Processamento de Linguagens
(3º ano de Licenciatura em Engenharia Informática)

Compilador para *LPIS*

Relatório de Desenvolvimento

André Geraldes (67673) Patrícia Barros (67665)
Sandra Ferreira (67709)

6 de Junho de 2015

Resumo

Este relatório descreve todo o processo de desenvolvimento e decisões tomadas para a realização do segundo trabalho prático da Unidade Curricular de Processamento de Linguagens.
O problema a resolver consiste na criação de uma linguagem imperativa simples e da respetiva **GIC** seguida do desenvolvimento de um compilador para a mesma que gera pseudo-código *Assembly*.

Conteúdo

Capítulo 1

Introdução

Contexto

Este trabalho foi realizado no âmbito da Unidade Curricular de Processamento de Linguagens e é inserido no contexto da matéria leccionada nas aulas acerca de expressões regulares, gramáticas, analisadores léxicos e analisadores sintáticos.

Motivação

A construção de uma linguagem de programação e da respetiva gramática são formas extremamente interessantes de consolidar os conhecimentos adquiridos nas aulas sobre as matérias relacionadas.

Objetivos

Com este trabalho pretende-se gerar um compilador para uma linguagem totalmente criada pelo grupo, deixada a cargo da sua criatividade, e através desse compilador gerar código pseudo-Assembly para uma Máquina Virtual. Com isto é pretendido que o grupo aumente a sua experiência em *engenharia de linguagens* e *programação generativa*. O aumento da experiência do uso do ambiente *Linux* e da linguagem C também fazem parte dos objetivos deste trabalho.

Resultados

Após a sua conclusão, este trabalho permitiu-nos observar as instruções Assembly geradas pelo nosso compilador e compará-las com o programa fonte. Os resultados foram os esperados para todos os testes efetuados, donde podemos concluir que o trabalho foi bem executado.

1.1 Estrutura do Relatório

A elaboração deste documento teve por base a estrutura do relatório fornecida pelo docente.
O relatório encontra-se então estruturado da seguinte forma: no Capítulo 1 é feita uma contextualização ao assunto tratado neste trabalho, seguindo-se uma introdução onde são apresentadas as metas de aprendizagem pretendidas.
Posteriormente, no Capítulo 2, é exposto o enunciado do trabalho e a descrição do problema.
Imediatamente após, no Capítulo 3, é apresentada a linguagem criada, a gramática correspondente e descrita a forma como foi gerado o código Assembly. São também descritos os módulos do programa e as estruturas utilizadas.
No Capítulo 4 são apresentados os testes realizados e os seus resultados.
Por último, no Capítulo 5, faz-se uma análise crítica relativa quer ao desenvolvimento do projeto quer ao seu estado final e ainda é feita uma abordagem ao trabalho futuro.

Capítulo 2

Análise e Especificação

2.1 Enunciado

Pretende-se que comece por definir uma linguagem de programação interativa simples (**LPIS**), a seu gosto.
Apenas deve ter em consideração que a **LPIS** terá de permitir manusear variáveis do tipo inteiro (escalar ou *array*) e realizar as operações básicas como atribuições de expressões a variáveis, ler do *standard input* e escrever no *standard output*. As instruções vulgares para controlo de fluxo de execução - *condicional* e *cíclica*- devem estar também previstas.
Sobre inteiros, estão disponíveis as habituais operações aritméticas, relacionais e lógicas, bem como operação de indexação sobre *arrays*.
Como é da praxe neste tipo de linguagens, as nossas variáveis deverão ser declaradas no inicio do programa e não pode haver re-declarações, nem utilizações sem declaração prévia; se nada for explicitado, o valor da variável após a declaração é indefinido.
Desenvolva, então, um compilador para a **LPIS**, com base na **GIC** criada acima e recurso ao Gerador **Yacc/Flex** ou **AnTLR**.
O compilador da **LPIS** deve gerar **pseudo-código**, Assembly da Máquina Virtual VM cuja documentação completa será disponibilizada no Bb.

