

NewProplog - *software* Interpretador de Lógica Proposicional com método de inferência baseado em resolução¹

Athila Augusto Oliveira e Silva²

Resumo

É proposto o desenvolvimento de um *software* que permita interpretar regras e fatos da lógica proposicional, com método de inferência baseado em resolução, para facilitar a compreensão de iniciantes no ensino de Lógica Matemática e Computacional e Linguagens Lógicas. É também detalhada a resolução e a forma com que as regras são interpretadas, bem como as estruturas de dados utilizadas para a inferência das cláusulas. Por fim, é formalizado a construção de um interpretador para cláusulas de lógica proposicional.

Palavras-Chave

Resolução, Cláusulas de Horn; Refutação; Prolog; Proplog; Lógica; Interpretador.

Abstract

It is considered the development of a software that allows to interpret rules and facts of the propositional logic with method of inference based on resolution to facilitate the understanding of beginning in the education of Mathematical and Computational Logic and Logical Languages. Also it is detailed the resolution and the form with that the rules are interpreted, as well as the structures of data used for the inference of the clauses. Finally the formalization of the construction of the interpreter in a general vision.

KeyWords

Resolution, Horn Clauses; Refutation; Prolog; Proplog; Logic; Interpreter.

¹ Trabalho desenvolvido na Disciplina de Tópicos Especiais VII - Universidade de Uberaba.

² Aluno do Curso de Bacharelado em Sistemas de Informação – Universidade de Uberaba.

1 Introdução

Segundo [GERSTING], algumas linguagens de programação, ao invés de procedurais, são declarativas ou descritivas. Linguagens declarativas são aquelas baseadas na lógica de predicados (cuja lógica proposicional faz parte). Tal linguagem já vem equipada com suas próprias regras de inferência. O intuito deste artigo é abordar o paradigma da lógica proposicional por meio da proposta de uma linguagem declarativa que vem mostrar como implementar este tipo de software, com características diferentes dos interpretadores de programas estruturados. Assim, a interpretação permitirá ao usuário inferir perguntas, procurando informações sobre conclusões possíveis, dedutíveis das hipóteses do programa a ser interpretado.

Segundo [MAIER], o Proplog é um subconjunto do Prolog que se estrutura em lógica proposicional. No presente trabalho o termo proplog será substituído pela denominação NewProplog por ter seus meios de implementação diferenciados dos utilizados pelo autor.

O NewProplog é um projeto de software que foi dividido em duas etapas para facilitar sua elaboração. A primeira, compreende o entendimento de como funciona o mecanismo de inferência e a aplicação das técnicas de compiladores dentro do contexto de linguagens declarativas, bem como a execução de um protótipo interpretador de lógica proposicional. Mostra técnicas de implementação e montagem de suas partes mais relevantes, as estruturas da tabela de símbolos e as listas encadeadas montadas para o processo de inferência. Em sua segunda parte a proposta é, a partir deste protótipo, incluir novas técnicas e mecanismos para resolver um outro subconjunto da lógica de predicados, que é comumente conhecido como Datalog [MAIER].

2 Revisão Bibliográfica

Dentro do conteúdo da revisão bibliográfica é abordado o conceitos de resolução. Para tal apresentam as Cláusulas de Horn, a Refutação e as regras de equivalência, que culminam nas especificação das regras proposicionais. As quais serão tratadas como a notação sintática e semântica utilizadas na implementação do software NewProplog.

2.1 Método de Resolução

O método que o Proplog utiliza para resolver as regras e fatos, e retornar resultados para o usuário, é baseado em resolução. Para o entendimento de tal método é preciso entender outros conceitos.

Cláusulas de Horn são *fbfs*³ compostas de proposições ou de negação de proposições conectadas por disjunções, de tal forma que no máximo uma proposição não esteja negada, [GERSTING], como visto na Figura 01.

Utilizando as Regras de Equivalência da lógica proposicional, pode-se utilizar as Leis de De Morgan e a regra Condicional. Ao final ter-se-á a representação de uma regra da lógica proposicional que possui sua equivalente para a abordagem Prolog [GERSTING], apresentada na Figura 01, sendo utilizada no NewProplog como modelo de representação do conhecimento.

$\neg p \vee \neg q \vee r$	Cláusula de Horn
$\neg[p \wedge q] \vee r$	Leis de De Morgan
$p \wedge q \rightarrow r$ ⁴	Condicional

Figura 01. Notações de equivalência das Cláusulas de Horn

Seguindo os conceitos, o ponto focal da programação em lógica consiste em identificar a noção de *computação* com a noção de *dedução*. Mais precisamente, os sistemas de programação em lógica reduzem a execução de programas à pesquisa da refutação de sentenças do programa em conjunto com a negação da sentença que expressa a consulta, seguindo a regra: "*uma refutação é a dedução de uma contradição*" [PALAZZO]. Como citado por [NEPOMUCENO], refutação é tentarmos primeiramente supor que a fórmula sendo provada não seja válida (contradição), e então cair em um absurdo. O absurdo é indicado por um símbolo proposicional como dois valores lógicos diferentes em uma mesma interpretação. Por Exemplo:

³ Fbf – uma cadeia que forma uma expressão válida e é denominada fórmula-bem-formulada. Gersting (2001)

⁴ Em prolog, a fórmula $p \wedge q \rightarrow r$ é representada como $r :- p, q$.

“Sejam A e B proposições e $P: A \rightarrow (B \rightarrow A)$. Para provarmos que P é válida, basta demonstrarmos que P não é válida. Para isto, devemos encontrar uma interpretação onde P seja falsa (um contra-exemplo), isto é, $I(P) = F$ (nossa hipótese). Para o caso da condicional, sua interpretação é falsa se, e somente se, o seu antecedente é verdadeiro e o seu conseqüente é falso, ou seja, $I(A) = V$ e $I(B \rightarrow A) = F$. Neste caso, para termos $I(B \rightarrow A) = F$, devemos ter $I(B) = V$ e $I(A) = F$, o que é um absurdo, pois $I(A)$ já foi determinado como sendo verdadeiro. Logo, podemos concluir que P é válida, pois é impossível encontrarmos uma interpretação onde P seja falsa e, portanto, toda interpretação para P é verdadeira. A tabela abaixo ilustra todos os passos descritos anteriormente.”, como na Figura 02.

