1. Ambiente CLIPS

1.1 Introdução a Sistemas Especialistas

O CLIPS (C Language Integration Production System) é um ambiente desenvolvido na década de 80 para construção de sistemas especialistas baseados em regras. Um sistema especialista baseado em regras objetiva reproduzir o comportamento de um especialista humano em um determinado contexto através de uma base de conhecimento construída a partir de fatos e regras. A arquitetura de um especialista baseado em regras é análoga a um sistema de produção, onde existe um conjunto de regras de produção, uma memória de trabalho e o ciclo se-então [LUGER,2004].

Uma regra de produção é um par condição-ação que define uma parte do conhecimento necessário para a solução do problema. Deste modo, a condição da regra (ou LHS – left hand side) determina quando a regra pode ser aplicada para uma situação do problema e a ação (ou RHS – right hand side) define o próximo passo na busca da solução para o problema.

Por sua vez, a memória de trabalho contém uma descrição do estado atual do problema em uma instância do processo de inferência. Esta descrição é um padrão que é comparado coma condição de uma produção para selecionar as ações apropriadas para resolução do problema. Quando o elemento da condição de uma regra combina com o conteúdo da memória de trabalho, a ação associada com esta condição pode ser executada.

Por fim, o ciclo se-então é executado quando é disparada a ação de uma regra. Uma ação pode modificar o estado da memória de trabalho, o que irá fazer com que todo o processo de inferência se repita. O processo irá terminar quando o conteúdo da memória de trabalho não combinar com nenhuma condição da regra.

1.2 Ambiente CLIPS

O mecanismo de inferência do CLIPS utiliza um algoritmo de casamento de padrões com encadeamento para frente (forward chaining). Esta estratégia consiste em selecionar as regras que se aplicam aos fatos existentes na base, montando uma lista de regras aplicáveis no contexto atual. Se a lista de regras aplicáveis estiver vazia, o processo é encerrado, caso contrário o algoritmo se encarregará de escolher a regra a ser aplicada. Após ser aplicada a regra escolhida, será analisado se esta ação implicou em alguma mudança nas premissas das regras contidas na lista de regras aplicáveis, como por exemplo, a adição de um fato. Se houver mudança na base de conhecimento, será procedido um novo encadeamento, caso contrário será escolhida outra regra contida na lista de regras aplicáveis para executar sua ação. Além desta estratégia, o CLIPS implementa o algoritmo RETE para indexação dos fatos e regras, objetivando a minimização do custo computacional das inferências na base de conhecimento.

1.3 Sintaxe do CLIPS

O ambiente CLIPS possui diversos meios para representação do conhecimento. É possível representar o conhecimento no CLIPS através de fatos simples ou dinâmicos, templates (fatos complexos, sendo uma composição de informações), regras ou produções e objetos ou instâncias. A sintaxe para representação destas estruturas no CLIPS é semelhante à gramática da linguagem LISP, isto é, todas as operações derivam de uma única estrutura chamada lista. Para facilitar o estudo da sintaxe, foi utilizada uma shell do CLIPS chamada CLIPSWin, onde é possível facilmente criar uma base de conhecimento e realizar inferências. Para descrever a sintaxe e comandos do CLIPS, este documento foi formatado da seguinte forma:

- Comandos CLIPS em geral: fonte Courier;
- Funções nativas do CLIPS: negrito e itálico; e
- Regras: negrito.

1.3.1 Adição de fatos

Para adicionar fatos na base de conhecimento, é utilizada a função ***assert***.

(***assert*** (<nome do fato> <valor>))

Exemplo:

(***assert*** (cor verde))

Para listar os fatos armazenados na base pode ser utilizada a função ***facts***. Cada fato adicionado é referenciado no CLIPS por um identificador inteiro, positivo, auto-incremental e exclusivo. Executando a função ***facts***, é possível observar que o identificador do fato (cor verde) adicionado é igual a 0. Caso seja adicionado outro fato qualquer, o seu identificador será 1. Este identificador pode ser utilizado para remover o fato da base de conhecimento, através da função ***retract***.

(***retract*** <ID do fato>)

Exemplo:

(***retract*** 0)

Como o identificador dos fatos é um valor inteiro positivo auto-incremental, ao remover o fato 0, por exemplo, este índice jamais será utilizado novamente.

1.3.2 Adição de regras

Para adicionar regras (produções) na base de conhecimento, é utilizada a função ***defrule***.

```
(defrule <nome da regra>
  <condição>
  =>
  <ação>)
```

Exemplo:

```
(defrule limao
   (cor verde)
   =>
   (assert (fruta limao)))
```

Para executar a regra acima, isto é, realizar a inferência na base de conhecimentos, deve ser utilizada a função ***run***.

É possível definir mais de uma pré-condição para uma regra. Para combinar pré-condições de modo similar ao operador lógico `AND`, basta adicionar mais de uma expressão que automaticamente serão conectadas com este operador. Para simular o operador lógico `OR`, deve ser utilizada função ***or***.

Exemplos:

```
(defrule limao
   (cor verde)
   (bebida citrica)
   =>
   (assert (fruta limao)))
(defrule clima
   (or (clima chuvoso)
       (clima nublado))
   =>
   (assert (levar guarda-chuva)))
```

1.3.3 Wildcards

Os exemplos acima ilustram a construção de regras estáticas. Contudo, o CLIPS permite a construção de regras dinâmicas através do operador `?`, chamado *wildcard*. A utilização de *wildcards* tornam as regras de produção mais flexíveis, permitindo que sejam combinados parcialmente os fatos contidos na pré-condição. Exemplo:

```
(defrule listar-fruta
   (fruta ?)
   =>
   (printout t "Fruta encontrada" crlf))
```

No exemplo acima, apenas o elemento `fruta` do fato `(fruta limao)` foi combinado. Porém, uma limitação do *wildcard* `?` é não permitir seu uso no primeiro elemento do fato (nome do fato).

