Departamento de Tecnologías de la Información

PRÁCTICA 1

LENGUAJES FUNCIONALES

Universidad de Huelva

Ciencias de la Computación e Inteligencia Artificial

Bienvenida

Tutor de prácticas: Antonio Palanco Salguero

¿qué hago aquí?

Teoría y prácticas INDEPENDIENTES: asistencia no obligatoria

No asiste: irá a examen

Evaluación independiente con la que se hace media.

¿Qué vamos a ver?

Paradigmas de la programación

Lenguajes Funcionales

HASKELL

1.1. Paradigmas de programación

Los paradigmas de programación son modelos para resolver problemas comunes con nuestro código.

1.1. Paradigmas de programación

Imperativa

El primer paradigma que se suele estudiar es el paradigma imperativo.

Para resolver un problema se deben realizar una serie de pasos y el programador es el encargado de describir de forma **ordenada y sistemática** los pasos que debe seguir el ordenador para obtener la solución.

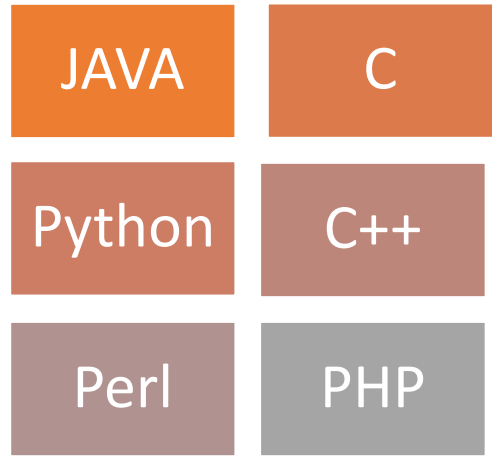
BASIC	C
Fortran	Pascal
Perl	PHP

Estructurada

Surge para mejorar los “códigos espagueti”.

3 estructura básicas

- **Secuencia**: ejecución en el orden de aparición.
- **Selección o condición**: se ejecuta sentencia en función del valor de una variable.
- **Iteración**: ejecuta 1 o un conjunto de sentencias cuando una variable booleana sea verdadera



Ejemplo de programa – ordenación burbuja (imperativo y estructurado)

```
void intercambiar(int *x,int *y){
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
void burbuja(int lista[], int n){
    int i,j;
    for(i=0;i<(n-1);i++)
        for(j=0;j<(n-(i+1));j++)
                if(lista[j] > lista[j+1])
                        intercambiar(&lista[j],&lista[j+1]);
}
```

50	26	7	9	15	27	Array Original
Primera Pasada:						
26	50	7	9	15	27	Se intercambian el 50 y el 26
26	7	50	9	15	27	Se intercambian el 50 y el 7
26	7	9	50	15	27	Se intercambian el 50 y el 9
26	7	9	15	50	27	Se intercambian el 50 y el 15
26	7	9	15	27	50	Se intercambian el 50 y el 27
Segunda Pasada:						
7	26	9	15	27	50	Se intercambian el 26 y el 7
7	9	26	15	27	50	Se intercambian el 26 y el 9
7	9	15	26	27	50	Se intercambian el 26 y el 15

Declarativa

Se basa en unidades conceptuales básicas que se pueden combinar según unas determinadas reglas para generar nueva información.

El dominio, será el conjunto de todas esas unidades conceptuales

Hay que “preguntar” a la máquina en base a ese dominio y es la “maquina” la que debe de dar respuesta, si existe, con alguna de las unidades o combinación de ellas

Se divide en dos subparadigmas: programación **lógica (predicado como unidad lógica) y** programación **funcional (función como unidad lógica)**

Declarativa: ventajas

Descripciones compactas y muy expresivas.

Desarrollo del programa no tan orientado a la solución de un único problema. Horarios.

No hay necesidad de emplear esfuerzo en diseñar un algoritmo que resuelva el problema ya que al escribir el código, no es necesario determinar el procedimiento según el cual se alcanza el resultado (ordenar los pasos).