$A \rightarrow (B \rightarrow A)$			
	F		Hipótese $I(P) = F$
V		F	Logo $I(A) = V$ e $I(B \rightarrow A) = F$
	V	F	Portanto $I(B) = V$ e $I(A) = F$
V		F	Absurdo $I(A) = V$ e $I(A) = F$

Figura 02. Prova por Refutação ou redução ao absurdo – [NEPOMUCENO]

O conhecimento em NewProplog é expresso por meio de cláusulas de dois tipos: fatos e regras. Um fato denota uma verdade incondicional, enquanto que as regras definem as condições que devem ser satisfeitas para que uma certa declaração seja considerada verdadeira [PALAZZO], como mostrado na figura 03.

a :- b,c.	Regra
b.	Fato
c.	Fato

Figura 03. Exemplos de Fatos e Regras NewProplog

Na figura 03, a regra “a” é satisfeita se “b” e “c” também são satisfeitas. Como “b” e “c” são fatos, a regra “a” pode ser satisfeita. Ou seja “a” pode ser inferido a partir destas regras.

É importante também salientar a nomenclatura da cláusula. No exemplo 03, a regra “a:-b,c”; o elemento “a” é o cabeça da cláusula e os elementos “b” e “c” são denominados corpo da cláusula.

Como fatos e regras podem ser utilizados conjuntamente, nenhum componente dedutivo adicional precisa ser utilizado. Além disso, como regras recursivas e o não-determinismo são permitidos, os programadores podem obter descrições muito claras, concisas e não-redundantes da informação que desejam representar. Não havendo distinção entre argumentos de entrada e de saída, qualquer combinação de argumentos pode ser empregada [PALAZZO].

Outro ponto importante a ser destacado é a utilização da regra de inferência Modus Ponens, que utilizando a notação sintática do Prolog, induz: seja **b** um fato e a regra **a:-b.**, inferida a regra obtém-se **a**.

Portanto, com estes conceitos que são utilizados na resolução de regras Prolog, pode-se induzir a resolução no NewProplog.

3 Materiais e Métodos

Para a descrição dos materiais e métodos deve-se descrever a linguagem utilizada para a implementação do software e, tratando-se da abordagem de um interpretador, a gramática na BNF⁵ da linguagem a ser gerada, bem como as outras ferramentas utilizadas e a implementação propriamente dita.

3.1 A Linguagem para Implementação

O Delphi 5.0, foi a linguagem escolhida, por se tratar de uma linguagem orientada a objetos e possuir interface visual. Também tendo como vantagem sua estrutura e linguagem fonte, o Object Pascal, bem conhecida no meio acadêmico. Isto facilitaria a implementação do compilador.

⁵ Bakus-Naur Form (BNF) é uma metalinguagem, uma linguagem utilizada para descrever outra linguagem. Cada sentença desta metalinguagem é chamada de regra de produção. [Mak]

3.2 A gramática da Linguagem a ser Gerada

O segundo ponto a ser descrito é a BNF da linguagem a ser gerada, que determina a sintaxe de tal linguagem. Esta deve ser simples e permitir a estruturação do programa. Portanto, a gramática BNF do NewProplog [MAIER], foi assim definida:

```
<pmg>:: <clause><pmg>
      | λ
<clause> :: <literal><tail>.
<tail>:: :- <proplist>
      | λ
<proplist>::<literal><proptail>
<proptail>:: ,<proplist>
      | λ
<literal>:: <ll>(<ll>|<lu>|<digit>)*
<ll>:: [a-z]
<lu> :: [A-Z]
<digit>::[0-9]
```

3.3 Outras Ferramentas

O próximo passo é comentar as outras ferramentas utilizadas na construção do NewProplog. Para implementar o editor, de forma a facilitar o entendimento da linguagem, foi utilizado um pacote de ferramentas denominado SynEdit [SYNEDIT], permitindo criar uma interface de programação, na qual o programa seria diferenciado pela atribuição de cores às diferentes estruturas do código, reforçando assim a notação sintática para o desenvolvedor e facilitando o entendimento para iniciantes. Esta ferramenta é freeware e está disponível na internet para desenvolvedores Delphi.

Outra ferramenta utilizada é o Glyd 2 [GLYD2], um software com interface visual, desenvolvido para unir os geradores de analisadores léxicos(Lex) e sintáticos (Yacc), com editor de texto próprio e sintaxe definida para o compilador Object Pascal. Este Permite especificar uma gramática para geração de uma determinada linguagem, e obter o código fonte dos analisadores léxico e sintático da linguagem a ser gerada. Esta ferramenta é de

suma importância para o desenvolvimento do NewProplog, tendo em vista que pode haver, durante o projeto, a necessidade de se modificar a estrutura dos analisadores.

3.4 O Protótipo

Após ter sido abordado os materiais utilizados, é introduzido a o protótipo. Onde será apresentado a interface, as estruturas de dados, bem como o funcionamento do NewProplog utilizando os conceitos de resolução.

3.4.1 A Interface

Para iniciar a descrição do protótipo do software é importante destacar a interface, na qual se encontra um editor de código e um executor, como mostrado na figura 03. Uma interface simples, porém eficiente, pois com o auxílio das ferramentas disponibilizadas no SynEdit, pode ser implementado a edição, verificando a sintaxe do programa, mesmo antes da compilação.