O CLIPS permite também definição de variáveis nas regras de produção. Para definir variáveis é necessário utilizar o *wildcard* `?` precedido pelo nome da variável.

```
(<nome do fato> ?<nome variável>)
```

Exemplos:

```
(defrule listar-fruta
  (fruta ?nome)
  =>
  (printout t "Fruta " ?nome " encontrada." crlf))
(defrule listar-fruta
  (fruta ?fruta)
  (fruta-cor ?cor ?fruta)
  =>
  (printout t "Fruta " ?nome " cor " ?cor crlf))
```

É possível atribuir o índice de um fato em uma variável. Para isto, é utilizada a sintaxe abaixo:

```
?<nome variável> <- <fato>
```

Exemplo:

```
(defrule remover-fato
  ?fato <- (fruta ?)
  =>
  (printout t "Removendo fato " ?fato crlf)
  (retract ?fato))
```

O *wildcard* `?` pode ser utilizado para realizar combinações em múltiplos fatos, entretanto, não há possível utilizá-lo em fatos com a quantidade de elementos variada. Observe os fatos abaixo:

```
(membros-de beatles john-lenon paul-mccartney ringo-starr)
(membros-de who roger-daltrey pete-townsend)
```

Analisando os exemplos acima, é possível observar que os dois fatos possuem o mesmo nome, porém, os elementos e a quantidade de elementos de cada fato são diferentes. Para estas situações, O CLIPS fornece um *wildcard* especial, denominado `$?`, capaz de combinar fatos com diferentes quantidades de elementos. O exemplo abaixo irá listar os membros de bandas mencionados anteriormente.

```
(defrule membro-banda
  (membros-de ?banda $? ?membro $?)
  =>
  (printout t ?membro " é membro da banda " ?banda crlf))
```

Se o objetivo for listar o último elemento do fato `membros-de`, basta inserir o wildcard `$?` seguido de uma variável, exemplo:

```
(defrule ultimo-elemento
  (membros-de ?banda $? ?ultimo)
  =>
  (printout t "Ultimo elemento: " ?ultimo crlf))
```

Até agora todos os exemplos vistos de regras de produção utilizam variáveis definidas na pré-condição (LHS). Entretanto, é possível definir variáveis na ação (RHS), através da função ***bind***.

(***bind*** <variável> <expressão>)

Exemplo:

```
(defrule soma
   (numero ?x)
   (numero ?y)
   =>
   (bind ?total (+ ?x ?y))
   (printout t "Soma: " ?total crlf))
```

1.3.4 Variáveis globais

O CLIPS oferece suporte à criação de variáveis globais através da função ***defglobal***. As variáveis globais podem ser manipuladas em qualquer regra de produção, mantendo seu valor a cada inferência realizada na base de conhecimento. Obrigatoriamente todas as variáveis globais devem possuir o caractere "*" no início e fim de seu nome. Para atribuir valores às variáveis globais, deve ser utilizada a função ***bind***.

```
(defglobal
   ?*<nome da variável>* = <valor inicial>
   ...)
```

Exemplo:

```
(defglobal
   ?*var1* = "teste"
   ?*var2* = 10)
```

1.3.5 Templates

Em geral, a utilização de fatos simples torna a base de conhecimento complexa quando existem relações entre os fatos. Por exemplo, em uma base para diagnóstico médico, cada paciente possui um conjunto de informações relativas ao seu estado de saúde, por exemplo: idade, peso, pressão sanguínea, se é fumante ou não, se pratica atividades físicas, dentre outros. Estas informações estão associadas ao paciente e, sua implementação através de fatos simples torna a base complexa, devido ao aumento no volume de fatos e de suas dependências. Para estas situações, pode ser utilizado templates para representar de maneira mais elegante um conjunto de informações. Um template é uma estrutura similar a um registro existente na maioria das linguagens de programação. Deste modo, é possível adicionar atributos (ou informações) em um template. Para definir a estrutura de um template no CLIPS, é utilizada a função ***deftemplate***. Para definir os atributos de um template, utiliza-se a função ***slot***. Existe também um atributo especial, o qual permite armazenar mais de uma informação. Este atributo é definido através da função ***multislot***.

```
(deftemplate <nome do template>
  (slot <nome do atributo>)
  (multislot <nome do atributo>)
  ...)
```

Exemplo:

```
(deftemplate paciente
  (slot nome)
  (slot idade)
  (slot tipoSanguineo)
  (slot peso)
  (slot fumante)
  (multislot pressaoSanguinea))
```

É importante ressaltar que o comando ***deftemplate*** não instancia um template, seu objetivo é definir a estrutura de um dado template. Para instanciar um template, é usado o mesmo comando para adicionar fatos, ***assert***. Entretanto, é possível instanciar um template sem a necessidade de informar todos os valores de seus atributos.

```
(assert
  (<nome do template>
    (<nome do atributo> <valor1 do atributo> ... <valorN do atributo>)
    ...)
  ))
```

Exemplo:

```
(assert
  (paciente
    (nome zeca)
    (idade 25)
    (pressaoSanguinea 130 80)
  ))
```

