JESS – the Rule Engine for the Java™ Platform

Francesca Volcan Pio[1], **Maiara Heil Cancian**[1], **Ricardo Bedin França**[1]

[1]Departamento de Automação e Sistemas - DAS
Universidade Federal de Santa Catarina - UFSC - Florianópolis, Brasil

1. Introdução

O presente trabalho tem como objetivo apresentar as possibilidades oferecidas pela ferramenta JESS - Java Expert System Shell, no desenvolvimento de sistemas especialistas, estudados no escopo da matéria DAS6607 - Inteligência Artificial Aplicada a Sistemas de Controle e Automação. Especificamente, deseja-se expôr os pontos interessantes do JESS em comparação com outras ferramentas (*shells*) de desenvolvimento para este tipo de sistema, bem como apresentar sua sintaxe de programação e seu uso em uma aplicação.

Este documento está estruturado da seguinte forma: a seção 2 expõe os conceitos básicos envolvidos na teoria de sistemas especialistas, a seção 3 apresenta um comparativo entre as *shells* disponíveis para o desenvolvimento desses sistemas e a seção 4 discute com mais detalhes o funcionamento do JESS. Para melhor entendimento destes conceitos, a seção 5 apresenta um exemplo de aplicação que utiliza um sistema especialista desenvolvido com o JESS. Por fim, as conclusões deste trabalho são expostas na seção 6.

2. Sistemas Especialistas

Quando é construído um sistema voltado para uma área muito específica, requerendo a experiência de um especialista humano, ele é denominado Sistema Especialista (SE). Este deve ser capaz de emitir uma decisão, apoiado em conhecimento justificado, a partir de uma base de informações, tal qual um especialista de determinada área do conhecimento humano [Russel and Norvig 1995]. Para emitir uma decisão sobre um determinado assunto, um especialista o faz a partir de fatos que encontra e de hipóteses que formula, buscando em sua memória um conhecimento prévio armazenado. Com esse processo de raciocínio pode-se não chegar a uma decisão se os fatos de que dispõe forem insuficientes, ou chegar a uma conclusão errada, sendo este erro justificado em função dos fatos que encontrou e do seu conhecimento acumulado previamente em sua memória de trabalho.

Um SE deve, além de inferir decisões, ter capacidade de adquirir novos conhecimentos e, desse modo, melhorar o seu desempenho de raciocínio. A arquitetura dos SEs apresentada na figura 1, deriva dos chamados Sistemas de Produção, que é um nome genérico para todos os sistemas baseados em regras de produção.

Estas regras de produção são pares de expressões consistindo em uma condição e uma ação [Bittencourt 1998]. A idéia inicial dos sistemas de produção foi introduzida por Post, em 1936, quando ele propôs os hoje chamados sistemas de Post [Post 1943]. Um sistema de Post consiste em um conjunto de regras para a especificação sintática de transformações sobre cadeias de caracteres, e representa um método geral para o processamento de dados.

Os sistemas de produção foram redescobertos durante os anos setenta como uma

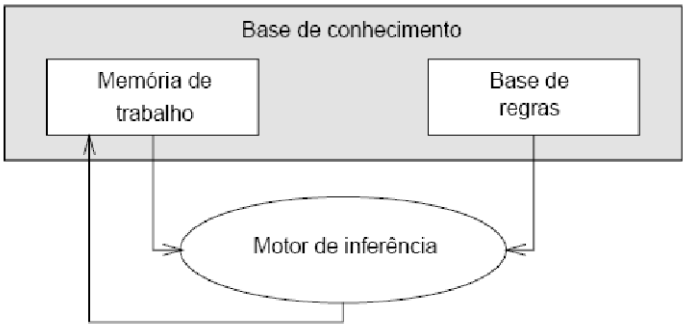

Figura 1. Arquitetura de um Sistema Especialista

ferramenta para a modelagem da psicologia humana. O formato condição-ação se adapta à modelagem de todos os comportamentos baseados em pares estímulo-resposta.

