ATIVAÇÃO DOS PREDEFINIDOS, PREDEFINIDOS DE E/S E CONTROLE DE FLUXO NO PRONUS PROLOG

Autor
Luciano Conrado Machado Dallolio
Orientador
Luiz Fernando Pereira de Souza

NCE-11/91

Grupo de Inteligência Artificial/NCE

agosto/91

Universidade Federal do Rio de Janeiro
Núcleo de Computação Eletônica
Caixa Postal 2324
20001 - Rio de Janeiro - RJ
BRASIL
E-mail : PROLOG@VAX1.UFRJ.RNP.BR

Ativação dos predefinidos, predefinidos de E/S e controle de fluxo no Pronus Prolog.

Sinopse.

Este relatório descreve a implementação do primeiro grupo de predefinidos do interpretador Pronus Prolog.

A primeira parte do trabalho aborda o problema da interface entre uma execução Prolog e a biblioteca de predefinidos, descrevendo-se a solução adotada.

Logo a seguir são estudados os primeiros grupos de predefinidos implementados, incluindo os predefinidos para manipulação da tabela de operadores, impressão de termos, manipulação de cadeias, avaliação de expressões, controle do fluxo da execução e entrada e saída em arquivos.

A fim de facilitar o entendimento deste relatório seria interessante que o leitor já estivesse familiarizado com o ambiente de execução do interpretador Pronus Prolog descrito no relatório "Ambiente de execução do Pronus Prolog" [Dallolio 91].

Built-in predicates activation, I/O predicates and flow control in Pronus Prolog.

Abstract.

This report describes the implementation of the first set of built-in predicates of Pronus Prolog interpreter.

The first part of this work discusses the way by which the interface between Prolog's execution and built-in predicates was implemented.

It is also presented the first set of built-in predicates: operator handling predicates, term writing predicates, string handling predicates, expression evaluation predicates, control flow predicates and file I/O predicates.

The reader should refer to the report "Pronus Prolog execution environment" [Dallolio 91] in order to get a background for an easier understanding of this report.

Índice.

1. Introdução 1
2. Ativação dos predefinidos 2
3. Predefinidos para manipulação da tabela de operadores 8
   3.1. Predefinido op(Prec, Assoc, Op) 8
   3.2. Predefinido current_op(Prec, Assoc, Op) 11
   3.3. Predefinido reset_op 11
   3.4. Tabela de operadores e análisa sintática 12
4. Predefinidos para impressão de termos 14
5. Predefinidos para manipulação de cadeias 20
   5.1. Predefinido string_term(Cadeia, Termo) 20
6. Predefinidos para avaliação de expressões 24
   6.2. Predefinido is(Res, Exp) 24
7. Predefinidos de controle do fluxo da execução 29
   7.1. Predefinido break/0 29
   7.2. Predefinidos call/1 e not/1 33
8. Predefinidos de entrada e saída em arquivos 37
   8.1. Gerência de arquivos para o padrão Edimburgo 37
   8.2. Predefinidos tell/1, telling/1 e told/0 38
   8.3. Predefinidos see/1, seeing/1 e seen/0 41
   8.4. Predefinidos stdout/2, stdin/2, stdinout/3 41
9. Conclusão 46
10. Referências bibliográficas 47

1. Introdução

Este relatório descreve a primeira fase do trabalho de implementação da biblioteca de predefinidos do Pronus Prolog. A biblioteca de predefinidos consiste de uma série de funções escritas na linguagem de programação C, que podem ser ativadas pelo interpretador Pronus durante a execução de um programa Prolog. Esta biblioteca fornece ao usuário um ambiente de programação compatível com a versão 4.0 do interpretador Arity Prolog.

Os predefinidos estão subdivididos em grupos lógicos de acordo com a sua função. Os grupos abordados neste relatório são os predefinidos para manipulação da tabela de operadores, impressão de termos, manipulação de cadeias, avaliação de expressões, controle do fluxo da execução e entrada e saída em arquivos.

Os predefinidos para manipulação da tabela de operadores criam novos operadores ou alteram a definição de operadores já existentes, enquanto que os predefinidos para impressão de termos convertem a representação interna dos termos em Prolog para uma representação que possa ser lida e entendida pelo usuário.

Do grupo dos predefinidos de manipulação de cadeias nós escolhemos descrever apenas o predefinido string_term/2, devido à interface que este predefinido realiza com os módulos de análise. Já do grupo dos predefinidos de avaliação de expressões nós descrevemos o predefinido is/2 mostrando como o resultado da avaliação de uma expressão pode ser afetado pela mudança na definição dos operadores envolvidos nesta expressão.

Os predefinidos para controle do fluxo de execução possuem alguns aspectos interessantes em sua implementação, como a geração de código a tempo de execução e a ativação de instâncias recursivas do módulo Mestre. Finalmente, temos a descrição dos predefinidos de entrada e saída em arquivos que também geram código a tempo de execução além de utilizar uma estrutura de dados especial.

É importante notar que nem todos os predefinidos da biblioteca foram descritos neste trabalho. Como o objetivo deste relatório é expor os problemas relacionados com a implementação de predefinidos, nós só descrevemos os predefinidos mais complexos, cuja implementação demandou a busca de soluções interessantes.

2. Ativação dos predefinidos.

Os predefinidos podem ser divididos em dois grupos distintos com base no comportamento dos mesmos quando ocorre um retrocesso. O primeiro grupo é composto pelos predefinidos que não sofrem uma nova avaliação na ocorrência de um retrocesso, enquanto que o segundo grupo é composto pelos predefinidos que são reavaliados a cada retrocesso. Daqui para frente, chamaremos os primeiros de predefinidos e os demais de predefinidos com retrocesso.

Para disparar a execução de um predefinido, o interpretador posiciona os argumentos nos registradores As e dispara a execução do predicado, ao qual o predefinido pertence, através de uma pseudo-instrução "call" ou "execute". Estas pseudo-instruções acessam a tabela de predicados buscando o ponteiro para o código do predicado e desviando a execução do programa Prolog para aquele endereço. Observe o exemplo abaixo, onde descrevemos o código gerado por uma cláusula que chama o predefinido atom/1, cujo índice hipotético na tabela de predefinidos é 47.

```
p :- f, atom(a), g.

(1) R : allocate
(2)     call    f/0, 0
(3)     put_constant  (ATOM, a), A1
(4)     call    atom/1, 0
(5)     deallocate
(6)     execute g/0, 0

(7) atom/1 : trap 47
(8)          proceed
```

No exemplo acima, a pseudo-instrução (3) "put_constant" posiciona o argumento do predicado no registrador A[1], enquanto que a pseudo-instrução (4) "call" acessa a tabela de predicados no campo em que estão guardadas as informações relativas ao predicado atom/1. Através desta tabela, é obtido o endereço do código de atom/1 e iniciada a sua execução. A pseudo-instrução (7) "trap" acessa a posição 47 da tabela de predefinidos ativando a função escrita em C ali referenciada. Ao final da função, temos dois comportamentos possíveis. Caso a função retorne com sucesso, retoma-

mos a execução do código intermediário a partir da pseudo-instrução (8) "proceed" que retorna para a instrução posterior ao último "call". Já no caso em que ocorre uma falha, o interpretador executa a pseudo_instrução "fail", iniciando um retrocesso.

A execução de um predefinido com retrocesso exige alguns cuidados adicionais. Para que, caso ocorra um retrocesso, seja possível uma nova avaliação da função que implementa o predefinido, é necessária a criação de um ponto de escolha e de um mecanismo que possibilite a recuperação do contexto da última avaliação desta função. Observe o exemplo abaixo, onde descrevemos o código gerado por uma cláusula que chama o predefinido com retrocesso clause/2, cuja entrada hipotética na tabela de predefinidos é 100.

```
p :- a, clause(a, X), b.

(1) R : allocate
(2)     call    a/0, 0
(3)     put_constant    (ATOM, a), A1
(4)     put_variable    X2, A2
(5)     call    clause/2, 0
(6)     deallocate
(7)     execute b/0

(8) clause/2 : put_variable    X3, A3
(9)            try_me_else     L , 3
(10)       L : trap 100
(11)           proceed
```

No exemplo acima, as pseudo-instruções (3) "put_constant" e (4) "put_variable" posicionam os argumentos do predicado clause/2 nos registradores A[1] e A[2], enquanto que a pseudo-instrução (5) "call" busca o código associado ao predicado da mesma maneira que no exemplo anterior. No código do predicado clause/2, temos importantes diferenças com relação ao primeiro exemplo.