1.3.6 Alterando valores dos slots

No exemplo acima, não foram informados os valores de alguns atributos durante a adição do template. Deste modo, os atributos que ficaram sem valores irão possuir valor ***nil*** (nulo). Entretanto, é possível modificar os valores dos atributos de instâncias de templates sem a necessidade de remover e adicionar a nova instancia contendo o valor atualizado. Isto é realizado através da função ***modify***.

```
(modify
  (<ID da instância do template>
    <fato atualizado>))
```

Exemplo:

```
(defrule aniversario
  ?aniversariante <- (aniversariante ?nome)
  ?paciente       <- (paciente (nome ?nome) (idade ?idade))
  =>
  (modify ?paciente (idade (+ ?idade 1)))
  (retract ?aniversariante))
```

O exemplo acima incrementa a idade de um paciente quando há um fato indicando que o mesmo é aniversariante. A inferência sobre uma instância de um template também pode ser realizada de forma parcial, isto é, em alguns atributos do template. Para simular este exemplo, adicione o fato `(aniversariante zeca)` e observe o incremento de sua idade.

1.3.7 Operador test

Até agora criamos regras apenas combinando elementos (parcial ou totalmente) de fatos distintos. Contudo, o CLIPS também suporta realizar operações condicionais sobre elementos. Para realizar condicionais, utiliza-se a função ***test***. O exemplo a seguir faz uso desta função para comparar um valor oriundo de uma instância de um dado template, contudo, este recurso pode ser utilizado para testar qualquer tipo de expressão booleana.

```
(test <expressão>)
```

Exemplo:

```
(defrule obesidade
  (paciente (nome ?nome) (peso ?peso))
  (test (> ?peso 100))
  =>
  (printout t ?nome " pesa " ?peso "kg - está obeso." crlf))
```

Os operadores booleanos suportados pelo CLIPS são:

- ***eq***: igual a (comparação de strings)
- ***neq***: diferente de (comparação de strings)
- **=**: igual a (comparação de valores numéricos)
- **<>**: diferente de (comparação de valores numéricos)
- **>**: maior que
- **<**: menor que
- **>=**: maior igual a
- **<=**: menor igual a
- ***not***: nega a expressão

1.3.8 Operadores exists e foreach

Além do operador condicional ***test*** listado acima, o CLIPS provê outros dois operadores condicionais especiais: ***exists*** e ***forall***. O operador condicional ***exists*** retornará verdadeiro, isto é, será satisfeito, caso um ou mais fatos combinarem com a expressão. É importante ressaltar que, ao utilizar o operador condicional ***exists***, caso existam mais de uma ocorrência na base de conhecimento que satisfaça a expressão, a ação da regra somente será executada uma vez, diferentemente de uma comparação tradicional.

```
(exists <expressão>)
```

Exemplo:

```
(assert
   (paciente
     (nome joca)
     (idade 43)
     (pressaoSanguinea 110 95)
   ))
(defrule existe_pacientes
   (exists (paciente (nome ?)))
   =>
   (printout t "Existe ao menos um paciente" crlf))
```

Ao executar o exemplo acima, haverá duas instâncias de paciente na base de conhecimento. Ao executar a regra do exemplo, apenas será executada uma vez sua ação devido ao comportamento do operador condicional ***exists***.

De modo contrário, o operador condicional ***forall*** somente executará a ação se todas as possíveis combinações para a expressão informada retornar verdadeiro.

```
(forall <expressão> <condição>)
```

Exemplo:

```
(defrule verifica_pacientes
   (forall (paciente (peso ?peso)) (test (< peso 90)))
   =>
   (printout t "Não existem pacientes obesos" crlf))
```

Neste exemplo não será executada a ação da regra **verifica_pacientes** pois nem todos os pacientes possuem um peso menor que 90 quilos. Para testar esta situação, remova todos os pacientes da base de conhecimento, inclua dois novos pacientes com peso menor que 90 quilos e realize a inferência na base.

1.3.9 Operador logical

Outro operador condicional muito utilizado para construção de sistemas de manutenção de verdade (TMS, Truth Maintenance System) no CLIPS é o operador ***logical***. Para compreendermos o funcionamento deste operador condicional, vamos observar o exemplo abaixo:

```
(defrule risco-cardiaco
  (paciente (nome ?nome) (fumante sim) (peso ?peso))
  (test (> ?peso 90))
  =>
  (assert (risco-cardiaco ?nome)))
```

Analisando o exemplo acima, é possível concluir que um paciente será notificado como potencial risco cardíaco quando o mesmo for fumante e tem um peso maior que 90 quilos. Contudo, se o paciente perder peso ou deixar de fumar, a notificação prevalecerá, pois não há nenhuma ligação entre o fato adicionado e a regra que o adicionou. Portanto, neste cenário a base de conhecimento ficaria inconsistente com a realidade. Para resolver este impasse, é utilizado o elemento ***logical*** para ligar a ação com as premissas da regra. Quando as premissas deixam de ser válidas, é desfeita a ação da regra em questão, exemplo:

```
(defrule risco-cardiaco
  (logical (paciente (nome ?nome) (fumante sim) (peso ?peso)))
  (logical (test (> ?peso 90)))
  =>
  (assert (risco-cardiaco ?nome)))
```

Adicione dois pacientes que se encaixam em risco cardíaco e realize a inferência. Após a inferência, haverá dois fatos `risco-cardiaco` na base de conhecimento. Contudo, ao alterar o peso de um dos pacientes para menos de 90 quilos, automaticamente o alerta de risco cardíaco deste paciente será removido. Este é um poderoso recurso para manter íntegra a base de conhecimento, não necessitando de um mecanismo para revisão das crenças.