Pode-se portanto distinguir três componentes essenciais: uma base de regras, uma memória de trabalho e um motor de inferência. A base de regras em conjunto com a memória de trabalho formam a chamada base de conhecimento do SE. O motor de inferência é o mecanismo responsável por buscar na base de regras os padrões encontrados na memória de trabalho e executar a ação apropriada. Para solucionar problemas, os SEs precisam acessar uma grande base de conhecimento do domínio da aplicação, portanto o sucesso de um SE depende enormemente da forma como o conhecimento é representado e dos mecanismos para a exploração deste conhecimento.

A seção seguinte tratará especificamente sobre distintos métodos de representação de conhecimento. Portanto, a seguir será dada ênfase à descrição das principais características de um motor de inferência, que vem a ser o mecanismo de exploração do conhecimento propriamente dito.

2.1. Motor de inferência

Classifica-se um motor de inferência pelas funcionalidades que este deve possuir: modo de raciocínio, estratégia de busca, resolução de conflito e representação de incerteza.

2.1.1. Modo de raciocínio

Basicamente, existem dois modos de raciocínio aplicáveis a regras de produção: encadeamento progressivo (*forward chaining*), e encadeamento regressivo (*backward chaining*). No encadeamento progressivo, também chamado encadeamento dirigido por dados, a parte esquerda da regra é comparada com a descrição da situação atual, contida na memória de trabalho. As regras que satisfazem a esta descrição têm sua parte direita executada, o que, em geral, significa a introdução de novos fatos na memória de trabalho. A diferença, no encadeamento regressivo, conhecido também como encadeamento dirigido por objetivos, o comportamento do sistema é controlado por uma lista de objetivos. Um objetivo pode ser satisfeito diretamente por um elemento da memória de trabalho, ou pela existência de regras que permitem a inferência de tal objetivo, contendo uma descrição deste em suas partes direitas. As regras que satisfazem esta condição têm as instâncias correspondentes às suas partes esquerdas adicionadas à lista de objetivos correntes. Caso

uma dessas regras tenha todas as suas condições satisfeitas diretamente pela memória de trabalho, o objetivo em sua parte direita é também adicionado à memória de trabalho. Um objetivo que não possa ser satisfeito diretamente pela memória de trabalho, nem inferido através de uma regra, é abandonado. Quando o objetivo inicial é satisfeito, ou não há mais objetivos, é cessado o processamento.

2.1.2. Estratégia de busca

Uma vez definido o tipo de encadamento, o motor de inferência necessita ainda de uma estratégia de busca para guiar a pesquisa na memória de trabalho e na base de regras. Este problema é conhecido como busca em espaço de estados. Este tópico é estudado em inteligência artificial, no contexto de solução de problemas. Descrições detalhadas dos algoritmos de busca podem ser encontradas em [Charniak and McDermott 1985].

2.1.3. Resolução de conflito

Após o processo de busca, o motor de inferência dispõe de um conjunto de regras que satisfazem à situação atual do problema, o chamado conjunto de conflito. Se esse conjunto for vazio, a execução é terminada; caso contrário, é necessário escolher as regras que serão executadas, e a ordem. Os métodos de resolução de conflito mais utilizados ordenam as regras de acordo com critérios como prioridades estáticas, complexidade, tempo decorrido desde a obtenção da regra, confiabilidade, grau de importância, e até seleção randômica. A utilização de um critério pode se tornar insuficiente para resolver os conflitos. Portanto, um SE pode combinar mais de um método, e em geral, é desejável que o usuário possa especificar quais métodos deverão ser usados.

2.1.4. Representação de incerteza

O tratamento de incerteza é uma ativa área de pesquisa em SEs, pois os domínios adequados à implementação de SEs se caracterizam exatamente por não serem modelados por nenhuma teoria geral, o que implica descrições incompletas, inexatas ou incertas. Diversos métodos foram propostos para tratar este problema, método bayesiano, fatores de certeza (conforme o modelo adotado no MYCIN), teoria de Dempster-Shafer, teoria dos conjuntos nebulosos, teoria de probabilidades subjetivas e teoria de possibilidades.

