

PLC

Trabalho Prático N°3 Compilador de Pickle

André Sequeira
A76372

Daniel Carvalho
A74718

15 de Janeiro de 2018

Resumo

O trabalho foi efetuado no âmbito da disciplina de Processamento de Linguagens e compiladores. Consiste na elaboração de uma linguagem de programação imperativa original, e o seu respetivo compilador. Compilador este que tem como missão gerar código assembly destinado a uma máquina virtual, implementada pelos docentes da disciplina. Nas figuras 1, 2 e 3 é possível verificar um pequeno exemplo que demonstra o código assembly gerado a partir dum programa implementado em Pickle e o respetivo resultado obtido na máquina virtual.

```
1 go
2 int y <- 2
3 int z <- 3
4 int t
5 t <- z * 2 + y
6 put 'resultado - ' t
7 end
```

Figura 1: Código em Pickle

```

1  start
2  pushi 2
3  pushi 3
4  pushi 0
5  pushg 1
6
7  pushi 2
8  mul
9
10 pushg 0
11 add
12 storeg 2
13 pushs "resultado - "
14 writes
15 pushg 2
16 writei
17 stop
18

```

Figura 2: Código Assembly gerado

Code

Index	Instruction	Value	Value	State
0	start			
1	pushi	2		
2	pushi	3		
3	pushi	0		
4	pushi	1		
5	pushi	2		
6	mul			
7	pushg	0		
8	add			
9	storeg	2		
10	pushs	"resultad...		
11	writes			
12	pushg	2		
13	writel			
14	stop			

Heap

Index	Value	Type

OPStack

Index	Value	Type
	0/2	integer
	1/3	integer
	2/8	integer

Call Stack

Pc value	Fp value

Pc value	Fp value

Strings: resultado - B

Figura 3: Máquina virtual

Conteúdo

1	Introdução	2
1.1	Estrutura do Relatório	2
2	Problema	3
2.1	Enunciado	3
3	Resolução do Problema	4
3.1	Conceção	4
3.1.1	Linguagem Pickle	4
3.1.2	Gerar Assembly	4
3.2	Implementação	5
3.2.1	Analizador léxico	5
3.2.2	Analizador Sintático	6
3.2.3	Analizador Sintático - Ações	7
3.2.4	Funções implementadas em C	8
4	Codificação e Testes	10
4.1	Alternativas, Decisões e Problemas de Implementação	12
4.2	Testes realizados e Resultados	13
5	Conclusão	16
A	Código do Programa	17

Capítulo 1

Introdução

O trabalho consiste, após se ter desenhado, numa primeira instância, uma linguagem de cariz imperativa, em projetar um analisador sintático, feito em Flex, de forma a reconhecer os terminais da nossa linguagem e retorna-los como tokens a um analisador léxico, implementado em Yacc, cuja função será analisar sequências destes e verificar se constituem frases válidas da nossa linguagem, e se sim, efetuar a respetiva ação que originará o resultado final, ou seja, o pseudo-código Assembly. Na figura 1.1 está presente um flowchart, para dar ao leitor uma melhor visualização dos processos envolventes no trabalho.

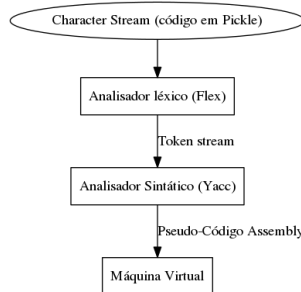


Figura 1.1: Flowchart

1.1 Estrutura do Relatório

No capítulo 2, faremos uma descrição formal do problema em questão, recorrendo ao enunciado proposto pelos Professores.

No capítulo 3, tentaremos descrever o mais detalhado possível a nossa conceção para a resolução do problema, e posteriormente mostraremos, com rigor, script a script, a nossa implementação, de forma ao leitor perceber os resultados obtidos que estarão presentes no capítulo 4. Também no capítulo 4 faremos uma breve análise ao trabalho realizado, referindo as decisões tomadas ao longo do trabalho, e os problemas que foram surgindo com a nossa implementação. Também neste capítulo faremos uma sugestão para uma distinta implementação e abordagem.