Ejemplo:

```
$nombres = array_values($listaparticipanes);
```

Lógica: subparadigma de programación declarativa

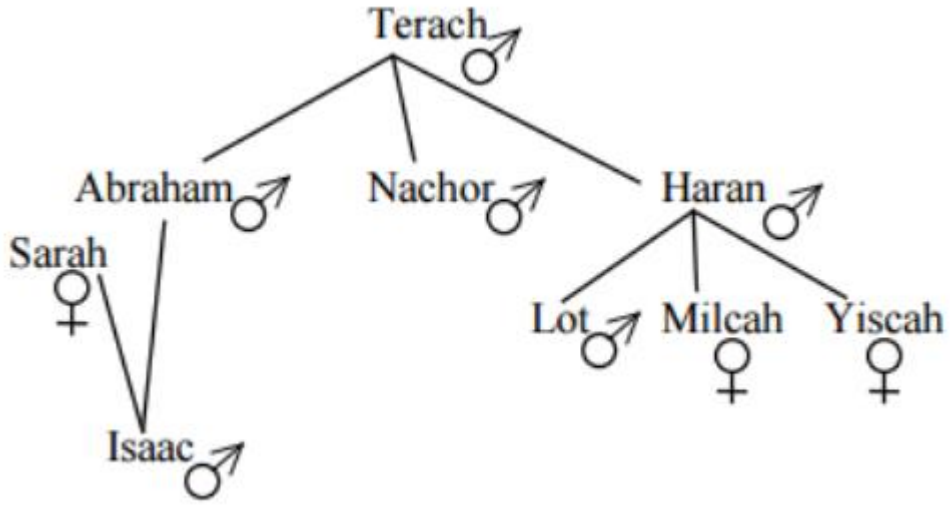
Utiliza el predicado lógico como ***concepto descriptivo básico***

El dominio (mundo) se describe mediante los predicados: que relacionan los objetos según las reglas de la lógica de predicados.

Resolución: se plantean afirmaciones que el sistema resolverá (si existe solución) en base a los predicados.

PROLOG

Lógica: ejemplo de relaciones de descendencia




```
es_padre(terach, abraham).
es_padre(terach, nachor).
es_padre(terach, haran).
es_padre(abraham, isaac).
es_padre(haran, lot).
es_padre(haran, milcah).
es_padre(haran, yiscah).

es_madre(sarah, isaac).

es_hombre(terach).
es_hombre(abraham).
es_hombre(nachor).
es_hombre(haran).
es_hombre(isaac).
es_hombre(lot).

es_mujer(sarah).
es_mujer(milcah).
es_mujer(yiscah).

es_hijo(X,Y):- es_padre(Y,X), es_hombre(X).

es_hija(X,Y):- es_padre(Y,X), es_mujer(X).

es_abuelo(X,Z):- es_padre(X,Y), es_padre(Y,Z).
```

Lógica: ejemplo de relaciones de descendencia

```
es_padre(terach, abraham).
es_padre(terach, nachor).
es_padre(terach, haran).
es_padre(abraham, isaac).
es_padre(haran, lot).
es_padre(haran, milcah).
es_padre(haran, yiscah).

es_madre(sarah, isaac).

es_hombre(terach).
es_hombre(abraham).
es_hombre(nachor).
es_hombre(haran).
es_hombre(isaac).
es_hombre(lot).

es_mujer(sarah).
es_mujer(milcah).
es_mujer(yiscah).

es_hijo(X,Y):- es_padre(Y,X), es_hombre(X).

es_hija(X,Y):- es_padre(Y,X), es_mujer(X).

es_abuelo(X,Z):- es_padre(X,Y), es_padre(Y,Z).
```

```
2 ?- es_padre(haran,lot), es_hombre(lot).

3 ?- es_padre(abraham,lot), es_hombre(lot).

4 ?- es_padre(abraham,X), es_hombre(X).

5 ?- es_padre(haran,X), es_mujer(X).

6 ?- es_hijo(lot,haran).

7 ?- es_hija(X,haran).
```

Funcional: subparadigma de programación declarativa

Definir QUÉ CALCULAR, pero no cómo calcularlo

Utiliza la función (no procedimiento) como unidad del dominio o ***concepto descriptivo básico***

No escribiremos programas, definiremos funciones que describen el problema.

