Prolog(SWI-Prolog)

Introducción al lenguaje de programación Prolog

Dr. Víctor de la Cueva
vcueva@itesm.mx

Comparaciones entre implementaciones

- En http://en.wikipedia.org/wiki/Comparison_of_Prolog_implementations

Se puede encontrar una tabla con las características de cada una de las implementaciones conocidas.

- Nosotros vamos a usar ISO-Prolog y una de sus versiones más extendida es SWI-Prolog (http://www.swi-prolog.org/).
- También es recomendable:
  - GNU-Prolog (http://www.gprolog.org/).
  - Visual Prolog (http://www.visual-prolog.com/).

Comentario sobre Prolog

- Prolog es un lenguaje de programación diseñado especialmente para hacer programación simbólica, es decir, para trabajar con símbolos.
- Es el lenguaje de programación que menos se alinea con lo que estamos acostumbrados a hacer con un lenguaje imperativo (mucho menos que los funcionales).
- No existen la mayor parte de los conceptos conocidos como función, ciclo, etc.
- Por lo tanto, las aplicaciones en donde conviene usar Prolog se deben seleccionar adecuadamente.

Elementos básicos

- En Prolog existen 3 elementos básicos para formar términos:
  - Números (e.g. 0 y 1972).
  - Átomos (e.g. lisp y algol60). Inician con letra minúscula.
  - Variables (e.g. X y Fuente). Inician con una letra mayúscula.
- Estos elementos son la base de todas las construcciones que se realizan en Prolog para hacer un programa.
- Un programa en Prolog se puede ver como:
  - Una base de datos lógicos, compuesta por hechos y reglas.
  - Un mecanismo para realizar búsquedas y hacer deducciones sobre ellos, el cual permite hacer consultas sobre la base.
- Recuerde que Prolog es la implementación de la Programación Lógica (basada en la Lógica Proposicional).

Términos

- Los hechos, las reglas y las consultas se especifican usando términos.
- Un término simple es:
  - Un número.
  - Una variable que comienza con letra mayúscula.
  - Un átomo que se representa a sí mismo.
- Un término compuesto consiste de un átomo seguido de una secuencia de subtérminos entre paréntesis. Ejemplo

  liga(bcpl, c)
- El átomo "liga" se conoce como funcionador (*functor*) y los subtérminos "bcpl" y "c", como argumentos.

Sintaxis básica de hecho, reglas y consultas

<hecho> ::= <término>.

<reglas> ::= <término> :- <términos>.

<consulta> ::= <términos>.

<término> ::= <número> | <átomo> | <variable>

| <átomo>(<términos>)

<términos> ::= <término> | <término>, <términos>

Notas

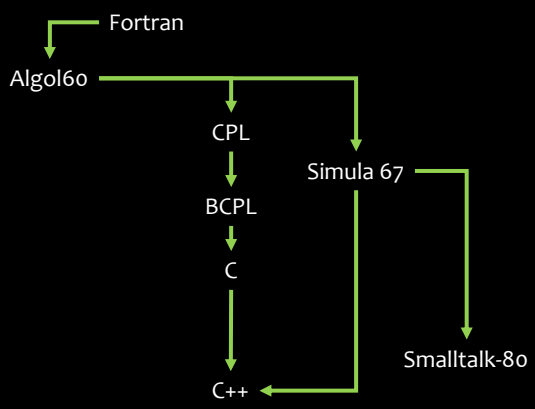
- Algunos operadores se pueden escribir tanto en notación infija como en prefija; por ejemplo, la notación =(X,Y) puede escribirse de manera equivalente como X=Y.
- La variable especial "_" es un lugar para colocar un término sin nombre.
- Todas las apariciones de "_" son independientes entre sí.

Interacción con Prolog

- Cuando se inicia el sistema aparece con el prompt ?-, para anunciar que se espera una consulta.
- El predicado (construcción) *consult* lee un archivo que contiene hechos y reglas, y agrega su contenido al final de la base de datos de reglas actuales.
- En algunos sistemas, como el que vamos a usar, es más fácil cargar un archivo desde el menú, en la opción File. Se puede hacer con Open o con Consult…
- Al terminar de cargar responde *true* (en algunos sistemas responde "*yes*").
- Algunos sistemas permite cargarlo al dar doble clic en el archivo (e.g. Windows).

Gráfico para los ejemplos

Ligas entre lenguajes:



Creando un archivo: ligas

```
liga(fortran, algol60).
liga(algol60,cpl).
liga(cpl, bcpl).
liga(bcpl, c).
liga(c, cmasmas).
liga(algol60, simula67).
liga(simula67, cmasmas).
liga(simula67, smalltalk80).

ruta(L, L).
ruta(L, M) :- liga(L, X), ruta(X, M).
```

Consulta de existencia

- Una consulta:

  <término>$_1$, <término>$_2$, … , <término>$_k$.

  Para k ≥ 1, corresponde al siguiente seudocódigo:

  ¿ <término>$_1$ **and** <término>$_2$ **and** … **and** <término>$_k$?

- Las consultas se conocen también como metas (*goals*).
- Algunas veces es conveniente referirse a los términos individuales de una consulta como submetas.
- No existe, sin embargo, una distinción formal entre metas y submetas, de la misma forma que no existe diferencia formal entre término y subtérmino.

Consultas sin variables

- Cuando no existen variables en la consulta, el sistema simplemente verifica si el hecho se cumple, es decir, si está en la base.
- Desde luego que se supone que todos los hechos en la base son VERDADEROS.
- Ejemplo:

```
?- liga(cpl, bcpl), liga(bcpl, c).
true
```

Variable en consultas

- Una variable dentro de una consulta hace referencia a la existencia de algunos objetos apropiados a las condiciones planteadas.
- Ejemplo:

  ?- liga(algol60, L), liga(L, M).

  Significa, ¿existen L y M tales que liga(algol60,L) and liga(L,M) sean verdaderas?
- Una solución a una consulta es aquella asociación de variables con valores que hacen verdadera la consulta.
- Se dice que una consulta con solución puede satisfacerse. El sistema responde con una solución a una consulta que puede satisfacerse:

  L = cpl

  M = bcpl

Opciones a una solución

- Si contestamos a una solución con un RET, Prolog responde *true* para indicar que puede haber más soluciones. Inmediatamente después pide la siguiente consulta.
- Si contestamos con un ; y luego RET, Prolog responde con otra solución o con *false*, si es que no hay más soluciones.
- ?- liga(algol60, L), liga(L, M).