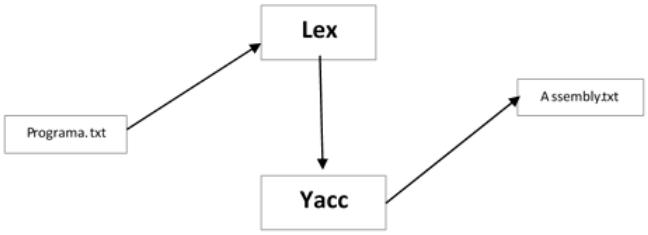
2.2 Descrição do Problema

Como descrito na secção anterior o desafio que nos foi proposto consiste na criação de uma linguagem imperativa simples apenas com as características mais básicas, o desenvolvimento de uma **GIC** para a mesma e ainda a criação de um compilador gerador de **pseudo-código** Assembly.
Posto isto tivemos então de, primeiramente, pensar e desenhar uma linguagem imperativa ao nosso gosto, seguindo os requisitos pedidos. De seguida foi necessário criarmos uma gramática que a descrevesse e, finalmente, com recurso

a essa gramática, tornar possível o armazenamento dos dados vindos de um programa escrito na nossa linguagem necessários à geração do **pseudo-código** Assembly.

Capítulo 3

Concepção/desenho da Resolução



Como representado na figura 1 a estrutura do nosso projeto terá de passar pelo desenvolvimento de um analisador léxico em *Flex* que fará o reconhecimento dos *tokens* que são utilizados no analisador semântico que gera o código *Assembly*.

3.1 Desenho da Linguagem

O principal princípio que seguimos aquando de decidir as caraterísticas da nossa linguagem foi a simplicidade desta, desenhando-a de forma a que os passos seguintes do trabalho não fossem excessivamente complicados. Posto isto, decidimos que a nossa linguagem suportaria apenas os requisitos definidos no enunciado.
Concluímos que nos seria útil que qualquer programa da nossa linguagem estivesse dividido em duas partes: as declarações de variáveis e o corpo, constituído por instruções. Percebemos também que seria mais fácil se tanto a operação condicional como o ciclo tivessem sempre marcadores do seu início e fim, e escolhemos neste caso usar como marcadores palavras reservadas para os delimitarem.
Assim, um programa tem **obrigatoriamente** de iniciar com a palavra reservada

BEGIN e fazer-se seguir de uma ou muitas declarações de variáveis (nunca poderá haver um programa sem nenhuma declaração pois consideramos que um programa sem variáveis não faz sentido), declarações estas que são constituídas por um tipo e uma ou mais variáveis (palavras) separadas por vírgulas. No caso de ser um *array* terá também de ser declarado o seu tamanho (um número entre parenteses colocado logo a seguir ao tipo). Cada declaração termina obrigatoriamente com um **;**. Os tipos das variáveis são palavras reservadas, podendo ser **INT** ou **ARRAY**.
Após terminadas as declarações damos então início ao corpo do programa, através da palavra reservada **BODY**.
O corpo do programa é constituído por uma ou mais instruções (mais uma vez, também o corpo não pode ser vazio pois achamos que isso não faria sentido). Cada instrução pode ser de um de cinco tipos diferentes: atribuição, condição, ciclo, escrita ou leitura e termina sempre com um ponto e vírgula.
Uma atribuição consiste numa variável seguida de um sinal de igual que precede uma expressão. Neste caso uma variável pode ser uma palavra ou uma palavra seguida de uma expressão entre parênteses (para a operação de indexação de um *array*). Por sua vez uma expressão é um termo ou uma expressão seguida de um operador de adição que precede um termo, sendo que um termo é um fator ou um termo seguido de um operador multiplicativo que precede um fator. Um fator pode ser uma palavra, um número ou uma condição entre parênteses, sendo que uma condição neste caso é uma expressão ou duas expressões separadas por um operador relacional.
Uma condição é sempre iniciada pela palavra reservada **IF** que precede uma ou várias instruções e pode ou não ser seguida da palavra reservada **ELSE** seguida de mais instruções. Uma condição termina sempre com a palavra reservada **ENDIF**.
Um ciclo é sempre iniciado pela palavra reservada **WHILE** seguida de uma condição entre parênteses que precede uma ou mais instruções e termina sempre com a palavra reservada **ENDWHILE**. As instruções de leitura escrita são representadas por duas palavras reservadas: **READ** e **WRITE** respetivamente. No caso da leitura a palavra reservada segue-se de uma variável (inteira ou a posição de um *array*) entre parênteses, e no caso da escrita segue-se de uma expressão entre parênteses.
Os símbolos que utilizamos para os operadores aditivos, multiplicativos e relacionais são os seguintes:

```
opA  "+"|"-"|"||"
opM "&&"|"*"|"/"
opR ">>"|"<<"|">="|"<="|"=="|"|=|"
```