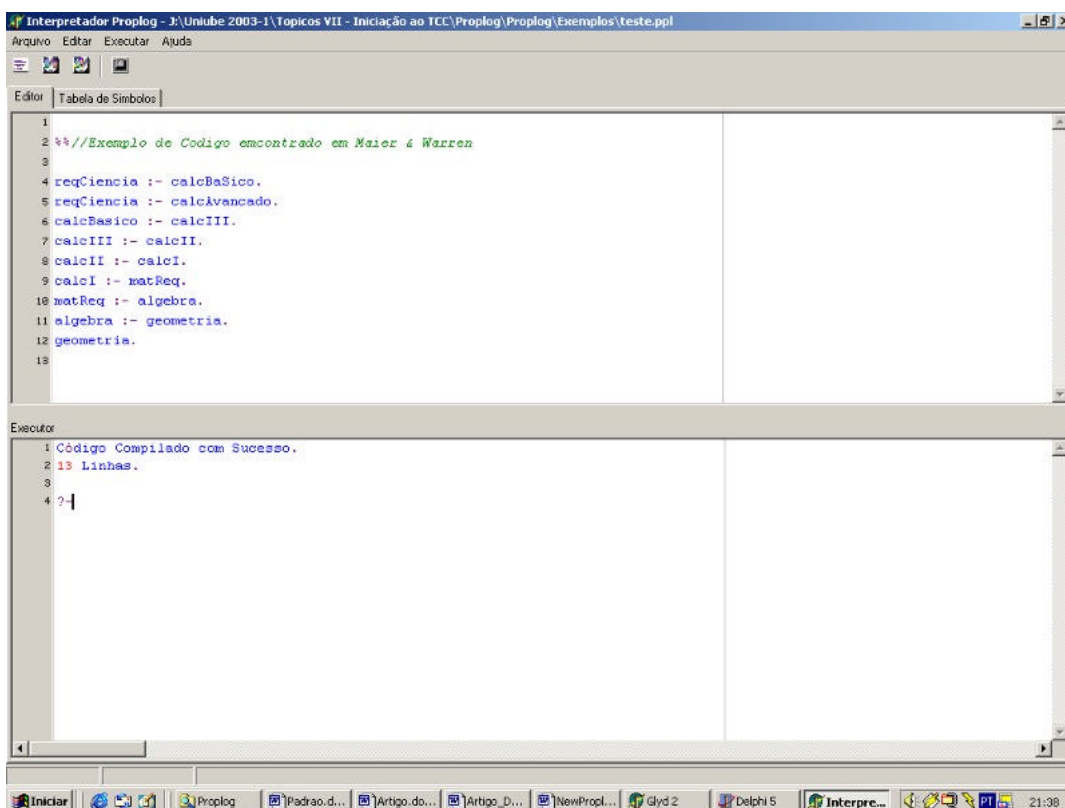


Figura 03. Interface do NewProplog

3.4.2 As Estruturas de Dados

As estruturas geradas durante o processo de compilação são, indiscutivelmente, a parte mais importante da implementação do NewProlog, pois são elas que permitem a busca e conseqüentemente a inferência às regras durante o processo de interpretação.

Estas estruturas são formadas durante a análise sintática. Porém, vale ressaltar a importância dos tipos (Type) criados para poder montar esta estrutura.

```
type
    PtrClause = ^Clause;
    Clause = Record
        IndTS: Integer;
        ProxLista: PtrClause;
        Case TPClause: Integer of
            1: (NextClause: PtrClause;);
            2: ();
    end;
```

Figura 04. Definição do tipo das Listas.

Este type permite a criação das listas encadeadas. Nele são criados ponteiros para a ligação dos termos da lista que contém um registro com os campos *IndTS*, o qual irá armazenar o índice da tabela de símbolos do elemento a ser inserido na lista; o campo *ProxLista* armazena o ponteiro do próximo elemento da lista; e por fim o campo *NextClause*, armazena o ponteiro de uma outra lista, caso uma regra NewProplog possuir alternativas. Deve-se ter uma atenção para este último campo, pois ele só existe na cabeça das cláusulas (regras), por isso a distinção no type criado referenciando 1, cabeça da cláusula, e 2, corpo da cláusula. Este type permite assim a elaboração de funções para criação e busca de termos da lista.

Outro tipo citado é o que armazena os tokens⁶ na tabela de símbolos, que permite gravar todos os dados de um token nesta tabela em uma mesma posição do vetor, tendo em vista que o armazenamento desta é feito num vetor. Pode-se ainda observar uma particularidade, o campo *Ponteiro*. Nele está contido o ponteiro que indica a lista criada a partir das regras, pois é sempre a partir deste campo que se inicia a inferência de uma regra.


```

type
  TToken = Record
    Lexema : string;
    Classe : string;
    Categoria:string;
    Ponteiro:PtrClause;
end;

```

Figura 5. Definição do tipo das Tokens

Após a apresentação dos tipos, expõe-se como estas estruturas são montadas e as funções que são encarregadas desta montagem, baseados no seguinte código de exemplo:

a :- b, c.

b.

c.

A primeira função a ser apresentada é a *instalartoken* que tem como parâmetro de entrada o lexema⁷ e retorna o índice do token na tabela de símbolos. Por Exemplo, supondo que o símbolo “a” seja o lexema , então, apos se chamar a função instalatoken, a tabela de símbolos ficará como mostrado na figura 6.

⁶ Token – nome utilizado para referenciar cada elemento, símbolo ou palavra dentro de um programa .

Segundo [Aho] token representa uma cadeia de caracteres logicamente coesiva.

⁷ Denominação utilizada para nomear qualquer palavra ou símbolo que seja devolvido pelo analisador léxico


Índice	Lexema	Ponteiro
1	a	
⋮	⋮	⋮
1024		

Figura 6. Instalação do lexema **a** na Tabela de Símbolos

Nesta função deve ser observado também a introdução da função hashing, para agilizar o processo de instalação e busca de tokens na tabela de símbolos (Vide Anexo A).

A seguir, a função *CriaLista* cria um elemento da lista, a qual depende do argumento passado, 1 se o elemento da cláusula é uma cabeça, e 2, se o elemento faz parte do corpo da cláusula como demonstrado no algoritmo da figura 07 (vide Anexo A). Esta função é executada sempre que o analisador sintático reconhece uma proposição.

```

função crialista(indice, tipodaclausula)
  início
    se tipodaclausula = 1 então
      criacabeça
    senão
      criacorpo
  fim

```

Figura 07. Função Cria Lista.

Esta função, possibilita então, a criação da estrutura de dados necessária para o processo de resolução das regras, como apresentada na figura 08.

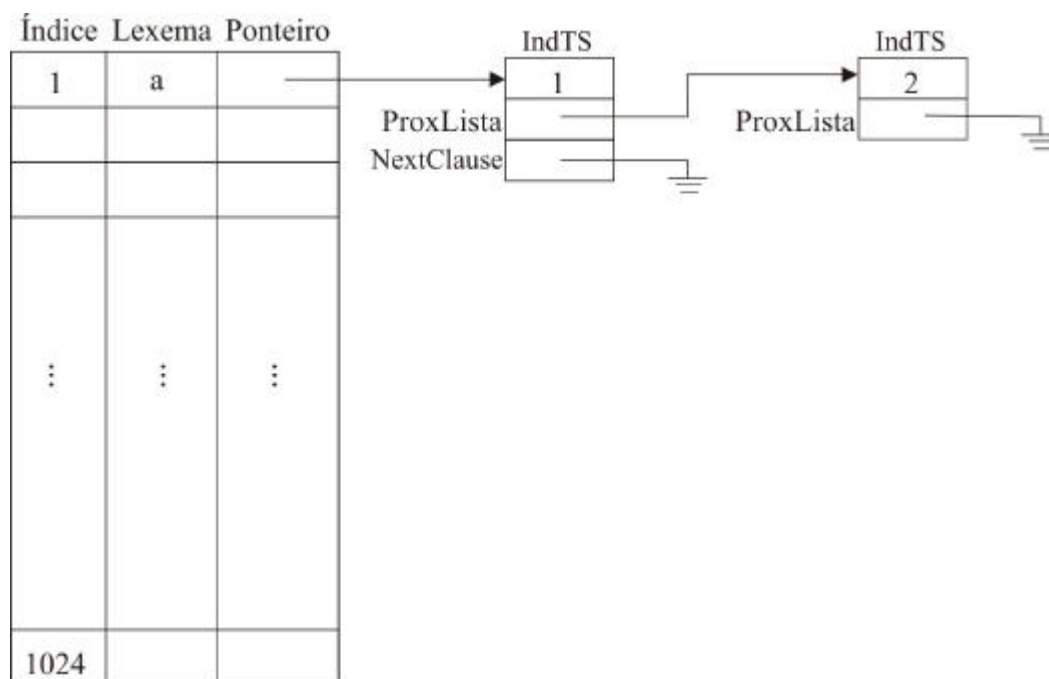


Figura 8. Estrutura montada a partir da função CriaLista.