  L = cpl L = simula67 L = simula67 no

  M = bcpl; M = cmasmas; M = smalltalk80;
- Las variables pueden aparecer en cualquier lado dentro de una consulta.

  ?- liga(L, bcpl) ?- liga(bcpl, M)

Nota técnica

- Desde el punto de vista técnico, a todas las variables de una consulta se les aplica el cuantificador de existencia o existencial (∃) de manera implícita.
- Con los cuantificadores en su lugar, la consulta del ejemplo se convierte en:

¿∃L,M. liga(algol60, L) and liga(L,M)?

Hechos y reglas

- La regla

<término> :- <término>$_1$, <término>$_2$, … , <término>$_k$.

para k ≥ 1, corresponde al siguiente pseudocódigo:

<término> **if** <término>$_1$ **and** <término>$_2$ **and** … **and** <término>$_k$.

- El término a la izquierda de :- se llama cabeza (*head*) y los términos a la derecha se llaman condiciones.
- El hecho es un caso especial de una regla que tiene cabeza pero no condiciones.

Ejemplo: definición de una ruta

- El hecho y la regla siguientes especifican una relación **ruta**:

  ruta(L, L).

  ruta(L, M) :- liga(L, X), ruta(X,M).

- La idea es que una ruta consiste de cero o más ligas.
  - Tenemos una ruta de cero ligas de L a L misma.
  - Una ruta de L a M comienza con una liga hacia X y continúa a través de la ruta de X hacia M.

Universal vs Existencial

- Cualquier objeto puede sustituirse por una variable en la cabeza de una regla.
- El hecho ruta(L,L), puede leerse como si tuviera una cuantificador universal:

  Para toda L

  ruta(L,L).

- La regla ruta(L, M) :- liga(L, X), ruta(X,M), tiene una variable X que aparece en las condiciones pero no en la cabeza. Tales variables representan algún objeto que satisface las condiciones, por lo que se puede leer como:

  Para toda L y M,

  ruta(L,M) if

  Existe una X tal que

  liga(L, X) and ruta(X, M).

Nota técnica

- Desde el punto de vista técnico, todas las variables en las reglas y hechos se cuantifican universalmente de manera implícita.
- Con los cuantificadores en su lugar, las regla del ejemplo anterior se escribe como:

  $\forall$L,M,X. ruta(L, M) if (liga(L, X) and ruta(X,M))
- Pero como X no aparece en la cabeza, la regla finalmente queda como:

  $\forall$L,M. ruta(L, M) if ($\exists$X. liga(L, X) and ruta(X,M))

La negación como fracaso

- Prolog responde *false* a una consulta si fracasa en satisfacerla.
- Tomar la negación como fracaso equivale a decir "Si no puedo probarlo, debe ser falso".
- Ejemplo. Los hechos de nuestra base de prueba no dicen nada con respecto a lisp y scheme, de ahí la respuesta de la siguiente consulta:

  ?- liga(lisp, scheme).

  false
- De manera similar, el operador not representa la negación como fracaso, más que como la verdadera negación lógica.
- Una consulta **not**(P) se trata como *true* si el sistema fracasa en deducir P.

Ejemplo

¿Existen dos lenguajes L y M que se liguen al mismo lenguaje N?

Primer intento:

```
?- liga(L, N), liga(M, N)
    L = fortan
    N = algol60
    M = fortran
```

Agregamos el requisito de que las variables L y M deben tener valores diferentes:

```
?- liga(L, N), liga(M, N), not(L=M)
    L = c              L = simula67        false
    N = cmasmas        N = cmasmas
    M = simula67;      M = c;
```

Regla práctica para el not

- Not puede usarse para verificar valores conocidos o que se conocen antes de aplicar el not.
- La consulta reordenada:

```
?- not(L=M) , liga(L, N), liga(M, N)
```

Fracasa porque los valores de L y M no se conocen al inicio de la consulta.

Los valores desconocidos pueden ser iguales, así que not(L=M) fracasa.

Instancia

- Una instancia (ejemplar) de un término T se obtiene mediante la sustitución de términos para una o más variables de T.
- El mismo subtérmino debe sustituir a TODAS las apariciones de una variable.
- Cuando a una variable se le asigna un valor se dice que es instanciada.
- Ejemplo :- f(X,b) = f(a,Y)
  - f(a,b) es una instancia de f(X,b) porque se obtiene al sustituir la variable X en el término f(X,b), por el subtérmino a.
  - De igual forma f(a,b) es una instancia de f(a,Y).
- Ejemplo: g(a,a) es una instancia de g(X,X), y también g(h(b),h(b)), pero no lo es g(a,b), porque X siempre se debe sustituir por el mismo término.

Unificación

- En Prolog, la deducción se basa en el concepto de unificación: dos términos T_1 y T_2 se *unifican* si tienen una instancia común U.
- Si una variable aparece tanto en T_1 como en T_2, entonces el mismo subtérmino debe sustituir a todas las apariciones de la variable tanto en T_1 como en T_2.
- Ejemplo: los términos f(X,b) y f(a,Y) se unifican porque tiene una instancia común f(a,b).
- La unificación ocurre de manera implícita cuando se aplica una regla.

Ejemplo de unificación

- Suponga que la relación de identidad se define por el hecho:

  identidad(Z,Z)

- La unificación se usa para encontrar la respuesta a la consulta:

  ?- identidad(f(X,b),f(a,Y)).

  X = a

  Y = b

- La respuesta se calcula mediante la unificación de identidad(Z,Z) con identidad(f(X,b),f(a,Y)), lo cual lleva a la unificación de Z con f(X,b) y con f(a,Y).

Y como f(X,b) se puede unificar con f(a,Y), se da la respuesta que lo logra.

Aritmética

- Como en todo lenguaje de programación, en Prolog existen operaciones aritméticas.
- El operador = representa la unificación en Prolog (es como un operador lógico de igualdad), por lo que:

  ?- X = 2+3.

  X = 2+3

  Simplemente asocia la variable X con el término 2+3.

