

8 Referências Bibliográficas

1. LEE, PETER. Peter Lee: **Proof-Carrying Code**. 3 p. Disponível em <http://www.cs.cmu.edu/~petel/papers/pcc/pcc.html>. Acesso em 07 de agosto de 2002.
2. IMPERIAL, JULIANA CARPES. **Correção de Programas**. Monografia (Projeto Final de Programação - INF 2102) - Departamento de Informática da PUC-Rio. Rio de Janeiro, 31 de julho de 2002. 56 p. Disponível em <http://www.inf.puc-rio.br/~juliana/trabalhos/monografia.zip>. Acesso em 24 de março de 2003.
3. NECULA, GEORGE C. Proof-Carrying Code. In Symposium on Principles of Programming Languages (POPL' 97), Paris, França, janeiro de 1997. **Proceedings**. p 106 - 119. Disponível em <http://www.cs.cmu.edu/~petel/papers/pcc/pcc-popl97.ps>. Acesso em 07 de agosto de 2002.
4. APPEL, ANDREW W.; FELTEN, EDWARD W.; SHAO, ZHONG. **Scaling Proof-Carrying Code to Production Compilers and Security Policies**. Princeton University e Yale University, janeiro de 1999. 19 p. Disponível em <http://www.cs.princeton.edu/sip/projects/pcc/whitepaper/>. Acesso em 07 de agosto de 2002.
5. SLONNEGER, KENNETH; KURTZ, BARRY L. **Formal Syntax and Semantics of Programming Languages**. A Laboratory Based Approach. Addison-Wesley, 1995. 637 p. Disponível em <http://www.cs.uiowa.edu/~slonnegr/plf/Book/>. Acesso em 19 de março de 2003.
6. SLONNEGER, KENNETH. Fontes dos exemplos do livro Formal Syntax and Semantics of Programming Languages. Disponível em <ftp://ftp.cs.uiowa.edu/pub/slonnegr/plf/>. Acesso em 19 de março de 2003.

7. FISHER, J. R. **Prolog Tutorial**. Disponível em http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.htm. Acesso em julho de 2001.
8. MENZIES, TIM. **Prolog Tutorial**. Disponível em <http://www.ece.ubc.ca/~elec415/prologtut.html>. Acesso em julho de 2001.
9. WIELEMAKER, JAN. **SWI-Prolog 3.4 Reference Manual**. SWI - University of Amsterdam. Amsterdam, The Netherlands, setembro de 2000. 221 p. Disponível em <http://www.swi.psy.uva.nl/projects/SWI-Prolog/>. Acesso em outubro de 2000.
10. ROCHA, HELDER. **Como criar a sua Home Page**. HTML. IBPI Press, Rio de Janeiro, 1996. 183 p.
11. NADATHUR, GOPALAN. The Metalanguage λ Prolog and its Implementation. In Symposium on Functional and Logic Programming, Tóquio, Japão, março de 2001. **Proceedings**. p 1 - 20. Disponível em <http://www-users.cs.umn.edu/~gopalan/papers/flops.pdf>. Acesso em 19 de agosto de 2002.
12. **Proof-Carrying Code**. Department of Computer Science - Princeton University. 2 p. Disponível em <http://www.cs.princeton.edu/sip/projects/pcc/>. Acesso em 07 de agosto de 2002.
13. APPEL, ANDREW W.; MCALLESTER, DAVID. **An Indexed Model of Recursive Types for Foundational Proof-Carrying Code**. Princeton University e AT&T Research. 21 de novembro de 2000. 17 p. Disponível em <ftp://ftp.cs.princeton.edu/techreports/2000/629.pdf>. Acesso em 07 de agosto de 2002.
14. APPEL, ANDREW W. Foundational Proof-Carrying Code. In Symposium on Logic in Computer Science, Boston, Massachusetts, junho de 2001. **LICS'01**. 10 p. Disponível em <http://www.cs.princeton.edu/~appel/papers/fpcc.pdf>. Acesso em 07 de agosto de 2002.

15. APPEL, ANDREW W.; SWADI, KEDAR N.; VIRGA, ROBERTO. Efficient Substitution in Hoare Logic Expressions. In International Workshop on Higher-Order Operational Techniques in Semantics (HOOTS 2000), Montreal, setembro de 2000. **Proceedings**. p 35 - 50. Disponível em <http://www.cs.princeton.edu/~appel/papers/subst.pdf>. Acesso em 07 de agosto de 2002.
16. PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO. Vice-Reitoria para Assuntos Acadêmicos. **Pós-Graduação PUC-Rio**. Normas para apresentação de teses e dissertações. PUC-Rio. Rio de Janeiro, 2001. 80 p. Disponível em http://www.dbd.puc-rio.br/depto_dbd/normas_pos.pdf. Acesso em 08 de outubro de 2002.
17. **Static Single-Assignment Form**. 2 p. Disponível em <http://www.cag.lcs.mit.edu/~cananian/Projects/ele580a/node9.html>. Acesso em 19 de setembro de 2002.
18. LEE, PETER; NECULA, GEORGE. **Research on Proof-Carrying Code for Mobile-Code Security**. In: Darpa Workshop on Foundations for Secure Mobile Code, Monterey, California, março de 1997. CMU. Pittsburgh, Pennsylvania, março de 1997. 6 p. Disponível em <http://www-2.cs.cmu.edu/~petel/papers/pcc/pcc-mobile.ps>. Acesso em 25 de setembro de 2002.
19. GUPTA, ANUPAM. **Proof-Carrying Code: A Survey**. 10 p. Disponível em http://www.cse.ucsc.edu/~abadi/UCB_CS263/projects/gupta.ps. Acesso em 01 de outubro de 2002.
20. NOGIN, ALEKSEY. **A Review of Theorem Provers**. Fevereiro de 2002. 12 transparências: coloridas. Disponível em http://www.cs.cornell.edu/Nuprl/PRLSeminar/PRLSeminar01_02/Nogin/PRLSeminar7b.pdf. Acesso em 01 de outubro de 2002.
21. DENTON, WILLIAN. **Gödel's Incompleteness Theorem**. 02 de julho de 2002. 4 p. Disponível em <http://www.miskatonic.org/godel.html>. Acesso em 03 de outubro de 2002.

22. NECULA, GEORGE; LEE, PETER. Safe Kernel Extensions Without Run-Time Checking. In Symposium on Operation Systems Design and Implementation, Seattle, Washington, outubro de 1996. **Proceedings**. p 229 - 243. Disponível em <http://www-2.cs.cmu.edu/~petel/papers/pcc/pcc-osdi96.ps>. Acesso em 25 de setembro de 2002.
23. FELTY, AMY. **Semantic Models of Types and Machine Instructions for Proof-Carrying Code**. In PCC 2000, Santa Barbara, EUA, junho de 2000. 27 transparências: coloridas. Disponível em <http://www.research.att.com/~trevor/PCC2000/slides/Felty.pdf>. Acesso em 07 de agosto de 2002.
24. PCC 2000 - WORKSHOP ON PROOF-CARRYING CODE, University of California, Santa Barbara, EUA, junho de 2000. 1 p. Disponível em <http://www.research.att.com/~trevor/PCC2000/>. Acesso em 07 de agosto de 2002.
25. FLOYD, ROBERT W. Assigning Meanings to Programs. **Mathematical Aspects of Computer Science**. American Mathematical Society, 1967. p 19 - 32. Disponível em <http://raw.cs.berkeley.edu/Papers/FloydMeaning.pdf>. Acesso em 09 de outubro de 2002.
26. PCC 2000 - WORKSHOP ON PROOF-CARRYING CODE, University of California, Santa Barbara, EUA, junho de 2000. Panel: **The Future of Proof-Carrying code**. 4 p. Disponível em <http://www.research.att.com/~trevor/PCC2000/panel.html>. Acesso em 07 de agosto de 2002.
27. HOARE, C. A. R. An Axiomatic Basis for Computer Programming. **Communications of the ACM**. Outubro de 1969. 12. p 576 - 580. Disponível em <http://www.cis.temple.edu/~wolfgang/cis551/Hoare-CACM-6910.PDF>. Acesso em 09 de outubro de 2002.
28. CORMEN, THOMAS H.; LEISERSON. CHARLES E.; RIVEST, RONALD L. **Introduction to Algorithms**. The MIT Press, England, 1990. 1028 p.

29. HOARE, C. A. R. **An Axiomatic Basis for Computer Programming**. 13 transparências: coloridas. Disponível em <http://www.cs.stir.ac.uk/~twi/formalmethods/cs266/hoare69.pdf>. Acesso em 09 de outubro de 2002.
30. MENEZES, PAULO BLAUTH; HAEUSLER, EDWARD HERMANN. **Teoria das Categorias para Ciência da Computação**. Editora Sagra Luzzatto, Instituto de Informática da UFRGS, Porto Alegre, 2001. 324 p.
31. DALEN, DIRK VAN. **Logic and Structure**. Terceira Edição. Springer, Berlin, 1997. 217 p.
32. PAPADIMITRIOU, CHRISTOS H. **Computational Complexity**. Primeira Edição. Addison-Wesley, novembro de 1993. 500 p.
33. AHO, ALFRED V.; ULLMAN, JEFFREY D. **Foundations of Computer Science** - C Edition. Computer Science Press, New York, 1995. 786 p.
34. **Pós-Graduação PUC-Rio**. Modelo de Texto MS WORD™ para formatação de teses e dissertações. 27 transparências: coloridas. Disponível em http://www.puc-rio.br/ensinopesq/ccpg/download/template_word_inst.html. Acesso em 15 de dezembro de 2002.
35. KOZEN, DEXTER. On Hoare Logic and Kleene Algebra with Tests. **ACM Transactions on Computational Logic**, Vol 1, No. 1, julho de 2000. p 60 - 76.
36. HAREL, DAVID. Dynamic Logic. **Handbook of Philosophical Logic**, Vol II, 1984. Capítulo 10, p 497 - 604.
37. FELTY, AMY. **A Tutorial on Lambda Prolog and its Applications to Theorem Proving**. Bell Labs, Lucent Technologies, setembro de 1997. 35 transparências coloridas. Disponível em <ftp://ftp.research.bell-labs.com/dist/felty/lprolog97.ps.gz>. Acesso em 09 de março de 2003.
38. TROELSTRA, ANNE. S.; SCHWICHTENBERG, H. **Basic Proof Theory**. Cambridge University Press, agosto de 2002. 430 p.

39. WAINER, STANLEY. S.; WALLEN, L. A. Basic Proof Theory. **Proof Theory**. Editores: Peter Aczel, Harold Simmons e Stanley Wainer. Cambridge University Press, 1993. 26 p.
40. **Manual para conversão de Teses e Dissertações em PDF**. 13 p. Disponível em <http://www.dbd.puc-rio.br/manuais/manualpdf.pdf>. Acesso em 21 de abril de 2003.
41. CARDELLI, LUCA et al. The Modula-3 Type System. In Symposium on Principles of Programming Languages (POPL' 89), Austin, Texas, janeiro de 1989. **POPL 1989**. p 102 - 112. Disponível em <http://research.microsoft.com/Users/luca/Papers/Modula3TypeSystem.A4.pdf> Acesso em 16 de agosto de 2003.
42. HARPER, ROBERT; HONSELL, FURIO; PLOTKIN, GORDON. A Framework for Defining Logics. **Journal of the Association for Computing Machinery** **40**. Janeiro de 2003. p 143 - 184. Disponível em http://www.dcs.ed.ac.uk/home/gdp/publications/Framework_Def_Log.pdf. Acesso em 17 de agosto de 2003.
43. SHAO, ZONG; LEAGUE, CHRISTOPHER; MONNIER, STEFAN. Implementing Typed Intermediate Languages. In International Conference on Functional Programming (ICFP' 98), Baltimore, Maryland. **Proceedings 1998**. p 313 - 323. Disponível em <http://flint.cs.yale.edu/flint/publications/imp.pdf>. Acesso em 17 de agosto de 2003.
44. NETTO, JOSÉ LUCAS MOURÃO RANGEL. **Notas de Aula da Cadeira de Compiladores da Graduação (INF 1715)**. PUC-Rio, Rio de Janeiro, Março de 1999. Disponível em <http://www-di.inf.puc-rio.br/~rangel/comp.html>. Acesso em 22 de agosto de 2003.
45. HOPCROFT, JOHN E.; ULLMAN, JEFFREY D. **Introduction to Automata Theory, Languages, and Computation**. Addison-Wesley, 1979. 418 p
46. PRAWITZ, DAG. **Natural Deduction: a proof-theoretical study**. Almqvist and Wiksell, Stockholm, 1965.

47. FISCHER, CHARLES N.; LEBLANC, RICHARD J. JR. **Crafting a Compiler with C**. Addison-Wesley, julho de 1991. 812 p.
48. CLAVEL, MANUEL et al. **Maude 2.0 Manual**. SRI, junho de 2003. 264 p. Disponível em <http://maude.cs.uiuc.edu/manual/maude-manual.ps>. Acesso em 29 de agosto de 2003.
49. ZELKOWITZ, MARVIN V. **Weakest Pre-Condition**. University of Maryland, 2001. 70 transparências: coloridas. Disponível em <http://www.cs.umd.edu/users/mvz/handouts/weakest-precondition.pdf>. Acesso em 31 de agosto de 2003.

Apêndice: Código Fonte do Corretor de Programas

lexica.pl

```

=====
%NOME: lexica.pl
%
%Projeto: Correção de Programas
%
%   Versão: V 2.00
%   Autora : Juliana Carpes Imperial - 0124811-7
%
%Descrição do módulo
%   Objetivo:
%       - implementação dos predicados que fazem a análise léxica da linguagem
%       LACC.
=====

%Versão anterior: V 1.00
%Modificações: adição de três entradas no predicado que reconhece os caracteres
%               simples para a aceitação de vetores e para a aceitação do
%               operador binário módulo.
=====

%Nome: fazerAnaLexica
%Descrição do predicado: predicado principal do módulo, o qual realiza todas as
%                       chamadas necessárias para realizar a análise léxica.
%
%Variável de saída:
%   - Tokens: a lista de tokens do programa em LACC.
%
%Assertiva de saída:
%   - o arquivo do programa em LACC lido está fechado.
=====

% lê o nome de um arquivo e tenta abri-lo. Se der erro, este é tratado; se o
% arquivo for aberto com sucesso, os tokens do arquivo LACC são lidos e se a
% leitura dos tokens foi feita com sucesso, tenta fechar o arquivo. Se der erro,
% este é tratado, se não, exibe os tokens lidos ao usuário.
fazerAnaLexica( Tokens )                               :-
    write( '\nEscaneando: análise léxica...\n' )        ,
    write( 'Entre nome do arquivo fonte entre \' \' e seguido de ponto: ' ) ,
    read( Arquivo )                                    ,
    catch( see( Arquivo ), _Excecao, tratarErro( abertura ) ) ,
    nl                                                 ,
    escanear( Tokens )                                 ,
    catch( seen, _Excecao, tratarErro( fechamento ) ) ,
    write( '\nAnálise léxica feita com sucesso.\n' )    ,
    write( '\ntokens = ' )                              ,
    write( Tokens )                                     ,
    nl                                                 ,
    !                                                 .
=====

%Nome: serEspaco
%Descrição do predicado: determina se um caractere lido é o espaço, ou seja, se
%                       seu código ASCII é 32.
%
%Variável de entrada:
%   - o código ASCII do caractere lido.
=====

```

```
serEspaco( 32 ) .
```

```

=====
%Nome: serTab
%Descrição do predicado: determina se um caractere lido é o tab, ou seja, se seu
%                          código ASCII é 9.
%
%
%Variável de entrada:
% - o código ASCII do caractere lido.
=====

```

```
serTab( 9 ) .
```

```

=====
%Nome: serFimLinha
%Descrição do predicado: determina se um caractere lido é o que determina fim de
%                          linha, ou seja, se seu código ASCII é 10.
%
%
%Variável de entrada:
% - o código ASCII do caractere lido.
=====

```

```
serFimLinha( 10 ) .
```

```

=====
%Nome: serFimArq
%Descrição do predicado: determina se um caractere lido é o que determina fim de
%                          arquivo, ou seja, se seu código ASCII é 26 ou -1.
%
%
%Variável de entrada:
% - o código ASCII do caractere lido.
=====

```

```
serFimArq( 26 ) .
serFimArq( -1 ) .
```

```

=====
%Nome: serPonto
%Descrição do predicado: determina se um caractere lido é o ponto ("."), ou
%                          seja, se seu código ASCII é 46.
%
%
%Variável de entrada:
% - o código ASCII do caractere lido.
=====

