

**Universidade Tecnológica Federal do Paraná
Departamento de informática – DAINF
Ciência da Computação**

**Jonathan Santos da Silva
Maria José Duarte**

Tradutor da Linguagem Natural NAG para a Linguagem Jason

**Ponta Grossa
2023**

Jonathan Santos da Silva e Maria José Duarte

Tradutor da Linguagem Natural NAG para a Linguagem Jason

Trabalho apresentado como requisito parcial à obtenção de nota do Curso de Ciência da Computação da Universidade Tecnológica Federal do Paraná, orientado por Gleifer Vaz Alves.

**Ponta Grossa
2023**

SUMÁRIO

1. INTRODUÇÃO.....	4
2. NOTAÇÃO BNF.....	5
3. ANÁLISE LÉXICA.....	6
4. ANÁLISE SINTÁTICA.....	7
5. FUNÇÕES EM C.....	8
5.1. Estrutura de dados.....	8
5.2. Funções.....	10
6. TESTES SIGNIFICATIVOS.....	11
6.1. Lista de agentes linguagem NAG.....	11
6.1.1. Código gerado Bob.....	13
6.1.2. Código gerado Diego.....	13
6.1.3. Código gerado Eduardo.....	14
6.2. Testes JaCaMo.....	14
7. CONCLUSÃO.....	16
8. REFERÊNCIAS.....	17

1. INTRODUÇÃO

A construção de um Tradutor da Linguagem Natural NAG (Natural Language for Agents) para a Linguagem Jason/AgentSpeak representa um desafio significativo no campo da tecnologia da linguagem e da programação. Este relatório descreve o trabalho empreendido na criação dessa ferramenta de tradução, destacando o uso de técnicas de análise léxica e análise sintática para tornar essa tradução possível.

O processo de desenvolvimento do tradutor envolveu a utilização da ferramenta Flex para a análise léxica, que permite a identificação de tokens e segmentação dos elementos da linguagem natural NAG. Em seguida, a análise sintática foi implementada com o auxílio da ferramenta Bison, que utiliza gramáticas da notação BNF para estruturar a tradução, sendo que ambas as etapas foram fundamentais para a tradução da Linguagem Natural NAG para a Linguagem Jason.

A gramática BNF desempenhou um papel central na estruturação da tradução, proporcionando uma base sólida para a compreensão da estrutura linguística. Além disso, a organização do código foi viabilizada por meio de arquivos em C e um arquivo header.h, que definem funções essenciais para a tradução e na organização do código do tradutor.

No presente relatório, exploraremos os detalhes do processo de construção e funcionamento do tradutor, abordando a integração das etapas de análise léxica e análise sintática, bem como a estrutura do código-fonte e o papel de cada arquivo envolvido.

2. NOTAÇÃO BNF

Figura 1: EBNF da linguagem NAG/AgentSpeak

```
<program> ::= agents

<agents> ::= ε
           | <agent> '%' <agents>

<agent> ::= '#' NAME BELIEFS ':' <beliefs> GOALS ':' <goals> PLANS ':' '{' <plans> '}'

<beliefs> ::= '{' <beliefsName> '}'

<beliefsName> ::= ε
               | NAME ';' <beliefsName>

<goals> ::= '{' <goalsName> '}'

<goalsName> ::= ε
             | NAME ';' <goalsName>

<plans> ::= ε
         | <plan> ';' <plans>

<plan> ::= ε
        | NAME <plansTuple>

<plansTuple> ::= '(' <triggerEvent> ';' <context> ';' <body> ')'

<triggerEvent> ::= NAME

<context> ::= ε
           | <logExp>
           | NAME

<logExp> ::= NAME AND NAME
          | NAME OR NAME
          | NOT NAME

<body> ::= '{' <bodysFormula> '}'

<bodysFormula> ::= ε
               | NAME ';' <bodysFormula>
```

fonte: autoria própria

A BNF é uma notação utilizada para definir a gramática de uma linguagem. No contexto apresentado, a BNF está sendo usada para definir a estrutura da linguagem natural NAG (*Natural Language for Agents*). Ela define as regras que determinam como os programas escritos nessa linguagem devem ser estruturados.

