BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

"IMPLEMENTACION DE PREFERENCIAS BASADA EN LA SEMANTICA DE PROGRAMACION LOGICA P-ESTABLE"

TESIS PROFESIONAL PRESENTADA POR

CESAR HUITZIL TELLO

COMO REQUISITO PARA OBTENER EL TÍTULO DE LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

"IMPLEMENTACION DE PREFERENCIAS BASADA EN LA SEMANTICA DE PROGRAMACION LOGICA P-ESTABLE"

Tesis profesional presentada por

Cesar Huitzil Tello

Como requisito para obtener el título de
Licenciado en Ciencias de la Computación

Asesores

Dra. Claudia Zepeda Cortés
Dr. José Luis Carballido

Índice

INTRODUCCIÓN 1

CAPÍTULO 1. MARCO TEÓRICO 5

1.1 Preferencias 5

1.2 Answer Set Programming (A-Prolog) 6
1.2.1 Semántica Estable 7
1.2.2 Sistemas ASP 8
1.2.2.1 Smodels 8
1.2.2.2 Clasp 8

1.3 Enfoque considerado para preferencias 9

CAPÍTULO 2. ANÁLISIS 15

2.1 Estructuras usadas para almacenar los datos 16
2.1.1 Almacenamiento del programa 16
2.1.2 Almacenamiento de las reglas de preferencia 16
2.1.3 Almacenamiento de los modelos (Answer Sets) 18
2.1.4 Almacenamiento de los grados de satisfacción 19
2.1.5 Almacenamiento del conjunto S 20

CAPÍTULO 3. DISEÑO 22

3.1 Identificación de reglas de preferencia y reglas que no son reglas de preferencia 22

3.2 Identificación y almacenamiento de la cabeza y el cuerpo de las reglas de preferencia 22
3.2.1 Obtención y almacenamiento de la cabeza 23
3.2.2 Obtención y almacenamiento del cuerpo 23

3.3 Obtención y almacenamiento de modelos 24
3.3.1 Obtención y almacenamiento de modelos con Smodels 24
3.3.2 Obtención y almacenamiento de modelos con Clasp 25

3.4 Obtención y almacenamiento de grados 26

3.5 Obtención y almacenamiento del conjunto S 27

3.6 Diseño de la interfaz del programa 28

CAPÍTULO 4. IMPLEMENTACIÓN 30

4.1 Método para buscar los modelos **31**

4.2 Método que ejecuta un comando y guarda la salida en modelos.txt **32**

4.3 Método para obtener los grados de satisfacción **34**

4.3.1 Código que ejecuta la regla 1 para obtener los grados de satisfacción 36

4.3.2 Código que ejecuta la regla 2 para obtener los grados de satisfacción 37

4.3.3 Código que ejecuta la regla 3 para obtener los grados de satisfacción 37

4.4 Método para obtener el conjunto S **38**

CAPÍTULO 5. RESULTADOS **41**

5.1 Ejemplo 1 .... **41**

5.2 Ejemplo 2 .... **43**

5.3 Ejemplo 3 .... **44**

CONCLUSIONES **47**

REFERENCIAS **48**

Introducción

Las elecciones que tomamos se guían por nuestras preferencias. Por lo tanto, una comprensión de los diversos aspectos del manejo de preferencia debe ser de gran importancia para cualquier persona que intente construir sistemas que interactúan con los usuarios o, simplemente, apoyar sus decisiones. Por ejemplo, esto podría ser un sitio de compras que intenta ayudarnos identificar el elemento más preferido, puede ser una búsqueda de información y el motor de recuperación de la información nos intenta proporcionar información más preferida, o agentes integrados más sofisticados, como robots, asistentes personales, etc. [3]

Cada una de estas aplicaciones se esfuerza por tomar medidas que lleven a un mayor estado preferido para nosotros: un mejor producto, los artículos más apropiados, el mejor comportamiento, etc. Los primeros trabajos de Inteligencia Artificial estaban centrados en alcanzar una meta, este paradigma aun sigue siendo dominante en la resolución de problemas de Inteligencia Artificial [3].

Un ejemplo es la recuperación de información de grandes Bases de Datos como es la Web, ya que el usuario no sabe cuál es el mejor documento disponible, por lo general va a terminar pidiendo algo que es inalcanzable o pidiendo muy poco. El usuario puede ir ajustando su búsqueda conforme a lo que vaya encontrando pero este modo no es muy atractivo ya que el espacio de soluciones de aplicaciones de este tipo puede se enorme o incluso infinita.

Las preferencias se pueden utilizar para comparar las soluciones factibles de un problema dado, en orden para establecer si hay un orden entre estas soluciones o establecer si tales soluciones son equivalentes con respecto a algunos requisitos. En la actualidad hay algunos enfoques generales en el razonamiento no monótono dado con preferencias [1]. Aquí se presentan las reglas de preferencia que permiten especificar las preferencias como un orden entre las posibles soluciones de un problema y la forma de darle solución. Estas normas se utilizan un nuevo conectivo,*, llamado operador de preferencia. El formalismo utilizado para desarrollar nuestro trabajo es Answer Set Programming (ASP) [2]. ASP es

una representación del conocimiento declarativo y lenguaje de programación lógica. ASP representa un nuevo paradigma para la lógica de programación que nos permite, utilizando el concepto de la negación como fracaso, manejar los problemas con el conocimiento y producir por defecto razonamiento no monotónico. Dos implementaciones de software más populares para calcular ASP son DLV[1] y SMODELS[2].

La eficacia de dichos programas autorizados para aumentar la lista de aplicaciones prácticas en las áreas de planificación, agentes lógicos e inteligencia artificial. La mayoría de las investigaciones en ASP y en particular acerca de las preferencias en ASP supone reglas sintácticamente simples. Esto se justifica ya que, la mayoría de las veces, esas sintaxis restringidas son suficientes para representar un amplia clase de problemas interesantes y relevantes. Podría parecer innecesario generalizar la noción de conjuntos de respuesta a algunas fórmulas más complicadas. Sin embargo, una sintaxis más amplia de las normas podría traer algunos beneficios. Por ejemplo, el uso de expresiones anidadas podría simplificar la tarea de escribir programas lógicos y mejorar su legibilidad. Por lo tanto, en este trabajo se presenta un enfoque sobre las preferencias a las teorías generales.

Un punto principal y tal vez la mayor debilidad de la lógica proposicional es su carácter monotónico, la cual, es aquella regla que nos dice que si agregamos un nuevo conocimiento o una nueva sentencia, dicha sentencia solo refuerza pero no modifica ni disminuye el conjunto de sentencias que componen nuestro modelo.

El problema con esto, es que este tipo de pensamiento no es práctico para representar el conocimiento de una inteligencia artificial que interactué y aprenda en el mundo real. Sabemos que en el mundo real, las cosas cambian con increíble rapidez, lo que el día de hoy sabemos y damos por sentado, el día de mañana resulta que es incierto, ayer (bueno hace algunos años) aprendimos que el sistema solar tenía nueve planetas, ahora tenemos que aprender que solo son ocho.

---

[1] http://www.dbai.tuwien.ac.at/proj/dlv
[2] http://www.tcs.hut.fi/Software/smodels

De tal forma que el conocimiento es no monotónico, es decir, cada nuevo conocimiento que adquirimos viene a modificar e incluso invalidar los conocimientos previos que teníamos almacenados [17].

El proyecto consiste en analizar e implementar el enfoque para representar preferencias. El enfoque a implementar es un enfoque basado en semántica de programación lógica por lo que la representación de la información corresponde a programas lógicos. En particular el enfoque de preferencias a implementar se basa en la semántica de programación lógica estable.

El objetivo de este trabajo es poder elegir la mejor opción que más convenga teniendo en cuenta las condiciones dadas. Este trabajo se va a implementar con la elaboración de un software en lenguaje Java con NetBeans IDE 7.0.1 y para obtener la semántica estable se utilizara Smodels y Clasp.

En el capítulo 1 se encuentra el marco teórico que se refiere al fundamento teórico con el que se realizo este trabajo, lo que son las preferencias basadas en Answer Set Programming, algunos sistemas solucionadores de Answer Set como son Smodels y Clasp. También se explica el enfoque en el que está basado este trabajo, así como la forma de expresar las reglas que se van a evaluar.

En el capítulo 2 se ve el análisis utilizado para poder representar las reglas de preferencia, el cual consiste en modelar y resolver el problema de preferencias. También se establecen las estructuras utilizadas para almacenar los datos y la forma en que se utilizan estas estructuras.

En el capítulo 3 se presenta el diseño tanto de la interfaz como de los algoritmos que procesan las preferencias y nos dan una respuesta a nuestro problema.

En el capítulo 4 se presenta el desarrollo e implementación del programa teniendo como referencia un ejemplo que contiene algunas reglas de preferencia para ir viendo la

evolución del software. También se presentan algunos fragmentos del código utilizado para realizar las tareas más importantes.

En el capítulo 5 se darán a conocer los resultados que nos arroja este programa al procesar las reglas que están en nuestro archivo de texto. Se agregan también las capturas correspondientes para ir visualizando los resultados que se obtienen en cada fase de la ejecución del programa.

Finalmente se da una conclusión acerca de los objetivos de este trabajo.

Capítulo 1. Marco Teórico

Las preferencias se pueden utilizar para comparar las soluciones factibles de un problema dado, para establecer si hay un orden entre estas soluciones o establecer si tales soluciones son equivalentes con respecto a algunos requisitos. En este capítulo se presenta brevemente lo que son las preferencias basadas en Answer Set Programming también conocida como Programación Lógica Estable, lo que es Semántica Estable ya que Answer Set Programming está basado en Semántica Estable y algunos sistemas solucionadores de Answer Set como son Smodels y Clasp y finalmente el enfoque que se está considerado para encontrar estas preferencias.

1.1 Preferencias

¿Qué se quiere decir con "preferencias"? Sobre algún conjunto de posibles opciones, una opción más conveniente precede a la menos deseable [3]. Por desgracia, mientras que la semántica de las relaciones de preferencia es bastante clara, trabajar con preferencias puede ser bastante difícil [3]. La cuestión más obvia es la dificultad cognitiva de especificar una relación de preferencia [3]. El esfuerzo cognitivo no es un problema cuando solo nos importa un único atributo. Por ejemplo, cuando una tienda vende un determinado modelo de teléfono, lo único que le podría preocupar seria el precio del teléfono. Cuando uno conduce al trabajo, todo lo que le preocuparía acerca de la elección de la ruta puede ser el tiempo de conducción [3].

Sin embargo, por lo general nuestras preferencias son más complicadas, e incluso en estos dos ejemplos a menudo nos preocupamos por distintos atributos: en el teléfono nos interesaría la garantía, el tiempo de envío, compañía, modelo, etc. Cuando conducimos nos preocupa el escenario, la suavidad de conducción, etc. Incluso ordenar estos dos resultados puede ser cognitivamente difícil debido a la necesidad de considerar las ventajas y desventajas, entre dependencias de los distintos atributos [3].

1.2 Answer Set Programming (A-Prolog)

Answer Set Programming (ASP, también conocida como Programación Lógica Estable o A-Prolog) es un área de representación del conocimiento enfocado en la lógica, basado en lenguajes para el modelado de problemas de cálculo en términos de restricciones [4, 9]. Los orígenes de Answer Set Programming están en la programación lógica [15, 16] y el razonamiento no monótono [12, 14]. ASP tiene sus inicios cuando estas dos áreas se fusionaron en el trabajo de Gelfond y Lifschitz [13] donde se define la semántica estable.

Un problema es modelado por un programa lógico de manera que el Answer Set del programa corresponde directamente a la solución del problema [9, 11].

Donde un programa lógico es entendido como una teoría proposicional. Debemos usar el lenguaje de lógica proposicional de forma usual, usando símbolos proposicionales: p, q,…, conectivos proposicionales $\wedge$, $\vee$, $\rightarrow$, $\bot$ y símbolos auxiliares (,). La formula proposicional bien formada $f \leftarrow g$ es otra forma de escribir $g \rightarrow f$. Se asume que para cualquier fórmula proposicional bien formada f, $\neg f$ es solo una abreviación de $f \rightarrow \bot$ y $\top$ es una abreviación de $\bot \rightarrow \bot$. Se señala que $\neg$ es la única negación usada en este trabajo. Un *átomo* es un símbolo proposicional. Una *literal* es ya sea un átomo a (una literal positiva) o la negación de un átomo $\neg a$ (una literal negativa). En particular $f \rightarrow \bot$ es llamado *restricción* y es también denotado como $\leftarrow f$. Dado un conjunto de formulas bien formadas F, definimos $\neg F = \{\neg f \mid f \in F\}$. Algunas veces podemos usar *not* en lugar de $\neg$ y a, b en lugar de $a \wedge b$ siguiendo la notación tradicional de programación lógica. Una *teoría regular* o *programa lógico* es solo un conjunto finito de formulas bien formadas o reglas, estas pueden ser llamadas *teoría* o *programa*. Definimos como una *regla* a cualquier fórmula bien formada de la forma: $f \leftarrow g$, la parte de la izquierda y derecha de "$\leftarrow$" se llaman cabeza y cuerpo de la regla respectivamente [21].

Los primeros sistemas de software que se desarrollaron para calcular Answer Set de los programas lógicos: DLV [5] y lparse/Smodels [8], demostraron que el paradigma de Answer Set Programming tiene un potencial para ser la base de la práctica del cálculo declarativo. Estos dos sistemas de software, sus descendientes y esencialmente todos los

otros sistemas que han desarrollado y aplicado hasta ahora ASP, contienen dos componentes principales [6]: un grounder que es una entrada del programa que se encarga de producir su equivalente proposicional y un solver (solucionador) que acepta el programa de entrada y calcula los Answer Set correspondientes.