```

```
serPonto( 46 ) .
```

```

=====
%Nome: serSimples
%Descrição do predicado: determina se um caractere lido é tal que, se for o
%                          primeiro de um token, é o próprio token.
%
%
%Variável de entrada:
% - o código ASCII do caractere lido.
%
%
%Variável de saída:
% - o nome do token correspondente ao caractere lido.
=====

```

```

serSimples( 40, parenEsq ) . % caractere: ( -- token: parenEsq
serSimples( 41, parenDir ) . % caractere: ) -- token: parenDir
serSimples( 91, colcheEsq ) . % caractere: [ -- token: colcheEsq
serSimples( 93, colcheDir ) . % caractere: ] -- token: colcheDir
serSimples( 42, mult ) . % caractere: * -- token: mult
serSimples( 43, soma ) . % caractere: + -- token: soma
serSimples( 45, menos ) . % caractere: - -- token: menos
serSimples( 47, divide ) . % caractere: / -- token: divide
serSimples( 59, ptEVirg ) . % caractere: ; -- token: ptEVirg
serSimples( 61, igual ) . % caractere: = -- token: igual

```

```
=====
```

```

%Nome: serDuplo
%Descrição do predicado: determina se um caractere lido é tal que, se for o
% primeiro de um token, ou é o próprio token ou é o
% primeiro de um token com dois caracteres.
%
%Variável de entrada:
% - o código ASCII do caractere lido.
%
%Variável de saída:
% - o nome do token correspondente ao caractere lido se o primeiro só contiver
% um caractere.
%=====

serDuplo( 58, doisPts ) . % caractere: : -- token: doisPts
serDuplo( 60, menor ) . % caractere: < -- token: menor
serDuplo( 62, maior ) . % caractere: > -- token: maior

%=====
%Nome: serPar
%Descrição do predicado: determina se dois caracteres lidos são tais que, se
% forem lidos em seqüência, formam um token de dois
% caracteres.
%
%Variáveis de entrada:
% - o código ASCII do primeiro caractere lido; e
% - o código ASCII do segundo caractere lido.
%
%Variável de saída:
% - o nome do token correspondente ao par de caracteres lidos.
%=====

serPar( 58, 61, atribui ) . % par de caracteres: := -- token: atribui
serPar( 60, 61, menorIg ) . % par de caracteres: <= -- token: menorIg
serPar( 62, 61, maiorIg ) . % par de caracteres: >= -- token: maiorIg
serPar( 60, 62, difer ) . % par de caracteres: <> -- token: difer

%=====
%Nome: serReserv
%Descrição do predicado: determina se um identificador corresponde a uma palavra
% reservada da linguagem LACC.
%
%Variável de entrada:
% - o nome do identificador.
%=====

serReserv( skip ) . % palavra reservada skip (comando vazio)
serReserv( if ) . % palavra reservada if (teste)
serReserv( then ) . % palavra reservada then (caso positivo do teste)
serReserv( fi ) . % palavra reservada fi (fim de if-then)
serReserv( else ) . % palavra reservada else (caso negativo do teste)
serReserv( esle_fi ) . % palavra reservada esle_fi (fim de if-then-else)
serReserv( while ) . % palavra reservada while (repetição)
serReserv( do ) . % palavra reservada do (o que faz a repetição)
serReserv( od ) . % palavra reservada od (fim de while)
serReserv( and ) . % palavra reservada and (conectivo booleano)
serReserv( or ) . % palavra reservada or (conectivo booleano)
serReserv( not ) . % palavra reservada not (conectivo booleano)
serReserv( true ) . % palavra reservada true (booleano primitivo)
serReserv( false ) . % palavra reservada false (booleano primitivo)
serReserv( mod ) . % palavra reservada mod (operador de módulo)

%=====
%Nome: serBranco
%Descrição do predicado: determina se um caractere lido é um branco.
%
%Variável de entrada:
% - Car: o código ASCII do caractere lido.
%=====

serBranco( Car ) :- % um caractere é um branco se:
    serEspaco( Car ) ; % é um espaço, ou
    serTab( Car ) ; % é um tab, ou
    serFimLinha( Car ) . % é um fim de linha

```

```

%=====
%Nome: serMinuscula
%Descrição do predicado: determina se um caractere lido corresponde a uma letra
%                               minúscula.
%
%Variável de entrada:
%   - Car: o código ASCII do caractere lido.
%=====

serMinuscula( Car ) :- % um caractere é uma letra minúscula se:
    Car >= 97      , % é 'a' (ASCII = 97) ou vem depois dele e
    Car <= 122    . % é 'z' (ASCII = 122) ou vem antes dele

%=====
%Nome: serMaiuscula
%Descrição do predicado: determina se um caractere lido corresponde a uma letra
%                               maiúscula.
%
%Variável de entrada:
%   - Car: o código ASCII do caractere lido.
%=====

serMaiuscula( Car ) :- % um caractere é uma letra maiúscula se:
    Car >= 65      , % é 'A' (ASCII = 65) ou vem depois dele e
    Car <= 90     . % é 'Z' (ASCII = 90) ou vem antes dele

%=====
%Nome: serDigito
%Descrição do predicado: determina se um caractere lido corresponde a um
%                               dígito decimal.
%
%Variável de entrada:
%   - Car: o código ASCII do caractere lido.
%=====

serDigito( Car ) :- % um caractere é um dígito decimal se:
    Car >= 48      , % é '0' (ASCII = 48) ou vem depois dele e
    Car <= 57     . % é '9' (ASCII = 57) ou vem antes dele

%=====
%Nome: serCarVal
%Descrição do predicado: determina se um caractere lido é válido dentro de um
%                               identificador.
%
%Variável de entrada:
%   - Car: o código ASCII do caractere lido.
%=====

serCarVal( Car )      :- % um caractere é válido dentro de um identificador se
    serMinuscula( Car ) ; % é uma letra minúscula, ou
    serMaiuscula( Car ) ; % é uma letra maiúscula, ou
    serDigito( Car )   ; % é um dígito decimal, ou
    Car == 95         . % é um sublinhado ('_', ASCII = 95)

%=====
%Nome: escanear
%Descrição do predicado: escanea a lista de caracteres e monta a lista de tokens
%                               do programa em LACC sendo lido.
%
%Variável de saída:
%   - ListaTokens: a lista de tokens do programa em LACC lido.
%
%Assertiva de entrada:
%   - o arquivo com o programa LACC está aberto.
%=====

escanear( [ Token | ListaTokens ] )      :-
    tab( 4 )                               , % imprime 4 brancos
    pegarCar( Car )                         , % pega o primeiro caractere
    pegarToken( Car, Token, NovoCar )      , % pega o primeiro token
    restoProg( Token, NovoCar, ListaTokens ) , % escanea o resto dos tokens
    !

```

```

%=====
%Nome: pegarCar
%Descrição do predicado: pega o próximo caractere do código fonte do programa em
%                          LACC sendo lido.
%
%
%Variável de saída:
% - Car: um caractere presente no código fonte do programa em LACC.
%
%
%Assertiva de entrada:
% - o arquivo com o programa LACC está aberto.
%=====

% lê um caractere, se houver erro, este é tratado; se não der erro e ele for fim
% de linha, uma linha é pulada e 4 brancos são impressos; já se ele for fim de
% arquivo, uma linha é pulada; por fim, se não for fim de linha ou de arquivo, o
% imprime
pegarCar( Car )                               :-
  catch( get0( Car ), _Excecao, tratarErro( leitura ) ) ,
  ( serFimLinha( Car )                               ,
    nl                                               ,
    tab( 4 )                                         ;
    serFimArq( Car )                               ,
    nl                                               ;
    put( Car ) )                                     .

%=====
%Nome: restoProg
%Descrição do predicado: escanea o restante da lista de caracteres e monta o
%                          restante da lista de tokens do programa em LACC sendo
%                          lido.
%
%
%Variáveis de entrada:
% - UltToken: o último token lido; e
% - Car:      o último caractere lido.
%
%
%Variável de saída:
% - ListaTokens: a lista de tokens do programa em LACC lido posteriores a
%                UltToken.
%
%
%Assertiva de entrada:
% - o arquivo com o programa LACC está aberto.
%=====

% se ler depois do fim de arquivo, não retorna tokens
restoProg( eop, _Car, [ ] ) .

% se não ler depois do fim de arquivo, pega o próximo token, o coloca na lista
% de tokens e pega o restante da lista de tokens recursivamente
restoProg( _UltToken, Car, [ Token | ListaTokens ] ) :-
  pegarToken( Car, Token, NovoCar ) ,
  restoProg( Token, NovoCar, ListaTokens ) .

%=====
%Nome: pegarToken
%Descrição do predicado: pega o próximo token do programa em LACC sendo lido.
%
%
%Variável de entrada:
% - Car: o último caractere lido.
%
%
%Variáveis de saída:
% - Token:      o token lido; e
% - NovoCar_2: o caractere lido mais recentemente.
%
%
%Assertiva de entrada:
% - o arquivo com o programa LACC está aberto.
%
%
%Assertiva de saída:
% - o arquivo manipulado continuará aberto a menos que um caractere inválido
%   seja lido, o que fará com que o programa seja abortado.
%=====

% se ler fim de arquivo, retorna o token correspondente
pegarToken( Car, eop, 0 ) :-

```

```

    serFimArq( Car )
.

% pega tokens com apenas um caractere
pegarToken( Car, Token, NovoCar_2 ) :-
    serSimples( Car, Token )
    ,
    pegarCar( NovoCar_2 )
.

% pega token que pode ser tanto simples ou duplo:
% se o primeiro caractere lido não fizer par com o próximo, o token é simples
% caso contrário, é duplo
pegarToken( Car, Token, NovoCar_2 ) :-
    serDuplo( Car, OutroToken )
    ,
    pegarCar( NovoCar_1 )
    ,
    ( serPar( Car, NovoCar_1, Token )
    ,
    pegarCar( NovoCar_2 )
    ;
    Token = OutroToken
    ,
    NovoCar_2 = NovoCar_1 )
.

% pega token que é um identificador, que tem como primeiro caractere uma letra
% o resto do token pode ser composto de letras, dígitos e sublinhados
% se o identificador for palavra reservada de LACC, apenas ele é retornado
% senão, ele é retornado como um predicado que diz que ele é identificador
pegarToken( Car, Token, NovoCar_2 ) :-
    ( serMinuscula( Car )
    ;
    serMaiuscula( Car ) )
    ,
    pegarCar( NovoCar_1 )
    ,
    pegarRestoId( NovoCar_1, ListaCar, NovoCar_2 )
    ,
    name( Id, [ Car | ListaCar ] )
    ,
    ( serReserv( Id )
    ;
    Token = Id
    ;
    Token = ide( Id ) )
.

% pega token que é um número, o qual é composto apenas de dígitos
pegarToken( Car, num( Num ), NovoCar_2 ) :-
    serDigito( Car )
    ,
    pegarCar( NovoCar_1 )
    ,
    pegarRestoNum( NovoCar_1, ListaCar, NovoCar_2 )
    ,
    name( Num, [ Car | ListaCar ] )
.

% se o último caractere lido é um branco, apenas lê um novo caractere
% e pega um token a partir dele
pegarToken( Car, Token, NovoCar_2 ) :-
    serBranco( Car )
    ,
    pegarCar( NovoCar_1 )
    ,
    pegarToken( NovoCar_1, Token, NovoCar_2 )
.

% se o último caractere lido é inválido, ou seja, não pode pertencer a nenhum
% token, isto é avisado ao usuário, e o programa é abortado, fechando antes o
% arquivo aberto
pegarToken( Car, _Token, _NovoCar_2 ) :-
    write( '\n\nCaractere ilegal: ' )
    ,
    put( Car )
    ,
    nl
    ,
    catch( seen, _Excecao, abort )
    ,
    abort
.

%=====
%Nome: pegarRestoId
%Descrição do predicado: pega o restante de um identificador.
%
%Variável de entrada:
% - Car: o último caractere lido.
%
%Variáveis de saída:
% - ListaCar: a lista de caracteres lidos que representa um identificador; e
% - NovoCar_2: o caractere lido mais recentemente.
%
%Assertiva de entrada:
% - o arquivo com o programa LACC está aberto.
%=====

% se o último caractere lido puder ser encontrado dentro de um identificador, a
% busca por caracteres que possam pertencer a ele continua recursivamente
pegarRestoId( Car, [ Car | ListaCar ], NovoCar_2 ) :-
    serCarVal( Car )
    ,
    pegarCar( NovoCar_1 )
    ,

```

```

    pegarRestoId( NovoCar_1, ListaCar, NovoCar_2 ) .

% se o último caractere lido não puder ser encontrado dentro de um
% identificador, pára a busca
pegarRestoId( Car, [ ], Car ) .

%=====
%Nome: pegarRestoNum
%Descrição do predicado: pega o restante de um número.
%
%Variável de entrada:
% - Car: o último caractere lido.
%
%Variáveis de saída:
% - ListaCar: a lista de caracteres lidos que representa um número; e
% - NovoCar_2: o caractere lido mais recentemente.
%
%Assertiva de entrada:
% - o arquivo com o programa LACC está aberto.
%=====

% se o último caractere lido for um dígito, a busca por dígitos continua
% recursivamente
pegarRestoNum( Car, [ Car | ListaCar ], NovoCar_2 ) :-
    serDigito( Car )
    ,
    pegarCar( NovoCar_1 )
    ,
    pegarRestoNum( NovoCar_1, ListaCar, NovoCar_2 ) .

% se o último caractere lido não for um dígito, pára a busca
pegarRestoNum( Car, [ ], Car ) .

```

sintatica.pl

```

%=====
%NOME: sintatica.pl
%
%Projeto: Correção de Programas
%
% Versão: V 2.00
% Autora : Juliana Carpes Imperial - 0124811-7
%
%Descrição do módulo
% Objetivos:
% - implementação das cláusulas que fazem a análise sintática da linguagem
% LACC, utilizando regras de DCG (Definite Clause Grammars);
% - implementação de um predicado que pega uma lista de tokens da
% linguagem LACC e reconhece as representações de vetores contidos nela;
% e
% - implementação de um predicado que imprime o primeiro erro de sintaxe
% encontrado, se for o caso.
%=====

%=====
%Versão anterior: V 1.00
%Modificações: adicionar a aceitação do operador binário módulo e o
% reconhecimento de vetores.
%=====

%=====
%Nome: fazerAnaSintatica
%Descrição do predicado: predicado principal do módulo, o qual realiza todas as
% chamadas necessárias para realizar a análise sintática.
%
%Variável de entrada:
% - Tokens: a lista de tokens do código fonte do programa em LACC.
%
%Variável de saída:
% - Arvore: a árvore sintática do programa em LACC.
%=====

```

```

% analisa os tokens lidos do código-fonte, achando os vetores neles existentes
% e monta a árvore sintática correspondente se o programa estiver
% sintaticamente correto (será abortado se não o estiver)
fazerAnaSintatica( Tokens, TokensVet, Arvore )      :-
    write( '\nInterpretando: análise sintática...\n' ) ,
    acharVetores( Tokens, [ ], TokensVet ) ,
    serComandos( Arvore, TokensVet, TokensVet, [ eop ] ) ,
    write( 'Análise sintática feita com sucesso.\n' ) ,
    write( '\narvore sintática = ' ) ,
    write( Arvore ) ,
    nl ,
    ! .

=====
%Nome: serComandos
%Descrição da cláusula: reconhece a lista de comandos do programa em LACC.
%
%Variável de entrada:
% - Tokens: a lista de tokens do código fonte.
%
%Variável de saída:
% - Comandos: a lista de comandos dados no código fonte se não houve erro
%             sintático. Se houve, será chamada a rotina de tratamento de
%             erro e o programa, abortado.
%
%Cláusula na BNF de LACC:
% - < cmds > ::= < cmd > | < cmd > ; < cmds >
=====

serComandos( Comandos, Tokens )      -->
    serComando( Comando, Tokens )    , % reconhece primeiro comando
    serRestoCmds( Comando, Comandos, Tokens ) . % reconhece os demais comandos

% se tentou esta cláusula, houve erro sintático, portanto, o erro será tratado
serComandos( _Comandos, Tokens ) -->
    tratarErro( Tokens ) .

=====
%Nome: serRestoCmds
%Descrição da cláusula: reconhece o restante da lista de comandos do programa em
%                       LACC.
%
%Variáveis de entrada:
% - Tokens: a lista de tokens do código fonte; e
% - Comando: o último comando reconhecido.
%
%Variável de saída:
% - Comandos: a lista de comandos reconhecidos, incluindo o último comando, se
%             não houve erro sintático; se houve, o programa será abortado.
=====

% reconhece o restante da lista de comandos, reconhecendo, antes, o ";" que
% veio depois de um comando já reconhecido
serRestoCmds( Comando, [ Comando | Comandos ], Tokens ) -->
    [ ptEVirg ] ,
    serComandos( Comandos, Tokens ) .

% não há mais comandos a serem reconhecidos; todos já o foram
serRestoCmds( Comando, [ Comando ], _Tokens ) -->
    [ ] .

=====
%Nome: serComando
%Descrição da cláusula: reconhece um comando do programa em LACC.
%
%Variável de entrada:
% - Tokens: a lista de tokens do código fonte.
%
%Variável de saída:
% - Comando: uma árvore com a estrutura do comando se não houve erro
%             sintático. Se houve, será chamada a rotina de tratamento de erro
%             e programa, abortado.
%
%Cláusula na BNF de LACC:

```

```

% - < cmd > ::= < var > := < exp > | skip | if < bexp > then < cmds > fi
%           | if < bexp > then < cmds > else < cmds > esle_fi
%           | while < bexp > do < cmds > od
%=====

% reconhece um comando de atribuição, reconhecendo primeiro a variável e,
% posteriormente, a expressão que é atribuída a primeira
serComando( atribui( Variavel, Expressao ), Tokens ) -->
  [ var( Variavel ), atribui ]
  serExpressao( Expressao, Tokens )

% reconhece um comando skip
serComando( skip, _Tokens ) -->
  [ skip ]

% reconhece um comando if, reconhecendo, nesta ordem, o identificador if, uma
% expressão booleana, o identificador then, os comandos que serão executados
% caso o teste do if seja bem sucedido e, por fim, o resto do comando if
% (lê a parte do else se ela existir)
serComando( Comando, Tokens ) -->
  [ if ]
  serBoolExpr( Teste, Tokens )
  [ then ]
  serComandos( Entao, Tokens )
  serRestoIf( Teste, Entao, Comando, Tokens )

% reconhece um comando while, reconhecendo, nesta ordem, o identificador
% while, uma expressão booleana, o identificador do, os comandos do corpo
% do while e o identificador od, indicando fim do while
serComando( while( Teste, Comandos ), Tokens ) -->
  [ while ]
  serBoolExpr( Teste, Tokens )
  [ do ]
  serComandos( Comandos, Tokens )
  [ od ]

% se tentou esta cláusula, houve erro sintático, portanto, o erro será tratado
serComando( _Comando, Tokens ) -->
  tratarErro( Tokens )

%=====
%Nome: serRestoIf
%Descrição da cláusula: reconhece o restante de um comando if.
%
%Variáveis de entrada:
% - Tokens: a lista de tokens do código fonte;
% - Teste: a expressão booleana a ser avaliada pelo comando if; e
% - Então: a lista de comandos executada se o teste do if for bem sucedido.
%
%Variável de saída:
% - if( ... ): a árvore com a estrutura do comando if.
%=====

% se for um if-then-else, reconhece, nesta ordem, o identificador else, os
% comandos que serão executados caso o teste do if não seja bem sucedido e o
% e o identificador esle_fi, indicando fim do if-then-else
serRestoIf( Teste, Entao, if( Teste, Entao, Senao ), Tokens ) -->
  [ else ]
  serComandos( Senao, Tokens )
  [ esle_fi ]

% se for um if-then, reconhece apenas o identificador fi, indicando fim do
% if-then, já que todo o comando if já foi reconhecido
serRestoIf( Teste, Entao, if( Teste, Entao ), _Tokens ) -->
  [ fi ]

%=====
%Nome: serExpressao
%Descrição da cláusula: reconhece uma expressão numérica ou booleana do programa
% em LACC.
%
%Variável de entrada:
% - Tokens: a lista de tokens do código fonte.
%
%Variável de saída:

```

```

% - Expressão: uma árvore com a estrutura da expressão se não houve erro
% sintático. Se houve, será chamada a rotina de tratamento de
% erro e o programa, abortado.
%
%Cláusula na BNF de LACC:
% - < exp > ::= < bexp > | < iexp >
%=====

serExpressao( Expressao, Tokens ) -->
    serIntExpr( Expressao ) ; % reconhece expressão numérica, ou
    serBoolExpr( Expressao, Tokens ) . % reconhece expressão booleana

% se tentou esta cláusula, houve erro sintático, portanto, o erro será tratado
serExpressao( _Expressao, Tokens ) -->
    tratarErro( Tokens ) .

%=====
%Nome: serIntExpr
%Descrição da cláusula: reconhece uma expressão numérica do programa em LACC.
%
%Variável de saída:
% - Expressão: uma árvore com a estrutura da expressão numérica se não houve
% erro sintático. Se houve, o programa será abortado.
%
%Cláusula na BNF de LACC:
% - < iexp > ::= < item > | < item > < fraco > < iexp >
%=====

serIntExpr( Expressao ) -->
    serIntTermo( Termo ) , % reconhece o primeiro termo da expr
    serRestoIntExpr( Termo, Expressao ) . % reconhece o resto da expr

%=====
%Nome: serRestoIntExpr
%Descrição da cláusula: reconhece o resto de uma expressão numérica do programa
% em LACC.
%
%Variável de entrada:
% - Termo: o último termo reconhecido da expressão sendo reconhecida.
%
%Variável de saída:
% - Expressão: uma árvore com a estrutura da expressão numérica se não houve
% erro sintático. Se houve, o programa será abortado.
%=====

% reconhece o operador da expressão (+ / -), o termo seguinte, monta um termo
% com o termo anterior, o lido desta vez e o operador e, por fim, reconhece o
% restante da expressão
serRestoIntExpr( Termo, Expressao ) -->
    serOperFrac( Operador ) ,
    serIntTermo( OutroTermo ) ,
    serRestoIntExpr( expInt( Operador, Termo, OutroTermo ), Expressao ) .

% toda a expressão já foi reconhecida; nada mais precisa ser reconhecido
serRestoIntExpr( Expressao, Expressao ) -->
    [ ] .

%=====
%Nome: serIntTermo
%Descrição da cláusula: reconhece um termo numérico do programa em LACC.
%
%Variável de saída:
% - Termo: uma árvore com a estrutura do termo numérico se não houve erro
% sintático. Se houve, o programa será abortado.
%
%Cláusula na BNF de LACC:
% - < item > ::= < ielem > | < ielem > < forte > < item >
%=====

serIntTermo( Termo ) -->
    serIntElem( Elemento ) , % reconhece o primeiro elem do termo
    serRestoIntTermo( Elemento, Termo ) . % reconhece o resto do termo

```

```

%=====
%Nome: serRestoIntTermo
%Descrição da cláusula: reconhece o resto de um termo numérico do programa em
%                       LACC.
%
%Variável de entrada:
% - Elemento: o último elemento reconhecido do termo sendo reconhecido.
%
%Variável de saída:
% - Termo: uma árvore com a estrutura do termo numérico se não houve erro
%          sintático. Se houve, o programa será abortado.
%=====

% reconhece o operador do termo (* / // mod), o elemento seguinte, monta um
% elemento com o elemento anterior, o lido desta vez e o operador e, por fim,
% reconhece o restante do termo
serRestoIntTermo( Elemento, Termo )      -->
    serOperForte( Operador )             ,
    serIntElem( OutroElem )              ,
    serRestoIntTermo( expInt( Operador, Elemento, OutroElem ), Termo ) .

% todo o termo já foi reconhecido; nada mais precisa ser reconhecido
serRestoIntTermo( Termo, Termo ) -->
    [ ] .

%=====
%Nome: serIntElem
%Descrição da cláusula: reconhece um elemento numérico do programa em LACC.
%
%Variável de saída:
% - uma árvore com a estrutura do elemento numérico se não houve erro
%   sintático. Se houve, o programa será abortado.
%
%Cláusula na BNF de LACC:
% - < ielem > ::= < num > | < var > | - ( < iexp > ) | ( < iexp > )
%=====

% o elemento numérico reconhecido é um número inteiro
serIntElem( num( Num ) ) -->
    [ num( Num ) ] .

% o elemento numérico reconhecido é uma variável
serIntElem( var( Identificador ) ) -->
    [ var( Identificador ) ] .

% o elemento numérico reconhecido é um elemento com um sinal de menos na frente
serIntElem( menos( Elemento ) ) -->
    [ menos ] , % reconhece sinal de menos
    serIntElem( Elemento ) . % reconhece o resto do elemento

% o elemento numérico reconhecido é uma expressão entre parênteses
serIntElem( Expressao ) -->
    [ parenEsq ] , % reconhece parêntese esquerdo
    serIntExpr( Expressao ) , % reconhece expressão numérica
    [ parenDir ] . % reconhece parêntese direito

%=====
%Nome: serBoolExpr
%Descrição da cláusula: reconhece uma expressão booleana do programa em LACC.
%
%Variável de entrada:
% - Tokens: a lista de tokens do código fonte.
%
%Variável de saída:
% - Expressão: uma árvore com a estrutura da expressão booleana se não houve
%              erro sintático. Se houve, será chamada a rotina de tratamento
%              de erro e o programa, abortado.
%
%Cláusula na BNF de LACC:
% - < bexp > ::= < bterm > | < bterm > or < bexp >
%=====

serBoolExpr( Expressao, Tokens )      -->
    serBoolTermo( Termo, Tokens )     , % reconhece o primeiro termo
    serRestoBoolExpr( Termo, Expressao, Tokens ) . % reconhece o resto da expr

```

```

% se tentou esta cláusula, houve erro sintático, portanto, o erro será tratado
serBoolExpr( _Expressao, Tokens ) -->
    tratarErro( Tokens )
.

%=====
%Nome: serRestoBoolExpr
%Descrição da cláusula: reconhece o resto de uma expressão booleana do programa
%
% em LACC.
%
%
%Variáveis de entrada:
% - Tokens: a lista de tokens do código fonte; e
% - Termo: o último termo reconhecido da expressão sendo reconhecida.
%
%
%Variável de saída:
% - Expressão: uma árvore com a estrutura da expressão booleana se não houve
% erro sintático. Se houve, o programa será abortado.
%=====

% reconhece o operador da expressão (or), o termo seguinte, monta um termo com o
% termo anterior, o lido desta vez e o operador e, por fim, reconhece o restante
% da expressão
serRestoBoolExpr( Termo, Expressao, Tokens ) -->
    [ or ]
    ,
    serBoolTermo( OutroTermo, Tokens )
    ,
    serRestoBoolExpr( expBool( or, Termo, OutroTermo ), Expressao, Tokens ) .

% toda a expressão já foi reconhecida; nada mais precisa ser reconhecido
serRestoBoolExpr( Expressao, Expressao, _Tokens ) -->
    [ ]
.

%=====
%Nome: serBoolTermo
%Descrição da cláusula: reconhece um termo booleano do programa em LACC.
%
%
%Variável de entrada:
% - Tokens: a lista de tokens do código fonte.
%
%
%Variável de saída:
% - Termo: uma árvore com a estrutura do termo booleano se não houve erro
% sintático. Se houve, o programa será abortado.
%
%
%Cláusula na BNF de LACC:
% - < bterm > ::= < belem > | < belem > and < bterm >
%=====

serBoolTermo( Termo, Tokens ) -->
    serBoolElem( Elemento, Tokens )
    , % reconhece o primeiro elem
    serRestoBoolTermo( Elemento, Termo, Tokens ) . % reconhece o resto do termo

%=====
%Nome: serRestoBoolTermo
%Descrição da cláusula: reconhece o resto de um termo booleano do programa em
% LACC.
%
%
%Variáveis de entrada:
% - Tokens: a lista de tokens do código fonte; e
% - Elemento: o último elemento reconhecido do termo sendo reconhecido.
%
%
%Variável de saída:
% - Termo: uma árvore com a estrutura do termo booleano se não houve erro
% sintático. Se houve, o programa será abortado.
%=====

% reconhece o operador do termo (and), o elemento seguinte, monta um elemento
% com o elemento anterior, o lido desta vez e o operador e, por fim, reconhece o
% restante do termo
serRestoBoolTermo( Elemento, Termo, Tokens ) -->
    [ and ]
    ,
    serBoolElem( OutroElem, Tokens )
    ,
    serRestoBoolTermo( expBool( and, Elemento, OutroElem ), Termo, Tokens ) .

% todo o termo já foi reconhecido; nada mais precisa ser reconhecido
serRestoBoolTermo( Termo, Termo, _Token ) -->

```

```

[ ] .

%=====
%Nome: serBoolElem
%Descrição da cláusula: reconhece um elemento booleano do programa em LACC.
%
%Variável de entrada:
% - Tokens: a lista de tokens do código fonte.
%
%Variável de saída:
% - uma árvore com a estrutura do elemento booleano se não houve erro
%   sintático. Se houve, o programa será abortado.
%
%Cláusulas na BNF de LACC:
% - < belem > ::= true | false | < var > | not ( < bexp > ) | ( < bexp > )
%   | < comp >
% - < comp > ::= < iexp > < rel > < iexp >
%=====

% o elemento booleano reconhecido é a constante true
serBoolElem( true, _Tokens ) -->
  [ true ] .

% o elemento booleano reconhecido é a constante false
serBoolElem( false, _Tokens ) -->
  [ false ] .

% o elemento booleano reconhecido é uma variável
serBoolElem( var( Identificador ), _Tokens ) -->
  [ var( Identificador ) ] .

% o elemento booleano reconhecido é um elemento negado
serBoolElem( neg( Elemento ), Tokens ) -->
  [ not ] , % reconhece operador not
  serBoolElem( Elemento, Tokens ) . % reconhece o resto do elemento

% o elemento booleano reconhecido é uma expressão entre parênteses
serBoolElem( Expressao, Tokens ) -->
  [ parenEsq ] , % reconhece parêntese esquerdo
  serBoolExpr( Expressao, Tokens ) , % reconhece expressão booleana
  [ parenDir ] . % reconhece parêntese direito

% o elemento booleano é uma comparação entre expressões numéricas, onde uma
% expressão numérica é reconhecida, seguida de um operador de relação de ordem
% e, por fim, é reconhecida outra expressão numérica
serBoolElem( expBool( Relacao, Expressao_1, Expressao_2 ), _Tokens ) -->
  serIntExpr( Expressao_1 ) ,
  serRelacao( Relacao ) ,
  serIntExpr( Expressao_2 ) .

%=====
%Nome: serOperFraco
%Descrição da cláusula: reconhece um operador de adição ou subtração.
%
%Variável de saída:
% - o operador reconhecido.
%
%Cláusula na BNF de LACC:
% - < fraco > ::= + | -
%=====

serOperFraco( soma ) -->
  [ soma ] . % reconhece o operador de adição

serOperFraco( menos ) -->
  [ menos ] . % reconhece o operador de subtração

%=====
%Nome: serOperForte
%Descrição da cláusula: reconhece um operador de multiplicação, divisão ou
%   módulo.
%
%Variável de saída:
% - o operador reconhecido.