Com o uso desta função, é possível, durante a análise léxica e sintática, construir a estrutura mostrada na figura 9.

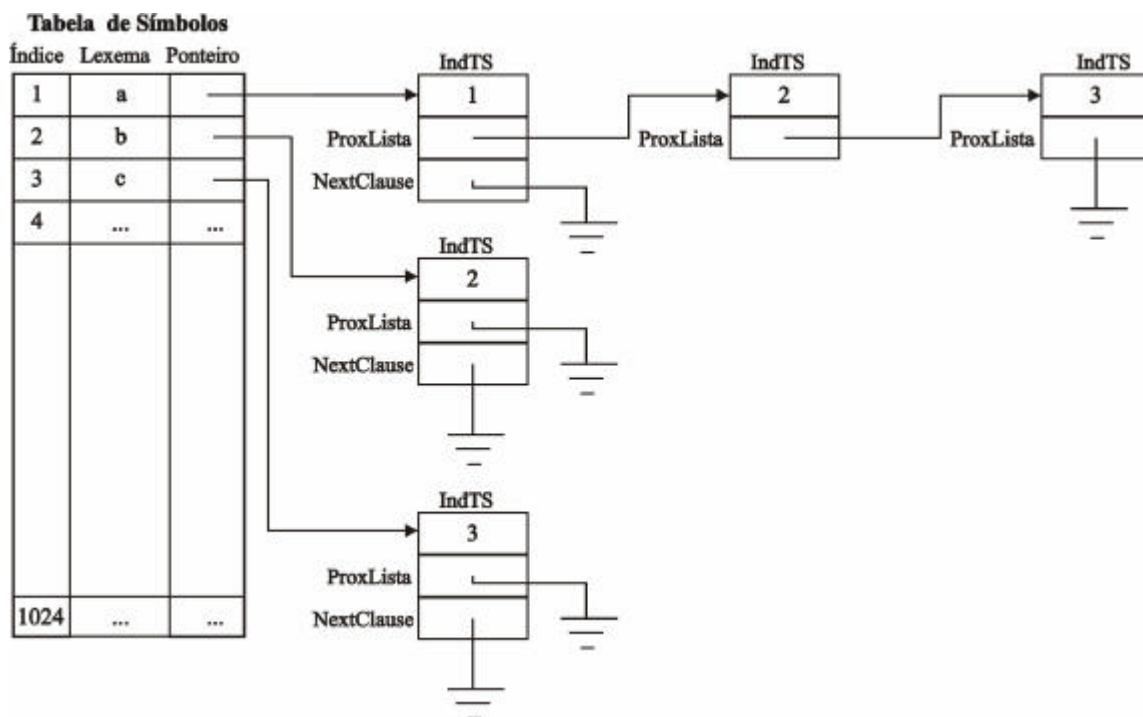


Figura 9. Tabela de Símbolos e Listas depois da Compilação.

Com base na estrutura montada, será apresentado a função *estabelecerinferencia*. Tal função gera realmente a inferência. O algoritmo ilustrado na figura 10 representa esta função. Para melhor explicar esta inferência, será utilizado o código exemplo utilizado anteriormente.

```

função estabelecerinferencia():boolean
  Início
    se pilhadeobjetivos = vazio então
      retorna verdadeiro
      sair da função
    senão
      pilhadeobjetivos=proximo
      estabelecerinferencia
  fim

```

Figura 10. Algoritmo da Função estabelecer inferencia.

A função *estabelecerinferencia* tem como referência uma pilha que é utilizada para guardar a lista de objetivos. Se a pilha de objetivos estiver vazia, ela retorna TRUE,

caso contrario ela começa a buscar os elementos das listas, tentando resolver as cláusulas, como mostrado na figura 11.

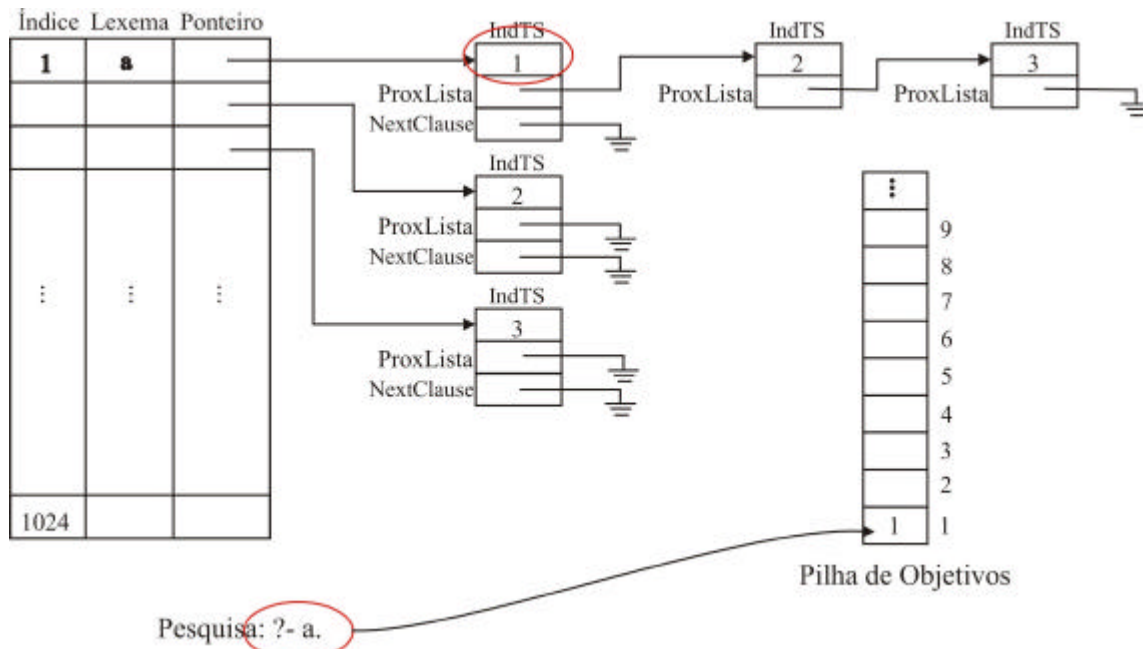
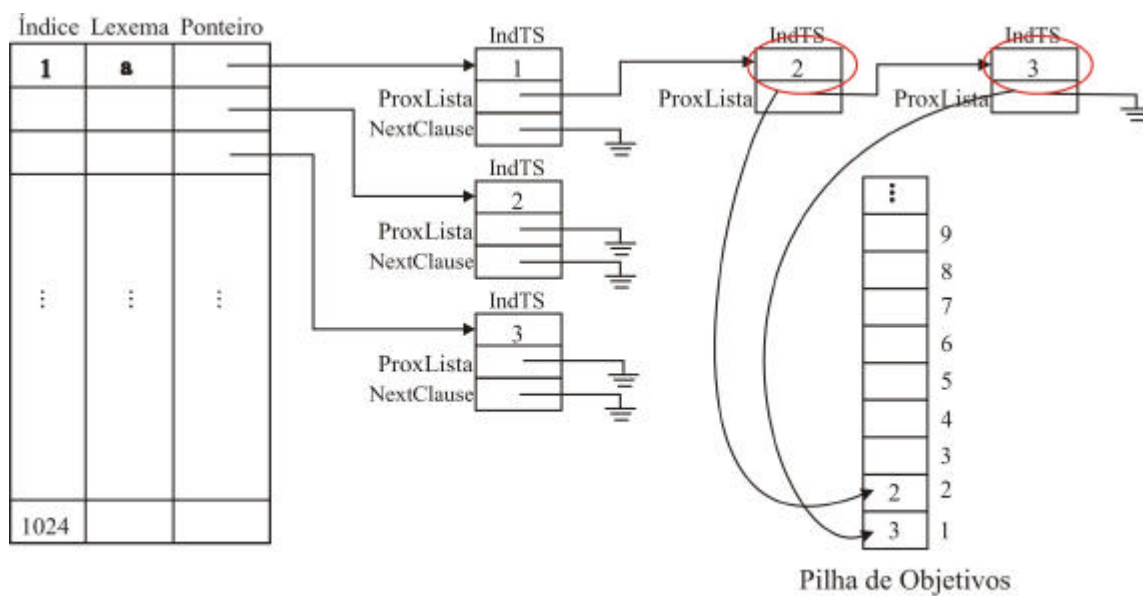


