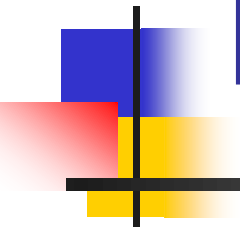
Programación Automática

MÁSTER EN CIENCIA Y TECNOLOGÍA INFORMÁTICA

Ricardo Aler Mur

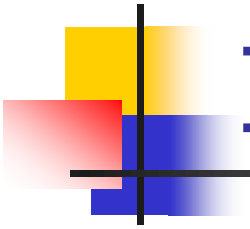


Programación Automática

Ricardo Aler Mur

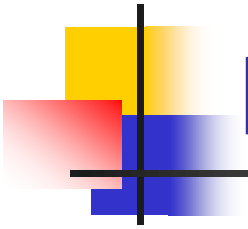
Universidad Carlos III de Madrid

http://www.uc3m.es/uc3m/dpto/INF/aler



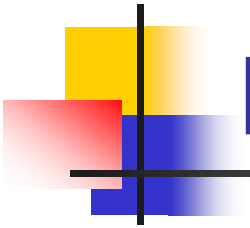
Índice

1. Programación Automática:
   1. Desde el punto de vista de la Ingeniería del Software
   2. Desde el punto de vista de la IA
2. Programación Automática Deductiva
3. Programación Automática Inductiva
   1. Síntesis de programas LISP



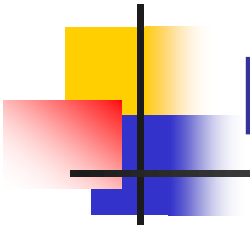
Programación Automática (I.S.)

- Desde el punto de vista de la Ingeniería del Software: mejora de la productividad de un programador
- Realmente es semi-automática



Programación Automática (I.S.)

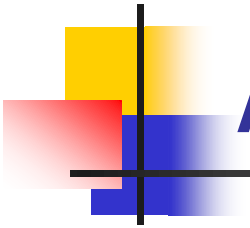
- Programación Generativa (Generative Programming), diferentes maneras de reutilizar código:
  - Component-based software engineering
  - Product family engineering.
- Optimización automática de código (compiladores)
  - Reglas de reescritura: $x^* 1 \rightarrow x$
  - memoization, orden sentencias, recursividad cola, ...
- Programación Gráfica (conexión de componentes)
- Wizards/Asistentes de programación (Apprentice)
- Generación de programas científicos (Kant)
- Lenguajes de muy alto nivel (SETL: conjuntos, SML)



Programación Automática (I.A.)

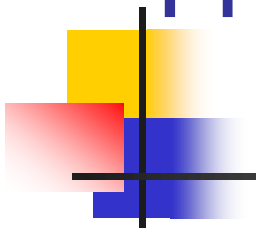
- Desde el punto de vista de la Inteligencia Artificial: que un ordenador sea capaz de generar código automática o semi-automáticamente
- Se pone más énfasis en que sea el ordenador el que genere el código
- Se usan técnicas de Inteligencia Artificial
- Mejor llamado: síntesis de programas (program shynthesis)

Tipos de Programación Automática



- **Deductiva:** Generar un programa a partir de una especificación a alto nivel (que sea más sencilla que el programa).
- **Inductiva:** Generación de programas a partir de ejemplos de uso
- **Deductiva-inductiva** (híbrida, multiestrategia, ...)

Programación Automática Deductiva



- Sistemas Transformacionales y Deductivos (especificaciones en lenguajes formales, lógica de predicados: Refine, KIDS; Manna & Waldinger 92)

¿Qué es la deducción?

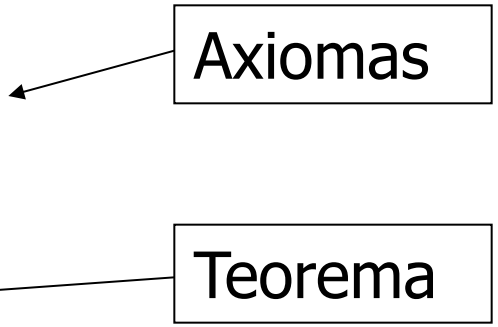
SI

1.∀X Hombre(X) → Mortal(X)

2. Hombre(aristoteles)

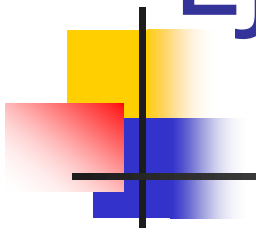
ENTONCES (por Modus Ponens)

3. Mortal(aristoteles)

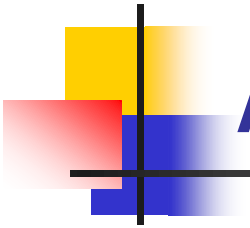


- Se parte de premisas o axiomas (1 y 2) y se alcanza una conclusión o teorema mediante reglas de inferencia (*modus ponens*, en este caso)
- Otra forma de verlo: se van transformando los axiomas mediante reglas de inferencia
- Normalmente, una deducción consiste de una cadena de Modus Ponens

Ejemplo: Amphion [Stickel, 95]

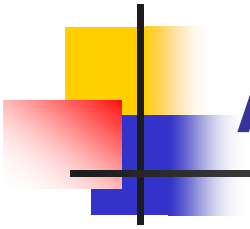


- "Deductive Composition of Astronomical Software from Subroutine Libraries"
- Dominio astronómico (sistema solar)
- Parte de una librería en geometría solar (SPICELIB), en lugar de instrucciones primitivas de un lenguaje de programación
- En general, existen muchas librerías de subrutinas, pero en muchas ocasiones los programadores prefieren reprogramar a reutilizar



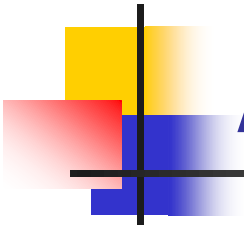
Amphion

- Interface gráfico para escribir especificaciones
- Ejemplo de especificación: “¿dónde está la sombra de la luna Io?”
- Se convierten a un teorema en lógica de predicados
- Que es demostrado por un demostrador automático de teoremas (SNARK [Manna & Waldinger, 92])
- La demostración es utilizada para componer un programa FORTRAN-77 a partir de la librería de subrutinas SPICELIB
- Especificación -> teorema -> demostración -> programa



Amphion: sistema deductivo

- ¿Qué son los axiomas o premisas?
  - Las descripciones en lógica de predicados de las subrutinas de SPICELIB
- ¿Qué es el teorema?
  - La pregunta que se hace al sistema (“¿dónde está la sombra de la luna Io?”)
  - $\exists y$ *es-punto-de-sombra(y)*
  - Queremos que el sistema encuentre un programa que de respuesta a esa pregunta



Amphion

- SNARK demuestra teoremas en lógica de primer orden (parecido a PROLOG)
- Son pruebas constructivas: demostrar que algo existe implica tener un método para calcular ese algo
- *Ej:*$\exists x\ \exists y\ P(x,y)$ encuentra un *y* que satisface el teorema y un método para calcular el *y*

Ejemplo PROLOG

AXIOMAS

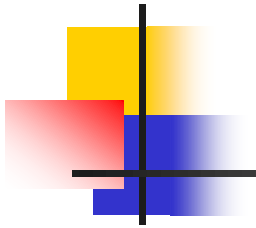
```
padre(juan,jose).
padre(juan,maria).
padre(pedro,juan).
padre(jose,luis).
```

```
padre(X,Y) => ancestro(X,Y)

Ancestro(X,Z) AND padre(Z,Y)
=>                    ancestro(X,Y)
```

```
?-padre(juan,jose).
yes
?-padre(X,jose).        X=juan
yes
?-ancestro(pedro,jose).
yes
?-ancestro(X,luis).
X=jose, X=juan
```

Amphion. El dominio astronómico



- 200 axiomas que describen las subrutinas y la geometría y cinemática del espacio: librería SPACELIB
- Los elementos de la geometría son **puntos** y **líneas**
- **Puntos** (o eventos): coordenada espaciotemporal (x,y,z,t)
- **Líneas:** *Lightlike?(e1,e2)*: cierto si un fotón puede ir de e1 a e2
- Ejemplos de rutinas de la librería:
  - *Ephemeris-object-and-time-to-event(o,t)*: función que devuelve un evento cuya posición es la de un objeto del sistema solar en un momento determinado
  - A-sent(o,d,ta): devuelve el momento en el que debe partir un fotón de *o*, para llegar a *d*, en un tiempo *ta*

Amphion. Ejemplo de axioma

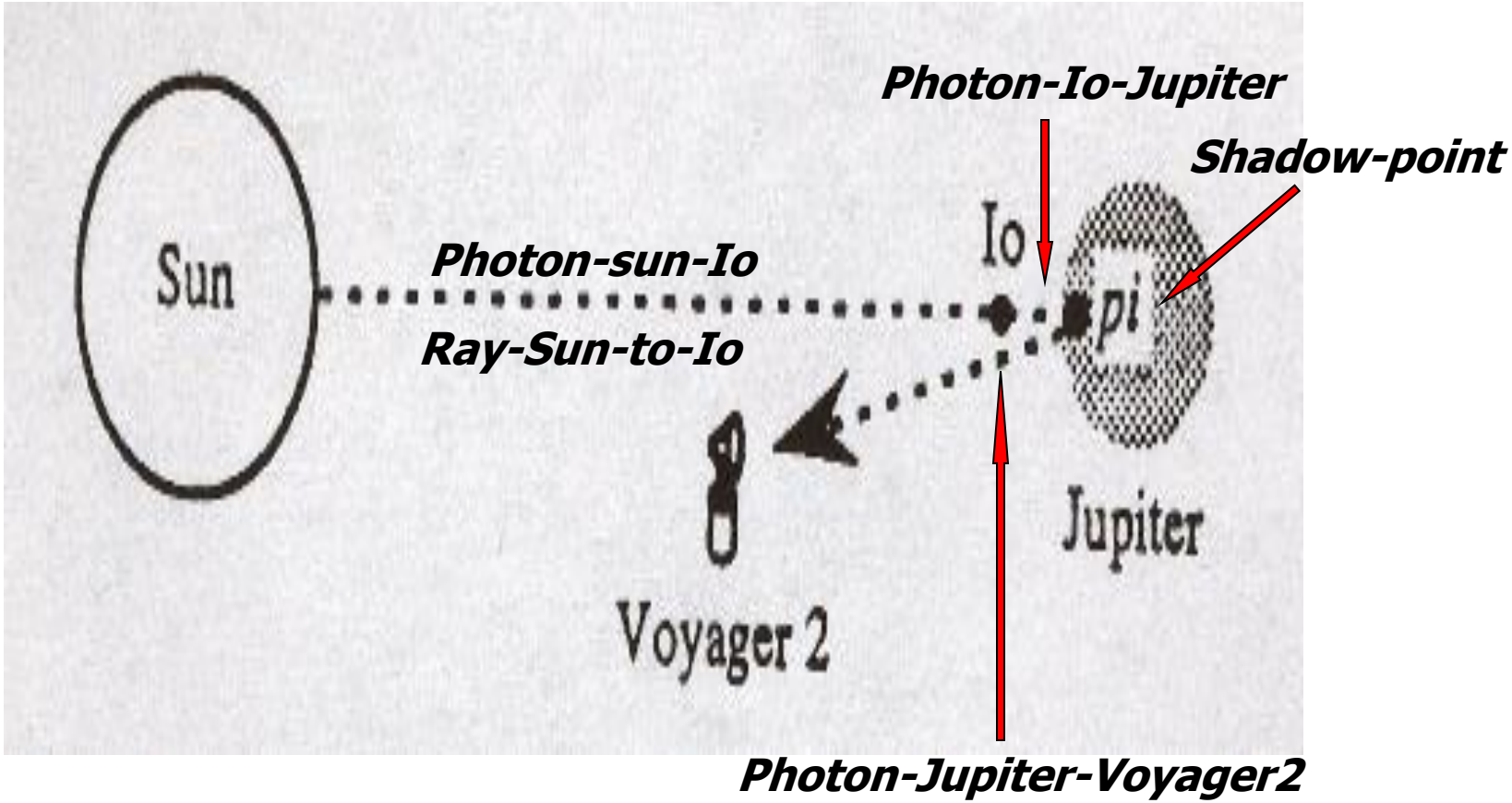
```
(all (o d ta)
  (lightlike?
    (ephemeris-object-and-time-to-event
      o (a-sent o d ta))
    (ephemeris-object-and-time-to-event d ta)))
```

