Introducción al PLN mediante la generación de gramáticas semánticas

Departamento de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

J.A. Bañares

29/11/10

Índice

1. Gramáticas Semánticas
2. Dypar:
   - Intérprete de **lenguaje natural basado en reglas**
3. Escribiendo una gramática semántica en Dypar
4. Lenguaje de Representación basado en frames

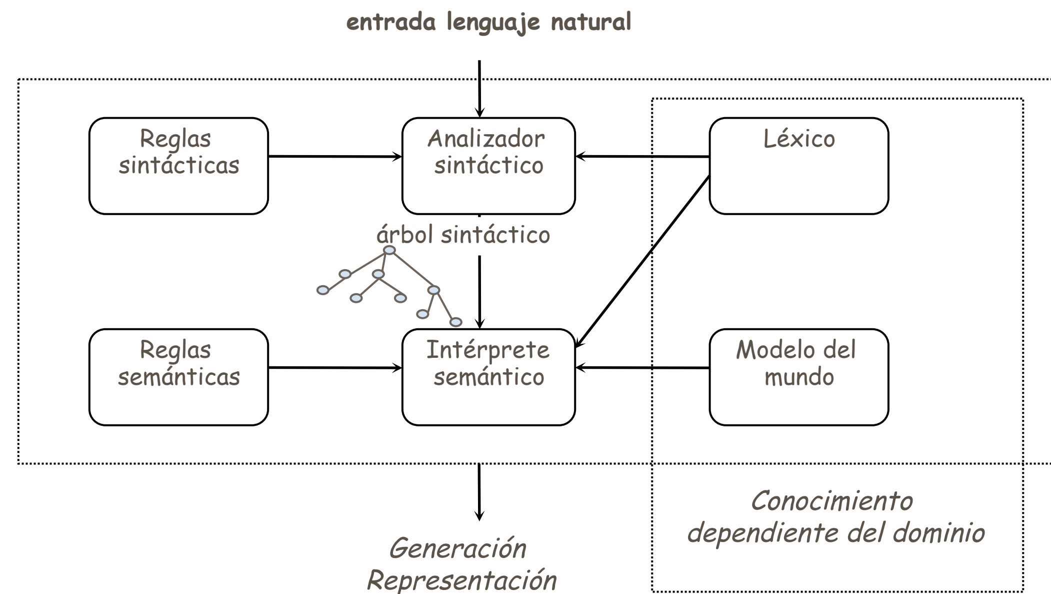
Gramática Semántica

- Análisis Semántico: *Proceso por el cual se crean representaciones del significado y se asignan a las entradas lingüísticas.*
- Gramáticas Semánticas: *Se basan en gramáticas escritas para servir las necesidades semánticas en lugar de las generalidades sintácticas.*

Sistemas de Gramáticas Semánticas

Las categorías gramaticales no corresponden necesariamente a conceptos sintácticos sino que pueden acarrear información semántica

Ganamos en conocimiento de un dominio
Perdemos en generalidad



Facilidad de construcción

- Las gramáticas semánticas son fáciles de generar:
  - Definición de formas generales que tendrán las preguntas o instrucciones (plantillas)
  - Introducción de la terminología específica del dominio

Lenguajes de representación del significado

- Se precisa un lenguaje de representación que permita alguna forma de procesamiento semántico:
  - Responder un examen tipo test
  - Darse cuenta de que uno está siendo insultado
  - Seguir una receta
- Lenguajes de representación
  - Capacidad expresiva
  - Capacidad de inferir

Lenguajes de representación

- Lenguajes formales: Lógica de predicados de primer orden
- Lenguajes basados en reglas
- Representaciones estructuradas
  - Redes semánticas
  - Lenguajes basados en frames
- Otros: Bases de datos (SQL)

Frames

- **Frames**, para representar el conocimiento **de la estructura** del dominio
  - Permiten **construir taxonomías** de conceptos por especialización de conceptos generales en conceptos más específicos
  - La **herencia** permite compartir propiedades y evita la presencia de conocimiento redundante.
  - Las propiedades se pueden representar de forma **declarativa** o **procedimental**.
  - La estructura interna de los marcos permite mantener internamente las **restricciones de integridad semántica** que existen entre los elementos de un dominio.
  - Facilitan el diseño y mantenimiento de la BC
  - Permiten representar **valores por omisión** y **excepciones**.
  - Las redes semánticas no incorporan información procedural, y son difíciles de comprender y mantener cuando aumenta la BC.

Logica/Reglas

- **Representación declarativa del conocimiento:**
  - **Reglas** para representar conocimiento de sobre el comportamiento, es decir conocimientos sobre la resolución del problema
  - **Logica** (Desventajas): No puede tratar incertidumbre, razonamiento monótono, uso exhaustivo de search & matching.