No quinto e último capítulo, estabeceremos uma última análise e reflexão sobre o trabalho, de modo a enriquecer a percepção do leitor face ao trabalho realizado.

Capítulo 2

Problema

2.1 Enunciado

Pretende-se que comece por definir uma linguagem de programação imperativa simples, a seu gosto. Apenas deve ter em consideração que essa linguagem terá de permitir:

- *declarar e manusear* variáveis atômicas do tipo *inteiro*, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas.
- *declarar e manusear* variáveis estruturadas do tipo *array* (a 1 ou 2 dimensões) de *inteiros*, em relação aos quais é apenas permitida a operação de indexação (índice inteiro).
- *efetuar* instruções algorítmicas básicas como a *atribuição de expressões a variáveis*.
- *ler* do *standard input* e *escrever* no *standard output*.
- *efetuar* instruções para controlo do fluxo de execução—*condicional* e *cíclica*—que possam ser aninhadas.
- *definir e invocar subprogramas* sem parâmetros mas que possam retornar um resultado atômico (opcional).

Como é da praxe neste tipo de linguagens, as variáveis deverão ser declaradas no início 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 0 (zero).

Desenvolva, então, um compilador para essa linguagem com base na GIC criada acima e com recurso ao Gerador Yacc/Flex.

O compilador deve gerar **pseudo-código**, **Assembly** da Máquina Virtual VM cuja documentação completa está disponibilizada no Bb.

Muito Importante:

Para a entrega do TP deve preparar um conjunto de testes (programas-fonte escritos na sua linguagem) e mostrar o código **Assembly** gerado bem como o programa a correr na máquina virtual VM. Esse conjunto terá de conter, no mínimo, os 6 exemplos que se seguem:

- ler 4 números e dizer se podem ser os lados de um quadrado.
- ler um inteiro N, depois ler N números e escrever o menor deles.
- ler N (constante do programa) números e calcular e imprimir o seu produtório.
- contar e imprimir os números impares de uma sequência de números naturais.
- ler e armazenar os elementos de um vetor de comprimento N; imprimir os valores por ordem decrescente após fazer a ordenação do array por trocas diretas.
- ler e armazenar N números num array; imprimir os valores por ordem inversa.

Capítulo 3

Resolução do Problema

3.1 Conceção

Com vista na elaboração de uma linguagem imperativa intuitiva e de fácil implementação do ponto de vista do programador, começamos por criar um analisador léxico em flex com o intuito de reconhecer os identificadores da nossa linguagem, de forma a retorna-los como tokens para que posteriormente o analisador sintático feito em yacc possa através de sequências destes fazer a análise sintática, recorrendo à gramática que caracteriza a nossa linguagem e por fim gerar o código assembly pretendido. De seguida explicaremos sucintamente a ideologia da linguagem Pickle, e posteriormente a conceção para a geração do pseudo-código Assembly gerado.

3.1.1 Linguagem Pickle

Como referido anteriormente, a nossa linguagem tem como objetivo ser intuitiva e de fácil implementação do ponto de vista do programador, dito isto, a nossa linguagem permite declarar e manusear variáveis atómicas dos tipos inteiro e string, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas. Também é permitida a declaração e manipulação de arrays de uma dimensão e instruções de controlo de fluxo tais como expressões condicionais e cíclicas. Um exemplo que demonstra a versatilidade da nossa linguagem, e a rapidez com que se pode implementar algo, deve-se a um pormenor que não gostamos na linguagem C, que é nomeadamente quando temos de mudar os valores de duas variáveis, ter de chamar uma variável temporária que guarda um dos valores para posteriormente se proceder a troca o que são três instruções. Em Pickle, conseguimos o mesmo apenas com uma instrução, dando ao compilador o trabalho de fazer as trocas por nós, e dando ao programador a liberdade de escrever apenas "swap x y" no seu código. É importante referir que a "função de ordem superior" SWAP, funciona para qualquer tipo na linguagem Pickle, sejam variáveis inteiros ou arrays. Por fim também é permitido ao programador o uso de funções auxiliares com ou sem parâmetros. Na figura 3.1, está presente um pequeno texto representado as instruções da nossa linguagem, com a tradução de cada uma das instruções para linguagem C, de forma a que o leitor tenha um primeiro contacto com a linguagem, tendo uma base de comparação.