La técnica de Answer Set Programming se basa en la posibilidad de representar ciertas colecciones de conjuntos como colecciones de modelos estables [7]. Gracias a la existencia de implementaciones eficientes para calcular modelos estables tales como Smodels[1], DLV[2], Clasp[3], enfocados a resolver problemas en base a la semántica estable, la gama de problemas que podían enfrentarse con este nuevo paradigma creció rápidamente para incluir problemas combinatorios, establecer y resolver problemas de planeación, modelar el comportamiento de agentes lógicos y aplicaciones de inteligencia artificial en general [10]

1.2.1 Semántica Estable

La introducción de la semántica de modelos estables (Stable Models), por Gelfond y Lifschitz [13], significó un importante adelanto en el área de la programación lógica, debido a que se ha convertido en la base de Answer Set Programming. La semántica de Answer Set Programming presenta una característica innovadora, que es el uso de dos tipos de negación [10]: la negación como falla y la negación clásica o explícita. Esta segunda negación se denota con el conectivo ~, que permite la especificación explícita de conocimiento que de antemano se sabe que es falso, por otra parte, la negación como falla que se denota por el símbolo ¬, significa que una fórmula ¬F es cierta si no es posible mostrar que esta fórmula F es verdadera con la información existente.

Por lo tanto, para que un programa lógico tenga un significado y pueda ser interpretado por programas computacionales se debe asignar una semántica, intuitivamente una semántica es una forma de determinar el tipo de conclusiones que se pueden establecer

[1] http://www.tcs.hut.fi/Software/smodels/
[2] http://www.dbai.tuwien.ac.at/proj/dlv/
[3] http://www.cs.uni-potsdam.de/clasp/

a partir de un conjunto de reglas. De tal manera, un problema debe codificarse en la forma de un programa lógico buscando que la semántica del programa capture precisamente las soluciones del problema en cuestión [10]. Debido a esto, surgen lenguajes que permiten escribir programas lógicos que soportan dichas expresiones tales como el uso de Answer Set Programming.

El éxito de la semántica de los modelos estables y posteriormente de los Answer Sets, se debió en gran medida a su capacidad para resolver problemas.

1.2.2 Sistemas ASP

Actualmente existen varios solucionadores para Answer Set, para este caso se hará mención de 2 de ellos: Smodels y Clasp, debido a que serán los solucionadores que se usarán para el desarrollo de la implementación.

1.2.2.1 Smodels

Smodels fue desarrollado por Patrik Simons en el Laboratorio de Ciencias Computacionales Teóricas de la Universidad Tecnológica de Helsinki, es una eficiente implementación de la semántica de modelos estables para programas lógicos, es un sistema para Answer Set Programming. A su vez Smodels necesita de lparse, el cual es un front – end que se encarga de transformar los programas del usuario de tal manera que Smodels pueda entenderlos.

1.2.2.2 Clasp

Clasp es un sistema solucionador de Answer Sets para extender programas lógicos normales que forma parte de la suite de los laboratorios de Potassco[4]. Clasp permite además el uso de declaraciones de optimización mediante las herramientas gringo y clingo de

[4] http://potassco.sourceforge.net/

Potassco, para establecer preferencias entre los answer sets resultantes y poder definir si se trata de un answer set óptimo para el programa lógico definido.

1.3 Enfoque considerado para preferencias

En esta sección se presenta el enfoque basado en ASP y que se va a implementar en este trabajo de tesis, también se explica cómo es la forma de expresar un programa de preferencias.

Definición 1 [24]. *Una regla de preferencia es una fórmula de la forma:* $f_1 * \cdots * f_n \xleftarrow{pr} a_1,\ldots, a_n, not\ b_1,\ldots, not\ b_k$ *donde* $f_1,\ldots, f_n, a_1,\ldots, a_n, b_1,\ldots, b_k$ *son átomos. Un programa de preferencia lógica es un conjunto finito de reglas de preferencia y un conjunto arbitrario de fórmulas bien formadas.*

Si la regla de preferencia no tiene *átomos* $a_1,\ldots, a_n$, *o* $b_1,\ldots, b_k$ *entonces* se puede escribir como $f_1 * \cdots * f_n \xleftarrow{pr}$. Las fórmulas $f_1 \ldots f_n$ se llaman opciones de una regla de preferencia.

Ejemplo 1. Juan quiere practicar algún deporte en su tiempo libre, por lo cual se ha inscrito en un club que tiene los siguientes deportes: futbol, voleibol, ciclismo y baloncesto. Si no hay futbol en el horario que tiene tiempo entonces él prefiere practicar ciclismo a baloncesto y baloncesto a voleibol. Pero en general a él le gusta más el futbol al ciclismo. Con el horario del que dispone no puede practicar algunos deportes: si elige practicar ciclismo entonces no podrá jugar baloncesto y viceversa, si elige jugar voleibol entonces no podrá practicar futbol y viceversa. Por lo tanto, las opciones que tiene para practicar deporte y las opciones que tiene el club pueden ser simplemente representadas como el siguiente programa PL,

```
ciclismo*baloncesto*voleibol ←pr not futbol.
futbol*ciclismo ←pr.
ciclismo ← not baloncesto.
baloncesto ← not ciclismo.
voleibol ← not futbol.
futbol ← not voleibol.
```

Los Answer Sets de un programa PL son los Answer Sets de la lógica del programa obtenido mediante la extracción de las reglas de preferencia desde el programa original PL.

Definición 2 [24]. *Dejemos Pref ser el conjunto de reglas de preferencia de un programa PL P. Sea M un conjunto de átomos. M es un Answer Set de P si y sólo si M es un Answer Set de P \ Pref.*

Así, los Answer Sets del programa PL del Ejemplo 1 son los siguientes Answer Sets:

```
{ voleibol baloncesto }, { voleibol ciclismo },
  { futbol ciclismo } y { futbol baloncesto }
```

Parte de la semántica de los programas de PL fue inspirado por la semántica de Programas Lógicos con Disyunciones Ordenadas (LPOD) presentado en [19]. Debido a la falta de espacio no se presenta la semántica de los programas LPOD, pero los lectores no familiarizadas con este enfoque puede revisar [19]. La primera diferencia entre la semántica LPOD y la semántica de los programas de PL es la sintaxis general de los programas PL. Por lo tanto, la semántica de los programas de PL se define para las teorías generales (ver Definición 1) y la semántica para LPOD no. Una segunda diferencia entre estas dos semánticas es el hecho de que LPOD representan una disyunción sobre opciones de preferencia y los programas de PL representan preferencia sobre la preferencia de opciones. Así, en algunos casos, el Answer Set preferido se obtiene por la aplicación de Semántica LPOD son diferentes de los Answer Set preferidos obtenidos por la aplicación de semánticas para programas PL a su representación respectiva del mismo problema.

Definición 3 [24]. *Sea M un Answer Set de un programa PL P. Sea:* $r = f_1 * \cdots * f_n \leftarrow^{pr} a_1, \ldots, a_n$, *not* $b_1, \ldots,$ *not* b_k *ser una regla de preferencia de P. Sea m el max* $\{n \mid f_1 * \cdots * f_n \leftarrow^{pr} a_1, \ldots, a_n$, *not* $b_1, \ldots,$ *not* b_k *es una regla de preferencia de P*$\}$. *Se define el grado de satisfacción de r en M, denotado por* $deg_M(r)$*, como una regla de correspondencia que define la siguiente función:*

1. 1 a_j *no* $\notin M$*, para alguna j, o* $b_i \in M$*, para alguna i,*

2. *Grado j* $(1 \leq j \leq n)$ *si toda* $a_j \in M$, $b_i \notin M$, *y* $j = min\{r \mid f_r \in M\}$.

3. m + 1 *si* $a_j \in M$ *y* $b_i \notin M$, *además no hay* $i_1, \ldots, i_n$ *tal que* $f_i \in M$.

La parte (1) de la Definición 3 indica que la regla r tiene un grado de satisfacción igual a 1 en el Answer Set, porque, la regla r no se aplica y este hecho hace la regla r irrelevante para preferir el Answer Set. La parte (2) de la definición 3 indica que esta regla r se cumple en el Answer Set en cierto grado. Finalmente, la parte (3) de la Definición 3 indica que la regla r tiene el mayor valor de grado de satisfacción en el Answer Set, porque ninguna de las opciones de preferencia de la regla r se mantiene.

De acuerdo con nuestra intuición, esta parte (3) de la Definición 3 será útil en caso que ninguna de las opciones de cada regla de preferencia en P se mantenga en todos los Answer Sets de P. Así, en este caso, todos los Answer Sets del programa PL deben ser preferidos.

Por ejemplo, si tengo una preferencia por las galletas sobre la fruta y el jugo de naranja sobre agua, y uno de los dos Answer Sets contiene sólo ensalada y el otro solo café. En este caso, ambos Answer Sets son incomparables y deben ser preferidos.

Ejemplo 2. Sea P el programa PL del Ejemplo 1. Por Definición 2 sabemos que el programa P tiene cuatro Answer Sets: M_1*={voleibol baloncesto}*, M_2*={voleibol ciclismo}*, M_3*={futbol ciclismo} y* M_4*={futbol baloncesto}*. De acuerdo con la definición 3 podemos comprobar que m = 3 y que,

$$deg_{M1}(r_1) = 1, \quad deg_{M1}(r_2) = 4$$
$$deg_{M2}(r_1) = 1, \quad deg_{M2}(r_2) = 2$$
$$deg_{M3}(r_1) = 1, \quad deg_{M3}(r_2) = 1$$
$$deg_{M4}(r_1) = 1, \quad deg_{M4}(r_2) = 1$$

Es importante mencionar que $deg_{M1}(r_2)$ es igual a M + 1 = 4, ya que ninguna de las opciones de r_2 está en M_1 (Véase la parte (3) de la definición 4).

Los siguientes teoremas y definiciones son de Answer Sets preferidos de un programa PL. Todos ellos son similares a las definiciones dadas en [19]. Sin embargo, no

hay que olvidar que se definen para las teorías generales (ver Definición 2) y se basan en nuestra propia definición de grado de satisfacción.

Teorema 1. *Sea Pref el conjunto de reglas de preferencia de un programa de PL P. Si M es un answer set de P, entonces M satisface todas las reglas en Pref hasta cierto punto.*

El grado de satisfacción de cada regla de preferencia nos permite definir el conjunto de reglas de preferencia con el mismo grado de satisfacción. Estos conjuntos se utilizarán para encontrar el Answer Set preferido del programa PL.

Definición 4 [24]. *Sea P un programa PL y Pref el conjunto de reglas de preferencia de P. Sea M un Answer Set de P. Definimos* $S^i_M(P) = \{r \in Pref \mid deg_M(r) = i\}$.

Ejemplo 3. Sea P el programa PL del Ejemplo 1. Vamos a considerar los grados de satisfacción de las reglas r_1 y r_2 en el Ejemplo 2. Entonces, podemos comprobar que,

$$S^1_{M1}(P) = \{r1\} \qquad S^2_{M1}(P) = \{\} \qquad S^4_{M1}(P) = \{r2\}$$
$$S^1_{M2}(P) = \{r1\} \qquad S^2_{M2}(P) = \{r2\} \qquad S^4_{M2}(P) = \{\}$$
$$S^1_{M3}(P) = \{r1, r2\} \qquad S^2_{M3}(P) = \{\} \qquad S^4_{M3}(P) = \{\}$$
$$S^1_{M4}(P) = \{r1, r2\} \qquad S^2_{M4}(P) = \{\} \qquad S^4_{M4}(P) = \{\}$$

La siguiente definición indica la forma de aplicar un criterio a los conjuntos de preferencia las reglas $S^i_M(P)$ de un programa PL, con el fin de saber si un Answer Set se prefiere a otro Answer Set y como obtener los Answer Sets más preferidos. El criterio utilizado es el conjunto de inclusión.

Definición 5 [24]. *Sea M y N Answer Sets de un programa PL P. M es preferido por inclusión a N, denotado como* $M >_i N$*, si y sólo si existe un i tal que* $S^i_N(P) \subset S^i_M(P)$ *y para todo* $j < i$, $S^j_M(P) = S^j_N(P)$.

Ejemplo 4. Sea P el programa PL del Ejemplo 1. Tomamos en cuenta los resultados de la definición 4 para saber que Answer Set es el preferido de entre dos Answer Sets.

Como resultado después de aplicar la definición 5 obtenemos las siguientes tablas comparativas:

M	N	Preferido
$S^1_{M1}(P)= \{r1\}$	$S^1_{M2}(P)= \{r1\}$	Ninguno
$S^1_{M2}(P)= \{r1\}$	$S^1_{M3}(P)= \{r1, r2\}$	$N >_i M$
$S^1_{M3}(P)= \{r1, r2\}$	$S^1_{M4}(P)= \{r1, r2\}$	Ninguno
$S^1_{M4}(P)= \{r1, r2\}$	$S^1_{M1}(P)= \{r1\}$	$M >_i N$
$S^1_{M3}(P)= \{r1, r2\}$	$S^1_{M1}(P)= \{r1\}$	$M >_i N$
$S^1_{M4}(P)= \{r1, r2\}$	$S^1_{M2}(P)= \{r1\}$	$M >_i N$

M	N	Preferido
$S^2_{M1}(P)= \{\}$	$S^2_{M2}(P)= \{r2\}$	$N >_i M$
$S^2_{M2}(P)= \{r2\}$	$S^2_{M3}(P)= \{\}$	$M >_i N$
$S^2_{M3}(P)= \{\}$	$S^2_{M4}(P)= \{\}$	Ninguno
$S^2_{M4}(P)= \{\}$	$S^2_{M1}(P)= \{\}$	Ninguno
$S^2_{M3}(P)= \{\}$	$S^2_{M1}(P)= \{\}$	Ninguno
$S^2_{M4}(P)= \{\}$	$S^2_{M2}(P)= \{r2\}$	$N >_i M$

M	N	Preferido
$S^4_{M1}(P)= \{r2\}$	$S^4_{M2}(P)= \{\}$	$M >_i N$
$S^4_{M2}(P)= \{\}$	$S^4_{M3}(P)= \{\}$	Ninguno
$S^4_{M3}(P)= \{\}$	$S^4_{M4}(P)= \{\}$	Ninguno
$S^4_{M4}(P)= \{\}$	$S^4_{M1}(P)= \{r2\}$	$N >_i M$
$S^4_{M3}(P)= \{\}$	$S^4_{M1}(P)= \{r2\}$	$N >_i M$
$S^4_{M4}(P)= \{\}$	$S^4_{M2}(P)= \{\}$	Ninguno

Lo cual nos indica cual Answer Set es el preferido si comparamos dos Answer Sets.