A pseudo-instrução (8) "put_variable" cria uma variável livre na pilha global apontada pelo registrador A[3], chamada variável do contexto. Esta variável será utilizada para armazenar o contexto da última chamada efetuada, de modo a restaurá-lo caso ocorra um retrocesso. A pseudo-instrução (9) "try_me_else" cria um ponto de escolha salvando no mesmo o conteúdo dos registradores A[1], A[2] (registradores que contém os argumentos) e A[3] (regis-

trador que aponta para a variável do contexto). Deste modo, no caso de ocorrer um retrocesso, os argumentos apontados por A[1] e A[2] podem ser restaurados tal como estavam na primeira chamada, enquanto que o registrador A[3] estará apontando para uma posição na pilha global que era inicialmente livre, mas que é modificada a cada nova chamada do predefinido de modo a conter o contexto da última chamada.

A variável do contexto se apresenta como um mecanismo para guardar informações relativas à última execução do predefinido, para que este funcione de modo correto em uma eventual avaliação subseqüente causada por um retrocesso. O uso dado a esta variável depende do predefinido que está sendo implementado. No caso do predefinido "clause", a base de dados é percorrida na busca de uma cláusula cuja cabeça unifique com o primeiro argumento do predefinido. Uma vez encontrada esta cláusula, o predefinido tenta unificar o corpo da cláusula com o segundo argumento. O processo é repetido até que as duas unificações sejam positivas ou até que não haja mais cláusulas a analisar na base de dados. Para efeito ilustrativo, considere que a seguinte base de dados esteja presente no interpretador no momento em que o predefinido "clause" é chamado.

```
b :- h,i,j(X).
b :- j,k,l.

a :- b,c,d.
a :- e,f,g.

p :- clause(a, X).
```

A primeira cláusula cuja cabeça irá unificar com o argumento "a" é a cláusula "a :- b,c,d". Como o segundo argumento é uma variável livre, a sua unificação com o corpo da cláusula fará com que a variável "X" assuma o valor "b,c,d". Na ocorrência de um retrocesso, o predefinido "clause" deve percorrer a base de dados a partir da cláusula que sucede a última cláusula analisada. No nosso exemplo, veremos que a próxima cláusula encontrada é a cláusula "a :- e,f,g", fazendo "X" valer "e,f,g". Torna-se importante então, saber qual a última cláusula analisada, de modo que soluções que já foram dadas não sejam repetidas na ocorrência de um retrocesso. Deste modo, a cada chamada do predefinido "clause" de-

vemos alterar o conteúdo da variável do contexto, de modo que esta aponte para a próxima cláusula a ser analisada.

Note que o conteúdo da posição da pilha global que contém a variável do contexto não se torna variável livre na ocorrência de um retrocesso, apesar de ter sido alterada após a criação do ponto de escolha do predefinido. Isso se explica pelo fato desta variável ser alterada dentro da função em C que implementa o predefinido sem que o seu endereço seja colocado na trilha. Assim, quando ocorre um retrocesso, o endereço desta variável não é encontrado na trilha e, por isso, ela não é desunificada, o que mantém inalterado o contexto da última chamada ao predefinido.

O ponto de escolha associado a um predefinido com retrocesso deve ser descartado através de uma chamada explícita à pseudo-instrução "trust_me_else_fail" dentro do corpo da função em C que implementa o predefinido em todos os pontos onde podem ser sinalizadas falhas. Para tornar mais fácil o entendimento do mecanismo de ativação dos predefinidos com retrocesso, vamos analisar um predefinido hipotético que recebe como argumento um número inteiro e que, a cada nova avaliação, imprime o valor da última chamada incrementada até que o valor seja maior ou igual que trinta.

```
int incrementa ()
{
     M_Objeto      arg;        /* Argumento do predicado        */
     int           ind;        /* Valor do índice               */
     M_Endereco    contexto;   /* Ponteiro p/ variável          */
                               /* do contexto                   */

                               /* Obtenção do argumento         */
     if (M_TIPO (A[1]) == ET_REF)
          arg = m_le_o (un_desref (M_VALOR (A[1])));
     else
          arg = A[1];
                               /* Verifica se tipo é válido     */
     if (M_TIPO (arg) != ET_INT)
     {
         i_tru_else_fail ();
         return (FALSE);
     }
                               /* Obtém endereço da variável    */
                               /* do contexto                   */
     contexto = un_desref(A[2]);
     if (M_TIPO (m_le_o (contexto)) == ET_REF)
                               /* Primeira chamada              */
         ind = m_valor_long (M_VALOR (arg));
     else
                               /* Chamada devido a um retrocesso*/
         ind = m_valor_long(M_VALOR (m_le_o (contexto)));

     ind++;
     printf ("Indice vale %d\n", ind);
                               /* Não há mais alternativas      */
     if (ind >= 30)
     {
         i_tru_else_fail ();
         return (FALSE);
     }
                               /* Guarda o contexto da chamada  */
                               /* atual para uso posterior      */
     m_esc_o (contexto, M_OBJ (ET_INT, ind));
     return (TRUE);
}
```

O primeiro passo na implementação de um predefinido com retrocesso é a obtenção dos argumentos e do contexto da última chamada. Para isso, devemos aplicar a função de desreferência sobre o conteúdo dos registradores que contém os argumentos (A[1] no exemplo) e sobre o conteúdo do registrador que aponta para a variável do contexto (A[2] no exemplo). No caso dos argumentos, o que nos interessa é o seu valor. Por isso, copiamos o conteúdo da desreferência para uma variável interna da função. Já no caso da

variável do contexto é importante termos o endereço da mesma, uma vez que ela deverá ser alterada mais tarde para conter o contexto da chamada atual. Assim, guardamos na variável "contexto" o endereço virtual desta variável que é retornado pela desreferência.

No nosso exemplo, o contexto da última chamada é representado por um número inteiro que pode ser mapeado em um único objeto. Entretanto, em predefinidos mais complexos, é necessário o uso de mais objetos para salvar o contexto da última chamada. Nesse caso, utilizam-se os objetos necessários a partir do topo corrente da pilha global, alterando-se o conteúdo da variável do contexto para que esta aponte para o início da área utilizada.

Finalmente, é importante observar que, em todos os pontos da função em que é sinalizada uma falha (return(FALSE)), temos uma chamada à pseudo-instrução "i_tru_else_fail" para descartar o ponto de escolha associado ao predefinido. Ao descartarmos o ponto de escolha, determinamos que o interpretador busque novas soluções em pontos de escolha mais antigos ou que este falhe pela falta de novas soluções alternativas.

3. Predefinidos para manipulação da tabela de operadores.

Os predefinidos para manipulação da tabela de operadores permitem ao usuário alterar a definição de operadores já existentes, criar novos operadores, consultar as definições que estão valendo em um dado momento e retornar a tabela ao estado inicial (dar um "reset" na tabela de operadores). As seções abaixo descrevem o funcionamento dos predefinidos do interpretador Pronus que implementam estas facilidades.

3.1 Predefinido op(Prec, Assoc, Op).

O predefinido op/3 é utilizado para alterar a tabela de operadores modificando definições já existentes ou criando novas definições. Este predefinido recebe três argumentos que qualificam a definição de um operador e que são descritos abaixo :

Precedência.

A precedência determina a ordem de avaliação dos operadores. Tomemos como exemplo a expressão "a + b * c". De acordo com a definição padrão da tabela de operadores no Pronus Prolog (a definição que está na tabela no momento em que o interpretador é ativado), a operação de multiplicação deve ser avaliada primeiro, isto é, o operador */2 tem precedência maior que o operador +/2. Deste modo, temos que a nossa expressão será traduzida na estrutura "+(a, *(b,c))".

A precedência de um operador no Pronus Prolog é representada por um número entre 0 e 1200 denominado nível de precedência. Quanto menor o número que representa nível de precedência de um operador, maior será a sua precedência. Assim, na definição padrão da tabela de operadores, o nível de precedência do operador */2 é menor que o nível de precedência do operador +/2. O nível de precedência de um operador no Pronus Prolog pode ser alterado pelo predefinido op/3 utilizando o argumento Precedência.