3.1.2 Gerar Assembly

Esta é a parte crucial do trabalho, sendo que temos de trabalhar os tokens retornados pelo analisador sintático de forma a que o analisador léxico faça a sua análise e com sequências destes, produza o respetivo pseudo-código Assembly. Com o intuito de gerar o código correto, temos de ter atenção à exata localização de cada variável na stack, logo é imperativo arranjarmos uma forma de guardar onde cada variável ocorre na stack, dito isto, implementamos um array que nos guarda o índice onde estas variáveis ocorrem. Como foi visto na descrição da linguagem Pickle, quando queremos ler valores do utilizador ou imprimir valores, as instruções em Pickle não referem os tipos das variáveis, pois achamos que o compilador deveria tratar isso por nós de forma a permitir ao programador, escrever menos e focar no mais importante. Dito isto, a nossa ideia foi criar um array que nos guarda o tipo de cada variável aquando da declaração da mesma, para posteriormente quando esta variável for chamada, através do uso de alguma manipulação de C, podermos produzir assembly adequado para o tipo da variável. É importante referir que a nossa linguagem

```

declarações:
int x ---- int x;
int x <- 2 ---- int x = 2;
string a ---- char *a;
string a = 'Grupo 12' ---- char *a = "Grupo 12";
int x[5] ---- int x[5];
int x[] <- {1 2 3 4 5} ---- int x[] = {1,2,3,4,5};
int x[y] ---- int x[y];

Atribuições, instruções e controles de fluxo:

go ---- main()
a <- 2 ---- a = 2;
2 -> a ---- a = 2;
a <- b ---- a = b;
get x ---- scanf
put x ---- printf
put x[i] ---- printf
x[i] <- x[j] ---- x[i] = x[j]
i++ ---- i++;
i-- ---- i--;
over ---- break;
out ---- return
if i < 5: something: ---- if(i<5){something} else {something else};
ahoy: something else:

i < 5 do:
something
i++: ---- while(i<5){something; i++;}
swap x y ---- tmp = x; x = y; y = tmp;

```

Figura 3.1: Tradução Pickle - C

utiliza funções auxiliares, dito isto, é de extrema relevância saber, a cada operação feita se a variável pertence ao main ou é uma variável que está definida dentro de uma função auxiliar, pois têm significados diferentes na medida que o pseudo-código gerado, não é o mesmo. Por exemplo para usar uma variável declarada no main, o código assembly deveria ser um "pushg" enquanto que para usar uma variável declarada no main mas usada como parâmetro da função auxiliar, o código assembly deverá ser "pushl". Dito isto, a nossa ideia foi ter um array nos diz se a variável é declarada no main ou na função, para posteriormente com manipulação em C, gerarmos o Assembly correto. As funções em C usadas serão descritas em detalhe no próximo capítulo.