- El operador infijo is evalúa una expresión:

  ?- X is 2+3

  X = 5

- Ya que el operador is asocia X con 5, se satisface la consulta:

  ?- X is 2+3, X = 5

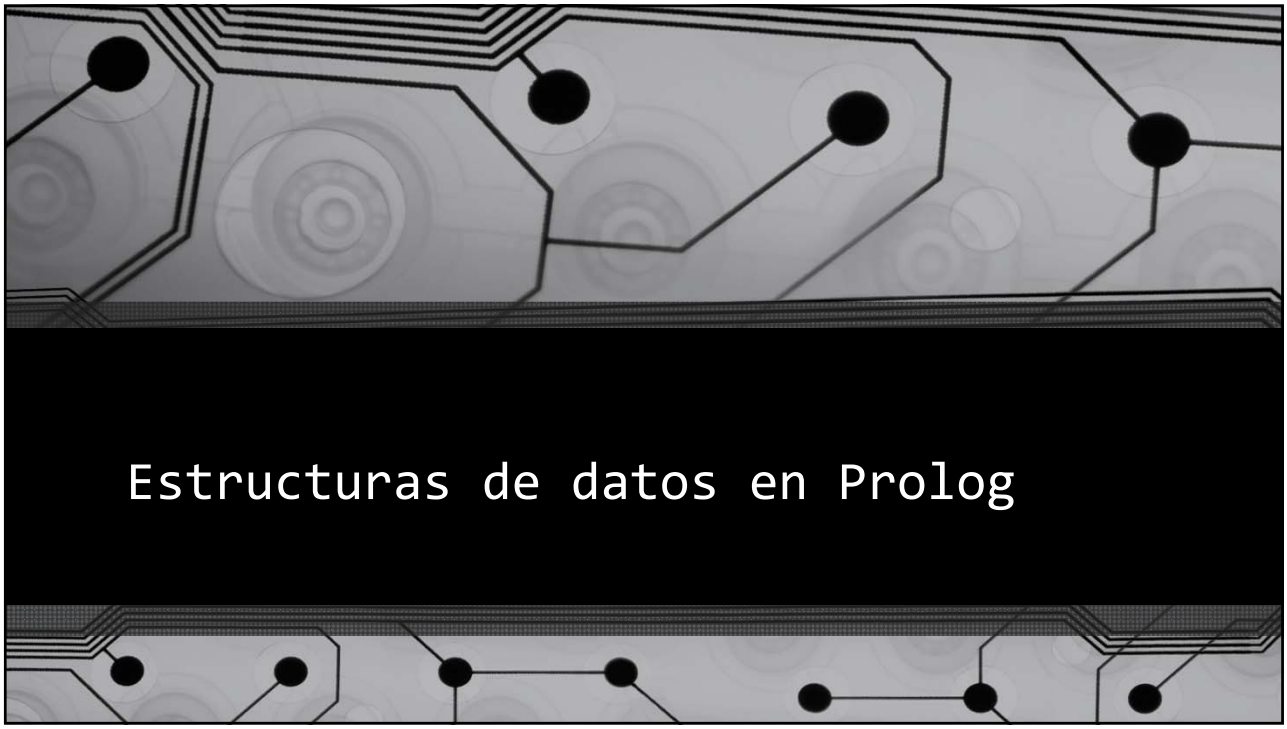
  X = 5

  pero no

  ?- X is 2+3, X = 2+3

  no

  fracasa porque 2+3 no se unifica con 5



Estructuras de datos en Prolog

Listas en Prolog

- Prolog proporciona varias notaciones para escribir listas a la manera de LISP.
- En realidad, estas nuevas formas de escribir listas sólo son un toque sintáctico para los términos ordinarios; es decir, suavizan la sintaxis sin añadir capacidades nuevas.
- La manera más sencilla de escribir una lista es enumerar sus elementos: e.g. [a, b, c]
- La lista vacía se representa como [].

Usando |

- También podemos especificar una secuencia inicial de elementos y una lista restante, separadas por | (pipe).

  La lista [a, b, c], también puede escribirse como:

  [a, b, c | []]

  [a, b | [c]]

  [a | [b, c]]

- Un caso especial de esta notación es una lista con cabeza H y cola T, representado como [H | T].
- La cabeza es el primer elemento de una lista (como el car de LISP), y la cola es la lista que contiene los elementos restantes (como el cdr de LISP).

La unificación en listas

- La unificación puede usarse para extraer elementos de una lista, de manera que no se requieren operadores específicos para extraer la cabeza y la cola.
- La solución a la consulta:

  ?- [H | T] = [a, b, c].

  H = a

  T = [b, c]

  Asocia la variable H con la cabeza y T con la cola de la lista [a,b,c].

- La consulta:

  ?- [a | T] = [H, b, c].

  T = [b, c]

  H = a

  Ilustra la capacidad de Prolog para manejar términos especificados parcialmente.

La relación **agrega**

- La relación agrega sobre listas se define por medio de las siguientes reglas:

```
agrega ([], Y, Y).
agrega ([H | X], Y, [H|Z]) :- agrega (X, Y, Z).
```

- Las siguientes consultas muestran que las reglas para agrega pueden usarse para calcular cualquiera de los argumentos a partir de los otros:

```
?- agrega ([a,b], [c,d], Z).
     Z = [a,b,c,d]
?- agrega ([a,b], Y, [a,b,c,d]).
     Y = [c,d]
?- agrega (X, [c,d], [a,b,c,d]).
     X = [a,b]
```

```
?- agrega (X, [d,c], [a,b,c,d]).
     false
```

Los argumentos inconsistentes son rechazados.

Términos como listas

- La conexión entre las listas y los términos es la siguiente: [H|T] es el toque sintáctico para el término (H, T):

```
?- .(H,T) = [a,b,c].
     H = a
     T = [b,c]
```