3. ANÁLISE LÉXICA

Figura 2: Estrutura de análise léxica - Flex

A screenshot of a code editor window with a dark background and a purple border. The editor displays Flex lexer code. At the top, there are three colored window control buttons (red, yellow, green). The code starts with a line: `%option noyywrap nodefault yylineno`. This is followed by a block `%{` containing two `#include` statements: `#include "header.h"` and `#include "analise-sintatica.tab.h"`. After the block, there is a line `%}`. Below this, there is a separator `%%`. The main part of the code is a list of tokens and their corresponding actions. Tokens are listed on the left, and actions are on the right, separated by a vertical bar. The tokens include: `":"`, `","`, `"{"`, `"}"`, `"("`, `"%"`, `"#"`, `")"`, `"E"`, `"OU"`, `"NAO"`, `"crencas"`, `"objetivos"`, `"planos"`, `[a-z][a-zA-Z0-9]*`, `[\t\r]+`, `[\n]`, `[\\n]`, and `.`. The actions for these tokens are: `{ return yytext[0]; }`, `{ return AND; }`, `{ return OR; }`, `{ return NOT; }`, `{ return BELIEFS; }`, `{ return GOALS; }`, `{ return PLANS; }`, `{ yylval.s = strdup(yytext); return NAME; }`, `{ }`, `{ }`, `{ }`, and `{ }`. The code ends with a separator `%%`.

fonte: autoria própria

O código em Flex implementa a análise léxica para a linguagem natural NAG (*Natural Language for Agents*). Ao quebrar o código-fonte em tokens ou símbolos significativos, o analisador léxico fornece uma representação mais gerenciável dos elementos individuais do programa, tornando mais fácil para o analisador sintático identificar e interpretar a estrutura hierárquica e a lógica subjacente.

No código define-se os tokens reconhecidos pela linguagem NAG e associa cada token a um valor específico. Essas palavras-chave são elementos essenciais da linguagem NAG. Qualquer caractere não reconhecido pela

especificação gerará um erro com a mensagem “Caractere não reconhecido”, indicando que o código fonte contém um caractere inválido.

Sendo assim, o código Flex desempenha um papel fundamental na análise léxica da linguagem NAG. Reconhece tokens importantes, como palavras-chave e identificadores, ao mesmo tempo que ignora caracteres de controle e sinaliza erros quando encontra caracteres inesperados. Esse é um passo crucial na construção de um tradutor para a linguagem NAG, pois permite que o código fonte seja dividido em unidades significativas para a análise sintática.

4. ANÁLISE SINTÁTICA

Para análise sintática, foi escolhido o uso do analisador sintático BISON. Seguindo o mesmo padrão da BNF implementamos uma gramática que segue a seguinte definição de variáveis e terminais:

$V = \{ \text{initial, agents, agent, beliefs, beliefsName, goals, goalsName, plans, plan, plansTuple, triggerEvent, context, logExp, body, bodysFormula} \}.$

$T = \{ \text{NAME, BELIEFS, GOALS, PLANS, OR, NOT, AND} \}.$

Com isso conseguimos seguir para definição criando todas as regras de produção, sendo representadas no bison, como demonstrado na figura 3.

Figura 3: Estrutura de análise léxica - Flex

```
%%  
  
initial: agents { printList($1); }  
;  
agents: { $$ = NULL; }  
| agent '%' agents { $$ = prependAgent($3, $1); }  
;  
agent: '#' NAME BELIEFS ':' beliefs GOALS ':' goals PLANS ':' '{' plans '}' { $$ = createAgent($2, $5, $8, $12); }  
;  
beliefs: '{' beliefsName '}' { $$ = $2; }  
;  
beliefsName: { $$ = NULL; }  
| NAME ';' beliefsName { $$ = prependBelieve($3, $1); }  
;  
goals: '{' goalsName '}' { $$ = $2; }  
;  
goalsName: { $$ = NULL; }  
| NAME ';' goalsName { $$ = prependGoal($3, $1); }  
;  
plans: { $$ = NULL; }  
| plan ';' plans { $$ = prependPlan($3, $1); }  
;  
plan: { $$ = NULL; }  
| NAME plansTuple { $$ = createPlan($1, $2); }  
;  
plansTuple: '(' triggerEvent ';' context ';' body ')' { $$ = createContent($2, $4, $6); }  
;  
triggerEvent: NAME { $$ = $1; }  
;  
context: { $$ = NULL; }  
| logExp { $$ = $1; }  
| NAME { $$ = $1; }  
;  
logExp: NAME AND NAME { $$ = newExp($1, $3, "E"); }  
| NAME OR NAME { $$ = newExp($1, $3, "OU"); }  
| NOT NAME { $$ = newExp(NULL, $2, "NAO"); }  
;  
body: '{' bodysFormula '}' { $$ = $2; }  
;  
bodysFormula: { $$ = NULL; } |  
NAME ';' bodysFormula { $$ = prependBody($3, $1); }  
;  
%%
```