```

```

%
%Cláusula na BNF de LACC:
% - < forte > ::= * | / | mod
%=====

serOperForte( mult ) -->
[ mult ] . % reconhece o operador de multiplicação

serOperForte( divide ) -->
[ divide ] . % reconhece o operador de divisão

serOperForte( mod ) -->
[ mod ] . % reconhece o operador de módulo

%=====
%Nome: serRelacao
%Descrição da cláusula: reconhece um operador de comparação entre expressões
% numéricas.
%
%Variável de saída:
% - o operador de relação reconhecido.
%
%Cláusula na BNF de LACC:
% - < rel > ::= <= | < | = | >= | <>
%=====

serRelacao( igual ) -->
[ igual ] . % reconhece o operador de igualdade

serRelacao( difer ) -->
[ difer ] . % reconhece o operador de desigualdade

serRelacao( menor ) -->
[ menor ] . % reconhece o operador de menor do que

serRelacao( maior ) -->
[ maior ] . % reconhece o operador de maior do que

serRelacao( menorIg ) -->
[ menorIg ] . % reconhece o operador de menor ou igual do que

serRelacao( maiorIg ) -->
[ maiorIg ] . % reconhece o operador de maior ou igual do que

%=====
%Nome: acharVetores
%Descrição do predicado: pega uma lista de tokens da linguagem LACC e reconhece
% as representações de vetores contidos nela, unindo-as
% em um só token.
%
%Variáveis de entrada:
% - Tokens: a lista de tokens do código fonte; e
% - TokensInv: a lista de tokens na ordem inversa com os vetores já
% reconhecidos.
%
%Variável de saída:
% - TokensVet: a lista de tokens com os vetores já reconhecidos na ordem
% em que eles aparecem no código fonte.
%=====

% toda a lista de tokens já foi processada. Portanto, todos os vetores já foram
% encontrados
acharVetores( [ ], TokensInv, TokensVet ) :-
reverse( TokensInv, TokensVet ) .

% se a representação de um vetor estiver no início da lista, os tokens que
% representam o vetor serão unidos em um só e mais vetores serão buscados
% recursivamente
acharVetores( Tokens, TokensInv, TokensVet ) :-
acharVetor( Tokens, Vetor, RestoTokens ) ,
!
acharVetores( RestoTokens, [ var( Vetor ) | TokensInv ], TokensVet ) .

% se uma variável estiver no início da lista de tokens, seu rótulo será
% mudado de "ide" para "var"

```

```

acharVetores( [ ide( Var ) | Tokens ], TokensInv, TokensVet ) :-
    !
    acharVetores( Tokens, [ var( Var ) | TokensInv ], TokensVet ) .

% caso contrário, simplesmente analisa a lista a partir do próximo token
% recursivamente
acharVetores( [ Token | Tokens ], TokensInv, TokensVet ) :-
    acharVetores( Tokens, [ Token | TokensInv ], TokensVet ) .

%=====
%Nome: acharVetor
%Descrição do predicado: pega uma lista de tokens da linguagem LACC e reconhece
%
% a representação de um vetor existente em seu início,
% unindo-a em um só token.
%
%Variável de entrada:
% - Tokens: a lista de tokens do código fonte.
%
%Variáveis de saída:
% - Vetor: a representação de vetor encontrada em um só token; e
% - RestoTokens: o restante da lista de tokens do código fonte ainda não
% processada.
%=====

% as duas partes finais do vetor já foram encontradas, que são o índice e o
% colchete à direita
acharVetor( [ Indice | Tokens ], Idx, Tokens ) :-
    ( Indice = ide( Idx ) ;
      Indice = num( Idx ) ) ,
    Tokens = [ colcheDir | _RestoTokens ] .

% a partes inicial e final do vetor foram encontradas, mas deve-se procurar por
% uma representação de vetor recursivamente no interior deste
acharVetor( [ ide( Var ), colcheEsq | Tokens ], Vetor, RestoTokens ) :-
    acharVetor( Tokens, RestoVetor, [ colcheDir | RestoTokens ] ) ,
    Vetor = vet( Var, RestoVetor ) .

%=====
%Nome: tratarErro
%Descrição do predicado: trata um erro encontrado durante a análise sintática.
%
%Variáveis de entrada:
% - Tokens: a lista de tokens do código fonte; e
% - Restantes: os tokens que ainda não foram reconhecidos pelo analisador
% sintático.
%
%Variável de saída:
% - Inutil: não retorna nada, já que o programa é abortado durante a execução
% deste predicado. Esta aí apenas por causa da transformação das
% cláusulas DCG em predicados PROLOG.
%=====

% acha os tokens reconhecidos, pois é a primeira parte da lista dos tokens do
% código fonte, cuja última parte são os que não foram reconhecidos por causa do
% erro
tratarErro( Tokens, Restantes, _Inutil ) :-
    nl ,
    write( 'Tokens reconhecidos: ' ) ,
    append( Lidos, Restantes, Tokens ) , % acha os tokens reconhecidos
    write( Lidos ) , % imprime os reconhecidos
    nl ,
    write( '\nHouve erro aqui: ' ) , % aponta onde houve o erro
    write( Restantes ) , % imprime os não reconhecidos
    nl ,
    abort . % aborta o programa

```

condicoes.pl

```

%=====

```

```

%NOME: condicoes.pl
%
%Projeto: Correção de Programas
%
%   Versão: V 2.00
%   Autora : Juliana Carpes Imperial - 0124811-7
%
%Descrição do módulo
%  Objetivos:
%    - ler as pré e pós-condições, e os invariantes que serão utilizados
%      durante a correção do programa em LACC;
%    - determinar se as pré e pós-condições, e os invariantes estão
%      sintaticamente corretos;
%    - achar os loops existentes no programa escrito em LACC reconhecido, para
%      que seus invariantes possam ser dados pelo usuário;
%    - procurar o invariante de um determinado loop em uma lista de
%      < loop, invariante >;
%    - ver se uma pré-condição é equivalente a uma pós-condição após um
%      comando de atribuição;
%    - ver se uma condição possui um termo que determina o valor de uma
%      variável;
%    - renomear uma variável ligada através de um quantificador; e
%    - determinar o nome de uma nova variável para substituir uma variável
%      ligada.
%=====

%=====
%Versão anterior: V 1.00
%Modificações:
%  - adição de predicados para a implementação das heurísticas para automatizar
%    a correção de programas;
%  - adição do tratamento aos quantificadores, vetores e ao operador binário de
%    módulo; e
%  - retirada do predicado que imprime uma condição em formato HTML.
%=====

%=====
%Descrição da consulta: declara os operadores para as condições contidas nas pré
% e pós-condições, e nos invariantes que não são
% pré-definidos pelo SWI-Prolog.
%=====

:- op( 700, xfx, '<>' ) , % diferente
   op( 700, xfx, '<=' ) , % menor ou igual
   op( 699, xfy, '<=>' ) , % equivalência
   op( 698, xfy, '=>' ) , % implicação
   op( 697, xfy, 'or' ) , % operador booleano or (\/)
   op( 696, xfy, 'and' ) , % operador booleano and (/\)
   op( 695, fy, 'not' ) . % operador booleano not (~)

%=====
%Nome: lerCondicoes
%Descrição do predicado: lê as condições dadas pelo usuário para pré e pós-
% condições, e invariantes.
%
%
%Variável de entrada:
%  - Tokens: a lista de tokens LACC lida.
%
%Variáveis de saída:
%  - Invars: a lista de invariantes para os loops do programa em LACC;
%  - PreCond: a pré-condição para o cálculo de hoare; e
%  - PosCond: a pós-condição para o cálculo de hoare.
%=====

lerCondicoes( Tokens, Invars, PreCond, PosCond ) :-
  lerCondicao( PreCond, '\nPré-condição: ' ) , % lê pré-condição
  lerCondicao( PosCond, '\nPós-condição: ' ) , % lê pós-condição
  lerInvariantes( Tokens, Invars ) , % lê invariantes
  nl ,
  ! .

%=====
%Nome: lerCondicao
%Descrição do predicado: lê uma condição dada pelo usuário para pré ou pós-
% condição, ou um invariante.
%
```

```

%
%Variável de entrada:
% - Pedido: instrução a ser impressa na tela pedindo um tipo de condição ao
%      usuário.
%
%Variável de saída:
% - Condicao: a condição dada pelo usuário.
%=====

% pede um determinado tipo de condição ao usuário, a lê e checa sua validade. Se
% for inválida, pede novamente
lerCondicao( Condicao, Pedido )      :-
    write( Pedido )                ,
    ( ( read( Condicao )             ,
        checarCondicao( Condicao ) )  ;
      ( write( '\nCondição inválida! Tente de novo!' ) ,
        lerCondicao( Condicao, Pedido ) ) ) .

%=====
%Nome: lerInvariantes
%Descrição do predicado: lê os ivariantes para os loops encontrados no programa
%      em LACC.
%
%Variável de entrada:
% - Tokens: a lista de tokens LACC lida.
%
%Variável de saída:
% - Invars: lista de invariantes, um para cada loop encontrado no programa.
%=====

lerInvariantes( Tokens, Invars ) :-
    acharLoops( Tokens, Loops ) , % acha os loops do programa em LACC
    pedirInvars( Loops, Invars ) . % lê os invariantes para os loops

%=====
%Nome: checarCondicao
%Descrição do predicado: determina se uma pré ou pós-condição, ou um invariante
%      está com a sintaxe correta (expressão booleana).
%
%Variável de entrada:
% - Condicao: a pré ou pós-condição, ou o invariante que terá sua sintaxe
%      checada.
%=====

% se a condição representada dentro do Prolog for uma variável livre (isso
% ocorre se o usuário entra variáveis começadas com letras maiúsculas sem ' '),
% ela é considerada inválida
checarCondicao( Condicao ) :-
    var( Condicao )      ,
    !                   ,
    fail                .

% testa se a condição é uma expressão booleana composta de conectivo binário:
% se-somente-se (<=>), se-então (=>), or, and ou igualdade (=) e checa
% recursivamente cada um dos operandos do conectivo
checarCondicao( Condicao ) :-
    ( Condicao = ( Cond_1 <=> Cond_2 ) ;
      Condicao = ( Cond_1 => Cond_2 ) ;
      Condicao = ( Cond_1 or Cond_2 ) ;
      Condicao = ( Cond_1 and Cond_2 ) ;
      Condicao = ( Cond_1 <> Cond_2 ) ;
      Condicao = ( Cond_1 = Cond_2 ) ) ,
    checarCondicao( Cond_1 )      ,
    checarCondicao( Cond_2 )      .

% testa se a condição é uma condição negada (not Condicao) e checa
% recursivamente a condição que vem depois do operador unário not (Condicao)
checarCondicao( not Condicao ) :-
    checarCondicao( Condicao ) .

% testa se a condição é uma condição ligada por um quantificador, o qual
% pode ser o existencial ou o universal. Posteriormente checa
% recursivamente a condição ligada e exige-se que a variável ligada ocorra
% na condição
checarCondicao( Condicao )      :-

```

```

( Condicao = ex( Var, Cond ) ;
  Condicao = all( Var, Cond ) ,
  atom( Var ) ;
  verificarCondicao( Cond ) ;
  acharVar( Var, Cond ) .

% testa se a condição é um vetor bem formado, com um átomo como o seu
% endereço e um índice que pode ser um vetor recursivamente
verificarCondicao( vet( Var, Idx ) ) :-
  atom( Var ) ;
  ( atom( Idx ) ;
    integer( Idx ) ;
    Idx = vet( _Var_1, _Var_2 ) ) ,
  verificarCondicao( Idx ) .

% testa se a condição é um átomo
verificarCondicao( Condicao ) :-
  atom( Condicao ) .

% testa se a condição é uma expressão booleana obtida através de uma comparação
% entre duas expressões aritméticas: =, <>, <, >, <= ou >= e checa
% recursivamente cada um dos operandos do conectivo recursivamente
verificarCondicao( Condicao ) :-
  ( Condicao = ( Exp_1 = Exp_2 ) ;
    Condicao = ( Exp_1 <> Exp_2 ) ;
    Condicao = ( Exp_1 < Exp_2 ) ;
    Condicao = ( Exp_1 > Exp_2 ) ;
    Condicao = ( Exp_1 <= Exp_2 ) ;
    Condicao = ( Exp_1 >= Exp_2 ) ) ,
  verificarExpressao( Exp_1 ) ,
  verificarExpressao( Exp_2 ) .

%=====
%Nome: verificarExpressao
%Descrição do predicado: determina se uma expressão aritmética está com a
% sintaxe correta (sintaxe igual a de um expressao
% inteira em LACC).
%
%
%Variável de entrada:
% - Expressao: a expressão aritmética que terá sua sintaxe checada.
%=====

% se a expressão representada dentro do Prolog for uma variável livre (isso
% ocorre se o usuário entra variáveis começadas com letras maiúsculas sem ' '),
% ela é considerada inválida
verificarExpressao( Expressao ) :-
  var( Expressao ) ,
  ! ,
  fail .

% testa se a expressão aritmética é composta de operador binário: +, -, * ou /
% e checa recursivamente cada um dos operandos da operação recursivamente
verificarExpressao( Expressao ) :-
  ( Expressao = ( Exp_1 + Exp_2 ) ;
    Expressao = ( Exp_1 - Exp_2 ) ;
    Expressao = ( Exp_1 * Exp_2 ) ;
    Expressao = ( Exp_1 / Exp_2 ) ;
    Expressao = ( Exp_1 mod Exp_2 ) ) ,
  verificarExpressao( Exp_1 ) ,
  verificarExpressao( Exp_2 ) .

% testa se a expressão aritmética é uma expressão negada (- Expressao) e checa
% recursivamente a expressão que vem depois do sinal de menos (Expressao)
verificarExpressao( - Expressao ) :-
  verificarExpressao( Expressao ) .

% testa se a expressão é um vetor bem formado, com um átomo como o seu
% endereço e um índice que pode ser um vetor recursivamente
verificarExpressao( vet( Var, Idx ) ) :-
  atom( Var ) ;
  ( atom( Idx ) ;
    integer( Idx ) ;
    Idx = vet( _Var_1, _Var_2 ) ) ,
  verificarExpressao( Idx ) .

% testa se a expressão é um átomo (variável em LACC) ou um inteiro

```

```

checarExpressao( Expressao ) :-
    atom( Expressao )      ;
    integer( Expressao )   .

%=====
%Nome: acharLoops
%Descrição do predicado: acha os loops existentes em uma lista de tokens lidos
%                          de um programa em LACC.
%
%Variável de entrada:
% - Tokens: a lista de tokens LACC lida.
%
%Variável de saída:
% - Loops: os loops encontrados na lista de tokens LACC dada.
%
%Assertiva de entrada:
% - A análise sintática do programa em LACC foi feita com sucesso.
%=====

% se todos os tokens já foram analisados, então já foram encontrados todos os
% loops (possivelmente nenhum), não havendo mais nenhum a ser encontrado
acharLoops( [ ], [ ] ) .

% se um token while for encontrado, é porque foi encontrado o início de um loop,
% o qual será lido até o final, e o resto dos tokens serão analisados para se
% saber se há algum outro loop
acharLoops( [ while | Tokens ], [ while( Teste, Corpo ) | Loops ] ) :-
    serComando( while( Teste, Corpo ), [ while | Tokens ], [ while | Tokens ],
                _Resto )
    ,
    acharLoops( Tokens, Loops )
    .

% se um token qualquer for encontrado, a busca por loops continua no próximo
% token
acharLoops( [ _Token | Tokens ], Loops ) :-
    acharLoops( Tokens, Loops )
    .

%=====
%Nome: pedirInvars
%Descrição do predicado: pede ao usuário os invariantes para os loops do
%                          programa em LACC.
%
%Variável de entrada:
% - Loops: a lista de loops do programa em LACC encontrados.
%
%Variável de saída:
% - Invars: a lista de invariantes para os loops.
%=====

% se todos os loops já foram percorridos, então já foram lidos todos os
% invariantes (possivelmente nenhum), não havendo mais nenhum invariante a ser
% lido
pedirInvars( [ ], [ ] ) .

% lê o invariante do primeiro loop e, recursivamente, lê os invariantes para os
% demais loops
pedirInvars( [ Loop | Loops ], [ inv( Loop, Invar ) | Invars ] ) :-
    nl
    ,
    write( Loop )
    ,
    lerCondicao( Invar, '\nInvariante: ' )
    ,
    pedirInvars( Loops, Invars )
    .

%=====
%Nome: procurarInvar
%Descrição do predicado: Procura o invariante para um dado loop.
%
%Variáveis de entrada:
% - Invars: a lista de pares < loop, invariante >; e
% - while( Teste, Corpo ): o loop a ter o invariante procurado.
%
%Variável de saída:
% - Invar: o invariante encontrado.
%=====

% se o par sendo analisado possui o loop cujo invariante é o que está sendo

