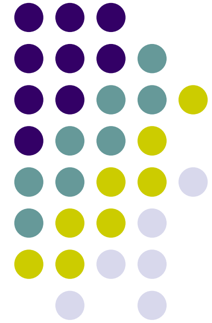
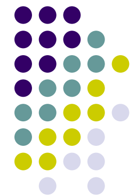
Programación Declarativa

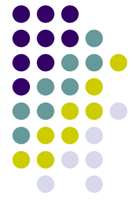
Tema 2: LISP (SCHEME)





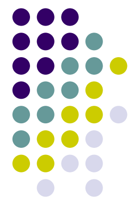
Tema 2: LISP (SCHEME)

- Introducción
- Representación de Datos
- Definición de Funciones
- Predicados
- Listas



Introducción (I)

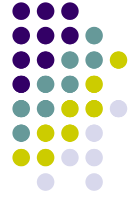
- SCHEME es un dialecto de LISP (*LISt Processing*).
- LISP es un lenguaje de programación funcional con una amplia base matemática (lambda-cálculo).
- La unidad de cálculo principal son los símbolos, en notación prefija, (+ 3 2).



Introducción (II)

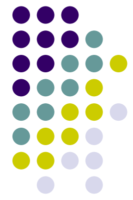
- Otro componente principal de LISP son las listas.
- Todo lo que esté encerrado entre paréntesis será considerado como una lista.
- En una lista, el primer elemento suele ser el nombre de la función que realiza, y el resto de elementos, los operandos.

  P.e., (+ 3 2)



Introducción (III)

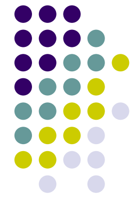
- La evaluación de los operandos se hace de manera ‘*perezosa*’, es decir, no se realiza hasta que hace falta.
- Es un lenguaje fuertemente funcional, ya que se compone exclusivamente de funciones anidadas en otras funciones.



Introducción (IV)

- No existe diferencia entre datos y código cuando hablamos dentro del ámbito de la sintaxis. Ejemplo:
  - Lista de tres elementos ⇒ (1 2 3)
  - Lista que suma dos elementos ⇒ (+ 1 2)

  - (<u>pepe</u> 1 2) ⇒ función
  - (<u>paco</u> 1 2) ⇒ constante



Introducción (V)

- El otro elemento son los átomos.
  - **Símbolos**: Cadena alfanumérica. No pueden aparecer paréntesis porque se usan para delimitar listas, pero puede aparecer cualquier otro carácter.

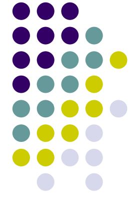
    Ejemplo: *mola*, *Hola*.
  - **Números**: 1, 2, 33, 7E3... También va a admitir números complejos.

    P.e., 2 + 3i

Introducción (VI)

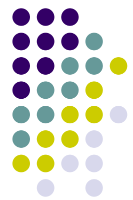
- Básicamente, las estructuras del lenguaje son listas y átomos.
- LISP no es sensible a mayúsculas y minúsculas.
- Para separar los elementos de una lista en LISP, tan sólo es necesario uno o varios espacios.

Evaluación de expresiones (I)



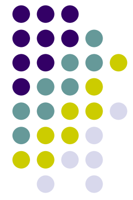
- LISP no ejecuta expresiones, tan sólo las evalúa.
- La evaluación consiste en:
  - Si evaluamos un número, el resultado será ese número.
    - 3 → 3
  - La evaluación de un símbolo consistirá en saber su valor, y el resultado será dicho valor. Si un símbolo no tiene asignado un valor, la evaluación falla.
    - símbolo → valor asignado

Evaluación de expresiones (II)



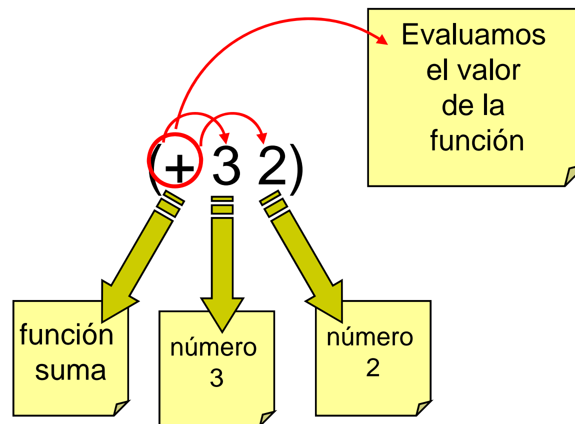
- Evaluación de listas:
  - Al evaluar una lista, se evalúa el primer elemento (normalmente, el nombre de la función) y a continuación los siguientes elementos, conforme van apareciendo (*evaluación 'perezosa'*).

    Cuando se evalúan todos los elementos que componen la lista, se pasa a evaluar ésta en conjunto.
    - lista → (_ _ _ _ _ _ _)

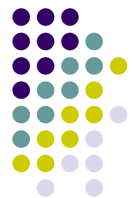


Evaluación de expresiones (III)

- Ejemplo:

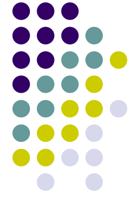


- El valor devuelto dependerá de cómo se haya definido la función (en este caso, la suma).



