsqueda en extensión son del orden de $O(bd)$, donde b es el factor de ramificación y d la profundidad de la solución (número de resoluciones necesario para llegar a ella). En este caso, además, b crece proporcionalmente con la profundidad de la búsqueda. En efecto, si el estado inicial, Δ_0, contiene $N = n + m$ cláusulas (n procedentes de las n premisas y m de la conclusión negada), puede haber hasta $N \times (N - 1)/2$ resolventes, cada una de ellas susceptible de, a su vez, resolverse con otra o con alguna cláusula de Δ_0, etc.

De entre las muchas estrategias propuestas para reducir la complejidad de la búsqueda comentaremos solamente la más utilizada, que se llama **resolución lineal con la entrada**: siempre se toma una generatriz perteneciente a la primera parte de Δ_0 ($\{P\}$) y otra de la segunda ($\neg C$) o descendiente de ella. Es decir, una generatriz procedente de las premisas iniciales (que forma el llamado conjunto de entrada) y otra que procede directa o indirectamente de la conclusión negada.

Tomando como ejemplo:

P_1:	$p \Rightarrow q$	$\rightsquigarrow$	C1: $\neg p \vee q$
P_2:	$q \Rightarrow (\neg s \Rightarrow r)$	$\rightsquigarrow$	C2: $\neg q \vee s \vee r$
P_3:	$\neg s$	$\rightsquigarrow$	C3: $\neg s$
$\neg C$:	$\neg(p \Rightarrow r)$	$\rightsquigarrow$	C4: p
		$\rightsquigarrow$	C5: $\neg r$

El proceso puede ser:

{C1, C4}	$\rightsquigarrow$	C6: q
{C2, C6}	$\rightsquigarrow$	C7: $s \vee r$
{C3, C7}	$\rightsquigarrow$	C8: r
{C4, C8}	$\rightsquigarrow$	Φ

El problema con esta estrategia es que no es completa. Consideremos esta inferencia deductiva:

P1: $p \Rightarrow q$

P2: $\neg p \Rightarrow q$

P3: $q \Rightarrow p$

C: $p \wedge q$

Pasando las premisas y la conclusión negada a forma clausulada resultan la cláusulas:

C1: $\neg p \vee q$

C2: $p \vee q$

C3: $p \vee \neg q$

C4: $\neg p \vee \neg q$

La estrategia obliga a que una de las cláusulas generatrices sea C1, C2 o C3, y la otra C4 o descendiente de ella.

{C1,C4} $\rightsquigarrow$ C5: $\neg p$

{C2,C4} $\rightsquigarrow p \vee \neg p$, $q \vee \neg q$ (tautologías)

{C3,C4} $\rightsquigarrow$ C6: $\neg q$

{C1,C6} $\rightsquigarrow$ C5

{C2,C5} $\rightsquigarrow$ C7: q

Aunque está claro que hay una refutación entre C6 y C7, las condiciones de la estrategia impiden aplicarla.

Sin embargo, esta estrategia es completa cuando se impone una cierta restricción sobre la forma de las cláusulas: que sean cláusulas de Horn.

4.5 Semántica de los programas lógicos.

En la Teoría de lenguajes de programación, la semántica es el campo que tiene que ver con el estudio riguroso desde un punto de vista matemático del significado de los lenguajes de programación. Esto se hace evaluando el significado de cadenas sintácticamente legales definidas por un lenguaje de programación específico, mostrando el proceso computacional involucrado. En el caso de que la evaluación fuera de cadenas sintácticamente ilegales, el resultado sería no-cómputo. La semántica describe el proceso que una computadora sigue cuando ejecuta un programa en ese lenguaje específico. Esto se puede mostrar describiendo la relación entre la entrada y la salida de un programa, o una explicación de cómo el

programa se ejecutará en cierta plataforma, y consecuentemente creando un modelo de computación.