Un fotón puede partir de *o* en el momento *a-sent(o,d,ta)* y llegar al objeto *d* en el momento *ta*

∀o,d,ta {lightlike?(eph-object(o, a-sent(o,d,ta)),

eph-object(d,ta)) }

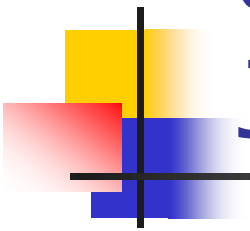
¿Dónde está la sombra de Io en Júpiter?



Amphion. Sombra de Io

- *Photon-sun-Io*: Fotón que sale del Sol y llega a Io (dos eventos)
- *Photon-Io-Jupiter*
- *Photon-Jupiter-Voyager2*
- *Ray-Sun-to-Io*: Rayo que sale del Sol, pasa por Io y atraviesa la superficie de Júpiter (*Júpiter-Ellipsoid*)
- *Shadow-Point*: punto de sombra sobre Júpiter

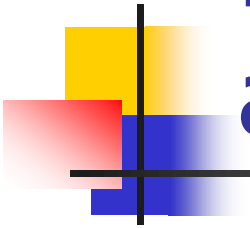
¿Dónde está la sombra de Io en Júpiter?. Primer intento



- Rayo= rayo(Sol, IO)
- Superficie = superficie(Jupiter)
- Sombra = interseccion(Rayo,Superficie)

- Problema: el Sol, Jupiter, etc, se mueven. ¿De qué instante de tiempo estamos hablando?

Resolución: propagación hacia atrás



- La manera de resolver el problema es calcular el momento en el que el fotón tiene que salir de Jupiter (*time-Jupiter*) para llegar al Voyager (*time-Voyager-2*)
- Después hay que calcular el momento en el que tiene que salir de Io (*time-Io*) para llegar a Júpiter
- Lo mismo con el Sol (*time-Sun*)
- Nótese que el problema es complejo, porque los objetos del Sistema Solar se están moviendo constantemente, pero la rutina A-sent se encarga de ello

¿Dónde está la sombra de Io en Júpiter?. Segundo intento

- TJUP= A-sent(Jupiter, V2, TV2)
- TIO= A-sent(IO, Jupiter, TJUP)
- TSOL= A-sent(Sol, IO, TIO)

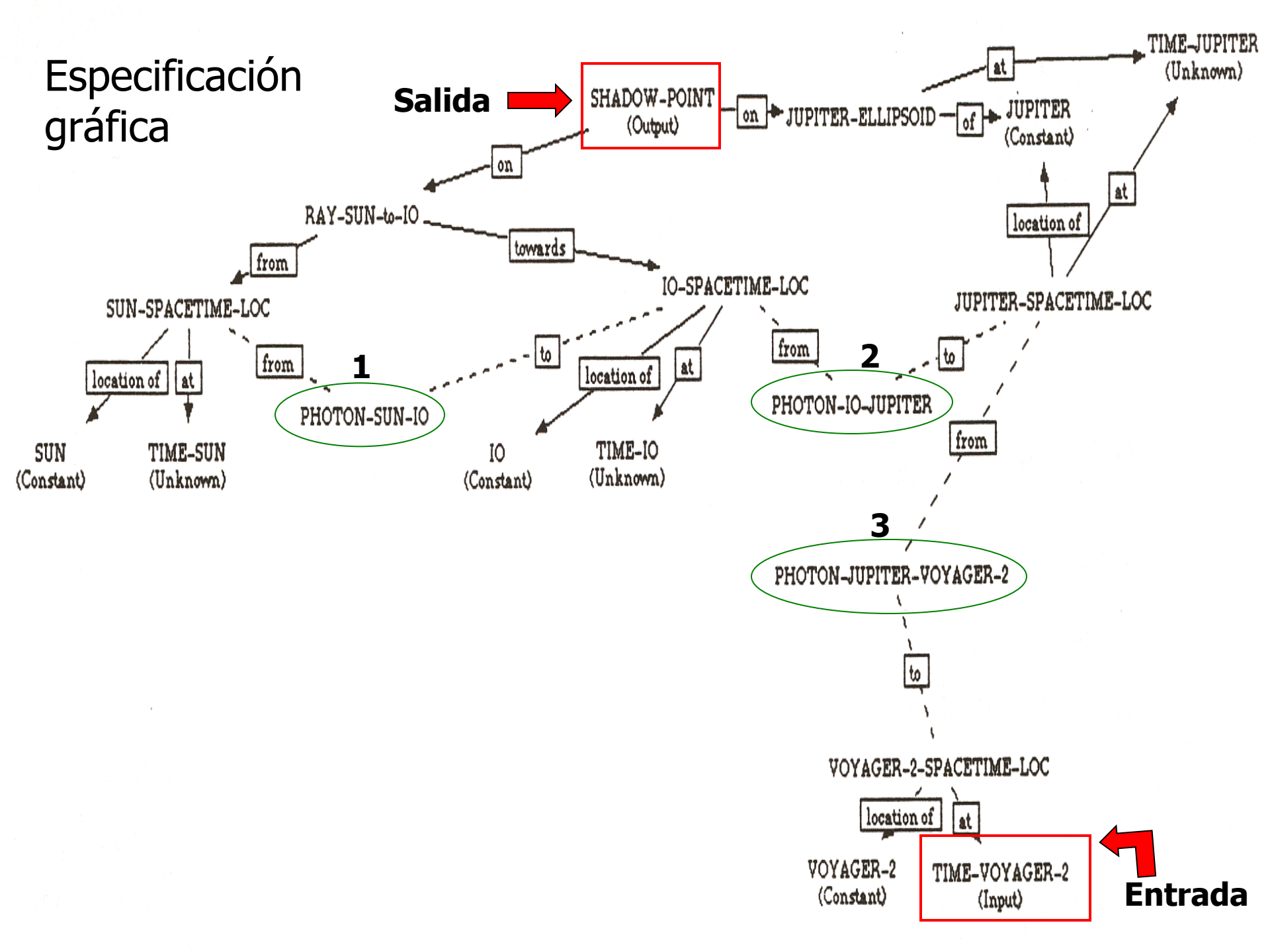
- etSOL = eph-object(eph-object(SOL,TSOL))
- etIO = eph-object(eph-object(IO,TIO))
- etJUP = eph-object(eph-object(Jupiter,TJUP))
- etV2 = eph-object(eph-object(Jupiter,TV2))

- Rayo= rayo(etSOL,etIO)
- Superficie = superficie(etJUP)
- Sombra = interseccion(Rayo,Superficie)

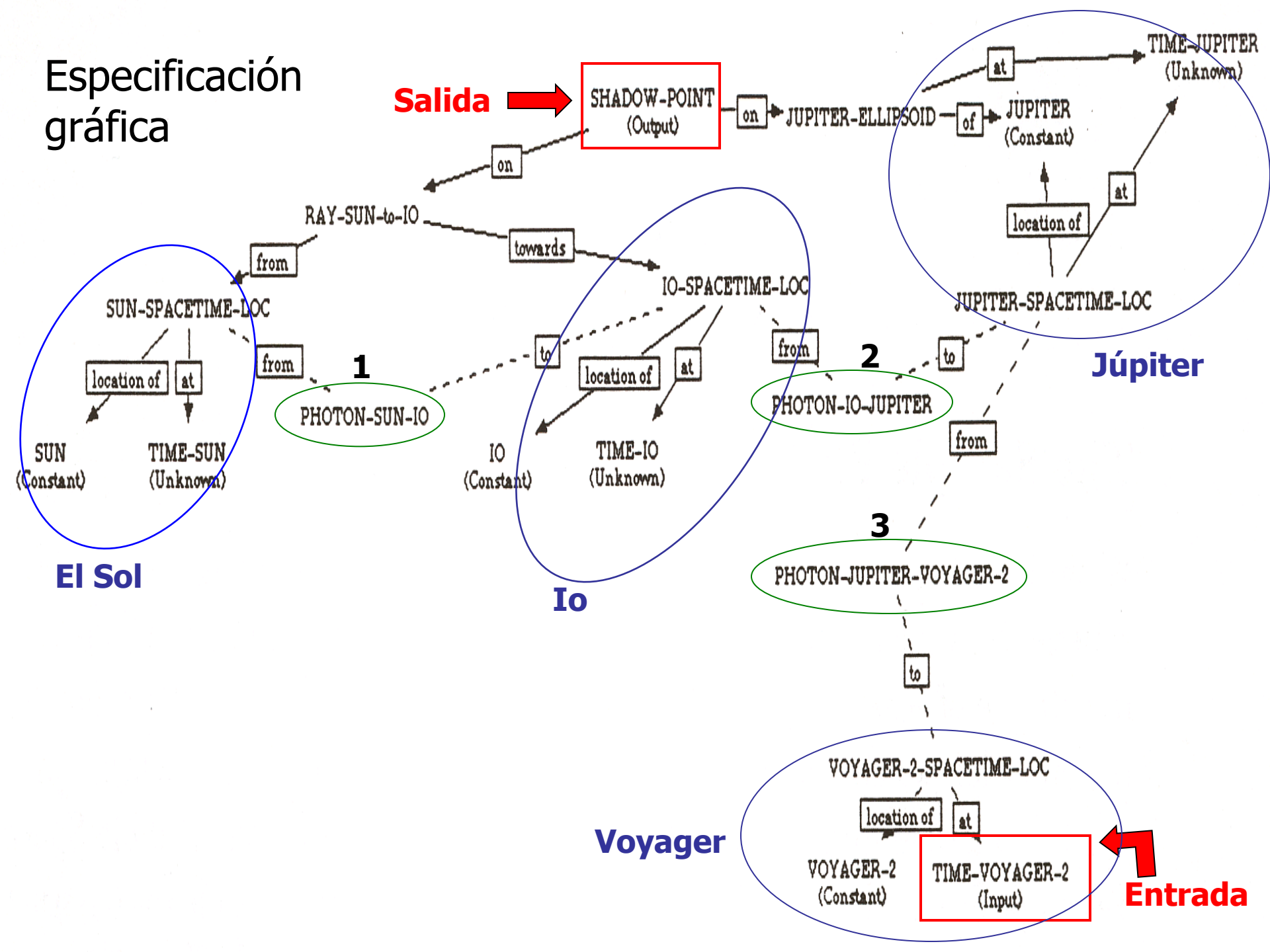
Amphion. Teorema sombra de Io

- Sea un fotón que sale del Sol, pasa por Io (*Ray-Sun-to-Io*), rebota en el *Júpiter-Ellipsoid,* y llega a Voyager2, que está en un lugar y tiempo concretos (entrada al programa)
- ¿Existe algún *shadow-point*, que sea la intersección de *Ray-Sun-to-Io* y el *Júpiter-Ellipsoid* ?
- Find: "∃"constructivo que encuentra una solución. Ej: *∃y P(y) => find y P(y)*