2. Integração CLIPS e C/C++

2.1 A API CLIPS

A API do CLIPS provê inúmeros métodos para integração da base de conhecimento com códigos definidos pelo usuário. O ambiente CLIPS foi inteiramente desenvolvido na linguagem de programação C, o que facilita a interação entre as linguagens C e C++. De fato, é possível invocar rotinas internas do CLIPS por meio de uma interface bem definida, permitindo assim a construção de sistemas especialistas mais sofisticados. Contudo, é necessário um conhecimento básico da interface e estruturas da API, pois a manipulação indevida das rotinas internas do CLIPS pode ocasionar em erros no ambiente como um todo. Para estudo e elaboração deste documento, foi utilizada a última versão (release 6.24) disponível do ambiente CLIPS. Nos próximos tópicos serão apresentadas as principais funções da API CLIPS. Os códigos de exemplo utilizados ao longo deste documento foram desenvolvidos em GNU C++ para maior portabilidade. Para facilitar a compreensão dos exemplos em C++, os códigos serão formados utilizando a fonte Courier tamanho 10, as funções nativas do CLIPS serão destacadas em negrito e os comandos da linguagem C++ serão destacados na cor azul.

2.2 Interface do CLIPS

Antes de fazer uso de qualquer função da API do ambiente CLIPS, é necessário incluir o cabeçalho `clips.h`. Conforme já mencionado, a API do CLIPS foi desenvolvida em C, portanto, para permitir a compatibilidade com a linguagem C++, as funções invocadas que são nativas do CLIPS necessitam de prototipação através do comando `extern "C"`.

2.2.1 Iniciando o ambiente

A primeira etapa antes de manipular ou realizar inferências em uma base de conhecimento é a inicialização de variáveis e estruturas do ambiente CLIPS, através da função **`InitializeCLIPS`**.

```
extern "C" void InitializeCLIPS(void);

#include <stdio.h>
#include "clips.h"

int main(int argc, char* argv[])
{
  InitializeCLIPS();
}
```

2.2.2 Carregando a base de conhecimento

Neste momento, o ambiente CLIPS foi iniciado, entretanto, ainda não carregada nenhuma base de conhecimento. Um recurso muito útil do CLIPS é a carga de uma base de conhecimento contida um

arquivo texto (fatos, regras, templates, dentre outros). Para tal, o arquivo texto deve conter a definição de uma base de conhecimento através da sintaxe do CLIPS. A função utilizada para carga da base de conhecimento a partir de um arquivo texto é **Load**. A função **Load** recebe como argumento uma string contendo o endereço físico do arquivo texto que define a base de conhecimento e retorna um valor inteiro, que representa o status da operação. Quando o valor retornado da função **Load** for igual a 0, o arquivo texto não foi encontrado ou não foi possível realizar a leitura. Caso o valor for igual a -1, ocorreu um erro durante a carga da base de conhecimento (sintaxe inválida, por exemplo). Do contrário, a carga da base de conhecimento foi efetuada com sucesso.

```
extern "C" {
  void InitializeCLIPS(void);
  int  Load(char*);
}

#include <stdio.h>
#include "clips.h"

int main(int argc, char* argv[])
{
  InitializeCLIPS();

  int ret = Load("Bases\\teste.clp");

  switch(ret)
    {
      case 0:
        printf("Base de conhecimento nao encontrada\n");
        break;
      case -1:
        printf("Erro ao ler a base de conhecimento\n");
        break;
      default:
        printf("Base de conhecimento carregada\n");
        break;
    }
}
```

2.2.3 Iniciando a base de conhecimento

Agora a base de conhecimento já está carregada no ambiente CLIPS. Porém, geralmente é necessário inicializar a base de conhecimento, que pode ser realizada através da função **Reset**. Esta função remove os fatos e limpa a memória de trabalho do CLIPS, mantendo apenas as regras definidas e os fatos contidos na estrutura ***deffacts***. Para realizar uma inferência na base de conhecimento, é utilizada a função **Run**. A função **Run** recebe como argumento um valor inteiro, que representa a quantidade limite de regras que podem disparadas ao realizar a inferência. Caso o valor for um valor negativo, serão disparadas todas as regras aplicáveis para a situação corrente da base de conhecimento. Desta forma, é possível realizar podas em uma inferência da base de conhecimento, limitando o espaço de busca do problema. Na seqüência, é ilustrado o uso das funções **Reset** e **Run**, a partir da base de conhecimento abaixo:

```
(deffacts iniciar
  (animal pato))

(defrule animal
  (animal pato)
  =>
  (assert (som quack))
  (printout t "Quack" crlf))
```

```
extern "C" {
  void InitializeCLIPS(void);
  void Reset(void);
  void Run(int);
  int  Load(char*);
}

#include <stdio.h>
#include "clips.h"

int main(int argc, char* argv[])
{
  InitializeCLIPS();
  int ret = Load("Bases\\teste.clp");

  switch(ret)
    {
      case 0:
        printf("Base de conhecimento nao encontrada\n");
        break;
      case -1:
        printf("Erro ao ler a base de conhecimento\n");
        break;
      default:
        printf("Base de conhecimento carregada\n");
        break;
    }
  Reset();
  Run(-1);
}
```