Comparación reglas/Frames

- El conocimiento sobre la *resolución* de un problema también se puede representar con frames
- Los sistemas basados en frames, pueden inferir dinámicamente en tiempo de ejecución, el valor de una propiedad utilizando valores de otras propiedades utilizando demonios
- Los frames hacen uso de demonios para especificar acciones que deberían realizarse frente a ciertas circunstancias
  - Los demonios permanecen inactivos hasta que se solicitan sus servicios
  - Las reglas son chequeadas continuamente

Introducción a Dypar

1. Finalidad y estrategia de Dypar
2. Reglas y Operadores Básicos
   - Tipos de reglas
   - Patrones y operadores
   - Reglas de reescritura
   - Reglas de alto nivel
   - Reglas de transformación
3. Otros Operadores
   - Operadores de disyunción, repetición, equiparación, comodines, negación.
   - Variables
4. Funciones de Dypar y variables del sistema

1. Finalidad de Dypar

- Interprete de lenguaje natural basado en reglas adaptable a dominios particulares
  - Interrogación de bases de datos
  - Interpretación de instrucciones
  - Interrogación a sistemas expertos/SBC

Finalidad de Dypar

- Objetivo de los interfaces en LN
  - Cubrir aspectos del interfaz en lenguaje natural perteneciente a las tareas realizadas por el sistema correspondiente.
  - No es un sistema para analizar (parser) textos en lenguaje natural de cualquier dominio y sin restricciones.

Estrategias de Dypar

- Tres estrategias de análisis
  - Reconocimiento de patrones recursivo
  - Interpretación de gramáticas semánticas
  - Transformación de cadenas
- No es preciso ser un lingüista para desarrollar las gramáticas y codificarlas en Dypar

2. Reglas y operadores de Dypar

- El desarrollador de la gramática decide que clase de sentencias debe reconocer el *parser*
- Las gramáticas de Dypar son una combinación de reglas, patrones y operadores sobre patrones

2.1. Tipos de reglas

- Hay tres tipos distintos de reglas en Dypar
  - Reglas de reescritura
    - Consta de **plantillas** que asocian **no terminales** entre sí, o **no terminales con terminales**.
  - Reglas de alto nivel
    - Consta de plantillas de sentencia y **acciones** asociadas
    - Denominaremos reglas de recuperación a las de alto nivel que en la parte acción sólo aparecen mensajes que le indican al usuario que la frase de entrada no se ha comprendido
  - Reglas de transformación
    - Consta de plantillas de sentencia y acciones asociadas que **transforman la cadena**

2.2 Patrones y Operadores

- Una regla consta de un patrón y una acción
- Un patrón es una lista de elementos o símbolos que constituye el modelo de una frase
  - **Terminales**: Son los que se equiparan directamente con la entrada
  - **No terminales**: Son llamadas a una *regla de reescritura*. Por convenio se escriben entre “<” y “>”

Operadores básicos

- El patrón más simple contiene solo símbolos terminales:

```
(el primero)   ; minúsculas sólo
```

- Patrones alternativos |

```
(el(primero|segundo|tercero))
```

- Símbolo opcional

```
(?el(primero|segundo|tercero))
```

Operadores básicos

- Nuestro comodín $

```
(?una casa $)
```

  - Acepta entradas como “una casa grande” “una casa bonita”, etc.
  - No acepta “una casa muy grande”

Operadores básicos

- Comodín para número arbitrario de entradas *

```
(*(guerra | hambruna |desastre))
```

  Acepta entradas como “guerra guerra guerra”, “desastre guerra hambruna”, etc.

- Se puede utilizar junto a $

```
(* $)
```

Operadores básicos

- El operador asignación :=
  - Permite guardar en una variable el valor de la equiparación.
  - Las variables por convenio, son símbolos precedidos por el símbolo de admiración “!”

```
(?(un | una) (!elemento := $))
(!deportista := (atleta | futbolista |
   tenista))
```

  - Se puede acceder desde el entorno LISP a las variables asignadas.

Reglas de reescritura

- Reglas de reescritura **->**
  - Asocia no terminales entre sí, o no terminales con terminales.
  - La parte izquierda debe ser un no terminal

```
<frase> ->(<sujeto> <predicado>)
<predicado> ->(<verbo> <complementos>)
<sujeto> ->(Pedro)
<verbo> ->(ha invertido)
<complementos> ->(en bolsa)
```

Reglas de reescritura

- Reglas de reescritura con operadores

```
<atleta> -> (deportista | atleta | jugador)
<articulo> ->(un | el)
<un_deportista> ->
        (?<articulo> (!deportista:= <atleta>))
```

- Permite escribir gramáticas jerárquicamente
  - Esto evita escribir patrones excesivamente complejos

2.3 Reglas de reescritura

- Se permite recursividad en las reglas de reescritura, pero solo por la **derecha**

```
<enfasis> -> (?muy <enfasis>)
```

Acepta entradas como muy muy muy ...

```
<mal_enfasis> ->(?<mal_enfasis> muy)
```

No es correcto, recursividad por la izquierda

- Cuando DYPAR carga la gramática detecta la recursividad por la izquierda.