Especificación gráfica



Especificación gráfica



```
(all (time-voyager-2-c)
     (find (shadow-point-c)
    (exists
    (time-sun sun-spacetime-loc time-io io-spacetime-loc
      time-jupiter jupiter-spacetime-loc time-voyager-2
      voyager-2-spacetime-loc shadow-point jupiter-ellipsoid
      ray-sun-to-io)
    (and
     (= ray-sun-to-io
(two-points-to-ray
 (event-to-position sun-spacetime-loc)
 (event-to-position io-spacetime-loc)))
     (= jupiter-ellipsoid
(body-and-time-to-ellipsoid jupiter
    time-jupiter))
     (= shadow-point
(intersect-ray-ellipsoid ray-sun-to-io jupiter-ellipsoid))
     (lightlike? jupiter-spacetime-loc voyager-2-spacetime-loc)
     (lightlike? io-spacetime-loc jupiter-spacetime-loc)
     (lightlike? sun-spacetime-loc io-spacetime-loc)
     (= voyager-2-spacetime-loc
(ephemeris-object-and-time-to-event voyager-2 time-voyager-2))
     (= jupiter-spacetime-loc
(ephemeris-object-and-time-to-event jupiter time-jupiter))
     (= io-spacetime-loc
(ephemeris-object-and-time-to-event io time-io))
     (= sun-spacetime-loc
(ephemeris-object-and-time-to-event sun time-sun))
     (= shadow-point (abs (coords-to-point j2000) shadow-point-c))
     (= time-voyager-2
(abs ephemeris-time-to-time time-voyager-2-c)))))))
```

Teorema

Teorema simplificado

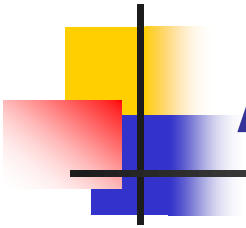
- *∃shadow-point-c*
  - *shadow-point-c* = intersect(Ray-sun-to-io, Jupiter-ellipsoid)
  - Ray-sun-to-io = ray(Sol, IO)
  - Jupiter-ellipsoid = ellipsoid(Jupiter)
  - Ligthlike?(Sol,IO), Ligthlike?(IO, Jupiter)
  - Ligthlike?(Jupiter, Voyager2)
  - Voyager2 el 1 de Junio de 1992

Amphion. Programa FORTRAN

```
SUBROUTINE SHADOW ( TIMEVO, SHADOW )
DOUBLE PRECISION TIMEVO                 ...
INTEGER JUPITE
PARAMETER (JUPITE = 599)                ...
DOUBLE PRECISION RADJUP ( 3 )           ...
CALL BODVAR ( JUPITE, 'RADII', DMYO, RADJUP )
TJUPIT = SENT ( JUPITE, VOYGR2, TIMEVO )
CALL FINDPV ( JUPITE, TJUPIT, PJUPIT, DMY2O )
CALL BODMAT ( JUPITE, TJUPIT, MJUPIT )
TIO = SENT ( IO, JUPITE, TJUPIT )
CALL FINDPV ( IO, TIO, PIO, DMY3O )
TSUN = SENT ( SUN, 1O, TIO )
CALL FINDPV ( SUN, TSUN, PSUN, DMY4O )
CALL VSUB ( PIO, PSUN, DPSPI )
CALL VSUB ( PSUN, PJUPIT, DPJPS )
CALL MXV ( MJUPIT, DPSPI, XDPSPI )
CALL MXV ( MJUPIT, DPJPS, XDPJPS )
CALL SURFPT ( XDPJPS, XDPSPI, RADJUP ( 1 ), RADJUP
              RADJUP ( 3 ), P, DMY9O )
CALL VSUB ( P, PJUPIT, DPJUPP )
CALL MTXV ( MJUPIT, DPJUPP, SHADOW )
END
```

Amphion. Programa FORTRAN

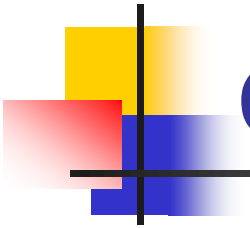
1. *Bodvar*: computa los radios de Júpiter (elipsoide) y los guarda en un array
2. *Sent*: computa el tiempo *tjupit* en el que un fotón debió salir de Júpiter para llegar al Vogager-2 en el tiempo *timev0*
3. *Findpv* y *bodmat* computan la orientación de Júpiter en el tiempo *tjupit.* Se representa como una matriz de 3x3
4. **Etc. El programa trata con matrices, reales, etc. La especificación trata con planetas, tiempos y elipsoides**



Amphion

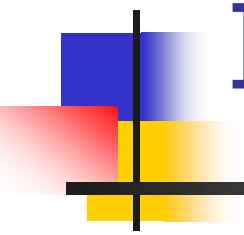
- Ventajas:
  - Fácil de usar (tras 1h de aprendizaje)
    - Expertos: sin Amphion, 30m, con: 5m
    - No-expertos: sin: días, con: 30m
  - Fácil revisar una especificación (en lugar del programa)
  - Fácil añadir subrutinas (añadir axiomas)
  - No limitado al dominio astronómico, basta con que haya una librería de subrutinas
- Limitaciones:
  - Amphion sólo compone un programa hecho de subrutinas, pero no trabaja con condicionales, bucles o recursividad

Ventajas/desventajas PA deductiva

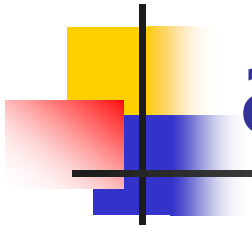


- +: Se garantiza que los programas generados son **correctos.** El programa construido no requiere más verificación, es **seguro** que cumple con la especificación
- +: Más rápidos de programar, menor coste
- +: Mantenibilidad
- - : Es **difícil escribir especificaciones correctas y completas**, especialmente para problemas mal definidos

Programación Automática Inductiva

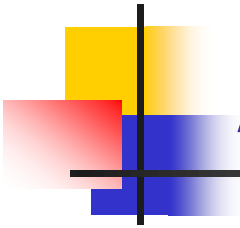


Ejemplo programación automática inductiva



- Entrada:
  - ([2,1], [1,2]); ([2,3,1], [1,2,3]);
  - ([3,5,4], [3,4,5]); ([],[]); ...
- Instrucciones:
  - (dobl start end work) (wismaller x y)
  - (swap x y) (wibigger x y)
  - (e1+ x) (e- x y) (e1- x)
- Especificaciones sencillas, pero difícil garantizar que el programa es correcto completamente

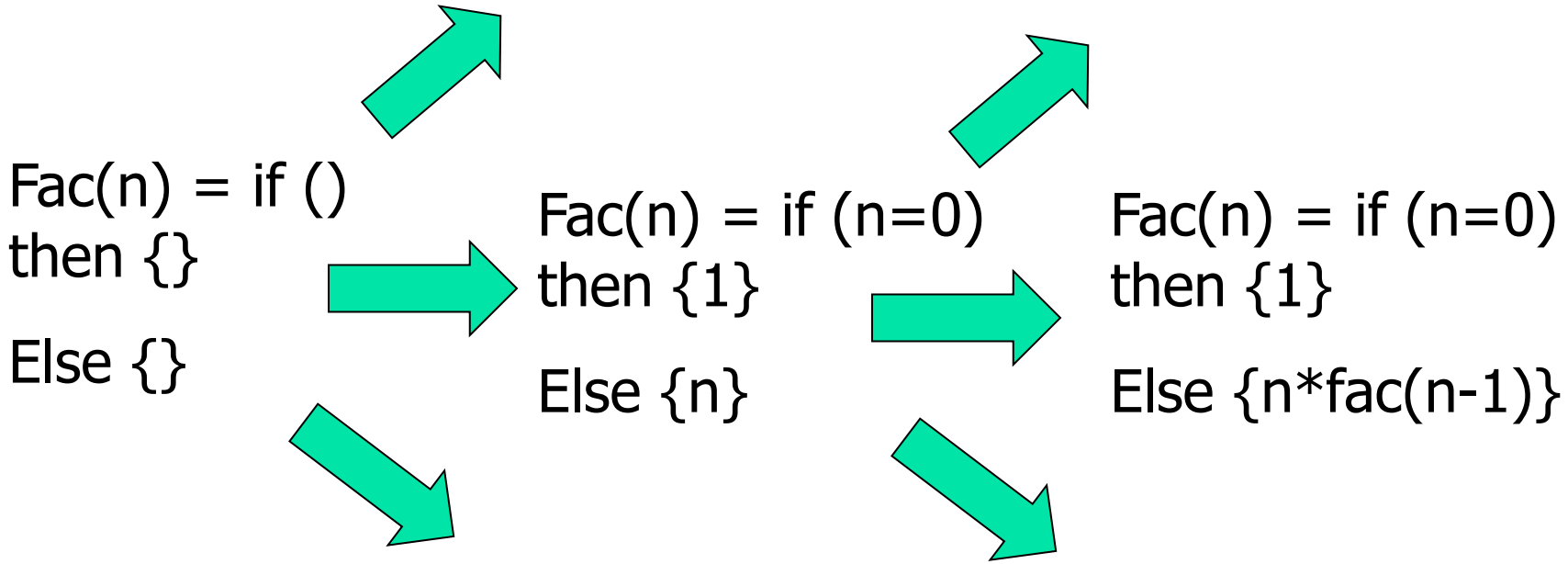
Aproximaciones PA inductiva



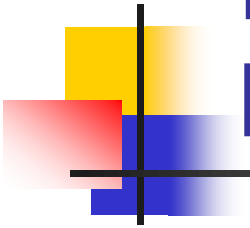
- **Búsqueda + Generar y probar:** el sistema genera varios candidatos programa, los ejecuta, y continua la búsqueda modificando aquellos candidatos que lo hagan mejor.
  - Algoritmos genéticos (Programación Genética,PIPE)
  - A* (ADATE)
  - …
- **Síntesis**: el sistema construye el programa trozo a trozo, pero nunca necesita ejecutarlo para ver que tal funciona.
  - Ej: síntesis de programas en LISP

Idea general PA inductiva mediante búsqueda + generar y probar

- Búsqueda en el espacio de programas de ordenador (transformar y probar, de manera incremental)