Definición 6 [24]. *Un conjunto de átomos M son el Answer Set mas preferido por inclusión de un programa P, si M es un Answer Set de P y no hay un Answer Set M' de P,* $M \neq M'$, *tal que* $M' >_i M$.

Ejemplo 5. Utilizando los resultados del ejemplo 3 y la tabla comparativa del ejemplo 4 podemos verificar por la definición 6 que M_3 es el Answer Set mas preferido por inclusión ya que en los resultados del ejemplo 3 se puede apreciar que $S^1_{M3}(P)= \{r1, r2\}$ es

el primero que contiene a los otros conjuntos del mismo grado, en este caso grado 1. Revisando la tabla comparativa donde M es $S^{1}_{M2}(P)= \{r1\}$ y N es $S^{1}_{M3}(P)= \{r1, r2\}$ podemos verificar que $N >_i M$. Entonces M_3 corresponde a *{ futbol ciclismo }*.

Esto quiere decir que la mejor opción es practicar futbol y ciclismo ya que el horario de Juan se ajusta para esos deportes.

Capítulo 2. Análisis

El proyecto consiste en analizar e implementar el enfoque para representar preferencias. El enfoque a implementar es un enfoque basado en semántica de programación lógica por lo que la representación de la información corresponde a programas lógicos. En particular el enfoque de preferencias a implementar se basa en la semántica de programación lógica estable. La implementación está hecha en el lenguaje Java y un software que obtiene la semántica estable.

Se va a analizar un programa lógico que es un conjunto de reglas. Una regla puede estar formada por cabeza y cuerpo los cuales están separados por el signo "←". Las reglas que tienen opciones separadas por el signo "*" son llamadas reglas de preferencia. Las reglas restantes son reglas que no pertenecen a las reglas de preferencia que se usaran para la obtención de los Answer Sets. Del programa lógico nos interesa obtener los modelos de las reglas que no pertenecen a preferencias, también los grados de satisfacción de cada regla de preferencia y los modelos más preferidos.

El objetivo de este trabajo está basado en el capítulo 1 el cual consiste en modelar y resolver el problema de preferencias, para lo cual se siguen los siguientes pasos:

1. Definir un conjunto de reglas.
2. Identificar y separar estas reglas en reglas de preferencias y reglas que no corresponden a las preferencias.
3. Encontrar los Answer Sets de las reglas que no son reglas de preferencia con Smodels o Clasp.
4. Calcular los grados de satisfacción con las reglas de preferencia y los Answer Sets.
5. Calcular el conjunto como se ve en el capítulo 1 definición 4.
6. Hacer comparaciones entre Answer Sets.
7. Obtener el Answer Set más preferido.

2.1 Estructuras usadas para almacenar los datos

Para almacenar los datos que nos servirán para obtener nuestro propósito necesitamos de algunas estructuras que presentamos a continuación:

2.1.1 Almacenamiento del programa

Tomemos en cuenta el ejemplo 1 del capítulo 1. Nuestro programa estará escrito en un archivo de texto de la siguiente forma, donde sustituimos $\leftarrow^{pr}$ por el signo “<” y $\leftarrow$ por “:-”, de tal forma que el contenido del archivo será lo siguiente:

```
ciclismo*baloncesto*voleibol < not futbol.
futbol*ciclismo.
ciclismo:- not baloncesto.
baloncesto:- not ciclismo.
voleibol:- not futbol.
futbol:- not voleibol.
```

2.1.2 Almacenamiento de las reglas de preferencia

El almacenamiento de cada regla de preferencia se hace en un arreglo dinámico, ya que no sabemos cuántas reglas tendrá el programa. Dicho arreglo se llama ***ReglaDePreferencia***, donde cada regla de preferencia tiene lo siguiente:

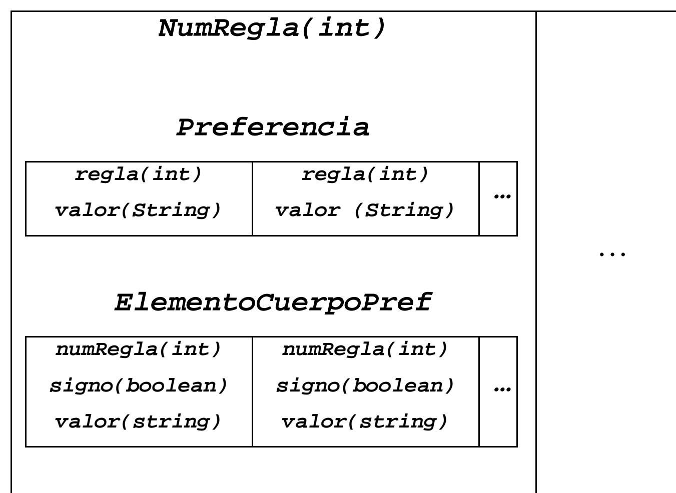
1. Una variable llamada ***NumRegla*** de tipo entero que nos indica propiamente el número de regla que estamos almacenando.
2. Un arreglo donde se guarda la cabeza de la regla llamado ***Preferencia***
3. Un arreglo donde se guarda el cuerpo de la regla llamado ***ElementoCuerpoPref***.

El arreglo ***Preferencia*** contiene a su vez lo siguiente: una variable llamada ***regla*** de tipo entero que nos indica el número de preferencias y una variable que almacena una preferencia (literal) de tipo String llamada ***valor.*** Por ejemplo en la

siguiente regla de preferencia: `ciclismo*baloncesto*voleibol` se tienen 3 preferencias que son: 1.- ciclismo, 2.- baloncesto y 3.- voleibol.

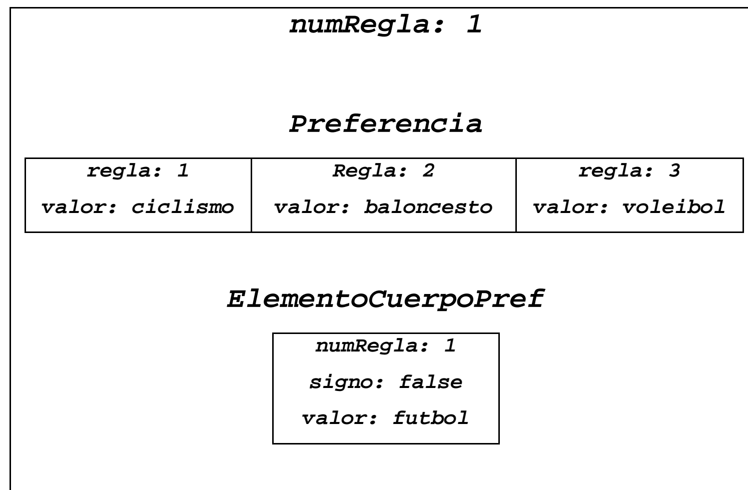
El arreglo ***ElementoCuerpoPref*** contiene lo siguiente: una variable llamada ***numRegla*** que es de tipo entero y que nos servirá para llevar el conteo de literales en una regla de preferencia. Para representar la palabra **not** como en *not futbol* tenemos una variable llamada ***signo*** de tipo boolean, en la cual si el signo es true entonces nos indica que no aparece la palabra *not*, si es signo es false entonces el signo es negativo porque aparece la palabra *not*. Para almacenar una literal del cuerpo también tenemos una variable llamada ***valor*** de tipo String.

La estructura del arreglo ***ReglaDePreferencia*** sería la siguiente:



Entonces si en el arreglo ***ReglaDePreferencia*** guardamos la siguiente regla, estará almacenada de la siguiente manera:

```
ciclismo*baloncesto*voleibol < not futbol.
```



Las reglas que no son reglas de preferencia son guardadas en un archivo de texto llamado Temporal.txt el cual se utilizara para obtener los modelos (Answer Set) con Clasp o Smodels.

2.1.3 Almacenamiento de los modelos (Answer Sets)

Para guardar los modelos utilizaremos una estructura parecida a la anterior. Estos modelos se guardan en un arreglo dinámico llamado ***`Modelo`***, donde cada modelo tiene lo siguiente:

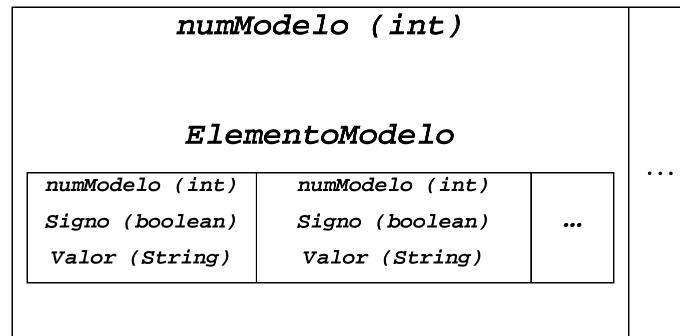
1. Una variable llamada ***`numModelo`*** de tipo entero que nos indica el conteo de los modelos que tenemos,
2. Un arreglo que contiene los modelos llamado ***`ElementoModelo`***.

El arreglo ***`ElementoModelo`*** contiene a su vez una variable que lleva el número de modelo (numero que corresponde a cada elemento del modelo encontrado) llamado ***`numModelo`*** de tipo entero, contiene también una variable llamada ***`signo`*** de tipo boolean que será siempre positiva (true) y nos servirá simplemente para hacer algunas comparaciones entre los modelos y el cuerpo de las reglas de preferencia para obtener los

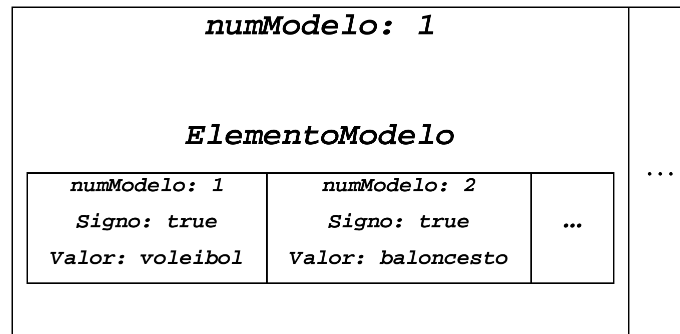
grados de satisfacción, contiene también una variable llamada ***valor*** que almacenara un elemento del modelo y es de tipo String.

Por ejemplo el modelo {`voleibol baloncesto`} tiene 2 elementos que son 1.- voleibol y 2.- baloncesto.

La estructura del arreglo **`Modelo`** sería la siguiente:



Por ejemplo, si tuviéramos el modelo { `voleibol baloncesto` } este estaría almacenado de la siguiente forma:



2.1.4 Almacenamiento de los grados de satisfacción

Una vez separada y almacenada la información, se necesita obtener los grados de satisfacción. El almacenamiento de los grados de satisfacción se hace en un arreglo dinámico llamado **`Grados`** que contiene lo siguiente:

1. El número de modelo que se almacena en la variable **`numMod`** de tipo entero.

2. El número de regla de preferencia que se almacena en la variable **`numPref`** de tipo entero .
3. El grado que se almacena en la variable **`grado`** de tipo entero.

Donde el número del modelo y número de regla de preferencia es el número de modelo y regla de preferencia que se está comparando y grado es el número obtenido después de aplicar las reglas para obtener los grados de satisfacción. La forma de obtener estos grados se describe en el capítulo 1 definición 3.

Teniendo en cuenta lo anterior, la estructura del arreglo **`Grados`** sería la siguiente:

numMod (int) ***numPref (int)*** ***grado (int)***	…

De esta manera, los siguientes grados obtenidos en el ejemplo 2 del capítulo 1 quedan almacenados de la siguiente forma:

$$deg_{M1}(r_1)= 1,\quad deg_{M1}(r_2)= 4$$
$$deg_{M2}(r_1)= 1,\quad deg_{M2}(r_2)= 2$$
$$deg_{M3}(r_1)= 1,\quad deg_{M3}(r_2)= 1$$
$$deg_{M4}(r_1)= 1,\quad deg_{M4}(r_2)= 1$$

NumMod:1 *NumPref:1* *Grado:1*	*NumMod:2* *NumPref:1* *Grado:1*	*NumMod:3* *NumPref:1* *Grado:1*	*NumMod:4* *NumPref:1* *Grado:1*	*NumMod:1* *NumPref:2* *Grado:4*	*NumMod:2* *NumPref:2* *Grado:2*	*NumMod:3* *NumPref:2* *Grado:1*	*NumMod:4* *NumPref:2* *Grado:1*

2.1.5 Almacenamiento del conjunto S

Una vez obtenidos los grados procederemos a almacenar algunos conjuntos los cuales ya se han explicado cómo se obtienen en el capítulo 1 definición 5. La forma de

almacenar estos conjuntos esta dado por el siguiente arreglo dinámico llamado ***ConjuntoS*** que contiene lo siguiente:

1. El número de modelo que se almacena en la variable ***modelo*** y es de tipo entero.
2. El grado se almacena en la variable ***grado*** de tipo entero.
3. Una colección llamada ***r*** de tipo Set que almacena las reglas de preferencia que coincidan con el modelo y el grado especificado.

La estructura del arreglo **ConjuntoS** sería la siguiente:

grado(int) ***modelo(int)*** ***r (set)***	**…**

Entonces los resultados del ejemplo 3 capítulo 1, los conjuntos quedan almacenados de la siguiente forma:

grado: 1 ***modelo: 1*** ***r*** ***r1***	***grado: 1*** ***modelo: 2*** ***r*** ***r1***	***grado: 1*** ***modelo: 3*** ***r*** ***r1 r2***	***grado: 1*** ***modelo: 4*** ***r*** ***r1 r2***	***grado: 2*** ***modelo: 1*** ***r*** ***-***	**…**

Capítulo 3. Diseño

Aquí se presenta el diseño tanto de la interfaz como de los algoritmos que procesan las preferencias y nos dan una respuesta a nuestro problema. A continuación se van a presentar los algoritmos que se han utilizado para hacer dichos cálculos. Estos algoritmos corresponden a lo mencionado en el análisis.

