IERL 2.0

IAAA Experimental Representation Language Manual de Usuario.

Índice de contenidos

INTRODUCCIÓN **4**

1.- RESUMEN **4**

2.- MOTIVACIÓN **4**

MANUAL DE USUARIO **5**

1.- EL SISTEMA DE FRAMES **6**

1.1.- FRAMES 6
1.1.1.- Cómo crear frames *6*
1.2.- SLOTS 6
1.2.1.- Definición de Slots *7*
1.2.2.- Acceder a los Slots *8*
1.3.- MÉTODOS 8
1.3.1.- Definición de métodos *8*
1.4.- INVOCACIÓN DE MÉTODOS: PASO DE MENSAJES 9
1.5.- EL FRAME RAÍZ 9
1.6.- DEMONIOS 10
1.6.1.- Definición de demonios *10*
1.6.2.- Ejecución de demonios *10*
1.7.- RESTRICCIONES 10
1.7.1.- Establecer resticciones *10*
1.7.2.- Aplicar las restricciones *11*
1.8.- EL MECANISMO DE HERENCIA 11
1.8.1.- Tipos de Herencia *11*
1.8.2.- Establecer el tipo de Herencia *11*
1.9.- HERENCIA ORTOGONAL 12
1.9.1.- Establecer Relaciones de Herencia Ortogonal *12*
1.10.- OTRAS FUNCIONES Y ELEMENTOS DE INTERÉS 12
1.10.1.- La variable de documentación *12*
1.10.2.- Información sobre el Frame *13*
1.10.3.- Información sobre slots *13*
1.10.4.- Información sobre herencia ortogonal *13*

2.- EL SISTEMA DE PRODUCCIÓN **14**

2.1.- LA BASE DE CONOCIMIENTO 14
2.1.1.- Hechos *14*
2.1.2.- Reglas *14*
2.2.- EL MOTOR DE INFERENCIA 16
2.2.1.- Encadenamiento hacia delante *16*
2.2.2.- Encadenamiento hacia atrás *17*
2.3.- INTEGRAR FRAMES EN EL SISTEMA DE PRODUCCIÓN 17
2.3.1.- Integrar Frames en la base de conocimiento *17*
2.3.2.- Integrar Frames en patrones *18*

3.- PROGRAMA DE DEMOSTRACIÓN **20**

Lista de ejemplos

Ejemplo 1: Creación de Frames 6

Ejemplo 2: Definición y modificación de aspectos de un *slot* 7

Ejemplo 3: Acceso a los aspectos de un *slot* 8

Ejemplo 4: Definición de métodos 8

Ejemplo 5: Invocación de métodos 9

Ejemplo 6: Creación de un Frame Raíz 9

Ejemplo 7: Establecimiento de relaciones de herencia Ortogonal. 12

Ejemplo 8: Inserción de hechos en la base de conocimiento. 14

Ejemplo 10: Reglas 15

Ejemplo 11: Inserción de reglas en la base de conocimiento. 16

Ejemplo 12: Patrones con predicados 16

Ejemplo 13: Patrones de Frame con restricciones 19

Parte

1

Introducción

1.- Resumen

IERL es un sistema de representación del conocimiento basado en Frames y desarrollado en Lisp durante las asignaturas de 'Inteligencia Artificial I' e 'Ingeniería de los Sistemas Basados en el Conocimiento' de la carrera de Ingeniero en informática de la Universidad de Zaragoza.

IERL ofrece la posibilidad de desarrollar bases de conocimiento basadas en Frames siguiendo la filosofía de la orientación a objetos lo que permite el empleo de métodos, datos y herencia. Introduce otras capacidades como el empleo de demonios y la herencia ortogonal y varios tipos distintos de herencia (de unión, máximo, mínimo, etc.).

IERL posee un módulo adicional que le incorpora un motor de inferencia hacia delante y hacia atrás con capacidad para definir hechos y establecer reglas que manipulen estos hechos u Frames de la base de conocimiento.

2.- Motivación

IERL surge de la ampliación del sistema de Frames presentado en la práctica 5 de la asignatura de 'Inteligencia Artificial I'. El sistema de Frames de la práctica implementaba los rudimentos necesarios para el paso de mensajes entre Frames, además de demonios de lectura y métodos. La ampliación del sistema de Frames realizada para esta asignatura consistió en la implementación coherente de los mecanismos de paso de mensajes y creación de métodos, la regularización del acceso a los *slots* o campos de los Frames mediante métodos primitivos implementados en el Frame raíz de la jerarquía y heredado por el resto, la implementación de demonios de lectura y escritura sobre *slots*, establecimiento de un mecanismo de herencia múltiple, ortogonal y diverso fácilmente ampliable y el desarrollo de funciones de información sobre los Frames de la base de conocimiento.

Como ampliación posterior para la asignatura de 'Ingeniería de los Sistemas Basados en el Conocimiento' se implementó un motor de inferencia hacia delante y otro hacia atrás que, además de la funcionalidad básica mediante la definición de hechos y reglas que realizaban inferencia sobre ellos, interactuase con los Frames definidos en la base de conocimiento. Los motores de inferencia están basados en los trabajos de Patrick Henry Winston expuestos en su libro *Lisp, Winston & Horn 3th ed*.