Polimorfismo (I)

- Estamos acostumbrados a ver el operador '+' como el operador suma, pero en algunos lenguajes puede usarse, p.e., como operador de concatenación.
  - *“Hola” + “mundo” ⇒ “Hola mundo”*
  - *3 + 2 ⇒ 5*
  - *“Hola” + 3 ⇒ “Hola 3”*
- Esto es lo que se conoce como polimorfismo.
- LISP admite esta característica.

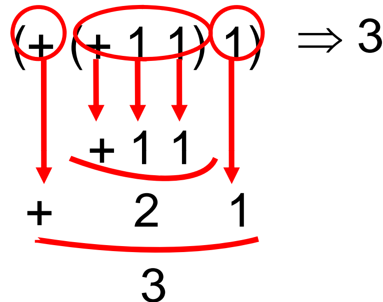


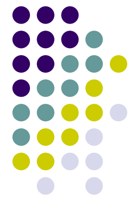
Polimorfismo (II)

- Ejemplo. Tenemos esta lista de tres elementos:

  (+ (+ 1 1) 1)

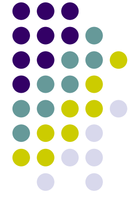
  - 1º un símbolo
  - 2º una lista
  - 3º un número
- LISP evalúa de este modo:





Transparencia Referencial (I)

- En LISP no existe el concepto de asignación (como sí ocurría en PROLOG).
- **Transparencia Referencial**: *"cuando una expresión* ***e*** *del lenguaje es sustituida por el valor* ***v****, y* ***v*** *es el resultado de evaluar* ***e****, la semántica del programa no se altera".*
- LISP aplica **transparencia referencial**, según la cual un objeto es o bien una incógnita o bien algo conocido. En cuanto una incógnita toma un valor, paso a ser algo conocido, que ya no puede cambiar.

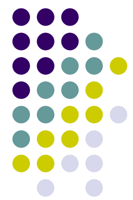


Transparencia Referencial (II)

- Lo más similar que existe en LISP para la asignación es

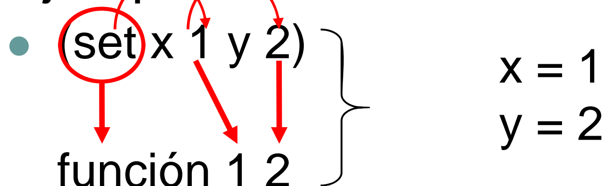
  (set _ _) (no válido en todas las implementaciones)

- Cuando LISP encuentra un *'set'*, evalúa sólo lo que encuentra en la tercera posición. El resultado es que el elemento en la posición par toma el valor del elemento en la posición impar.
- Es responsabilidad del programador el que una variable tenga un valor asignado.



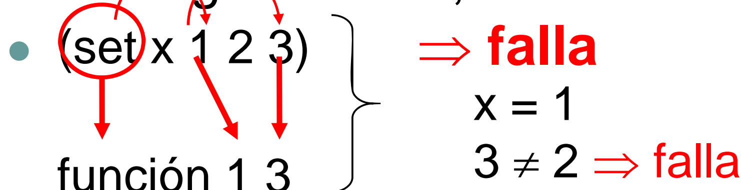
Transparencia Referencial (III)

- Ejemplo:



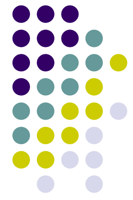
x = 1
y = 2

- Si en lugar de eso, tuviéramos:



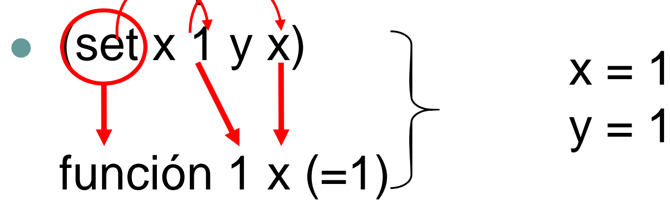
⇒ **falla**

x = 1
3 ≠ 2 ⇒ falla



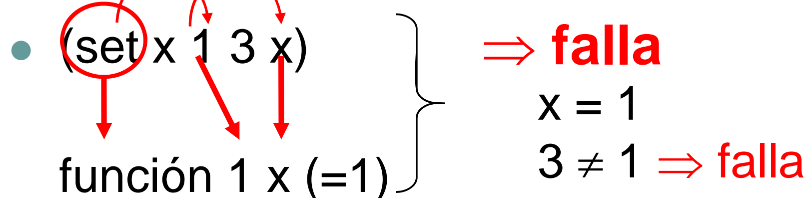
Transparencia Referencial (IV)

- Y si lo que tenemos es:
  - (set x 1 y x)



x = 1
y = 1

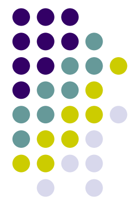
- O bien:
  - (set x 1 3 x)