Resolución: evaluación de una función basada en las previamente definidas.

http://www.haskell.org/haskellwiki/Introduction

Funciones puras

¿Qué es una función pura?

Una función representa una correspondencia entre dos conjuntos que asocia a cada elemento en el primer conjunto un único elemento del segundo

f(x)=a

Esto quiere decir que si evaluamos una función ***f* sobre un valor *x*** y se genera el resultado ***a*** **(es decir, *f(x)=a*)** entonces **este resultado será válido siempre**. Es más, se puede sustituir *f(x)* por *a* en cualquier expresión.

a + b = f(x) + b

Funciones puras: no se respeta en lenguajes imperativos

PROCEDIMIENTOS VS FUNCIÓN

```
a = f(x);
b = f(x);
```

Pueden generar valores diferentes para las variables *a* y *b*. Esto se debe a que la función *f* podría tener efectos colaterales (como modificar variables globales, por ejemplo).

Funciones puras

Se denomina *razonamiento ecuacional* al razonamiento lógico que utiliza la condición de inmutabilidad de las funciones.

Para poder sacar provecho a este tipo de propiedades es necesario restringir las reglas de definición de funciones para que no puedan tener efectos colaterales.

Esto afecta sobre todo al concepto de *variable*. En los lenguajes clásicos (*imperativos*) una *variable* es un contenedor que puede almacenar diferentes valores en diferentes momentos. Sin embargo, la noción matemática de *variable* se refiere a un cierto valor que puede ser conocido o desconocido, pero que es inmutable.

Funciones puras

Funciones puras

Se denomina por tanto ***función pura*** al tipo de función que sigue exactamente las nociones matemáticas y utiliza variables inmutables.

Para trabajar con variables inmutables…

Asignación, salvo en declaración.

No existen los bucles…

No es posible usar variables índices

Funciones puras

Por ejemplo, en un lenguaje imperativo, el factorial se puede calcular de la siguiente forma

```
int factorial( int x )
{
  int index = 1;
  int fact = 1;
  while(index < x)
  {
    index = index +1;
    fact = fact * index;
   }
  return fact;
}
```

FACTORIAL LENGUAJE IMPERATIVO

Funciones puras

En un lenguaje que utiliza funciones puras, el factorial se puede calcular de la siguiente forma

```
int factorial( int x )
{
  if(x <= 1) return 1;
  else return x * factorial(x-1);
}
```

FACTORIAL EN LENGUAJE FUNCIONAL PURO

RECURSIVIDAD

Funciones de orden superior

En ciencias de la computación las **funciones de orden superior** son funciones que cumplen al menos una de las siguientes condiciones:

- Tomar una o más funciones como entrada: funciones como parámetros
- Devolver una función como salida: función como salida

Para poder utilizar funciones como datos es necesario definir el tipo de dato y se suele utilizar el operador "flecha" y la **declaración de tipo** que permita asociar al identificador a un cierto tipo de dato funcional

typedef intfun = int -> int

funciones con un argumento entero que generan como resultado un valor entero

Funciones de orden superior

La sintaxis de declaración de tipos de datos sería, por ejemplo:

TypeDecl ::= **typedef** **id** **equal** TypeExpr **semicolon**

TypeExpr ::= (TypeList **arrow**)* TypeList

TypeList ::= TypeBase (**comma** TypeBase)*

TypeBase ::= Type | **lparen** TypeExpr **rparen**

Funciones de orden superior

La sintaxis de declaración de tipos de datos sería, por ejemplo:

```
int -> int -> int
```

Una función con un argumento de tipo entero que genera como resultado otra función. Esta segunda función toma como argumento un valor entero y genera como resultado un valor entero.