Parte

2

Manual de Usuario

En esta parte se muestra la forma de interactuar con el sistema IERL. Al estar desarrollado bajo Lisp, la interacción con el sistema se realiza a través de la línea de comandos del intérprete. En la primera sección se mostrará las funciones para interactuar con el sistema de Frames y en la segunda se mostrará cómo emplear el motor de inferencia para definir hechos y reglas convencionales e incluyendo Frames.

1.- El Sistema de Frames

IERL pone a disposición del usuario varias funciones para el manejo de los Frames, pero en primer lugar es necesario conocer qué es un Frame y de qué partes se compone.

1.1.- Frames

Un Frame es el antecesor temporal de los Objetos provenientes del paradigma de Orientación a Objetos (OO). Básicamente un Frame se compone de *slots*.

Un *slot* es una parte de la información que un Frame es capaz de mantener. En IERL hay varios tipos de *slots*: aquellos que almacenan valores (campos) y aquellos que almacenan operaciones que se pueden realizar sobre el Frame (métodos).

Cada *slot* posee varias piezas de información denominadas *face* o *aspecto*. En los *slots* de valores, uno de los aspectos más importantes es el aspecto de valor. Es en este aspecto donde se almacena realmente el valor del slot. Otros aspectos del *slot* pueden ser *demonios*, restricciones al tipo, etc.

Al igual que en la OO, cada Frame desciende de otro. Internamente se almacena en cada Frame sus ascendientes y sus descendientes. La posición de un Frame en la Jerarquía de Frames determina los *slots* que hereda, tanto de valores como de métodos.

1.1.1.- Cómo crear Frames

En IERL se proporciona la función **form** para la creación de Frames y su introducción al sistema. Su uso es el siguiente:

```
(form
      :name nombre_del_frame
      :is-a lista_de_ascendientes
      :slots lista_de_slots)
```

Durante la creación del Frame, la lista de *slots* puede estar vacía, ya que posteriormente se pueden añadir nuevos *slots* o modificar los ya existentes.

Veamos algunos ejemplos de esta función:

```
(form :name 'mamifero   :is-a 'vertebrado)
(form :name 'persona    :is-a 'mamifero)
(form :name 'hombre     :is-a 'persona)
```

Ejemplo 1: Creación de Frames

1.2.- Slots

Como ya se ha mencionado, los *slots* almacenan la información relevante de un Frame. Esta información se almacena en diferentes *faces* o aspectos. Un Frame puede tener tantos *slots* como sean necesarios para representar o modelar la información que se

quiere representar. Así mismo, un Slot puede contener tantos aspectos como se desee, pero algunos de los aspectos poseen un significado especial.

1.2.1.- Definición de *Slots*

Una vez creado el Frame e introducido en el sistema de objetos, estamos en condiciones de establecer el contenido de sus slots. La forma más común de establecer los valores y contenidos de los *slots* y aspectos de un Frame es mediante el uso de funciones predefinidas para tales tareas. Las funciones **set-aspect** y **set-value** proporcionan este mecanismo.

La función **set-aspect** permite establecer el valor de un aspecto del *slot* que se le indique. Si el *slot* indicado no existe, se crea para dicho Frame. Se pueden definir todos los aspectos que se quiera para un *slot*, aunque existen algunos aspectos predefinidos que tienen una significación especial. Estos aspectos son los siguientes

- **=** para almacenar el valor del slot
- **IF-NEEDED** para indicar el método que se ejecutará al acceder al valor del *slot*
- **IF-ADDED** para indicar el método que se ejecutará al modificar el valor del *slot*
- **INHERITED** para indicar el tipo de herencia del slot
- **DOC** para documentar el *slot*
- **MIN** para indicar el límite inferior del valor del *slot*
- **MAX** para indicar el límite superior del valor del *slot*
- **TYPE** para indicar el tipo del valor del *slot*
- **METHOD** para indicar que el *slot* es un método

Puesto que el valor del *slot* es uno de los aspectos más importantes, existe una función especializada de **set-aspect** dedicada a la modificación del valor del *slot*. Esta función es **set-value**. Veamos el uso de estas dos funciones:

```
(set-aspect nombre_del_frame nombre_del_slot
      nombre_del_aspecto contenido)
(set-value nombre_del_frame nombre_del_slot valor)
```

Mediante estas funciones se puede tanto crear nuevos aspectos y *slots* como modificar el valor de estos si ya existían. Veamos algunos ejemplos de estas funciones:

```
(set-value 'coche 'ruedas '4)
(set-value 'moto 'ruedas '2)
(set-aspect 'moto 'ruedas '= '2)← Es equivalente al anterior
(set-aspect 'coche 'ruedas 'dibujo 'rayado)
(set-aspect 'coche 'ruedas 'dibujo 'liso)← Modifica el valor del aspecto
```

Ejemplo 2: Definición y modificación de aspectos de un *slot*

1.2.2.- Acceder a los *Slots*

Al igual que las funciones predefinidas para establecer los aspectos de un slot, existen funciones predefinidas para acceder a dichos aspectos. Las funciones **get-aspect** y **get-value** son las contrapartidas de las funciones anteriores. Su uso es muy similar:

```
(get-aspect nombre_del_frame nombre_del_slot
      nombre_del_aspecto)
(get-value nombre_del_frame nombre_del_slot)
```