fonte: autoria própria

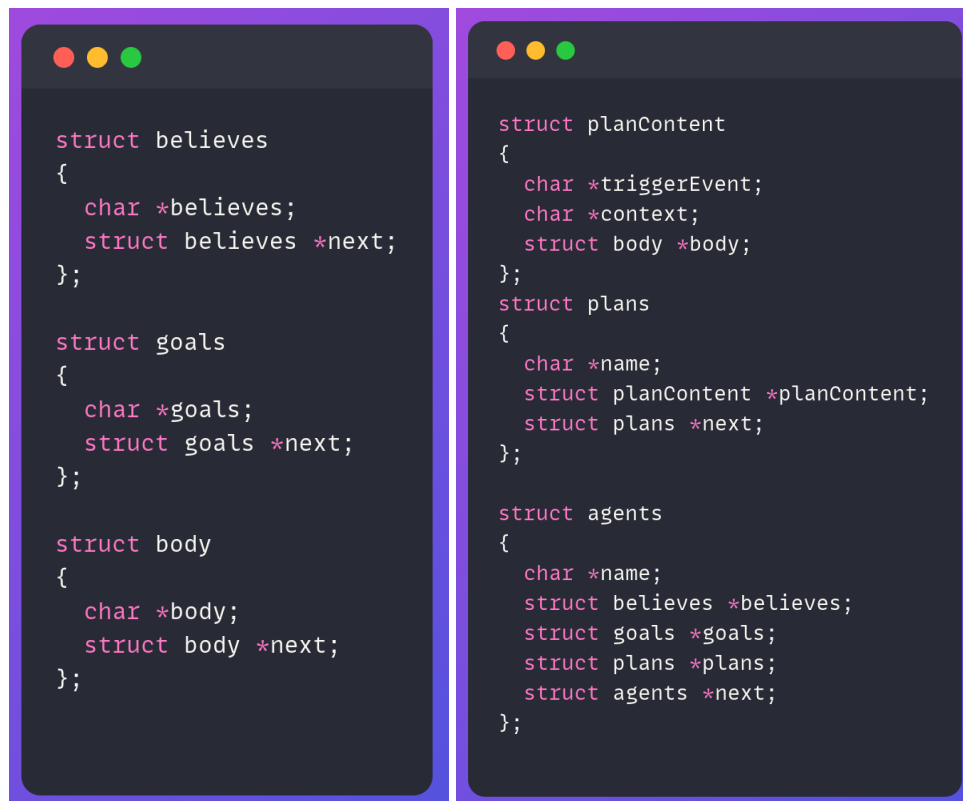
5. FUNÇÕES EM C

5.1. Estrutura de dados

Para estrutura de dados, foi utilizado o modelo de lista, por sua versatilidade e por ser um modelo de desenvolvimento mais ágil, liberando espaços para refinamentos.

A estrutura se define como a partir dos itens mais individuais, como crenças, objetivos e corpo, para os mais completos e que possuem dependências dos outros, nesse caso, a estrutura de agentes.

Figura 4: Estrutura de dados utilizada no projeto




fonte: autoria própria

5.2. Funções

As funções em C foram separadas por contextos, onde existe pelo menos uma função para cada estrutura de dados. Essas funções têm dois objetivos principais: Criar uma instância da estrutura de dados; Inserir essa instância no começo da lista encadeada da estrutura.

A inserção no início foi escolhida, por conta de que durante a análise sintática, o bison realiza uma análise ascendente, o que faz com que ele pegue os tokens de name em uma ordem invertida. Utilizando a inserção no início, esse problema já é resolvido.