```

```

% procurado, o invariante do par é a resposta
procurarInvar( [ inv( Loop, Invar ) | _Invars ], while( Teste, Corpo ),
              Invar ) :-
    Loop == while( Teste, Corpo )
.

% caso contrário, o invariante é procurado nos demais pares
procurarInvar( [ _LoopInv | Invars ], while( Teste, Corpo ), Invar ) :-
    procurarInvar( Invars, while( Teste, Corpo ), Invar )
.

%=====
%Nome: serIgual
%Descrição do predicado: analisa se uma pré-condição é equivalente a uma pós-
%                        condição após um comando de atribuição.
%
%
%Variáveis de entrada:
% - PreCond: a pré-condição;
% - Var:     a variável que recebe um valor na atribuição Var := Exp;
% - Exp:     a expressão que é atribuída na atribuição Var := Exp; e
% - PosCond: a pós-condição.
%=====

% a pré-condição é equivalente a pós-condição após o comando de atribuição
% porque é uma comparação entre uma expressão e uma variável. Esta regra serve
% para evitar o backtracking que permita pós-condições do tipo Var op Var, onde
% op é um operador condicional
serIgual( PreCondicao, Var, Exp, PosCondicao ) :-
    ( ( PreCondicao = ( Exp = Exp )           , % forma Var = Exp
      PosCondicao = ( Var = Exp )           ) ;
      ( PreCondicao = ( Exp <> Exp )         , % forma Var <> Exp
      PosCondicao = ( Var <> Exp )         ) ;
      ( PreCondicao = ( Exp < Exp )         , % forma Var < Exp
      PosCondicao = ( Var < Exp )         ) ;
      ( PreCondicao = ( Exp > Exp )         , % forma Var > Exp
      PosCondicao = ( Var > Exp )         ) ;
      ( PreCondicao = ( Exp <= Exp )       , % forma Var <= Exp
      PosCondicao = ( Var <= Exp )       ) ;
      ( PreCondicao = ( Exp >= Exp )       , % forma Var >= Exp
      PosCondicao = ( Var >= Exp )       ) )
    !
.

% neste caso, a pré-condição é equivalente a pós-condição após o comando de
% atribuição, porque são iguais e Var não aparece na pós-condição (se Var não
% aparece na pós-condição, nada tem que ser substituído)
serIgual( Cond, Var, _Exp, Cond ) :-
    \+ var( Cond )
    ,
    \+ acharVar( Var, Cond )
    !
.

% a pré-condição precisa ter a mesma forma de uma pós-condição para ser
% equivalente a mesma após um comando de atribuição, e este predicado verifica
% se ambas são da forma ex( x, A ) ou all( x, A ), onde A também deve ser checado
% recursivamente para ver se possui esta propriedade. Mais ainda, a variável
% ligada não pode ocorrer na expressão para que a substituição não seja feita
% incorretamente
serIgual( PreCond, Var, Exp, PosCond ) :-
    ( ( PreCond = ex( VarLig, PreCondLig )   ,
      PosCond = ex( VarLig, PosCondLig ) ) ;
      ( PreCond = all( VarLig, PreCondLig ) ,
      PosCond = all( VarLig, PosCondLig ) ) )
    ,
    \+ acharVar( VarLig, Exp )
    ,
    serIgual( PreCondLig, Var, Exp, PosCondLig )
.

% a pré-condição precisa ter a mesma forma de uma pós-condição para ser
% equivalente a mesma após um comando de atribuição, e este predicado verifica
% se ambas são da forma A op B, onde A e B também devem ser checados
% recursivamente para ver se possuem esta propriedade
serIgual( PreCond, Var, Exp, PosCond ) :-
    PosCond =.. [ Oper, Exp_1, Exp_2 ] ,
    PreCond =.. [ Oper, Exp_3, Exp_4 ] ,
    Oper \== all
    ,
    Oper \== ex
    ,
    Oper \== vet
    ,
    serIgual( Exp_3, Var, Exp, Exp_1 ) ,
    serIgual( Exp_4, Var, Exp, Exp_2 )
.

% a pré-condição precisa ter a mesma forma de uma pós-condição para ser

```

```

% equivalente a mesma após um comando de atribuição, e este predicado verifica
% se ambas são da forma not A, onde A também deve ser checado recursivamente
% para ver se possui esta propriedade
serIguar( not PreCond, Var, Exp, not PosCond ) :-
    serIguar( PreCond, Var, Exp, PosCond ) .

% a pré-condição precisa ter a mesma forma de uma pós-condição para ser
% equivalente a mesma após um comando de atribuição, e este predicado verifica
% se ambas são da forma - A, onde A também deve ser checado recursivamente para
% ver se possui esta propriedade
serIguar( - PreCond, Var, Exp, - PosCond ) :-
    serIguar( PreCond, Var, Exp, PosCond ) .

% neste caso, a pré-condição é equivalente a pós-condição após o comando de
% atribuição, com Exp sendo substituído por Var na pré-condição
serIguar( Exp, Var, Exp, Var ) .

%=====
%Nome: procurarIgualdade
%Descrição do predicado: ver se uma condição possui um termo que determina o
%                          valor de uma determinada variável.
%
%
%Variáveis de entrada:
% - Var:          a variável cujo valor se deseja encontrar; e
% - Condicao:     a condição onde se deseja procurar o valor da variável Var.
%
%Variáveis de saída:
% - CondRes:     a condição de entrada sem a conjunção que associa a variável
%                procurada a seu respectivo valor; e
% - Expressao:   o valor da variável procurado.
%=====

% para se achar o valor da variável, deve-se procurar nas conjunções da condição
% seu valor. Se uma delas for igual a true, esta parte é suprimida.
procurarIgualdade( Var, Cond_1 and Cond_2, CondRes, Expressao ) :-
    procurarIgualdade( Var, Cond_1, Cond_3, Expressao ) ,
    procurarIgualdade( Var, Cond_2, Cond_4, Expressao ) ,
    ( Cond_3 == true ->
      CondRes = Cond_4 ;
      ( Cond_4 == true ->
        CondRes = Cond_3 ;
        CondRes = Cond_3 and Cond_4 ) )
    ! .

% procura o valor da variável dentro do quantificador. Ela não pode estar
% ligada nem a variável ligada pode estar presente na expressão que define o
% valor da variável cujo valor se deseja achar
procurarIgualdade( Var, Cond, CondRes, Exp ) :-
    ( ( Cond = ex( VarLig, CondLig ) ,
      CondRes = ex( VarLig, CondResLig ) ) ;
      ( Cond = all( VarLig, CondLig ) ,
        CondRes = all( VarLig, CondResLig ) ) ) ,
    Var \== VarLig ,
    procurarIgualdade( Var, CondLig, CondResLig, Exp ) ,
    nonvar( Exp ) ,
    \+ acharVar( VarLig, Exp )
    ! .

% se a variável for achada, a expressão que determina seu valor é removida da
% condição
procurarIgualdade( Var, Igualdade, true, Expressao ) :-
    ( Igualdade = ( Var = Expressao ) ; % Var = Expressao
      Igualdade = ( Expressao = Var ) ;
      ( Igualdade = Var , % Var = true
        Expressao = true ) ;
      ( Igualdade = ( not Var ) , % Var = false
        Expressao = false ) )
    ! .

% o valor da variável não foi encontrado
procurarIgualdade( _Var, Condicao, Condicao, _Expressao ) .

%=====
%Nome: renomearLig
%Descrição do predicado: renomear uma variável ligada através de um

```

```

%
%               quantificador para que na hora da substituição de
%               variáveis não ocorram substituições erradas.
%
%
%Variáveis de entrada:
% - Var:       a variável que será substituída;
% - Exp:       a expressão que substituirá a variável Var; e
% - Condicao:  a condição onde a variável Var deverá ser substituída pela
%             expressão Exp.
%
%
%Variável de saída:
% - NovaCondicao: a condição com variáveis ligadas renomeadas de forma que a
%               substituição da variável Var pela expressão Exp não cause
%               uma substituição inválida.
%=====

% como a variável Var não está presente na condicao Condicao, nada precisa ser
% substituído e, portanto, não é necessário renomear nenhuma variável ligada
renomearLig( Var, _Exp, Condicao, Condicao ) :-
    \+ acharVar( Var, Condicao )
    !
    .

% a condição é do tipo A op B, onde op é um operador de comparação e ambas as
% partes da condição precisam ser tratadas recursivamente
renomearLig( Var, Exp, Condicao, NovaCondicao ) :-
    ( ( Condicao = ( Cond_1 and Cond_2 )           , % forma A and B
      NovaCondicao = ( Cond_3 and Cond_4 ) ) ) ;
    ( Condicao = ( Cond_1 or Cond_2 )             , % forma A or B
      NovaCondicao = ( Cond_3 or Cond_4 ) ) ) ;
    ( Condicao = ( Cond_1 => Cond_2 )             , % forma A => B
      NovaCondicao = ( Cond_3 => Cond_4 ) ) ) ;
    ( Condicao = ( Cond_1 <=> Cond_2 )           , % forma A <=> B
      NovaCondicao = ( Cond_3 <=> Cond_4 ) ) ) ;
    ( Condicao = ( Cond_1 = Cond_2 )             , % forma A = B
      NovaCondicao = ( Cond_3 = Cond_4 ) ) ) ;
    ( Condicao = ( Cond_1 <> Cond_2 )           , % forma A <> B
      NovaCondicao = ( Cond_3 <> Cond_4 ) ) ) ,
    renomearLig( Var, Exp, Cond_1, Cond_3 ) , % análise recursiva
    renomearLig( Var, Exp, Cond_2, Cond_4 ) . % análise recursiva

% a condição é uma negação, cuja parte negada deve ser tratada recursivamente
renomearLig( Var, Exp, not Condicao, not NovaCondicao ) :-
    renomearLig( Var, Exp, Condicao, NovaCondicao )
    .

% a variável ligada não está contida na expressão. Portanto, esta variável
% não precisa ser renomeada e o resto da condição deve ser tratado
% recursivamente
renomearLig( Var, Exp, Condicao, NovaCondicao ) :-
    ( ( Condicao = ex( VarLig, Cond )             ,
      NovaCondicao = ex( VarLig, NovaCond ) ) ) ;
    ( Condicao = all( VarLig, Cond )             ,
      NovaCondicao = all( VarLig, NovaCond ) ) ) ,
    \+ acharVar( VarLig, Exp )
    ,
    renomearLig( Var, Exp, Cond, NovaCond )
    ,
    !
    .

% a variável ligada está contida na expressão. Portanto, esta variável
% precisa ser renomeada e substituída na condição e a nova condição deve
% ser tratada recursivamente
renomearLig( Var, Exp, Condicao, NovaCondicao ) :-
    ( ( Condicao = ex( VarLig_1, Cond )           ,
      NovaCondicao = ex( VarLig_2, NovaCond ) ) ) ;
    ( Condicao = all( VarLig_1, Cond )           ,
      NovaCondicao = all( VarLig_2, NovaCond ) ) ) ,
    acharNovaVarLig( Var, Exp, Cond, VarLig_1, VarLig_2 ) ,
    serIgual( CondTemp, VarLig_1, VarLig_2, Cond ) ,
    renomearLig( Var, Exp, CondTemp, NovaCond )
    ,
    !
    .

% se a condição não estiver no formato dos casos anteriores, nada precisa
% ser feito com ela
renomearLig( _Var, _Exp, Cond, Cond ) .

%=====
%Nome: acharNovaVarLig
%Descrição do predicado: determinar o nome de uma nova variável para substituir
% uma variável ligada para que na hora da substituição de

```

```

%                               variáveis não ocorra substituições erradas.
%
%Variáveis de entrada:
% - Var:      a variável que será substituída;
% - Exp:      a expressão que substituirá a variável Var;
% - Cond:     a condição onde a variável Var deverá ser substituída pela
%             expressão Exp; e
% - VarLig_1: o nome original da variável ligada que deve ser modificado.
%
%Variável de saída:
% - VarLig_2: o novo nome da variável ligada.
%=====
% se a variável ligada VarLig é diferente da variável Var que será substituída
% pela expressão Exp e não aparece em Exp e na condição Cond, não precisa ser
% renomeada
acharNovaVarLig( Var, Exp, Cond, VarLig, VarLig ) :-
    Var \== VarLig      ,
    \+ acharVar( VarLig, Cond )      ,
    \+ acharVar( VarLig, Exp )      ,
    !                               .

% caso contrário, coloca-se um clique no nome da variável ligada e,
% recursivamente, testa-se se este novo nome satisfaz as condições de
% substituição dadas na regra acima
acharNovaVarLig( Var, Exp, Cond, VarLig_1, VarLig_2 ) :-
    name( VarLig_1, VarLista_1 )      ,
    append( VarLista_1, [ 39 ], VarListaTemp )      ,
    name( VarLigTemp, VarListaTemp )      ,
    acharNovaVarLig( Var, Exp, Cond, VarLigTemp, VarLig_2 ).

```

expressoes.pl

```

%=====
%NOME: expressoes.pl
%
%Projeto: Correção de Programas
%
% Versão: V 2.00
% Autora : Juliana Carpes Imperial - 0124811-7
%
%Descrição do módulo
% Objetivos:
% - converter uma expressão lida pelo analisador sintático para o formato
%   das condições utilizado durante a correção de programas;
% - procurar uma variável em uma expressão; e
% - renomear uma variável em uma expressão.
%=====
%
%Versão anterior: V 1.00
%Modificações:
% - adição de predicados para a implementação das heurísticas para automatizar
%   a correção de programas; e
% - colocar mais cláusulas para tratar vetores e o operador binário de módulo.
%=====
%Nome: converter
%Descrição do predicado: converte uma expressão lida pelo analisador sintático
%                        para o formato das condições utilizado durante a
%                        correção de programas.
%
%Variável de entrada:
% - a expressão a ser convertida.
%
%Variável de saída:
% - a expressão convertida para o mesmo formato de uma condição.
%=====
converter( num( Num ), Num ) . % converte um número

```

```

converter( var( Var ), Var ) . % converte uma variável
converter( true, true ) . % converte a constante booleana true
converter( false, false ) . % converte a constante booleana false

% converte uma soma, convertendo recursivamente seus operandos
converter( expInt( soma, Exp_1, Exp_2 ), Exp_3 + Exp_4 ) :-
    converter( Exp_1, Exp_3 )
    converter( Exp_2, Exp_4 )

% converte uma diferença, convertendo recursivamente seus operandos
converter( expInt( menos, Exp_1, Exp_2 ), Exp_3 - Exp_4 ) :-
    converter( Exp_1, Exp_3 )
    converter( Exp_2, Exp_4 )

% converte uma multiplicação, convertendo recursivamente seus operandos
converter( expInt( mult, Exp_1, Exp_2 ), Exp_3 * Exp_4 ) :-
    converter( Exp_1, Exp_3 )
    converter( Exp_2, Exp_4 )

% converte uma divisão, convertendo recursivamente seus operandos
converter( expInt( divide, Exp_1, Exp_2 ), Exp_3 / Exp_4 ) :-
    converter( Exp_1, Exp_3 )
    converter( Exp_2, Exp_4 )

% converte uma operação de módulo, convertendo recursivamente seus operandos
converter( expInt( mod, Exp_1, Exp_2 ), Exp_3 mod Exp_4 ) :-
    converter( Exp_1, Exp_3 )
    converter( Exp_2, Exp_4 )

% converte uma inversão de sinal, convertendo recursivamente a expressão a ter o
% sinal invertido
converter( menos( Exp_1 ), - Exp_2 ) :-
    converter( Exp_1, Exp_2 )

% converte um and lógico, convertendo recursivamente seus operandos
converter( expBool( and, Exp_1, Exp_2 ), Exp_3 and Exp_4 ) :-
    converter( Exp_1, Exp_3 )
    converter( Exp_2, Exp_4 )

% converte um or lógico, convertendo recursivamente seus operandos
converter( expBool( or, Exp_1, Exp_2 ), Exp_3 or Exp_4 ) :-
    converter( Exp_1, Exp_3 )
    converter( Exp_2, Exp_4 )

% converte uma negação, convertendo recursivamente a expressão negada
converter( neg( Exp_1 ), not Exp_2 ) :-
    converter( Exp_1, Exp_2 )

% converte uma comparação de igualdade, convertendo recursivamente seus
% operandos
converter( expBool( igual, Exp_1, Exp_2 ), Exp_3 = Exp_4 ) :-
    converter( Exp_1, Exp_3 )
    converter( Exp_2, Exp_4 )

% converte uma comparação de desigualdade, convertendo recursivamente seus
% operandos
converter( expBool( difer, Exp_1, Exp_2 ), Exp_3 <> Exp_4 ) :-
    converter( Exp_1, Exp_3 )
    converter( Exp_2, Exp_4 )

% converte uma comparação de maior, convertendo recursivamente seus operandos
converter( expBool( maior, Exp_1, Exp_2 ), Exp_3 > Exp_4 ) :-
    converter( Exp_1, Exp_3 )
    converter( Exp_2, Exp_4 )

% converte uma comparação de menor, convertendo recursivamente seus operandos
converter( expBool( menor, Exp_1, Exp_2 ), Exp_3 < Exp_4 ) :-
    converter( Exp_1, Exp_3 )
    converter( Exp_2, Exp_4 )

% converte uma comparação de maior ou igual, convertendo recursivamente seus
% operandos
converter( expBool( maiorIg, Exp_1, Exp_2 ), Exp_3 >= Exp_4 ) :-
    converter( Exp_1, Exp_3 )
    converter( Exp_2, Exp_4 )

% converte uma comparação de menor ou igual, convertendo recursivamente seus