$\Rightarrow$ **falla**

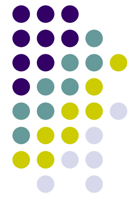
x = 1

$3 \neq 1 \Rightarrow$ falla



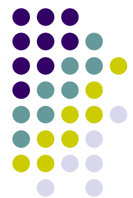
Transparencia Referencial (V)

- No todas las implementaciones de LISP aceptan ‘set’.
  - En Common LISP, se utiliza ‘setq’.
    - Ej.: *(setq a 3 b (sqrt 4))*
  - En SCHEME, se pueden utilizar *(define _ _)* o también *(set! _ _)*.
    - *Ojo: No aceptan asignaciones en cadena.*



Transparencia Referencial (VI)

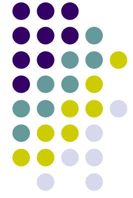
- Asignación de cadenas de caracteres a variables.
  - *(define x (+ 2 3))* $\Rightarrow$ *x = 5*
- Si lo que queremos es asignar toda la cadena a *'x'*:
  - *(define x '(+ 2 3))* $\Rightarrow$ *x = (+ 2 3)*
  - La cadena de la derecha, *(+ 2 3)*, puede evaluarse más adelante, si es necesario.
- Operador ' ó *quote*: devuelve la lista sin evaluarla.
  - *(+ 2 3)* $\Rightarrow$ *5*
  - *(quote (+ 2 3))* $\Rightarrow$ *(+ 2 3)*
  - *'(+ 2 3)* $\Rightarrow$ *(+ 2 3)*



Transparencia Referencial (VII)

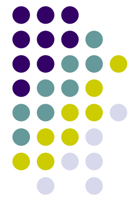
- Si tenemos una lista que no queremos evaluar pero que contiene variables, éstas se evaluarán según el contexto.
  - *(define x '(+ 2 y))* $\Rightarrow$ *x = (+ 2 y)*
  - En este caso dependerá de si *'y'* es algo conocido, del valor que tenga. Si *'y'* es una incógnita, al intentar evaluar más tarde *'x'*, fallará.
  - Va a permitir construir expresiones más complejas de manera dinámica.
- Por último, distinguiremos dos símbolos especiales:
  - *#T* $\Rightarrow$ *True*
  - *#F ó NIL ó ()* $\Rightarrow$ *False* (también representa una lista vacía)

Algunas propiedades de SCHEME

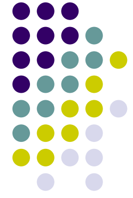


- Transparencia referencial
- Evaluación perezosa
- Polimorfismo
- Abstracción de datos
  - Es posible construir datos complejos a partir de datos más simples
- Reconocimiento de patrones
  - SCHEME es capaz de reconocer a partir de datos y expresiones
  - SCHEME es capaz de reconocer la función a la que se está llamando a partir de una llamada.
  - Reconoce un predicado o expresión y diferencia entre datos y símbolos de función.
- Funciones de alto orden
  - Funciones pueden ser argumentos de otras funciones.
- No existen tipos de datos como tales

Tema 2: LISP (SCHEME)



- **Introducción**
- **Representación de Datos**
- **Definición de Funciones**
- **Predicados**
- **Listas**

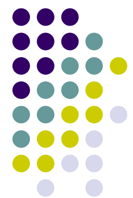


Representación de datos (I)

- Números: 3.15, 3, #B (Binarios), #D (Decimales), #O (Octales), #X (Hexadecimales)
  - Si se omite la base, tomará por defecto la decimal.
  - Ejemplo:

    #O-3.567

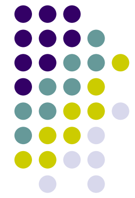
    Es el número -3.567 en base octal.
- Números racionales: 3/5.
  - Para escribirlos correctamente basta con no dejar espacios entre las cifras.
- Números complejos: 3+2i



Representación de datos (II)

- Literales: *'literal*
  - Son símbolos que no se evalúan.
  - Ejemplo:

    '(+ 3 2)
  - Cualquier cosa se convierte en literal cuando le añadimos el apóstrofe a la izquierda.
- Cadenas de caracteres: *"Hola mundo"*
- Booleanos:
  - *#T → True*
  - *#F → False*



Tema 2: LISP (SCHEME)

- **Introducción**
- **Representación de Datos**
- **Definición de Funciones**
- **Predicados**
- **Listas**

Definición de funciones (I)

- SCHEME es un paradigma dentro de la programación funcional. Todo en SCHEME son funciones.
- Algunas palabras reservadas son:
  - *DEFINE*
  - *LAMBDA*
- Para definir una función:

```
(DEFINE Nombre
    (LAMBDA (A B)
      (
      ...
      )))
```

Definición de funciones (II)

```
(DEFINE Nombre
   (LAMBDA (A B)
     (
     ...
     )))
```

- Todo en SCHEME se define mediante listas.
  - Al definir una función, el segundo argumento de la lista será el nombre de la función.
  - El tercer argumento es otra **lista**, en la que definimos los argumentos (en una **lista**) y cuyo último argumento es otra **lista** que ya contiene el cuerpo de la función.