Se o usuário altera o nível de precedência de um operador para 0, este operador passa a ser inativo. Operadores com nível de precedência maior ou igual que 1000 não podem ser utilizados como

argumento de um funcional, a menos que estes venham contidos entre parênteses. O exemplo abaixo ilustra o problema :

```
(1) p :- f, g(a :- b, c).

(2) p :- f, g((a :- b, c)).
```

Na cláusula (1), o analisador sintático irá assinalar uso inválido de operador, já que o operador ":-" tem nível de precedência 1200. Na realidade, o que ocorre é que o analisador sintático não consegue decidir entre as estruturas "g(:-(a, ','(b,c)))" (vírgula encarada como funcional) e "g(:-(a,b), c)" (vírgula encarada como um separador dos argumentos) retornando um erro. Já no exemplo (2), a presença dos parênteses determinam uma chamada recursiva do analisador sintático, que passa a tratar a expressão "a :- b, c" independentemente do fato da mesma expressão ser argumento da predicado g/1. Deste modo, a ambigüidade é removida, pois se a expressão não é mais encarada como um argumento, a vírgula não pode mais ser um separador, devendo portanto ser tratada como um operador gerando a expressão "g(:-(a, ','(b,c)))".

Associatividade e Posição.

A posição determina a relação do operador com os seus operandos. Um operador pode ser prefixo (aparece antes do seu operando como em -3), posfixo (aparece após o seu operador como em 3!), ou infixo (aparece entre seus operadores como em 2 + 3). Já a associatividade determina a ordem em que operadores que têm mesma precedência são traduzidos. A associatividade de um operador pode ser da esquerda para a direita, da direita para a esquerda ou o operador pode ser não associativo, o que significa que operadores com a mesma precedência não podem aparecer na mesma expressão.

Em Prolog, a associatividade e posição de um operador são definidos por átomos predefinidos de acordo com a seguinte convenção :

- Os átomos são montados a partir das letras "x", "f" e "y" ; onde "x" e "y" representam os operandos e "f" representa o operador. A posição relativa das letras "x", "y" e "f" no átomo montado determinam se o operador é prefixo, pós-fixo ou infixo.

- As letras "x" e "y" determinam a associatividade do operador. Uma letra "x" indica que a expressão que estiver naquele lugar deverá ter precedência maior que a do operador, enquanto que uma letra "y" indica que a expressão que estiver naquele lugar pode ter uma precedência maior ou igual que a precedência do operador. A tabela abaixo descreve todas as combinações possíveis de átomos acompanhadas do seu significado quanto à associatividade e posição. No predefinido op/3, a associatividade e posição de um operador podem ser alterados pelo argumento Associatividade.

Átomo	Posição	Associatividade
xfx	Infixo	Não Associativo
xfy	Infixo	Direita ->Esquerda
yfx	Infixo	Esquerda ->Direita
fx	Prefixo	Não Associativo
fy	Prefixo	Direita ->Esquerda
xf	Pós-Fixo	Não associativo
yf	Pós-Fixo	Esquerda ->Direita

Nome do operador.

O último argumento do predefinido op/3 é o nome do operador. No caso em que o operador referenciado já existe, suas atribuições de precedência, associatividade e posição são alteradas. Entretanto, se o operador referenciado não está presente na tabela, então uma nova entrada é criada na tabela de operadores para conter o novo operador. Considere os exemplos abaixo que descrevem o funcionamento do predefinido op/3 em algumas situações interessantes.

Tabela antes	Comando	Tabela depois
(500, fx, +) (500, yfx, +)	op(500, xfx, +)	(500, fx, +) (500, xfx, +)

Tabela antes	Comando	Tabela depois
(400, yfx, *) -	op(500, fx, *)	(400, yfx, *) (500, fx, *)

Tabela antes	Comando	Tabela depois
-	op(500, fx, #@)	(500, fx, #@)

3.2 Predefinido current_op(Prec, Assoc, Op).

O predefinido current_op/3 é um predefinido com retrocesso que possibilita ao usuário consultar o conteúdo da tabela de operadores. Seu funcionamento consiste em percorrer a tabela de operadores buscando entradas que unifiquem com os argumentos que lhe foram passados. Havendo um retrocesso, o predefinido current_op/3 retorna a próxima entrada que unifica com os seus argumentos ou falha se nenhuma outra entrada foi satisfatória. No caso deste predefinido, a variável do contexto aponta para a última entrada analisada.

3.3 Predefinido reset_op.

O predefinido reset_op faz com que a tabela de operadores volte à situação em que ela estava no momento em que o interpretador foi ativado. Esta configuração é denominada definição padrão de operadores e está descrita na tabela abaixo :

Operador	Associatividade	Precedencia
:-	xfx	1200
:-	fx	1200
-->	xfx	1200
?-	fx	1200
->	xfx	800
/\	yfx	500
\/	yfx	500
:	xfy	500
+	yfx	500
-	yfx	500
+	fx	500
-	fx	500
*	yfx	400
/	yfx	400
>>	yfx	400
<<	yfx	400
mod	xfx	300
^	xfy	200
//	yfx	400
nospy	fy	900
spy	fy	900
not	fy	900
\+	fy	900
=	xfx	700
is	xfx	700
=..	xfx	700
\=	xfx	700
@<	xfx	700
@=<	xfx	700
@>	xfx	700
@>=	xfx	700
=\=	xfx	700
<	xfx	700
>	xfx	700
=<	xfx	700
>=	xfx	700
==	xfx	700
\==	xfx	700
=:=	xfx	700
;	xfy	1100
,	xfy	1000

3.4 Tabela de operadores e análise sintática.

A tabela de operadores determina a maneira pela qual o analisador sintático deve montar a estrutura que representa uma expressão envolvendo operadores. Do mesmo modo, quando esta estrutura é escrita, a tabela de operadores determina como converter

esta estrutura para a notação usual. O exemplo abaixo mostra como a definição de operadores é importante na análise das expressões :

(1) p :- f, g, #@(a,b).

(2) p :- f, g, a #@ b.

Se tentarmos introduzir a cláusula (2) na base de dados do interpretador sem alterar a definição padrão de operadores, obteremos um erro de associatividade de operador, de modo que apenas a cláusula (1) é válida. Entretanto, se alterarmos a tabela de operadores introduzindo a definição do operador infixo "#@/2" veremos que o analisador passa a aceitar ambas as cláusulas. Fica claro portanto, que a tabela de operadores influi diretamente na análise das expressões. Na próxima seção, veremos que as mudanças na análise de expressões têm efeitos sobre os predefinidos de avaliação e entrada e saída.

4. Predefinidos para impressão de termos.

O interpretador Pronus Prolog possui em sua biblioteca três predefinidos destinados à impressão de termos via saída padrão que são o write/1, writeq/1 e display/1.