Para quantificar de alguma forma a confiança do especialista, estes métodos atribuem aos fatos e regras uma medida numérica. Os métodos utilizados não mantém entre si nenhuma coerência, e cada um adapta-se melhor a determinados tipos de problemas. Também aqui é desejável que o usuário possa ter a escolha do método mais adequado para seu problema, e de limites mínimos para a medida de incerteza, para truncar fatos ou regras que se encontram abaixo do limiar.

3. Comparativos entre *Shells*

3.1. WebLS

O WebLS é um *shell* que utiliza uma máquina de inferência PROLOG para fornecer diagnósticos de suporte técnico. Sua comunicação com servidores *web* é feita utilizando

CGI (Common Gateway Interface), e ele é capaz de gerar HTML dinamicamente para fornecer soluções ao usuário, facilitando a exibição de *links* para sítios *web* e arquivos.

O WebLS utiliza uma linguagem proprietária e modular, com algumas funções de suporte escritas em C. Ao contrário de outros sistemas especialistas, o WebLS foi criado com a intenção de não necessitar de um engenheiro do conhecimento para a construção de aplicações, podendo ser utilizado por especialistas do domínio, já que sua linguagem é bastante simples.

3.2. CLIPS

Lançado em 1986, o CLIPS é um *shell* bastante conhecido e durante a década de 90 foi o mais utilizado. Dada sua longa existência, ele é caracterizado por ser bastante estável, confiável e possuir uma documentação de alta qualidade. Programado em C, o CLIPS também é caracterizado por sua portabilidade, e pode ser integrado com linguagens como C/C++ e Ada através de chamadas procedurais. Seu desempenho é garantido pelo algoritmo RETE utilizado no motor de inferência.

A linguagem utilizada na construção de SEs com o CLIPS é bastante similar ao LISP, e a gama de aplicações onde é possível utilizar este *shell* é bastante ampla.

3.3. WebCLIPS

O WebCLIPS é uma implementação do CLIPS como uma aplicação CGI, buscando facilitar a integração do CLIPS com aplicações *web* feitas em HTML. Assim como o WebLS, o WebCLIPS tem a capacidade de gerar páginas e formulários HTML dinamicamente, e de modo análogo ao CLIPS, o WebCLIPS foi programado em C e possui uma sintaxe bastante similar ao LISP para a criação de SEs.

3.4. JESS

O JESS, que será visto mais detalhadamente nas seções 4 e 5, busca facilitar a integração de sistemas especialistas em aplicações Java, dada a crescente utilização desta linguagem, especialmente em aplicações *web*. A construção de SEs com JESS é feita com uma sintaxe bastante similar à do CLIPS, porém a integração do CLIPS com C é substituída pelo uso do Java, o que pode ocasionar dificuldades na migração de aplicações de um para outro *shell*. O JESS possui a desvantagem inerente de aplicações Java de ser pesado em termos de execução, o que não chega a ser um problema quando executado em estações preparadas para suportar aplicações Java.

4. JESS – Java Expert System Shell

O JESS foi desenvolvido por Ernest Friedmal-Hill no Sandia National Labs como parte de um projeto interno de pesquisa. A primeira versão do JESS foi escrita em 1995. É um *script shell* para construção de sistemas especialistas baseado em regras inteiramente desenvolvido em Java da Sun Microsystems e possui um ambiente similar a linguagem de programação CLIPS (C Language Integrated Production System) desenvolvida pela NASA.