Definición de funciones (III)

- Ejemplo: Definición de la función ‘suma’ con dos argumentos.
  - (SUMA A B)

```
(DEFINE SUMA
  (LAMBDA (A B)
          (+ A B)))
```

- Ejemplo: Función que calcula el cuadrado de un número.
  - (CUADRADO A)

```
(DEFINE CUADRADO
  (LAMBDA (A)
          (* A A)))
```

- Las operaciones utilizadas, ‘+’ y ‘*’, deberán estar previamente implementadas (suele estar implementadas internamente).

Definición de funciones (IV)

- Para sumar más de dos valores:

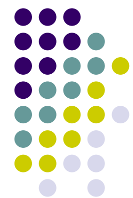
  (+ (+ A B) C)

  - En SCHEME existe ya implementada un función para sumar cualquier número de argumentos.
- Ejemplo:
  - *(SUMA_CUADRADOS X Y)* ⇒ $X^2 + Y^2$

```
(DEFINE SUMA_CUADRADOS
  (LAMBDA (X Y)
          (SUMA (CUADRADO X) (CUADRADO Y))))
```

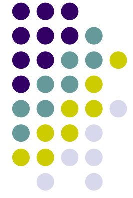
O bien,

```
(DEFINE SUMA_CUADRADOS
  (LAMBDA (X Y)
          (+ (* X X) (* Y Y))))
```



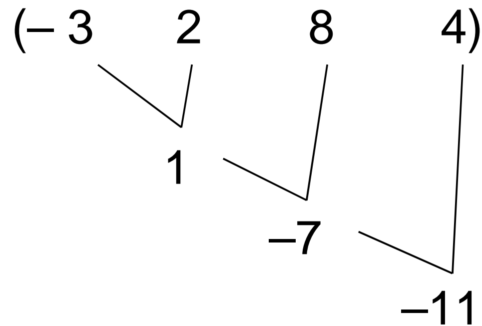
Funciones Aritméticas (I)

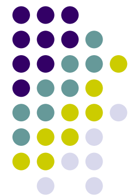
- Por fortuna, SCHEME ya trae implementadas una serie de funciones aritméticas genéricas:
  - $(+\ I_1\ I_2\ I_3\ \ldots\ I_n)$
  - $(-\ I_1\ I_2\ I_3\ \ldots\ I_n)$



Funciones Aritméticas (II)

- Ejemplo:





Funciones Aritméticas (III)

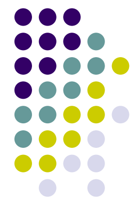
- Otras operaciones aritméticas:
  - (* I_1 I_2 I_3 ... I_n)

  - (/ I_1 I_2 I_3 ... I_n)

- Con ésta última hemos de tener cuidado, ya que si el número de argumentos es alto e I_1 es un valor pequeño, la función convergerá a 0. También hay que vigilar el que ninguno de los argumentos valga 0.

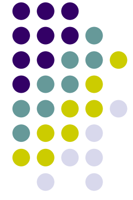
Funciones Aritméticas (IV)

- *(QUOTIENT I_1 I_2)*
  - Devuelve el *cociente* de la división entera
- *(MODULO I_1 I_2)*
  - Devuelve el *resto* de la división entera
- *(GCD I_1 … I_2)*
  - Calcula el *máximo común divisor* de los n elementos
- *(LCM I_1 … I_2)*
  - Calcula el *mínimo común múltiplo* de los n elementos



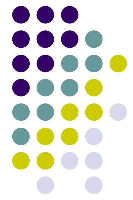
Operaciones Relacionales

- Existe una serie de funciones genéricas también para implementar las operaciones relacionales:
- *(< n_1 n_2)*
  - Devuelve *True (#T)* si el valor n_1 es menor que n_2.
- *(> n_1 n_2)*
  - Devuelve *True (#T)* si el valor n_1 es mayor que n_2.
- *(= n_1 n_2)*
  - Devuelve *True (#T)* si ambos valores son iguales.
- *(<= n_1 n_2)*
  - Devuelve *True (#T)* si el valor n_1 es menor o igual que n_2.
- *(>= n_1 n_2)*
  - Devuelve *True (#T)* si el valor n_1 es mayor o igual que n_2.
- Por convenio, todas aquellas funciones en SCHEME que devuelven un booleano (*#T* ó *#F*) se denominan *predicados*.



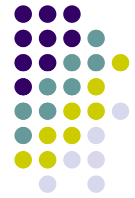
Operadores Lógicos

- SCHEME también implementa operadores lógicos:
  - *(AND I_1 I_2 I_3 ... I_n)*
    - Los argumentos I_i pueden ser constantes booleanas o predicados. En el momento en que uno de ellos sea falso, el resultado de toda la operación será también falso.
  - *(OR I_1 I_2 I_3 ... I_n)*
    - Los argumentos I_i pueden ser constantes booleanas o predicados. Devuelve *#T* en cuanto evalúa un argumento que sea cierto.
  - *(NOT E)*
    - Niega la constante booleana o predicado que tenga como argumento:
      - *NOT #F ⇒ #T*
      - *NOT #T ⇒ #F*



Funciones Numéricas (I)

- Implementación de funciones numéricas en SCHEME:
  - *(ABS n)*
    - Devuelve el valor absoluto del número.
  - *(COS n)*
    - Devuelve el coseno del número.
  - *(SIN n)*
    - Devuelve el seno del número.
  - *(TAN n)*
    - Devuelve la tangente del número.