Funciones anidadas

- Se denominan ***funciones anidadas*** a funciones que pueden ser definidas dentro del cuerpo de otras funciones, pudiendo acceder a los valores de los argumentos y variables locales de dicha función (lo que se conoce como su *ámbito léxico* o *lexical scope*).

1.2. Lenguajes funcionales

Por tanto

Se denominan ***lenguajes funcionales*** a los lenguajes de programación que soportan funciones de orden superior que admiten funciones anidadas con ámbito léxico. Ejemplos de este tipo de lenguajes son ***Scheme***, ***ML*** o ***Smalltalk***.

Se denominan ***lenguajes funcionales puros*** a los lenguajes de programación que incluyen funciones de orden superior y solo admiten la definición de funciones puras. Por ejemplo, el subconjunto funcional puro de *ML* o el lenguaje ***Haskell***.

También existen lenguajes de programación que solo admiten funciones puras pero no soportan las funciones de orden superior. Por ejemplo ***SISAL***.

1.2. Lenguajes funcionales

ESTRICTOS

Se denomina evaluación estricta a la técnica de programación en la que en tiempo de ejecución una expresión siempre es evaluada y sustituida por su valor

NO ESTRICTOS

Se denomina ***evaluación perezosa*** (***lazy***) o no estricta a la técnica de programación en la que las expresiones solo son evaluadas cuando es necesario utilizar su valor.

En tiempo de ejecución las expresiones pueden no ser evaluadas si no son requeridas para ello

1.3. HASKELL

¿QUÉ ES LA PROGRAMACIÓN FUNCIONAL?

El objetivo en la programación funcional es definir QUÉ CALCULAR, pero no cómo calcularlo.

Haskell es un lenguaje de programación:

- **funcional puro**
- con **tipos polimórficos estáticos**
- con evaluación perezosa (**lazy**)

El nombre lo toma del matemático Haskell Brooks Curry especializado en lógica matemática. Haskell está basado en el **lambda cálculo**.

1.3. HASKELL

Características de Haskell

- Inferencia de tipos. La declaración de tipos es opcional. El compilador puede calcular el tipo a partir de las expresiones a partir de un análisis estático de las definiciones previas y de las variables del cuerpo.
- Evaluación perezosa: sólo se calculan los datos si son requeridos
- Versiones compiladas e interpretadas
- Todo es una expresión
- Las funciones se pueden definir en cualquier lugar, utilizarlas como argumento y devolverlas como resultado de una evaluación.

Implementaciones de Haskell

Existen diferentes implementaciones de Haskell

- GHC: el más utilizado
- Hugs: Muy utilizado para aprender Haskell. Compilación rápida. Desarrollo rápido de código.
- Nhc98
- Yhc.

Para la realización de las prácticas utilizaremos la implementación Hugs (**http://haskell.org/hugs/**).