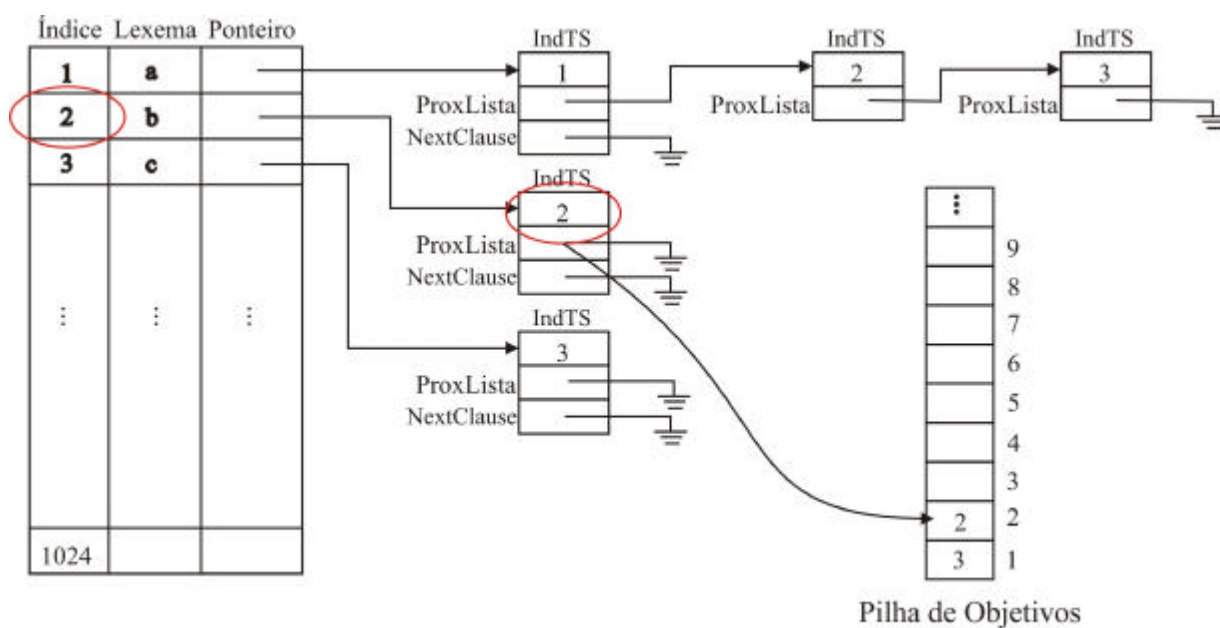
Figura 11. Parâmetro da pesquisa passado para a Pilha de Objetivos

Para tal ela utiliza uma copia de cada elemento da lista inserindo-o na pilha de objetivos, se o próximo elemento da pilha de objetivos for o elemento 1, como na figura 11, que leva ao lexema *a*, ela irá copiar o corpo da cláusula substituindo o elemento 1 da pilha de objetivos por 3 empilhando ainda o elemento 2, como indicado na figura 12. Como a pilha de objetivos não está vazia, será chamado novamente a função estabelecerinferencia novamente e tentará resolver o elemento 2, como mostrado na figura 13.



Pesquisa: ?- a.

Figura 12. Cópia do corpo da lista para a Pilha de Objetivos.



Pesquisa: ?- a.

Figura 13. Inferência do Próximo símbolo da pilha de objetivos-“b”.

Logo “2” é reconhecido como um fato, pois indica o símbolo “b” então a pilha de objetivos é decrementada e restará o elemento “3” ilustrado na figura 14. Seguindo a inferência sobre o elemento “3” que será reconhecido como fato, a pilha de objetivos será decrementada novamente e estará vazia. Quando a pilha de objetivos estiver vazia e a estrutura de listas requisitadas estiver sido totalmente percorrida, ela retornará TRUE, caso contrario retornará FALSE. No caso de TRUE, o interpretador retornará a mensagem “Sim”, pois o elemento “a” pode ser inferido a partir da base de regras mencionada, como demosntrado na figura 15.