O predefinido write/1 imprime termos realizando análise de operadores. A análise de operadores consiste em se consultar a tabela de operadores para o funcional de cada estrutura verificando se este funcional está definido como operador e imprimindo o termo de acordo com a definição encontrada. O código abaixo descreve como o predefinido write/1 realiza a análise de operadores.

```
int      p_nível;                /* Se p_nível é TRUE, então     */
                                 /* estamos imprimindo argumentos*/
                                 /* de uma estrutura ou lista    */


int p_write (termo)
M_Objeto    termo;               /* Termo a ser impresso         *.
{

    se termo é inteiro, real, átomo ou variável
        imprime (termo);
    senão se termo é cadeia
    {
        imprime ("$");
        imprime (cadeia);
        imprime ("$");
    }
    senão se termo é referência
    {
        M_Endereco    desref;  /* Desreferência                */
        M_Objeto      objeto;  /* Um objeto                    */

        desref = desreferência (termo);
        objeto = m_le_o (desref);

        se o tipo (objeto) é uma referência
            imprime ("_desref");
        senão
            p_write (objeto);
    }
```

```
senão se termo é lista
{
    imprime ("[");
    enquanto há termos na lista
    {
        objeto = próximo termo na lista;
        p_nível = TRUE;
        p_write (objeto);
    }
    p_nível = FALSE;
    imprime ("]");
}
senão se termo é uma estrutura
{
    int    s_nível;        /* Salva variável p_nível        */

    se a estrutura tem aridade um
    e o funcional é um operador prefixo
    {
        se precedência >= 1000 e p_nível = TRUE
        {
            s_nível = p_nível;
            p_nível = FALSE;
            imprime ("(");
        }
        senão
            s_nível = FALSE;

        se o argumento da estrutura é uma estrutura
        {
            se teste_de_parênteses para
            argumento dois deu verdadeiro
            {
                imprime ("operador(");
                p_write (argumento);
                imprime (")");
            }
            senão
            {
                imprime ("operador ");
                p_write (argumento);
            }
        }
        senão
        {
           imprime ("operador ");
           p_write (argumento);

        }

        se precedência >= 1000 e s_nível = TRUE
        {
           p_nível = s_nível;
           imprime (")");
        }

    }
```

```
senão se a estrutura tem aridade um
e o funcional é um operador posfixo
{
    se precedência >= 1000 e p_nível = TRUE
    {
        s_nível = p_nível;
        p_nível = FALSE;
        imprime ("(");
    }
    senão
        s_nível = FALSE;

    se o argumento da estrutura é uma estrutura
    {
        se teste_de_parênteses para
        argumento um deu verdadeiro
        {
            imprime (" (");
            p_write (argumento);
            imprime (")operador");
        }
        senão
        {
            p_write (argumento);
            imprime (" operador");
        }
    }
    senão
    {
        p_write (argumento);
        imprime (" operador");
    }

    se precedência >= 1000 e s_nível = TRUE
    {
        p_nível = s_nível;
        imprime (")");
    }

}
senão se estrutura tem aridade dois e
o funcional é operador infixo
{
    se precedência >= 1000 e p_nível = TRUE
    {
        s_nível = p_nível;
        p_nível = FALSE;
        imprime ("(");
    }
    senão
        s_nível = FALSE;
```

```
        se o primeiro argumento é uma estrutura
        {
            se teste_de_parênteses para
            argumento um deu verdadeiro
            {
                imprime (" (");
                p_write (argumento);
                imprime (")");
            }
            senão
                p_write (argumento);
        }
        senão
            p_write (argumento);

        imprime (" operador ");

        se o segundo argumento é uma estrutura
        {
            se teste_de_parênteses para
            argumento dois deu verdadeiro
            {
                imprime (" (");
                p_write (argumento);
                imprime (")");
            }
            senão
                p_write (argumento);
        }
        senão
            p_write (argumento);

       se precedência >= 1000 e s_nível = TRUE
       {
           p_nível = s_nível;
           imprime (")");
       }
    }
    senão                    /* Funcional não é operador      */
    {
        imprime ("funcional(");

        for (n = 1 ;; n = n + 1)
        {
            Seja arg o nésimo argumento da estrutura;
            p_nível = TRUE;
            p_write (arg);

            se n < aridade
                imprime (",");
            senão
                sai do laço;
        }
        p_nível = FALSE;

    }
}
```

```
int     teste_de_parênteses (precedência, associatividade,
                             precedência_argumento, narg )
int     precedência;            /* Precedência do operador        */
int     associatividade;        /* Associatividade do operador    */
int     narg;                   /* Indica se é primeiro ou        */
                                /* segundo argumento              */
{
    se precedência_argumento > precedência
        retorne (TRUE);
    senão se precedência_argumento == precedência
    {
        se narg == 1
            retorne ( associatividade == xf  ||
                      associatividade == xfx ||
                      associatividade == xfy );
        senão
            retorne ( associatividade ==  fx ||
                      associatividade == xfx ||
                      associatividade == yfx );
    }
    senão
        retorne (FALSE);
}
```

Se encontrarmos um operador com nível de precedência maior ou igual que o da vírgula (nível de precedência 1000) em uma expressão que é argumento de uma estrutura ou lista, devemos colocar a expressão entre parênteses. Para verificar esta condição, criamos a variável p_nível. Quando a variável p_nível está no nível lógico "TRUE" a rotina p_write está imprimindo o argumento de uma estrutura ou lista. Nestas condições, operadores com nível de precedência maior ou igual que 1000 determinam que a expressão seja colocada entre parênteses.

O predefinido writeq/1 difere do predefinido write/1 pelo fato que ele diferencia os átomos quotados dos não quotados. Já o predefinido display/1 difere dos predefinidos write/1 e writeq/1 pelo fato de não realizar análise de operadores ao imprimir uma estrutura. Deste modo, este predefinido limita-se a percorrer a estrutura imprimindo o funcional e os seus argumentos. Observe o exemplo abaixo, onde apresentamos as saídas dos predefinidos write/1, writeq/1 e display/1 dada a expresão "1 + '$' + 3" e considerando-se a definição padrão de operadores.

```
?- write(1 + '$' + 3).
1 + $ + 3

?- writeq(1 + '$' + 3).
1 + '$' + 3

?- display(1 + '$' + 3).
+(+(1,$), 3)
```

As semelhanças no funcionamento dos diversos predefinidos para impressão de termos levaram à implementação de uma única rotina de impressão com quatro modos de funcionamento. O modo P_WRITE habilita a análise de operadores mas não diferencia os átomos quotados e não quotados. Os modos P_WRITEQ e P_STRING habilitam a análise de operadores e diferenciam os átomos quotados dos não quotados. Entretanto, enquanto o modo P_WRITEQ escreve a sua saída em um arquivo, o modo P_STRING copia sua saída para uma estrutura de dados em memória que será utilizada pelo predefinido string_term/2 descrito na seção 3.4.1. Finalmente, temos o modo P_DISPLAY que desliga a análise de operadores e não diferencia os átomos quotados dos não quotados.

5. Predefinidos para manipulação de cadeias.

As cadeias de carateres são uma solução mais eficiente para a manipulação e armazenagem de texto por parte do Pronus Prolog. Esta implementação apresenta algumas vantagens sobre o uso de átomos e listas de caracteres que relacionamos a seguir :

- O uso de átomos para armazenar de texto implica em se instalar cada novo átomo na tabela de símbolos. Um número excessivo de átomos na tabela de símbolos resultará na degeneração do tempo de acesso aos símbolos, à medida em que ocorrerem colisões na função de espalhamento.
- As listas de caracteres gastam uma quantidade consideravelmente maior de memória para armazenar o mesmo texto que uma cadeia de caracteres. Além disso, ao serem impressas, estas listas de caracteres aparecem como listas de código ASCII o que dificulta a sua leitura.

5.1 Predefinido string_term(Cadeia, Termo).

O predefinido string_term/2 converte uma cadeia em um termo ou vice-versa. No caso em que é feita uma conversão de termo para cadeia, é importante que os operadores sejam tratados de acordo com a definição de operadores corrente. Deste modo, a rotina de impressão de termos é chamada com o modo de funcionamento P_STRINGi que trata os operadores e preenche uma estrutura de dados em memória com os caracteres gerados pela impressão do termo. A seguir, temos a definição da estrutura de dados que é preenchida pela rotina de impressão.

```
typedef struct p_encad P_Encad;

struct p_encad
{
    char        p_cad[100];     /* Área para conter caracteres    */
    P_Encad     *p_encad;       /* Ponteiro de encadeamento       */
}
```

```
typedef struct p_lista P_Lista;

struct p_lista
{
    int         lst_tam;        /* Número total de caracteres    */
    P_Encad     lst_cabeca;     /* Cabeca da lista encadeada     */
    P_Encad     *lst_rabo;      /* Último elemento da lista      */
};
```

É importante esclarecer que, caso o termo contenha variáveis livres, é necessário efetuar algumas alterações na representação do termo para que seja possível a sua conversão em cadeia. O problema reside no fato de que, durante a execução de um programa Prolog, as variáveis livres são apenas referências que apontam para si mesmas, enquanto que para efetuar a conversão de um termo em cadeia, é importante que as variáveis estejam associadas a símbolos. O exemplo abaixo ilustra mais claramente o problema da conversão de referências livres em símbolos.

```
?- string_term(X, livro(Npag, Ncap, Npag)).

X = $livro(A, B, A)$
```

Ao efetuarmos a consulta acima, disparamos o predefinido string_term/2 com dois argumentos, sendo que o primeiro é uma variável livre e o segundo é um termo (mais precisamente uma estrutura de aridade três). Ao se iniciar a execução do predefinido string_term/2, o segundo argumento pode ser representado pelo esquema abaixo :

```
            A[2]                    Pilha Global

+---------+--------+       +---------+--------+
| ET_ESTR |   *----+----->|      livro/3      |
+---------+--------+       +---------+--------+
                           | ET_REF  |    *   |<--+
                           +---------+--------+   |
                           | ET_REF  |    *   |   |
                           +---------+--------+   |
                           | ET_REF  |    *---+---+
                           +---------+--------+
```