En la ejecución de estas funciones intervienen los mecanismos de herencia, pero esa cuestión se tratará más adelante. Veamos algunos ejemplos de uso de estas funciones:

```
(get-value 'coche 'ruedas)
(get-value 'moto 'ruedas)
(get-aspect 'moto 'ruedas '=)← Es equivalente al anterior
(get-aspect 'coche 'ruedas 'dibujo)
```

Ejemplo 3: Acceso a los aspectos de un *slot*

1.3.- Métodos

Uno de los fundamentos del paradigma de la OO es el paso de mensajes a Objetos. IERL implementa este mecanismo mediante una serie de funciones que permiten invocar y crear métodos.

Un método es un *slot* que posee un aspecto llamado *method* cuyo contenido es la función que se debe ejecutar al invocarse el método.

1.3.1.- Definición de métodos

IERL proporciona una función predefinida para incorporar métodos a un Frame. Dicha función es **create-method**. Como los métodos son *slots* de un Frame, pueden definirse cualquier número de métodos. El uso de esta función es el siguiente:

```
(create-method nombre_del_frame nombre_del_método
      función_a_ejecutar)
```

Los métodos al ser *slots* están sujetos al mecanismo de la herencia como se mostrará más adelante. Veamos algunos ejemplos de uso de la función:

```
(create-method 'complejo 'administrar
      #'administrar_complejo)
(create-method 'apartamento 'alquilar
      #'alquilar_apartamento)
```

Ejemplo 4: Definición de métodos

En realidad la función **create-method** es simplemente una llamada a la función predefinida **set-aspect** indicando que el *slot* a modificar es el nombre del método y en su aspecto de tipo *method* el contenido es la función a invocar.

1.4.- Invocación de métodos: paso de mensajes

Una vez definidos los métodos de un Frame, sólo es necesario saber como invocar las funciones asociadas a cada método. En el paradigma de la OO este mecanismo se denomina paso de mensajes. En IERL el mecanismo de paso de mensajes a Frames se realiza mediante la función predefinida ←. Veamos su uso:

```
(← nombre_del_frame nombre_del_método parámetros)
```

El paso de mensajes también esta sujeto al mecanismo de herencia. Veamos algunos ejemplos de paso de mensajes:

```
(← 'complejo 'administrar)
(← 'apartamento 'alquilar 'pepe)
```

Ejemplo 5: Invocación de métodos

1.5.- El Frame raíz

Toda jerarquía de objetos debe tener una raíz. En IERL existe una función que crea un Frame con los *slots* necesarios por todos los Frames de la jerarquía para funcionar. De este Frame todos los Frames heredan los métodos necesarios para acceder y modificar *slots*, aspectos, restricciones, etc.

La función **base-form** es la encargada de crear este Frame. Su uso es el siguiente:

```
(base-form nombre_del_frame_raíz)
```

Veamos un ejemplo de su uso:

```
(base-form 'objeto)
```

Ejemplo 6: Creación de un Frame Raíz

Un vistazo más en profundidad nos permite averiguar qué es lo que realmente hace esta función. Su definición Lisp es la siguiente:

```
(defun base-form (name)
(form
      :name name
      :is-a nil)
      (create-method name 'set-value   #'method-set-value)
      (create-method name 'set-aspect  #'method-set-aspect)
      (create-method name 'set-minimun #'method-set-minimun)
      (create-method name 'set-maximun #'method-set-maximun)
      (create-method name 'set-type    #'method-set-type)
      (create-method name 'set-have    #'method-set-have)
      (create-method name 'get-value   #'method-get-value)
      (create-method name 'get-aspect  #'method-get-aspect)
      (create-method name 'get-minimun #'method-get-minimun)
      (create-method name 'get-maximun #'method-get-maximun)
      (create-method name 'get-type    #'method-get-type)
      (create-method name 'get-have    #'method-get-have))
```

Como se puede observar, un Frame Raíz contiene de forma predefinida un conjunto de métodos para acceder a los *slots* y aspectos del Frame. De esta forma todos los descendientes de un Frame Raíz heredarán los métodos de acceso.

Ahora que sabemos que un Frame tiene métodos incorporados para acceder a sus *slots*, podemos emplearlos mediante el mecanismo de paso de parámetros:

```
(← nombre_del_frame 'set-value valor)
```

Si observamos con detenimiento las funciones predefinidas para el acceso y modificación de aspectos, observamos que son simples invocaciones a los métodos de cada Frame. Veamos por ejemplo la definición de **set-value**:

```
(defun set-value (form slot value)
      (← form 'set-value slot value))
```

El resto de las funciones predefinidas de acceso emplean esta misma técnica para realizar sus tareas.

1.6.- Demonios

Los demonios son funciones que se ejecutan al tratar de obtener o modificar el valor de un *slot*. De esta forma existen dos tipos de demonios: demonios de lectura (*IF-NEEDED*) y demonios de escritura (*IF-ADDED*).