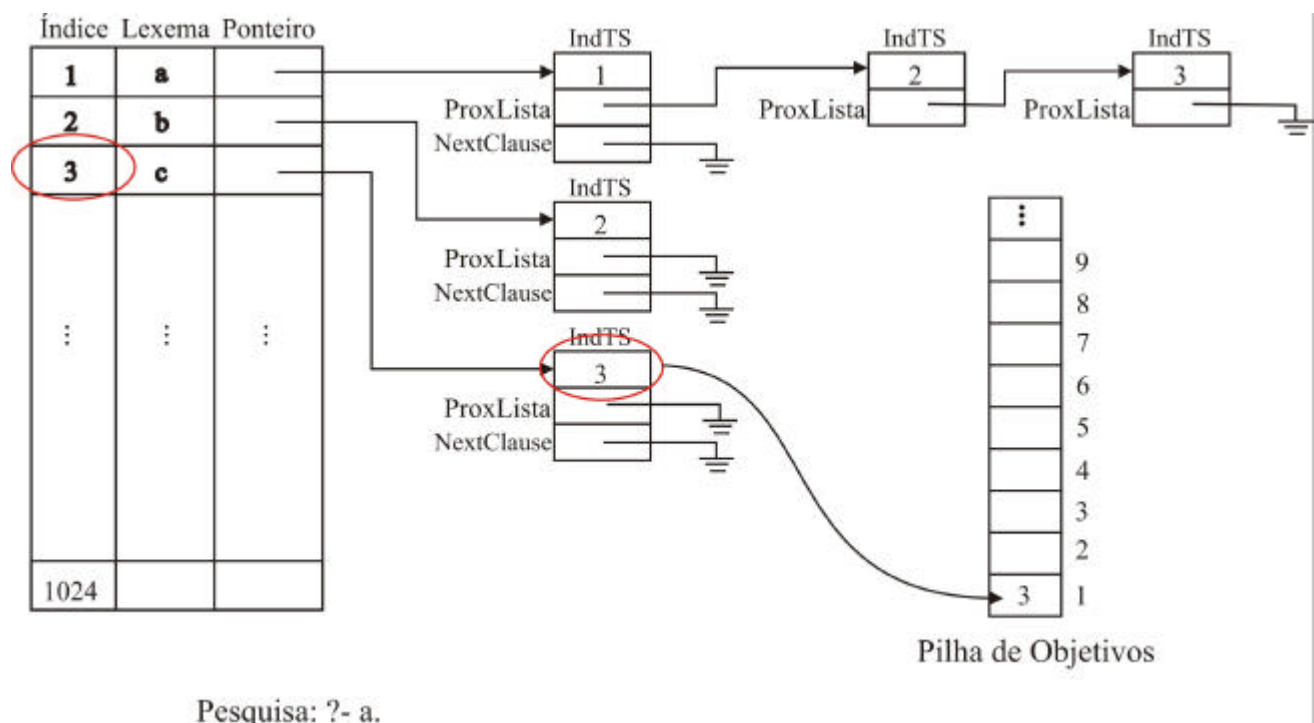


Figura 14. Inferência do Próximo símbolo da pilha de objetivos- “c”.

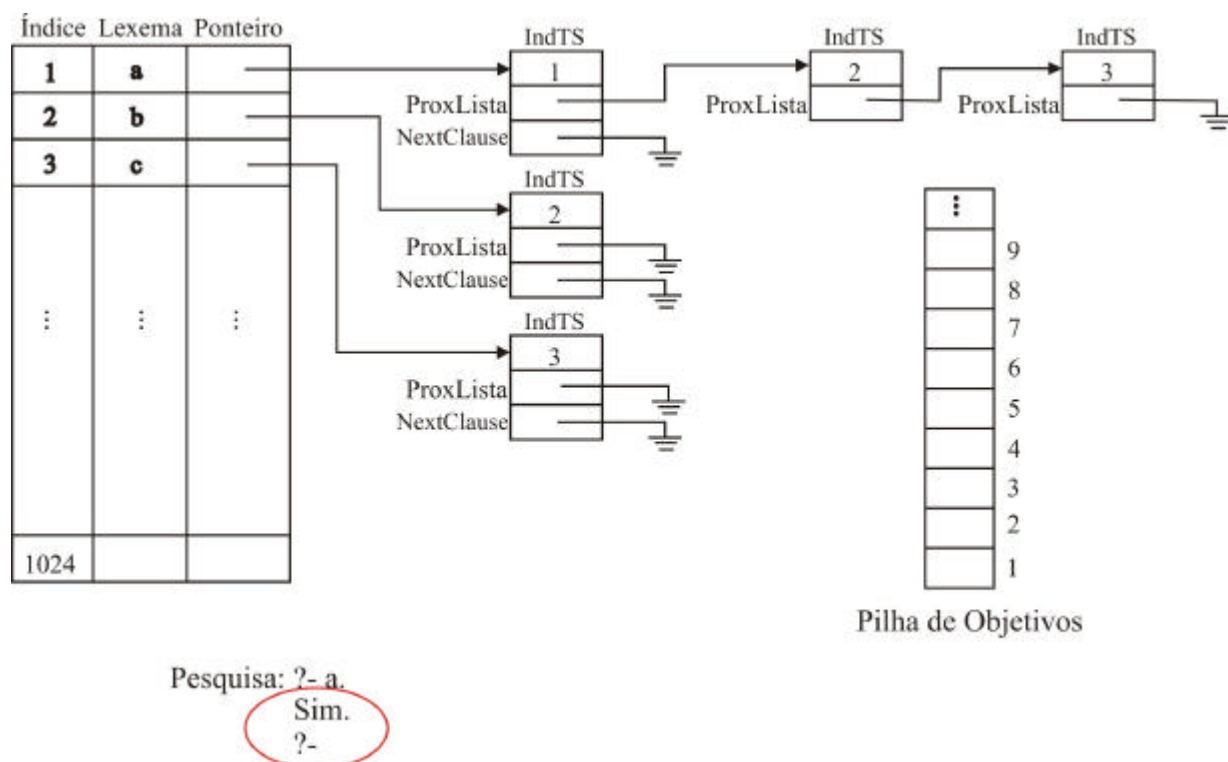


Figura 15. Termino da Interpretação e retorno da mensagem.

Com estas estruturas e funções corretamente localizada dentro do interpretador, cria-se processos suficientes para interpretar regras NewProplog .

3.4.3 O Interpretador

Para a construção do interpretador, dentro do contexto dos analisadores léxico e sintático, forma escolhidas as ferramentas Lex e Yacc, para a geração de tais analisadores respectivamente. Esta abordagem se mostra eficiente, como dito anteriormente, quando existe a necessidade de modificar a estrutura léxica e sintática durante o projeto, o que se faz constante na construção do NewProplog. Utilizando-se desta ferramenta, surge a necessidade de se definir a gramática geradora da linguagem dentro das especificações do Yacc, bem como a formulação das expressões regulares da linguagem gerada para a construção do analisador léxico. Estas especificações podem ser acompanhadas de funções e procedimentos auxiliares necessários para a construção da linguagem, como feito no NewProplog (vide Anexo A). O projeto foi totalmente baseado nesta abordagem utilizando-se a ferramenta Glyd 2, como visto anteriormente (Item 3.3) .

4 Conclusões

A produção deste projeto de software, assim como o artigo, propiciaram ao autor uma nova visão de linguagens lógicas, bem como da construção de compiladores. Também sendo observado os paradigmas de linguagens de programação, e a introdução de conceitos sobre novas ferramentas para construção de softwares.

O trabalho também vem reforçar idéia adquiridas durante disciplinas do curso, tais como: Matemática Discreta, Linguagens Formais e Autômatos, Compiladores, Estrutura de Dados dentre outras.