Semánticas formales ayudan, por ejemplo, a escribir compiladores, a tener un mejor entendimiento de lo que un programa está haciendo y a hacer determinadas pruebas, como demostrar que el siguiente código.

```
if 1 == 1 then S1 else S2
```

Vistazo general

El campo de las semánticas formales abarca todo lo que sigue:

- Definición de modelos semánticos
- Relaciones entre diferentes modelos semánticos
- Relaciones entre los distintos enfoques de significado
- Relación entre computación y las estructuras matemáticas subyacentes de varios campos como la lógica, teoría de conjuntos, teoría de modelos, teoría de categorías, etc.

También tiene vínculos cercanos con otras áreas de la ciencia de la computación como el diseño de lenguajes de programación, teoría de tipos, intérpretes y compiladores, verificación de programas y modelos.

Descripción

Hay muchos enfoques a las semánticas formales, las cuales pertenecen a tres categorías principales:

- **Semántica denotacional**, por medio de las cuales cada frase en el lenguaje es interpretada como una denotación. Tales denotaciones a menudo son objetos matemáticos que habitan espacios matemáticos, pero no es un requerimiento que éstas deban serlo. Como una necesidad práctica, las denotaciones se describen usando alguna forma de notación matemática, la cual en turno puede ser formalizada como un metalenguaje denotativo. Por ejemplo, las semánticas denotacionales de lenguajes funcionales muchas veces traducen el lenguaje en teoría de dominio. Las descripciones semánticas denotacionales también pueden servir como

traducciones de composición de un lenguaje de programación en el metalenguaje denotativo y se utiliza como base para el diseño de compiladores.

- **Semántica operacional**, donde la ejecución del lenguaje se describe directamente (en vez de hacerse mediante el uso de una traducción). Las semánticas operacionales tienen que ver con la interpretación, aunque nuevamente el "lenguaje de implementación" del intérprete es de forma general un formalismo matemático. Las semánticas operacionales pueden definir una máquina abstracta (como la máquina SECD), y dan significado a las frases describiendo las transiciones que ellas inducen en los estados de la máquina. De forma alternativa, como con el cálculo lambda puro, las semánticas operacionales pueden ser definidas vía transformaciones sintácticas sobre frases del lenguaje en sí mismo.

- **Semántica axiomática**, a través de la cual se le da significado a las frases describiendo los axiomas lógicos que se aplican a ellas. Las semánticas axiomáticas no hacen distinción entre un significado de una frase y las fórmulas lógicas que la describen, su significado es exactamente lo que se puede probar de ella en alguna lógica. El ejemplo canónico de semánticas axiomáticas es la lógica de Hoare.

Las diferencias entre estas tres amplias clases de aproximaciones puede que a veces sean difusas, pero todas las aproximaciones conocidas a las semánticas formales usan las técnicas de arriba, o alguna combinación de ellas.

Aparte de la elección entre aproximación denotacional, operacional o axiomática, la mayoría de las variaciones en los sistemas de semántica formal vienen de la elección de la base en el formalismo matemático.

4.6 Representación causada del conocimiento.

Lo que nos interesa es la representación, es decir, la modelización del conocimiento Orientaciones:

- Simbólica: la descripción del comportamiento inteligente se basa en sistemas simbólicos, más o menos formalizados

- Conexionista: para describir el comportamiento inteligente se modelizan sistemas neuronales

La **representación del conocimiento** y el razonamiento es un área de la inteligencia artificial cuyo objetivo fundamental es representar el conocimiento de una manera que facilite la inferencia (sacar conclusiones) a partir de dicho conocimiento. Analiza cómo pensar formalmente - cómo usar un sistema de símbolos para representar un dominio del discurso (aquello de lo que se puede hablar), junto con funciones que permitan inferir (realizar un razonamiento formal) sobre los objetos. Generalmente, se usa algún tipo de lógica para proveer una semántica formal de cómo las funciones de razonamiento se aplican a los símbolos del dominio del discurso, además de proveer operadores como cuantificadores, operadores modales, etc. Esto, junto a una teoría de interpretación, dan significado a las frases en la lógica.