Para melhor compreensão da descrição acima segue-se um exemplo de um programa simples escrito na nossa linguagem.

```
1 BEGIN
2 INT x,y,y;
3 ARRAY(100) vect , vects;
```

```
4 BODY
5 x=1;
6 y=1+2;
7 IF (x>>5) y=y+1; ENDIF;
8 x=2;
9 WHILE (y<<2) y=y+1; IF (x>>5) w=y+1; ENDIF; ENDWHILE;
10 IF (x==1) y=y+1; y=2; ELSE y=3; ENDIF;
11 END
```

3.2 Desenho da Gramática

Após desenhada e descrita a linguagem imperativa que iremos usar, resta transformar todas as regras que definimos sobre ela numa **GIC**.
Os símbolos não-terminais da gramática que definimos são os seguintes: programa, declaracoes, declaracao, tipo, variaveis, variavel instrucoes, instrucao, atribuicao, expressao, termo, fator, condicao, senao, ciclo e cond.
Por sua vez os símbolos terminais são: num, pal, IF, ENDIF, WHILE, ENDWHILE, BEGINP, MIDDLE, ENDP, INT, ARRAY, OPM, OPR, OPA, ";", ",", "(",")", "", e "=".
Na concepção da nossa gramática utilizamos em todos os casos recursividade à esquerda. Segue-se então a gramática gerada:

```
programa : BEGINP declaracoes MIDDLE instrucoes ENDP

declaracoes : declaracao
| declaracoes  declaracao
;

declaracao : tipo variaveis ';'
| tipo '(' num ')' variaveis ';'
;

tipo : INT
| ARRAY
;

variaveis : variavel
| variaveis ',' variavel
;

variavel  : pal
;

instrucoes : instrucao ';'
| instrucoes  instrucao ';'
```

```
;

instrucao : atribuicao
|  condicao
|  ciclo
| READ '(' pal ')'
|  READ  '(' pal '('expressao')' ')'
| WRITE '('expressao')'
;

atribuicao  : pal '=' expressao
| pal
;


cond : expressao
|  expressao OPR expressao
;

expressao :  termo
|  expressao OPA termo
;

termo :  fator
|  termo OPM fator
;

fator :  pal
|  pal '('expressao')'
|  num
| '(' cond ')'
;

condicao :  IF '('cond')'  instrucoes   senao
;

senao : ENDIF
|  ELSE instrucoes  ENDIF
;

ciclo  : WHILE '('cond')' instrucoes ENDWHILE
;
```