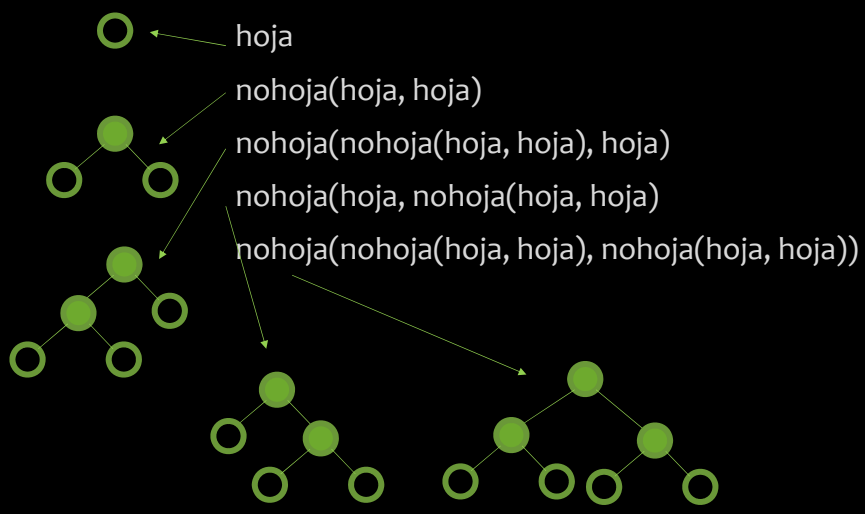
- El operador punto "." corresponde al CONS de LISP, y las listas son términos.
- El término para la lista [a,b,c] es:

```
.(a, .(b, .(c, [])))
```

Árboles y términos

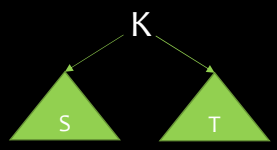
- Existe una correspondencia uno a uno entre árboles y términos, es decir, un árbol puede escribirse como término y un término puede dibujarse como árbol.
- Cualquier estructura de datos con capacidad de simularse usando árboles puede, por lo tanto, simularse mediante el uso de términos.
- Por ejemplo, los árboles binarios pueden escribirse como términos, usando un átomo **hoja** para una hoja y un funcionador **nohoja**, con dos argumentos, para un nodos que no sea hoja.

Ejemplo de árbol binario



Árboles binarios de búsqueda

- Los árboles binarios de búsqueda tienen una contraparte directa en Prolog.
- Consideremos que el átomo **vacío** representa un árbol binario de búsqueda vacío y el término **nodo(K, S, T)**, representa un árbol con un valor entero K en la raíz, un valor S en el subárbol izquierdo y un valor T en el subárbol derecho.



La relación **miembro** en ABB

- Las siguientes reglas definen la relación miembro, para verificar si un entero aparece en algún nodo del árbol.

```
miembro (K, nodo(K, _, _)).
miembro (K, nodo(N,S,_)) :- K < N, miembro(K, S).
miembro (K, nodo(N, _, T)) :- K > N, miembro(K, T).
```

- Los dos argumentos de miembro son un entero y un árbol.
- La primera regla se puede escribir también como:

```
miembro(K,U) :- U = nodo(N,S,T), K = N.
```

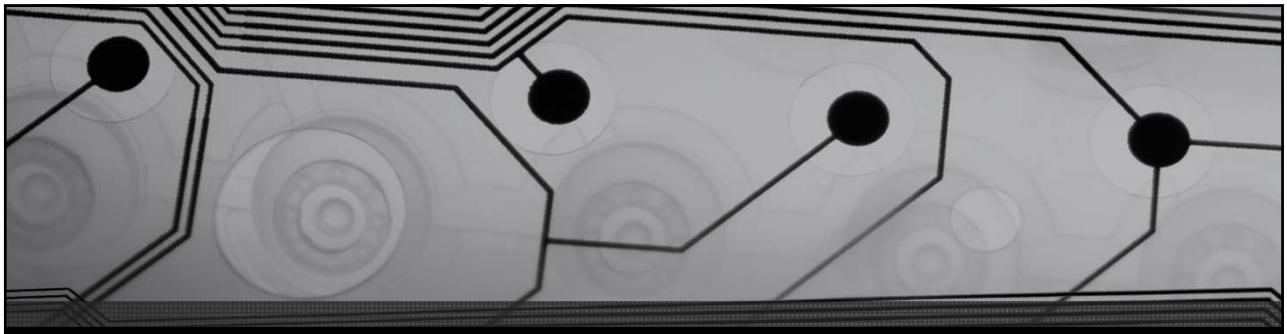
Inserta y borra

- Implementar la relación inserta para un árbol binario de tal forma que:

  inserta K en S para obtener T

- Implementar la relación borra, para eliminar un entero de un árbol.

NOTA: Además de los árboles, las variables en Prolog permiten a los términos representar estructuras de datos compartidas. Por ejemplo, el término nodo(K,S,S) representa el grafo:



Ejercicios

Nota para Ejercicios

- Vamos a hacer varios ejercicios simples.
- Seguiremos los árboles para entender la ejecución.
- Todas las respuestas se entregan por medio de variables.
- Sí se pueden hacer impresiones con instrucciones especiales pero no se usan mucho en Prolog.
- La evaluación de los predicados es de izquierda a derecha.
- Busca predicados en la base de arriba a abajo.
- Puede ser de gran ayuda la instrucción **trace**, la cual permite ver paso a paso la búsqueda.
- Para detener la muestra de la búsqueda use **notrace**.

Ejercicio: Familia

- Usando las relaciones:

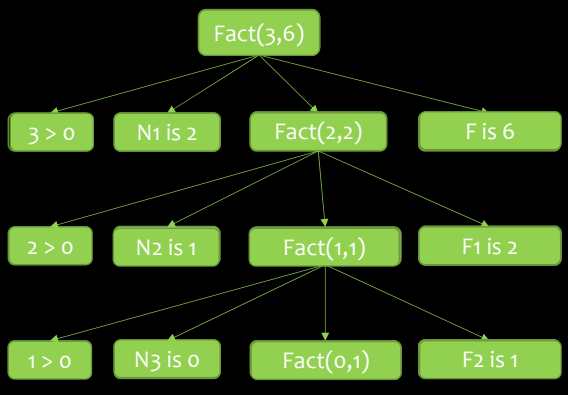
padre(X,Y)	X es el padre de Y	defina las relaciones para
madre(X,Y)	X es la madre de Y	a) hermanos
femenino(X)	X es femenino	b) hermana
masculino(X)	X es masculino	c) nieto
		d) primo
		e) descendiente

- Desde luego que las relaciones definidas se deben tener en una base de hechos. Por ejemplo, **padre(victor1,victor2)** debe estar en la base de datos de mi familia.

Ejercicio: Factorial

- Hacer una relación que corresponda a la función factorial.
- Revise su definición usando un árbol antes de correrlo.