Note que temos duas referências à variável livre "Npag" e uma referência à variável livre "Ncap" na estrutura livro/3 (as variáveis livres estão representadas pelos "*" sem setas) e que ao fornecermos a solução "X = $livro(A, B, A)$" estas variáveis estão associadas aos nomes A e B. Devemos então alterar a representação do termo antes de convertê-lo para cadeia de modo que esta associação entre variáveis e símbolos seja possível. Em um primeiro momento, devemos aplicar o algoritmo que transforma variáveis livres em variáveis código. Após este primeiro passo a nova estrutura tem a seguinte representação :

```
+---------+--------+
| ET_ESTR |    *---+---+
+---------+--------+   |
|      livro/3      |<--+
+---------+--------+
| ET_VAR  |    1   |
+---------+--------+
| ET_VAR  |    2   |
+---------+--------+
| ET_VAR  |    1   |
+---------+--------+
```

O segundo passo na alteração da representação do termo consiste em se converter as variáveis-código em variáveis-símbolo para que o termo possa ser impresso. Ao final do segundo passo, o nosso exemplo teria um esquema semelhante ao descrito abaixo se considerarmos que a variável "Npag" esteja associada ao símbolo "A" e que a variável "Ncap" esteja associada ao símbolo "B".

```
+---------+--------+
| ET_ESTR |    *---+---+
+---------+--------+   |
|      livro/3      |<--+
+---------+--------+
| ET_VAR  |    A   |
+---------+--------+
| ET_VAR  |    B   |
+---------+--------+
| ET_VAR  |    A   |
+---------+--------+
```

A partir deste momento, estamos prontos para gerar a impressão do termo em uma área de rascunho e converter esta impressão em uma cadeia de caracteres. No caso de uma conversão de

cadeia para termo, nós simplesmente alimentamos o analisador léxico com a cadeia a ser convertida em termo. A partir daí, tudo acontece como se o interpretador estivesse lendo um termo do teclado ou de um arquivo. Para o predefinido, resta apenas o trabalho de unificar o termo lido com o segundo argumento.

6. Predefinidos para avaliação de expressões.

Os predefinidos de avaliação são utilizados na manipulação de expressões matemáticas. As expressões são avaliadas por uma função única e os seus resultados são utilizados de acordo com o predefinido em questão. No caso do interpretador Pronus, podemos unificar o resultado de uma expressão aritmética com um objeto ou comparar os resultados de expressões aritméticas.

6.1 Predefinido is(Res, Exp).

O predefinido is/2 recebe dois argumentos sendo que o primeiro é um objeto que pode ser um inteiro, um real ou uma variável livre, enquanto que o segundo é uma estrutura. O predefinido avalia a estrutura como uma expressão matemática e unifica o resultado obtido com o seu primeiro argumento. Nos casos em que a estrutura recebida não pode ser avaliada, o primeiro argumento é unificado com o átomo "err", indicando um erro de avaliação. A sintaxe "Res is Exp" também é válida para o analisador de expressões pois o operador is infixo está presente na definição padrão de operadores.

A fim de implementar os predefinidos de avaliação, foi criada uma rotina única de avaliação. Esta rotina recebe como argumentos uma estrutura do Prolog e o endereço de uma estrutura da linguagem C que deverá conter o resultado da avaliação. A definição da estrutura que contém o resultado da avaliação está descrita abaixo :

```
struct av_retorno
{
    int    tipo;              /* Indica se o resultado é      */
                              /* inteiro ou real              */
    long   vint;              /* Armazena o valor inteiro     */
    double vreal;             /* Armazena o valor real        */
};
```

A criação de uma estrutura para conter o valor final da avaliação da expressão matemática é justificada pelo fato de que o tipo do resultado da avaliação (inteiro ou real) só é conhecido após a avaliação da mesma. Além disso, buscou-se uma implementação

que possibilitasse armazenar os resultados intermediários da avaliação sem que fosse necessário utilizar a memória virtual como área de rascunho, pois isso seria mais lento.

É importante notar que, como o usuário pode alterar as atribuições de um operador através do predefinido op/3, nem sempre a mesma expressão conduz a um mesmo resultado. Entretanto, não é o predefinido is/2 quem determina a diferenciação entre os resultados, e sim a montagem das estruturas por parte do analisador sintático. Considere o exemplo abaixo, onde avaliamos a mesma estrutura com definições diferentes de operadores.

```
avalia1 :- Y is 1 + 2 * 3,
           op(300, yfx, +),
           string_term('1 + 2 * 3', Z),
           W is Z.


Y = 7
W = 9


avalia2 :- Y is 1 + 2 * 3,
           op(300, yfx, +),
           W is 1 + 2 * 3.


Y = 7
W = 7
```

No primeiro exemplo, a cláusula do predicado avalia1/0 avalia duas estruturas montadas em momentos diferentes pelo analisador. A primeira é montada quando a cláusula é lida para a base de dados e a segunda é montada durante a sua execução pelo predefinido string_term/2. Neste caso, as estruturas montadas são diferentes, uma vez que as definições de operadores não eram as mesmas. O esquema abaixo mostra como são as estruturas montadas pelo sintático antes e depois da alteração das atribuições do operador "+/2" pelo predefinido op/3.

Antes.

```
+---------+--------+
| ET_ESTR |    *---+------+
+---------+--------+      |
|        +/2       |<-----+
+---------+--------+
| ET_INT  |    1   |
+---------+--------+
| ET_ESTR |    *---+------+
+---------+--------+      |
|        */2       |<-----+
+---------+--------+
| ET_INT  |    2   |
+---------+--------+
| ET_INT  |    3   |
+---------+--------+
```

Depois.

```
+---------+--------+
| ET_ESTR |    *---+------+
+---------+--------+      |
|        */2       |<-----+
+---------+--------+
| ET_ESTR |    *---+------+
+---------+--------+      |
| ET_INT  |    3   |      |
+---------+--------+      |
|        +/2       |<-----+
+---------+--------+
| ET_INT  |    1   |
+---------+--------+
| ET_INT  |    2   |
+---------+--------+
```

No segundo exemplo, a cláusula do predicado avalia2/0 avalia duas vezes a mesma estrutura, uma vez que ambas são montadas no momento em que a cláusula é lida para a base de dados. Neste caso, os dois resultados coincidem e a estrutura montada é análoga à do esquema abaixo.

```
+---------+--------+
| ET_ESTR |    *---+------+
+---------+--------+      |
|        +/2       |<-----+
+---------+--------+
| ET_INT  |    1   |
+---------+--------+
| ET_ESTR |    *---+------+
+---------+--------+      |
|        */2       |<-----+
+---------+--------+
| ET_INT  |    2   |
+---------+--------+
| ET_INT  |    3   |
+---------+--------+
```

Também é importante notar que, apesar de o usuário poder definir novos operadores através do predefinido op/3 o predefinido is/2 só reconhecerá como operadores aritméticos aqueles operadores que estiverem internamente definidos como tal. A lista a seguir apresenta os operadores reconhecidos pelos predefinidos de avaliação como operadores aritméticos e as funções matemáticas que podem ser utilizadas nas expressões aritméticas.

A + B	- Adição.
A - B	- Subtração.
A * B	- Multiplicação.
A / B	- Divisão.
A // B	- Divisão real.
-A	- Menos unário.
A /\ B	- Conjunção lógica.
A \/ B	- Disjunção lógica.
\(A)	- Inversão lógica.
A << B	- Deslocamento à esquerda.
A >> B	- Deslocamento à direita.
A mod B	- Resto da divisão.
abs(A)	- Valor absoluto.
acos(A)	- Arco-cosemo.
asin(A)	- Arco-seno.
atan(A)	- Arco-tangente.
cos(A)	- Coseno.
exp(A)	- Exponencial.

ln(A) - Logaritmo neperiano.

log(A) - Logaritmo na base 10.

sin(A) - Seno.

sqrt(A) - Raiz quadrada.

tan(A) - Tangente.

round(A,N) - A arredondado com N casas decimais. N é um inteiro entre 0 e 15.