3.3 Geração de Pseudo-Código Assembly

Para gerar o código Assembly do programa é necessário definir ações semânticas no *YACC*. Estas ações semânticas são blocos de código C que são executados aquando o reconhecimento da expressão que os antecede.
Na parte das declarações de variáveis é necessário inseri-las na estrutura que críamos e gerar o código Assembly para alocar memória para elas. Para isso decidimos usar variáveis globais para guardar o tamanho, o endereço, o tipo e o nome de uma variável. Sempre que o analisador sintático reconhece uma declaração de um inteiro coloca a variável correspondente a 1 e se reconhecer um *array* coloca nessa mesma variável o seu tamanho. Da mesma forma é atualizada a variável correspondente ao tipo. Quanto à variável correspondente ao endereço é uma variável global com valor inicial de zero que é incrementada sempre que uma nova variável é adicionada com sucesso à estrutura. Quando é encontrado o nome de uma variável já tem então guardado o seu tamanho e tipo e encontra-se já em condições de guardá-la na estrutura de dados. Nesta altura é feita a verificação da já existência de uma variável com o mesmo nome ou não. Caso já exista o programa termina e é gerada uma mensagem de erro. Caso contrário é gerado o código Assembly para empilhar a variável na *Stack*: PUSHN X, em que X é o tamanho da variável. De seguida é incrementada a variável do endereço tantas vezes quanto o tamanho da variável.
Após todas as declarações terem sido reconhecidas é gerada a instrução Assembly START que indica o início das instruções do programa.
Em todas as instruções, sempre que é encontrada uma variável é feita a verificação da sua existência na estrutura. Se não existir o programa termina com um erro. Se existir, é guardado o seu registo para ser utilizado no código Assembly. No caso de se tratar de um acesso a uma posição de um *array* é também feita a verificação do seu tipo.
Nas instruções de atribuição o código Assembly gerado, no caso de ser uma variável inteira escalar, é apenas um STOREG X, em que X é o endereço da variável em questão. No caso de ser um acesso a um vetor, é necessário aquando do reconhecimento do nome da variável efetuar um PUSHGP seguido de um PUSHI X, em que X é o endereço da variável em causa, e de um PADD. De seguida, após o reconhecimento das expressões constituintes da atribuição é gerado um STOREN.
Para as expressões, termos e condições que utilizam operadores aditivos, multiplicativos e relacionais, respetivamente, foi feito o reconhecimento de que operador estava a ser utilizado, e tendo em conta isso gerado o código Assembly respetivo a essa operação. De notar que não existe uma instrução Assembly para as operações lógicas e e ou, logo utilizamos a multiplicação e a soma, respetivamente, para representá-las.
Para as instruções de escrita o único código Assembly necessário é um WRITEI.
Para as instruções de leitura, no caso de a leitura ser feita para uma variável escalar, é gerado o código READ para ler uma*string* do teclado, seguido do código ATOI para transformá-la num inteiro, e de seguida guardá-la no endereço da variável em questão com a instrução STOREG X, em que X é o endereço. Já no

caso de se tratar de um vetor é necessário um PUSHGP seguido de um PUSHIX e de um PADD antes de proceder à leitura em si, que se dá da mesma forma que no caso do escalar, mas substituindo o STOREG por um STOREN.
No caso dos ciclos utilizamos *labels* para marcar o início e o fim de um ciclo. Depois de reconhecida a expressão da condição do ciclo fazemos um JZ (salta se a expressão for falsa) para a *label* do fim de ciclo. No fim do ciclo temos um JUMP não condicional que salta sempre para o início do ciclo (antes da condição).
Quando às instruções condicionais, recorrendo também às *labels*, e aos *jumps* seguimos a seguinte lógica: após a condição do *if* temos um JZ para o início do bloco *else* caso este exista e para o fim do *if* caso contrário.

3.4 Módulos da Aplicação

tp2.l Ficheiro com código de um analisador léxico que serve de suporte à nossa gramática.
tp2.y Ficheiro que contém o código **YACC** correspondente à nossa gramática e inclui nesta o código necessário para a geração do código Assembly resultante.
arvore.c Ficheiro onde se encontra o código da estrutura definida para guardar as informações das variáveis encontradas no programa.
arvore.h Ficheiro com a estrutura definida para guardar informações das variáveis e com as assinaturas das funções definidas no arvore.c.