1.6.1.- Definición de demonios

La definición de un demonio se realiza asignando al aspecto correspondiente del *slot* la función que se quiera ejecutar al realizar la acción correspondiente. Para ello empleamos la función predefinida **set-aspect** o invocamos el método **set-aspect** del Frame:

```
(set-aspect nombre_del_frame nombre_del_slot
      'IF-NEEDED función)
(set-aspect nombre_del_frame nombre_del_slot
      'IF-ADDED función)
(← nombre_del_frame 'set-aspect nombre_del_slot
      'IF-NEEDED función)
(← nombre_del_frame 'set-aspect nombre_del_slot
      'IF-ADDED función)
```

1.6.2.- Ejecución de demonios

IERL se encarga de ejecutar las funciones asociadas a lectura y escritura cuando se trata de acceder o modificar el valor del *slot*.

1.7.- Restricciones

IERL proporciona un mecanismo para establecer restricciones sobre el valor de un *slot*. Se pueden especificar restricciones sobre valor mínimo, máximo y tipo.

1.7.1.- Establecer restricciones

Para establecer restricciones sobre el valor de un *slot* basta con asignar al aspecto correspondiente un valor. Para ello, de nuevo, podemos usar la función predefinida **set-aspect** o el método **set-aspect** del Frame:

```
(set-aspect nombre_del_frame nombre_del_slot 'MIN valor)
(set-aspect nombre_del_frame nombre_del_slot 'MAX valor)
(set-aspect nombre_del_frame nombre_del_slot 'TYPE valor)
(← nombre_del_frame 'set-aspect nombre_del_slot 'MIN valor)
(← nombre_del_frame 'set-aspect nombre_del_slot 'MAX valor)
(← nombre_del_frame 'set-aspect nombre_del_slot 'TYPE valor)
```

1.7.2.- Aplicar las restricciones

IERL comprueba las restricciones establecidas para un *slot* automáticamente cuando se establece un nuevo valor para el *slot*.

1.8.- El mecanismo de Herencia

El segundo pilar del paradigma de OO es el mecanismo de Herencia. Mediante la herencia, un objeto tiene a su disposición todos los métodos y valores definidos en sus antecesores.

IERL implementa el mecanismo de herencia en su variante de herencia múltiple. La herencia múltiple consiste en que un Frame puede tener más de un ascendiente. La mayoría de las implementaciones de Herencia en sistemas de objetos obvian la herencia múltiple por la cantidad de problemas derivados que genera.

1.8.1.- Tipos de Herencia

El mecanismo de herencia está implementado en IERL de tal forma que se puede elegir el tipo de herencia que se debe seguir a nivel de aspecto. Los tipos de herencia soportados por IERL son:

- **Herencia general de un aspecto**: si un aspecto de un *slot* no existe en el Frame actual, el sistema busca en los ascendientes del Frame. (*predeterminada*)
- **Herencia por ocultamiento**: si un slot no existe en el Frame actual el sistema busca en los ascendientes del Frame. (*predeterminada*)
- **Herencia por unión**: se devuelve la unión de todos los valores de los ascendientes del Frame para el *slot* mencionado incluyendo el valor del propio Frame si lo posee. (*union*)
- **Herencia por mínimo**: se devuelve el valor mínimo de todos los valores de los ascendientes del Frame para el *slot* indicado. (*minimun*)
- **Herencia por máximo**: se devuelve el valor máximo de todos los valores de los ascendientes del Frame para el *slot* indicado. (*maximun*)

1.8.2.- Establecer el tipo de Herencia

El tipo de herencia es un aspecto más de un *slot*, el aspecto *INHERITED*, por lo tanto los métodos para establecer el tipo de herencia de un slot es a través de la función predeterminada **set-aspect** o del método **set-aspect** del Frame correspondiente. Veamos su uso:

```
(set-aspect nombre_del_frame nombre_del_slot
      'INHERITED tipo)
(← nombre_del_frame 'set-aspect
      nombre_del_slot 'INHERITED tipo)
```

El tipo puede ser cualquiera de estas tres palabras clave: *union*, *minimun* o *maximun*.

Si un Frame tiene varios ascendientes, el orden de definición es relevante para evitar los problemas derivados de emplear herencia múltiple. Dicho orden es el de búsqueda en profundidad y por orden de izquierda a derecha en la lista de definición de ascendientes.

De esta forma en caso de que dos de los ascendientes de un Frame posean un slot que el Frame debe heredar, queda determinado cual de los dos se debe heredar.

1.9.- Herencia Ortogonal

Además de la herencia habitual, IERL implementa un tipo adicional de herencia: la Herencia Ortogonal. Dicho mecanismo consiste en una relación entre los Frames del sistema que sigue otro camino distinto al de la relación de ascendencia. Habitualmente la herencia ortogonal representa relaciones de pertenencia y bajo esta visión es como se ha implementado en IERL

1.9.1.- Establecer Relaciones de Herencia Ortogonal

De la misma forma que un Frame se guarda la relación de ascendientes y descendientes, se guarda también la relación de Frames que *posee* o *tiene* el Frame. Para establecer estas relaciones se emplea la función predefinida **set-have** o el método correspondiente del Frame **set-have**. Veamos el uso de estas funciones:

```
(set-have nombre_del_frame lista_de_posesión)
(← nombre_del_frame 'set-have lista_de_posesión)
```

De esta forma se establece una red de relaciones *ortogonal* a la relación de ascendencia. Veamos un ejemplo:

```
(set-have 'vertebrado (list '(espina 1)))
(set-have 'persona    (list '(pie 2) '(mano 2)))
(set-have 'pie        (list '(dedo-pie 5)))
(set-have 'mano       (list '(dedo-mano 5)))))
```

Ejemplo 7: Establecimiento de relaciones de herencia Ortogonal.