3.1 Identificación de reglas de preferencia y reglas que no son reglas de preferencia

Una vez que tenemos nuestro programa lógico guardado en un archivo de texto haremos lo siguiente para identificar estas reglas:

1. Leer línea a línea el texto guardado.
2. Por cada línea leída leer carácter a carácter si tiene algún signo *.
3. Si encontramos un * quiere decir que es una regla de preferencia y se deja intacta.
4. Si encontramos una línea sin * entonces la guardamos en un archivo de texto llamado Temporal

3.2 Identificación y almacenamiento de la cabeza y el cuerpo de las reglas de preferencia

Como hemos dicho anteriormente en el capítulo 1, una regla de preferencia puede tener cabeza y cuerpo, los cuales están separados por el signo “<”, es decir, si analizamos la siguiente regla:

```
ciclismo*baloncesto*voleibol < not futbol.
```

Podemos decir que la cabeza es: `ciclismo*baloncesto*voleibol` y el cuerpo es: `not futbol.`

Comenzaremos a analizar este regla de preferencia de la siguiente manera:

1. La línea se almacena en un arreglo de caracteres.
2. Se empieza a leer carácter por carácter de izquierda a derecha.
3. Hasta donde se encuentre el signo “<” corresponde a lo que sería la cabeza.
4. Lo restante de la línea se considerara como el cuerpo de la regla.

3.2.1 Obtención y almacenamiento de la cabeza

Para el almacenamiento de la cabeza se utilizara una parte de la estructura 2.1.2 del capítulo 2 en el que se tiene la variable **numRegla** que almacena el numero de regla de preferencia que se está analizando y el arreglo llamado **Preferencia** que almacena el numero y contenido de la literal.

Entonces teniendo la cabeza `ciclismo*baloncesto*voleibol,` se procede a realizar lo siguiente para su almacenamiento:

1. Los signos * y los puntos se reemplazan por espacios quedando solo las literales
2. Se va guardando en el arreglo **Preferencia** una sola literal y el numero de literal correspondiente.
3. Se guarda también el numero de regla de preferencia que estamos analizando.

3.2.2 Obtención y almacenamiento del cuerpo

Para el almacenamiento del cuerpo de una regla de preferencia se utiliza la parte restante de la estructura 2.1.2 del capítulo 2 donde se utilizara un arreglo llamado ElementoCuerpoPref que precisamente guarda el cuerpo de las preferencias. Teniendo entonces el cuerpo `not futbol` se realizara lo siguiente para almacenar el cuerpo:

1. Se separa el cuerpo en partes por el carácter “,” quedando cada parte con su literal y la palabra not si la hay.
2. Ahora cada parte es separada por espacios y se empieza a comparar cada palabra en busca de la palabra not.

3. Se empieza a almacenar cada palabra si esa palabra es igual a la cadena “not” entonces se almacena el numero de regla que corresponde al número de cuerpo encontrado, el signo es false porque se encontró la palabra not y el contenido que corresponda a la literal.
4. Si no se encuentra la palabra “not” entonces se almacena lo del puto anterior solo que el signo es true (ausencia de not).

3.3 Obtención y almacenamiento de modelos

La obtención de modelos se hará dependiendo de lo que el usuario elija para obtenerlos, pudiendo elegir Smodels o Clasp.

3.3.1 Obtención y almacenamiento de modelos con Smodels

Si se selecciona buscar modelos con Smodels se hará lo siguiente para guardar el resultado que nos de la ejecución de Smodels con las instrucciones en el archivo Temporal.txt:

1. Se manda a ejecutar el comando: “cmd /c lparse Temporal.txt | smodels 0”, donde Temporal.txt es el archivo que contiene las reglas que no son de preferencias.
2. Con getInputStream() capturamos lo que nos arroja Smodels.
3. Se guarda esta salida en un archivo de texto llamado Modelos.txt ubicado en la carpeta llamada Carpeta Resultados.

Hasta aquí tenemos en un archivo todo lo que nos devuelve Smodels, ahora procederemos a seleccionar solo los modelos, ya que cuando Smodels nos devuelve los modelos lo hace con información que no necesitamos, por lo cual es necesario hacer una selección de lo que nos sirve poder continuar con el análisis. Para poder almacenar esta información haremos uso de la estructura 2.1.3 del capítulo 2:

1. Abrimos el archivo llamado Modelos.txt
2. Leemos cada línea buscando la palabra “Stable” ya que a partir de esta palabra están los modelos estables que necesitamos.

3. Se elimina la cadena “Stable Model:”.
4. Y con el texto restante se separa en nuevas cadenas teniendo como delimitador de separación el espacio vacío.
5. Ahora se guarda cada una de estas cadenas en el arreglo **Modelo** donde **numModelo** es el numero de modelo que estamos analizando, en el arreglo interno **ElementoModelo** guardamos de forma numerada cada uno de los elementos encontrados, agregamos también el signo que siempre es true y el modelo encontrado.

3.3.2 Obtención y almacenamiento de modelos con Clasp

Si se selecciona buscar los modelos con Clasp se hará lo siguiente:

1. Se manda a ejecutar el comando: “cmd /c gringo Temporal.txt | clasp 0”, donde Temporal.txt es el archivo que contiene las reglas que no son de preferencia.
2. Con getInputStream() capturamos lo que nos arroja Clasp.
3. Se guarda esta salida en un archivo de texto llamado Modelos.txt ubicado en la carpeta llamada Carpeta Resultados.

Hasta aquí tenemos en un archivo todo lo que nos devuelve Clasp, ahora procederemos a seleccionar solo los modelos ya que como habíamos dicho anteriormente en Smodels, Clasp también nos devuelve información que no necesitamos

1. Abrimos el archivo llamado Modelos.txt
2. Leemos cada línea buscando la palabra “Answer” ya que a partir de esta palabra están los modelos estables que necesitamos
3. Empezaremos a separar las palabras a partir de la siguiente línea.
4. Ahora separamos estas palabras teniendo como delimitador el espacio vacío.
5. Se guarda cada una de estas cadenas en el arreglo **Modelo** donde **numModelo** es el numero de modelo que estamos analizando, en el arreglo interno **ElementoModelo** guardamos de forma numerada cada uno de los elementos encontrados, agregamos también el signo que siempre es true y el modelo encontrado.

3.4 Obtención y almacenamiento de grados

Cuando el usuario ya ha dado clic en buscar modelos y los ha encontrado se puede proceder a encontrar los grados de satisfacción basados en la definición 3 del capítulo 1. Dicha definición tiene 3 reglas, si una regla no se cumple entonces se aplica la siguiente regla y así sucesivamente. Aquí se utiliza la estructura definida en 2.1.4 del capítulo 2.

Para la primera regla se analizara el cuerpo de la regla de preferencia y los modelos obtenidos los cuales ya han sido almacenados previamente.

1. Se va a comparar el cuerpo de la regla de preferencia con los modelos. Cada elemento del cuerpo se comparara contra todos los elementos almacenados en el arreglo Modelo.
2. Si en el cuerpo aparece la palabra "not" entonces la(s) litreal(s) frente a esta palabra no deben aparecer en el modelo, de lo contrario el grado de satisfacción es 1.
3. Si no aparece la palabra "not" entonces esta(s) literal(s) deben aparecer en el modelo en caso contrario el grado será 1.
4. Si no se encontró el grado entonces pasamos a la regla 2.
5. Si se encontró el grado de satisfacción el cual es 1, entonces se almacena el número de modelo y el número de regla preferencia que se está utilizando y el grado 1.

Para la segunda regla se analizara solo la parte de la cabeza y los modelos obtenidos para saber así su grado.

1. Cada elemento del modelo se compara contra todos los elementos de la cabeza.
2. Si el elemento del modelo es igual al elemento de la cabeza entonces el grado de satisfacción es la posición del elemento de la cabeza que coincidió con el elemento del modelo. Siendo la posición 1 el primer elemento de la cabeza.
3. Si ya se ha encontrado el grado de satisfacción entonces se guarda el número de modelo, el número de regla de preferencia que se está utilizando y el grado, el cual es la posición en la que se encontró la igualdad.
4. Si ningún elemento del modelo coincide con algún elemento de la cabeza entonces se pasa a la regla 3

Para la tercera regla se analizara solo la cabeza.

1. Asignamos a una variable m el número máximo de preferencias
2. El grado de satisfacción será m+1 para esta comparación.
3. Entonces procedemos guardar el numero de modelo que se estaba comparado, el numero de regla de preferencia que se estaba utilizando y en grado guardamos la variable m.

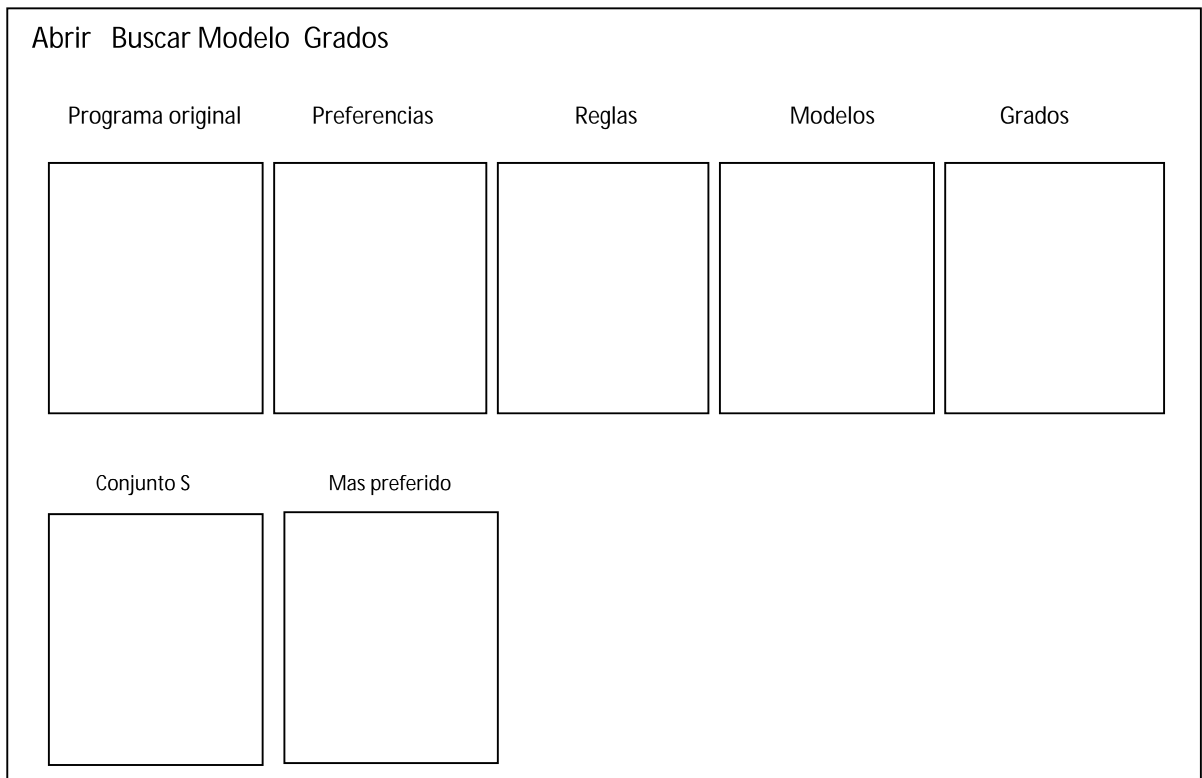
3.5 Obtención y almacenamiento del conjunto S

El conjunto S es: $S^{i}_{M,}$ donde i es el grado y M es el modelo, en este caso se hace una combinación entre los grados y los modelos, y se busca las reglas de preferencia que tengan grado i y modelo M. La forma de hacer las comparaciones y encontrar este conjunto esta explicado en la definición 4 del capitulo 1.

1. Se hará un ciclo entre el máximo de grados y el máximo de modelos
2. Si en el arreglo Grado el campo "**grado**" es igual al número de la iteración en curso y el campo "**modelo**" del arreglo Modelo es igual al número de la iteración en curso entonces se concatena el campo "**numPref**" del arreglo Grados.
3. Cada que deje de coincidir el grado y el modelo entonces se guardara en la colección r el modelo, grado y la cadena que contiene todos los números de preferencia que coincidieron.

3.6 Diseño de la interfaz del programa

En cuanto a la interfaz del programa como primera aproximación tenemos lo siguiente:



Teniendo en cuenta lo anterior se piensa tener una ventana que contenga los elementos necesarios para poder abrir y seleccionar la forma de evaluar el conjunto de reglas. Para eso de va a diseñar dicha ventana en donde contenga los siguientes menús:

- El menú **abrir** nos permite buscar un archivo que contenga el programa a procesar.
- El menú **buscar modelos** permite buscar los modelos en base a las reglas que no pertenecen a las preferencias.
- El menú de **grados** permite obtener los grados haciendo las comparaciones indicadas en las reglas para la obtención de grados.

También se pondrán algunas áreas de texto donde podremos visualizar el programa y los resultados obtenidos. Estas áreas son las siguientes:

- **Programa original** que nos despliega el programa contenido en el archivo.
- **Preferencias** donde obtenemos solo las preferencias.