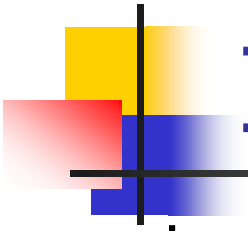


Problemas PA inductiva mediante búsqueda + generar y probar

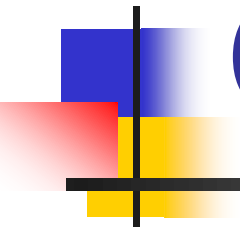


- **Problema 1:** dado un lenguaje, el número de programas posibles crece exponencialmente
- **Problema 2:** los programas recursivos, iterativos, o con bucles son **"frágiles"**
- **Problema 3:** un programa puede no terminar (o tardar mucho)
- **Problema 4:** no se garantiza que el programa obtenido sea completamente correcto (inducción)

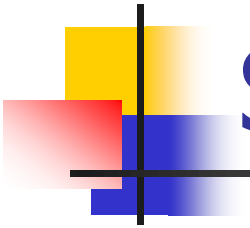
Programación Automática Inductiva



- Los comienzos: síntesis de programas LISP
  - "Programming by Example" o "Programming by Demonstration"
  - Juegos: behavioral cloning
- Búsqueda genética (Genetic Algorithms (GA)
  - Programación Genética (GP)
- Búsqueda mediante estimaciones de distribuciones de probabilidad Estimation of Distribution Algorithms (EDA)
  - Evolución Incremental de Programas (PIPE)
- Búsqueda A*:
  - ADATE
- Búsqueda general-específico:
  - Programación Lógica Inductiva (Inductive Logic Programming ILP)
- Búsqueda de estrategias:
  - Aprendizaje por Refuerzo (Reinforcement Learning RL)

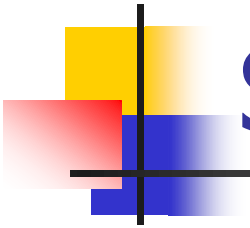


Síntesis de programas LISP (versión en pseudocódigo)



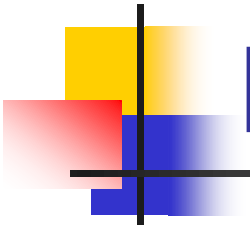
Síntesis de programas LISP

- Summers P. **1977**. "A Methodology for LISP Program Construction from Examples," Journal of the ACM
- Smith, D. 1984. “The Synthesis of LISP Programs from examples. A survey”. Mac Millan Publishing.
- Kitzelmann, E., Schmidt, U., Mühlpfordt, M., and Wysotzki, F. 2002. Inductive Synthesis of Functional Programs. Artificial Intelligence, Automated Reasoning, and Symbolic Computation, Joint International Conference



Síntesis de programas LISP

- Idea: "Para algunas clases de programas, unos pocos ejemplos (entrada/salida) bien elegidos, indican el programa general"
- Ejemplo: [a]→a; [a b]→b; [a b c]→c
  - T_1: a=primero([a])
  - T_2: b=primero(resto([a b]))
  - T_3: c=primero(resto(resto([a b c])))
  - T_K: ultimo=primero(resto ... resto(lista)) (o sea, aplicar resto k-1 veces)
- O sea, se obtienen trazas a partir de los ejemplos de entrada y después se obtiene el patrón general



El lenguaje (pseudocódigo)

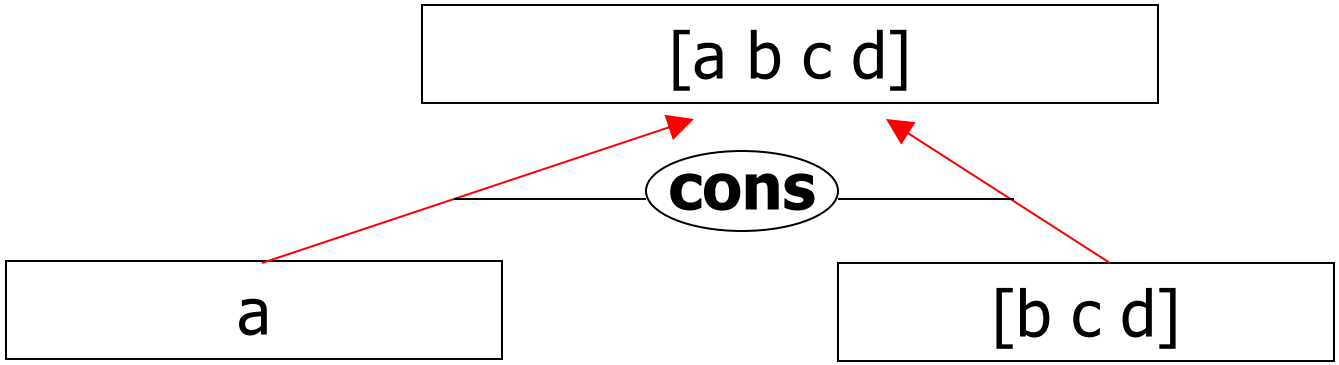
- Tipos de datos:
  - Átomos: a, 3
  - Listas: [a 3 5 [a b c]]
  - Booleanos:
    - True: cierto
    - False: falso

Sublenguaje a utilizar

- Listas:
  - **car**([a b c]) = a (Primero de la lista)
  - **cdr**([a b c]) = [b c] (Resto de la lista)
  - **cons**(a,[b c]) = [a b c] (Construcción de listas)
- Predicados: (cierto o falso)
  - **null**([]) = True (¿Es la lista vacía?)
- Condicional:

```
if (x==0)
  then return(1)
  else return(x*factorial(x-1))
```

El “cons”



Ejemplo de programa a aprender (Summers 77)

- Entradas/salidas:
  - []→[]; [a]→[a]; [a b]→[b a]; [a b c]→[c b a]

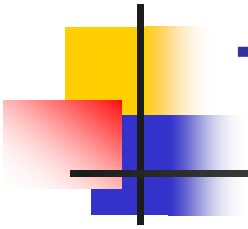
- inv(x) = invertir(x,[]) **Nota**: inv es el programa aprendido
- invertir(x,z)
  if null(x) **then** return(z)
  else return(invertir(cdr(x),cons(car(x),z)))

- En el fondo se trata de pasar de una pila a otra:
  1. invertir([a b c], []) → invertir([b c],[a]) →
  2. invertir([c],[b a]) → invertir ([],[c b a]) →
  3. [c b a]

Ejemplo de programa a aprender (Summers 77)

- Importante: invertir es recursivo (si hay recursividad, no hacen falta bucles)

```
inv(x) = invertir(x,[])
invertir(x,z)
        if null(x) then return z
        else return invertir(cdr(x),cons(car(x),z))
```



Tipos de expresiones

- **Básicas**: composición de car y cdr
  - Ej: cadar = car(cdr(car x))
- **Estructuras *cons***:
  - Ej: cons(f_1,f_2)
  - Ej: cons(car(x),cddr(x))
- **Predicados o condiciones**: (devuelven cierto o falso)
  - Ej: null(cadar(x))

Esquema de programa en seudocódigo (recursivo)

- F(x,z) =

  Case

  $p_1(x)$: $f_1(x,z)$

  ...

  $p_k(x)$: $f_k(x,z)$

  else H(x, F(b(x), G(x,z)))

- *H* y *G* son estructuras cons
- *p1*, ..., *pk* son predicados (booleanos)
- *f1*, ..., *fk* son estructuras cons
- *b* es una expresión básica, b(x) solo sale **una vez en F**

Esquemas de programa. Forward Loop

F(x) =

Case

$p_1(x)$: $f_1(x)$

...

$p_k(x)$: $f_k(x)$

else H(x, F(b(x))

F(x,z) =

Case

$p_1(x)$: $f_1(x,z)$

...

$p_k(x)$: $f_k(x,z)$

Else H(x, F(b(x), G(x,z)))

Esquemas de programa. Reverse Loop

F(x,z) =

Case

$p_1(x)$: $f_1(x,z)$

...

$p_k(x)$: $f_k(x,z)$

else F(b(x), G(x,z)))

F(x,z) =

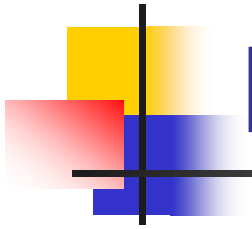
Case

$p_1(x)$: $f_1(x,z)$

...

$p_k(x)$: $f_k(x,z)$

else H(x, F(b(x), G(x,z)))



Ejemplos del esquema

- ultimo(x) =

  case null(cdr(x))

  True: return car(x)

  else return ultimo(cdr(x))

ultimo([a b **c**]) = **c**

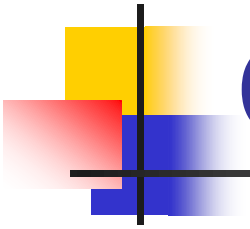
- menos-ultimo(x) =

  case null(cdr(x))

  True: return []

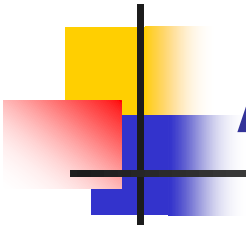
  Else return cons(car(x), menos-ultimo(cdr(x)))

menos-ultimo([**a b** c]) = **[a b]**



Quedan fuera del esquema

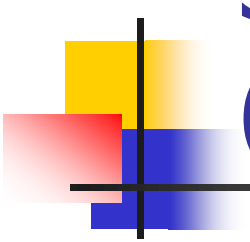
- NO:
  - Funciones con argumentos numéricos (nth, length, ...). Sólo trabaja con listas
  - Funciones de dos argumentos (append, ...). Sólo un argumento (y otro auxiliar)



Algoritmo Summers 1977

1. Obtención de trazas
2. Generación de predicados (para el *case*)
3. Detección de recursividad
4. O bien añadir nuevas variables y detectar recursividad

Síntesis de programas LISP (recordatorio)



- Idea: "Para algunas clases de programas, unos pocos ejemplos (entrada/salida) bien elegidos, indican el programa general"
- Ejemplo: [a]→a; [a b]→b; [a b c]→c
  - T_1: a=primero([a])
  - T_2: b=primero(resto([a b]))
  - T_3: c=primero(resto(resto([a b c])))
  - T_K: ultimo=primero(resto ... resto(lista)) (o sea, aplicar resto k-1 veces)
- O sea, se obtienen trazas a partir de los ejemplos de entrada y después se obtiene el patrón general