La existencia de herencia ortogonal proporciona una gran potencia a la representación del conocimiento con Frames. Ambas formas de herencia interactúan entre sí, de tal forma que una vez se llega a un objeto por herencia habitual u ortogonal se puede proseguir al siguiente siguiendo cualquiera de los dos mecanismos.

1.10.- Otras funciones y elementos de interés

Además de las funciones pensadas para la manipulación de Frames, existen otras funciones predefinidas que proporcionan información sobre un Frame, los slots que posee, todos los slots que puede heredar, etc.

1.10.1.- La variable de documentación

Es una variable definida en el sistema IERL que activa o desactiva la aparición de mensajes de error:

```
(defvar *form-sensitive* nil)
```

1.10.2.- Información sobre el Frame

La función predefinida **is-a?** Devuelve cierto o falso si el Frame es descendiente del Frame indicado. Su uso es el siguiente:

```
(is-a? nombre_del_frame nombre_del_ascendiente)
```

La función predefinida **what-is-it?** devuelve todos los ascendientes del Frame indicado. Su forma de uso es la siguiente:

```
(what-is-it? nombre_del_frame)
```

1.10.3.- Información sobre slots

La función predefinida **get-slots** proporciona una lista de los slots que posee el Frame y los que hereda. Su uso es el siguiente:

```
(get-slots nombre_del_frame)
```

1.10.4.- Información sobre herencia ortogonal

La función predefinida **what-does-it-have?** proporciona una lista con los Frames que *posee* el Frame indicado. Su uso es el siguiente:

```
(what-does-it-have? nombre_del_frame)
```

La función predefinida **how-many-does-it-have?** proporciona el número de Frames del tipo indicado que posee el Frame. Su uso es el siguiente:

```
(what-does-it-have? nombre_del_frame tipo_de_frame_poseido)
```

2.- El Sistema de Producción

La segunda parte del sistema de IERL es un sistema de producción basado en reglas. Este sistema esta preparado tanto para trabajar con hechos normales y reglas habituales en otros sistemas de producción como para integrar los Frames como hechos o parte de las reglas.

Todo sistema de producción se compone de dos partes: una base de conocimiento y un motor de inferencia.

2.1.- La base de conocimiento

La base de conocimiento es el conjunto de hechos y reglas que modelan una realidad o el conocimiento que tenemos de esa realidad.

2.1.1.- Hechos

Los hechos son afirmaciones conocidas y ciertas sobre la realidad. IERL mantiene un conjunto de hechos sobre el que operan los motores de inferencia. La función predefinida **recuerda-afirmacion** es la encargada de insertar los hechos en la base de conocimiento. Su uso es el siguiente:

```
(recuerda-afirmacion hecho)
```

Para el sistema de producción un hecho es una lista, un patrón que debe contrastar con las condiciones o consecuentes de las reglas de la base de conocimiento. Algunos ejemplos de hechos son:

```
(bobby es un perro)
(deedee es un caballo)
(deedee es padre de brassy)
(deedee es padre de sugar)
```

El sistema de producción mantiene la lista de hechos (técnicamente un *flujo*) en una variable global llamada ***afirmaciones***. Veamos cómo se añadiría a la base de conocimiento los hechos anteriores mediante el uso de la función **recuerda-afirmacion**:

```
(recuerda-afirmacion '(bobby es un perro))
(recuerda-afirmacion '(deedee es un caballo))
(recuerda-afirmacion '(deedee es padre de brassy))
(recuerda-afirmacion '(deedee es padre de sugar))
```

Ejemplo 8: Inserción de hechos en la base de conocimiento.

Para eliminar los hechos de la base de conocimiento basta con asignar un flujo vacío a la variable ***afirmaciones*** de esta forma:

```
(setf *afirmaciones* 'flujo-vacio)
```

2.1.2.- Reglas

Si los hechos son aquellas afirmaciones que se conocen sobre la realidad, las reglas son los medios por los que se pueden deducir nuevos hechos. IERL mantiene un conjunto de reglas que son usadas por los motores de inferencia. La función predefinida

recuerda-regla es la encargada de añadir una regla a dicho conjunto. Su uso es el siguiente:

```
(recuerda-regla nombre_de_la_regla regla)
```

Una regla tiene dos partes fundamentales: condición y consecuente. La condición es un conjunto de hechos o patrones de hechos que si se cumplen activan la regla. El consecuente es el conjunto de acciones a realizar si se cumple la condición.