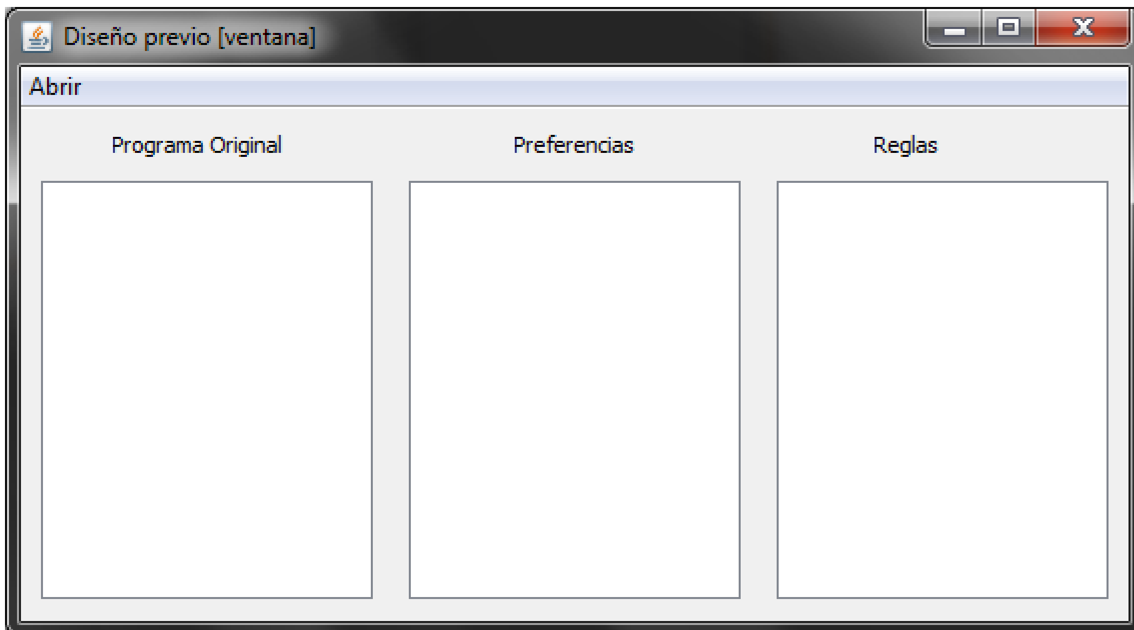
- **Reglas** donde obtenemos las reglas que no corresponden a preferencias.
- **Modelos** donde obtenemos los modelos que tanto Smodels y Clasp nos arrojen al seleccionar una opción del menú "Buscar modelos".
- **Grados** donde visualizaremos los grados correspondientes.
- **Conjunto S** donde visualizaremos los conjuntos formados con las reglas de preferencia y los modelos.
- En la sección de **Mas preferidos** se visualizará que modelo es el más preferido.

Capítulo 4. Implementación

En este capítulo se presenta el desarrollo e implementación del programa. Como primera instancia se ha diseñado una pequeña ventana donde lo único que tiene es un menú para abrir el archivo de texto que contiene las instrucciones utilizaremos y tres área de texto para presentar la información. Conforme se vaya avanzando en el programa se irán agregando menús y áreas de texto para ir viendo la evolución de este trabajo.

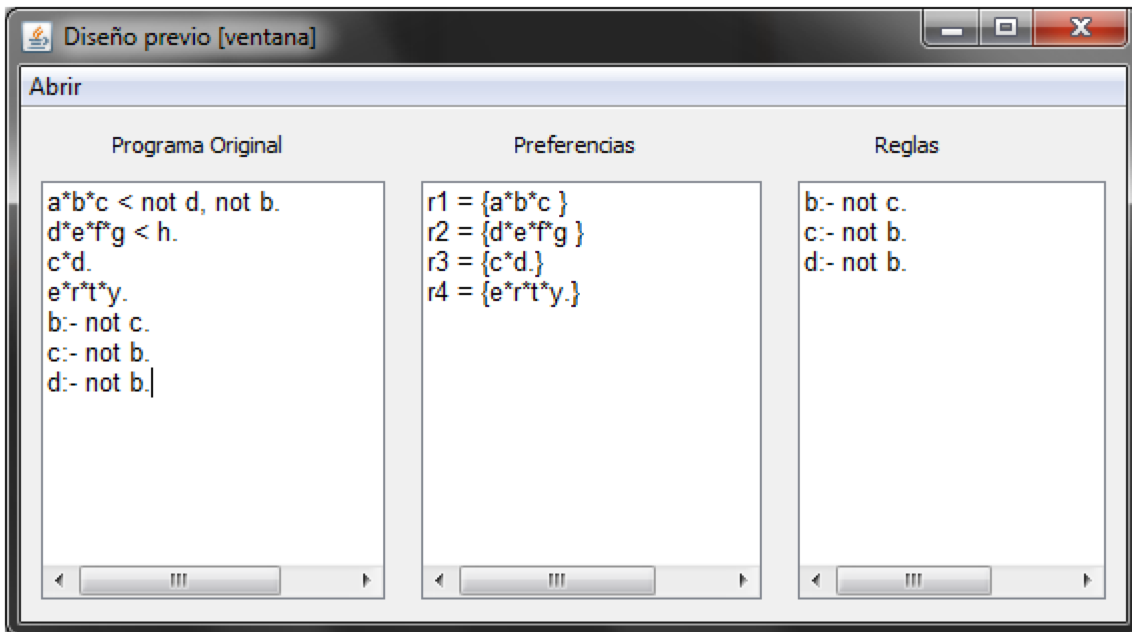
El archivo de texto utilizado contendrá las siguientes reglas como ejemplo:

```
a*b*c < not d, not b.
d*e*f*g < h.
c*d.
e*r*t*y.
b:- not c.
c:- not b.
d:- not b.
```



El menú Abrir sirve para buscar el archivo previamente escrito con las instrucciones, al pulsar en él se abrirá un cuadro de dialogo y seleccionaremos nuestro

archivo y pulsamos aceptar. Inmediatamente después de haber hecho esto aparecerá lo siguiente en nuestra ventana:



Donde en Programa Original aparecerá propiamente el programa que está en el archivo de texto, en la zona de Preferencias aparecerá solamente la parte de preferencias etiquetadas con $\mathbf{r}_i$, en este caso solo tenemos cuatro preferencias. En la zona de Reglas aparecerán solo las líneas que sean reglas del archivo original, en este caso solo son tres.

4.1 Método para buscar los modelos

Posteriormente se ha añadido un nuevo menú llamado Buscar Modelos, este menú tiene dos opciones: Smodels y Clasp.

Si se selecciona Smodels quiere decir que con Smodels se van a buscar los modelos estables de las reglas que no corresponden a preferencias las cuales serian las siguientes:

```
b:- not c.
c:- not b.
d:- not b.
```

Si se selecciona Clasp quiere decir que se van a buscar con Clasp los modelos estables de las mismas reglas que con Smodels.

Parte del código para obtener los modelos es el siguiente:

Commandos:

```
private static final String CLASP_COMMAND = "cmd /c gringo Temporal.txt | clasp 0 ";
private static final String SMODELS_COMMAND = "cmd /c lparse Temporal.txt | smodels 0";
```

```
switch (modelo) {
        case CLASP:
            ejecuta(CLASP_COMMAND);
            identificaModelosClasp();
            gui.getMenuGrados().setEnabled(true);
            gui.setNumReglasClasp(1);
            break;
        case SMODELS:
            ejecuta(SMODELS_COMMAND);
            identificaModelosSmodels();
            gui.getMenuGrados().setEnabled(true);
            gui.setNumReglasSmodels(1);
            break;
    }
```

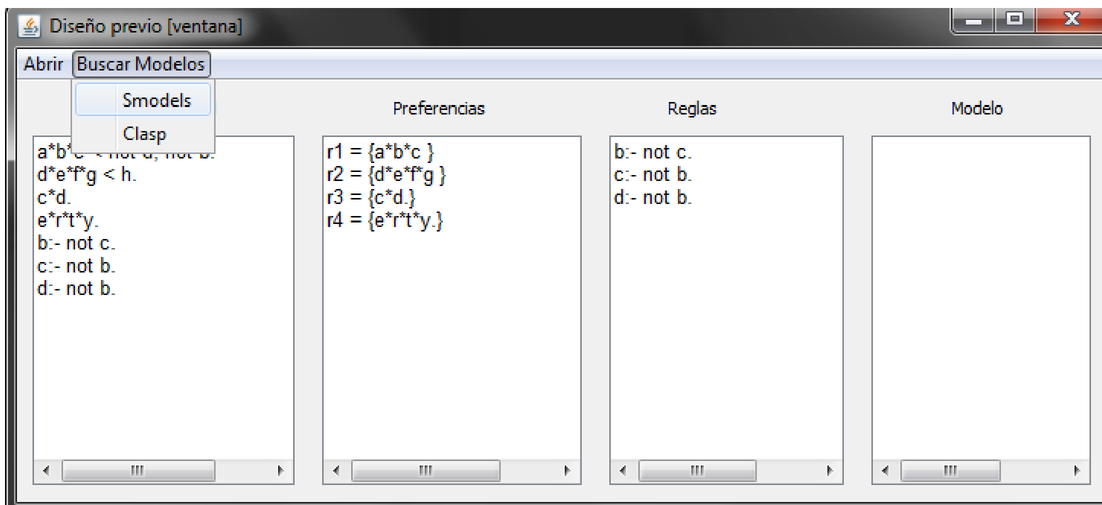
4.2 Método que ejecuta un comando y guarda la salida en modelos.txt

```
private void ejecuta(String comando) {
    try {
        Process p = Runtime.getRuntime().exec(comando);
        InputStream is = p.getInputStream();
```

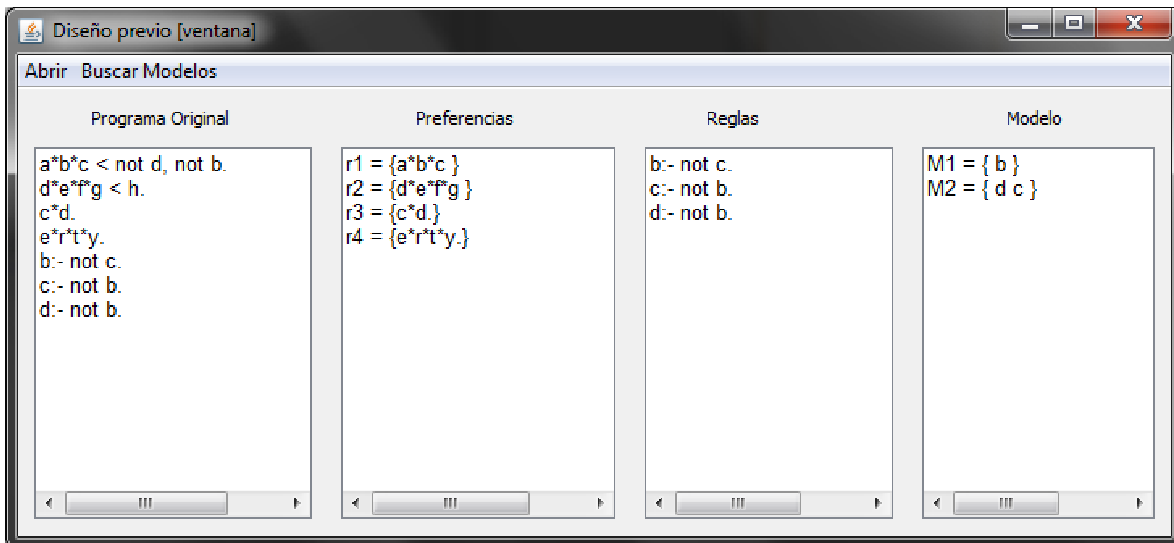
```
        DataInputStream entrada = new DataInputStream(is);
        String linea = "";
        FileWriter fichero = null;
        PrintWriter pw = null;
        File directorio = new File("Carpeta Resultado");
        directorio.mkdir();
        fichero = new FileWriter("Carpeta Resultado//Modelos.txt");
        pw = new PrintWriter(fichero);
        while ((linea = entrada.readLine()) != null) {
            pw.println(linea);
        }
        if (null != fichero) {
            fichero.close();
        }

    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

Esta es la ventana con el menú Buscar Modelos.



Al pulsar por ejemplo en Smodels obtenemos en la zona de Modelo lo siguiente:

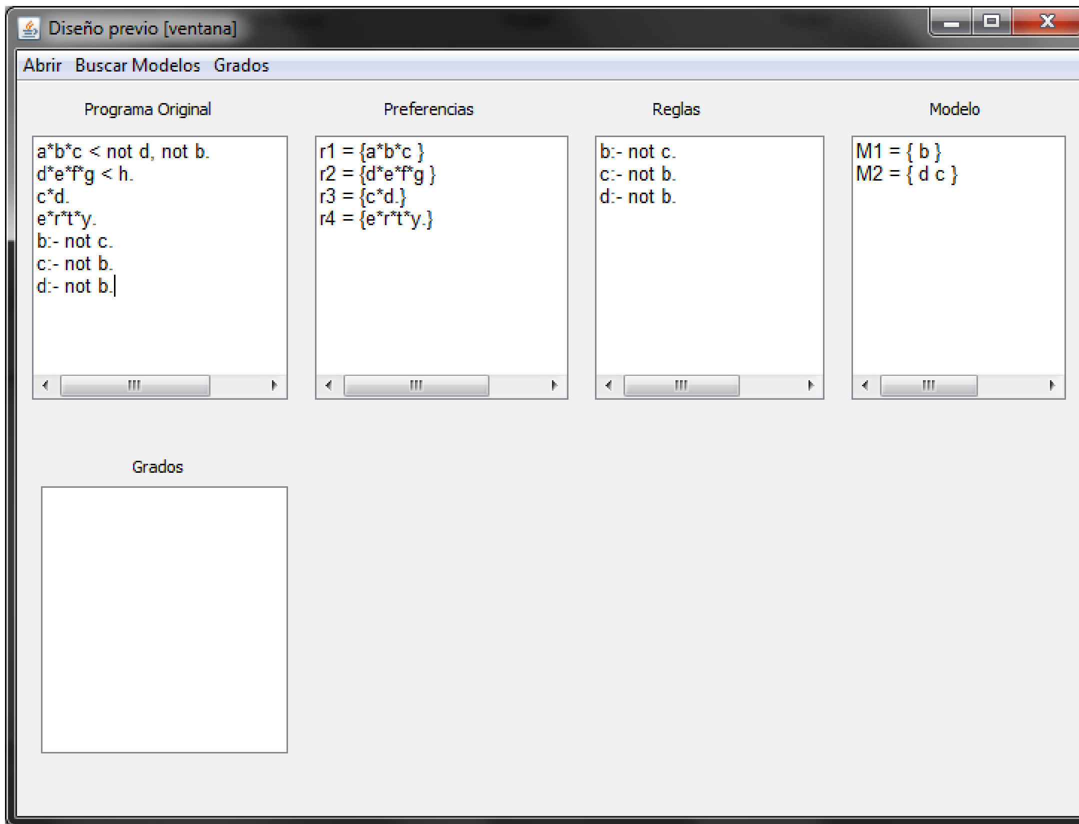


Lo que quiere decir que esos son los modelos estables de las reglas analizadas anteriormente. Si se analizara con Clasp debería de aparecer lo mismo o algo muy similar ya que la forma de analizar las reglas con Smodels o Clasp son muy similares.

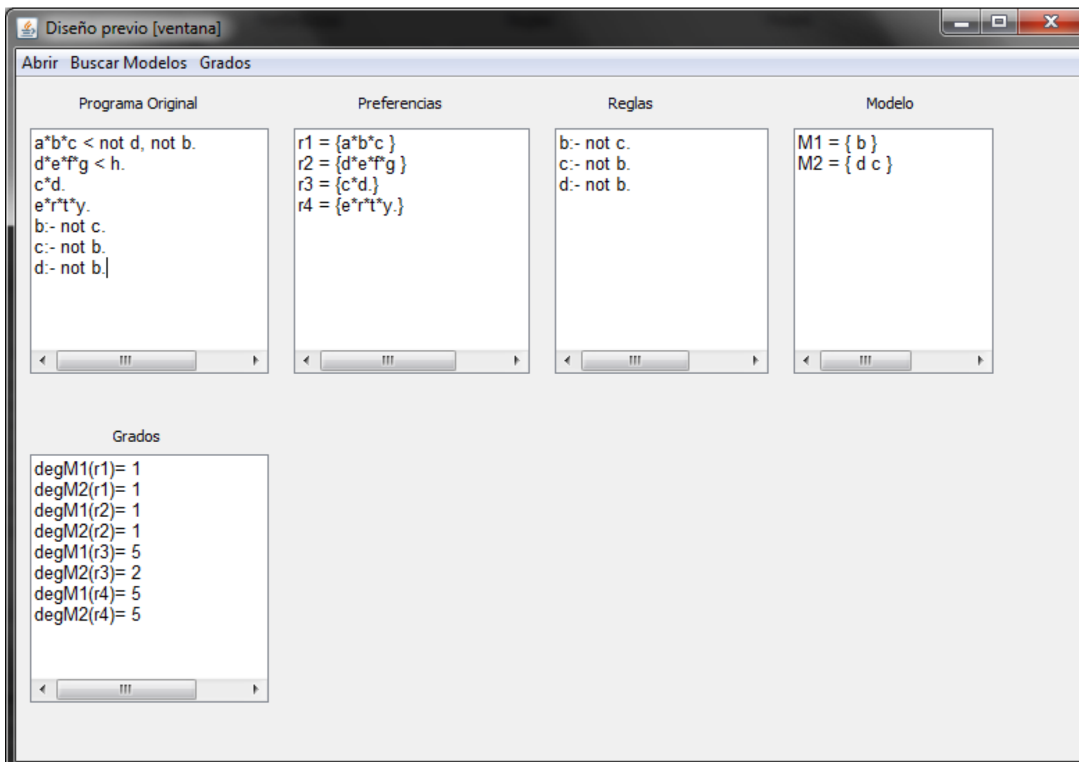
4.3 Método para obtener los grados de satisfacción

Una vez teniendo las Preferencias y modelos podemos obtener los grados de satisfacción del modelo M_i con respecto a la regla r_i.

Para esto se va a agregar un nuevo menú que nos va a servir para obtener sus grados de satisfacción y una caja de texto donde aparezcan dichos grados. La forma de obtener estos grados de satisfacción ya se ha descrito en el capítulo 2, en la parte de Diseño:



Pulsando en el menú Grados se obtiene en la zona de Grados lo siguiente:



Lo cual nos indica el grado de satisfacción que hay en M_i con respecto a r_i.

Parte del código que calcula los grados es el siguiente:

```
for (modelo.ReglaDePreferencia pref : gui.getReglasDePrefrencia()) {
        for (modelo.Modelo mod : gui.getModelos()) {
            if (pref.tieneCuerpo()) {
                regla1(pref, mod);
            } else {
                int num = pref.getNumRegla();
                regla2(pref, mod);
            }
        }
    }
```

4.3.1 Código que ejecuta la regla 1 para obtener los grados de satisfacción

```
public int regla1(ReglaDePreferencia r, modelo.Modelo m) {
        List<ElementoCuerpoPref> cuerpo = r.getCuerpo();
        for (ElementoCuerpoPref elemento : cuerpo) {
            if (!m.contiene(elemento)) {
                gui.getTextGrados().append("degM" + m.getNumModelo() + "(r" +
r.getNumRegla() + ")= " + "1");
                gui.getTextGrados().append(System.getProperty("line.separator"));
                Grados gra = new Grados(m.getNumModelo(), r.getNumRegla(), 1);
                gui.getGrados().add(gra);
                return 1;
            }
        }
        return regla2(r, m);
    }
```

4.3.2 Código que ejecuta la regla 2 para obtener los grados de satisfacción

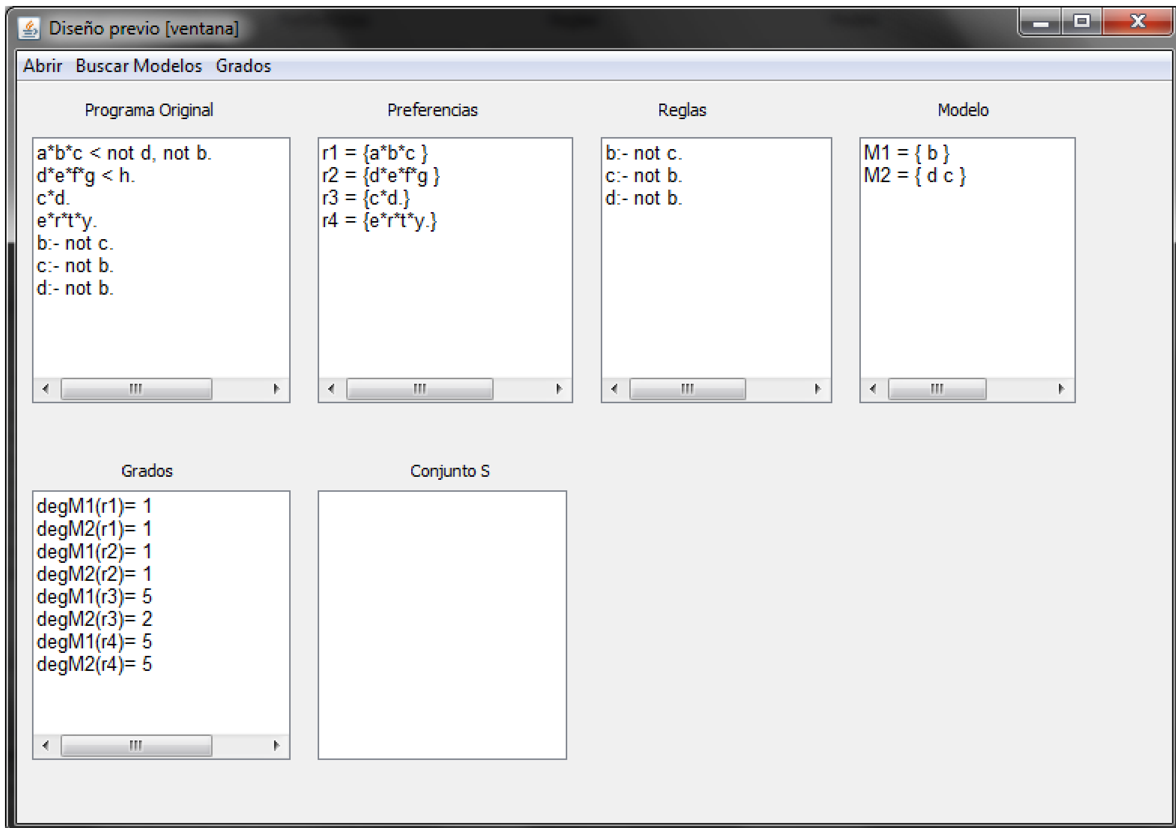
```
public int regla2(ReglaDePreferencia r, modelo.Modelo m) {
    List<Preferencia> cabeza = r.getCabeza();
    List<ElementoModelo> modelo = m.getElementos();
    for (int x = 0; x < modelo.size(); x++) {
        for (int y = 0; y < cabeza.size(); y++) {
            for (int i = 0; i < cabeza.size(); i++) {
                if (modelo.get(x).getValor().equals(cabeza.get(y).getValor())) {
                    int yy = y + 1;
                    gui.getTextGrados().append("degM" + m.getNumModelo() + "(r" +
r.getNumRegla() + ")= " + yy);
                   gui.getTextGrados().append(System.getProperty("line.separator"));
                   Grados gra = new Grados(m.getNumModelo(), r.getNumRegla(), yy);
                    gui.getGrados().add(gra);
                    return i;
                }
            }
        }
    }
    return regla3(r, m);
}
```

4.3.3 Código que ejecuta la regla 3 para obtener los grados de satisfacción

```
public int regla3(ReglaDePreferencia r, modelo.Modelo m) {
    int eme = gui.getM() + 1;
    gui.getTextGrados().append("degM" + m.getNumModelo() + "(r" + r.getNumRegla()
+ ")= " + eme);
    gui.getTextGrados().append(System.getProperty("line.separator"));
    Grados gra = new Grados(m.getNumModelo(), r.getNumRegla(), eme);
    gui.getGrados().add(gra);
```

```
    return 0;
}
```

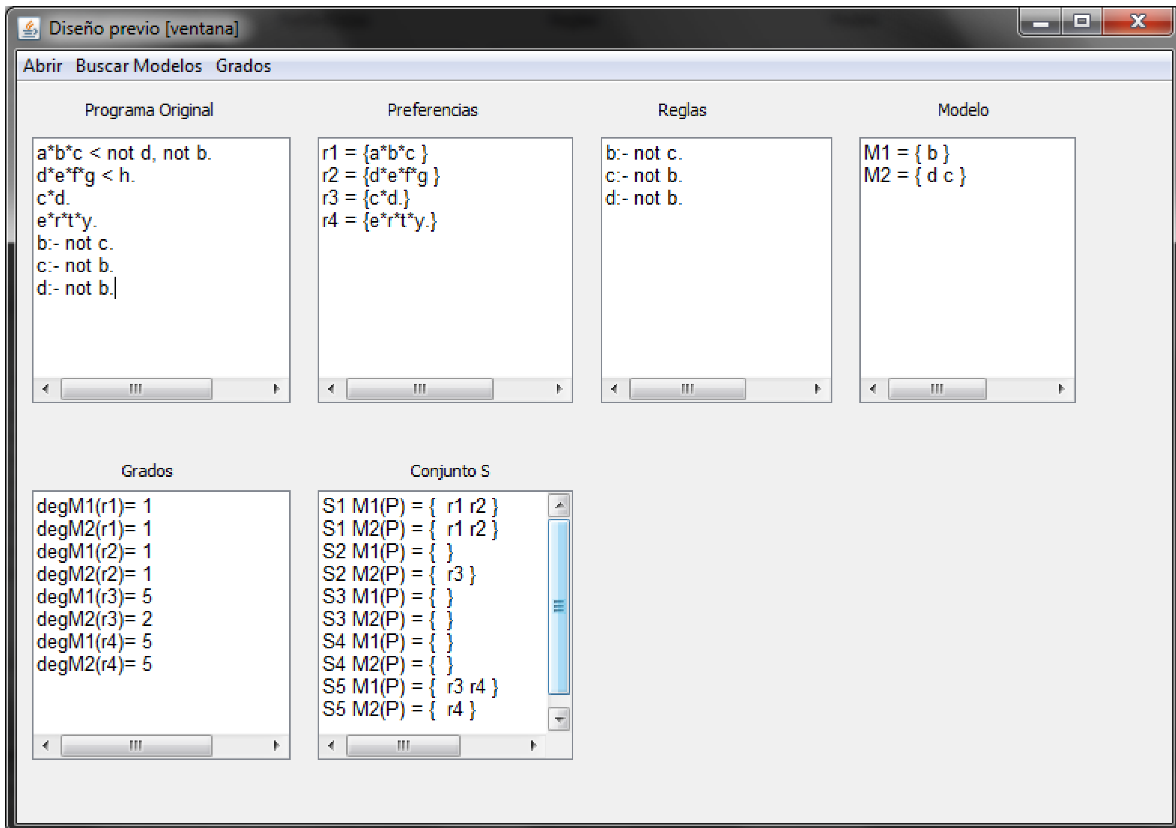
Posteriormente buscaremos al conjunto S, entonces agregamos otra área de texto para poder obtener lo que queremos, así que nuestra ventana quedaría de la siguiente forma:



4.4 Método para obtener el conjunto S

Al pulsar en el menú de Grados también se obtiene el conjunto S. La forma de obtener el Conjunto S se ha descrito en el capítulo 2, en la parte de diseño.

Entonces nuestro resultado sería el siguiente:



El cual nos indica qué reglas r_i de los grados de satisfacción pertenecen a Mi y tiene un mismo grado.

El código usado para obtener este conjunto es el siguiente:

Obtenemos el maximo de los grados

```
for (Grados g : gui.getGrados()) {
        if (maxG < g.getGrado()) {
            maxG = g.getGrado();
        }
    }
```

Obtenemos el maximo de los modelos

```
for (modelo.Modelo modelo : gui.getModelos()) {
        if (maxM < modelo.getNumModelo()) {
```

```
            maxM = modelo.getNumModelo();
        }
    }
```

Calculamos el conjunto S