3.5 Estruturas de Dados

Para guardar as informações relativas às variáveis que se encontram nos programas escritos na nossa linguagem foi necessário criar uma estrutura.
A estrutura mais apropriada para este efeito é uma Tabela de Hash devido a ser mais eficiente a procura por *strings*, no entanto, devido à nossa inexperiência em trabalhar com esse tipo de estrutura, optamos por utilizar uma Árvore Binária de Procura, com a qual estamos mais familiarizados, para este efeito.
Foi necessário guardar, para cada variável declarada, o seu nome, tipo, registo e tamanho.
Decidimos que o tamanho seria 1 para todas as variáveis de tipo inteiro, ficando os *arrays* com o tamanho definido aquando da declaração.
O valor do registo é sequencial, ou seja, existe uma variável global inteira que é incrementada de cada vez que uma variável é declarada. No caso dessa variável ser um *array* o valor do registo é incrementado tantas vezes quanto o tamanho do *array*.
Segue-se o código referente à declaração da estrutura que utilizamos para guardar estas informações:

```
typedef struct node {
char* nome;
```

```
char* tipo;
int tamanho;
int registo;
struct node *esq;
struct node *dir;
} Node, *Tree;
```

Capítulo 4

Testes realizados e Resultados

De seguida apresentamos exemplos de ficheiros de entrada de programas escritos na nossa linguagem e o respetivo ficheiro Assembly gerado.

4.1 Exemplo 1

Ficheiro de Entrada:

```
BEGIN
INT x,y,z;
ARRAY(20) vect;
BODY
x=1;
y=1+2;
x=2;
vect(10)=4;
x=y*vect(5);
vect(4)=z;
READ(z);
WRITE(x);
IF(x>>y)
WRITE(vect(1));
ELSE
WRITE(vect(2));
ENDIF;
WHILE((1<<z)&&(x>>2))
x=x+1;
```

```
ENDWHILE;
END
```

Ficheiro de Saída:

```
PUSHN 1
PUSHN 1
PUSHN 1
PUSHN 20
START
PUSHI 1
STOREG 0
PUSHI 1
PUSHI 2
ADD
STOREG 1
PUSHI 2
STOREG 0
PUSHGP
PUSHI 3
PADD
PUSHI 10
PUSHI 4
STOREN
LOADG 1
PUSHGP
PUSHI 3
PADD
PUSHI 5
LOADN
MUL
STOREG 0
PUSHGP
PUSHI 3
PADD
PUSHI 4
LOADG 2
STOREN
READ
ATOI
STOREG 2
LOADG 0
WRITEI
LOADG 0
LOADG 1
SUP
JZ lab0
```

```
PUSHGP
PUSHI 3
PADD
PUSHI 1
LOADN
WRITEI
JUMP fse0
lab0: NOP
PUSHGP
PUSHI 3
PADD
PUSHI 2
LOADN
WRITEI
fse0: NOP
labinicio1: NOP
PUSHI 1
LOADG 2
INF
LOADG 0
PUSHI 2
SUP
MUL
JZ fwhile1
LOADG 0
PUSHI 1
ADD
STOREG 0
JUMP labinicio2
fwhile1: NOP
STOP
```

Capítulo 5

Conclusão

Após descrito todo o processo de desenvolvimento deste trabalho, desde o desenho da linguagem até à geração do código Assembly, passando pela construção da gramática, e apresentados os testes e respetivos resultados resta apresentar uma breve conclusão sobre todo o processo.
Atualmente o projeto encontra-se totalmente funcional: recebendo um ficheiro com um programa escrito na nossa linguagem gera sempre as instruções Assembly corretas.
No entanto, como trabalho futuro, seria interessante mudar a estrutura de dados que guarda a informação sobre as variáveis numa tabela de *Hash* para ser mais eficiente.
Pessoalmente achamos este projeto muito cativante não só pelo incentivo à nossa criatividade como por acharmos todo o conceito de criar um compilador muito interessante, tarefa que antes de fazermos este projeto nos parecia dificílima mas que depois de percebermos o seu funcionamento se tornou muito acessível.