O JESS utiliza o algoritmo RETE na sua máquina de inferência. Este algoritmo é caracterizado por organizar as regras numa árvore, de modo que sejam divididas em padrões, assim, regras semelhantes percorrerão o mesmo ramo da árvore enquanto

Ferra-menta	Tipo /Uso	Linguagem Desenvolvi-mento	Qualidade da Documen-tação	Usuário Alvo	Licença
JESS	Applets, servlets, aplicações	JESS *language*	Boa	Engenheiro do Conhecimento ou de Software	450 US $ Free se pesquisa
CLIPS	Biblioteca de funções	CLIPS (parecida com LISP)	Ótima	Eng. de Conhecimento ou Software	Free
WebCLIPS	CGI	CLIPS + targets HTML	Média	Eng. Conhecimento ou Software	GNU
WebLS	CGI	Própria (modular)	Ótima	Eng. conhecimento ou especialista do domínio	Free

Figura 2. Comparação entre Shells de SEs

possuírem cláusulas iguais. Tal organização é mais onerosa em termos de memória, mas proporciona grandes melhorias na velocidade de verificação das regras. O motor de inferência JESS permite tanto o encadeamento regressivo quanto progressivo.

A linguagem Jess é uma linguagem de programação de propósito geral e pode acessar diretamente todas as classes e bibliotecas Java. Por esta razão, Jess é muito usado como ambiente de desenvolvimento rápido de aplicações e scripts. Enquanto o código Java tem que ser compilado antes de rodar, uma linha de código do Jess é executada imediatamente após ser digitado. Isto permite que os usuários experimentem as API´s Java interativamente e desenvolvam programas incrementalmente.

4.1. Comandos Principais do Jess

Abaixo alguns comandos principais do Jess, para inicialização, execução e programação:

- iniciar o Jess: no prompt do MS-DOS, diretório Jess60, digitar o seguinte comando: java -classpath jess.jar jess.Main;
- ler um script: (batch NomePrograma.clp);
- carregar na memória: (reset);
- executar as regras: (run);
- comentários: símbolo de ponto e vírgula “;;” ;
- encerrar o Jess: (exit).

4.1.1. Átomos

O átomo é a unidade básica de representação no Jess. Ele pode conter letras, números, caracteres como, $, *, =, +, /, ¡, ¿, _,?, # e valores booleanos, como `TRUE` e `FALSE`. O Jess é case sensitive, ou seja, a digitação da mesma palavra em minúsculo e em maiúsculo tem significados diferentes.

4.1.2. Números e Textos (Strings)

O Jess permite ponto flutuante e números inteiros, como outras linguagens de programação. Os textos devem ser sempre representados por aspas duplas, como por exemplo:

```
''Exemplo de string''
```

4.1.3. Listas

Lista é uma coleção de um ou mais átomos, textos, números ou listas entre parênteses. A seguir, a sintaxe utilizada no Jess: `Jess> (+ 1 2)`
`Jess> 3`

4.1.4. Variáveis

As variáveis devem ser representadas com o ponto de interrogação (?) e não devem ser declaradas antes de serem utilizadas. Utiliza-se o comando *bind* para associar alguma um valor a uma variável, como representado abaixo: `;;A palavra Teste será armazenada na variável criada de nome x`
`Jess> (bind ?x ''Teste'')`

4.1.5. Fatos

Pode-se adicionar fatos na base de conhecimento, usando o comando *assert*. Fatos podem ser excluídos através do comando *retract*, conforme mostrado abaixo:

```
     ;;inclui os fatos na base de conhecimento
(assert (Pessoa (idade 25) (sexo feminino)))
;; remove um fato da base de conhecimento
(retract ?perfil)
```

4.1.6. Templates

Templates podem ser comparadas com estruturas de classe em linguagem C++ ou Java, por exemplo. Para a definição de um *template* utiliza-se o comando *deftemplate*. Os campos utilizados dentro do *template* são chamados de *slots*, e são declarados como a seguir:

```
     (deftemplate usuariorenda
(slot estadocivil)
(slot dependentes)
(slot divida)
)
```

4.1.7. Regras

Regras são responsáveis por verificar ações, baseando-se nos fatos da base de conhecimento. As regras em geral, podem ser comparadas com o "if-else" da linguagem procedural. Utiliza-se o comando defrule para definir uma regra conforme são satisfeitas as condições do lado esquerdo da regra, são disparadas as instruções à direita:

```
     ;;definindo a regra de nome Regra1 através do comando
defrule
(defrule Regra1
(estadousuario (estadocivil casado))
=>
(assert (renda comprometida))
)
```