```
    for (int x = 1; x < maxG + 1; x++) {
        for (int y = 1; y < maxM + 1; y++) {
            System.out.println("S" + x + " M" + y + "(P) = { " + buscaR(x, y) + " }");
            gui.getTextS().append("S" + x + " M" + y + "(P) = { " + buscaR(x, y) + " }");
            gui.getTextS().append(System.getProperty("line.separator"));
            ConjuntoS coS = new ConjuntoS(y, x, buscaR(x, y));
            gui.getConjuntoS().add(coS);
        }
    }
}

public String buscaR(int x, int y) {
    String rr = "";
    for (Grados gr : gui.getGrados()) {
        if (gr.getGrado() == x && gr.getNumMod() == y) {
            rr += " r" + gr.getNumPref();
        }
    }
    return rr;
}
```

Capítulo 5. Resultados

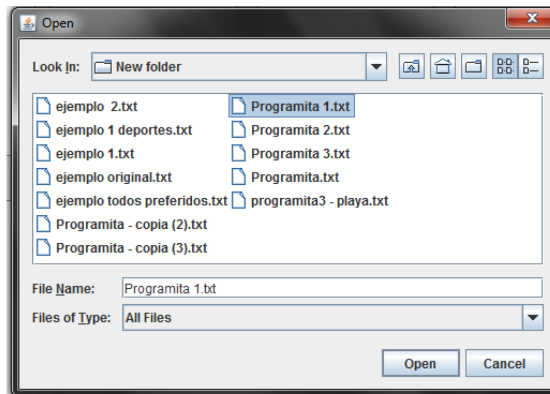
En este capítulo se darán a conocer los resultados que nos arroja este programa al procesar las reglas que están en nuestro archivo de texto. Se darán 3 ejemplos y con esto comprobaremos su funcionalidad.

5.1 Ejemplo 1

El primer programa que se analizara será el siguiente, el cual ya está en un archivo de texto:

```
a*b < d, b.
b:- not c.
```

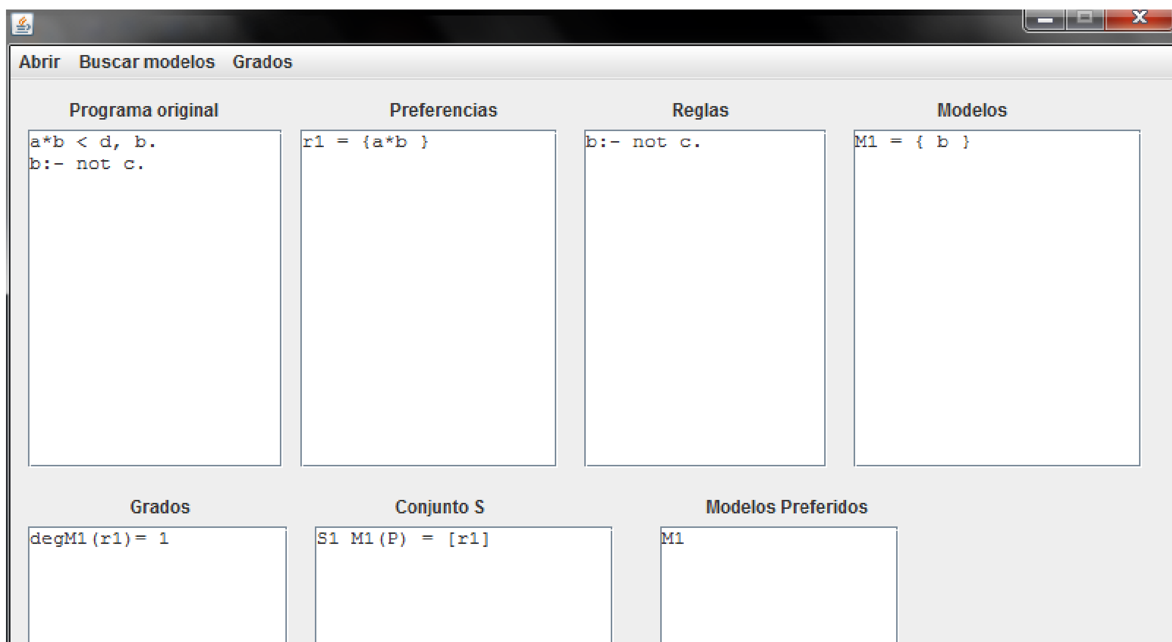
Como se ha descrito en capítulos anteriores, el primer paso será buscar nuestro archivo de texto que contenga las reglas que se van a analizar, esto se realiza haciendo clic en el menú Abrir.



Y posteriormente obtendremos lo siguiente en la ventana principal:

Ahora buscamos los modelos en el menú Buscar modelos eligiendo buscarlos con Smodels o Clasp, al término de esto obtendremos los modelos en la ventana principal:

Ahora resta calcular los grados haciendo clic en el menú Grados y eligiendo Calcular grados obteniendo así los grados, el conjunto S y el modelo preferido



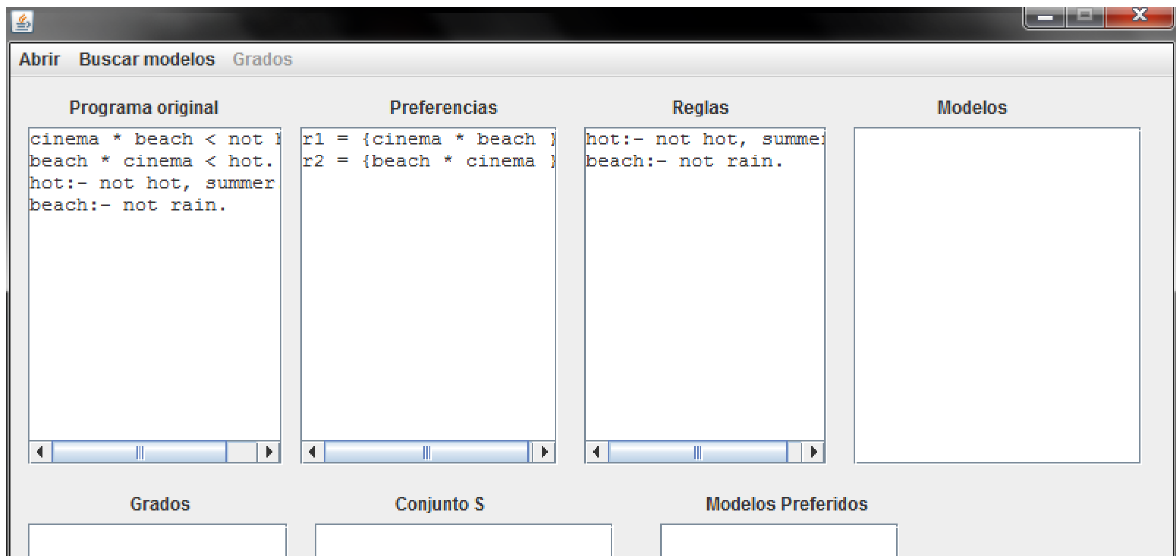
En este ejemplo sencillo se puede ver que el modelo preferido es el modelo M1 = {b}.

5.2 Ejemplo 2

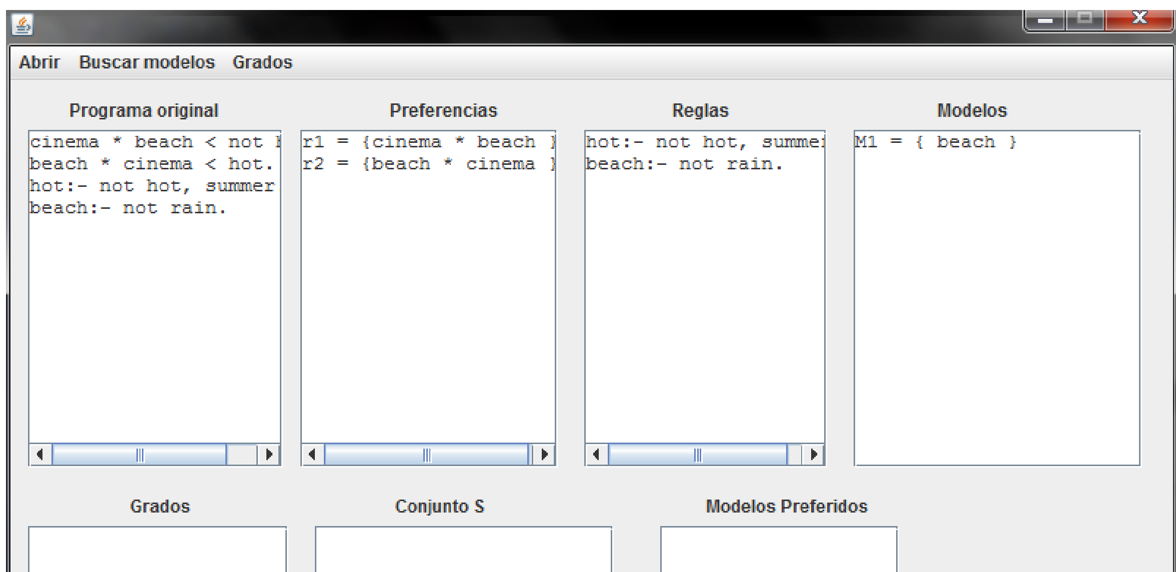
El siguiente conjunto de reglas que se analizara será el siguiente:

```
cinema * beach < not hot.
beach * cinema < hot.
hot:- not hot, summer.
beach:- not rain.
```

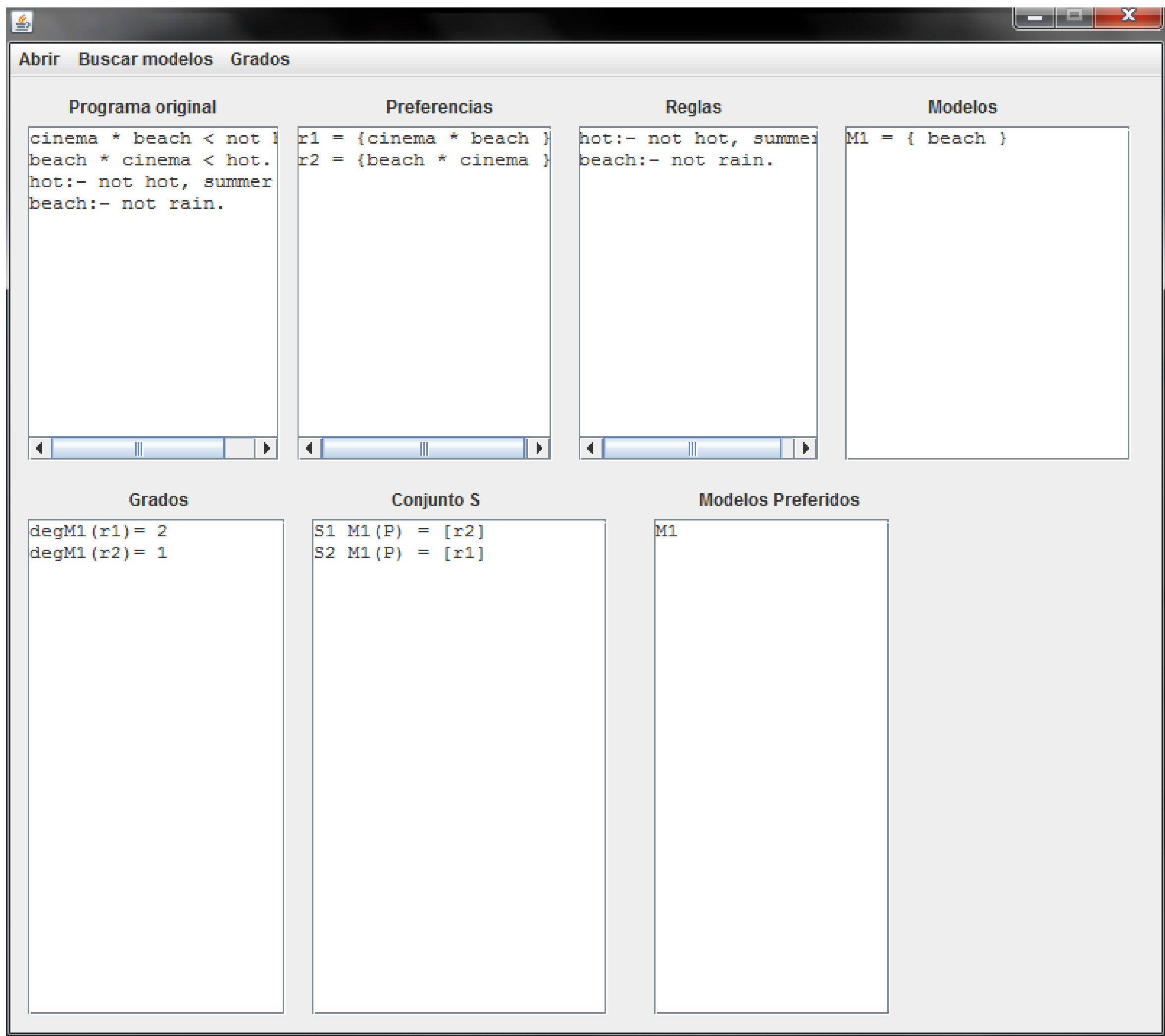
Entonces al abrir el archivo de texto tenemos lo siguiente:



Al buscar los modelos obtenemos:



Y al buscar los grados obtenemos los grados, el conjunto S y los modelos preferidos:



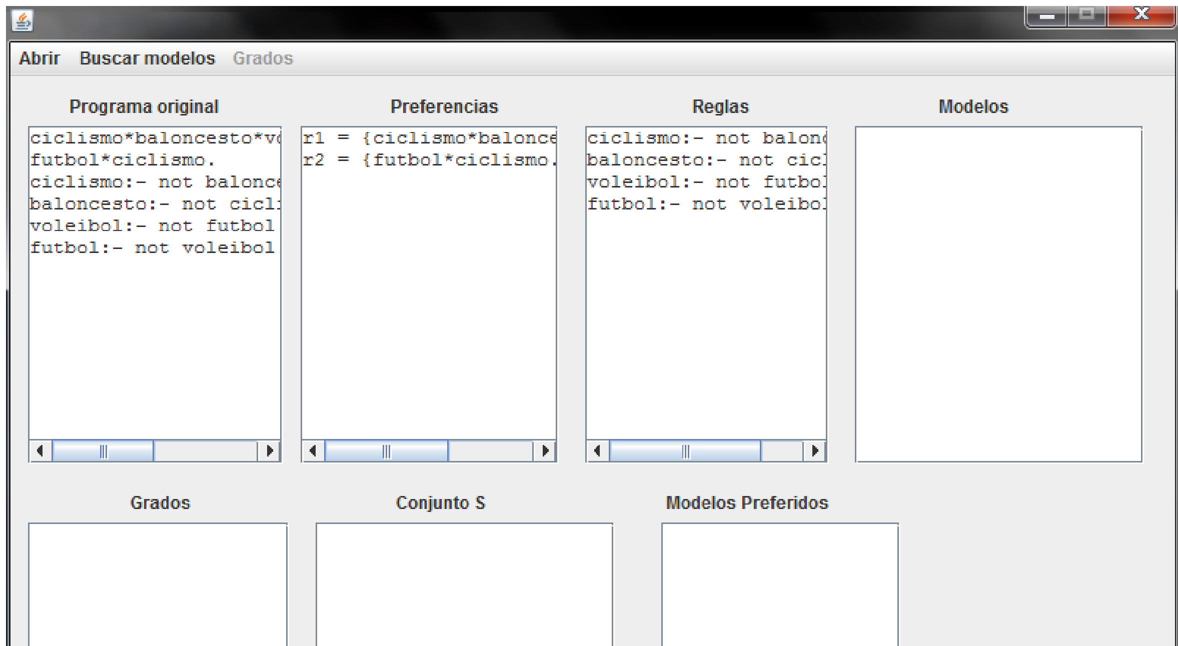
En este ejemplo se puede ver que el modelo preferido es M1 = {beach}

5.3 Ejemplo 3

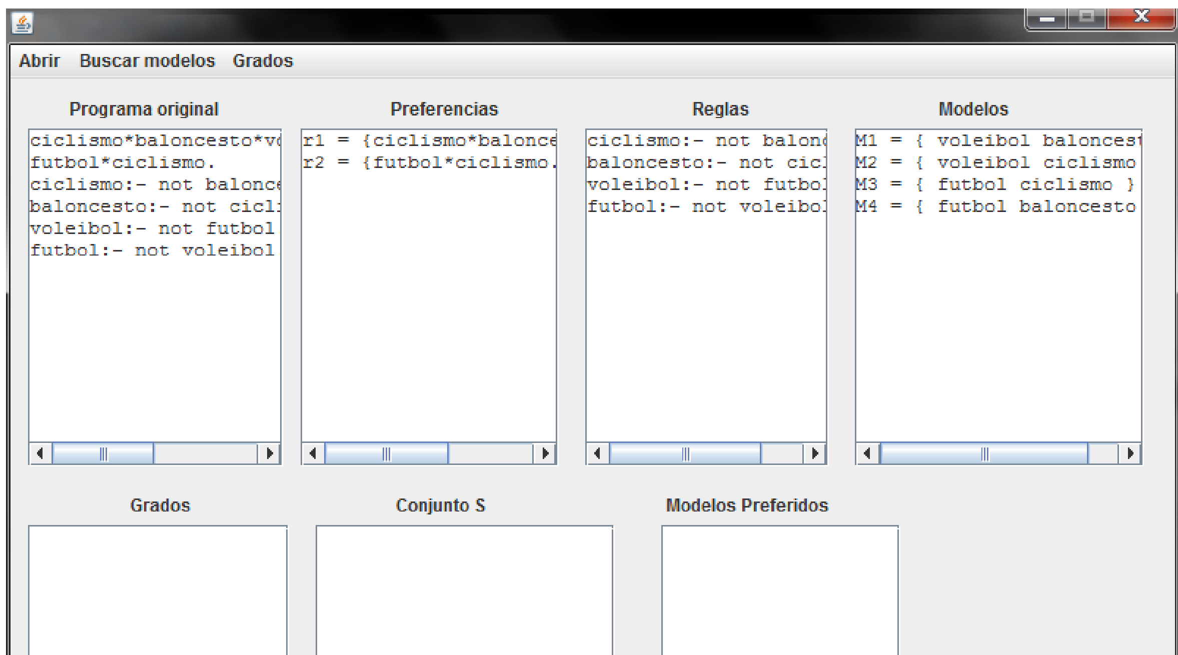
El conjunto de reglas que se va a analizar será el siguiente:

```
ciclismo*baloncesto*voleibol < not futbol.
futbol*ciclismo.
ciclismo:- not baloncesto.
baloncesto:- not ciclismo.
voleibol:- not futbol.
futbol:- not voleibol.
```

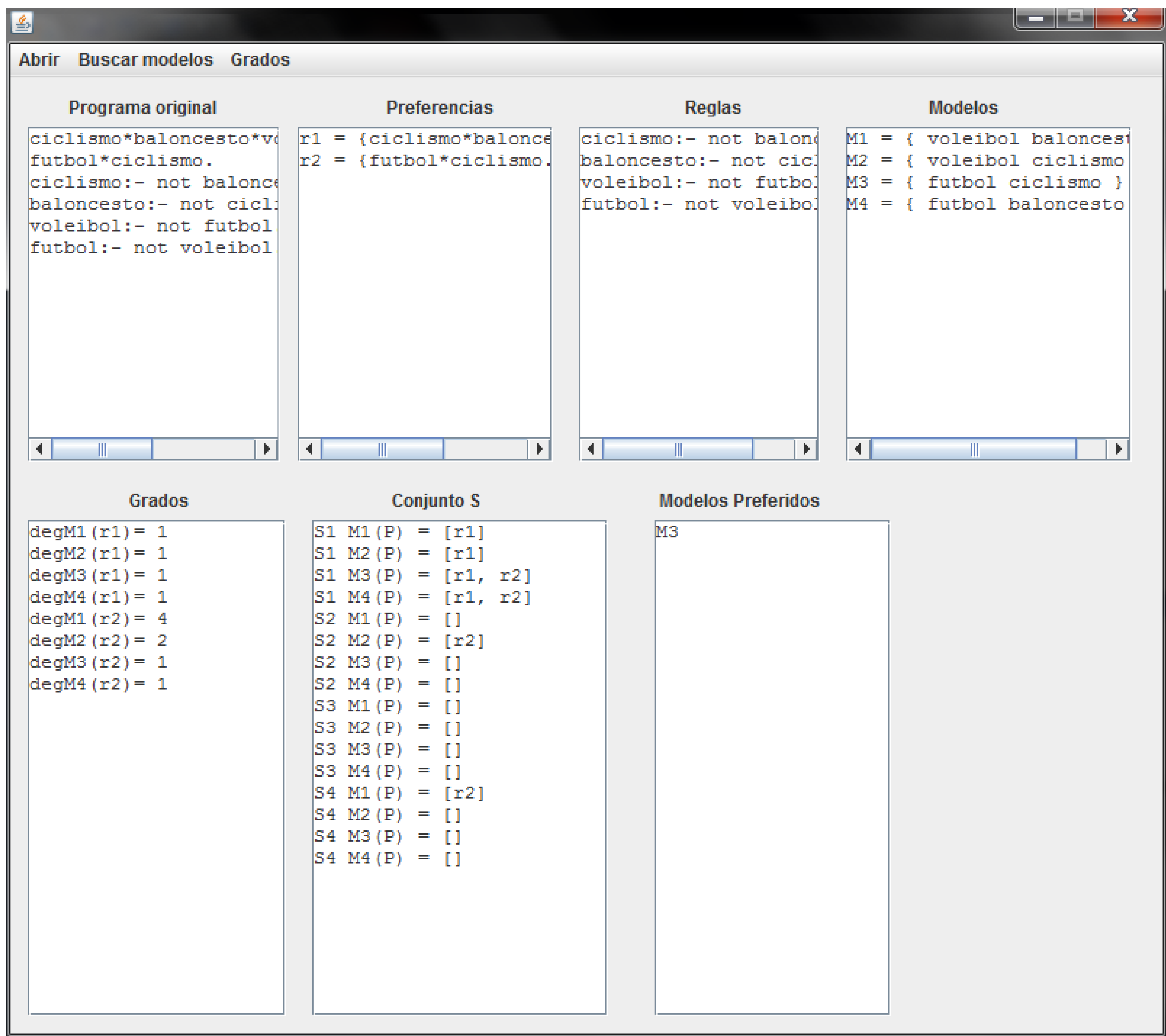
Al abrir el archivo de texto nos aparece lo siguiente en la pantalla principal:



Ahora buscamos los modelos:



Posteriormente buscamos los grados obteniendo así los modelos preferidos:



De esta manera podemos observar que de los 4 modelos que nos arroja Smodels o Clasp es preferido solo el modelo M3 = {futbol ciclismo}

Conclusiones

Durante el desarrollo de este trabajo se explico el enfoque considerado para escribir un programa lógico y así obtener las preferencias basadas en las reglas de nuestro programa. También se implementó un software para poder analizar las reglas de preferencia y así verificar que realmente los resultados que nos arroja este software son los resultados esperados si se hiciera a mano.

Uno de los problemas enfrentados al realizar este trabajo fue la comprensión de las reglas para poder analizar los programas lógicos, posteriormente la codificación también fue un obstáculo más ligero ya que la programación en java me ha sido un poco confusa pero con este trabajo he aprendido un poco más sobre este lenguaje.

Finalmente se hicieron pruebas con muchos ejemplos, desde ejemplos muy sencillos hasta un poco más complicados teniendo como resultados los modelos preferidos y en otras ocasiones ningún resultado ya que esas reglas de preferencia no generaban ningún resultado. En general el objetivo se cumplió al obtener los resultados esperados de la ejecución del software. Cabe destacar que con este software se verificaron programas con al menos 9 líneas de texto entre reglas de preferencias y reglas que no corresponden a preferencias. También se probó con el sistema operativo Linux teniendo los mismos resultados.

Probablemente una versión más completa se pueda desarrollar más adelante para analizar programas más complejos o con otro tipo de reglas, mientras tanto, este software funciona para las reglas establecidas.

Referencias

[1] J. Delgrande, T. Schaub, H. Tompits, and K. Wang. A classification and survey of preference handling approaches in monotonic reasoning. Computational Intelligence, 2004.

[2] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalsky and K. Bowen, editors, 5th Conference on Logic Programming, páginas 1070 – 14080. MIT Press, 1988.

[3] Ronen I. Brafman and Carmel Domshlak, Preference Handling – An Introductory Tutorial, Association for the Advancement of Artificial Intelligence, páginas 3-5, 2009.

[4] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving whit Answer Sets*. Cambridge University Press, Cambridge, 2003.

[5] Eiter T., Leone N., Mateis C., Pfeifer G., Scarcello F. A deductive system for nonmonotonic reasoning. *In Dix J., Furbach U., Nerode A., eds. Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer. 364–375, 1997.

[6] Gebser M., Liu L., Namasivayam G., Neumann A., Schaub T., and Truszczyński A. The First Answer Set Programming System Competition. Journal. Online http://cs.engr.uky.edu/ai/papers.dir/asp-contest.pdf (última fecha de verificación mayo 2012), páginas 15.

[7]http://www.ia.urjc.es/cms/sites/default/files/userfiles/file/ia3/teoria/Intro-ASP.pdf (última fecha de verificación mayo 2012)

[8] Ilkka Niemelä and Patrick Simmons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In Dix J., Furbach U., Nerode A.,

eds. Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning. Springer, 420–429, 1997.

[9] Ilkka Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence. 25:3-4, 241–273, 1999.

[10] Juan Antonio Navarro. *Lógica Aplicada a Answer Sets*. Tesis de Licenciatura. Universidad de las Américas Puebla, 2003. Online http://www.mpi-sws.org/~jnavarro/papers/uthesis.pdf (última fecha de verificación mayo 2012), páginas 61.

[11] Marek V., Truszczyński M. *Stable models and an alternative logic programming paradigm*. In Apt K., Marek W., Truszczyński M., Warren D., eds. The Logic Programming Paradigm: a 25-Year Perspective. Springer. 375–398, 1999.

[12] McCarthy J. Circumscription – a form of nonmonotonic reasoning. Artificial Intelligence. 13:1-2, 27–39, 1980.

[13] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors. 5th Conference on Logic Programming. MIT Press, pages 1070–1080, 1988.

[14] Reiter R. *A logic for default reasoning*. Artificial Intelligence 13:1-2, 81–132, 1980.

[15] Colmerauer A., Kanoui H., Pasero R., Roussel P. Un systeme de communication home machine en Francais. *Technical report, University of Marseille*, 1973.

[16] Kowalski R. Predicate logic as a programming language. In Rosenfeld J., ed. *Proceedings of the Congress of the International Federation for Information Processing*. North Holland. 569–574, 1974.

[17] http://sistemasumma.com/2010/11/16/logica-monotonica-y-no-monotonica/ (última fecha de verificación mayo 2012)

[20] José Pedro García Sabater, Julien Maheut, Grupo de Investigación ROGLE. Modelos y Métodos de Investigación de Operaciones. Procedimientos para Pensar. *(Modelado y Resolución de Problemas de Organización Industrial mediante Programación Matemática)*

[21] G. Brewka. Logic Programming with Ordered Disjunction. In *Proceedings of the 18th National Conference on Artificial Intelligence, AAAI-2002*. Morgan Kaufmann, 2002.

[22] M. Osorio, J. A. Navarro, and J. Arrazola. Applications of Intuitionistic Logic in Answer Set Programming. *Theory and Practice of Logic Programming (TPLP)* 4:325-354, May 2004.

[23] C. Zepeda. *Evacuation Planning using Answer Sets*. PhD thesis, Universidad de las Américas, Puebla and Institut National des Sciences Appliquees de Lyon, 2005.

[24] Mauricio Osorio, Claudia Zepeda. Preferences for General Theories in Answer Sets. Universidad de las Américas, CETIA, paginas 1-7, 2006.