Agora é esperado que no próximo semestre, tenha-se a possibilidade de trabalhar a segunda parte da abordagem do Prolog, o DataLog, pretendendo introduzi-lo como Trabalho de Conclusão de Curso.

5 Referências

[MAIER] – MAIER, Daivid, et al ; Computing with logic. Melo Park, Californian : The Benjamin/Cummings Publishing Company, Inc, 1988.

[GERSTING] – GERSTING, Judith L.; Fundamentos Matemáticos para a Ciência da Computação. Rio de Janeiro: LTC – Livros Técnicos e Científicos Editora S.A., 2001.

[AHO] - AHO, Alfred V., et al.; Compiladores: Princípios, Técnicas e Ferramentas. Rio de Janeiro: Editora Guanabara Koogan S.A., 1995.

[PALAZZO] – PALAZZO, Luiz A. M.; Introdução a Programação Prolog Pelota – RS : EDUCAT – Editora da Universidade Católica de Pelotas, 1997.

[NEPOMUCENO] – NEPOMUCENO, Rogério Melo; Lógica Formal. Universidade de Uberaba. Disponível em:
<http://www.uniube.br/uniube/cursos/graduacao/tpd/Disciplinas/rogerio/MatDis/index.htm>

[MAK] - MAK, Ronald; Writing Compilers and Interpreters. Estados Unidos da América: Editora: Wiley Computer Publishing, 1996.

[PRICE] - PRICE, Ana Maria Alencar, et al.; Implementação de Linguagens de Programação. Editora Sagra-Luzzatto, 2000.

[GRUNE] – GRUNE, Dick, et al.; Projeto Moderno de Compiladores – Implementação e Aplicações. Rio de Janeiro : Editora Campus, 2001.

6 Referencias de Software

[SYNEDIT] **SynEdit**
<http://synedit.sourceforge.net/>

[GLYD2] **Glyd 2.0**
http://www.musikwissenschaft.uni-mainz.de/~ag/tply/Glyd_2_0.zip

[LEXYACC] **Lex-Yacc**
<http://www.musikwissenschaft.uni-mainz.de/~ag/tply/tply41a.zip>

Anexo A

Códigos da Implementação

Códigos elaborados para o Lex (Analisador Léxico)

```
(* Analizador lexico do Interpretador Proplog *)

LU  [A-Z]
LL  [a-z]
LETRA  [A-Za-z]
ALFA  [A-Za-z0-9]
DIG  [0-9]
SE  [:][-]
E  [,]
CMT  [%]
BRANCO  [ \t\n\r]
%%

var result : integer;
(* Reconhecimento de literais simples *)

{E} begin
    return(VIR)
end;

{LL}{ALFA}* begin
    yyval.yyInteger := TabelaSim.InstalarToken(yytext);
    return(LITERAL)
end;

{SE} begin
    return(OPERADORSE)
end;

"." begin
    return(OPERADORFIM)
end;

{DIG}+(\.{DIG}+)? begin
    return(NUM)
end;

{CMT}(.)* ;

{BRANCO}* ;
```

Códigos elaborados para o Lex (Analisador Léxico)

```
// Analisador Síntatico e Funções das Ações Semânticas
```

```
%{
```

```
unit
```

```
    unitSintProplog;
```

```
interface
```

```
uses
```

```
    SysUtils, YaccLib, LexLib, Dialogs;
```

```
Type
```

```
    TCompilador = Class(TObject)
```

```
    private
```

```
        // daqui a pouco
```

```
    public
```

```
        strCadeia : string;
```

```
        strSaida : string;
```

```
        intTamanho:integer;
```

```
        Procedure Compilador();
```

```
        Procedure Inicializar();
```

```
        Procedure memovazio();
```

```
    end;
```

```
type
```

```
    PtrClause = ^Clause;
```

```
    Clause = Record
```

```
        IndTS: Integer;
```

```
        ProxLista: PtrClause;
```

```
        Case TPClause:Integer of
```

```
            1: (NextClause: PtrClause;);
```

```
            2: ();
```

```
    end;
```

```
// Declaração do token */
```

```
type
```

```
    TToken = Record
```

```
        Lexema : string;
```

```
        Classe : string;
```

```
        Categoria:string;
```

```
        Ponteiro:PtrClause;
```

```
end;
```

```
// Declaração da tabela de simbolos */
```

```
type
  TSimbol = class
    public
      RegistroS : Array of TToken;
      Indice: Integer;
      strMostrarLista:string;
      GoalList: Array[1..1024] of integer;
      s:integer;
      Procedure InicializarTabSimb();
      function InstalarToken(Lexema:ShortString):Integer;
      function HashColisao(Lexema:String):Integer;
      // Function LocalizarToken(Lexema:String):integer
      function CriaLista(IndiceTabSim:Integer; TipoClause:Integer):PtrClause; //TipoClause
- 1)Cabeça - 2)Corpo
      function MostraLista(): String;
      function estabelecerinferencia():boolean;
End;
```

```
// Declaração da tabela de palavras reservada
```

```
type
  TTabReservadas = class
    private
      RegistroR : array of string;
      Tamanho : integer;
    public
      Procedure CriarTabelaReservadas ();
```

```
end;
```

```
var
  TabelaSim : TSimbol;
  parse : Tparser;
  tabres :TTabReservadas;
  TamanhoMemo:integer;
  TPClause:Integer;
  Tmp: PtrClause;
  memoSaida:String;
```

```

% }

%token <Integer> LITERAL /* Variaveis*/
%token NUM /* Iden NUM*/
%token OPERADORSE /* Implicação*/
%token OPERADORFIM /* Determina o fim da regra*/
%token VIR /* Determina o conjunção da regra*/

%type pmg clause vazio compilado
%type <PtrClause> proplist
%type <PtrClause> tail /* Não Terminais */
%type <PtrClause> proptail

%start linhas

%%

linhas : pmg {if (yyerrflag = 0) then begin yyaccept; memoSaida := 'Compilação efetuada
com sucesso!!!'; end;}
;

pmg : clause pmg {}
    | vazio
    ;

clause : LITERAL {TabelaSim.RegistroS[$1].Ponteiro := TabelaSim.CriaLista($1, 1);}
tail {TabelaSim.RegistroS[$1].Ponteiro^.ProxLista := $3;} OPERADORFIM
;
tail : OPERADORSE proplist {$$ := $2;}
    | vazio {$$ := nil;}
    ;

proplist
    : LITERAL proptail {New(Tmp); Tmp^.IndTS := $1; Tmp^.ProxLista := $2; $$ :=
Tmp;}
    ;

proptail
    : VIR proplist {$$ := $2;}
    | vazio {$$ := nil;}
    ;

vazio :