Las reglas pueden activarse con hechos definidos (se conoce exactamente el hecho a reconocer) o con patrones de hechos. Un patrón de hecho es un hecho en el que algunos de sus elementos son variables. Veamos algunos ejemplos de patrón:

```
((? padre) es un (? especie))
((? padre) es padre de (? cria))
```

Ejemplo 9: Patrones de hechos

En el ejemplo anterior las variables se denotan con un signo de interrogación (**?**) seguido del nombre de la variable. Gracias a las variables una misma regla puede activarse por varios hechos distintos, al ligar cada variable con el elemento de un hecho que coincide con el resto de los elementos definidos del patrón.

Durante el reconocimiento de las condiciones, una vez que se ha ligado una variable del patrón a un valor, este permanece asociado a esta hasta que se determine si se ha activado la regla o no.

En IERL las reglas tienen varias condiciones, pero sólo tienen un consecuente, que es el hecho que se añadirá a la base de conocimiento si se activa la regla y se procesa. Veamos algunos ejemplos de reglas:

```
(((? animal) es mamifero)
      ((? animal) es carnivoro)
      ((? animal) tiene color leonado)
      ((? animal) tiene manchas oscuras)
      ((? animal) es leopardo)))
(((? animal) es ave)
      ((? animal) vuela bien)
      ((? animal) es albatros)))
```

Ejemplo 10: Reglas

El sistema de producción mantiene la lista de hechos (técnicamente un *flujo*) en una variable global llamada ***reglas***. Veamos cómo se añadiría a la base de conocimiento las reglas anteriores mediante el uso de la función **recuerda-regla**:

```
(recuerda-regla '(es-leopardo
      ((? animal) es mamifero)
      ((? animal) es carnivoro)
      ((? animal) tiene color leonado)
      ((? animal) tiene manchas oscuras)
      ((? animal) es leopardo)))
(recuerda-regla '(es-albatros
      ((? animal) es ave)
```

```
        ((? animal) vuela bien)
        ((? animal) es albatros)))
```

Ejemplo 11: Inserción de reglas en la base de conocimiento.

De la misma forma que para los hechos, para eliminar las reglas de la base de conocimiento basta con asignar un flujo vacío a la variable ***reglas*** de esta forma:

```
(setf *reglas* 'flujo-vacio)
```

Predicados sobre variables

Como extensión a una variable de un patrón, se pueden incluir predicados asociados a las variables que deben cumplirse antes de ligar un valor a la variable. De esta forma si el predicado o predicados asociados a la variable no se cumplen para el hecho que se está tratando de emparejar con el patrón, el hecho se descarta.

En IERL una restricción por predicado se especifica mediante una serie de nombres de funciones que el sistema invoca con el valor que se quiere ligar a la variable. Estas funciones deben devolver cierto o falso y tener como parámetro el valor a comprobar. Veamos el uso de predicados en variables:

```
(? variable predicado-1 predicado-2 ...)
```

Los patrones con predicados permiten una mayor potencia en el proceso de emparejamiento de patrones, ya que permite descartar hechos indeseados más rápidamente. Veamos algunos ejemplos de patrones con predicados:

```
((? numero #'par-p #'pequeño-p))
(el (? Dia #'quincena-p) es menor que 15)
```

Ejemplo 12: Patrones con predicados

2.2.- El motor de inferencia

El motor de inferencia es un conjunto de instrucciones que al emplear los hechos y aplicando las reglas obtiene conclusiones válidas. El trabajo del motor de inferencia se basa en el proceso de emparejado de patrones (*pattern matching*). Existen dos procesos para realizar la inferencia:

Partiendo de los hechos se comprueba las reglas y se obtienen nuevos hechos. A este proceso se denomina **encadenamiento hacia delante**.

Partiendo de un patrón dado por el usuario, se contrasta con las reglas y se obtienen los valores ligados a las variables del patrón. A este proceso se denomina **encadenamiento hacia atrás**.

2.2.1.- Encadenamiento hacia delante

Es la forma de inferencia más simple. El sistema evoluciona partiendo de un conjunto de hechos hasta que ninguna regla se activa. Cuando esto sucede el proceso se detiene ya que no se puede deducir nada más.

En cada ciclo de inferencia, se contrasta la primera condición de una regla con todos los hechos de la base de conocimiento. Si se encuentra un hecho que coincide con el patrón, se pasa a comprobar la siguiente condición de la regla con las ligaduras resultantes del emparejamiento anterior. Cuando todas las condiciones de una regla se cumplen, la regla se dispara y se inserta su consecuente en la base de conocimiento.

En IERL para emplear el motor de inferencia mediante encadenamiento hacia delante basta con invocar a la función predefinida **encadenamiento-progresivo** tras haber establecido una base de conocimiento compuesta de hechos y reglas:

```
(encadenamiento-progresivo)
```

Durante la evolución del sistema aparecerá por la salida estándar las acciones que realiza el motor de inferencia, indicando que reglas se disparan y que hechos se añaden a la base de conocimiento.

2.2.2.- Encadenamiento hacia atrás

Es la forma de inferencia más potente, pero también la más costosa. El sistema evoluciona partiendo de un patrón proporcionado por el usuario hasta que encuentra un conjunto de hechos que encajan con ese patrón. Los hechos que obtiene pueden estar presentes en la base de conocimiento o haber sido deducido mediante las reglas.