Após a chamada da função **Run**, a regra **animal** foi disparada. Porém, a base foi carregada de maneira estática, através da estrutura ***deffacts***. É comum os fatos necessitarem ser carregados dinamicamente na base de conhecimento. Observe a base de conhecimento abaixo:

```
(deffacts iniciar
  (som quack))

(defrule animal
  (som quack)
  (tipo ave)
  =>
  (assert (animal pato))
  (printout t "Sou um pato" crlf))
```

2.2.4 Adicionando fatos na base de conhecimento

A regra **animal** não será disparada, pois somente existe na base o fato `(som quack)` adicionado pela estrutura ***deffacts*** e não há nenhuma outra regra que possa adicionar o fato `(tipo ave)`. Deste modo, o único meio de disparar a regra **animal** é adicionar dinamicamente o fato `(tipo ave)` na base de conhecimento. Para tal, o CLIPS provê a função **AssertString** para adicionar fatos na base semelhante à função **assert**. A função **AssertString** recebe como argumento uma string contendo a definição do fato e retorna um ponteiro da estrutura do fato no CLIPS (`struct fact`).

```
extern "C" {
  void  InitializeCLIPS(void);
  void  Reset(void);
  void  Run(int);
  void* AssertString(char*);

  int   Load(char*);
}


#include <stdio.h>
#include "clips.h"

int main(int argc, char* argv[])
{
  InitializeCLIPS();
  int ret = Load("Bases\\teste.clp");

  switch(ret)
    {
      case 0:
        printf("Base de conhecimento nao encontrada\n");
        break;
      case -1:
        printf("Erro ao ler a base de conhecimento\n");
        break;
      default:
        printf("Base de conhecimento carregada\n");
        break;
    }
  Reset();
  AssertString("(tipo ave)");
  Run(-1);
}
```

A função **AssertString** também permite adicionar instâncias de templates, uma vez que a definição do template exista na base de conhecimento. Observe o exemplo abaixo:

```
(deftemplate paciente
      (slot nome)
      (slot idade)
      (slot peso)
      (slot fumante)
      (multislot pressaoSanguinea))
```

```
(defrule listagem
  (paciente (nome ?nome) (pressaoSanguinea ?i ?f))
  =>
  (printout t "Paciente: " ?nome " pressao: (" ?ini "," ?fim ")"
crlf))

// ...
AssertString("(paciente (nome jose) (fumante n) (pressaoSanguinea 90
110))");
// ...
Run(-1);
// ...
```

Conforme mencionado anteriormente, a função **AssertString** retorna o ponteiro de uma instância da estrutura `fact`. Em outras palavras, este ponteiro retornado refere-se ao próprio fato adicionado. Deste modo, é possível manipular o fato em questão através desta estrutura. Por exemplo, para remover este fato adicionado, basta utilizar a função **Retract** e passar como argumento o ponteiro do fato a ser removido.

```
// ...
fact* fato = (fact*) AssertString("(tipo ave)");
// ...
Retract(fato);
// ...
```

É possível adicionar instâncias de templates de outra maneira, a qual faz uso das funções nativas do CLIPS. Isto é necessário quando o fato manipula objetos complexos em seus slots (ponteiros para objetos, estruturas, funções, dentre outros). Para executar o exemplo abaixo, utilize a base de conhecimento que contém a definição do template `paciente` e a regra `listagem`.

```
extern "C" {
  struct fact;
  struct dataObject;
  struct fact* CreateFact(void*);

  void* Assert(void*);
  void* AssertString(char*);
  void* FindDeftemplate(char*);
  void* AddSymbol(char*);
  void* AddLong(int);
  void* AddDouble(double);
  void* CreateMultifield(int);

  long  Run(long);
  int   Load(char*);
  int   Retract(void*);
  int   PutFactSlot(void*,char*,dataObject*);
  int   AssignFactSlotDefaults(void*);

  void  Reset(void);
  void  InitializeCLIPS(void);
}

#include <stdio.h>
#include "..\\CLIPS\\clips.h"
```

```
int main(int argc, char* argv[])
{
  InitializeCLIPS();
  int ret = Load("Bases\\teste.clp");

  if(ret != 1)
  {
    printf("Erro ao ler a base de conhecimento\n");
    exit(-1);
  }
  Reset();

  deftemplate* temp = (deftemplate*) FindDeftemplate("paciente");
  fact*        fato = (fact*) CreateFact(temp);
  dataObject   valor;

  valor.type  = SYMBOL;
  valor.value = AddSymbol("jose");
  PutFactSlot(fato,"nome",&valor);

  valor.type  = INTEGER;
  valor.value = AddLong(25);
  PutFactSlot(fato,"idade",&valor);

  valor.type  = FLOAT;
  valor.value = AddDouble(80.5);
  PutFactSlot(fato,"peso",&valor);

  valor.type  = SYMBOL;
  valor.value = AddSymbol("sim");
  PutFactSlot(fato,"fumante",&valor);

  multifield* multislot = (multifield*) CreateMultifield(2);
  SetMFType(multislot,1,FLOAT);
  SetMFValue(multislot,1,AddDouble(90));
  SetMFType(multislot,2,FLOAT);
  SetMFValue(multislot,2,AddDouble(110));

  SetDOBegin(valor,1);
  SetDOEnd(valor,2);

  valor.type  = MULTIFIELD;
  valor.value = multislot;
  PutFactSlot(fato,"pressaoSanguinea",&valor);

  AssignFactSlotDefaults(fato);
  Assert(fato);

  Run(-1);
}
```