Apêndice A

Código Flex

```
%{
        //#include "y.tab.h"
        //void yyerror(char *);
%}

num             [0-9]+
palavra                 [a-zA-Z]+
espacos         [ \t]
opA             "+"|"-"|"||"
opM                     "&&"|"*"|"/"
opR                     ">>"|"<<"|">="|"<="|"=="|"|="
inicio          "BEGIN"
corpo           "BODY"
fim                     "END"
condicao        "IF"
ifelse          "ELSE"
fimCond         "ENDIF"
ciclo           "WHILE"
fimciclo        "ENDWHILE"
inteiro         "INT"
vetor           "ARRAY"
read            "READ"
write           "WRITE"

%option yylineno

%%
[,;()={}]               { return yytext[0]; }
{inicio}                        { return BEGINP; }
{corpo}                         { return MIDDLE; }
{fim}                           { return ENDP; }
{condicao}                      { return IF; }
{ifelse}                        { return ELSE; }
```

```
34 {fimCond}                          { return ENDIF; }
35 {ciclo}                           { return WHILE; }
36 {fimciclo}                        { return ENDWHILE; }
37 {opA}                             { yylval.vals = strdup(yytext)
       ;return (OPA); }
38 {opR}                             { yylval.vals = strdup(yytext)
       ;return (OPR); }
39 {opM}                             { yylval.vals = strdup(yytext)
       ;return (OPM);}
40 {vetor}                           { yylval.vals = strdup(yytext)
       ;return (ARRAY);}
41 {inteiro}                         { yylval.vals = strdup(yytext)
       ;return (INT);}
42 {read}                            { return (READ);}
43 {write}                           { return (WRITE);}
44 {num}                             { yylval.vali = atoi(yytext);
       return num; }
45 {palavra}                  { yylval.vals = strdup(yytext); return
        pal; }
46
47 .|\n                              { ; }
48
49
50
51 %%
52
53 int yywrap(){
54         return(1);
55 }
```