Funciones Numéricas (II)

- También:
  - *(EXP n)*
    - Devuelve el resultado de calcular e^n.
  - *(LOG a)*
    - Devuelve el valor de logaritmo natural de *a*, *ln a*.
  - *(SQRT n)*
    - Devuelve la raíz cuadrada del número.

Funciones Numéricas (III)

- Otras funciones de interés:
  - *(RANDOM n)*
    - Devuelve un número pseudo aleatorio entre *0* y *n – 1*.
  - $(MIN\ l_1\ l_2\ l_3\ ...\ l_n)$
    - Devuelve el menor valor de entre todos los argumentos.
  - $(MAX\ l_1\ l_2\ l_3\ ...\ l_n)$
    - Devuelve el mayor valor de entre todos los argumentos.

Funciones Numéricas (IV)

- Funciones de conversión numérica en SCHEME:
  - *(ROUND n)* $\rightarrow$ Redondeo al alza.
    - Ejemplo:

      (ROUND 3.6) $\Rightarrow$ *4*

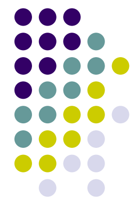
      (ROUND 3.2) $\Rightarrow$ *3*

      (ROUND 3.5) $\Rightarrow$ *4*
  - *(TRUNCATE n)* $\rightarrow$ Redondea a la baja.
    - Ejemplo:

      (TRUNCATE 3.6) $\Rightarrow$ *3*

      (TRUNCATE 3.2) $\Rightarrow$ *3*

      (TRUNCATE 3.5) $\Rightarrow$ *3*



Funciones Numéricas (V)

- Funciones de conversión numérica en SCHEME:
  - *(FLOOR n)* $\rightarrow$ Devuelve el mayor entero no superior a n.
    - Ejemplo:

      (FLOOR 3.5) $\Rightarrow$ *3*

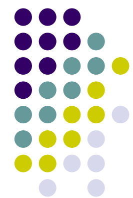
      (FLOOR 3) $\Rightarrow$ *3*
  - *(CEILING n)* $\rightarrow$ Devuelve el menor entero no inferior a n.
    - Ejemplo:

      (CEILING 3.6) $\Rightarrow$ *4*

      (CEILING 3) $\Rightarrow$ *3*

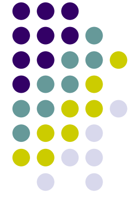
Tema 2: LISP (SCHEME)

- Introducción
- Representación de Datos
- Definición de Funciones
- Predicados
- Listas



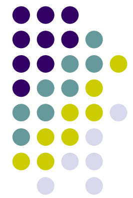
Predicados (I)

- Existe en SCHEME una serie de predicados para comprobar la validez de los datos.
  - *(NUMBER? x)*
    - Devuelve True si 'x' es un número.
  - *(INTEGER? x)*
    - Devuelve True si 'x' es un número y es entero.
  - *(REAL? x)*
    - Devuelve True si 'x' es un número real.
  - *(RATIONAL? x)*
    - Devuelve True si 'x' es un racional.
  - *(COMPLEX? x)*
    - Devuelve True si *'x'* es un número complejo.



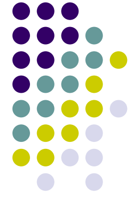
Predicados (II)

- Más predicados para comprobar la validez de los datos:
  - *(EVEN? x)*
    - Devuelve True si *'x'* es par, y False en caso contrario.
  - *(ODD? x)*
    - Devolverá True si *'x'* es impar, y False en caso contrario.
  - *(POSITIVE? x)*
    - Números positivos. Devuelve True si *'x'* está por encima de 0.
  - *(NEGATIVE? x)*
    - Números negativos. Devolverá True si *'x'* está por debajo de 0.
  - *(ZERO? x)*
    - Si *'x'* es cero, devuelve True. En otro caso, devolverá False.



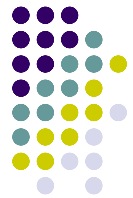
Operadores condicionales (I)

- *(IF <P> <E_1> <E_2>)*
  - Implementa el operador condicional *'if'*
    - <*P*> es un predicado.
    - Si el resultado de evaluar <*P*> es cierto, evalúa <E_1> y devuelve el resultado de evaluar ésta. En otro caso, evalúa <E_2> y devuelve lo que ésta valga.



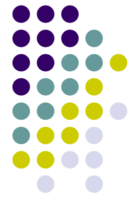
Operadores condicionales (II)

- *(COND (<P_1> <E_1>)*
  (<P_2> <E_2>)
  ...
  (<P_n> <E_n>)
  (ELSE <E>))
  - Evalúa <P_1> en primer lugar, y si es cierto, evalúa <E_1> y devuelve lo que valga <E_1>.
    Si <P_1> era falso, entonces evaluará <P_2>. Si <P_2> es cierto, evaluará <E_2> y devolverá lo que valga ésta.
    Y así sucesivamente, si el predicado anterior es falso, seguirá evaluando hasta encontrar un <P_i> que sea cierto.
    En caso de que todos los <P_i> sean falsos, devuelve el resultado de evaluar <*E*>.