No exemplo acima, foram utilizadas as seguintes funções:

- **`FindDeftemplate`**: retorna um ponteiro da estrutura de um determinado template contido na base de conhecimento. Esta função recebe como argumento o nome do template e, caso o mesmo não seja encontrado na base de conhecimento, será retornado `NULL`.
- **`CreateFact`**: retorna um ponteiro da estrutura de um determinado fato. É importante ressaltar que o fato ainda não faz parte da base de conhecimento, isto é, apenas sua estrutura foi criada. O fato somente irá pertencer à base de conhecimento após ser invocada a função **`Assert`**. A

função **`CreateFact`** recebe como argumento o ponteiro de um template, sendo que o fato a ser criado será baseado na estrutura deste template. Portanto, esta função apenas é utilizada para a construção de fatos baseados em templates.

- **`AddSymbol`**: define o símbolo (string, símbolos, instâncias, dentre outros) a ser atribuído para um dado slot do template.
- **`AddLong`**: define um número inteiro a ser atribuído para um dado slot do template.
- **`AddDouble`**: define um número com precisão para ser atribuído a um slot do template.
- **`PutFactSlot`**: adiciona o conteúdo de um slot em um determinado fato. Esta função recebe como argumento:
    1. Ponteiro do fato a ser adicionado o conteúdo de um determinado slot;
    2. Nome do slot a ser atribuido o conteúdo; e
    3. Conteúdo do slot (`struct dataObject`). Deve ser especificado o tipo do conteúdo (`SYMBOL`, `LONG`, `DOUBLE`, etc.) e o conteúdo propriamente dito, através dos membros `type` e `value` da estrutura `dataObject`.
- **`CreateMultifield`**: cria uma estrutura para um slot de múltiplos valores (multislot). Esta função recebe como argumento a quantidade de valores que o slot irá possuir. Será retornado um ponteiro para uma estrutura de múltiplos slots (`struct multislot`).
- **`SetMFType`**: define o tipo do conteúdo de um determinado item em um slot de múltiplos valores. Esta função recebe como argumento:
    1. Ponteiro do slot de múltiplos valores a ser definido o tipo do conteúdo de um determinado item;
    2. Posição do item no slot de múltiplos valores; e
    3. Tipo de conteúdo do item (`SYMBOL`, `LONG`, `DOUBLE`, etc.).
- **`SetMFValue`**: define o conteúdo de um determinado item em um slot de múltiplos valores. Esta função recebe como argumento:
    1. Ponteiro do slot de múltiplos valores a ser definido o conteúdo de um determinado item;
    2. Posição do item no slot de múltiplos valores; e
    3. Conteúdo do item.
- **`SetDOBegin`**: indica o índice de início dos slots de múltiplos valores. Esta função recebe como argumento o ponteiro para o conteúdo do slot e o índice de início do slot de múltiplos valores.
- **`SetDOEnd`**: indica o índice de término dos slots de múltiplos valores. Esta função recebe como argumento o ponteiro para o conteúdo do slot e o índice de término do slot de múltiplos valores.
- **`AssignFactSlotDefaults`**: atribui os valores padrão para os slots não inicializados em determinado fato. Esta função recebe como argumento o ponteiro do fato a ter os slots vazios inicializados com os respectivos valores padrões.
- **`Assert`**: adiciona um fato na base de conhecimento. Esta função recebe como argumento o ponteiro do fato a ser adicionado na base de conhecimento.

2.3 Integrando o CLIPS com Funções Externas C ou C++

De fato, um dos mais importantes recursos do CLIPS é a possibilidade de integrá-lo a funções externas ou aplicações. Tal recurso torna o ambiente bastante flexível, permitindo assim realizar ações complexas durante a inferência da base de conhecimento. O CLIPS provê mecanismos para integração com as linguagens de programação C ou C++, Ada e FORTRAN, entretanto, este relatório tratará apenas os modelos de integração com funções desenvolvidas em C ou C++.

Todas as funções externas devem ser descritas no ambiente CLIPS dentro das funções **`UserFunctions`** ou **`EnvUserFunctions`**. Para descrever uma função externa ao ambiente CLIPS, é necessário utilizar a função nativa do CLIPS abaixo:

`DefineFunction``(<função no CLIPS>,<retorno>,<ponteiro>,<nome função>)`

Os argumentos da função **`DefineFunction`** são:

1. Nome da função a ser referenciada na base de conhecimento. Este nome é uma string que representa o nome da função que será usado quando chamada dentro do CLIPS;
2. Tipo do retorno da função externa. O tipo de retorno deve ser uma string que representa o tipo do conteúdo retornado pela função, conforme a listagem abaixo:
    - `'a'`: Endereço externo (`pointer`);
    - `'b'`: Booleano (`int`, `unsigned int`);
    - `'c'`: Caractere (`char`);
    - `'d'`: Número com dupla precisão (`double`);
    - `'f'`: Número com única precisão (`float`);
    - `'i'`: Número inteiro (`int`, `unsigned int`);
    - `'j'`: Tipo desconhecido (símbolo, string ou nome de instância);
    - `'k'`: Tipo desconhecido (símbolo ou string);
    - `'l'`: Número inteiro longo (`long`);
    - `'k'`: Tipo desconhecido (símbolo ou string);
    - `'m'`: Slot de múltiplos valores (`pointer`);
    - `'n'`: Tipo desconhecido (`integer` ou `float`);
    - `'o'`: Nome de instância (`char*`);
    - `'s'`: String (`char*`);
    - `'u'`: Tipo desconhecido (qualquer tipo);
    - `'v'`: Sem retorno (`void`);
    - `'w'`: Símbolo (`char*`); e
    - `'x'`: Endereço de instância (`pointer`).
3. Ponteiro da função externa. Caso a função externa não esteja dentro do próprio arquivo fonte com a definição da função **`UserFunctions`**, deve ser utilizado o comando `extern` no prototipação da função.
4. Nome real da função externa. Este argumento deve ser uma string contendo exatamente o nome físico da função.