4.1.8. Funções

No Jess pode-se declarar funções. Utiliza-se o comando *deffunction* para definir uma função e o nome da própria função entre parênteses para fazer a chamada (nomefunção), conforme a sintaxe a seguir:

```
     ;; declarando uma função através do comando
deffunction
(deffunction nomefunção ()
......
(assert (Perfil (x ?x) (economico TRUE)))
)
```

5. Aplicação - Sistema de Gerenciamento de Ofertas

5.1. Motivação

O mercado atual exige que as empresas criem constantemente promoções e ofertas especiais para obter algum diferencial em relação às outras, de modo que mais clientes sejam atraídos. Neste cenário, é importante o uso de um mecanismo eficiente e automatizado para especificação e cálculo de descontos, especialmente quando o número de produtos e de possibilidades de desconto é alto.

Tendo em mente que o domínio deste problema – as ofertas a serem gerenciadas – é bem conhecido pela empresa e pode ser visto como uma série de regras, a utilização de um sistema especialista surge como opção. Em comparação com linguagens tradicionais de programação, o armazenamento do "conhecimento" do sistema somente em estruturas de dados apresenta uma importante vantagem sobre o armazenamento de parte das informações na estrutura do programa: a facilidade na modificação e atualização deste conhecimento, sem a necessidade de recompilação do programa. Para esta aplicação, tal característica é desejável, visto que as ofertas são atualizadas regularmente. A utilização do JESS para este exemplo demonstra um dos pontos favoráveis a esta ferramenta em especial: a possibilidade de integração de um sistema especialista num programa comum, feito em Java. Assim, o sistema especialista comunica-se facilmente com outras partes da aplicação.

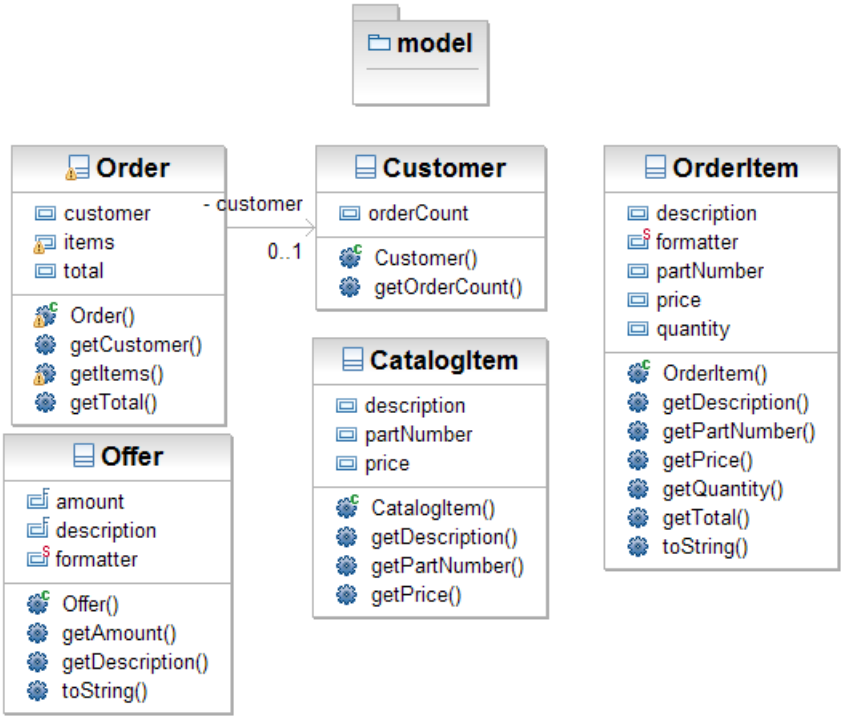

Figura 3. O pacote *model*

5.2. Estrutura da Aplicação

A aplicação onde deseja-se incluir o sistema especialista é composta por três pacotes (*packages*): o pacote *model* contém as classes que definem as estruturas de dados utilizadas na aplicação, o pacote *pricing* contém a classe que instancia o motor de inferência do sistema especialista e o pacote *demo* contém uma pequena base de dados e a função *main()*.