Algunos Ejemplos (I)

- Función *factorial*
  $0! \rightarrow 1$
  $1! \rightarrow 1$
  ...
  $n! \rightarrow n*(n - 1)!$
- Utilizaremos la siguiente lista para implementar la función:
  -
```
(DEFINE FACTORIAL
  (LAMBDA (N)
      (IF (< N 2)
          1
          (* N (FACTORIAL (- N 1))))))
```



Algunos Ejemplos (II)

- Función de *Fibonacci*

  $F(0) \rightarrow 0$

  $F(1) \rightarrow 1$

  ...

  $F(n) \rightarrow F(n-1)+F(n-2)$

  - *(DEFINE FIBO*

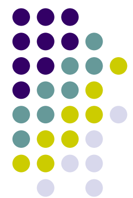
    (LAMBDA (N)

    (COND

    ((= N 0) 0)

    ((= N 1) 1)

    (ELSE (+ (FIBO (– N 1)) (FIBO (– N 2)))))))



Algunos Ejemplos (III)

- Función de *Fibonacci*. Otra forma:
  - *(DEFINE FIBO*

    (LAMBDA (N)

    (COND

    ((ZERO? N) 0)

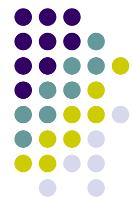
    ((= N 1) 1)

    (ELSE (+ (FIBO (– N 1)) (FIBO (– N 2)))))))

Algunos Ejemplos (IV)

- Función de *Fibonacci.* Una última posibilidad a contemplar sería aquel caso en el que diéramos como argumento un valor negativo.
  - La función de *Fibonacci* no está definida para valores negativos y por tanto debería fallar.
  - Debemos incluir otra condición.
  - *(DEFINE FIBO*

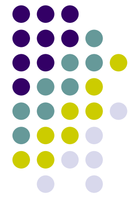
```
(LAMBDA (N)
  (COND
        ((NEGATIVE? N) #F)
        ((ZERO? N) 0)
        ((= N 1) 1)
        (ELSE (+ (FIBO (– N 1)) (FIBO (– N 2)))))))
```



Algunos Ejemplos (V)

- *TAU*: Número de divisores de N
  - TAU(N) = #{ d: d/N, d>0, N>0}
  - TAU(6) = #{1, 2, 3, 6} $\Rightarrow$ TAU(6) = 4
    - # $\rightarrow$ Cardinalidad (número de elementos) del conjunto.
- *(DEFINE TAU*

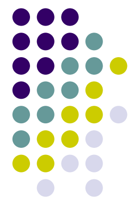
```
    (LAMBDA (N)
          (IF (POSITIVE? N)
                (+ 1 (siguientes_TAU N 2))
                "Error")))
```



Algunos Ejemplos (VI)

- Función auxiliar ‘siguientes_TAU’:
  - *(DEFINE siguientes_TAU*

```
(DEFINE siguientes_TAU
(LAMBDA (N D)
  (IF (<= D N)
          (IF (= (MODULO N D) 0)
                  (+ 1 (siguientes_TAU N (+ 1 D)))
                  (siguientes_TAU N (+ 1 D)))
          0)))
```



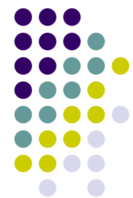
Algunos Ejemplos (VII)

- *SIGMA*: Suma de los divisores de N

```
(DEFINE SIGMA
(LAMBDA (N)
  (IF (POSITIVE? N)
          (+ 1 (siguientes_SIGMA N 2))
          "Error")))
```

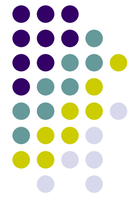
Algunos Ejemplos (VIII)

- Función auxiliar *'siguientes_SIGMA'*:
  -
```
(DEFINE siguientes_SIGMA
(LAMBDA (N D)
  (IF (<= D N)
        (IF (= (MODULO N D) 0)
              (+ D (siguientes_SIGMA N (+ 1 D)))
              (siguientes_SIGMA N (+ 1 D)))
        0)))
```
- Otra manera de hacer notar el error: Como ambas funciones están definidas sólo para números positivos, podemos hacer que devuelva –1, aunque con "Error" también nos sirve.



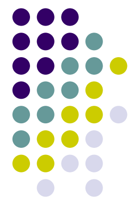
Tema 2: LISP (SCHEME)

- **Introducción**
- **Representación de Datos**
- **Definición de Funciones**
- **Predicados**
- **Listas**



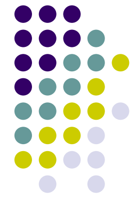
Listas (I)

- En SCHEME, podemos construir un **par ordenado** mediante el siguiente predicado:
  - *(CONS A B) → (A . B)*
  - (A . B) es un **par ordenado** porque no se puede cambiar, es decir, se respeta el orden de los argumentos.
- Se puede usar para construir listas:
  - *(CONS A B)*
    - A → Primer elemento de la lista
    - B → Resto de la lista (debe ser una lista)



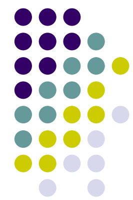
Listas (II)

- Ejemplo:
  - *(CONS 1 '(2 3)) → (1 2 3)*
- Se puede anidar:
  - *(CONS 1 (CONS 2 '(3))) → (1 2 3)*
- También tenemos la siguiente función:
  - *(LIST 1 5 3) → (1 5 3)*
  - Esta función lista todo aquello que se le pase como argumento.