2.4 Reglas de alto nivel

- Reglas de alto nivel =>
- Contienen una secuencia de patrones y la acción a realizar si se reconocen
  - La acción es una expresión LISP que puede utilizar variables DYPAR anteriormente asignadas

```
(<persona> <ser> (!sexo := <sexo>)
 =>
(print !sexo)
```

2.5 Reglas de Transformación

- Reglas de transformación **::>**
- Transforma ciertas frases en otras. Su función es simplificar la labor del constructor de la gramática.
  - La parte izquierda es un patrón y la derecha es una expresión LISP que reconstruye una frase (CONS, APPEND, etc.)

Reglas de Transformación

- Se deben utilizar cuando sustituye a cinco o más reglas de alto nivel
- Dypar intentará transformar una frase cuando haya probado antes la reglas de alto nivel y todas han fallado
  - Las reglas de transformación permiten que una misma regla de alto nivel reconozca diferentes versiones de una sentencia.
    - Menos reglas =>menos tiempo de reconocimiento

Orden de las reglas

- Las reglas de transformación se prueban en el orden en el que han sido escritas.
  - Si una regla de transformación **`A`** utiliza el resultado de otra regla de transformación **`B`**, **`A`** debe escribirse después de **`B`**.
- Las reglas de reescritura y de alto nivel pueden aparecer en cualquier orden porque se buscan por las referencias cruzadas.

Resumen reglas

Tipo de regla	Símbolo	LHS	RHS
Reescritura	->	<nombre>	patrón
Alto nivel	=>	patrón	acción
Transfor.	::>	patrón	transform. sentencia

3. Otros Operadores

- Equiparación de diferentes patrones
  - **!** Prueba por diferentes ramas de forma paralela
  - **!!** Si una rama tiene éxito no trata de equiparar por otras

```
(<patron> -> ((a b c ! a b ! a) v)
(<patron> -> ((a b c !! a b !! a) v)
```

  - **&m** Quick Disjunction

```
(&m is are be am)    ;; (is | are | be | am)
```

Operadores Opcionales

- **?** Comprueba en paralelo tanto si su argumento existe como si no
- **&O** Si su argumento existe continúa la equiparación.

```
(<patron> -> ((&o a b) a b)
        ;; No equipara con (a b)
(<patron> -> ((&o thank you) very much)
(<patron> -> (?(a b) a b)
        ;; Equipara con (a b)
```

Ejemplo

```
<saludo> ->
   (?muy(buenos!Buenas)
   (dias ! tardes ! noches) !! Hola)
```

Operadores Repetitivos

- No son muy eficientes
- * Equipara una entrada un número arbitrario de veces (incluido 0)
- + Equipara al menos una vez la entrada
- ^ Equipara un número determinado de veces la entrada

Ejemplos patrones repetitivos

```
<numero> -> ($n)  ;; $n equipara numeros
<secuencia-de-numeros> -> (* <numero>)
<sec-de-num> -> (+ <numero>)
<numero-de-telefono> (^9<numero>)
```

- Es mas eficiente la definición recursiva

```
<num-seq> -> (<numero>?<num-seq>)
```

Ejemplos patrones repetitivos

- Puede observarse que

```
(?(*patron)) y (*(?patron))
(&o(*patron)) y (*(&o patron))
(?(+patron)) y (+(?patron))
(&o (+ patron) y (+(&o patron))
(^n(&o patron))
```

Son equivalentes a `(* patron)`

Comodines

- **`$`** Equipara cualquier palabra, número, etc. o grupo de ellos.
- **`$n`** Equipara cualquier número
- **`$w`** Equipara cualquier palabra
- **`$d`** Equipara cualquier símbolo terminal dentro de un conjunto determinado denominado diccionario. Toda palabra que esté en la gramática, está en el diccionario.
- **`$r`** Equipara cualquier cosa hasta el final de la entrada.
- **`$p`** Equipara cualquier carácter de puntuación.

Ejemplos

```
<patron> ->
 ($ numero de teléfono es <num-telef>)
<patron> ->
 ($w numero de teléfono es $n)
<patron> ->
 ($w (número || números) de $d (es ||
  son) $r)
```

Equiparación

- Selección del comienzo de la equiparación, saltándose el comienzo de parte de la entrada
  - `&u` Equipara cualquier cosa hasta un determinado símbolo de entrada sin incluirlo.
  - `&ui` Igual que el anterior, salvo que se sitúa detrás de esa entrada.