Apêndice B

Código Yacc

```
%{
#include <stdio.h>
#include <string.h>
#include "arvore.c"


Tree arvore;

#define inteiro 1
#define vetor 2


FILE *file;
int cont=0;
int f;
int aux;
int proxReg=0;
int registo;
char* varAtual;
int tip;
int tamanho=1;



//int yyerror(char *s);
//extern int yylineno;
%}

%union {
int vali;
```

```
char * vals;
}



%token <vali>num
%token <vals>pal OPM OPA OPR
%token BEGINP MIDDLE ENDP IF ENDIF WHILE ENDWHILE ELSE WRITE READ INT ARRAY



%%

programa : BEGINP declaracoes{ fprintf(file,"START\n"); } MIDDLE
instrucoes ENDP{fprintf(file,"STOP\n");f=fclose(file);}

declaracoes : declaracao
| declaracoes  declaracao
;

declaracao : tipo {tamanho=1;} variaveis ';'
| tipo '(' num ')' {tamanho=$3;} variaveis ';'
;

tipo : INT  {tip=inteiro;}
| ARRAY {tip=vetor;}
;

variaveis : variavel  { ; }
| variaveis ',' variavel { ; }
;

variavel  : pal  { varAtual=$1;
aux=insertBinTree(arvore, varAtual, tip, tamanho, proxReg);
if (aux==-1) { yyerror("A variável já foi declarada!");
exit(0);} else
{ fprintf(file,"PUSHN %d\n",tamanho);
proxReg=proxReg+tamanho;} }
;

instrucoes : instrucao ';'{ ; }
| instrucoes  instrucao ';'{ ; }
;

instrucao : atribuicao { ; }
|  condicao { ; }
```

```
|  ciclo { ; }
| READ '(' pal ')'  { aux=existsBinTree(arvore,$3); if(aux==0)
{ yyerror("A variável não foi declarada!");
exit(0); } else { registo=registoVar(arvore,$3);
fprintf(file,"READ\nATOI\nSTOREG %d\n",registo);} }
|  READ  '(' pal  { aux=existsBinTree(arvore,$3); if(aux==0)
{ yyerror("A variável não foi declarada!"); exit(0); }
else { tip=tipoVar(arvore,$3); if(tip!=vetor)
{ yyerror("A variável não é um array!"); exit(0);} else
{ registo=registoVar(arvore,$3);
fprintf(file,"PUSHGP\nPUSHI %d\nPADD\n",registo);} } }
'('expressao')' ')'{ fprintf(file,"READ\nATOI\nSTOREN\n"); }
| WRITE '('expressao')'  { fprintf(file,"WRITEI\n"); }
;

atribuicao  : pal '=' expressao  { aux=existsBinTree(arvore,$1); if(aux==0)
{yyerror("A variável não foi declarada!"); exit(0); } else
{ registo=registoVar(arvore,$1); fprintf(file,"STOREG %d\n",registo); } }
| pal  { aux=existsBinTree(arvore,$1); if(aux==0)
{yyerror("A variável não foi declarada!"); exit(0); } else
{  tip=tipoVar(arvore,$1); if(tip!=vetor)
{ yyerror("A variável não é um array!"); exit(0);} else
{ registo=registoVar(arvore,$1);
fprintf(file,"PUSHGP\nPUSHI %d\nPADD\n",registo); } } }
'('expressao')''=' expressao  { fprintf(file,"STOREN\n"); }
;


cond : expressao { ; }
|  expressao OPR expressao  { if( strcmp($2,">>")==0 ) { fprintf(file,"SUP\n"); }
if( strcmp($2,"<<")==0 ) { fprintf(file,"INF\n"); }
if( strcmp($2,"==")==0 ) { fprintf(file,"EQUAL\n"); }
if( strcmp($2,">=")==0 ) { fprintf(file,"SUPEQ\n"); }
if( strcmp($2,"<=")==0 ) { fprintf(file,"INFEQ\n"); }
if( strcmp($2,"|=|")==0 ) { fprintf(file,"EQUAL\nNOT\n"); } }
;

expressao :  termo { ; }
|  expressao OPA termo { if( strcmp($2,"+")==0 ) { fprintf(file,"ADD\n"); }
if( strcmp($2,"-")==0 ) { fprintf(file,"SUB\n"); }
if( strcmp($2,"||")==0 ) { fprintf(file,"ADD\n"); } }
;

termo :  fator { ; }
|  termo OPM fator { if( strcmp($2,"*")==0 ) { fprintf(file,"MUL\n"); }
if( strcmp($2,"/")==0 ) { fprintf(file,"DIV\n"); }
```

```
if( strcmp($2,"&&")==0 ) { fprintf(file,"MUL\n"); } }
;

fator :  pal { aux=existsBinTree(arvore,$1); if(aux==0)
{yyerror("A variável não foi declarada!"); exit(0); } else
{ registo=registoVar(arvore,$1); fprintf(file,"LOADG %d\n",registo);}}
|  pal   { aux=existsBinTree(arvore,$1); if(aux==0)
{ yyerror("A variável não foi declarada!"); exit(0); } else
{ tip=tipoVar(arvore,$1); if(tip!=vetor)
{ yyerror("A variável não é um array!"); exit(0);} else
{ registo=registoVar(arvore,$1);
fprintf(file,"PUSHGP\nPUSHI %d\nPADD\n",registo);} } }
'('expressao')'{ fprintf(file,"LOADN\n");}
|  num { fprintf(file,"PUSHI %d\n",$1); }
| '(' cond ')'
;

condicao :  IF '('cond')' { fprintf(file,"JZ lab%d\n",cont);}  instrucoes   senao
;

senao : ENDIF { fprintf(file,"lab%d: NOP\n",cont++); }
|  ELSE { fprintf(file,"JUMP fse%d\nlab%d: NOP\n",cont,cont); }
instrucoes  ENDIF { fprintf(file,"fse%d: NOP\n",cont++); }
;

ciclo  : WHILE { fprintf(file,"labinicio%d: NOP\n",cont);}'('cond')'
{ fprintf(file,"JZ fwhile%d\n",cont);} instrucoes ENDWHILE
{ fprintf(file,"JUMP labinicio%d\nfwhile%d: NOP\n",cont,cont++);}
;

%%
#include "lex.yy.c"

int yyerror(char *s) {
fprintf(stderr, "Erro na linha ( %d! ) %s\n", yylineno, s);
return 0;
}

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

arvore=initBinTree();
file=fopen("assembly.txt","w+");

yyparse();

return 0;
```

```
}
```