Ejemplo función *segundo*

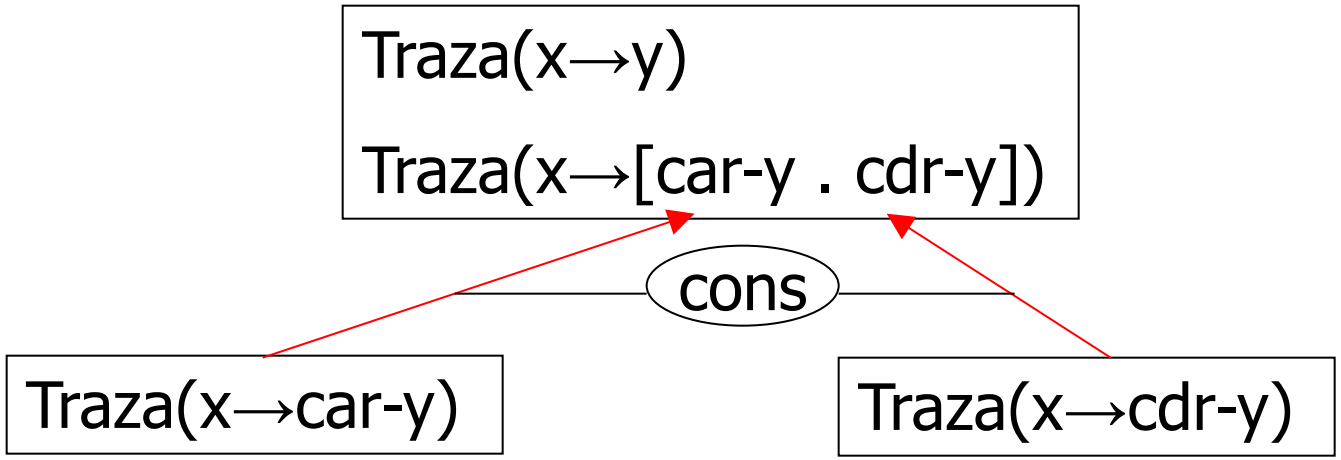
- Obtener el segundo elemento de cada sublista. Parejas entrada/salida x→y:
  - []→[]
  - [[a b]]→[b]
  - [[a b] [c d]]→[b d]
  - [[a b] [c d] [e f]]→[b d f]

- Importante: el algoritmo supone que:
  - Las parejas x→y están dadas en orden de complejidad
  - Ningún átomo aparece dos veces en x
  - Todos los átomos que aparecen en *y* aparecen también en x (autocontenidas). Si esto ocurre, cada pareja tiene una única semi-traza

1. Obtención de (semi)trazas

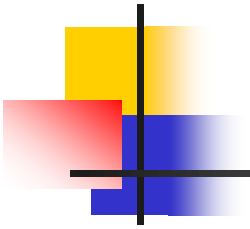
- Para cada pareja entrada/salida (x,y), se quiere encontrar una relación y = f(x)
- Idea: si se encuentra una expresión básica (car y cdr) que convierta x en y, esa es la traza
  - Ej: traza ([a b]→b) : y = (cadr x)
- Si no encuentra una relación directa, entonces prueba a encontrar una relación:
  - Entre x y el car(y): $traza_1$=traza(x→car(y))
  - Entre x y el cdr(y): $traza_2$=traza(x→cdr(y))
  - Como y=cons(car(y),cdr(y)), la traza final será:
    - traza(x→y) = cons($traza_1$, $traza_2$) =
    - = cons(traza(x→car(y)), traza(x→cdr(y)))

1. Obtención de (semi)trazas



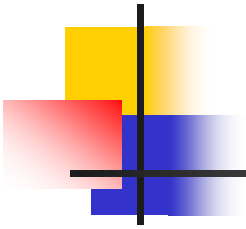
1. Obtención de (semi)trazas

- Algoritmo: traza(x,y)
  - Si (y == [])
    - entonces la traza es **[]**
  - Si (y ≠ []) & (y == b(x))
    - entonces la traza es **b(x)**
  - En caso contrario, la traza es:
    - **cons(traza(x→car(y)),**
      traza(x→cdr(y)))



1. Obtención de (semi)trazas

- Obtener el segundo elemento de cada sublista. Parejas entrada/salida x→y:
  1. []→[]
  2. [[a b]]→[b]
  3. [[a b] [c d]]→[b d]
  4. [[a b] [c d] [e f]]→[b d f]

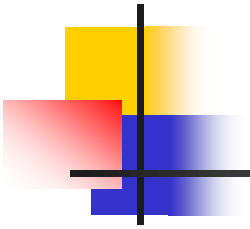


1. Obtención de (semi)trazas

- traza ([]→[]) : y = []

1. Obtención de (semi)trazas

- ¿Traza(x,y) = Traza([[a b]]→[b])?
- ¿Podemos encontrar una expresión básica que convierta x en y?
  - car([[a b]]) = [a b]
    - cdar([[a b]]) = [b]
      - cadar([[a b]]) = b
      - cddar([[a b]]) = []
    - caar([[a b]]) = a
  - cdr([[a b]]) = []



1. Obtención de (semi)trazas

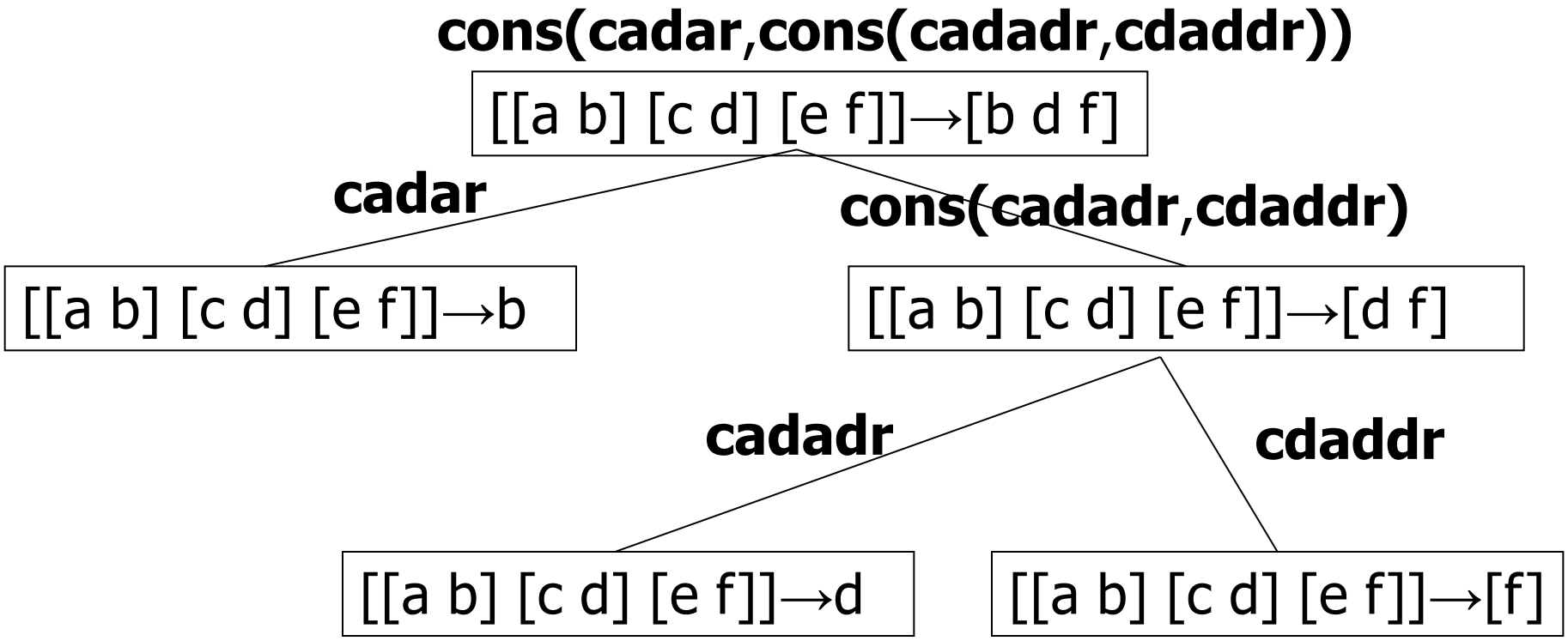
- ¿Traza(x,y) = Traza([[a b] [c d]]→[b d])?
- car([[a b] [c d]]) = [a b]
  - …
- cdr([[a b] [c d]]) = [[c d]]
  - cadr([[a b] [c d]]) = [c d]
    - … va a ser imposible que queden b y d, luego habrá que descomponer
  - cddr([[a b] [c d]]) = []

1. Obtención de (semi)trazas

- ¿Traza(x,y) = Traza([[a b] [c d]]→[b d])?
  - T_1 = Traza([[a b] [c d]] →b) (convierte el car)
    - T_1 = cadr(car(x)) = cadar(x)
  - T_2 = Traza([[a b] [c d]]→[d]) (convierte el cdr)
    - T_2 = cdar(cdr(x)) = cdadr(x)
  - Traza(x,y) = cons(T_1,T_2) =
    - = cons(cadar(x), cdadr(x))

1. Obtención de (semi)trazas

- ¿Traza([[a b] [c d] [e f]]→[b d f])?



1. Obtención de (semi)trazas

1. []→[]

   f_1 = []

2. [[a b]]→[b]

   f_2 = cdar

3. [[a b] [c d]]→[b d]

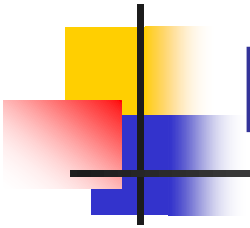
   f_3 = cons(cadar,cdadr)

4. [[a b] [c d] [e f]]→[b d f]

   f_4 = cons(cadar,cons(cadadr,cdaddr))

Visualización de la recursividad

1. []→[]
   f_1 = []
2. [[a b]]→[b]
   f_2 = cdar
3. [[a b] [c d]]→[b d]
   f_3 = cons(cadar,cda**d**r) =
   = cons(cadar(x),f2(cdr(x)))
4. [[a b] [c d] [e f]]→[b d f]
   f_4 = cons(cadar,cons(cada**d**r,cdad**d**r)) =
   = cons(cadar(x),f3(cdr(x)))
5. f_{n+1} = cons(cadar(x),f_n(cdr(x)))



Programa final

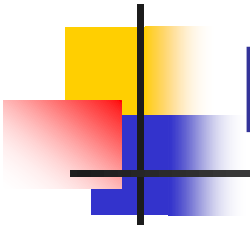
F(x) =

Case

null(x): return []

1 elemento: return cdar(x)

Else return cons(cadar(x),F(cdr(x)))



Programa final

F(x) =

Case

null(x): return []

null(cdr(x)): return cdar(x)

Else return cons(cadar(x),F(cdr(x)))

2. Generación de predicados (para el *case*)

- Supongamos de momento que no hay recursividad
- F(x) =

  Case

  $p_1(x)$: $f_1(x)$

  $p_2(x)$: $f_2(x)$

  $p_3(x)$: $f_3(x)$

  $p_4(x)$: $f_4(x)$
- Hay que encontrar p_i que diferencien unas x's de otras x's

2. Generación de predicados (para el *case*)

- Se supone que las (x_i,y_i) están ordenadas por complejidad
- Si tenemos (x_i,y_i) e (x_{i+1},y_{i+1}), se supone que x_i es "más simple" que x_{i+1}
- Hay que encontrar una manera (predicados p_i) de diferenciar entre x_i e x_{i+1}
- Los predicados tienen que ser de la forma:
  - *(null (b x))* (*b* es función básica, ej: *cadar*, *cdar*, ...)