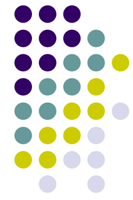
Listas (III)

- Otras dos funciones muy prácticas son:
  - *(CAR lista)*
    - *(CAR '(1 2 3)) → 1*
  - *(CDR lista)*
    - *(CDR '(1 2 3)) → (2 3)*
  - **CAR** devuelve el primer argumento (cabecera) de la lista que se le pasa, y **CDR** devuelve una lista con el resto de argumentos de la lista que se le pasa. Se puede decir que actúan de forma similar al operador barra (|) de PROLOG.



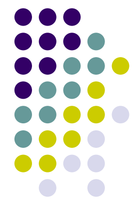
Listas (IV)

- Ejemplos:
  - *(CAR (CONS A B)) → A*
  - *(CDR (CONS A B)) → B*
  - *(CAR (CONS 1 (CONS 2 '(3)))) → 1*
  - *(CDR (CONS 1 (CONS 2 '(3)))) → (2 3)*
  - *(CAR (LIST 1 5 3)) → 1*
  - *(CDR (LIST 1 5 3)) → (5 3)*



Operaciones sobre listas (I)

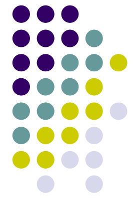
- Ejemplo: Definir una función para calcular la longitud de una lista.
  - En PROLOG se hacía así:
    - *longitud([ ], 0).*
    - *longitud([_ | Cola], N) :- longitud(Cola, T), N is T+1.*



Operaciones sobre listas (II)

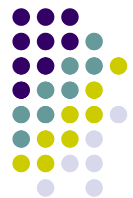
- Ejemplo: Definir una función para calcular la longitud de una lista.
  - En SCHEME:
    - ```
      (DEFINE LONG
      (LAMBDA (L)
              (IF (NULL? L)
                      0
                      (+ 1 (LONG (CDR L))))))
      ```

      ```



# Operaciones sobre listas (III)

- Consideraciones:
  - Predicado *(NULL? Lista)*
    - Devuelve #T si la lista está vacía
  - Lista vacía
    - En LISP, se nota como *NIL* ó *( )*
      - En SCHEME, usar *( )*
  - Función *(LENGTH Lista)*
    - Devuelve la longitud de la lista
    - Ya viene implementada en SCHEME



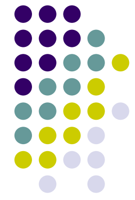
# Operaciones sobre listas (IV)

- Ejemplo: Encadenar dos listas
  - *(DEFINE ENCADENA*

```
(DEFINE ENCADENA
(LAMBDA (L1 L2)
  (IF (NULL? L1)
          L2
          (CONS (CAR L1)
                  (ENCADENA (CDR L1) L2)
          ))))
```

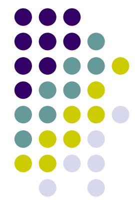
  - En SCHEME, tenemos la función
    - *(APPEND* $lista_1$ … $lista_n$*)*

Consideración sobre las funciones *'car'* y *'cdr'* (I)



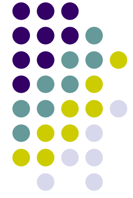
- En SCHEME hay definidas unas funciones especiales para facilitar el uso de *'car'* y *'cdr'*, con la siguiente notación:
  - *CxxxxR*, donde *'x'* puede ser *'a'* ó *'d'*.

Consideración sobre las funciones *'car'* y *'cdr'* (II)



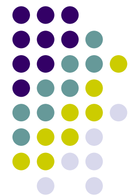
- Las siguientes operaciones se podrían simplificar bastante:
  - *(car (car '((a b) (c d)))) →*
    (caar '((a b) (c d))) → a
  - *(cdr (car '((a b) (c d)))) →*
    (cdar '((a b) (c d))) → (b)
  - *(car (cdr (car (cdr '((a b) (c d)))))) →*
    (cadadr '((a b) (c d))) → d

Consideración sobre las funciones *'car'* y *'cdr'* (III)

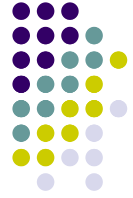


- Ejemplo: Definición manual de la función *'cadr'*
  - *(DEFINE cadr*
    (LAMBDA (X)
    (car (cdr X))
)
)
- *(cadr '((a b) (c d)))* $\rightarrow$ *(c d)*

Funciones de Alto Orden (I)