```

```

;

%%
{$I lexproplog.pas}

procedure TParser.parse();
begin
    if yyparse=0 then { done };
end;

Procedure TSimbol.InicializarTabSimb();

var
    i:integer;
begin

    indice:=0;
    s:=0;
    tabres := TTabReservadas.Create;
    SetLength(RegistroS, 1024);
    tabres.CriarTabelaReservadas();

    For i := 1 To (tabres.Tamanho - 1) Do
    Begin
        tabelasim.RegistroS[i].Lexema := tabres.RegistroR[i];
        tabelasim.RegistroS[i].Classe := 'RESERVADA';
        tabelasim.RegistroS[i].Ponteiro:= nil;
    end;
end;

Procedure TTabReservadas.CriarTabelaReservadas ();
var i:integer;
Begin
    for i := 1 to 1023 do
        begin
            if (tabelasim.RegistroS[i].Lexema = "") then
                tabelasim.RegistroS[i].Lexema := 'nil';
        end;
    Tamanho := 12;
    setlength(RegistroR,Tamanho);
    //Valores dos registros.
    RegistroR [11] := 'assert';
    RegistroR [1] := 'retract';
    RegistroR [2] := 'write';
    RegistroR [3] := 'is';
    RegistroR [4] := 'fail';
    RegistroR [5] := 'help';

```



```

    RegistroR [6] := 'get';
    RegistroR [7] := 'put';
    RegistroR [8] := 'atom';
    RegistroR [9] := 'asserta';
    RegistroR [10] := 'assertz';

End;

Procedure Tcompilador.Inicializar();
    var
        i:integer;
begin
    TamanhoMemo := intTamanho;
    TabelaSim := TSimbol.Create;
    TabelaSim.InicializarTabSimb();
end;

Procedure Tcompilador.Compilador();

begin
    // yydebug := Debug.Down;
    strSaida := "";
    memoSaida := "";

    if (strCadeia = "") then
        memovazio;
        yyinput_text := strCadeia;
        yyinit;
        //
        parse := TParser.Create;
        parse.parse;
        parse.Free;
        //
        strSaida := yyoutput.GetText;
        strSaida := strSaida + memoSaida;

end;

Procedure TCompilador.memovazio();

begin
    if (strCadeia = "") then
        ShowMessage('Entre com as regras a serem interpretadas');
end;

```

```

function TSimbol.InstalarToken(Lexema:String):Integer;
var
    i:integer;
    j:integer;

begin
    Indice:=Indice + 1;

    For i := 1 To Length (Lexema) Do
        begin
            j := j + ord(Lexema[i]);
        end;
    j:= j div Length(Lexema);

    if ( TabelaSim.RegistroS[j].Lexema = 'nil') then
        begin
            TabelaSim.RegistroS[j].Lexema := Lexema;
            TabelaSim.RegistroS[j].Classe := 'ID';
            TabelaSim.RegistroS[j].Categoria := 'Literal';
            TabelaSim.RegistroS[j].Ponteiro := nil;
            result:=j;
        end
    else if ( TabelaSim.RegistroS[j].Lexema = Lexema )then
        begin
            result := j;
        end
    else
        begin
            // Chama rotina de colisão
            result:=HashColisao(Lexema);
        end;

end;

function TSimbol.HashColisao(Lexema:String):integer;
var
    i,j:integer;
begin
    j:=0;
    For i := 1 To Length (Lexema) Do
        begin
            j := j + ord(Lexema[i]);
        end;
    j:= j div Length(Lexema);
    while (TabelaSim.RegistroS[j].Lexema <> 'nil') do
        j:= j + 1;
    TabelaSim.RegistroS[j].Lexema := Lexema;

```

```

TabelaSim.RegistroS[j].Classe := 'ID';
TabelaSim.RegistroS[j].Categoria := 'Literal';

result:=j;
end;

function TSimbol.CriaLista(IndiceTabSim:Integer; TipoClause:integer):PtrClause;
var AUX: PtrClause;
    N : PtrClause;
    INT :integer;
begin

    TPClause:=TipoClause;
    if (TPClause = 1) then
        begin
            new(N);
            N^.IndTS:= IndiceTabSim;
            N^.NextClause:= nil;
            N^.ProxLista:= nil;

result:= N ;
            end
        else if (TPClause = 2)then
            begin
                new(N);
                AUX:= TabelaSim.RegistroS[IndiceTabSim].Ponteiro;
                while AUX^.ProxLista <> nil do
                    begin
                        AUX := (AUX^.ProxLista).ProxLista;
                    end;
                N^.IndTS := IndiceTabSim;
                N^.ProxLista := nil;
                AUX^.ProxLista := N;
                result:= N;
            end
        else
            begin
                result:= PtrClause(nil);
            end;
        end;
    end;

function TSimbol.MostraLista(): String;
var strAUX, strAUX2, strINDTS: string;
    AUX: PtrClause;
    i,j:integer;
begin
    strAUX:="";
    strAUX2:="";

```

```

For i := 12 To 1023 Do
begin
  if TabelaSim.RegistroS[i].Lexema <> 'nil' then
  begin
    if TabelaSim.RegistroS[i].Ponteiro <> nil then
    begin
      strINDTS := IntToStr(TabelaSim.RegistroS[i].Ponteiro^.IndTS);
      strAUX:= 'Índice na Tabela de Símbolos --> ' + strINDTS + ' Cabeça --> ' +
strINDTS + ' Corpo --> ';
      if TabelaSim.RegistroS[i].Ponteiro^.ProxLista <> nil then begin
        AUX:= TabelaSim.RegistroS[i].Ponteiro^.ProxLista;
        while AUX <> nil do
        begin
          strAUX:= strAUX + IntToStr(AUX^.IndTS) + ' --> ';
          AUX := AUX^.ProxLista;
        end;
        strAUX := strAUX + ' nil ' + #10 + #10;
        strAUX2:= strAUX2 + strAUX;
      end;
    end;
  end;

  result := strAUX2;
end;

function TSimbol.estabelecerinferencia():boolean;
var
  ProxClause:PtrClause;
  i,j, k, intaux:integer;
  aux : array[1..1024] of integer;
begin
  result := false;
  for k:=1 to 1024 do
    aux[k]:=0;
  ProxClause:=nil;

  if (goallist[s] = 9999) then
  begin
    result := true;
    exit;
  end;
  if TabelaSim.RegistroS[goallist[s]].Ponteiro <> nil then
  begin
    ProxClause := TabelaSim.RegistroS[goallist[s]].Ponteiro;
    i := 1;

```

```

while ProxClause^.ProxLista <> nil do
begin

    aux[i]:= ProxClause^.ProxLista.IndTS;
    i:= i + 1;
    ProxClause := ProxClause^.ProxLista;
end;
i:=i-1;
j := i;
for i := j downto 1 do
begin
    goallist[s]:= aux[i];
    s := s+1;
end;
s:=s-1;
if establecerinferencia() then
    result := true;
end
else
begin
    result := false;
    exit;
end;

end;

end.

```