2. Generación de predicados (para el *case*)

- x_i es "más simple" que x_{i+1} si:
  - X_i es átomo o la lista vacía
  - O bien car(x_i) mas simple que car(x_{i+1}) y cdr(x_i) mas simple que cdr(x_{i+1})

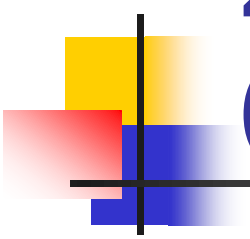
2. Generación de predicados (para el *case*)

- Hay que encontrar una manera

  predicados (null ($b(x_i)$))

  de diferenciar entre x_i e x_{i+1}
- Idea: aplicar car y cdr a x_i e x_{i+1}, para reducirlos, hasta que $b(x_i)$ sea la lista vacía y $b(x_{i+1})$ no lo sea
- PG(x_i,x_{i+1},b):
  - (Si x_{i+1} vacío, devolver *b)*
  - Si x_i es vacío y x_{i+1} no, devolver {*b*}
  - Si no, probar con:
    - PG(car(x)$_i$,car(x_{i+1}),(car b))
    - PG(cdr(x)$_i$,cdr(x_{i+1}),(cdr b))

2. Generación de predicados (para el *case*)

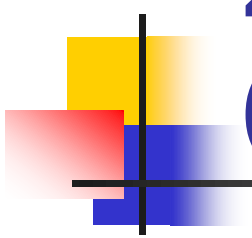


- Ejemplo: diferenciar x_1=[] de x_2=[[a b]]
- Hay que encontrar una expresión básica que anule x_1 pero no x_2
- Fácil, x_1 ya es nulo:
  - null([]) = True
  - null([[a b]]) = False

2. Generación de predicados (para el *case*)

- Ejemplo: diferenciar x_2=[[a b]] de x_3=[[a b][c d]]
- car(x)
  - car(x_2)=[a b]
  - car(x_3)=[a b]
- cdr(x)
  - cdr(x_2)=[]
  - cdr(x_3)=[[c d]]
- cdr nos valdría porque:
  - null(cdr(x_2)) = True
  - null(cdr(x_3)) = False

2. Generación de predicados (para el *case*)



- Ejemplo: diferenciar
  - x_3=[[a b][c d]]
  - de x_4=[[a b] [c d] [e f]]
- Si hicieramos la búsqueda de las expresiones cxxxx...xxr, veríamos que la primera que nos sirve es cddr(x) porque:
  - null(cddr(x_3)) = True
  - null(cddr(x_4)) = False

2. Generación de predicados (para el *case*)

F(x) =
Case
  $p_1(x)$: $f_1(x)$
  $p_2(x)$: $f_2(x)$
  $p_3(x)$: $f_3(x)$
  $p_4(x)$: $f_4(x)$

- Nos queda que:
  - $p_1(x) = null(x)$
  - $p_2(x) = null(cdr(x))$
  - $p_3(x) = null(cddr(x))$
  - $p_4(x) = null(cdddr(x))$
- Detección de recursividad. Vemos que podemos expresar p_{i+1} en términos de p_i:
  - $p_{i+1}(x) = p_i(cdr(x))$

El programa sin recursividad nos queda así:

1. []→[]
2. [[a b]]→[b]
3. [[a b] [c d]]→[b d]
4. [[a b] [c d] [e f]]→[b d f]

F(x) =
Case
$p_1(x)$: $f_1(x)$
$p_2(x)$: $f_2(x)$
$p_3(x)$: $f_3(x)$
$p_4(x)$: $f_4(x)$

F(x) =
Case
null(x): return []
null(cdr(x)): return cdar(x)
null(cddr(x)): return cons(cadar,cons(cadadr,cdaddr))
null(cdddr(x)): return cons(cadar,cons(cadadr,cdaddr))

3. Detección de recursividad

- **Basic synthesis theorem**
- Si podemos expresar f_{i+1} en términos de f_i
- Y podemos expresar p_{i+1} en términos de p_i
- Entonces podemos expresar el programa de manera recursiva, válido para listas de cualquier longitud:

F(x) =
Case
  $p_1(x)$: $f_1(x)$
  $p_2(x)$: $f_2(x)$
  $p_3(x)$: $f_3(x)$
  $p_4(x)$: $f_4(x)$

F(x) =
Case
  $p_1(x)$: $f_1(x)$
  else: H(F(b(x)),x)

3. Detección de recursividad

- **Basic synthesis theorem**
- **Si ocurre:**
  - $\forall i$ $p_{i+1}(x) = p_i(b(x))$
  - $\forall i$ $f_{i+1}(x) = H(f_i(b(x)),x)$
    - $f_i(b(x))$ aparece sólo una vez en H
- **Entonces:**
  - F(x) = **Case**

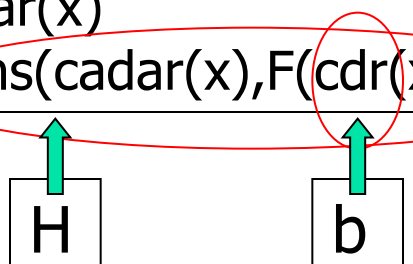
    $p_1(x)$: $f_1(x)$

    else: H(F(b(x)),x)

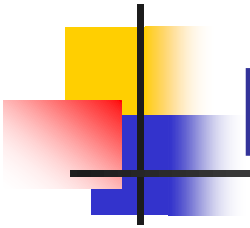
3. Detección de recursividad

- Hemos visto que para n>=2:
  1. f_{n+1} = cons(cadar(x),f_n(cdr(x)))
  2. $p_{i+1}(x) = p_i(cdr(x))$
- Luego:

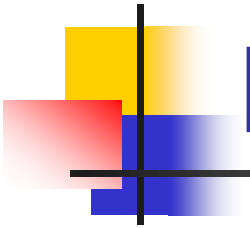
```
F(x) =
Case
  null(x):        return []
  null(cdr(x)): return cdar(x)
  else:           return cons(cadar(x),F(cdr(x)))
```



Conclusiones. Inducción programas LISP



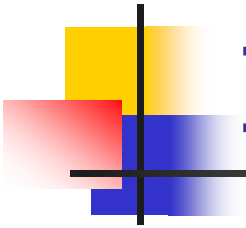
- [Summers, 78] Detecta recursividad
- Solución analítica, fuertemente teórica:
  - No es necesario hacer búsqueda
  - Utiliza sólo unos pocos ejemplos
- Funciona porque:
  - Restringe el esquema de programa a aprender (funciones recursivas en un solo argumento)
  - Se limita a problemas de transformaciones de listas (invertir, segundo, ultimo, menos-ultimo, …). Structural tasks
  - Restringe las parejas entrada/salida para que la traza que convierte la entrada en la salida sea única
  - Sólo permite recursividad lineal



Extensiones

- Sistemas IGOR1 e IGOR2 de Ute Schmid.
- Permite recursividad lineal, de cola (tail) y de árbol
- Sólo tareas estructurales

- http://www.inductive-programming.org/
- Pierre Flener, Ute Schmid: *An introduction to inductive programming*. Artif. Intell. Rev. 29(1): 45-62 (2008)
- Martin Hofman: *Automated construction of XSL-templates : an inductive programming approach*



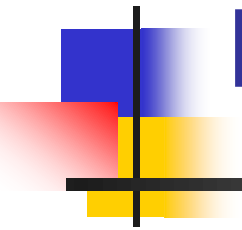
Inducción de gramáticas

- L. Miclet and C. de la Higuera, editors. ***Grammatical Inference****: Learning Syntax from Sentences*, number 1147 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.
- J. Gregor. Data-driven inductive inference of **finite-state automata**. *International Journal of Pattern Recognition and Artificial Intelligence*, 8(1):305-322, 1994.
- Relacionado con la inducción de automatas de estado finito.

Resultados

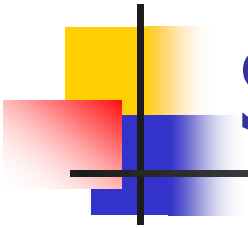
Tiempo total

function	#expl	#eqs	times in sec
last	4	2	.003 / .001 / .004
init	4	1	.004 / .002 / .006
evenpos	7	2	.01 / .004 / .014
switch	6	1	.012 / .004 / .016
unpack	4	1	.003 / .002 / .005
lasts	10	2	.032 / .032 / .064
mult-lasts	11	3	.04 / .49 / .53
reverse	6	4	.031 / .036 / .067



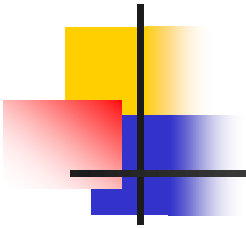
Síntesis de programas LISP (en LISP)

Repetición de lo anterior, pero directamente en lenguaje LISP.



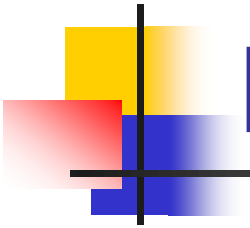
Síntesis de programas LISP

- Idea: "Para algunas clases de programas, unos pocos ejemplos (entrada/salida) bien elegidos, indican el programa general"
- Ejemplo: [(A),A]; [(A B), B]; [(A B C), C]
  - T_1: A=primero((A))
  - T_2: B=primero(resto((A B)))
  - T_3: C=primero(resto(resto((A B C))))
  - T_K: ultimo=primero(resto ... resto(lista)) (o sea, aplicar resto k-1 veces)
- O sea, se obtienen trazas a partir de los ejemplos de entrada y después se obtiene el patrón general



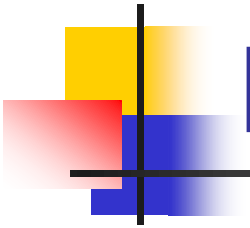
El lenguaje LISP

- LISP = List Programming (o "Lots Of Irritating Superfluous Parentheses")
- Notación prefija:
  - Infija: 3 + (4*5)
  - Prefija: +(3,*(4 5))
  - En LISP: (+ 3 (* 4 5))
- En LISP, todo son listas:
  - Datos: '(1 2 3), '(), '(a (2 b))
  - Programas:
    - (+ 3 (* 4 5))
    - (defun factorial (x)
      (if (= x 0) 1 (* x (factoiral (- x 1))))



El lenguaje LISP

- Como los programas son también datos (listas), es fácil que un programa construya otros programas
- LISP es funcional, todas las expresiones devuelven un valor
  - (if (= x 0) 3 4) -> 3
- En LISP, es fácil ejecutar programas:
  - (eval mi-programa)
- Desventaja: lenguaje interpretado (ligeramente lento)



El lenguaje LISP

- Tipos de datos:
  - Átomos: A, 3, nil, ‘(), T
  - Listas: ‘(a 3 5 (a b c))
  - Booleanos:
    - T: cierto
    - NIL = ‘(): falso
- La comilla ‘ (quote) indica si algo es una constante o hay que ejecutarlo:
  - ‘(+ 3 4): lista de 3 elementos
  - (+ 3 4): expresión a evaluar

El lenguaje LISP. Sublenguaje

- Listas:
  - (car '(a b c)) = a — (Primero de la lista)
  - (cdr '(a b c)) = (b c) — (Resto de la lista)
  - (cons 'a '(b c)) = (a b c) — (Construcción de listas)
- Predicados: (cierto o falso)
  - (null '()) = T; (null nil) = T — (?Es la lista vacía?)
- Condicional:
  - (cond ((= x 0) 1)
    (T (* x (factorial (- x 1)))))