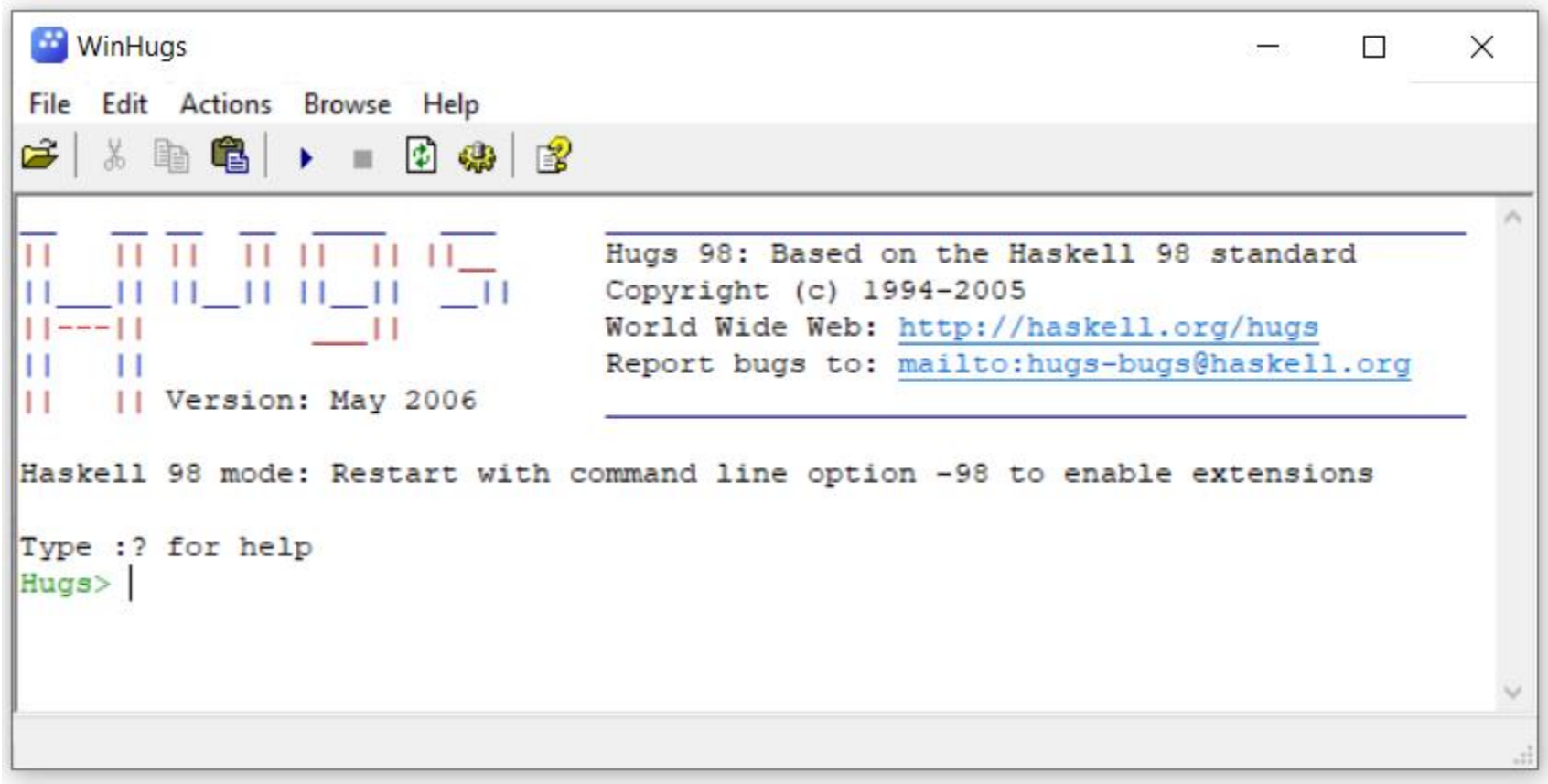
Implementaciones de Haskell

Existen diferentes implementaciones de Haskell

- GHC: el más utilizado
- Hugs: Muy utilizado para aprender Haskell. Compilación rápida. Desarrollo rápido de código.
- Nhc98
- Yhc.

Para la realización de las prácticas utilizaremos la implementación Hugs (**http://haskell.org/hugs/**).

Implementaciones de Haskell



1.3. HASKELL

Objetivo de las prácticas

- veremos una pequeña introducción a Haskell
- construir pequeñas funciones
- tener una idea del modo de programar en Haskell
- Y una pequeña visión sobre sus posibilidades.
- No veremos la conexión de Haskell con otros programas usando herramientas como **HaskellDirect**
- **Programación final en Scala**

Scala

Escala es un leguaje de programación híbrido

- Orientado a objetos
- Programación funcional

Se utiliza para aplicaciones en clústeres u ordenadores multicore.