7. Predefinidos de controle do fluxo da execução.

Os predefinidos de controle de fluxo interferem na seqüência de execução de um programa Prolog. As seções abaixo descrevem o funcionamento de alguns dos mais importantes predefinidos deste grupo.

7.1 Predefinido break/0.

O predefinido break/0 chama uma nova instância do módulo Mestre, suspendendo a execução corrente até que seja detectado um fim de arquivo. A título da exemplo, suponha que o interpretador esteja carregado com a seguinte base de dados :

```
a(1) :- write('Executando o predicado a/1'), nl.
b(2) :- write('Executando o predicado b/1'), nl.
```

Se dispararmos a consulta "?- a(X), break, b(Y)", teremos a seguinte saída :

```
Executando o predicado a/1
[Break]

??-
```

O símbolo "??-" indica que estamos em uma nova instância do interpretador com nível de recursão dois, o que pode ser inferido pelo número de interrogações (?) que aparecem no símbolo. A partir de agora, a execução da consulta "a(X), break, b(Y)" está suspensa até que a nova instância do interpretador seja desativada.

Para desativar uma instância recursiva do interpretador, basta sinalizar um fim de arquivo através do átomo predefinido end_of_file. No nosso exemplo, ao teclarmos end_of_file obteremos a seguinte resposta :

```
??- end_of_file.
[Fim do break]
Executando o predicado b/1

X = 1
Y = 2 ->

Sim.

?-
```

O símbolo "?-", indica que a instância principal do interpretador está novamente ativa. Note que a execução da consulta "a(X), break, b(Y)", foi retomada a partir do ponto em que ela havia parado e terminada com sucesso. Assim, quando uma execução é suspensa, diversas informações devem ser guardadas para que seja possível restaurar o contexto mais tarde.

Vamos analisar o exemplo anterior em que disparamos a consulta "a(X), break, b(Y)". Ao ler a estrutura que representa o objetivo, os módulos de análise associam a variável "X" à posição zero da tabela de variáveis e a variável "Y" à posição um. O comando break suspende a execução da consulta e chama uma nova instância do interpretador, a qual reinicializa a tabela de variáveis. Uma vez abortada a segunda instância do interpretador através do átomo predefinido end_of_file, o controle é retornado à instância principal do Mestre e a execução, que antes estava suspensa, segue o seu curso até que a trap sucesso imprima as soluções. O corpo da função que implementa a trap sucesso está descrito pelo código abaixo :

```
int p_sucesso ()
{

    Se existe alguma variável da cláusula não anônima
    {
        Para todas as variáveis da cláusula faça
        {
            Se a variável não é anônima
            {
                Imprime o nome da variável;
                Imprime conteúdo da variável;
            }
        }

        Leia um caracter
        Se caracter é um ponto e vírgula
            Dispara um retrocesso retornando FALSE;
        senão
            Imprime "Sim." e termina a execução;
    }
    senão
        Imprime "Sim." e termina a execução;

}
```

Os nomes das variáveis estão inicialmente armazenados no campo "tva_ptsimb" da tabela de variáveis. Note que ao retomarmos a execução da consulta, a informação relativa aos nomes das variáveis não está mais disponível na tabela de variáveis, uma vez que esta tabela já foi reinicializada. Além disso, os registros da máquina Prolog também estão alterados. Deste modo, torna-se necessário salvar estas informações em estruturas de dados à parte. A estrutura de dados que é utilizada para guardar o nome das variáveis de uma consulta suspensa está definida abaixo :

```
typedef struct tv_salva Tv_Salva;

struct tv_salva
{
    int         tvs_nsexp;      /* Número de explícitas salvas  */
    int         tvs_nanon;      /* Número de anônimas salvas    */
    Ts_Simbolo **tvs_salva;     /* Aponta p/ área onde estão    */
                                /* explícitas                   */
    Tv_Salva    *tvs_prox;      /* Aponta p/ o próximo nó       */
};
```

O campo tvs_nsexp guarda o número de variáveis salvas. O campo tvs_anon guarda o número de variáveis anônimas que estão contidas na consulta. O campo tvs_salva aponta para um vetor alocado dinâmicamente que contém ponteiros para os nomes de todas as variáveis da consulta. O ponteiro tvs_prox implementa uma lista encadeada destas estruturas onde cada instância do Mestre gera um novo nó nesta lista. A cabeça da lista guarda as informações referentes às variaveis da instância atual do Mestre. O esquema abaixo descreve a situação da lista encadeada após duas chamadas recursivas ao módulo Mestre :

```
+----------------+                          +---------------+
|     Cabeça     |                          |     Rabo      |
+----------------+                          +---------------+
        |                                          |
        v                                          v
+----------------+   +----------------+   +---------------+
|   Variáveis    |   |   Variáveis    |   |   Variáveis   |
|    salvas      |-->|    salvas      |-->|    salvas     |
|    Mestre      |   |    Mestre      |   |    Mestre     |
|    nível 3     |   |    nível 2     |   |   principal   |
+----------------+   +----------------+   +---------------+
```

A estrutura de dados utilizada para salvar os registros da máquina Prolog está descrita abaixo :

```
typedef struct me_regs Me_Regs;

struct me_regs
{
    M_Endereco    mer_CP;      /* Registro CP                  */
    M_Endereco    mer_E;       /* Registro E                   */
    M_Endereco    mer_B;       /* Registro B                   */
    M_Endereco    mer_CR;      /* Registro CR                  */
    M_Endereco    mer_TR;      /* Registro TR                  */
    M_Endereco    mer_H;       /* Registro H                   */
    M_Endereco    mer_HB;      /* Registro HB                  */
};
```

Cada campo desta estrutura armazena um dos registros da máquina Prolog. Toda vez que o Mestre é ativado o valor atual dos registradores é armazenado em uma variável local deste tipo. Toda instância do Mestre restaura o conteúdo dos registradores ao final de sua ativação, recuperando o contexto da ativação anterior.

Note que utilizamos duas estruturas de dados distintas para salvar as informações necessárias à retomada de uma execução suspensa. O mecanismo que salva estas informações foi implementado desta maneira porque enquanto o nome das variáveis deve ser uma informação acessível a módulos do interpretador que estão fora do escopo do módulo Mestre, os registros salvos da máquina Prolog são uma informação que só deverá ser acessada ao final de uma ativação recursiva do Mestre, podendo portanto ficar guardada entre as variaveis locais ao Mestre.

7.2 Predefinidos call/1 e not/1.

Os predefinidos call/1 e not/1 recebem como argumento estruturas que representam objetivos a serem executados. O predefinido call/1 gera código para a cláusula fornecida como argumento e posiciona o ponteiro P para o início do código gerado, disparando uma execução. Já o predefinido not/1 gera o código associado à estrutura e um código de interface entre o predefinido e o módulo de execução. Este código de interface determina que o predefinido not/1 termine com sucesso se a execução do código do objetivo falhar ou falhe se a mesma execução terminar com sucesso. As seções abaixo descrevem em detalhes o funcionamento destes predefinidos.

call(Objetivo).

Conforme já vimos, o predefinido call/1 recebe como argumento uma estrutura que representa um objetivo a ser executado. Para executar este objetivo, devemos gerar o código correspondente à estrutura fornecida como argumento e posicionar o registro P para o início do código gerado.

A estrutura do objetivo representa as variáveis como referências e não está compatibilizada. Para gerar o código do objetivo, precisamos de uma estrutura compatibilizada em que as variáveis estejam associadas a posições na tabela de variáveis. Deste modo, antes de compatibilizar a estrutura do objetivo, devemos submetê-la ao algoritmo que converte variáveis livres em variáveis-código. Considere o exemplo abaixo, em que disparamos a consulta "?- call(a(X,Y))", onde as variáveis "X" e "Y" estão livres.