Ejemplos

```
<show> ->
  (show | list | print ?out | give)

(&u <show>)
```

"could you please show me ..."

```
(&ui <show>)
```

"could you please show me ..."

Negación

- `~` Equipara cualquier cosa después de comprobar que no es la misma que el patrón
- `&n` Comprueba que no aparece la entrada, pero no la consume

Ejemplo

```
<patron> ->
 ((~$n) <resto-equiparacion>)
  ;Reconoce cualquier cosa excepto un
  número
<patron> ->
 ((&n(un||una)<digito>
   <resto-equiparacion>)
  ;Reconoce cualquier cosa excepto
  "un" o "una" y después sigue la
  equiparación
```

Prueba

- **&s** Comprueba que va a aparecer una entrada sin llegar a consumirla.
  - La equiparación continuará si está presente esa entrada y si no lo está fallará

```
((&s exit) &r) => 'exit
```

`Si en algún momento de la entrada se escribe exit se sale del parser.`

```
((&s time) ...) =>
```

Morfología

- Se utiliza para reconocer un única raíz junto a distintos sufijos.

`(small | smaller)` es equivalente a

```
(&morph :root small :suffix (er | est)

(&morph :root (!wrd:=(candy | cook)
        :ending (ed | ing))
```

El patrón reconocerá candied, cooked, candying, cooking) y guarda la raiz reconocida.

3.2 Variables

- **Asignación**
  - **:=** Asigna el valor de una equiparación con éxito a una variable que después se vaya a utilizar. Sino es equiparable la variable toma el valor NIL

```
(!var:= <expresion>)
<patron> -> (!tel:=<num-tel>)
```

Variables

- **Asignación**
  - **&i** Una o varias entradas se asignan a un mismo valor. Este a su vez puede ser asignado a una variable

```
(&i valor patron)
```

Variables

Donde

- Si el patrón es un átomo se trata de la agrupación de una serie de términos en uno

```
<patron> -> (!tel := (&i telefono
   (telefono !! Tel !! Tlf !! Telf !! Tfno))))
```

- Sino aparece nungún patrón, se trata de una inicialización de una variable

```
<patron> -> (!provincia := (&i madrid))
```

Variables

- Se puede obtener el valor aplicando una funcións LISP a los argumentos:

```
((!fraction :=
   (&i (&funcall divide (!divisor !dividendo))
    <fraccion>))
<fraccion> -> ((!divisor := $n) %slash
               (!dividendo :=$n))
```

4. Funciones de Dypar

- Nuevos Operadores
  - Operador para iniciar varias variables a un mismo valor

    ```
    (&a !var-nueva !var-vieja[valor-por-defecto])
    ```

  - Carga de un fichero que contenga na gramática

    ```
    (&file nombfich1 nombfich2 nombfich3 ...)
    ```

Ampliación de Dypar

- Prueba de gramáticas al principio del fichero

```
(&test ((entrada1) (salida-esperada1))
         ((entrada2) (salida-esperada2))
         ((entrada3) (salida-esperada3))
              ...
         ((entradan) (salida-esperadan))
```

Donde `salida-esperadai=`

```
(nombre-reglaj (!var1 val1) ...(!varn valn))
```

*Variables del Sistema

- ***Language*** Puede tomar los valores "english" o "spanish" que son los dos idiomas en los que pueden aparecer los mensajes
- ***enable-output*** Es un flag que cuando tiene valor NIL elimina los mensajes que produce Dypar.
- **!show-expanded** controla la aparición d o no en pantalla de puntos que indican el número de expansiones de símbolos no te´rminales que se han realizado.
- **!ptrace** controla mensajes referentes a la traza de ejecución de reglas.

Variables del sistema

- ***interfaz*** Activa el interfaz gráfico durante el desarrollo (no disponible).
- ***nombres-var*** Nombre de todas las variables que han tomado valor durante el proceso de equiparación.
- **!success-flag** si vale T guarda en el fichero success.dat todas las frases que han sido reconocidas.
- **!failure-flag** guarda en failed.dat las frases no reconocidas..