```

```

% operandos
converter( expBool( menorIg, Exp_1, Exp_2 ), Exp_3 <= Exp_4 ) :-
    converter( Exp_1, Exp_3 )
    converter( Exp_2, Exp_4 )

%=====
%Nome: acharVar
%Descrição do predicado: procurar uma variável em uma expressão.
%
%Variáveis de entrada:
% - Var: a variável sendo procurada na expressão; e
% - Expressao: a expressão onde a variável está sendo procurada.
%=====

% para a variável ser encontrada na expressão, ela deve ser livre
acharVar( Var, Expressao ) :-
    ( Expressao = ex( VarLig, Exp ) ; % testa se é um existencial
      Expressao = all( VarLig, Exp ) ) , % testa se é um universal
    ! , % ex e all não podem ser operadores
    Var \== VarLig , % testa se ela é livre
    acharVar( Var, Exp ) . % faz a busca recursivamente

% a variável procurada é o endereço inicial do vetor
acharVar( Var, vet( Var, _Idx ) ) :-
    ! .

% se a variável procurada não é a do endereço inicial do vetor, a mesma é
% procurada recursivamente no índice dele
acharVar( Var, vet( _End, Idx ) ) :-
    \+ integer( Idx ) ,
    acharVar( Var, Idx ) .

% a variável procurada é um vetor e, portanto, os índices devem ser
% testados recursivamente
acharVar( vet( Var, Idx_1 ), vet( Var, Idx_2 ) ) :-
    ! ,
    ( acharVar( Idx_1, Idx_2 ) ;
      Idx_1 == Idx_2 ) .

% se a expressão for da forma A op B, A e B devem ser verificados recursivamente
% para se procurar a variável sendo buscada
acharVar( Var, Expressao ) :-
    Expressao =.. [ _Oper, Exp_1, Exp_2 ] , % testa se é da forma A op B
    ( acharVar( Var, Exp_1 ) ; % faz a busca recursivamente
      acharVar( Var, Exp_2 ) ) . % faz a busca recursivamente

% se a expressão for da forma not A, A deve ser verificada recursivamente
% para se procurar a variável sendo buscada
acharVar( Var, not Expressao ) :-
    acharVar( Var, Expressao ) .

% se a expressão for da forma - A, A deve ser verificada recursivamente para
% se procurar a variável sendo buscada
acharVar( Var, - Expressao ) :-
    acharVar( Var, Expressao ) .

acharVar( Var, Var ) :- % a variável foi encontrada na expressão
    atom( Var ) .

%=====
%Nome: acharNovaVar
%Descrição do predicado: renomear uma variável em uma expressão para que a
% inclusão dela em uma expressão com uma variável ligada
% torne uma de suas variáveis erroneamente ligada.
%
%Variáveis de entrada:
% - Expressao: a expressão onde se quer renomear a variável; e
% - Var: a variável cujo nome está se testando para fazer a renomeação.
%
%Variável de saída:
% - NovaVar: a variável cujo nome será usado para fazer a renomeação
%=====

% se o nome da variável sendo testado não aparecer livre na expressão, pode ser
% utilizado na renomeação

```

```

acharNovaVar( Expressao, Var, Var ) :-
    \+ acharVar( Var, Expressao ) ,
    ! .

% se a variável que se quer renomear for um vetor, utiliza-se o nome de seu
% endereço para criar o novo nome de variável
acharNovaVar( Expressao, Var, NovaVar ) :-
    Var = vet( End, _Idx ) ,
    acharNovaVar( Expressao, End, NovaVar ) .

% caso contrário, coloca-se um clique no nome da variável sendo testado e,
% recursivamente, testa-se se este novo nome satisfaz a condição dada na regra
% acima
acharNovaVar( Expressao, Var, NovaVar ) :-
    name( Var, VarLista ) ,
    append( VarLista, [ 39 ], VarListaTemp ) ,
    name( VarTemp, VarListaTemp ) ,
    acharNovaVar( Expressao, VarTemp, NovaVar ) .

```

erro.pl

```

%=====
%NOME: erro.pl
%
%Projeto: Correção de Programas
%
% Versão: V 1.00
% Autora : Juliana Carpes Imperial - 0124811-7
%
%Descrição do módulo
% Objetivo:
% - tratar erro durante as operações de abertura, leitura, escrita e
% fechamento em arquivos.
%=====

%=====
%Nome: tratarErro
%Descrição do predicado: trata o erro que aconteceu durante as operações com
% arquivos.
%
%Variável de entrada:
% - Erro: o tipo do erro que aconteceu.
%
%Assertiva de entrada:
% - o arquivo com o programa LACC está aberto.
%
%Assertiva de saída:
% - o arquivo com o programa LACC está fechado e o programa foi abortado.
%=====

% exibe as mensagens adequadas para cada tipo de erro (abertura, leitura,
% escrita e fechamento de arquivo) e, por fim, fecha o arquivo e aborta o
% programa
tratarErro( Erro ) :-
    catch( seen, _Excecao, true ) ,
    catch( told, _Excecao, true ) ,
    ( ( Erro == abertura ,
        write( '\nErro durante a abertura do arquivo.\n' ) ) ;
      ( Erro == escrita ,
        write( '\nErro durante a escrita no arquivo.\n' ) ) ;
      ( Erro == leitura ,
        write( '\nErro durante a leitura do arquivo.\n' ) ) ;
      ( Erro == fechamento ,
        write( '\nErro durante o fechamento do arquivo.\n' ) ) ) ,
    abort .

```

hoare.pl

```

%=====
%NOME: hoare.pl
%
%Projeto: Correção de Programas
%
% Versão: V 2.00
% Autora : Juliana Carpes Imperial - 0124811-7
%
%Descrição do módulo
%  Objetivos:
%    - implementação das regras de inferência do cálculo de Hoare para fazer
%    a correção de programas escritos em LACC; e
%    - implementação de um filtro que retira provas de programas que possuem
%    o conjunto de sentenças a ser provado igual a de outras.
%=====

%=====
%Versão anterior: V 1.00
%Modificações:
%  - adição de predicados para a implementação das heurísticas para automatizar
%  a correção de programas; e
%  - retirada do predicado que coloca a saída do programa em uma página HTML.
%=====

%=====
%Nome: fazerHoare
%Descrição do predicado: aplica as regras de inferência do cálculo de hoare para
% fazer a correção do programa em LACC.
%
%
%Variáveis de entrada:
%  - Invars: a lista de pares < loop, invariante >;
%  - PreCond: a pré-condição para o trecho de programa sendo corrigido;
%  - Cmds: o trecho de programa sendo corrigido;
%  - PosCond: a pós-condição para o trecho de programa sendo corrigido; e
%  - SentsAnt: sentenças a serem demonstradas antes de se aplicar a regra.
%
%Variáveis de saída:
%  - Prova: a árvore de prova resultante da aplicação da regra; e
%  - SentsDep: sentenças a serem demonstradas depois da aplicação da regra.
%=====

% comando skip; regra é axioma:  $\frac{}{\{ P \} \text{ skip } \{ P \}}$ 
%
% nesse caso, ou uma das condições não está ligada ou ambas são iguais
fazerHoare( _Invars, Cond, [ skip ], Cond, Prova, Sents, Sents ) :-
    Prova = [ [ Cond, [ skip ], Cond ] ]
    !

% nesse caso, as condições não são iguais. Portanto, é necessário fortalecer a
% pré-condição
fazerHoare( _Invars, PreCond, [ skip ], PosCond, Prova, SentsAnt, SentsDep ) :-
    Prova = [ [ PreCond, [ skip ], PosCond ], [ PreCond => PosCond ],
              [ [ PosCond, [ skip ], PosCond ] ] ]
    SentsDep = [ PreCond => PosCond | SentsAnt ]
    !

% comando de atribuição; regra é axioma:  $\frac{}{\{ P[ x/E ] \} x := E \{ P \}}$ 
%
% nesse caso, a variável a receber o valor é uma posição de um vetor e a
% pós-condição contém propriedades em relação ao vetor inteiro e não apenas
% à posição sendo modificada. Logo, é necessário achar uma nova pós-condição
% que contenha propriedades a respeito da posição modificada e que implique na
% anterior. Ainda, a pré-condição deve estar definida para o programa não
% entrar em loop
fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
            SentsDep ) :-
    nonvar( PosCond )
    converter( Exp, EquivExp )
    serIgual( CondTemp, Var, EquivExp, PosCond )
    CondTemp == PosCond

```

```

Var = vet( End, _Idx )
acharVar( End, PosCond )
( nonvar( PreCond )
  ( !
    fail ) )
fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], NovaPosCond,
            ProvaFilha, SentsAnt, SentsTemp )
NovaPosCond \= ex( _VarLig, _CondLig )
ProvaPai = [ PreCond, [ atribui( Var, Exp ) ], PosCond ]
Prova = [ ProvaPai, ProvaFilha, [ NovaPosCond => PosCond ] ]
SentsDep = [ NovaPosCond => PosCond | SentsTemp ]
!

% nesse caso, a pós-condição é determinada, a substituição de variáveis pode ser
% feita sem a necessidade de renomeação de variáveis e a correção será feita de
% traz para frente. Se já houver uma pré-condição determinada, diferente da
% encontrada pela substituição, é necessário realizar o fortalecimento da
% pré-condição
fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
            SentsDep ) :-
  nonvar( PosCond )
  converter( Exp, EquivExp )
  serIgual( CondTemp, Var, EquivExp, PosCond )
  ProvaPai = [ PreCond, [ atribui( Var, Exp ) ], PosCond ]
  ( var( PreCond )
    PreCond = CondTemp
    true )
  ( PreCond == CondTemp
    ( Prova = [ ProvaPai ]
      SentsDep = SentsAnt )
    ( Prova = [ ProvaPai, [ PreCond => CondTemp ],
              [ [ CondTemp, [ atribui( Var, Exp ) ], PosCond ] ] ]
      SentsDep = [ PreCond => CondTemp | SentsAnt ] ) )
  !

% nesse caso, a pós-condição é determinada e a substituição de variáveis não
% pode ser feita sem a necessidade de renomeação de variáveis. Portanto, uma
% nova pós-condição com as necessárias renomeações de variáveis é encontrada,
% a antiga é enfraquecida para ser igual a nova e a correção é feita
% recursivamente
fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
            SentsDep ) :-
  nonvar( PosCond )
  converter( Exp, EquivExp )
  renomearLig( Var, EquivExp, PosCond, NovaPosCond )
  PosCond \== NovaPosCond
  ProvaPai = [ PreCond, [ atribui( Var, Exp ) ], PosCond ]
  !
  fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], NovaPosCond,
              ProvaFilha, SentsAnt, SentsTemp )
  Prova = [ ProvaPai, ProvaFilha, [ NovaPosCond => PosCond ] ]
  SentsDep = [ NovaPosCond => PosCond | SentsTemp ]

% nesse caso, a pós-condição não é determinada e a atribuição é do tipo
% { P } x := E { Q }. Nesta situação é possível encontrar a pós-condição
% mais forte. Se P = true ou P = ( E = E ), Q = ( x = E ). Senão,
% Q = P and ( x = E ). Se x for a posição de um vetor e seu endereço estiver
% em P, a heurística não pode ser aplicada para não gerar contradições
fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
            SentsDep ) :-
  \+ acharVar( Var, PreCond )
  ( Var = vet( End, _Idx )
    \+ acharVar( End, PreCond )
    true )
  converter( Exp, EquivExp )
  \+ acharVar( Var, EquivExp )
  Igualdade = ( EquivExp = EquivExp )
  ( ( PreCond == true
    PreCond == Igualdade )
    PosCond = ( Var = EquivExp )
    PosCond = PreCond and ( Var = EquivExp ) )
  ProvaPai = [ PreCond, [ atribui( Var, Exp ) ], PosCond ]
  ( ( PreCond == true
    Prova = [ ProvaPai, [ true => Igualdade ],
              [ [ Igualdade, [ atribui( Var, Exp ) ], PosCond ] ] ]
    SentsDep = [ true => Igualdade | SentsAnt ] )
    ( PreCond == Igualdade
  )

```

```

    Prova = [ ProvaPai ]
    SentsDep = SentsAnt
    ( Prova = [ ProvaPai, [ PreCond => ( PreCond and Igualdade ) ],
      [ [ PreCond and Igualdade, [ atribui( Var, Exp ) ],
        PosCond ] ] ]
      SentsDep = [ PreCond => ( PreCond and Igualdade ) | SentsAnt ] ) )
!

% nesse caso, a pós-condição não é determinada e a atribuição é do tipo
% { P } x := E( x ) { Q }. Nesta situação é possível encontrar a pós-condição
% mais forte. Se P = true ou P = ex( y, E( x ) = E( y ) ),
% Q = ex( y, x = E( y ) ). Senão, Q = P and ex( y, x = E( y ) ). Se x for a
% posição de um vetor e seu endereço estiver em P, a heurística não pode ser
% aplicada para não gerar sentenças inválidas
fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
  SentsDep ) :-
  \+ acharVar( Var, PreCond )
  ( Var = vet( End, _Idx )
    \+ acharVar( End, PreCond )
    true )
  converter( Exp, EquivExp )
  acharVar( Var, EquivExp )
  acharNovaVar( EquivExp, Var, NovaVar )
  serIgual( NovaExp, Var, NovaVar, EquivExp )
  NovoTermo = ex( NovaVar, EquivExp = NovaExp )
  ( ( PreCond == true
    PreCond == NovoTermo )
    PosCond = ex( NovaVar, Var = NovaExp )
    PosCond = PreCond and ex( NovaVar, Var = NovaExp ) )
  ProvaPai = [ PreCond, [ atribui( Var, Exp ) ], PosCond ]
  ( ( PreCond == true
    Prova = [ ProvaPai, [ true => NovoTermo ],
      [ [ NovoTermo, [ atribui( Var, Exp ) ], PosCond ] ] ]
    SentsDep = [ true => NovoTermo | SentsAnt ] )
  ( PreCond == NovoTermo
    Prova = [ ProvaPai ]
    SentsDep = SentsAnt )
  ( Prova = [ ProvaPai, [ PreCond => ( PreCond and NovoTermo ) ],
    [ [ PreCond and NovoTermo, [ atribui( Var, Exp ) ],
      PosCond ] ] ]
    SentsDep = [ PreCond => ( PreCond and NovoTermo ) | SentsAnt ] ) )
!

% nesse caso, a pós-condição não é determinada e a atribuição é do tipo
% { P( x ) } x := E { Q } ou { P( x ) } x := E( x ) { Q }. Nesta situação é
% possível encontrar a pós-condição mais forte se for possível encontrar o
% valor de x na pré-condição, o qual será chamado de a. Se P = ( x = a ),
% Q = ( x = E ) ou Q = ( x = E( a ) ), respectivamente. Senão,
% Q = P( a ) and ( x = E ) ou Q = P( a ) and ( x = E( a ) ), respectivamente.
fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
  SentsDep ) :-
  acharVar( Var, PreCond )
  converter( Exp, EquivExp )
  procurarIgualdade( Var, PreCond, NovaCond, Expressao )
  nonvar( Expressao )
  serIgual( ExpSubst, Var, Expressao, EquivExp )
  ( NovaCond == true
    ( NovaPreCond = ( EquivExp = ExpSubst )
      PosCond = ( Var = ExpSubst ) )
    ( serIgual( CondTemp, Var, Expressao, NovaCond )
      NovaPreCond = ( CondTemp and ( EquivExp = ExpSubst ) )
      PosCond = ( CondTemp and ( Var = ExpSubst ) ) ) )
  ProvaPai = [ PreCond, [ atribui( Var, Exp ) ], PosCond ]
  ProvaFilha = [ NovaPreCond, [ atribui( Var, Exp ) ], PosCond ]
  Prova = [ ProvaPai, [ PreCond => NovaPreCond ], [ ProvaFilha ] ]
  SentsDep = [ PreCond => NovaPreCond | SentsAnt ]
!

% nesse caso, a pós-condição não é determinada e a atribuição é do tipo
% { P( x ) } x := E { Q } ou { P( x ) } x := E( x ) { Q }. Nesta situação é
% possível encontrar a pós-condição mais forte através da inserção do
% quantificador existencial já que é difícil ou impossível determinar o valor de
% x em P( x )
fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
  SentsDep ) :-
  acharVar( Var, PreCond )
  converter( Exp, EquivExp )