Apêndice C

Código da Árvore Binária

```
1 #include <stdio.h>
2
3 #include <string.h>
4 #include <stdlib.h>
5 #include "arvore.h"
6
7 Tree initBinTree(){
8         Tree t=(Tree)malloc(sizeof(struct node));
9
10        t->nome=(char*)malloc(sizeof(char*));
11        t->nome=NULL;
12        t->tipo=-1;
13        t->tamanho=-1;
14        t->registo=-1;
15        t->esq=NULL;
16        t->dir=NULL;
17
18        return t;
19 }
20
21
22
23 int existsBinTree(Tree t, char* n){
24        Tree aux;
25
26        if(t->nome==NULL)
27                return -1;
28        aux=t;
29
30        while(aux != NULL && (strcmp(n,aux->nome)!=0)){
31
32                if((strcmp(n,aux->nome)>0)) //procura a
                      direita
```

```
33                          aux=aux->dir;
34
35                  else //procura a esquerda
36                          aux=aux->esq;
37          }
38
39          if(aux==NULL)
40           //se aux==NULL, entao o elemento e nao existe
41                  return 0;
42          return 1;
43  }
44
45  int insertBinTree(Tree t, char* n, int ty, int tam, int reg){
46          Tree new;
47
48
49
50          if(existsBinTree(t,n)==1)
51                  //
52                  return -1;
53
54          if(!t->nome){
55                  t->nome=n;
56                  t->tipo=ty;
57                  t->tamanho=tam;
58                  t->registo=reg;
59                  t->esq=NULL;
60                  t->dir=NULL;
61          }
62
63          else{
64                  new=(Tree)malloc(sizeof(struct node));
65                  new->nome=n;
66                  new->tipo=ty;
67                  new->tamanho=tam;
68                  new->registo=reg;
69                  new->esq=NULL;
70                  new->dir=NULL;
71                  Tree aux = t;
72
73                  while(aux!=NULL){
74                          if((strcmp(n,aux->nome)>0)){
75                                  if(aux->dir==NULL){
76                                          aux->dir=new;
77                                          return 0;}
78
79                                  else
80                                          aux=aux->dir;
81                          }
82                          else{
```

```
83                                      if(aux->esq==NULL){
84                                              aux->esq=new;
85                                              return 0;
86                                      }
87                                      else{
88                                              aux=aux->esq;
89                                      }
90                              }
91                      }
92              }
93
94              return 0;
95      }
96
97      int registoVar (Tree t, char *nomeVar){
98              Tree aux;
99
100             if(t->nome==NULL)
101                     return -1;
102             aux=t;
103
104             while(aux != NULL && (strcmp(nomeVar,aux->nome)!=0)){
105
106                     if((strcmp(nomeVar,aux->nome)>0)) //procura a
                            direita
107                             aux=aux->dir;
108
109                     else //procura a esquerda
110                             aux=aux->esq;
111             }
112
113             if(aux==NULL)
114              //se aux==NULL, entao o elemento e nao existe
115                     return -1;
116
117             return aux->registo;
118     }
119
120
121     int tipoVar(Tree t, char* nomeVar){
122
123             Tree aux;
124
125             if(t->nome==NULL)
126                     return -1;
127             aux=t;
128
129             while(aux != NULL && (strcmp(nomeVar,aux->nome)!=0)){
130
```

```
131                 if((strcmp(nomeVar,aux->nome)>0)) //procura a
                        direita
132                         aux=aux->dir;
133
134                 else //procura a esquerda
135                         aux=aux->esq;
136         }
137
138         if(aux==NULL)
139          //se aux==NULL, entao o elemento e nao existe
140                 return -1;
141
142         return aux->tipo;
143
144         return 0;
145 }
```