Ao iniciar a execução do predefinido, o registro A[1] aponta para a posição na pilha global onde está localizada a estrutura que representa o objetivo. As variáveis da consulta "X" e "Y" estão associadas a posicões da pilha local alocadas pelo código que dispara a consulta. O esquema abaixo representa a situação inicial. Os endereços são fictícios, utilizados apenas para ilustrar o exemplo.

```
                Registro A[1]
          +---------+--+----------------+
          | ET_ESTR |xx|     1058       |
          +---------+--+----------------+

          Corpo da estrutura (PILHA GLOBAL)
          +-------------------------------+
     1058 |            a/2                |
          +-------------------------------+
     1060 | ET_REF  |xx|     1060       |  X
          +---------+--+----------------+
     1062 | ET_REF  |xx|     1062       |  Y
          +---------+--+----------------+

        Variáveis da consulta (PILHA LOCAL)
          +---------+--+----------------+
     2080 | ET_REF  |xx|     1060       |
          +---------+--+----------------+
     2082 | ET_REF  |xx|     1062       |
          +---------+--+----------------+
```

Ao convertermos as variáveis livres "X" e "Y" em variáveisi-código, os registradores A[1] e A[2] passam a apontar para as posições 1060 e 1062 respectivamente. A estrutura compatibilizada será "gol(X,Y) :- a(X,Y)".

A associação realizada entre as variáveis livres da estrutura convertida e os registros A's da máquina Prolog, substitui o código que posiciona os parâmetros de um objetivo antes de sua chamada. No caso em que o objetivo está inserido no fluxo normal da execução, seus argumentos são sempre posicionados pelo código da cláusula que o está chamando. Entretanto, no caso de uma chamada feita pelo predefinido call/1 devemos simular o código que posicionaria os parâmetros, associando os registros As às variáveis livres da estrutura do objetivo.

A título de exemplo, suponha que a cláusula "a(1,2) :- true" seja a única cláusula do predicado a/2. Ao executarmos este predicado, as variáveis da consulta "X" e "Y" serão

unificadas com os inteiros "1" e "2" através dos registros A[1] e A[2]. Ao final da consulta a trap sucesso acessa as posições do ambiente associadas às variaveis da consulta e a estrutura Tv_Salva, fornecendo as mensagens "X = 1" e "Y = 2". O esquema abaixo mostra a situação após a chamada ao predicado a/2.

```
                    Registro A[1]
          +---------+--+-----------------+
          | ET_REF  |xx|      1060       |
          +---------+--+-----------------+

                    Registro A[2]
          +---------+--+-----------------+
          | ET_REF  |xx|      1062       |
          +---------+--+-----------------+

          Corpo da estrutura (PILHA GLOBAL)
          +-------------------------------+
          |             a/2               |
          +-------------------------------+
     1060 | ET_INT  |xx|        1         |
          +---------+--+-----------------+
     1062 | ET_INT  |xx|        2         |
          +---------+--+-----------------+

        Variáveis da consulta (PILHA LOCAL)
          +---------+--+-----------------+
     2080 | ET_REF  |xx|      1060       |
          +---------+--+-----------------+
     2082 | ET_REF  |xx|      1062       |
          +---------+--+-----------------+
```

not(Objetivo).

O predefinido not/1 funciona de maneira análoga ao predefinido call/1. A única diferença reside no fato que este predefinido gera um código de interface que determina uma falha se o objetivo suceder ou termina a execução com sucesso se o objetivo falhar. O código de interface está descrito abaixo :

```
(1)      try_me_else    R,0
(2)      allocate
(3)      call_goal      ENDER,NEXP
(4)      cut
(5)      deallocate
(6)      fail
(7) R :  trust_me_else_fail
(8)      proceed
```

Início da execução.

(1) - Cria um ponto de escolha
(2) - Cria um ambiente
(3) - Dispara a execução do objetivo

Término com sucesso.

(4) - Descarta todos os pontos de escolha que estão abaixo do registro CR. Neste momento o registro CR está acima do ponto de escolha criado pela intrução (1), de modo que ele é descartado.
(5) - Descarta o ambiente criado pela instrução (2).
(6) - Causa uma falha.

Término com falha.

(7) - Descarta o ponto de escolha criado pela instrução (1).
(8) - Executa o retorno de uma instrução call.

8. Predefinidos de entrada e saída em arquivos.

Os predefinidos de entrada e saída de arquivos implementam uma série de mecanismos que possibilitam a manipulação de arquivos do usuário e redirecionamento dos arquivos padrão de entrada e saída do Pronus. As seções abaixo descrevem como é feita a gerência de arquivos no Pronus.

8.1 Gerência de arquivos para o padrão Edimburgo.

Como já foi dito anteriormente, os predefinidos de entrada e saída do padrão de Edimburgo utilizam as entradas da tabela de nomes para guardar as informações relativas aos arquivos que estão associados a átomos.

Os campos taa_simb e taa_desc realizam a associação entre um átomo e uma entrada da tabela de descritores. Inicialmente, estas variáveis apontam para as entradas relativas ao arquivo padrão de entrada e saída associado ao símbolo "user". Já os campos taa_estado e taa_pos são utilizados no reconhecimento do estado do arquivo que está sendo acessado. Abaixo nós definimos todos os possíveis estados de um arquivo acessado pelos predefinidos de entrada e saída do padão de Edimburgo e as ações associadas a cada um destes estados.

TA_INEXISTENTE : Indica que o arquivo deve ser criado e instalado na tabela de nomes e na tabela de descritores.

TA_FECHADO : Indica que o arquivo não possui mais uma entrada na tabela de descritores.

TA_LEITURA : Indica que o arquivo é o atual de entrada padrão do interpretador.

TA_ESCRITA : Indica que o arquivo é o atual de saída padrão do interpretador.

TA_LEI_HIB : Indica que o arquivo está na tabela de descritores e que a última posição lida deste arquivo está anotada no campo taa_pos da tabela de nomes.

TA_ESC_HIB : Indica que o arquivo já foi utilizado como arquivo padrão de saída devendo portanto ser reaberto com "a" (append).

TA_LE_ESC : Indica que o mesmo arquivo é o arquivo padrão de entrada e saída, aberto com "w+" ou "a+".

TA_LEH_ESC : Indica que o arquivo já foi utilizado como o arquivo padrão de entrada e que este arquivo é o arquivo atual de saída.

TA_LE_ESCH : Indica que o aquivo já foi utilizado como o arquivo padrão de saída e que este arquivo é o arquivo atual de entrada.

TA_LEH_ESCH : Indica que o arquivo já foi utilizado como arquivo padrão de entrada e como arquivo padrão de saída, devendo ser reaberto com "a+".

8.2 Predefinidos tell/1, telling/1 e told/0.

O predefinido tell/1 recebe como argumento um átomo e redireciona a saída padrão do Pronus para o arquivo associado a este átomo. O pseudo-código abaixo descreve claramente o funcionamento deste predefinido.

```
int     p_tell ()
{
        Seja arq o arquivo passado como parâmetro.

        Se (arq != saida padrão atual)
        {
            Se (saida padrão atual != arquivo "user")
            {
                                    /* Trata o fechamento do arquivo */
                                    /* padrao de saida atual         */
                switch (estado do arquivo atual)
                {
                    case TA_ESCRITA:

                        fecha (atual);
                        estado de atual = TA_ESC_HIB;

                    case TA_LEI_ESC:

                        estado de atual = TA_LE_ESC;

                    case TA_LEH_ESC:

                        fecha (atual);
                        estado de atual = TA_LEH_ESCH;

                }
            }
```

```
        Se (arq != user)
        {
                                /* Trata estado do novo arquivo */
                                /* de saída padrão              */
            switch (estado de arq)
            {
                case TA_INEXISTENTE:

                    pa = fopen (arq, "w");
                    instala(pa);
                    estado de arq = TA_ESCRITA;

                case TA_FECHADO:

                    pa = fopen (arq, "w");
                    instala(pa);
                    estado de arq = TA_ESCRITA;

                case TA_LEITURA:

                    fclose (arq);
                    pa = fopen(arq, "w+");
                    instala(pa);
                    estado de arq = TA_LE_ESC;

                case TA_LEI_HIB:

                    pa = fopen (arq, "w+");
                    instala(pa);
                    estado de arq = TA_LEH_ESC;

                case TA_ESC_HIB:

                    pa = fopen (arq, "a");
                    instala(pa);
                    estado de arq = TA_ESC;

                case TA_LE_ESCH:

                    estado de arq = TA_LE_ESC;

                case TA_LEH_ESCH:

                    pa = fopen(arq, "a+");
                    instala(pa);
                    estado de arq = TA_LEH_ESC;

            }
        }

        saída padrão = arq;

    }

    return (TRUE);

}
```

O predefinido telling/1 consulta variáveis internas do interpretador para descobrir qual é o átomo associado ao arquivo de saída atual. Este átomo é então unificado com o argumento do predefinido informando ao usuário qual é o arquivo de saída corrente.