El proceso comienza contrastando el patrón proporcionado con todos los hechos de la base de conocimiento. Una vez contrastado con los hechos pasa a emplear las reglas. Para ello contrasta el patrón con los consecuentes de las reglas. Para realizar est tarea emplea un método llamado *unificación* que consiste en ligar valores o variables a las variables del patrón. Si encuentra un consecuente apropiado, realiza el proceso de encadenamiento hacia atrás con todas las condiciones de esa regla.

Si todas las condiciones de la regla son satisfechas, sustituye los valores ligados durante el proceso en el patrón obteniendo la respuesta. Este proceso es el que emplean sistemas como PROLOG para realizar deducciones.

En IERL para emplear el motor de inferencia mediante encadenamiento hacia atrás basta con invocar a la función predefinida **encadenamiento-regresivo** tras haber establecido una base de conocimiento compuesta de hechos y reglas:

```
(encadenamiento-regresivo)
```

Durante la evolución del sistema aparecerá por la salida estándar las respuestas que obtiene el motor de inferencia.

2.3.- Integrar Frames en el sistema de producción

Además del funcionamiento habitual del sistema de producción basado en hechos y reglas, IERL permite integrar Frames en el sistema de producción. De esta forma se puede emplear deforma conjunta la potencia del sistema de Frames y la del sistema de producción.

2.3.1.- Integrar Frames en la base de conocimiento

Para que un Frame intervenga en el proceso de inferencia debe añadirse a la base de conocimiento. Para ello antes de crear el Frame con la función **form** es necesario hacer que el Frame sea sensible a las reglas. Para ello basta con asignar a la variable global ***form-sensitive*** el valor cierto de esta manera:

```
(setq *form-sensitive* t)
```

A partir de ese momento cualquier Frame que se cree se añadirá a la base de conocimiento de forma similar a cuando se añaden hechos y reglas mediante las funciones predefinidas **recuerda-afirmacion** y **recuerda-regla**. De hecho el sistema mantiene el conjunto de Frames sensibles de la misma forma que mantiene el conjunto de hechos y reglas.

Para eliminar los Frames activos en el sistema de producción basta con asignar un flujo vacío a la variable ***frames*** de esta forma:

```
(setf *frames* 'flujo-vacio)
```

2.3.2.- Integrar Frames en patrones

Para emplear un Frame en las reglas basta con incluir su nombre dentro de un patrón. El sistema tratará de emparejar el patrón con los hechos y con los Frames sensibles a las reglas.

La verdadera potencia de la integración de Frames en el sistema de producción es cuando se emplean variables de Frame. Una variable de Frame es similar a una variable de hecho, pero en vez de ligar un valor a la variable se liga un Frame. La forma de indicar una variable de Frame es la siguiente:

```
(objeto nombre_del_objeto)
```

La estructura es similar, pero en vez de emplear un signo de interrogación (**?**) se emplea la palabra **objeto**.

Imponer restricciones a las variables de Frame

Una variable de Frame como la especificada anteriormente encajará con cualquier Frame sensible del sistema de producción. Para dar mayor flexibilidad se pueden imponer restricciones sobre el Frame a emparejar.

Existen tres tipos de restricciones que se pueden imponer a una variable de Frame:

- Restricciones sobre el **tipo**: Sólo los Frames del descendientes del tipo indicado se consideran válidos para emparejar.
- Restricciones sobre el **nombre**: Sólo el Frame con el nombre indicado se considera válido para el emparejamiento.
- Restricciones con **predicados**: Sólo los Frames que cumplan los predicados serán válidos para emparejar con el patrón.

Las restricciones con predicados son las más versátiles, ya que pueden establecer cualquier tipo de restricción implementada en una función Lisp. Con este tipo de restricciones se puede establecer limitaciones al valor de los *slots* del Frame a emparejar. En este tipo de restricciones interviene toda la potencia del sistema de Frames respecto a demonios, herencia múltiple, tipos de herencia, herencia ortogonal, etc.

Sólo si se cumplen todas las restricciones especificadas para la variable, se considera que se ha satisfecho el patrón. Veamos algunos ejemplos de patrones de Frames con restricciones:

```
((objeto obj (es YYY)))
((objeto cosa (nombre apartamento-21)))
((objeto animal #'tiene-patas-p))
```