A função **DefineFunction** irá retornar 0 se algum erro ocorrer durante a descrição da função externa ao ambiente CLIPS, caso contrário, será retornado 1. A seguir será exemplificado o uso de funções externas no ambiente CLIPS.

```
extern "C" {
  void* AssertString(char*);

  long  Run(long);
  int   Load(char*);
  int   DefineFunction(char*,char,int (*pointer)(void),char*);

  void  Reset(void);
  void  InitializeCLIPS(void);
  void  UserFunctions(void);
}

#include <stdio.h>
#include "..\\CLIPS\\clips.h"

int main(int argc, char* argv[])
{
  InitializeCLIPS();

  int ret = Load("Bases\\teste.clp");

  if(ret != 1)
  {
    printf("Erro ao ler a base de conhecimento\n");
    exit(-1);
  }

  Reset();
  AssertString("(paciente (nome jose) (idade 45) (peso 80.5))");
  Run(-1);
}

void teste()
{
  printf("Dentro de uma função externa\n");

  AssertString("(tem-funcao-externa ok)");
}

void UserFunctions(){
  DefineFunction("FuncaoExterna",'v',PTIF teste,"teste");
}
```

No exemplo acima, foi definida uma função chamada `teste` para ser integrada ao CLIPS. Para tal, a função **UserFunctions** foi sobrescrita a fim de descrever esta função `teste` a ser integrada. Deste modo, dentro da função **UserFunctions**, foi descrita a função externa `teste` através da função **DefineFunction** do CLIPS, especificando o endereço de memória da função externa, seu retorno e o nome que a função irá possuir dentro do ambiente CLIPS. Note que o nome físico da função e o nome da função no CLIPS são diferentes, isto é, a função `teste` será representada no CLIPS pelo nome `FuncaoExterna`. Para testar este exemplo, utilize a base de conhecimento definida abaixo:

```
(deftemplate paciente
      (slot nome)
      (slot idade)
      (slot peso)
      (slot fumante)
      (multislot pressaoSanguinea))

(defrule listagem
  (paciente (nome ?nome) (idade ?idade) (peso ?peso))
  =>
  (FuncaoExterna)
  (printout t "Paciente: " ?nome " idade: " ?idade " peso: " ?peso
crlf))

(defrule testa-funcao-externa
  (tem-funcao-externa ok)
  =>
  (printout t "Função externa OK" crlf))
```

Neste exemplo, há duas regras, **listagem** e **testa-funcao-externa**. A regra **listagem** terá sua ação invocada quando houver um paciente com nome, idade e peso definidos na base de conhecimento. Note que antes de realizar a inferência na base, é realizado um **AssertString** para adicionar um paciente nestas condições e forçar a execução da ação da regra **listagem**. Ao disparar a ação da regra **listagem**, é invocada a função no `FuncaoExterna` no CLIPS, que por sua vez irá invocar a função externa `teste`. Dentro da função externa `teste`, é realizado mais um **AssertString**, a fim de forçar a execução da ação da regra **testa-funcao-externa**. Por fim, esta ação irá apenas imprimir uma mensagem no prompt.

Até agora, vimos como integrar funções externas ao CLIPS através da função **DefineFunction**. Contudo, as funções externas integradas com a função **DefineFunction** não podem possuir argumentos. Para suprir esta deficiência, existe a função **DefineFunction2** capaz de integrar funções externas ao CLIPS que recebam argumentos. Neste caso, é necessário especificar os argumentos da função externa em questão, conforme abaixo:

```
DefineFunction2(<função no CLIPS>,
                <retorno>,
                <ponteiro função>,
                <nome função>,
                <definição dos argumentos>)
```

A diferença entre a função **DefineFunction2** e **DefineFunction** a é a existência de um quinto argumento para especificar os argumentos necessários para invocar a função externa. Esta especificação é realizada através de uma string no seguinte formato:

```
<número mínimo de argumentos> <número mínimo de argumentos> [<tipo padrão>
<tipo>*]
```

As informações de número mínimo e máximo de argumentos são obrigatórias. Ambos os casos devem ser um valor numérico (de 0 a 9) ou o símbolo * que identifica a inexistência de restrições da quantidade de argumentos mínima ou máxima. Observe os exemplos abaixo:

- `"25"`: a função pode receber entre 2 e 5 argumentos;
- `"33"`: a função recebe 3 argumentos;

- `"4*"`: a função pode receber 4 ou mais argumentos;
- `"*6"`: a função pode receber até 6 argumentos; e
- `"**"`: a função não tem restrições de argumentos.