Figura 5: Cabeçalho das funções em C



```
// believes
struct believes *prependBelieve(struct believes *believes, char *newBelieve);
// goals
struct goals *prependGoal(struct goals *goals, char *newGoal);

struct body *prependBody(struct body *body, char *newBody);
// plans
struct plans *createPlan(char *name, struct planContent *planContent);
struct plans *prependPlan(struct plans *plans, struct plans *newPlan);

struct planContent *createContent(char *triggerEvent, char *context, struct body *body);

// agents
struct agents *createAgent(char *name, struct believes *believes, struct goals *goals, struct plans *plans);
struct agents *prependAgent(struct agents *agents, struct agents *newAgent);

// print
void printAgent(struct agents *agents);
void printList(struct agents *list);
void printAgentInFile(struct agents *agent);

// exp
char *newExp(char *leftSide, char *rightSide, char *operator);
```

fonte: autoria própria

Figura 6: Exemplo de implementação

```
struct body *prependBody(struct body *body, char *newBody)
{
    if (newBody == NULL)
    {
        return body;
    }
    if (body == NULL)
    {
        struct body *new = (struct body *)malloc(sizeof(struct body));

        new->body = (char *)malloc(sizeof(char) * strlen(newBody) + 100);

        strcat(new->body, ".printf(\"");
        strcat(new->body, newBody);
        strcat(new->body, "\").");
        new->next = body;

        return new;
    }

    struct body *aux = body;

    struct body *new = (struct body *)malloc(sizeof(struct body));

    new->body = (char *)malloc(sizeof(char) * strlen(newBody) + 100);

    strcat(new->body, ".printf(\"");
    strcat(new->body, newBody);
    strcat(new->body, "\");");
    new->next = body;

    return new;
}
```

fonte: autoria própria

Essas implementações podem ser vistas no código fonte do repositório <https://github.com/jowjow22/nag-to-jason-translate> disponível publicamente no Github

6. TESTES SIGNIFICATIVOS

Foram gerados 3 agentes para representar os testes de transpilação, sendo eles Bob, Diego e Eduardo. Os mesmos foram testados dentro da plataforma JaCaMo para validar seu funcionamento de acordo com a sintaxe do JASON.

6.1. Lista de agentes linguagem NAG

Figura 7: Agentes bob, diego e eduardo

```
#bob crenças: { estaChovendo ; naotenhoGuardaChuva ; }
objetivos: { comprarGuardaChuva ; naoPegarChuva ; }
planos: { plano1 ( comprarGuardaChuva ; estaChovendo E naotenhoGuardaChuva ;
{ sair; procurarLoja; comprarGuardaChuva; } ) ;
plano2 ( naoPegarChuva ; NAO estaChovendo ; { sair; jogarBola ; } ) ;
plano3 ( naoPegarChuva ; estaChovendo E naotenhoGuardaChuva ;
{ ficarEmCasa ; estudar ; } ) ; }
%

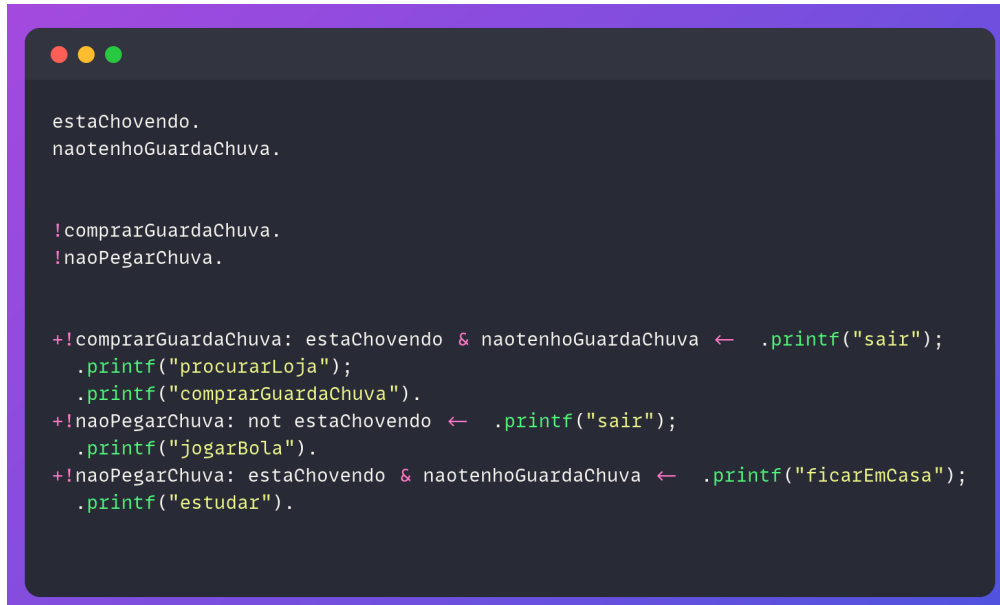
#diego
crenças: { estaSoleado ; tenhoOculosDeSol ; tenhoProtetorSolar ; }
objetivos: { aproveitarDia ; protegerPeleDosRaios ; }
planos: {
plano1 ( aproveitarDia ; estaSoleado E tenhoOculosDeSol ;
{ sair; praticarEsportesAoArLivre; aplicarProtetorSolar; } ) ;
plano2 ( protegerPeleDosRaios ; estaSoleado E tenhoOculosDeSol ;
{ ficarNaSombra; usarChapeu; aplicarProtetorSolar; } ) ;
}
%

#eduardo
crenças: { estaEnsolarado ; tenhoRedeDeVolei ; tenhoBola ; tenhoProtetorSolar ; }
objetivos: { jogarVolei ; aproveitarDia ; }
planos: {
plano1 ( jogarVolei ; tenhoRedeDeVolei E tenhoBola ;
{ sair; montarRede; pegarBola; jogarVolei; } ) ;
plano2 ( aproveitarDia ; estaEnsolarado E tenhoProtetorSolar ;
{ sair; aplicarProtetorSolar; fazerAtividadesAoArLivre; } ) ;
}
%
```