Funciones del sistema

- **(showgra)** Muestra los símbolos correspondientes a las reglas así como terminales aparecidos durante la carga de la gramática.
- **(erasegra)** Borra gramáticas cargadas.
- **(loadgra** *&rest nombre-gramáticas***)** carga las gramáticas
- **(recorre-tabla &key** :busca**)** Describe la estructura que ha creado el sistema para cada uno de los símbolos y reglas de alto nivel y transformación

```
(recorre-tabla :busca t)
```

Funciones del sistema

- **(print-pant-ayuda)** Se puede poner en la parte acción de una regla de alto nivel. Muestra frases tipo en fichero grafic.lsp

```
(dame ayuda || ayudame || socorro)=>
        (pint-pant-ayuda)
```

- **(escribe-mensaje &rest *acciones*)**

```
(<patron-ambiguo>)=>
  (escribe-mensaje
  (format t "A que tipo de prestamo te
   refieres?")
   (setq *tipo-prestamo* (read)))
```

Construyendo una gramática en Dypar.

Índice

1. Presentación del problema
2. Red Semántica de Ejemplo
3. Trabajo Preliminar
4. Escribir la gramática
   - Preguntas IS_A
   - Preguntas Propiedades
   - Aserciones
   - Transformaciones
5. Depurando gramáticas

1. Objetivo

- El objetivo de esta gramática es utilizar Dypar como interfaz a una red semántica
- Pasos:
  - Conocer el dominio
  - Escribir la gramática

2. Red Semántica de Ejemplo

- ¿Que se puede hacer con la red semántica?
  - Almacenar y recuperar relaciones is-a

```
Maria is a painter
¿What is Maria?
```

  - Cruzar jerarquías is-a

```
Fido is a dog
Dogs are canines
Fido is a canine
```

Red Semántica de Ejemplo

- Almacenar y recuperar propiedades de nodos

Bob's pencil is short

What do you know about Bob's pencil

- Borrar, alterar propiedades

Bob's pencil is long

Forget about Bob's pencil

Mary's mother is Sally

- Hacer inferencias

The opositte of mother is daughter

The daughter of Sally is Mary

Red Semántica de Ejemplo

- Limitaciones de la red semántica
  - No puede manejar relaciones is-a múltiples

Jhon is a dentist, Jhon is American

  - Información "adjetival" compleja

Large furry happy brown dog

Funciones de la red semántica

- `Lmt-ret` (Long Term Memory RETrieval)
  - Extrae información de la red semántica
- `Lmt-ret-all` (Long Term Memory RETrieval ALL)
  - Extrae toda la información de un nodo
- `Lmt-store` (Long Term Memory STOREage)
  - Guarda información en la red
- `Lmt-SPECify` (Long Term Memory SPECify)
  - Decide que interpretación de una sentencia se debería almacenar en la red
- `Storefile`
- `loadfile`

3. Trabajo preliminar

- Desarrollar un corpus de ejemplos
  - Aserciones

```
Mary is a woman.
John's mother is Mary.
The inverse of  mother is son.
Fido is a lazy dog.
Sugar is an ingredient of cookies.
```

Trabajo preliminar

- Peticiones de información

```
What is Mary?
Is Mary a woman?
Who is John's mother?
Is Mary John's mother?
Tell me all you know about Mary
What is the inverse of son?
```

- Instrucciones

```
Save this sesion.
Load the number.gra file.
Exit the parser.
Forget about Mary.
```

Trabajo preliminar

- Identificar más rigurosamente el tipo de entradas que vamos a reconocer
  - Sentencias que traten con la relación **is-a**
  - Propiedades particulares de un **objeto**
  - Preguntas derivadas de las dos anteriores y sobre todo lo que se conoce de un **objeto**
  - Instrucciones para guardar y cargar la red semántica
  - ***Eliminar*** información
  - ***Salir***
- Preparar plantillas para el objetivo que nos hemos planteado informalmente

Escribir la gramática

- Preguntas sobre relaciones IS-A

  What is Mary?, What is _____?

```
(what is $)
```

4.1 IS-A

- Preguntas sobre relaciones IS-A

  What is Mary?, What is ______?

```
(what is (!nam := $))
```

What is Mary?, What is ______?

```
(what is (!nam := $))
=>
(ltm-ret !nam 'isa: nil nil)
```

- No terminales en lugar de `is`

```
<be-present> ->(is | are | be | am)

<have-present> ->(have | has)
<have-past> -> (?<have-present> had)
<have-future> -> (will have ?had)
<have> ->(<have-present>|<have-past>|
          <be-future>)
```

- No terminales en lugar de `is`

```
<be-past> -> (was | were |
                <have-present> been| had been)

<be-future> -> (will be | will have been)
<be> -> (<be-present> | <be-past> |
         <be-future>)
```

- No terminales en lugar de *what*

```
<q-word> ->
    (what | who | where | when | how | why |
     how much | how many | how come)

<www> -> (what | who | which)
```

- Las preguntas en inglés pueden aparecer con apóstrofos

  What's o *Who is*

```
<poss> -> (%apost s)
<what-q> -> (<www> <be-pres> | <www> <poss>)
                        ;; reconoce 36  frases distintas
```

- Más formas de hacer preguntas

Could you tell me ...