Cuando diseñamos una representación del conocimiento (y un sistema de representarían del conocimiento para interpretar frases en la lógica para poder derivar inferencias de ellas) tenemos que hacer elecciones a lo largo de un número de ámbitos de diseño. La decisión más importante que hay que tomar es la *expresividad* de la representación del conocimiento. Cuanto más expresiva es, decir algo es más fácil y más compacto. Sin embargo, cuanto más expresivo es un lenguaje, más difícil es derivar inferencias automáticamente de él. Un ejemplo de una representación del conocimiento poco expresiva es la lógica proposicional. Un ejemplo de una representación del conocimiento muy expresiva es la lógica autoepistémica. Las representaciones del conocimiento poco expresivas pueden ser tanto completas como consistentes (formalmente menos expresivas que la teoría de conjuntos). Las representaciones del conocimiento más expresivas pueden ser ni completas ni consistentes.

El principal problema es encontrar una representación del conocimiento y un sistema de razonamiento que la soporte, que pueda hacer las inferencias que necesita una aplicación dentro de los límites de recursos del problema a tratar. Los desarrollos recientes en la representación del conocimiento han sido liderados por la web semántica, y han incorporado el desarrollo de lenguajes y estándares de representación del conocimiento

basados en XML, que incluyen Resource Description Framework (RDF), RDF Schema, DARPA Agent Markup Language (DAML), y Web Ontology Language (OWL).

Visión general

Existe un conjunto de técnicas de representación como los marcos, las reglas, el etiquetado y las redes semánticas, que tienen su origen en teorías del procesamiento de la información humana. Como el conocimiento se usa para conseguir comportamiento inteligente, el objetivo fundamental de la representación del conocimiento es representar el conocimiento de manera que facilite el razonamiento. Una buena representación del conocimiento debe ser declarativa, además de conocimiento fundamental. Qué es la representación del conocimiento se entiende mejor en términos de cinco roles fundamentales que juega, cada uno crucial para la aplicación:[1,2]

- Una representación del conocimiento es fundamentalmente un sucedáneo, un sustituto para el objeto en sí, usado para activar una entidad a efectos de determinar las consecuencias, pensando en lugar de actuando, o sea, razonando acerca del mundo en lugar de tomando acción en él.
- Es un grupo de compromisos ontológicos, una respuesta a la pregunta sobre los términos en que se debe pensar acerca del mundo.
- Es una teoría fragmentaria del razonamiento inteligente, expresado en términos de tres componentes: (i) El concepto fundamental de la representación del razonamiento inteligente; (ii) El conjunto de inferencias que la representación sanciona; y (iii) El conjunto de inferencias que recomienda.
- Es un medio para una computación pragmáticamente eficiente, el entorno computacional en que el pensamiento tiene lugar. Una contribución para esta eficiencia pragmática viene dada por la guía que una representación provee para organizar información, de modo que facilite hacer las inferencias recomendadas.
- Es un modo de expresión humana, un lenguaje en el que se dicen cosas sobre el mundo.

Algunas cuestiones que surgen en la representación del conocimiento, desde el punto de vista de la inteligencia artificial, son: -Cómo se representa el conocimiento -Cuál es la naturaleza del conocimiento -el carácter particular o general del dominio de un esquema

de representación -Cuán expresivo es un esquema de representación o lenguaje formal -el carácter declarativo o procesal de los esquemas

En el campo de la inteligencia artificial, la solución de problemas puede ser simplificada con una elección apropiada de *representación del conocimiento*. Algunos problemas son más fáciles de resolver al representar el conocimiento de un modo determinado. Por ejemplo, es más fácil dividir números representados en el sistema arábigo que números representados en el sistema romano.

4.7 Consulta de una base de cláusulas

Un programa en Prolog está constituido por una secuencia de cláusulas. Estas cláusulas deben representar todo el conocimiento necesario para resolver el problema. Se pueden diferenciar tres tipos de Cláusulas:

• Hechos (afirmaciones), que a su vez pueden representar:

a) Objetos.

b) Propiedades de objetos.

c) Relaciones entre objetos.

• Reglas.