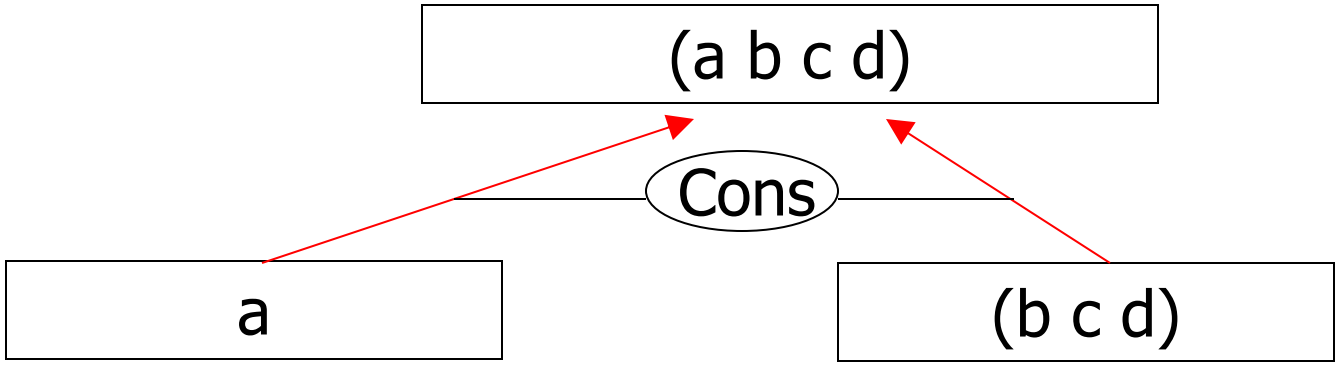
```
If (x==0)
  Then Return(1)
  Else Return(x*factorial(x-1))
```

El lenguaje LISP. Sublenguaje

- Listas:
  - car((a b c)) = a (Primero de la lista)
  - cdr((a b c)) = (b c) (Resto de la lista)
  - cons(a,(b c)) = (a b c) (Construcción de listas)
- Predicados: (cierto o falso)
  - Null(()) = T (?Es la lista vacía?)
- Condicional:

```
If (x==0)
  Then Return(1)
  Else Return(x*factorial(x-1))
```

Lenguaje LISP. El “cons”



Ejemplo de programa a aprender (Summers 77)

- Entradas/salidas:
  - [(),()]; [(a),(a)];
  - [(a b), (b a)]; [(a b c), (c b a)]

- inv(x) = invertir(x,()) **Nota**: inv es el programa aprendido
- invertir(x,z)

  If null(x) **then** return(z)

  Else return(invertir(cdr(x),cons(car(x),z)))

- En el fondo se trata de pasar de una pila a otra:
  1. invertir ((a b c), ()) -> invertir((b c),(a)) ->
  2. invertir ((c), (b a)) -> invertir ((), (c b a)) ->
  3. (c b a)

Ejemplo de programa a aprender (Summers 77)

- Entradas/salidas: [(),()]; [(a),(a)]; [(a b), (b a)]; [(a b c), (c b a)]

- Programa aprendido:

```
(inv x) = (invertir x '())
(invertir x z) =
      (cond  ((null x) z)
                  (T (invertir (cdr x) (cons (car x) z))))
```

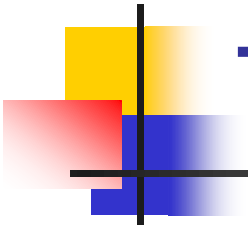
- En el fondo se trata de pasar de una pila a otra:
  1. (invertir '(a b c) '()) -> (invertir '(b c) '(a)) ->
  2. (invertir '(c) '(b a)) -> (invertir '() '(c b a)) ->
  3. '(c b a)

Ejemplo de programa a aprender (Summers 77)

- Importante: invertir es recursivo (si hay recursividad, no hacen falta bucles)

```
(inv x) = (invertir x '())
(invertir x z) =
        (cond  ((null x) z)
                        (T (invertir (cdr x) (cons (car
                x) z))))
```

```
inv(x) = invertir(x,())
invertir(x,z)
        If null(x) then return(z)
        Else return(invertir(cdr(x),cons(car(x),z)))
```



Tipos de expresiones

- **Básicas**: composición de *car* y *cdr*
  - Ej: cadar = (car (cdr (car x))
- **Estructuras *cons***:
  - Ej: (cons f1 f2)
  - Ej: (cons (car x) (cddr x))
- **Predicados o condiciones**: (devuelven cierto o falso)
  - Ej: (null (cadar x))

Esquema de programa en seudocódigo (recursivo)

- F(x,z) =

  Case

  p1(x): f1(x,z)

  ...

  pk(x): fk(x,z)

  Else H(x, F(b(x), G(x,z)))

- *H* y *G* son estructuras cons
- *p1*, ..., *pk* son predicados (booleanos)
- *f1*, ..., *fk* son estructuras cons
- *b* es una expresión básica, b(x) solo sale **una vez en F**

Esquema de programa en LISP (recursivo)

- (F x z) =

```
(cond
    ((p1 x) (f1 x z))
    ...
    ((pk x) (fk x z))
    (T (H x (F (b x) (G x z))))))
```

- *H* y *G* son programas (como F)
- *p1*, ..., *pk* son predicados (booleanos)
- *f1*, ..., *fk* son estructuras cons
- *b* es una función básica , b(x) solo sale **una vez en F**

Esquemas de programa. Forward Loop

- (F x) =

  (**cond**

  ((p1 x) (f1 x z))

  ...

  ((pk x) (fk x z))

  (**T** (H x (F (b x))))))

- F(x) =

  Case

  p1(x): f1(x)

  ...

  pk(x): fk(x)

  Else H(x, F(b(x))

Esquemas de programa. Reverse Loop

- (F x z) =

```
(cond
    ((p1 x) (f1 x z))
    ...
    ((pk x) (fk x z))
    (T (F (b x) (G x z)))))
```

- F(x,z) =

```
Case
    p1(x): f1(x,z)
    ...
    pk(x): fk(x,z)
    Else F(b(x), G(x,z)))
```

Ejemplos del esquema

- (ultimo x) =

```
(cond
    ((null (cdr x))
        (car x))
    (T (ultimo (cdr x)))))
```

(ultimo '(a b c)) = c

- (menos-ultimo x) =

```
(cond
    ((null (cdr x)) nil)
    (T (cons (car x) (menos-ultimo (cdr x)))))
```

(menos-ultimo '(a b c)) = '(a b)

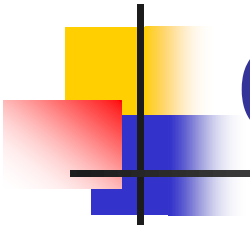
Ejemplos del esquema

- ultimo(x) =
  if null(cdr(x))
  then return car(x)
  else return cdr(x)

ultimo((a b c)) = c

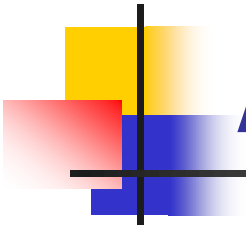
- menos-ultimo(x) =
  if null(cdr(x))
  then return ()
  Else return cons(car(x), menos-ultimo(cdr(x)))

menos-ultimo((a b c)) = (a b)



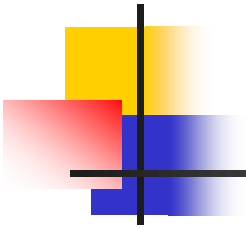
Quedan fuera del esquema

- NO:
  - Funciones con argumentos numéricos (nth, length, ...). Sólo trabaja con listas
  - Funciones de dos argumentos (append, ...). Sólo un argumento (y otro auxiliar)



Algoritmo Summers 1977

1. Obtención de trazas
2. Generación de predicados (para el *case*)
3. Detección de recursividad
4. O bien añadir nuevas variables y detectar recursividad



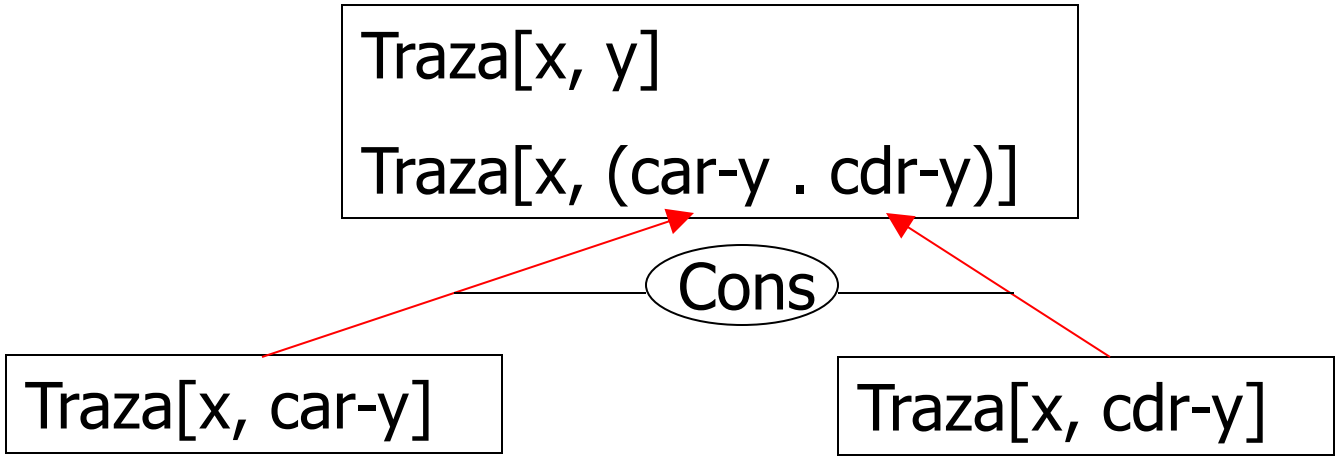
Ejemplo función *invertir*

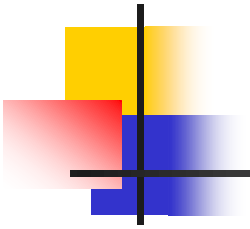
- Parejas entrada/salida *(x,y)*:
  - [(), ()]
  - [(A), (A)]
  - [(A B), (B A)]
  - [(A B C), (C B A)]
- Importante: el algoritmo supone que:
  - Las parejas *(x,y)* están dadas en orden de complejidad
  - Ningún átomo aparece dos veces en *x*
  - Todos los átomos que aparecen en *y* aparecen también en *x* (autocontenidas). Si esto ocurre, cada pareja tiene una única semi-traza

1. Obtención de (semi)trazas

- Para cada pareja entrada/salida *(x,y),* se quiere encontrar una relación *y = f(x)*
- Idea: si se encuentra una relación directa entre *x* e *y*, esa es la traza
  - Ej: traza[(A),A] : *y = (car x)*
- Si no encuentra una relación directa, entonces prueba a encontrar una relación:
  - Entre *x* y el *car(y)* **;** Entre *x* y el *cdr(y)*
  - Ej: traza[(A B),(B A)] :
    - *y1 = traza([(A B], B]) : y1 = (cadr x)*
    - *y2 = traza([(A B), (A)]) : y2 = (cons (car x) ())*
    - *y = (cons y1 y2)*