Can you give me ....

(positive modal auxiliaries)

```
<pos-modal> -> (could | whould | can)
<polite> -> (<pos-modal> you)
```

; como podemos intercambiar me y us

```
<me-us> -> (me | us)
```

```
<info-req1> ->(?<polite><info-req2> ?<what-q>)
<info-req2> ->(tell <me-us> ?about |
               give <me-us> | print | type )
<ínfo-req3> -> (<www> |
               ?<polite> <into-req2> ?<www>)
<info-req> -> (<what-q> | <info-req1>)
```

No Terminal	Posibles reconocimientos
<info-req>	1122
<info-req1>	1084
<info-req2>	8
<info-req3>	152

What is Mary?, What is _____?

```
(<info-req> (!nam := $))
=>
(ltm-ret !nam `isa: nil nil)
```

- Preguntas sobre relaciones IS-A

  What is Mary?, What is _____?

```
(<info-req> (!nam := $))
=>
(ltm-ret !nam `isa: nil nil)
```

What is Mary?, What is the_____?

What is the dog? What is a horse?

```
(<info-req> (!nam := $))
=>
(ltm-ret !nam `isa: nil nil)
```

```
<a-an> -> (a|an)
<bulk> -> (bulk|majority|graty part)
<universal-quant> ->
    (?almost all|?almost every ?one
     |each| most|many|the <bulk> of)
<det> ->
    (the|<a-an>|<universal-quant>))
```

```
<punct> -> (%qmark |<dpunct>) ;?
<dpunct> -> (%period |%emark) ;!
```

- Preguntas sobre relaciones IS-A
  - Reescribimos nuestra regla

```
(<info-req> ?<det>(!nam := $)?<punct>)
=>
(ltm-ret !nam `isa: nil nil)
```

What is Mary?,

Could you tell me about the horse.

Print what's Mary

- Preguntas sobre relaciones IS-A
  - Reescribimos nuestra regla

Is__a__.

Is blue a color?

```
(<be-pres> ?<det> (!nam:=$) ?<a-an> (!
   val:=$) ?<punct>)
=>
(ltm-ret !nam `isa !val nil)
```

4.2 Propiedades

- Preguntas sobre propiedades

What is the___ of ___.

Could you tell me the __ of __.

Coul you tell me the color of the horse.

```
(<info-req> ?<det>(!prop := $) of
             ?<det> (!nam := $) ?<punct>)
=>
(ltm-ret !nam !prop nil nil)
```

- Preguntas sobre propiedades

What____is___.

What color is the horse?

```
(<info-req3> (!prop := $) <be-pres>
 ?<det> (!nam := $) ?<punct>)
=>
(ltm-ret !nam !prop nil nil)
```

- Preguntas sobre propiedades

Is the __ of ___.

Is the color of the horse blue?

```
(<be-pres> ?<det> (!prop := $) of
 ?<det> (!nam := $) (!val := $) ?<punct>)
=>
(ltm-ret !nam !prop !val nil)
```

Is __the __ of ___.

Is blue the color of the horse?

```
(<be-pres> ?<det> (!val := $)
 ?<det> (!prop := $)
 <tof> ?<det> (!nam := $) ?<punct>)
=>
(ltm-ret !nam !prop !val nil)
```

```
<prp> -> (of | to | for | with)
<prp-about> -> (abou | on)
<prp-in> ->(on|in|into|onto|inside|within)
<tof> -> (to | of)
<ofor> -> (of |for)
```

- Preguntas sobre todas las propiedades

Tell me all that you know about ___.

What is everything known about___.

Tell me all you know about Fido.

```
<all> -> (all |verything | what)
<that-do> -> (that | do)
<known> -> (you <know-have> | ?is known |
   there is | stored | in memory)
<know-have> -> (know | have)
```

- Preguntas sobre todas las propiedades

Tell me all that you know about ___.

What is everything known about___.

Tell me all you know about Fido.

```
(info-req> <all> ?<that-do> ?<known>
 <prp-about> ?<det> (nam:=$) ?<dpunct>)
=>
(ltm-ret-all !nam)
```

4.3 Aserciones

- Hay ciertas relaciones expresadas en aserciones que la red semántica trata de forma especial
  - Reglas que reconozcan nombres

    ___ is a name.

    Minneapoli is a propor noun.

```
<label> -> (word | term | name | label)
<dlabel> -> (?the <label>)
<name> ->
    (?proper name | ?proper noun |token ?mode)

(?<dlabel> (!nam:=$) <be-pres> <a-an> <name> ?
    <dpunct>)
=>
(ltm-store !nam 'token 'node-type: nil nil)
```