Árbol para fact(3,F)

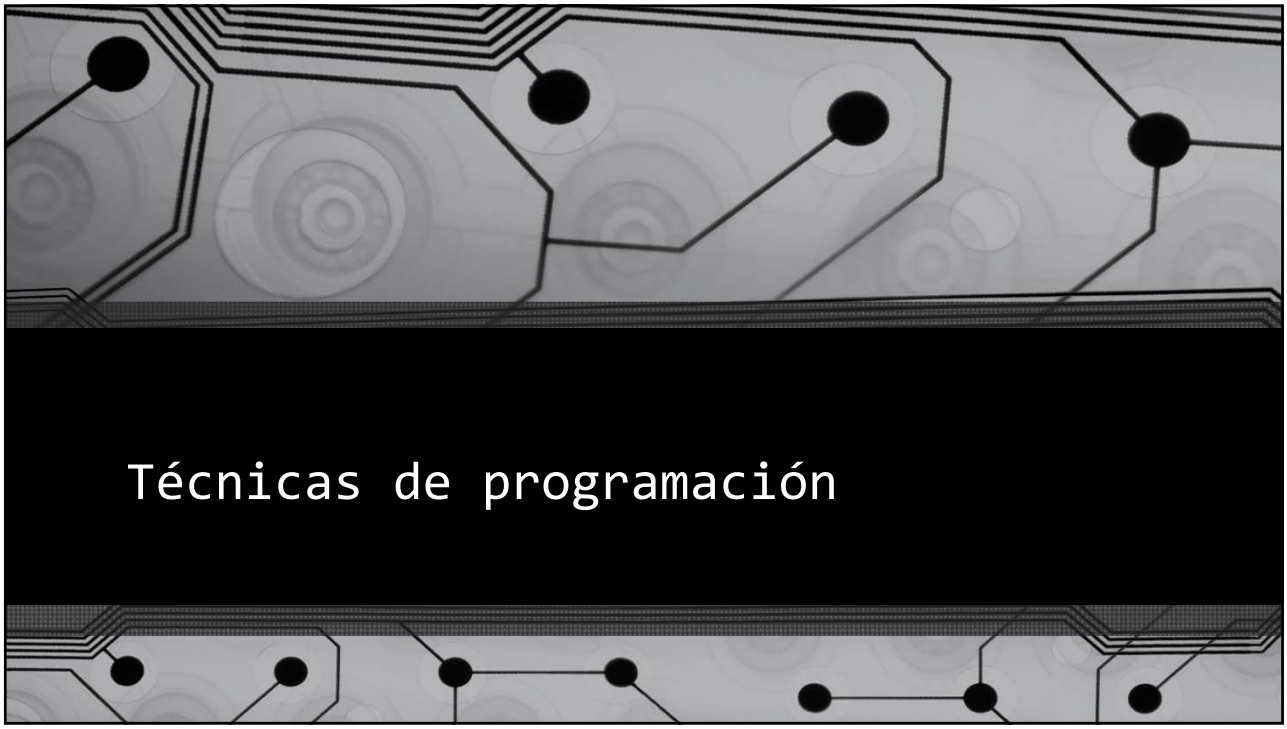


Más ejercicios

1. Hacer una relación que corresponda a una versión con recursión Tail de la función factorial.
2. Hacer una relación que obtenga la potencia del primer número elevado al segundo.
3. Hacer una relación que cuente el número de elementos en una lista.
4. Hacer una relación que indique si un elemento es miembro de una lista.
5. Hacer una relación que entregue una lista al revés.

Tarea 4

- Hacer un programa en Prolog (un conjunto de definición de relaciones), que determine si una lista:
  - Si es un palíndroma (llámela **palindroma**, con un argumento).
  - Tiene un número par de elementos (llámela **longitudPar**, con un argumento).
  - Es una permutación de otra (llámela **esPermutacion**, con dos argumentos).
- Ejemplo:
  - La lista [1,2,3,4,5] es una permutación de [2,1,5,4,3] porque tiene los mismo elementos aunque en diferente orden.
  - La lista [a,b,c,d,c,b,a] y la lista [1,2,3,3,2,1] son palíndromas, porque se leen igual de izquierda a derecha que de derecha a izquierda.
  - La lista [1,2,3,4,5,6] tiene un número par de elementos.



Técnicas de programación

Técnicas de programación

- Las técnicas de programación son una serie de tips que permiten explotar la fuerza de Prolog, es decir, el backtracking y la unificación.
- El backtracking permite encontrar una solución, si es que existe una.
- La unificación permite usar variables como lugares para acomodar datos, lugares que se utilizarán después.
- La idea es que el uso de estas técnicas nos conduzcan a programas eficientes.
- Las técnicas se presentan suponiendo evaluación de izquierda a derecha de las submetas.

Propone y Verifica

- Una consulta "propone y verifica" tiene la siguiente forma:

  Existe una S tal que

  propone(S) and *verifica(S)*?

  Donde *propone(S)* y *verifica(S)* son submetas.

- Prolog responde a tal consulta generando soluciones a *propone(S)* hasta que encuentra una que satisfaga *verifica(S)*.
- Otro nombre de estas consultas es "genera y prueba".
- De forma general, la consulta "propone y verifica" tiene la siguiente forma:

  Conclusión(…) if propone(… , S,…) and verifica(… , S,…)

Ejemplo

traslapan(X,Y) :- miembro(M,X), miembro(M,Y).

- Las listas X y Y se traslapan si existe alguna M que es miembro de ambas listas.
- La primera meta miembro(M,X) es la que propone una solución y la segunda, miembro(M,Y) verifica si M aparece en Y.
- Sugerencia: como los cálculos proceden de izquierda a derecha, el orden de las submetas en una consulta "propone y verifica" puede afectar la eficiencia.
- Ejemplo:

  ?- X = [1,2,3], miembro(a,X) contesta no, mientras que ?-miembro(a,X),X=[1,2,3],tiene un cálculo infinito, número infinito de soluciones: X = [a|_]; X = [_,a |_]; X = [_,_,a |_];

Variables como lugares para acomodar datos

- Hasta el momento, las variables sólo se han usado en reglas y consultas pero también se pueden usar en términos que representan objetos.
- Los términos que contienen variables pueden usarse para simular estructuras de datos modificables; las variables sirven como lugares que serán ocupados para colocar subtérminos posteriormente.
- Tales términos se pueden utilizar, por ejemplo, para implementar COLAS eficientemente (recuerde el uso de los términos para implementar árboles).