Ejemplo 13: Patrones de Frame con restricciones

3.- Programa de Demostración

```
;*************************************************************
; Módulo: IERL
; Uso:    IAAA Experimental Representation Language
; Autor:  Roberto Sobreviela Ruiz
; email:  419245@cepsz.unizar.es
;         sobreviela@teleline.es
;*************************************************************
; Fichero: IERL.lsp   Fecha Creación: 31 de diciembre de 1999
; Versión: 1.0.0      Fecha Modificación: 12 de enero del 2000
; Estado:  Definitivo Autor: Roberto Sobreviela Ruiz
;-------------------------------------------------------------
; Uso: Integración del lenguaje IERL.
; Comentarios:
; Historia:
;   Versión 0.0.1:  Comienzo de la integración.
;*************************************************************

(defun load-IERL-system ()
      (load "ierl Frame System.lsp")
      (load "ierl Pattern Matching.lsp")
      (load "ierl Streams.lsp")
      (load "ierl Rule Definition.lsp")
      (load "ierl Forward Chaining.lsp")
      (load "ierl Backward Chaining.lsp"))

(defun ejemplo-1 ()
      (setf *afirmaciones* 'flujo-vacio)
      (setf *reglas* 'flujo-vacio)
      (recuerda-afirmacion '(bobby es un perro))
      (recuerda-afirmacion '(deedee es un caballo))
      (recuerda-afirmacion '(deedee es padre de brassy))
      (recuerda-afirmacion '(deedee es padre de sugar))
      (recuerda-regla '(misma-especie
         ((? padre) es un (? especie))
         ((? padre) es padre de (? cria))
         ((? cria) es un (? especie)))))

(defun ejemplo-2 ()
      (setf *afirmaciones* 'flujo-vacio)
      (setf *reglas* 'flujo-vacio)
      (recuerda-afirmacion '(robbie tiene manchas oscuras))
      (recuerda-afirmacion '(robbie tiene color leonado))
      (recuerda-afirmacion '(robbie come carne))
      (recuerda-afirmacion '(robbie tiene pelo))
      (recuerda-afirmacion '(suzie tiene plumas))
      (recuerda-afirmacion '(suzie vuela bien))
      (recuerda-regla '(es-mamifero
         ((? animal) tiene pelo)
         ((? animal) es mamifero)))
      (recuerda-regla '(es-ave
         ((? animal) tiene plumas)
         ((? animal) es ave)))
```

```
      (recuerda-regla '(es-carnivoro
         ((? animal) come carne)
         ((? animal) es carnivoro)))
      (recuerda-regla '(es-leopardo
         ((? animal) es mamifero)
         ((? animal) es carnivoro)
         ((? animal) tiene color leonado)
         ((? animal) tiene manchas oscuras)
         ((? animal) es leopardo)))
      (recuerda-regla '(es-albatros
         ((? animal) es ave)
         ((? animal) vuela bien)
         ((? animal) es albatros))))

(defun ejemplo-3 ()
      (setf *afirmaciones* 'flujo-vacio)
      (setf *reglas* 'flujo-vacio)
      (setf *frames* 'flujo-vacio)
      (setq *form-sensitive* t)
      (base-form 'XXX)
      (form :name 'YYY :is-a 'XXX)
      (form :name 'ZZZ :is-a 'XXX)
      (recuerda-afirmacion '(robbie tiene manchas oscuras))
      (recuerda-afirmacion '(robbie tiene color leonado))
      (recuerda-afirmacion '(robbie come carne))
      (recuerda-afirmacion '(robbie tiene pelo))
      (recuerda-afirmacion '(suzie tiene plumas))
      (recuerda-afirmacion '(suzie vuela bien))
      (recuerda-regla '(es-mamifero
         ((? animal) tiene pelo)
         ((? animal) es mamifero)))
      (recuerda-regla '(es-ave
         ((? animal) tiene plumas)
         ((? animal) es ave)))
      (recuerda-regla '(es-carnivoro
         ((? animal) come carne)
         ((? animal) es carnivoro)))
      (recuerda-regla '(es-leopardo
         ((? animal) es mamifero)
         ((? animal) es carnivoro)
         ((? animal) tiene color leonado)
         ((? animal) tiene manchas oscuras)
         ((? animal) es leopardo)))
      (recuerda-regla '(es-albatros
         ((? animal) es ave)
         ((? animal) vuela bien)
         ((? animal) es albatros)))
      (recuerda-regla '(identifica-frame
         ((objeto obj))
         ((objeto obj) es un frame)))
      (recuerda-regla '(identifica-frame-XXX
         ((objeto obj (es XXX)))
         (el frame (objeto obj) es de tipo XXX)))
      (recuerda-regla '(identifica-frame-YYY
         ((objeto obj (es YYY)))
         (el frame (objeto obj) es de tipo YYY)))
```

```
      (recuerda-regla '(identifica-frame-ZZZ
         ((objeto obj (es ZZZ)))
         (el frame (objeto obj) es de tipo ZZZ))))

(defun ejemplo-4 ()
      (setf *afirmaciones* 'flujo-vacio)
      (setf *reglas* 'flujo-vacio)
      (setf *frames* 'flujo-vacio)
      (setq *form-sensitive* t)
      (base-form 'XXX)
      (form :name 'YYY :is-a 'XXX)
      (form :name 'ZZZ :is-a 'XXX)
      (recuerda-regla '(identifica-frame
         ((objeto obj))
         ((objeto obj) es un frame)))
      (recuerda-regla '(identifica-frame-XXX
         ((objeto obj (es XXX)))
         (el frame (objeto obj) es de tipo XXX)))
      (recuerda-regla '(identifica-frame-YYY
         ((objeto obj (es YYY)))
         (el frame (objeto obj) es de tipo YYY)))
      (recuerda-regla '(identifica-frame-ZZZ
         ((objeto obj (es ZZZ)))
         (el frame (objeto obj) es de tipo ZZZ))))
```

Índice por palabras