• Consultas (cláusulas negativas).

Como se ha mencionado, un programa Prolog es una secuencia de cláusulas, donde cada cláusula puede estar formada por uno o varios predicados. Las cláusulas deben terminar obligatoriamente en punto.

Términos

En Prolog hay tres tipos de términos: CONSTANTES, VARIABLES YESTRUCTURAS

Tipo	*Comentarios*	*Ejemplos*	*Casos Especiales*
Constante			
Átomo	En ***Minúsculas***. Nombre de: • Objetos • Propiedades • Relaciones	luis pedro edad color padre 'rey león'	Átomos especiales :- ?-
Número			
Entero	Sólo dígitos, +, -	63	
Real	Y punto decimal	365.2425	
Variable	Comienzan con ***Mayúscula o con _***	Suma X _Y	Variable anónima: _
Estructura	Término compuesto por otros Términos	edad(luis, 63) padre_de(luis, eva)	

4.8 Espacios de búsqueda

Cuando se resuelve un problema, se busca la mejor solución entre un conjunto de posibles soluciones. Al conjunto de todas las posibles soluciones a un problema concreto se llama espacio de búsqueda. Cada punto en el espacio de búsqueda representa una posible solución. Cada posible solución se le puede asociar un fitness o un valor que indicará cómo de buena es la solución para el problema. Un algoritmo genético (AG) devolverá la mejor solución de entre todas las posibles que tenga en un momento dado.

Existen muchos métodos que se usan para buscar una solución válida, pero no necesariamente obtienen la mejor solución. Algunos de estos métodos son los algoritmos de escalada, backtracking o vuelta atrás, búsqueda a ciegas y los algoritmos genéticos. Las soluciones que encuentran estos tipos de búsqueda suelen ser buenas soluciones, pero no siempre encuentran la óptima.

En optimización, espacio de búsqueda se refiere al dominio de la función a ser optimizada. En el caso de los algoritmos de búsqueda, que manejan espacios discretos, se refiere al conjunto de todas las posibles soluciones candidatas a un problema.

Fijada una regla de cálculo, los distintos cálculos posibles para un programa constituido por una cuestión Q y una base de conocimiento P se pueden representar gráficamente mediante un árbol de búsqueda caracterizado por:

su raíz, etiquetada con la cuestión Q o primer resolvente;
y los demás nodos etiquetados con los resolventes producidos por pasos de resolución; además

para cada nodo etiquetado con un resolvente no vacío, si A es la fórmula que selecciona la regla de cálculo, existirán tantos descendientes como cláusulas con cabecera coincidente con A existan en el procedimiento de definición del correspondiente predicado, los nodos etiquetados con resolventes vacíos no tienen descendientes. Las distintas reglas de búsqueda representan distintas formas de recorrido de los árboles de búsqueda.

EJEMPLO
si definimos:
Masuno(X, Y): - Y is X+1.
Xmasuno (X, Y): - Y = X+1.
Observaremos el siguiente comportamiento:
?- masuno (,5).
Yes
?- xmasuno(4,5)
No

4.9 Programación lógica con números, listas y árboles.

Numéricos

En PROLOG los objetos numéricos pueden corresponder a tipos integer o float de C. Para realizar operaciones numéricas, se tiene el predicado is, que se comporta como una asignación en un lenguaje imperativo. Así, el objetivo X is <expresión> será verdadero cuando X unifique con el resultado numérico de evaluar <expresión>.

Lista

Este algoritmo es my ineficiente en el peor caso tendrá que generar las N! permutaciones para una lista de largo N.

Se representan utilizando términos compuestos f(t1, . . . , tn) (para los naturales ya hemos utilizado una estructura de datos s(_)).

4.10 Control de búsqueda en programas lógicos.

El predicado! (leído corte) proporciona control sobre el mecanismo de backtracking de programación lógica: siempre tiene éxito, pero tiene el efecto lateral de podar todas las elecciones alternativas en el nodo correspondiente en el árbol de búsqueda, Este es un predicado meta-lógico o extra-lógico: es un predicado que afecta al comportamiento operacional de programación lógica. Es en cierto sentido un predicado impuro (ajeno a la lógica) pero es un predicado muy útil. . . casi fundamental para el programador de programación lógica.