Listas abiertas

- La lista [a, b | X], está especificada de una manera parcial porque no sabemos aún qué es lo que X representa.
- A este tipo de listas se les conoce como Listas Abiertas.
- Terminan con una variable conocida como variable de marca final de la lista.
- La lista abierta vacía consiste sólo de la variable de marca final.
- Internamente, Prolog usa variables generadas por la máquina que se escriben con un guion bajo _ al principio, seguido de un entero (SWI no muestra estas variables, simplemente les deja el mismo nombre).

Ejemplos

- En la siguiente interacción, la variable _1, la cual es generada por la máquina, corresponde a la marca de final X.

```
?- L = [a, b | X].          SWI-Prolog contesta:
    L = [a, b | _1]              L = [a, b | X]
    X = _1
```

- Prolog genera variables nuevas cada vez que responde a una consulta.
- Una lista abierta puede modificarse mediante la unificación de sus marcas de final de lista.
- La siguiente consulta hace de L una lista abierta nueva con marca final de Y:

```
?- L = [a, b | X], X = [c, Y].
    L = [a, b, c | _2]
    X = [c | _2]
    Y = _2
```

Asignación

- La unificación de una variable de marca final es semejante a la asignación a esa variable.
- Una ventaja de trabajar con listas abiertas es que se puede tener un acceso al final de lista en un tiempo constante, a través de su marca final, ya que es como un apuntador.
- Esto permite la implementación eficientemente de colas en Prolog.

Colas

- Las reglas para implementar colas son:

```
construye(q(X,X)).
entrada(A, q(X,Y), q(X,Z)) :- Y = [A | Z].
salida(A, q(X, Z), q(Y, Z)) :- X = [A | Y].
enlace(q([], [])).
```

- La relación entrada(a, Q, R) se describe como:
  - Cuando el elemento a entra a la cola Q, obtenemos la cola R.
- En forma similar se describe salida(a, Q y R).
- Cuando se crea una cola se representa por medio de un término de la forma q(L,E), donde L es una lista abierta con marca de final E.

Meta final

- La meta final enlace(U), verifica que las operaciones de entrada y salida dejen a U en un estado inicial q(L,E), donde L es una lista abierta vacía con marca de final E.
- De otro modo, q(L,E) no se unificará con q([],[]).

NOTA: sorpresivamente, las reglas para colas permiten colas con "déficit", en las cuales un elemento puede salir antes de entrar. De manera más precisa, un elemento aún no especificado, representado por una variable, puede salir y después tomar un valor mediante una operación de entrada.

Ejemplos de manejo de colas

Un ejemplo normal:

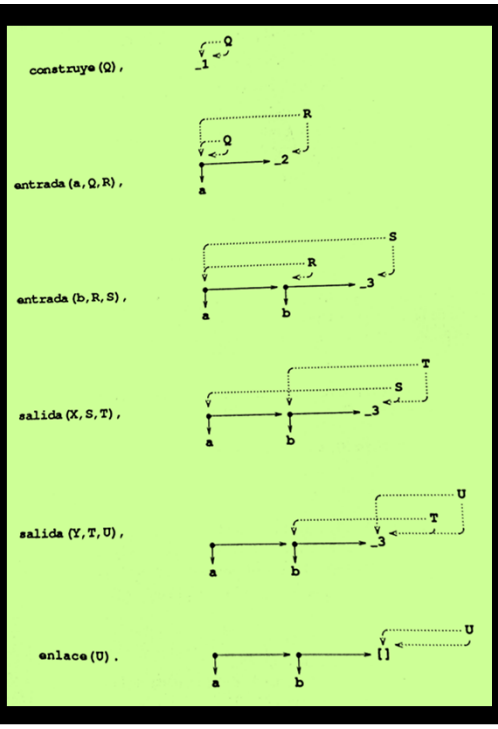
?- construye(Q), entrada(a, Q, R), entrada(b, R, S),
salida(X, S, T), salida(Y, T, U), enlace(U).

U hace referencia a q(_3,_3) y enlace(_3,_3) unifica _3 con [].

Un ejemplo de listas con déficit:

?- contruye(Q), salida(X, Q, R), entrada(a, R, S), enclace(S).

Gráficamente



Listas de diferencias

- Las aplicaciones que usan listas pueden adaptarse para usar en su lugar listas abiertas.
- Las versiones de listas abiertas requieren mayor cuidado , pero pueden ser más eficientes.
- El cuidado es necesario porque una lista abierta cambia cuando su marca de final se unifica.
- Las listas de diferencias son una técnica para agilizar tales cambios.

Notación

- Se representan por un predicado con dos variables, e.g. dl(L,E)
- El nombre del predicado es arbitrario pero lo debe mantener en toda su aplicación.
- Se llaman lista de diferencias porque se construyen a partir de dos listas, L y E, donde E se unifica con un sufijo de L.
- El contenido de las listas de diferencias consta de los elementos que se encuentran en L pero no en E, es decir L-E en notación de conjuntos.
- Las listas L y E pueden ser abiertas o cerradas. Por lo general, son abiertas.

Ejemplos

- Los siguientes son ejemplos de listas de diferencias con contenidos [a,b]:

```
dl([a,b], []).
dl([a,b,c], [c]).
dl([a,b | E], E).
dl([a,b,c | F], [c | F]).
```

Lista de diferencia con listas abiertas

- Debido a que las variables en dl(L,E) no permiten referirnos directamente a los puntos finales de sus contenidos, entonces L y E pueden ser abiertas o cerradas.
- Por lo general son abiertas ya que, de esta forma, las operaciones con listas se simplifican.
- Cuando trabajamos con listas de diferencias abiertas, a la notación dl(L,E), la podemos interpretar como la lista L que tiene como marca de final de lista a E.
- De estas forma, el llamado a este predicado sería, por ejemplo:

```
dl([a, b | F], F)
```

En donde queda muy clara la interpretación anterior.

Ejemplos de operaciones

- La operación agrega (append) sobre listas de diferencias puede implementarse en un tiempo constante usando una regla no recursiva con listas abiertas:
  - Si X se extiende desde L hasta M y Y se extiende desde M hasta N, entonces Z, el resultado de agregar X y Y, se extiende desde L hasta N:

  agrega_dl(X, Y, Z) :- X = dl(L,M), Y=dl(M,N), Z=dl(L,N).
- Un ejemplo de aplicación del append es:

  agrega_dl(dl([a,b|X],X), dl([1,2|Y],Y),Z).