O predefinido told/0 fecha o arquivo padrão de saída aberto por tell/1, atualiza o estado deste arquivo e redireciona a saída padrão para o arquivo user. Sempre que um arquivo é fechado pelo predefinido told/0 o seu símbolo é mantido na tabela de nomes, de modo que uma nova ativação dos predefinidos do padrão de Edimburgo para o mesmo átomo leve em consideração o estado deste arquivo.

8.3 Predefinidos see/1, seeing e seen/0.

O predefinido see/1 funciona de maneira análoga ao predefinido tell/1 com a diferença que ele redireciona a entrada padrão do interpretador e não a saída. Deste modo, quando ativamos o predefinido see/1, o analisador léxico é redirecionado para um novo arquivo passando a ler o seu conteúdo.

O predefinido seen/0 fecha o arquivo aberto por see/1, atualiza o seu estado e redireciona o analisador léxico para o arquivo padrão de saída "user".

O predefinido seeing/1 funciona de maneira idêntica ao predefinido telling/1. A única diferença é que este predefinido informa ao usuário qual o átomo associado ao arquivo que está sendo lido pelo léxico, ao invés de informar sobre o átomo associado ao arquivo em que está sendo escrita a saída.

8.4 Predefinidos stdout/2, stdin/2 e stdinout/3.

Os predefinidos stdout/2, stdin/2 e stdinout/3 são uma alternativa ao uso dos predefinidos de entrada e saída do padrão de Edimburgo, sendo utilizados para redirecionar os arquivos padrões de entrada e saída durante a execução de um objetivo dado.

stdout(Objetivo, Arquivo).

O predefinido stdout/2 recebe como argumentos uma estrutura que representa um objetivo a ser executado e o nome de um arquivo. Este arquivo será aberto para escrita (ou criado se não existir) e utilizado como saída padrão do interpretador durante a execução do objetivo. O pseudo-código abaixo decreve o funcionamento deste predefinido :

```
p_stdout ()
{
    Pega o nome do arquivo;
    Abre o arquivo para escrita ou cria se ele não existe;
    Pega a estrutura que representa o objetivo;
    Gera o código associado a esta estrutura;
    Salva os registros H, CP e P;
    Salva o arquivo de saída padrão atual;
    Redireciona a saída padrão para o novo arquivo;
    Dispara a execução do objetivo através da função ex_stdexec;
    Restaura o antigo arquivo de saída padrão;
    Restaura os registros H, CP e P;
    ex_fim = FALSE;
    Retorne o valor da variável p_stdsucesso;
}
```

A geração do código associado à estrututura do objetivo é análoga à geração de código realizada pelo predefinido call/1, não sendo necessário descrever o processo novamente.

A execução do código associado ao objetivo é gerenciada pela função ex_stdexec. Esta função monta um código de interface entre o interpretador e a execução. Ao final do objetivo este código de interface determina um retorno ao ponto onde a execução foi disparada, a fim de que o antigo arquivo de saída do interpretador seja restaurado. Note que em outros predefinidos que geram código a tempo de execução, como o call/1, não há essa preocupação. Nestes predefinidos, o código é gerado, o ponteiro P é posicionado e a função retorna, deixando que a execução siga o seu curso. O algoritmo abaixo descreve o corpo da função ex_stdexec.

```
int   ex_stdexec (código)
M_Endereço   código;            /* Código do objetivo          */
{
    Monta o código de interface;
    Posiciona o registro P no início do código de interface;
    Chama a função ex_exec ();
    Retorne (p_stdresultado);
}
```

O esquema abaixo descreve o código de interface montado pela função p_stdexec

```
 (1)          try_me_else  R,0
 (2)          continue     0
 (3)          allocate
 (4)          call_goal    código,NVAR
 (5)          cut
 (6)          deallocate
 (7)          trust_me_else_fail
 (8)          trap         3
 (9)      R:  trust_me_else_fail
(10)          trap         4
```

Início da execução.

(1) - Cria um ponto de escolha.

(2) - Posiciona o registro CR no ponto de escolha criado pela instrução (1).

(3) - Cria um ambiente

(4) - Dispara a execução do objetivo.

Término com sucesso.

(5) - Descarta todos os pontos de escolha abaixo do registro CR.

(6) - Descarta o ambiente criado pela instrução (3).

(7) - Descarta o ponto de escolha criado pela instrução (1).

(8) - Dispara a trap 3.

Término com falha.

A ocorrência de uma falha no objetivo da consulta, descarta todos os pontos de escolha localizados abaixo daquele criado pela instrução (1).

(9) - Descarta o ponto de escolha criado pela instrução (1).

(10) - Dispara a trap 4.

O pseudo código abaixo descreve as funções chamadas pelas instruções "trap 3" (p_stdsucesso) e "trap 4" (p_stdfalha).

```
int   p_stdsucesso
{
    ex_fim = TRUE;
    p_stdresultado = TRUE;
}

int   p_stdfalha
{
    ex_fim = TRUE;
    p_stdrsultado = FALSE;
}
```

A função ex_exec é chamada pela função p_stdexec, sendo responsável pela gerência da execução do código do objetivo, avançando o registro P até que que seja sinalizado o fim deste código através da alteração da variável de controle ex_fim, tal como está descrito nas funções p_stdsucesso e p_stdfalha (ex_fim = TRUE). Deste modo, nós garantimos que ao final da execução do código do objetivo o controle será retornado à função p_stdout, onde serão restaurados o arquivo de saída e os registradores P, CP e H da máquina virtual Prolog [Souza 88]. A partir deste ponto o interpretador sinaliza que a execução deverá prosseguir normalmente (ex_fim = TRUE). A variável p_stdresultado informa se o objetivo sucedeu ou falhou.

stdin(Objetivo, Arquivo).

O predefinido stdin/2 funciona de maneira idêntica ao predefinido stdout/2. A única diferença entre estes dois predefinidos é que, enquanto stdout/2 redireciona o arquivo de saída padrão, o predefinido stdin/2 redireciona o arquivo de entrada padrão.

stdinout(Objetivo, Arquivo_in, Arquivo_out).

O predefinido stdinout/3 recebe como argumentos o nome de dois arquivos e um objetivo. Durante a execução do objetivo os arquivos padrão de entrada e saída do interpretador são simultâneamente redirecionados.

9. Conclusão.

Esta primeira parte do trabalho de implementação dos predefinidos já garante uma série de facilidades que podem ser utilizadas na aplicação do usuário, proporcionando um rasoável ambiente de programação.

A parte final do trabalho de implementação prevê apenas mais dois grupos de predefinidos que são os predefinidos para manipulação da base dados e coleta de soluções.

Os predefinidos para manipulação da base de dados salvam em disco as cláusulas presentes na base de dados do interpretador para posterior consulta, o que garante a manipulação de programas de uma forma muito mais confortável. Já os predefinidos para coleta de soluções são uma poderosa ferramenta de programação Prologi, definidos como uma extensão à linguagem [Clocksin 81].

10. Referências bibliográficas.

[Warren 83] David H. D. Warren,
An Abstract instruction set,
Tecnical note 309,
SRI International,
Menlo Park, California, Outubro de 1983.

[Souza 88] Pseudo-Instruções Prolog e seu ambiente de execução,
Luiz Fernando Pereira de Souza,
Valério Machado Dallolio,
Relatório Técnico NCE/UFRJ, Rio de Janeiro - RJ,
outubro de 1988,
código NCE-12/88, Parte I 29p. e parte II 40p.

[Clocksin 81] Programming in Prolog,
W. F. Clocksin,
C. S. Mellish,
Springer-Verlag,
Berlin Heidelberg, 1981, 281p.

[Dallolio 89] Pronus Prolog - Arquitetura e implementação,
Valério Machado Dallolio,
Luiz Fernando Pereira de Souza,
Relatório Técnico NCE/UFRJ, Rio de Janeiro - RJ,
março de 1989,
código NCE-01/89, 24p.

[Dallolio 91] Ambiente de execução do Pronus Prolog,
Luciano Conrado Machado Dallolio,
Luiz Fernando Pereira de Souza,
Relatório Técnico NCE/UFRJ, Rio de Janeiro - RJ,
código NCE-10/91, 41p