As informações tipo padrão e tipo são opcionais e definem o tipo de dado de cada argumento. Os tipos de dados podem ser:

- `'a'`: Endereço externo;
- `'d'`: Número com dupla precisão;
- `'e'`: Endereço de instância, nome de instância ou símbolo;
- `'f'`: Número com única precisão;
- `'g'`: Número inteiro, número com única precisão ou símbolo;
- `'h'`: Endereço de instância, nome de instância, endereço de fato, número inteiro ou símbolo;
- `'i'`: Número inteiro;
- `'j'`: Símbolo, string ou nome de instância;
- `'k'`: Símbolo ou string;
- `'l'`: Número inteiro;
- `'m'`: Slot de múltiplos valores;
- `'n'`: Número inteiro ou número com única precisão;
- `'o'`: Nome de instância;
- `'p'`: Nome de instância ou símbolo;
- `'q'`: Símbolo, string ou slot de múltiplos valores;
- `'s'`: String;
- `'u'`: Qualquer tipo de dado;
- `'w'`: Símbolo;
- `'x'`: Endereço de instância;
- `'y'`: Endereço de fato; e
- `'z'`: Endereço de fato, número inteiro e símbolo.

O tipo padrão pode ser considerado como uma regra para o tipo de dados de todos os argumentos de uma dada função externa. Caso for definido um tipo de dado para um determinado argumento, a regra é anulada e será considerado o tipo de dado informado como exceção. Observe os exemplos a seguir:

- `"25l"`: a função pode receber entre 2 e 5 argumentos, todos do tipo inteiro;
- `"26ls"`: a função recebe de 2 a 6 argumentos, sendo que o primeiro é do tipo string e os demais são inteiros; e
- `"4*lusd"`: a função pode receber 4 ou mais argumentos, sendo que o segundo é do tipo string, o terceiro é um número com dupla precisão e os demais são inteiros.

A seguir será exemplificado o uso da função **`DefineFunction2`** para integrar funções externas que recebem argumentos do CLIPS:

```
extern "C" {
  void* AssertString(char*);

  long  Run(long);
  long  RtnLong(int);

  int   Load(char*);
  int   RtnArgCount(void);
  int   DefineFunction2(char*,char,int (*pointer)(void),char*,char*);

  void  Reset(void);
  void  InitializeCLIPS(void);
  void  UserFunctions(void);

  char* RtnLexeme(int);
}

#include <stdio.h>
#include "..\\CLIPS\\clips.h"

int main(int argc, char* argv[])
{
  InitializeCLIPS();

  int ret = Load("Bases\\teste.clp");

  if(ret 1= 1)
  {
    printf("Erro ao ler a base de conhecimento\n");
    exit(-1);
  }

  Reset();

  AssertString("(paciente (nome jose) (idade 45) (peso 80))");

  Run(-1);
}

void teste()
{
  printf("Dentro de uma função externa\n");

  int   argc = RtnArgCount();
  int   arg1;
  char* arg2;
  int   arg3;

  switch(argc)
  {
    case 0:
      printf("Nao recebeu nenhum argumento\n");
      break;
    case 1:
      arg1 = RtnLong(1);

      printf("Recebeu 1 argumento (%d)\n",arg1);
      break;
    case 2:
      arg1 = RtnLong(1);
      arg2 = RtnLexeme(2);
```

```
      printf("Recebeu 2 argumentos (%d,%s)\n",arg1,arg2);
      break;
    case 3:
      arg1 = RtnLong(1);
      arg2 = RtnLexeme(2);
      arg3 = RtnLong(3);

      printf("Recebeu 3 argumentos (%d,%s,%d)\n",arg1,arg2,arg3);
      break;
  }

  AssertString("(tem-funcao-externa ok)");
}

void UserFunctions(){
  DefineFunction2("FuncaoExterna",'v',PTIF teste,"teste","*3ius");
}
```

No exemplo acima, foi definida uma função externa chamada `teste` para ser integrada ao CLIPS. Entretanto, esta função externa `teste` pode receber até três argumentos, sendo que o segundo argumento é do tipo string e os demais são do tipo inteiro. Portanto, a função externa `teste` foi descrita através da função **`DefineFunction2`** do CLIPS. Para obter a quantidade de argumentos informada à função, foi utilizada a função **`RtnArgCount`**. Esta função retorna um número inteiro que indica a quantidade de argumentos que foram informados ao invocar a função corrente. Para retornar os valores de cada argumento, foram utilizadas as funções **`RtnLong`** e **`RtnLexeme`**. Estas funções requerem que seja informada a posição do argumento. Em outras palavras, é necessário informar a posição do argumento ao ser invocada a função corrente. Existem três funções para retornar os valores dos argumentos de funções externas:

- **`RtnLong`**: retorna valores de argumentos do tipo inteiro e inteiro longo;
- **`RtnDouble`**: retorna valores de argumentos do tipo número com única ou dupla precisão; e
- **`RtnLexeme`**: retorna valores de argumentos do tipo símbolo, string, nomes de instancias e outros.

Para testar este exemplo, utilize a base de conhecimento definida abaixo:

```
(deftemplate paciente
   (slot nome)
   (slot idade)
   (slot peso)
   (slot fumante)
  (multislot pressaoSanguinea))

(defrule listagem
  (paciente (nome ?nome) (idade ?idade) (peso ?peso))
  =>
  (FuncaoExterna)
  (FuncaoExterna ?idade)
  (FuncaoExterna ?idade ?nome)
  (FuncaoExterna ?idade ?nome ?peso)
  (printout t "Paciente: " ?nome " idade: " ?idade " peso: " ?peso "
crlf))
```

```
(defrule testa-funcao-externa
  (tem-funcao-externa ok)
  =>
  (printout t "Função externa OK" crlf))
```