No pacote *model*, visto na figura 3, estão presentes as seguintes classes:

CatalogItem Descreve um item presente no catálogo da empresa.
Customer Representa o cliente que efetua a compra, armazenando o número de compras já efetuadas por este.
Offer Descreve um desconto ou oferta a ser incluída numa compra.
Order Descreve uma compra, com os itens comprados, suas quantidades e preços.
OrderItem Representa um item presente numa compra.

O pacote *pricing* (figura 4) contém a classe *PricingEngine*, que tem entre seus atributos o motor de inferência do JESS e uma interface de banco de dados. Visto que o foco do trabalho está no sistema especialista, o banco de dados foi mantido no próprio código Java, sem a necessidade de implementá-lo numa linguagem específica de bancos de dados. O pacote *demo*, também visto na figura 4, contém a implementação do banco de dados na classe *DemoDatabase* e a classe *Demo*, que contém a função *main()* que instancia os objetos necessários para a execução do programa.

5.3. O Sistema Especialista

Para que a aplicação em Java tenha acesso às classes e métodos do JESS, é necessário importá-las (`import jess.*`). Conforme pode-se ver nas linhas 6-11 da figura 5, a

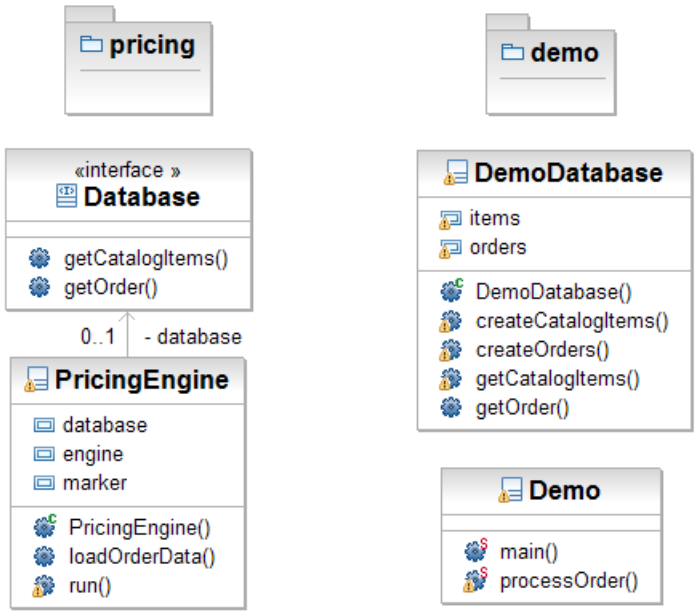

Figura 4. Os pacotes *pricing* e *demo*

classe *PricingEngine* cria um motor de inferência instanciando um objeto da classe Rete, e configura seu uso inicializando-o com o método *reset()*, para em seguida executar o arquivo JESS onde encontram-se as regras, adicionar os itens do banco de dados como *facts* na memória de trabalho do sistema especialista, e criar um *marker* para armazenar o estado atual da memória de trabalho. Isto é útil quando se deseja reiniciar a execução do SE sem carregar novamente o arquivo JESS.

O método criado para executar o SE em uma compra pode ser visto nas linhas 15-20: ele reinicia o sistema para o estado armazenado em sua criação, isto é, com o banco de dados mas sem registros de compras, em seguida carrega os dados da compra solicitada e executa o sistema de regras para verificar quais ofertas podem ser geradas para a compra solicitada. Estas ofertas são retornadas através do uso de um filtro que busca apenas objetos da classe *Offer*.

Para que as estruturas de dados criadas na aplicação Java sejam traduzidas para *templates* JESS, o arquivo JESS deve importar as classes Java, bem como conter as declarações destes *templates*, conforme pode-se ver nas linhas 1-5 da figura 6. Deve-se notar que não foi necessário definir um *template* para a classe *Offer* pois o motor de inferência não utiliza nenhum objeto desta classe para disparar as regras.