- Hasta ahora sólo hemos utilizado *'DEFINE'* para definir funciones de orden bajo, es decir, funciones que tiene argumentos que no son funciones.
- SCHEME también admite **funciones de orden alto**, que son aquéllas que aceptan funciones como parámetros o que devuelven funciones a la salida.
- Ejemplo: Definir la función que calcula la suma de los cuadrados de dos números.
  - Función 'cuadrado':
    (DEFINE CUADRADO
    (LAMBDA (A)
    (A A)))*



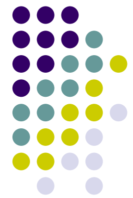
Funciones de Alto Orden (II)

- Ejemplo: Definir la función que calcula la suma de los cuadrados de dos números.
  - Suma de cuadrados:

```
(DEFINE SUMA_CUADRADOS
 (LAMBDA (X Y)
        (+ (CUADRADO X) (CUADRADO Y))
 ))
```

  - Otra forma:

```
(DEFINE SUMA_CUADRADOS
 (LAMBDA (X Y)
        (+ (* X X) (* Y Y))
 ))
```



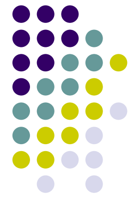
Funciones de Alto Orden (III)

- Pero, ¿y si ahora quisiéramos definir la diferencia de los cuadrados de dos números? ¿Qué modificaciones habría que hacer?
  - (DEFINE RESTA_CUADRADOS

```
 (LAMBDA (X Y)
        (– (CUADRADO X) (CUADRADO Y))
 ))
```

- En principio, bastaría con sustituir el operador *'suma'* por el de *'resta'*. Y así con cualquier operación aritmética sobre los cuadrados de dos números.
  - Pero hemos de hacer esas modificaciones manualmente, y una a una.
- Podríamos definir una función de alto orden que además de recibir los dos argumentos *'X'* e *'Y'*, recibiera el operador necesario en cada momento.

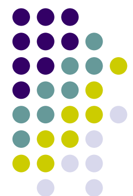
Funciones de Alto Orden (IV)



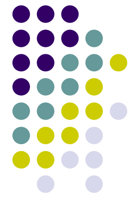
- Ejemplo:
  - *(DEFINE CONSTRUCTOR*
    (LAMBDA (OP)
    (LAMBDA (X Y)
    (OP (CUADRADO X) (CUADRADO Y))
    )))
- Hemos definido una función que aplica el operador *'OP'* a *'X'* e *'Y'*. Modo de uso:
  - *(DEFINE suma_cuadrado (CONSTRUCTOR +))*

  o también,
  - *(DEFINE resta_cuadrado (CONSTRUCTOR –))*

Funciones de Alto Orden (V)

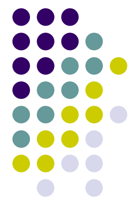


- Los operadores tipo *'CxxxxR'* de SCHEME se definen realmente usando funciones de alto orden, así:
  - *(DEFINE compose*
    (LAMBDA (p1 p2)
    (LAMBDA (X)
    (p1 (p2 X))
    )))



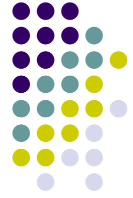
Funciones de Alto Orden (VI)

- Ejemplos:
  - Definición de *'caar'*:
    (DEFINE caar (compose car car))
  - *'cdar'*:
    (DEFINE cdar (compose cdr car))



Funciones de Alto Orden (VII)

- Y así con todas las posibles combinaciones, para ir subiendo de nivel. Por ejemplo, *'caaaar'*:
  - *(DEFINE caaaar (compose caar caar))*
- *'caaddar'*:
  - *(DEFINE caaddar (compose caaddr car))*
- Las posibilidades son muchas. En SCHEME tenemos implementados estos operadores hasta el cuarto nivel (es decir, hasta cuatro *'x'*, *CxxxxR*).
  - Estas mismas y todas de nivel superior que necesitemos podemos definirlas de la forma en que hemos visto.

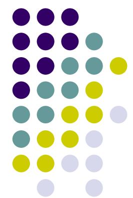


Funciones de Alto Orden (VIII)

- (APPLY arg1 arg2)
  - Devuelve el resultado de aplicar el primer argumento a los elementos en su segundo argumento.
  - *(APPLY + ‘(7 5 3))*

    ⇒ 15
  - *(APPLY max ‘(3 7 2 9))*

    ⇒ 9



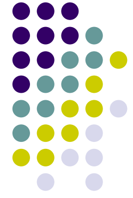
Funciones de Alto Orden (IX)

- (MAP arg1 arg2)
  - Devuelve una lista con el resultado de aplicar el primer argumento a cada uno de los elementos del segundo argumento.
  - *(MAP odd? ‘(2 3 4 5 6))*

    ⇒ (#F #T #F #T #F)
  - *(MAP (* 2) ‘(2 3 4 5 6))*

    ⇒ (4 6 8 10 12)

Funciones de Alto Orden (y X)



- (MAP arg1 arg2)
  - *(DEFINE triple*
    (LAMBDA (x)
    (3 x)))*
  - *(MAP triple '(2 3 4 5 6))*
    ⇒ (6 9 12 15 18)

Programación Declarativa

Tema 2: LISP (SCHEME)