1. Obtención de (semi)trazas





1. Obtención de (semi)trazas

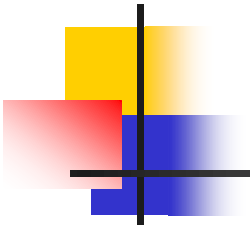
- Algoritmo: traza(x,y)
  - Si (y <> nil) & (y = b(x))
    - entonces la traza es (**b x)**
  - Si (y = nil)
    - entonces la traza es **nil**
  - En caso contrario, la traza es:
    - (cons traza(x, car(y)) traza(x,cdr(y)))

1. Obtención de (semi)trazas

- traza[(), ()] : y = nil
- traza[(A), (A)] : y = (cons (car x) nil)
  - (car '(A)) = A; (cdr '(A)) = '()
  1. (cons traza((A),A) traza((A),())))
  2. (cons (car '(A)) nil)
  3. y = (cons (car x) nil)

1. Obtención de (semi)trazas

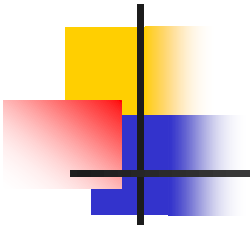
- traza[(A B), (B A)] : y = (cons (cadr x) (cons (car x) nil))
  1. (cons traza((A B),car((B A))) traza((A B), cdr((B A))))
  2. (cons traza((A B),B) traza((A B), (A)))
  3. (cons (cadr (A B)) **(cons traza((A B), car((A))) traza((A B), cdr((A))))**)
  4. (cons (cadr x) **(cons traza((A B), A) traza((A B), nil)**)
  5. (cons (cadr x) **(cons car((A B)) nil)**)
  6. y = (cons (cadr x) **(cons car(x) nil)**)



1. Obtención de (semi)trazas

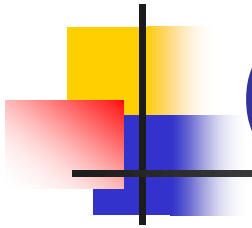
- traza[(A B C), (C B A)] :
  - y = (cons (caddr x) (cons (cadr x) (cons (car x) nil)))

1. Obtención de (semi)trazas



1. **[(),()]:** (f1 x) = nil
2. **[(A),(A)]:** (f2 x) = (cons (car x) nil)
3. **[(A B), (B A)]:** (f3 x) = (cons (cadr x) (cons (car x) nil))
4. **[(A B C), (C B A)]:** (f4 x) = (cons (caddr x) (cons (cadr x) (cons (car x) nil)))

2. Generación de predicados (para el *case*)



- Supongamos que no hay recursividad
- F(x) =

  Case

  p1(x): f1(x)

  p2(x): f2(x)

  p3(x): f3(x)

  p4(x): f4(x)
- Hay que encontrar p_i que diferencien unas *x's* de otras *x's*

2. Generación de predicados (para el *case*)

- Se supone que las (x_i,y_i) están ordenadas por complejidad
- Si tenemos (x_i,y_i) e (x_{i+1},y_{i+1}), se supone que x_i es "más simple" que x_{i+1}
- Hay que encontrar una manera (predicados p_i) de diferenciar entre x_i e x_{i+1}
- Los predicados tiene que ser de la forma:
  - *(null (b x))* (*b* es función básica, ej: *cadar*, *cdar*, ...)

2. Generación de predicados (para el *case*)

- x_i es "más simple" que x_{i+1} si:
  - X_i es átomo
  - O bien car(x_i) mas simple que car(x_{i+1}) y cdr(x_i) mas simple que cdr(x_{i+1})

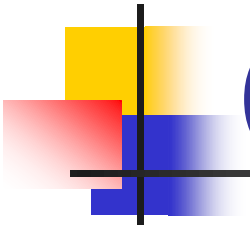
2. Generación de predicados (para el *case*)

- Hay que encontrar una manera

  predicados *(null (b x_i))*

  de diferenciar entre x_i e x_{i+1}
- Idea: aplicar *car* y *cdr* a x_i e x_{i+1}, para reducirlos, hasta que *b(x_i)* sea la lista vacía y *b(x_{i+1})* no lo sea
- PG(x_i,x_{i+1},b):
  - (Si x_{i+1} vacío, devolver *b)*
  - Si x_i es vacío y x_{i+1} no, devolver {*b*}
  - Si no, probar con:
    - *PG(car(x)$_i$,car(x_{i+1}),(car b))*
    - *PG(cdr(x)$_i$,cdr(x_{i+1}),(cdr b))*

2. Generación de predicados (para el *case*)



1. [(), ()] : p1= (null x)
2. [(A), (A)] : p2= (null (cdr x))
3. [(A B), (B A)] : p3= (null (cddr x))
4. [(A B C), (C B A)] : p4= (null (cdddr x))

2. Generación de predicados (para el *case*)

- F(X) = (cond
  ((**null x) nil)**
  (**(null (cdr x))** (cons (car x) nil))
  (**(null (cddr x))** (cons (cadr x) (cons (car x) nil)))
  (**(null (cddr x))** (cons (caddr x) (cons (cadr x)
  (cons (car x) nil)))))

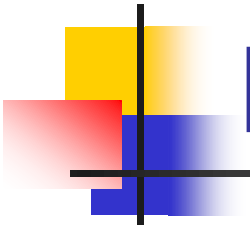
3. Detección de recursividad

- **Basic synthesis theorem**
- **Si ocurre:**
- $\forall i \;\; p_{i+1}(x) = p_i(b(x))$
- $\forall i \;\; f_{i+1}(x) = H(f_i(b(x)),x)$
  - $f_i(b(x))$ aparece sólo una vez en H
- **Entonces:**
- $F(x) =$ case

  $p_1(x) : f_1(x)$

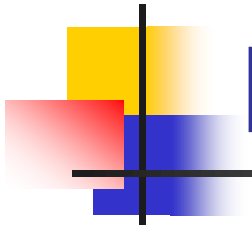
  Else:$H(F(b(x)),x)$

3. Detección de recursividad. Ejemplo



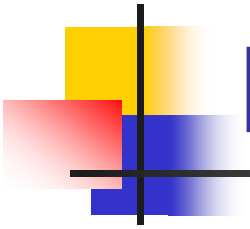
- Obtener el segundo elemento de cada sublista
- [nil, nil]
- [((a b)), (b)]
- [((a b) (c d)), (b d)]
- [((a b) (c d) (e f)), (b d f)]

3. Detección de recursividad. Ejemplo



- (f1 x) = nil
- (f2 x) = (cons (cadar x) nil)
- (f3 x) = (cons (cadar x) **(cons (cadadr x) nil)**)
- (f4 x) = (cons (cadar x) **(cons (cadadr x) (cons (cadaddr x) nil))**)
- Vemos que (f3 x) = (cons (cadar x) (f2 (cdr x)))
- Vemos que (f4 x) = (cons (cadar x) (f3 (cdr x)))
- (f_{i+1} x) = (cons (cadar x) (f_i (cdr x)))

3. Detección de recursividad. Ejemplo



- (p1 x) = (null x)
- (p2 x) = (null (cdr x))
- (p3 x) = (null (cddr x))
- (p4 x) = (null (cdddr x))
- $(p_{i+1}\ x) = p_i(cdr(x))$

3. Detección de recursividad. Ejemplo

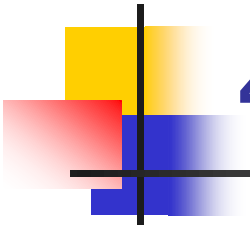
- $(f_{i+1}\ x) = (\text{cons } (\text{cadar } x)\ (f_i\ (\text{cdr } x)))$
- $(p_{i+1}\ x) = p_i(\text{cdr}(x))$
- (F x) =

```
(cond ((null x) nil)
      (T (cons (cadar x) (F (cdr x)))))
```

- (F x) =

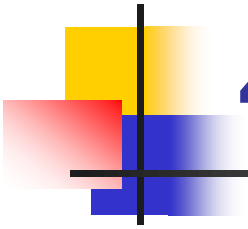
```
(cond ((null x) nil)
      (T (H x (F (cdr x)))))
```

- (H x z) = (cons (cadar x) z)



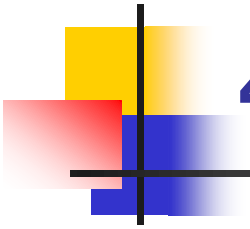
4. Añadir variables

- Para el caso de *invertir*, no se puede hacer inducción de recursividad
- (f1 x) = nil
- (f2 x) = (cons (car x) nil)
- (f3 x) = (cons (cadr x) (cons (car x) nil))
- (f4 x) = (cons (caddr x) (cons (cadr x) (cons (car x) nil)))
- No se puede poner como H(F(b(x)),x)
  - f_{i+1} = (cons ? f_i(x))
  - El "?" no es fijo: (car x), (cadr x), (caddr x)



4. Añadir variables

1. Buscar en las trazas $f_i(x)$ una subexpresión a que se repita en todas
2. Reescribir $f_i(x) = g_i(x,a)$
3. Generalizar $g_i(x,z)$
4. Encontrar una relación de recurrencia para las g_i. Si no se encuentra, volver a 1
5. Generar un programa $G(x,z)$
6. Definir $F(x) = G(x,a)$



4. Añadir variables

- (f1 x) = (g1 x a=nil) = a
- (f2 x) = (g2 x a=nil) = (cons (car x) a)
- (f3 x) = (g3 x a=nil) = (cons (cadr x) (cons (car x) a))
- (f4 x) = (g4 x a=nil) = (cons (caddr x) (cons (cadr x) (cons (car x) a)))

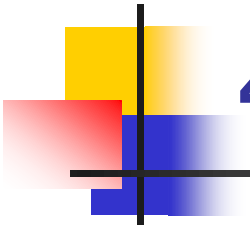
4. Añadir variables

- (g1 x z) = z
- (g2 x z) = (cons (car x) z)
- (g3 x z) = (cons (cadr x) (cons (car x) z))
- (g4 x z) = (cons (caddr x) (cons (cadr x) (cons (car x) z)))

- (g3 x z) = (g2 (cdr x) (cons (car x) z))
- (g4 x z) = (g3 (cdr x) (cons (car x) z))
- (g_{i+1} x z) = (g_i (cdr x) (cons (car x) z))

4. Añadir variables

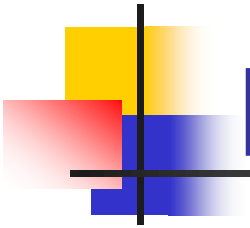
- (g2 x z) = (cons (car x) z)
- (g3 x z) = (cons (cadr x) (cons (car x) z))
- Si en *g2* en lugar de *x* ponemos *(cdr x)*
- Y en lugar de poner *z* ponemos *(cons (car x) z)*
- Nos queda *g3*
- Igual con el resto
- $(g_{i+1}$ x z) = $(g_i$ (cdr x) (cons (car x) z))
- En general, podemos cambiar una sub-semitraza que ocurre en cada traza por una variable *z*



4. Añadir variables

```
(REVERSE x) = (G x nil)
(G x z) = (cond ((null x) z)
                (T (G (cdr x)
                      (cons (car x) z))))
```

Conclusiones. Inducción programas LISP



- [Summers, 78] Detecta recursividad, añade variables auxiliares
- Solución analítica, fuertemente teórica, sin búsqueda
- Funciona porque:
  - Restringe el esquema de programa a aprender (funciones recursivas en un solo argumento
  - Restringe las parejas entrada/salida