fonte: autoria própria

6.1.1. Código gerado Bob

Figura 8: Código Jason bob



```
estaChovendo.  
naotenhoGuardaChuva.  
  
!comprarGuardaChuva.  
!naoPegarChuva.  
  
+!comprarGuardaChuva: estaChovendo & naotenhoGuardaChuva ← .printf("sair");  
  .printf("procurarLoja");  
  .printf("comprarGuardaChuva").  
+!naoPegarChuva: not estaChovendo ← .printf("sair");  
  .printf("jogarBola").  
+!naoPegarChuva: estaChovendo & naotenhoGuardaChuva ← .printf("ficarEmCasa");  
  .printf("estudar").
```

fonte: autoria própria

6.1.2. Código gerado Diego

Figura 9: Código Jason Diego



```
estaSoleado.  
tenhoOculosDeSol.  
tenhoProtetorSolar.  
  
!aproveitarDia.  
!protegerPeleDosRaios.  
  
+!aproveitarDia: estaSoleado & tenhoOculosDeSol ← .printf("sair");  
  .printf("praticarEsportesAoArLivre");  
  .printf("aplicarProtetorSolar").  
+!protegerPeleDosRaios: estaSoleado & tenhoOculosDeSol ← .printf("ficarNaSombra");  
  .printf("usarChapeu");  
  .printf("aplicarProtetorSolar").
```