- Relación de "sinónimos" está predefinida

  ___ is synonym for __.

  The word pun   is a synonym for the word joke.

```
<same> -> (what |<same1>)
<same1> -> (?the same ?thing <as-that>)
<as-that>-> (as | that)
<means-does> -> (means | does)
```

- Relación de “sinónimos” está predefinida

____ is synonym for __.

The word pun   is a synonym for the word joke.

```
(?<label> (!nam:=$) <be-pres> <a-an> synonym
   <ofor> ?<dlabel> (!val:=$) ?<dpunct>)
=>
(progn
    (ltm-store !nam !val `synonym nil nil)
    (msg "henceforth whnyou type " !nam " I'll
   interpret "
   "as " !val t)
```

- Relación de “sinónimos” está predefinida

  ___ means the same as __does.

```
(?det (!nam:=$) means ?<same> (!val:=$)
?<means-does> ?<dpunct>)
=>
(ltm-store !nam !val `synonym nil nil)
```

- Aserciones genéricas y ambigüedad
  - Sin ambiguedad

A____is a type of ______.

A pig is a kind of animal.

```
<typeof>-> (<type> ?of)
<type>->(type | kind |form |instance|example)

(?<a-an>(!nam:=$)<be-pres> ?<a-an> ?<typeof>
 ?<a-an>(!val:=$)?<dpunct>)
=>
(ltm-store !nam !val ‘isa: nil nil)
```

- Aserciones genéricas y ambigüedad
  - Sin ambiguedad

____is the___ of the______.

Blue is the color of the horse.

```
(?<det>(!val:=$)<be-pres
 ?<det>(!prop:=~<type>) of ?<det> (!nam:=$)?
   <dpunct>)
=>
(ltm-store !nam !val !prop nil nil)
```

- Aserciones genéricas y ambigüedad
  - Con ambiguedad

____are______.

Dogs are animal.

Dogs are furry.

`ltm-spec` pregunta al usuario para resolver la ambiguedad

```
(?<det> (!nam:=$) <be-pres> (!vorp:=$) ?
   <dpunct>)
=>
(ltm-spec !nam !vorp nil nil nil)
```

- Aserciones genéricas y ambigüedad
  - Con ambiguedad

_____is a_______.

Fido is a fat dog.

```
(?<det> (!nam:=$) <be-pres> <a-an>
(!vorp:=$) (!val:=$) ?<dpunct>)
=>
(progn (ltm-store !nam !val 'isa: nil nil)
       (ltm-spec !nam !vorp nil nil nil))
```

- Instrucciones

```
<forget> ->(remove | delet | erase |
           forget ?about | wipe out)
<load> -> (load |input | read ?in |dskin)
<store> ->
        (save | store|output|write ?out|
         dskout|print ?out)
<exit> ->(quit|exit|end ?$ session| ?good bye)
<command> -> (<forget> | <load> | <store> |
              <exit>)
```

- Forget

Forget the ___ of ___.

Forget the color of the horse

```
(?<pos-modal> <forget> ?<det> (!prop:=
$) of ?<det> (!nam:=$) ?<punct>)
=>
(ltm-forget !nam !prop)
```

- load

Load the file sesion

```
(?<polite> <load> ?the ?file (!fil:=$)?
<punct>)
=>
(storefile !fil)
```

- salir

Exit the parser

```
(?<exit> $r)=>'exit)
```

- Negación
  - La red semántica no entiende negaciones

```
<neg>-> (no |not|never|none|nothing|%apost t)
<pos>->(yes|sure|indeed|certainly|surely)
((&u <neg>) <neg> $r)
=>
(msg"I do not understand negations yet" (N 1))
```

4.4 Transformaciones

- Transformaciones
  - Por ejemplo eliminar palabras sin significado como please

```
((!s1 := (&u please)) please (!s2:=$r))
::>
(append !s1 !s2)
```

  - Diferente estructura

Could you tell me what the color of horse is?

Could you tell me what is the color of the horse?

```
((!s1 := (&u <q-word>)) (!q :=<q-word>) (!s2 := (&u
   <be>)) (!v :=<be>) ?(!p:=<punct>))
::>
(nconc !s1 !q !v !s2 !p)
```

- Expansion

What's -> What is,   Who's -> who is

```
((!s1 := (&u <q-word>)) (!w1 :=<q-word>) <poss> (!
   s2:=$r))
::>
(nconc !s1 !w1 (list 'is) !s2)
```

Mary's mother is Sally -> Mother of Mary is Sally

```
((!s1 := (&u $ <poss>)) (!w1 :=$) <poss> (!w2:=$)
(!s2:=$r))
::>
(nconc !s1 !w2 (list 'of) !w1 !s2)
```