```

```

acharNovaVar( PreCond, Var, VarTemp )
( acharVar( Var, EquivExp )
  acharNovaVar( EquivExp, VarTemp, NovaVar )
  NovaVar = VarTemp )
serIgual( NovaCond, Var, NovaVar, PreCond )
( acharVar( Var, EquivExp )
  ( serIgual( ExpSubst, Var, NovaVar, EquivExp )
    NovaPreCond = ex( NovaVar, NovaCond and ( EquivExp = ExpSubst ) )
    PosCond = ex( NovaVar, NovaCond and ( Var = ExpSubst ) ) )
  ( NovaPreCond = ( ex( NovaVar, NovaCond ) and ( EquivExp = EquivExp ) ) )
  PosCond = ( ex( NovaVar, NovaCond ) and ( Var = EquivExp ) ) )
  ProvaPai = [ PreCond, [ atribui( Var, Exp ) ], PosCond ]
  ProvaFilha = [ NovaPreCond, [ atribui( Var, Exp ) ], PosCond ]
  Prova = [ ProvaPai, [ PreCond => NovaPreCond ], [ ProvaFilha ] ]
  SentsDep = [ PreCond => NovaPreCond | SentsAnt ]
!

% se nenhuma das heurísticas da atribuição pôde ser aplicada, então é necessário
% pedir ajuda ao usuário, pedindo para ele a nova pós-condição. A partir desta,
% acha-se a nova pré-condição correspondente e monta-se a prova
fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
  SentsDep ) :-
  write( 'Pré-condição: ' )
  imprimirCond( PreCond )
  write( '\nComando: ' )
  write( atribui( Var, Exp ) )
  write( '\nEntre a nova pós-condição contendo ' )
  imprimirVetor( Var )
  write( ' explicitamente: ' )
  read( PosCond )
  checarCondicao( PosCond )
  converter( Exp, EquivExp )
  serIgual( NovaPreCond, Var, EquivExp, PosCond )
  ProvaPai = [ PreCond, [ atribui( Var, Exp ) ], PosCond ]
  ProvaFilha = [ NovaPreCond, [ atribui( Var, Exp ) ], PosCond ]
  Prova = [ ProvaPai, [ PreCond => NovaPreCond ], [ ProvaFilha ] ]
  SentsDep = [ PreCond => NovaPreCond | SentsAnt ]
!

% se a condição dada for inválida, deve-se pedir a pós-condição novamente
fazerHoare( Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
  SentsDep ) :-
  write( '\nNova pós-condição inválida! Tente de novo!\n' )
  fazerHoare( Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova,
    SentsAnt, SentsDep )
!

% regra para mais de um comando: { P } CMD1 { R } { R } CDM2 { Q }
%
%
%
% deve haver mais de um comando para que a regra funcione corretamente. São
% tentadas duas alternativas, uma começando pelo primeiro comando e outra
% pelos demais comandos

% se o último comando não é um if, a correção pode ser começada de traz para
% frente desde que a pós-condição seja determinada
fazerHoare( Invars, PreCond, Comandos, PosCond, Prova, SentsAnt, SentsDep ) :-
  length( Comandos, TamCmds )
  TamCmds > 1
  nonvar( PosCond )
  last( Comandos, Ultimo )
  append( RestCmds, [ Ultimo ], Comandos )
  Ultimo \= if( _Teste, _Entao )
  Ultimo \= if( _Teste, _Entao, _Senao )
  !
  fazerHoare( Invars, CondTemp, [ Ultimo ], PosCond, ProvaFilha_2, SentsAnt,
    SentsTemp )
  fazerHoare( Invars, PreCond, RestCmds, CondTemp, ProvaFilha_1, SentsTemp,
    SentsDep )
  Prova = [ [ PreCond, Comandos, PosCond ], ProvaFilha_1, ProvaFilha_2 ]

% caso contrário, desde que a pré-condição seja determinada, começa-se a prova
% a partir do primeiro comando
fazerHoare( Invars, PreCond, [ Cmd | Cmds ], PosCond, Prova, SentsAnt,
  SentsDep ) :-
  length( Cmds, TamCmds )
  TamCmds > 0

```

```

nonvar( PreCond )
!
fazerHoare( Invars, PreCond, [ Cmd ], CondTemp, ProvaFilha_1, SentsAnt ,
SentsTemp ) ,
fazerHoare( Invars, CondTemp, Cmds, PosCond, ProvaFilha_2, SentsTemp,
SentsDep ) ,
Prova = [ [ PreCond, [ Cmd | Cmds ], PosCond ], ProvaFilha_1,
ProvaFilha_2 ] .

% regra do if sem else: { P and B } CMD { Q } P and not B => Q
%
%
% 
$$\frac{\{ P \} \text{ if } B \text{ then } \text{CMD} \text{ fi } \{ Q \}}{\{ P \text{ and } B \} \text{ CMD } \{ Q \} \text{ P and not } B \Rightarrow Q}$$

% o teste do if deve ser convertido para o formato de condição antes da
% tentativa de aplicar a regra. A sentença a ser demonstrada no if é incluída
% no conjunto de sentenças.

% nesse caso, a pós-condição é conhecida e a prova será feita de traz para
% frente
fazerHoare( Invars, PreCond, [ if( Teste, Entao ) ], PosCond, Prova, SentsAnt,
SentsDep ) :-
nonvar( PosCond )
!
converter( Teste, EquivTeste )
fazerHoare( Invars, PreCond and EquivTeste, Entao, PosCond, ProvaFilha,
SentsAnt, SentsTemp ) ,
SentsDep = [ ( PreCond and not EquivTeste ) => PosCond | SentsTemp ] ,
Prova = [ [ PreCond, [ if( Teste, Entao ) ], PosCond ], ProvaFilha,
[ ( PreCond and not EquivTeste ) => PosCond ] ] .

% nesse caso, a pós-condição não é conhecida. Então ela, ao final da aplicação
% da regra, deve ser a disjunção da pós-condição do ramo then com a conjunção
% da pré-condição do if e da negação do teste do mesmo
fazerHoare( Invars, PreCond, [ if( Teste, Entao ) ], PosCond, Prova, SentsAnt,
SentsDep ) :-
converter( Teste, EquivTeste )
fazerHoare( Invars, PreCond and EquivTeste, Entao, PosCond_1, ProvaNeta,
SentsAnt, SentsTemp ) ,
PosCond_2 = ( PreCond and not EquivTeste )
PosCond = ( PosCond_1 or PosCond_2 )
ProvaFilha = [ [ PreCond and EquivTeste, Entao, PosCond ], ProvaNeta,
[ ( PosCond_1 => PosCond ) ] ]
SentsDep = [ ( PosCond_2 => PosCond ), ( PosCond_1 => PosCond )
| SentsTemp ]
Prova = [ [ PreCond, [ if( Teste, Entao ) ], PosCond ], ProvaFilha,
[ PosCond_2 => PosCond ] ] .

% regra do if com else: { P and B } CMD1 { Q } { P and not B } CMD2 { Q }
%
%
% 
$$\frac{\{ P \} \text{ if } B \text{ then } \text{CMD1} \text{ else } \text{CMD2} \text{ esle\_fi } \{ Q \}}{\{ P \text{ and } B \} \text{ CMD1 } \{ Q \} \{ P \text{ and not } B \} \text{ CMD2 } \{ Q \}}$$

% o teste do if deve ser convertido para o formato de condição antes da
% tentativa de aplicar a regra.

% nesse caso, a pós-condição é conhecida e a prova será feita de traz para
% frente
fazerHoare( Invars, PreCond, [ if( Teste, Entao, Senao ) ], PosCond, Prova,
SentsAnt, SentsDep ) :-
nonvar( PosCond )
!
converter( Teste, EquivTeste )
fazerHoare( Invars, PreCond and EquivTeste, Entao, PosCond, ProvaFilha_1,
SentsAnt, SentsTemp ) ,
fazerHoare( Invars, PreCond and not EquivTeste, Senao, PosCond,
ProvaFilha_2, SentsTemp, SentsDep ) ,
Prova = [ [ PreCond, [ if( Teste, Entao, Senao ) ], PosCond ],
ProvaFilha_1, ProvaFilha_2 ] .

% nesse caso, a pós-condição não é conhecida. Então ela, ao final da aplicação
% da regra, deve ser a disjunção das pós-condições de cada ramo.
fazerHoare( Invars, PreCond, [ if( Teste, Entao, Senao ) ], PosCond, Prova,
SentsAnt, SentsDep ) :-
converter( Teste, EquivTeste )
!
fazerHoare( Invars, PreCond and EquivTeste, Entao, PosCond_1, ProvaNeta_1,
SentsAnt, SentsTemp_1 ) ,
fazerHoare( Invars, PreCond and not EquivTeste, Senao, PosCond_2,
ProvaNeta_2, SentsTemp_1, SentsTemp_2 ) ,
PosCond = ( PosCond_1 or PosCond_2 )

```

```

ProvaFilha_1 = [ [ PreCond and EquivTeste, Entao, PosCond ], ProvaNeta_1,
                 [ ( PosCond_1 => PosCond ) ] ] ,
ProvaFilha_2 = [ [ PreCond and not EquivTeste, Senao, PosCond ],
                 ProvaNeta_2, [ ( PosCond_2 => PosCond ) ] ] ,
SentsDep = [ ( PosCond_1 => PosCond ), ( PosCond_2 => PosCond )
             | SentsTemp_2 ] ,
Prova = [ [ PreCond, [ if( Teste, Entao, Senao ) ], PosCond ],
          ProvaFilha_1, ProvaFilha_2 ] .

% regra do while:          { P and B } CMD { P }
%
%          { P } while B do CMD od { P and not B }
% o teste do while deve ser convertido para o formato de condição antes da
% tentativa de aplicar a regra.

% Na primeira, a pré-condição já é igual ao invariante ou ainda não foi
% determinada e a pós-condição é igual a conjunção do invariante com a negação
% do teste do while ou ainda não foi determinada. Tem-se a certeza de que é o
% invariante certo que está sendo usado pela consulta a tabela de invariantes
% para cada loop.
fazerHoare( Invars, Invar, [ while( Teste, Corpo ) ], Invar and not EquivTeste,
            Prova, SentsAnt, SentsDep ) :-
    procurarInvar( Invars, while( Teste, Corpo ), Invar )
    ,
    converter( Teste, EquivTeste )
    ,
    !
    ,
    fazerHoare( Invars, Invar and EquivTeste, Corpo, Invar, ProvaFilha,
                SentsAnt, SentsDep )
    ,
    Prova = [ [ Invar, [ while( Teste, Corpo ) ], Invar and not EquivTeste ],
              ProvaFilha ]
    .

% neste caso, a pré-condição é o invariante (P), mas a pós precisa ser
% fortalecida para ser igual a P and not B, para que a regra do while escrita
% acima possa ser aplicada. O teste do while deve ser convertido para o formato
% de condição antes da recursão e a pós-condição não pode ser variável livre
% para que a sentença a ser demonstrada seja bem formada. Esta é incluída no
% conjunto de sentenças.
fazerHoare( Invars, Invar, [ while( Teste, Corpo ) ], PosCond, Prova, SentsAnt,
            SentsDep ) :-
    procurarInvar( Invars, while( Teste, Corpo ), Invar )
    ,
    converter( Teste, EquivTeste )
    ,
    !
    ,
    fazerHoare( Invars, Invar, [ while( Teste, Corpo ) ],
                Invar and not EquivTeste, ProvaFilha, SentsAnt, SentsTemp )
    ,
    SentsDep = [ ( Invar and not EquivTeste ) => PosCond | SentsTemp ]
    ,
    Prova = [ [ Invar, [ while( Teste, Corpo ) ], PosCond ], ProvaFilha,
              [ ( Invar and not EquivTeste ) => PosCond ] ]
    .

% neste caso, a pré-condição não é o invariante (P), que antes precisa ser
% encontrado na lista de < loop, invariante > para que possa ser enfraquecido
% para ser igual a P, para que a regra do while escrita acima possa ser aplicada
% caso a pós também já esteja no formato correto. A pré-condição não pode ser
% variável livre para que a sentença a ser demonstrada seja bem formada. Esta é
% incluída no conjunto de sentenças.
fazerHoare( Invars, PreCond, [ while( Teste, Corpo ) ], PosCond, Prova,
            SentsAnt, SentsDep ) :-
    procurarInvar( Invars, while( Teste, Corpo ), Invar )
    ,
    !
    ,
    fazerHoare( Invars, Invar, [ while( Teste, Corpo ) ], PosCond, ProvaFilha,
                SentsAnt, SentsTemp )
    ,
    SentsDep = [ PreCond => Invar | SentsTemp ]
    ,
    Prova = [ [ PreCond, [ while( Teste, Corpo ) ], PosCond ],
              [ PreCond => Invar ], ProvaFilha ]
    .

```

principal.pl

```

%=====
%NOME: principal.pl
%
%Projeto: Correção de Programas
%
```

```

%   Versão: V 1.00
%   Autora : Juliana Carpes Imperial - 0124811-7
%
%Descrição do módulo
%   Objetivo:
%       - ser o módulo principal do corretor de programas.
%=====

%=====
%Descrição da consulta: carrega os demais arquivos com o código-fonte do
%                       corretor de programas no interpretador de Prolog.
%=====

:- [ 'erro.pl', 'lexica.pl', 'sintatica.pl', 'condicoes.pl', 'expressoes.pl' ] ,
    [ 'hoare.pl', 'html.pl' ] .

%=====
%Nome: fazerAnalise
%Descrição do predicado: executa todos os passos necessários para realizar a
%                       correção formal de um programa.
%=====

% faz a análise léxica e sintática de um programa. Posteriormente, lê as pré e
% pós condições, e os invariantes. Depois, aplica o cálculo de Hoare no programa
% para provar a sua correção. Por fim, pega a prova e coloca-a em uma página
% HTML
corrigirPrograma                                     :-
    fazerAnaLexica( Tokens )                          ,
    fazerAnaSintatica( Tokens, TokensVet, Arvore )    ,
    lerCondicoes( TokensVet, Invars, PreCond, PosCond ) ,
    fazerHoare( Invars, PreCond, Arvore, PosCond, Prova, [ ], Sents ) ,
    imprimirSaida( Prova, Sents )                    .

```

html.pl

```

%=====
%NOME: html.pl
%
%Projeto: Correção de Programas
%
%   Versão: V 1.00
%   Autora : Juliana Carpes Imperial - 0124811-7
%
%Descrição do módulo
%   Objetivo:
%       - implementação de predicados para criar uma página HTML contendo a prova
%         de correção do programa utilizando o cálculo de Hoare juntamente com as
%         sentenças que precisam ser demonstradas para que a prova seja válida.
%=====

%=====
%Nome: imprimirSaida
%Descrição do predicado: cria uma página HTML contendo várias provas de programa
%                       utilizando o cálculo de Hoare juntamente com as
%                       sentenças que precisam ser demonstradas para que as
%                       provas sejam válidas.
%
%Variável de entrada:
%   - Respostas: a lista de < provas, sentenças a serem demonstradas > com todas
%               as provas possuindo um conjunto de sentenças a serem
%               demonstradas diferentes entre si.
%
%Assertiva de saída:
%   - a página HTML foi criada com sucesso e seu arquivo foi fechado, ou foi
%     detectado um erro de abertura, escrita ou fechamento de arquivo o que leva
%     o programa a ser abortado.
%=====

% tenta abrir o arquivo da página HTML para escrita. Se der erro, este é