fonte: autoria própria

6.1.3. Código gerado Eduardo

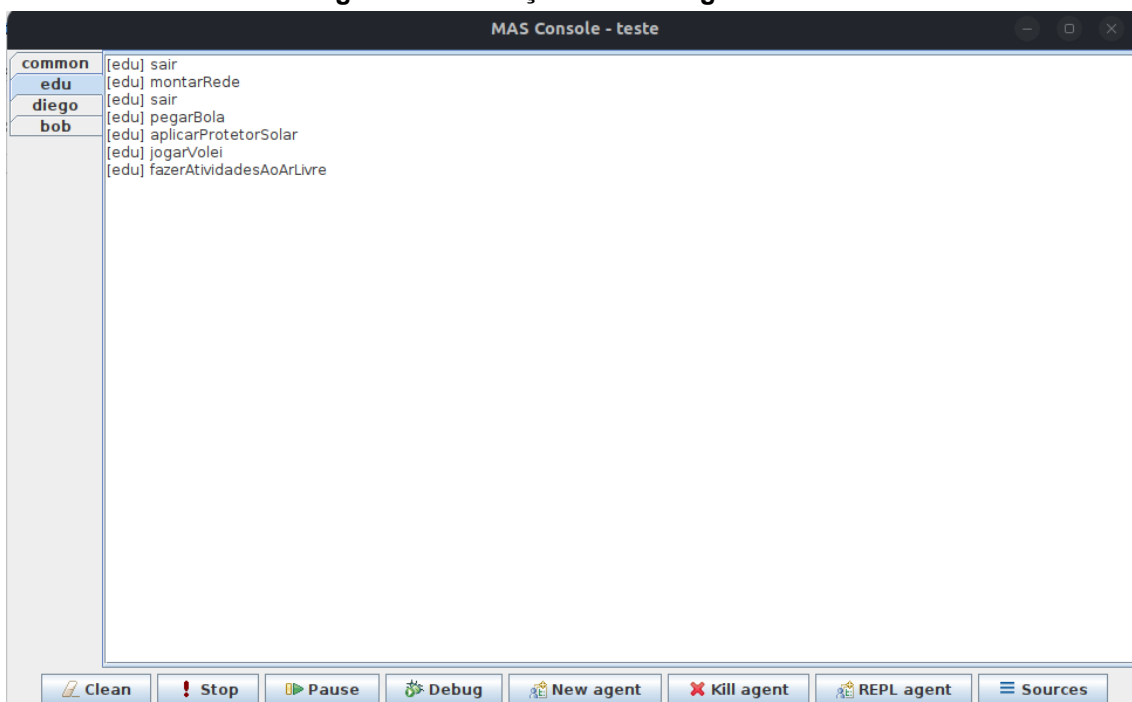
Figura 10: Código Jason Eduardo

```
estaEnsolarado.  
tenhoRedeDeVolei.  
tenhoBola.  
tenhoProtetorSolar.  
  
!jogarVolei.  
!aproveitarDia.  
  
+!jogarVolei: tenhoRedeDeVolei & tenhoBola ← .printf("sair");  
    .printf("montarRede");  
    .printf("pegarBola");  
    .printf("jogarVolei").  
+!aproveitarDia: estaEnsolarado & tenhoProtetorSolar ← .printf("sair");  
    .printf("aplicarProtetorSolar");  
    .printf("fazerAtividadesAoArLivre").
```