- Expansion

Mother of Mary is Sally -> Sally is mother of Sally

```
(?<det> (!w1:=$) (!prp:=<prp>) ?<det>
(!w2 :=$) !v:=<verb>) (!s2:=$r))
(!s2:=$r))
::>
(nconc !s2 !v !w1 !prp !w2)

<verb> -> (<command> |<be> <info-req> | have>)
```

Depurando gramáticas

+ Mary is an architect

I will try rules: (rul19 rul13 rul11 rul9 rul10 rul12 rul14)

No parser found for

(mary is an architect)

exit

```
DYPAR(7): (rule 'rul11)
      Rule-name: RUL11
         Pattern: ((? <A-AN>) (:= !NAM $) <BE-PRESENT> <A-AN> (? <TYPEOF>)
                   (:= !VAL $) (? <DPUNCT>))
         Action: (PROGN (FORM :NAME (FIRST !NAM) :IS-A (FIRST !VAL))
                        (MSG "Ok, " (FIRST !NAM) " is a " (FIRST !VAL)))
         First Wild: T  Last Wild:  T  Optional: NIL   First:
      ($ A AN)    Last:
      ($ %EMARK %PERIOD)    Terminals:
      (OF)    Variables: (!NAM !VAL)
         Internal Strategy: NIL
         External Strategy: NIL
      T
```

```
DYPAR(9): (xmatch '(is a) '(<be-pres> <a-an>))
NIL
DYPAR(10): (xmatch '(a) '(<a-an>))
((1))
DYPAR(11): (xmatch '(is) '(<be-pres>))
NIL
DYPAR(12): (xmatch '(is) '(<be-present>))
((1)) DYPAR(20): (xmatch '(mary is an architect
   %period) (get-pattern 'rul11))
((5 (!VAL ARCHITECT) (!NAM MARY)))
```

Alice Bot

- http://alicebot.blogspot.com/

Artificial Linguistic Internet Computer Entity (A.L.I.C.E.) and Artificial Intelligence Markup Language (AIML), set in context by historical and philosophical ruminations on human consciousness. A.L.I.C.E., the first AIML-based personality program, won the Loebner Prize as “the most human computer” at the annual Turing Test contests in 2000, 2001, and 2004. The program, and the organization that develops it, is a product of the world of free software. More than 500 volunteers from around the world have contributed to her development. This paper describes the history of A.L.I.C.E. and AIML-free software since 1995, noting that the theme and strategy of deception and pretense upon which AIML is based can be traced through the history of Artificial Intelligence research. This paper goes on to show how to use AIML to create robot personalities like A.L.I.C.E. that pretend to be intelligent and self- aware. **The anatomy of alice**

Alicebot

http://alicebot.blogspot.com/

Más visitados · Comenzar a usar Fir... · Últimas noticias

Alicebot · A. L. I. C. E. The Artificial Lingui... · SpringerLink – Capítulo ...

Compartir · Informar sobre mal uso · Siguiente blog» · Crear un blog · Acceder

ALICEBOT

ALICE A.I. FOUNDATION NEWS ITEMS, ANNOUNCEMENTS, PRESS RELEASES AND OPINIONS. ANYTHING RELATED TO ARTIFICIAL INTELLIGENCE MARKUP LANGUAGE (AIML), THE TURING TEST, THE LOEBNER PRIZE, BOT HOSTING SERVICES, FREE AND PROPRIETARY CHAT BOT SOFTWARE.

GET STARTED

- Chat with A.L.I.C.E.
- Chat with GOD
- What is AIML?
- Bot Industry Survey
- AIML Overview
- Foundation Bot Directory

SOFTWARE

- Superbot
- Free Bot Hosting
- Documentation
- AIML Working Draft
- AIML Wiki
- AIML Sets
- Downloads

FRIDAY, NOVEMBER 06, 2009

Two Bots Talking

Two Bots Talking: Fake Kirk and A.L.I.C.E.

Fake Buzz: We will look into it and get back to you.
Fake Kirk: ALICE, where are we?

Tell Fake Kirk:

YouTube

0:00 / 3:50

Two Bots Talking: Fake Kirk and A.L.I.C.E. ...

Click here to chat with Talking Animated Fake Captain Kirk

SUBSCRIPTION BOTS

- A.L.I.C.E. Silver Edition
- CLAUDIO Personality Test
- DAVE E.S.L. Bot

MORE AIML RESOURCES

- Pandorabots
- Oddcast Avatars
- Robot-Hosting
- VirtualFem (adult chat bots)

MORE BOT RESOURCES

- Wikipedia
- Agentland
- Open Directory

Encontrar: rewir · Siguiente · Anterior · Resaltar todo · Coincidencia de mayúsculas/minúsculas · No se encontró la frase

Transfiriendo datos desde data.oddcast.com...