3.2 Implementação

3.2.1 Analisador léxico

Para que possamos identificar os operandos da nossa linguagem, é necessário o desenvolvimento de analisador léxico que recorre a expressões regulares, de forma a filtrar os identificadores provenientes do source code, e retorna-los ao analisador sintático. Na figura 3.2 pode-se observar o analisador utilizado.

```

5  \}      {return yytext[0];}
6  \{      {return yytext[0];}
7  \[      {return yytext[0];}
8  \]      {return yytext[0];}
9  \(\     {return yytext[0];}
10 \)      {return yytext[0];}
11 \[ \n]   {}
12 \[      {return yytext[0];}
13 \[      {return yytext[0];}
14 \[      {return yytext[0];}
15 \[      {return yytext[0];}
16 "=="     {return EQUAL;}
17 "!="     {return DIF;}
18 ">"     {return GREATER;}
19 ">="    {return GREATEREQ;}
20 "<"     {return SMALLER;}
21 "<="    {return SMALEQ;}
22 "<."    {yyval.pal = strdup(yytext);return INTO;}
23 ">."    {yyval.pal = strdup(yytext);return TO;}
24 "[|]"    {return OR;}
25 "=="     {return AND;}
26 "[a-zA-Z0-9]+/+" {yyval.pal = strdup(yytext);return INC;}
27 "[a-zA-Z0-9]+/+" {yyval.pal = strdup(yytext);return DEC;}
28 "++"     {}
29 "--"     {}
30 "int"     {yyval.pal = strdup(yytext);return TYPE;}
31 "go"      {yyval.pal = strdup(yytext);return GO;}
32 "if"      {return IF;}
33 "else"    {return ELSE;}
34 "while"   {return WHILE;}
35 "for"     {return FOR;}
36 "return"  {return RETURN;}
37 "end"     {return END;}
38 "over"    {return BREAK;}
39 "#include" {return INCLUDE;}
40 \.        { return yytext[0]; }
41 \+        { return yytext[0]; }
42 \*        { return yytext[0]; }
43 \^        { return yytext[0]; }
44 \%        { return yytext[0]; }
45 \^        { return yytext[0]; }
46 "[0-9]+"  {yyval.num = atoi(yytext);return NUM;}
47 "[\na-zA-Z0-9\.\-\%]+" {yyval.pal = strdup(yytext);return WORD;}
48 " "       {}
49 " "       {}

```

Figura 3.2: Analisador Léxico

3.2.2 Analisador Sintático

Como foi referido acima, o analisador Sintático tem como função, receber os tokens provenientes do analisador léxico e numa primeira instância verificar se sequências destes constituem uma frase válida definida pela gramática correspondente que caracteriza a nossa linguagem(figura 3.3 e 3.4) e posteriormente gerar o código assembly correspondente a cada ação das respetivas produções gramaticais.

```

prog: GO decl LInstr END func

func:
  func DEF FUNC '(' seq ')' ';' decl LInstr ';'

seq:
  WORD
  seq WORD
  NUM
  seq NUM
  WORD '[' EXP ']'
  seq WORD '[' EXP ']'

decl:
  decl TYPE WORD BTO STRING
  decl TYPE WORD
  decl TYPE WORD BTO NUM
  decl TYPE WORD BTO WORD
  decl TYPE WORD '[' NUM ']'
  decl TYPE WORD '[' WORD ']'
  decl TYPE WORD '[' NUM ']' BTO ..

LInstr: Instr
  LInstr Instr
  ;

Instr: decl
  GET WORD '[' exp ']'
  WORD BTO FUNC '(' seq ')'
  PUT NUM
  PUT WORD
  PUT WORD '-' WORD
  PUT STRING WORD
  PUT WORD '[' exp ']'
  NUM TO WORD
  WORD TO WORD
  WORD BTO EXP
  INC
  DEC
  BREAK
  WORD '[' exp ']' BTO EXP
  SWAP WORD WORD
  SWAP WORD '[' exp ']' WORD '[' exp ']'

```

Figura 3.3: Analisador Sintático


```

    SWAP WORD [' exp ']' WORD [' exp ']'
    OUT value
    EXP DO : LInstr
    IF EXP ']' LInstr Else
Else:
    ELSE ']' LInstr ']'
;
EXP :
exp '+' exp
exp '-' exp
exp GREATER exp
exp SMALLER exp
exp GREATERQ exp
exp EQUAL exp
exp DIF exp
exp
exp :
exp '+' value
exp '-' value
exp '/' value
exp '%' value
value
;
value:
    '-' WORD
    WORD
    NUM
    WORD [' exp ']'

```

Figura 3.4: Analisador Sintático

3.2.3 Analisador Sintático - Ações

Devido à complexidade das ações, seria muito extenso e exaustivo fazer uma análise detalhada das mesmas, logo faremos apenas uma análise superficial, de forma ao leitor ter uma base da implementação. Se o autor assim o desejar, o código implementado no analisador sintático está presente no Anexo A, para um estudo mais profundo.