¿Cómo opera el corte?

Cuando se resuelve un objetivo p con una cláusula de la forma:

P: - q1, …, qn !, r1, …, rm

Programacion lógica intenta resolver los objetivosq1, …, qn normalmente (haciendo backtraking sobre cada uno de ellos si es necesario);

EJEMPLO

Definamos un predicado para incrementar en 1 los enteros de una lista (dejando intactos los no enteros):

```
incLst2( [ ] , [ ] ) .
incLst2( [ X| Xs] , [ Y| Ys] ) : - integer( X) , ! , Y is X+1, incLst2( Xs, Ys) .
incLst2( [ X| Xs] , [ X| Ys] ) : - incLst2( Xs, Ys) .
```

4.11 Manipulación de términos predicados metalógicos.

El primer mecanismo es el = que usamos para unificar términos.
El operador ==/2 y se satisface cuando sus dos operando son términos literalmente iguales, la diferencia con = es que == no fuerza unificación. El operador \==/ es la negación del operador anterior y se satisface cuando sus operandos son términos literalmente diferentes:

?- f(X) == f (Y). No
?- X = Y, f(X) \== f (Y). No

Existen operadores para el chequeo de tipos de términos, se pueden usar para manejo de errores, el predicado var/1 se satisface cuando su argumento es una variable no unificada, nonvar/1 es su negación, ejemplo:

```
Masuno (X,Y) :- var (X), write ('ERROR:'),
Write('primer argumento no instanciado!!'),
!, fail.
```

La construcción sirve para pasar nombres de predicados como argumentos. El siguiente predicado determina si cierta lista esta ordenada según algún criterio que no se conoce a priori:

```
ordC ([_],_).
ordC ([X, Y | L], Criterio) : -
Menor =.. [Criterio, X, Y],
call (Menor),
ordC ([Y|L], Criterio).
```

BIBLIOGRAFÍA BÁSICA Y COMPLEMENTARIA

- Disciplina de tipos Consultado el 03 de abril 2014. Recuperado de:
- http://programacionlogicayfuncional.wordpress.com/2014/02/13/disciplina-de-tipos/
- Consultado el 03 de abril 2014. Recuperado
- http://programacionlogicafun.blogspot.mx/2014/02/14-disciplina-de-tipos-en-distintos.html
- Consultado el 05 de mayo 2015. Recuperado
- http://programacionlogyfun.blogspot.com/2016/03/programacion-logica-y-funcional-la.html
- Programación lógica y funcional
- http://itpn.mx/recursosisc/8semestre/programacionlogicayfuncional/Unidad%201.pdf
- Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, Communications of the ACM, August 1978, Vol 21 (8)
- Chakravarty, M. M. and Keller, G., The risks and benefits of teaching purely functional programming in first year. Journal of Functional Programming 14, 1 (Jan. 2004), 113-123
- Doets, K. y van Eijck, J. The Haskell Road to Logic, Math and Programming, King's College Publications, Londres, 2004.
- FDPE 08: Proceedings of the 2008 international workshop on Functional and declarative programming in education, ACM, New York, USA, 2008.
- Fokker, J. Functional Programming, Department of Computer Science, Utrecht University, 1995
- Iverson, K. A Programming language, John Wiley and Sons, New York, 1962
- Labra G., J. Introducción al Lenguaje Haskell, Universidad de Oviedo, Departamento de Informática, 1998
- Ruiz, B., Gutiérrez, F., Guerrero, P. y Gallardo, J.E. Razonando con Haskell, Un curso sobre programación funcional, ThompsonParaninfo, Madrid, 2004
- Becerra, César. Algoritmos. Ed. César Becerra, 1993.

- Séller Arthur. Programación en Pascal. Ed Mc Graw Hill
- Konvalina John Stanley Wileman. Programación con Pascal. Ed Mc Graw Hill.