```

```

% tratado. Se o arquivo for aberto com sucesso, seu cabeçalho é escrito,
% juntamente com as provas e seus conjuntos de sentenças devidamente formatados,
% e, por fim, são escritas as tags do fim da página. Se todo este processo de
% escrita no arquivo for feito com sucesso, ele é fechado. Se der algum erro de
% escrita ou fechamento, este é tratado
imprimirSaida( Prova, Sents )                               :-
  catch( tell( 'saida.html' ), _Excecao, tratarErro( abertura ) )
  ,
  catch( write( '<HTML>\n<HEAD>' ), _Excecao, tratarErro( escrita ) )
  ,
  catch( write( '<TITLE> Prova de ' ), _Excecao, tratarErro( escrita ) )
  ,
  catch( write( 'Correção de Programas' ), _Excecao, tratarErro( escrita ) )
  ,
  catch( write( ' </TITLE> </HEAD>\n' ), _Excecao, tratarErro( escrita ) )
  ,
  catch( write( '<BODY> <B> <PRE>\n' ), _Excecao, tratarErro( escrita ) )
  ,
  catch( imprimirCorrecao( Prova, Sents ), _Excecao, tratarErro( escrita ) )
  ,
  catch( write( '\n</B> </PRE> </BODY>' ), _Excecao, tratarErro( escrita ) )
  ,
  catch( write( '\n<HTML>' ), _Excecao, tratarErro( escrita ) )
  ,
  catch( told, _Excecao, tratarErro( fechamento ) )
  .

%=====
%Nome: imprimirCorrecao
%Descrição do predicado: imprime a prova e suas sentenças a serem demonstradas
%                          com formatação HTML.
%
%
%Variáveis de entrada:
% - Prova: a prova a ser impressa; e
% - Sents: seu conjunto de sentenças a ser demonstrado.
%
%Assertiva de entrada:
% - O arquivo da página HTML está aberto.
%=====

% imprime uma prova em cálculo de hoare e o conjunto de sentenças a ser
% demonstrado dela caso não seja vazio
imprimirCorrecao( Prova, Sents )                               :-
  write( '<H4>Prova: </H4>\n' )
  ,
  imprimirHoare( Prova, 1 )
  ,
  ( Sents \== [ ]
  ->
    ( write( '\n<H4>Sentenças a serem provadas: </H4>' )
    ,
    imprimirSents( Sents ) )
  ;
  true )
  .

%=====
%Nome: imprimirHoare
%Descrição do predicado: imprime as provas de correção de programa com
%                          formatação HTML.
%
%
%Variáveis de entrada:
% - Provas: a árvore com a prova de correção de programa; e
% - Nivel: o nível que está sendo impresso a partir da raiz da árvore.
%
%Assertiva de entrada:
% - O arquivo da página HTML está aberto.
%=====

% está se imprimindo um axioma, que é um trecho de prova sem filhos (uma folha
% da árvore), o que inclui as regras do skip e de atribuição
imprimirHoare( [ ProvaPai ], Nivel ) :-
  imprimirProva( ProvaPai, Nivel ) .

% está se imprimindo um ramo de prova que só tem um filho (while). O nível de um
% filho é sempre maior do que o do pai em um
imprimirHoare( [ ProvaPai, ProvaFilha ], Nivel ) :-
  imprimirProva( ProvaPai, Nivel )
  ,
  imprimirHoare( ProvaFilha, Nivel + 1 )
  .

% está se imprimindo um ramo de prova que só tem dois filhos (regras do if,
% enfraquecimento de pré-condição, fortalecimento de pós-condição e para mais de
% um comando). O nível de um filho é sempre maior do que o do pai em um
imprimirHoare( [ ProvaPai, ProvaFilha_1, ProvaFilha_2 ], Nivel ) :-
  imprimirProva( ProvaPai, Nivel )
  ,
  imprimirHoare( ProvaFilha_1, Nivel + 1 )
  ,
  imprimirHoare( ProvaFilha_2, Nivel + 1 )
  .

%=====
%Nome: imprimirProva

```

```

%Descrição do predicado: imprime um ramo de prova de correção de programa com
%                          formatação HTML.
%
%
%Variáveis de entrada:
% - Ramo: um ramo da árvore de uma prova de correção de programa; e
% - Nivel: o nível que está sendo impresso a partir da raiz da árvore.
%
%Assertiva de entrada:
% - O arquivo da página HTML está aberto.
%=====

% o ramo é um trecho se programa sendo corrigido da forma { P } CMD { Q }
imprimirProva( [ PreCond, Prog, PosCond ], Nivel ) :-
    tab( ( Nivel - 1 ) * 3 ) ,
    write( '<FONT COLOR = "red">{ ' ) ,
    imprimirCond( PreCond ) ,
    write( ' } </FONT>' ) ,
    write( Prog ) ,
    write( '<FONT COLOR = "red"> { ' ) ,
    imprimirCond( PosCond ) ,
    write( ' } </FONT>\n' ) .

% o ramo é uma sentença a ser demonstrada da forma P => Q
imprimirProva( Condicao, Nivel ) :-
    Condicao = ( _Cond_1 => _Cond_2 ) ,
    tab( ( Nivel - 1 ) * 3 ) ,
    write( '<FONT COLOR = "blue">' ) ,
    imprimirCond( Condicao ) ,
    write( ' </FONT>' ) ,
    nl .

%=====
%Nome: imprimirSents
%Descrição do predicado: imprime uma lista de sentenças, uma em cada linha.
%
%Variável de entrada:
% - Sents: a lista de sentenças a ter seus elementos impressos.
%=====

imprimirSents( [ ] ) . % a lista já foi totalmente impressa

imprimirSents( [ Sent | Sents ] ) :-
    nl , % pula linha
    imprimirCond( Sent ) , % imprime uma sentença
    imprimirSents( Sents ) . % imprime o resto da lista recursivamente

%=====
%Nome: imprimirCond
%Descrição do predicado: imprime uma pré ou pós-condição, invariante, uma
%                          sentença a ser demonstrada ou uma parte desses.
%
%Variável de entrada:
% - Condição: a condição a ser impressa.
%=====

% a condição a ser impressa é da forma A <=> B, onde A e B são impressos
% recursivamente, sendo a condição impressa com a devida parentização caso
% necessário
imprimirCond( Cond_1 <=> Cond_2 ) :-
    ( Cond_1 = ( _Cond_3 <=> _Cond_4 ) ->
        ( write( '(' ) ,
          imprimirCond( Cond_1 ) ,
          write( ')' ) ) ;
        imprimirCond( Cond_1 ) ) ,
    write( ' <=> ' ) ,
    ( Cond_2 = ( _Cond_3 <=> _Cond_4 ) ->
        ( write( '(' ) ,
          imprimirCond( Cond_2 ) ,
          write( ')' ) ) ;
        imprimirCond( Cond_2 ) ) ,
    ! .

% a condição a ser impressa não é da forma A <=> B
imprimirCond( Condicao ) :-
    imprimirImpl( Condicao ) .

```

```

%=====
%Nome: imprimirImpl
%Descrição do predicado: imprime uma pré ou pós-condição, invariante, uma
%                          sentença a ser demonstrada ou uma parte desses.
%
%Variável de entrada:
%  Condição: a condição a ser impressa.
%=====

% a condição a ser impressa é da forma A => B, onde A e B são impressos
% recursivamente, sendo a condição impressa com a devida parentização caso
% necessário
imprimirImpl( Cond_1 => Cond_2 ) :-
  ( Cond_1 = ( _Cond_3 => _Cond_4 ) ->
    ( write( '(' )
      , imprimirImpl( Cond_1 )
      , write( ')' ) )
    ;
    imprimirImpl( Cond_1 )
  , write( ' => ' )
  , ( Cond_2 = ( _Cond_3 => _Cond_4 ) ->
    ( write( '(' )
      , imprimirImpl( Cond_2 )
      , write( ')' ) )
    ;
    imprimirImpl( Cond_2 ) )
  , !
  .

% a condição a ser impressa não é da forma A => B
imprimirImpl( Condicao ) :-
  imprimirOr( Condicao ) .

%=====
%Nome: imprimirOr
%Descrição do predicado: imprime uma pré ou pós-condição, invariante, uma
%                          sentença a ser demonstrada ou uma parte desses.
%
%Variável de entrada:
%  Condição: a condição a ser impressa.
%=====

% a condição a ser impressa é da forma A or B, onde A e B são impressos
% recursivamente
imprimirOr( Cond_1 or Cond_2 ) :-
  imprimirOr( Cond_1 )
  , write( ' or ' )
  , imprimirOr( Cond_2 )
  , !
  .

% a condição a ser impressa não é da forma A or B
imprimirOr( Condicao ) :-
  imprimirAnd( Condicao ) .

%=====
%Nome: imprimirAnd
%Descrição do predicado: imprime uma pré ou pós-condição, invariante, uma
%                          sentença a ser demonstrada ou uma parte desses.
%
%Variável de entrada:
%  Condição: a condição a ser impressa.
%=====

% a condição a ser impressa é da forma A and B, onde A e B são impressos
% recursivamente
imprimirAnd( Cond_1 and Cond_2 ) :-
  imprimirAnd( Cond_1 )
  , write( ' and ' )
  , imprimirAnd( Cond_2 )
  , !
  .

% a condição a ser impressa não é da forma A and B
imprimirAnd( Condicao ) :-
  imprimirElemBol( Condicao ) .

```

```

%=====
%Nome: imprimirElemBol
%Descrição do predicado: imprime uma pré ou pós-condição, invariante, uma
%                          sentença a ser demonstrada ou uma parte desses.
%
%
%Variável de entrada:
%  Condição: a condição a ser impressa.
%=====

% a condição a ser impressa é da forma A = B, A <> B, A >= B, A <= B, A < B ou
% A > B, onde A e B são impressos recursivamente, sendo e A e B expressões
% aritméticas
imprimirElemBol( Condição )      :-
    Condição =.. [ Op, Exp_1, Exp_2 ] ,
    checarExpressao( Exp_1 )      ,
    checarExpressao( Exp_2 )      ,
    ( Op == '='                    ;
      Op == '<>'                   ;
      Op == '>='                  ;
      Op == '<='                  ;
      Op == '<'                   ;
      Op == '>'                   ) ,
    imprimirSomDif( Exp_1 )       ,
    write( ' ' )                  ,
    write( Op )                   ,
    write( ' ' )                  ,
    imprimirSomDif( Exp_2 )       ,
    !                              .

% a condição a ser impressa é da forma A = B ou A <> B, onde A e B são impressos
% recursivamente, sendo a condição impressa com a devida parentização caso
% necessário, e A e B expressões booleanas
imprimirElemBol( Condição )      :-
    Condição =.. [ Op, Cond_1, Cond_2 ] ,
    ( Op == '='                    ;
      Op == '<>'                   ) ,
    ( ( Cond_1 = ( _Cond_3 = _Cond_4 ) ;
      Cond_1 = ( _Cond_3 <> _Cond_4 ) ) ->
      ( write( '(' )                ,
        imprimirElemBol( Cond_1 )   ,
        write( ')' )                ) ;
      imprimirElemBol( Cond_1 )     ) ,
    write( ' ' )                   ,
    write( Op )                     ,
    write( ' ' )                   ,
    ( ( Cond_2 = ( _Cond_3 = _Cond_4 ) ;
      Cond_2 = ( _Cond_3 <> _Cond_4 ) ) ->
      ( write( '(' )                ,
        imprimirElemBol( Cond_2 )   ,
        write( ')' )                ) ;
      imprimirElemBol( Cond_2 )     ) ,
    !                              .

% a condição a ser impressa é um átomo (variável ou as constantes true ou false)
imprimirElemBol( Condição ) :-
    atom( Condição ) ,
    write( Condição ) ,
    !                .

% a condição a ser impressa é da forma ex( x, A ) ou all( x, A ), onde A é
% impresso recursivamente
imprimirElemBol( Condição ) :-
    ( ( Condição = ex( Var, Cond ) ,
      write( 'ex( ' ) ) ,
      ( Condição = all( Var, Cond ) ,
        write( 'all( ' ) ) ) ,
      write( Var ) ,
      write( ', ' ) ,
      imprimirCond( Cond ) ,
      write( ' )' ) ) ,
    !                .

% a condição a ser impressa é um vetor e será exibida da forma a[ i ]
imprimirElemBol( vet( Var, Idx ) ) :-
    imprimirVetor( vet( Var, Idx ) ) ,
    !                .

```

```

% a condição a ser impressa é da forma not A, onde A é impresso recursivamente
% e A não é atômico (ou seja, é da forma B op C) e, por isso, é impressa
% devidamente parentizada
imprimirElemBol( not Condicao )      :-
    Condicao =.. [ Op, _Cond_1, _Cond_2 ] ,
    Op \== all                       ,
    Op \== ex                        ,
    Op \== vet                       ,
    write( 'not ( ' )                ,
    imprimirCond( Condicao )          ,
    write( ' )' )                    ,
    !                                .

% a condição a ser impressa é da forma not A, onde A é impresso recursivamente
% e A é atômico (ou seja, não é da forma B op C)
imprimirElemBol( not Condicao ) :-
    write( 'not ' )                  ,
    imprimirElemBol( Condicao )      ,
    !                                .

% a condição a ser impressa precisa ser parentizada para que não fique ambígua
imprimirElemBol( Condicao ) :-
    write( '( ' )                    ,
    imprimirCond( Condicao )          ,
    write( ' )' )                    .

=====
%Nome: imprimirSomDif
%Descrição do predicado: imprime as operações de adição e subtração de uma
%                        expressão inteira.
%
%
%Variável de entrada:
%   Expressao: a expressão a ser impressa.
=====

% a expressão a ser impressa é uma soma ou uma subtração, e parênteses podem vir
% a ser necessários no caso da subtração para evitar ambiguidades
imprimirSomDif( Expressao )      :-
    Expressao =.. [ Op, Exp_1, Exp_2 ] ,
    ( Op == '+'                       ;
      Op == '-' )                    ,
    imprimirSomDif( Exp_1 )          ,
    write( ' ' )                      ,
    write( Op )                       ,
    write( ' ' )                      ,
    ( ( Op == '-'
      \+ atom( Exp_2 )
      \+ integer( Exp_2 ) )          ->
      ( write( '( ' )
        imprimirSomDif( Exp_2 )
        write( ' )' ) )
      imprimirSomDif( Exp_2 ) )      ,
    !                                .

% nada mais há para ser impresso de soma e subtração neste nível, portanto,
% passa-se a imprimir multiplicações, divisões e operações de módulo
imprimirSomDif( Expressao )      :-
    imprimirMulDiv( Expressao ) .

=====
%Nome: imprimirMulDiv
%Descrição do predicado: imprime as operações de multiplicação, divisão e módulo
%                        de uma expressão inteira.
%
%
%Variável de entrada:
%   Expressao: a expressão a ser impressa.
=====

% a expressão a ser impressa é uma multiplicação, divisão ou uma operação de
% módulo, e parênteses podem vir a ser necessários no caso da subtração para
% evitar ambiguidades
imprimirMulDiv( Expressao )      :-
    Expressao =.. [ Op, Exp_1, Exp_2 ] ,
    ( Op == '*'                       ;
      Op == '/' )                      ;

```

```

Op == mod )
( ( Exp_1 = ( _Exp_3 * _Exp_4 ) ;
  Exp_1 = ( _Exp_3 / _Exp_4 ) ;
  Exp_1 = ( _Exp_3 mod _Exp_4 ) ) ->
( write( ' ( ' ) ;
  imprimirMulDiv( Exp_1 ) ;
  write( ' )' ) ;
  imprimirMulDiv( Exp_1 ) ;
write( ' ' ) ;
write( Op ) ;
write( ' ' ) ;
( ( Exp_2 = ( _Exp_3 * _Exp_4 ) ;
  Exp_2 = ( _Exp_3 / _Exp_4 ) ;
  Exp_2 = ( _Exp_3 mod _Exp_4 ) ) ->
( write( ' ( ' ) ;
  imprimirMulDiv( Exp_2 ) ;
  write( ' )' ) ;
  imprimirMulDiv( Exp_2 ) ;
!
.

% nada mais há para ser impresso de multiplicação, divisão e módulo neste
% nível, portanto, passa-se a imprimir trocas de sinal, átomos, inteiros e
% vetores
imprimirMulDiv( Expressao ) :-
  imprimirElem( Expressao ) .

%=====
%Nome: imprimirElem
%Descrição do predicado: imprime trocas de sinal, átomos, inteiros e vetores de
% expressões inteiras.
%
%
%Variável de entrada:
% Condição: a condição a ser impressa.
%=====

% imprime uma troca de sinal onde a expressão a ser impressa não é atômica, o
% que torna necessária a parentização
imprimirElem( - Expressao ) :-
  Expressao =.. [ Op, _Cond_1, _Cond_2 ] ,
  Op \== vet ;
  write( '- ( ' ) ;
  imprimirSomDif( Expressao ) ;
  write( ' )' ) ;
!
.

% imprime uma troca de sinal em uma expressão atômica
imprimirElem( - Expressao ) :-
  write( '- ' ) ;
  imprimirElem( Expressao ) ;
!
.

% a condição a ser impressa é um átomo (variável) ou um inteiro
imprimirElem( Expressao ) :-
  ( atom( Expressao ) ;
  integer( Expressao ) ) ,
  write( Expressao ) ;
!
.

% a condição a ser impressa é um vetor e será exibida da forma a[ i ]
imprimirElem( vet( Var, Idx ) ) :-
  imprimirVetor( vet( Var, Idx ) ) ;
!
.

% a expressão a ser impressa precisa ser parentizada para que não fique ambígua
imprimirElem( Expressao ) :-
  write( ' ( ' ) ;
  imprimirSomDif( Expressao ) ;
  write( ' )' ) ;
.

%=====
%Nome: imprimirVetor
%Descrição do predicado: imprime um vetor no formato a[ i ].
%
%
%Variável de entrada:
% vet( Var, Idx ) : a condição a ser impressa.

```

```
%=====
% imprime o endereço do vetor e o índice recursivamente, já que o índice pode
% ser outro vetor
imprimirVetor( vet( Var, Idx ) ) :-
    write( Var )           ,
    write( '[' )           ,
    imprimirVetor( Idx )   ,
    write( ']' )           ,
    !                       .

% o índice é apenas uma variável ou um número, portanto pode ser impresso
% diretamente
imprimirVetor( Idx ) :-
    write( Idx )           .
```