Com os *templates* devidamente declarados, pode-se especificar as regras do sistema. As linhas 7-11 da figura 6 mostram, como exemplo, a especificação de uma regra baseada no preço total de uma compra. No caso da regra ser disparada, um objeto da classe *Offer* é adicionado à memória de trabalho, com a execução do método Java *add()*.

A execução deste programa em uma compra fictícia com o custo superior a $ 100 dispara a regra descrita acima, e mostra ao usuário o desconto aplicado:

```
Items for order 123:
   1 CD Writer: 199,99
```

```
1  public class PricingEngine {
2      private Rete engine;
3      private WorkingMemoryMarker marker;
4      private Database database;
5      public PricingEngine(Database aDatabase) throws JessException {
6          engine = new Rete();
7          engine.reset();
8          engine.batch("pricing.clp");
9          database = aDatabase;
10         engine.addAll(database.getCatalogItems());
11         marker = engine.mark();
12     }
13
14     }
15     public Iterator run(int orderNumber) throws JessException {
16         engine.resetToMark(marker);
17         loadOrderData(orderNumber);
18         engine.run();
19         return engine.getObjects(new Filter.ByClass(Offer.class));
20     }
21 }
```

Figura 5. A utilização do motor de inferência JESS

```
1  (import gov.sandia.jess.example.pricing.model.*)
2  (deftemplate Order       (declare (from-class Order)))
3  (deftemplate OrderItem   (declare (from-class OrderItem)))
4  (deftemplate CatalogItem (declare (from-class CatalogItem)))
5  (deftemplate Customer    (declare (from-class Customer)))
6
7  (defrule 10%-volume-discount
8      "Give a 10% discount to everybody who spends more than $100."
9      (Order {total > 100})
10     =>
11     (add (new Offer "10% volume discount" (/ ?total 10))))
```

Figura 6. O arquivo JESS utilizado

```
   2 AA Batteries: 9,98
Offers for order 123:
   10% volume discount: -$21,00
```

6. Conclusão

Este trabalho apresentou a ferramenta JESS para desenvolvimento de sistemas especialistas, comparando-a com outras ferramentas disponíveis, expondo sua sintaxe para construção de tais sistemas, e um exemplo do uso do JESS no desenvolvimento de tais aplicações. Tanto o comparativo quanto o exemplo mostram claramente que o uso do JESS é bastante recomendado quando deseja-se embarcar um sistema especialista em uma aplicação Java, e, sabendo que há uma grande expansão no uso de Java para aplicações voltadas à *Web*, o JESS torna-se mais recomendável que outros *shells* disponíveis.

Referências

Bittencourt, G. (1998). *Inteligência Artificial: Ferramentas e Teorias*. Editora da UFSC, Florianópolis.

Charniak, E. and McDermott, D. (1985). *Introduction to artificial intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Chun, I.-G. and Hong, I.-S. (2001). The implementation of knowledge-based recommender system for electronic commerce using java expert system library. In *IEEE International Symposium on Industrial Electronics*.

Deitel, H. M. and Deitel, P. J. (2001). *Java How to Program*. Prentice Hall.

Friedman-Hill, E. Jess, the expert system shell for the Java platform. `http://herzberg.ca.sandia.gov/jess/`.

Post, E. (1943). Formal reductions of the general combinatorial problem. In *American Journal of Mathematics*, pages 197–268.

Riley, G. What is CLIPS? `http://www.ghg.net/clips/CLIPS.html`.

Russel, S. and Norvig, P. (1995). *Artificial Intelligence, a Modern Approach*. Prentice Hall.

Sehmi, A. and Kroening, M. WebLS: A Custom Prolog Rule Engine for Providing Web-Based Tech Support. `http://www.clip.dia.fi.upm.es /miscdocs/lp-internet/amzi/lspap.html`.