fonte: autoria própria

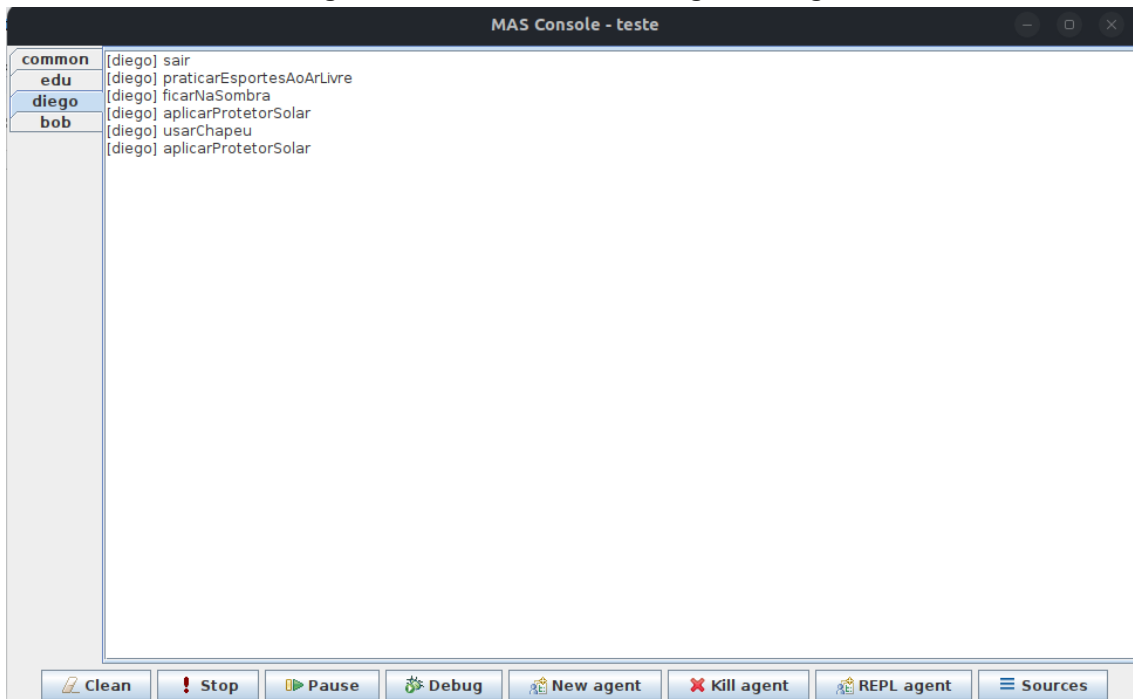
6.2. Testes JaCaMo

Figura 11: Execução de teste agente Edu



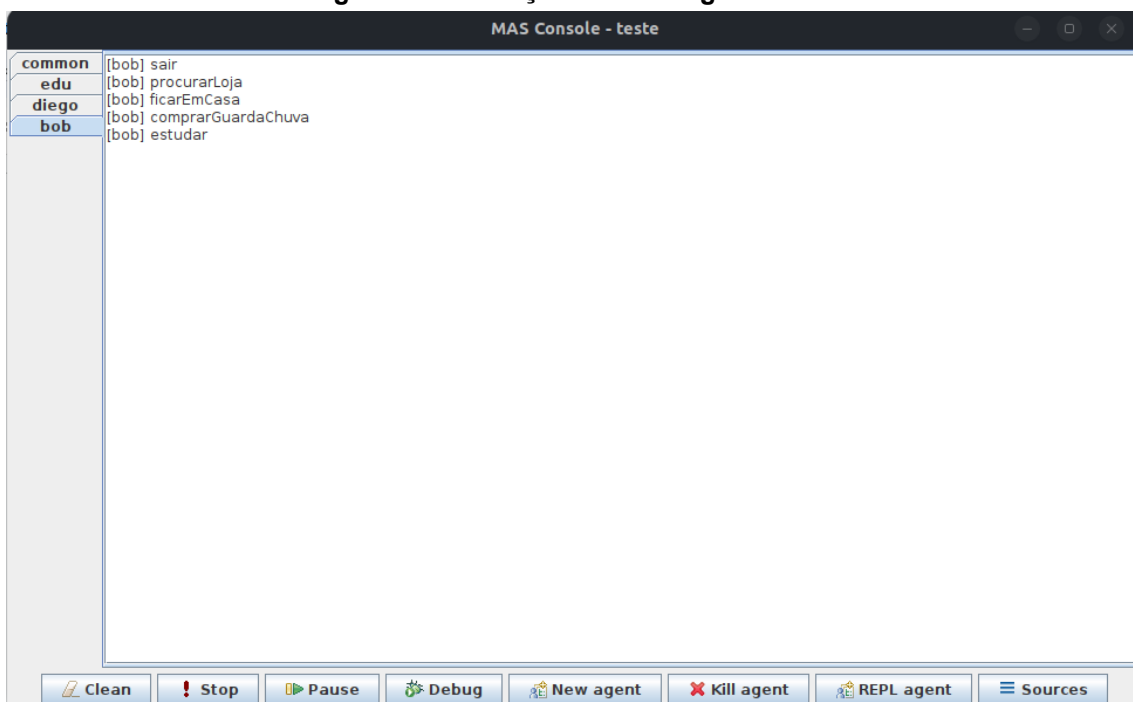
fonte: autoria própria

Figura 12: Execução de teste agente Diego



fonte: autoria própria

Figura 13: Execução de teste agente Bob



fonte: autoria própria

7. CONCLUSÃO

A análise sintática e análise léxica, com o uso do Flex e do Bison, demonstrou ser um componente essencial para a construção bem sucedida do Tradutor da Linguagem Natural NAG para a Linguagem Jason. A BNF desempenha um papel crucial na definição da estrutura da tradução, garantindo que os padrões linguísticos fossem respeitados.

Além disso, a organização do trabalho, com arquivos em C e um arquivo header.h, simplificou a manutenção e o desenvolvimento contínuo do tradutor. Este trabalho não apenas apresentou uma solução técnica viável, mas também lançou as bases para aplicações promissoras na área de tradução e processamento de linguagem natural.

8. REFERÊNCIAS

[1] RIGO, Sandro. Análise Léxica. Universidade Estadual de Campinas. Disponível em: <https://www.ic.unicamp.br/~sandro/cursos/mc910/slides/cap2-lex>. Acesso em: 15 de outubro de 2023.

[2] JUNIOR, Celso. Compiladores. Universidade Estadual Paulista. Disponível em: <https://docs.fct.unesp.br/docentes/dmec/olivete/compiladores/arquivos/Aula5.pdf>. Acesso em: 15 de outubro de 2023.

[3]