  X = [1, 2|Y],

  Z = dl([a, b, 1, 2|Y], Y).

Sustituciones

- Una sustitución es una función de variables a términos.
- Es un conjunto de elementos de la forma X→T, donde la variable X tiene una correspondencia con el término T.
- A menos que se establezca otra cosa, su una sustitución hace corresponder X con T, entonces la variable X no aparece en el término T.
- Ejemplo {V→[b,c], Y→[a,b,c]}
- Tσ es una notación normalizada para el resultado de aplicar una sustitución σ al término T.

Reglas para la sustitución

- El resultado de aplicar una sustitución a un término está dado por:

$X\sigma = U$	si $X \rightarrow U$ está en σ
$X\sigma = X$	de lo contrario, para la variable X
$(f(T_1,T_2))\sigma = f(U_1,U_2)$	si $T_1\sigma=U$, $T_2\sigma=U_2$

Esta definición generaliza a los operadores funcionales f con k≥0 argumentos.

En otras palabras

- Si σ contiene $X \rightarrow U$, entonces el resultado de aplicar σ a la variable X es U; de lo contrario, $X\sigma$ es simplemente X.
- El resultado de aplicar σ a un término $f(T_1, \ldots, T_k)$, para k≥0, se obtiene aplicando σ a cada subtérmino.
- Ejemplo

  $Y\{V \rightarrow [b,c], Y \rightarrow [a,b,c]\} = [a,b,c]$

  $Z\{V \rightarrow [b,c], Y \rightarrow [a,b,c]\} = Z$

  $(agrega([], [a|V], Y)\{V \rightarrow [b,c], Y \rightarrow [a,b,c]\} = agrega([],[a,b,c],[a,b,c])$

Unificación

- El término U es un ejemplar de T, si U = Tσ, para alguna sustitución σ.
- Los términos T_1 y T_2 se unifican si $T_1\sigma$ y $T_2\sigma$ son idénticos, para alguna sustitución σ.
- Llamamos a σ unificador de T_1 y T_2.
- La sustitución σ es el unificador de máxima generalidad de T_1 y T_2 si para todos los demás unificadores σ', $T_1\sigma$ es una instancia de $T_1\sigma'$.

Ejemplo

- Los términos:

agrega([], Y, Y) y agrega([],[a | V],[a,b,c]

se unifican porque ambos tienen en común el ejemplar:

agrega([],[a,b,c],[a,b,c])

- Su unificador de máxima generalidad es la sustitución:

{V→[b,c], Y→[a,b,c]}

Cortes

- Informalmente, un corte (cut) reduce o "corta" una parte inexplorada de un árbol (poda el árbol) de búsqueda de Prolog.
- Por lo tanto, los cortes pueden utilizarse para hacer más eficiente un cálculo mediante la eliminación de búsquedas infructuosas y bracktrackings inútiles.
- Los cortes también pueden usarse para implantar una forma de negación, algo que las cláusulas de Horn no pueden hacer.
- Los cortes son controversiales porque son impuros porque hace que Prolog se aleje más de la lógica pura (como el orden) al grado de que un programa debe leerse en términos de sus cálculos.

Definición del corte

- El corte se representa con !
- Aparece como una condición dentro de una regla.
- Cuando la regla

$$B :- C_1, \ldots , C_{j-1}, !, C_{j+1}, \ldots, C_k$$

se aplica durante un cálculo, el corte indica a Prolog realizar backtracking de $C_{j-1}, \ldots, C_1$, B sin considerar ninguna de las reglas restantes.

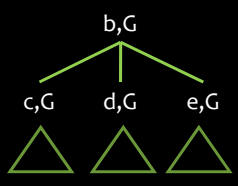
Un corte como primera condición

- Considere una regla de la forma B:-!,C, en la cual un corte aparece en la primera condición.
- Si la meta C falla, entonces el control realiza backtracking hasta B sin considerar ninguna de las reglas restantes para B.
- De esta forma, el corte tiene el efecto de llevar al fracaso a B si falla C.

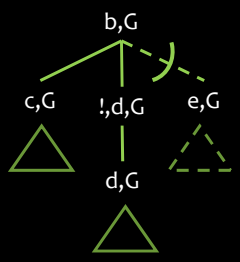
Ejemplo de corte

- Suponga que la condición en un nodo es b,G, donde G representa algunas submetas adicionales.
- Suponga además las tres siguiente reglas para b:

b :- c.
b :- d.
b :- e.



b :- c.
b :- !, d.
b :- e.



- El corte como primera submeta en !,b,G se satisface inmediatamente, dejando a d,G como la nueva meta a ser cumplida.
- Durante el backtracking el corte tiene el efecto lateral de no considerar la tercera regla b:-e.

Otro ejemplo

- Suponga la siguiente BD:

  a(1) :- b.

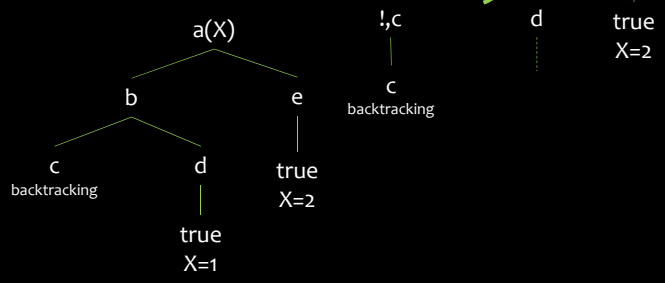
  a(2) :- e.

  b :- c.

  b :- d.

  d.

  e.



- La consulta tiene dos soluciones (probarla).
- Si la regla b:-c se cambia por b:-!,c el árbol es podado y sólo da una solución

El corte en medio

- En el siguiente ejemplo se considera el efecto de insertar un corte en medio de una regla del tipo "propone y verifica".
- Recordemos que el lado derecho de una regla de este tipo tiene la forma propone(S), verifica(S), donde propone(S) genera resultados potenciales hasta que se encuentra uno que soluciona verifica(S).
- El efecto de insertar un corte entre esas reglas, como en:

  conclusión(S) :- propone(S), !, verifica(S).

  Es eliminar todas las propuestas excepto la primera.
- Probarlo en el factorial.

Aplicaciones de cortes en programación

- Un uso relativamente benigno de los cortes es podar partes de un árbol de búsqueda de Prolog que quizá no han de llegar a una solución.
- Tales cortes se conocen como cortes verdes; hacen eficiente el programa sin cambiar sus soluciones.
- Los cortes que no son verdes se conoce como rojos.
- Al restringir el backtracking, los cortes reducen los requerimientos de memoria de un programa.
- Sin los cortes, cada aplicación simple de una regla y cada unificación tienen que almacenarse hasta que el cálculo general tenga éxito o se genere un backtracking.

La negación como fracaso

- En Prolog, el operador **not** se implementa con las reglas:

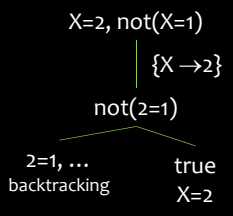
```
not(X) :- X, !, fail.
not(_) .
```

- La primera regla trata de satisfacer el argumento **X** de **not**.
- Si la meta X tiene éxito, entonces el corte y **fail** se alcanzan.
- El constructor fail fuerza al fracaso y el corte evita quye se considere la segunda regla.
- Por otro lado, si la meta X fracasa, entonces la segunda regla tiene éxito porque _ se unifica con cualquier término.

Ejemplo de negación

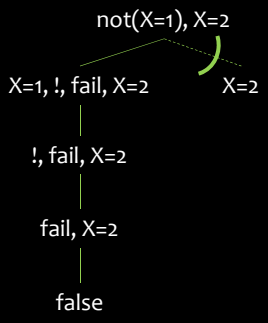
?- X = 2, not(X=1).

X = 2



?- not(X=1), X=2.

no



Explicación primera consulta

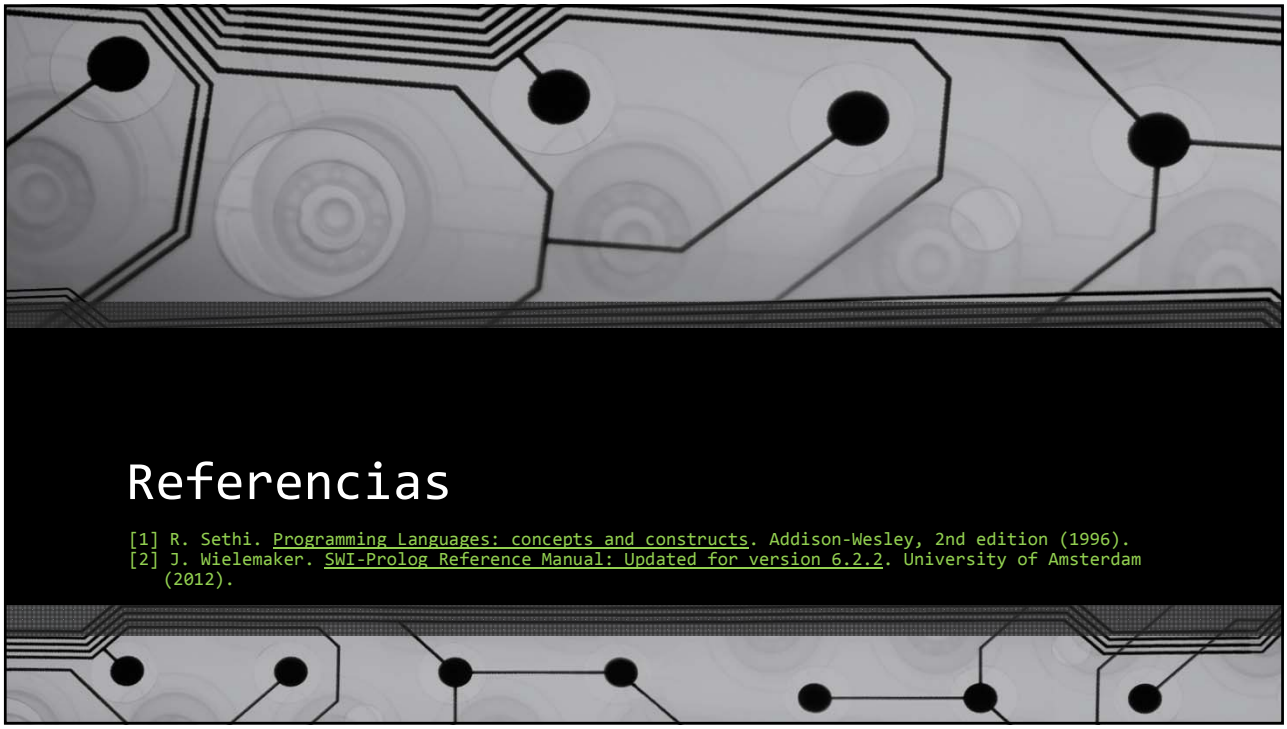
- En la primera de estas dos consultas, la submeta X=2 unifica a X con 2, dejando a not(2=1) como meta actual.
- La primera regla del not nos lleva a la nueva regla 2=1,!,fail.
- Como 2=1 fracasa, no se alcanza el corte (a partir de aquí el corte no se muestra en el árbol).
- Después se aplica la segunda regla del not y la meta not(2=1) tiene éxito.

Explicación segunda consulta

- En el árbol de la segunda consulta, la primera regla del not nos lleva a la meta actual **X=1, !, fail, X=2**.
- La submeta **X=1** tiene éxito, se satisface el corte y se alcanza **fail**.
- El corte elimina la posibilidad de considerar la otra regla de not, de manera que todo el cálculo fracasa.
- Observe que la meta **X=2** nunca se alcanza.
- En general, es seguro aplicar **not** a un término sin variables ya que tales términos no tiene variables que cambien por una unificación.

Proyecto 2

- Dada una BD con los siguientes hechos:
  - Caminos de una ciudad a otra y su distancia:
  - Ej. camino(A,B,24).
- Encontrar el camino más corto de una ciudad a otra.
- Seleccione un algoritmo para ello y explíquelo en la Introducción (e.g. Dijkstra, A*, DFS, BFS, etc.)
- El programa debe entregar una lista ordenada con las ciudades que se deben visitar, incluyendo el origen y el destino.
- El orden de la lista es el orden de visita de las ciudades.



Referencias

[1] R. Sethi. Programming Languages: concepts and constructs. Addison-Wesley, 2nd edition (1996).
[2] J. Wielemaker. SWI-Prolog Reference Manual: Updated for version 6.2.2. University of Amsterdam (2012).