Declarações

Nas declarações, sabemos que é a primeira vez que uma variável aparece, logo apenas temos de guardar o seu índice onde ira aparecer na stack, no array, para posteriormente conseguirmos manipular variáveis e caso a variável tenha sido declarada com um valor, mete-mos esse mesmo valor na stack, caso apenas tenha sido declarada, mas sem qualquer valor, mete-mos o valor 0 na stack, de forma a ocupar o espaço na mesma. Na figura 3.5 está presente a gramática e respetivas ações usadas em declarações.

```
decl: { ${{=}}";  
    decl TYPE WORD BTO STRING {var[ind++]=${3};types[i++]=$2;asprintf(&${$, "%  
    pushes "${$, "\n", $1, $5);}  
    decl TYPE WORD {var[ind++]=${3};types[i++]=$2;asprintf(&${$, "%pushi 0\n",  
    $1);}  
    decl TYPE WORD BTO NUM {var[ind++]=${3};types[i++]=$2;asprintf(&${$, "%s  
    pushi %d\n", $1, $5);}  
    decl TYPE WORD WORD BTO WORD {var[ind++]=${3};types[i++]=$2;asprintf(&${$, "%s\n",  
    $1, $1, loadVar($, $pop);}  
    decl TYPE WORD ' ' ' NUM ' ' {var[ind++]=${3};types[i++]=$2;asprintf(&${$, "%s  
    pushn %d\n", $1, $5);ind+=ind$5);}  
    decl TYPE WORD ' ' ' WORD ' ' {var[ind++]=${3};types[i++]=$2;asprintf(&${$, "%s  
    %pushs %d\narray; %pushsg %d\npushi 1 %pushnj array2 %pushsg %d\n  
    pushi 1 %pushns store %pushi 0 %pushnj array2 %pushi 0\n", $1,  
    loadIndex("FuncParam", $pop), loadIndex($, $pop), loadIndex($, $pop),  
    loadIndex($, $pop);}  
    decl TYPE WORD ' ' ' NUM ' ' BTO ' ' ' seq ' ' {tam=$5;var[ind++]=${3};types[i+  
    +=$2;asprintf(&${$, "%s\n %s\n", $1, $5);ind+=ind$5);}
```

Figura 3.5: Declarações

Atribuições, instruções e controlo de fluxo

Nesta parte, o mais importante é usar o índice correto de cada variável (posição na stack) a manipular, de forma a que o pseudo-código Assembly consiga fluir corretamente sem que esteja a usar a variáveis erradas. Se a instrução necessitar de saber o tipo da variável para gerar o pseudo-código correto, faz-se uso da função `loadType` que é reponsavel por verificar no array onde foi guardado o tipo da variável aquando da sua declaração. É importante voltar a referir que a cada manipulação de variáveis, temos de ter em conta se são variáveis que servem de parâmetro de funções auxiliares ou se são variáveis comuns definidas no main, pois estas têm significados diferentes ao nível do assembly que temos de produzir. Dito isto é usada a função `loadVar`, que nos verifica o caracter da variável. A implementação destas funções pode ser vista na subsecção seguinte, onde explicaremos mais detalhadamente o uso destas funções. Na figura 3.6 pode ser vista a implementação da gramática e respetivas ações usadas.

```

LInstr: Instr {asprintf(&$$, "%s", $1);}
| LInstr Instr {asprintf(&$$, "%s", $1, $2);}
;

Instr: decl {yerror("Redeclaração");}
| GET WORD {tip=loadIndex($2, pop); asprintf(&$$, "%s", loadtype(tip-
pop, 0), tip);}
| GET WORD {'[' exp ']' {asprintf(&$$, "pushgp\n%snpushi %d\nadd\npadd\n
read\natoi\nstore 0\n", $4, loadIndex($2, pop));}
| WORD BTO FUNC {'[' seq ']' {asprintf(&$$, "%spusha %s\ncall\nnop\npop
%dn", $5, $3, pop);}
| PUT NUM {asprintf(&$$, "pushi %d\nwrite\n", $2);}
| PUT WORD {tip=loadIndex($2, pop); asprintf(&$$, "pushg %d\n%sn",
loadIndex($2, pop), loadtype(tip, 1));}
| PUT WORD '-' WORD {tip=loadIndex($2, pop); tip2=loadIndex($4, pop);
asprintf(&$$, "pushg %d\n%snpushg %d\n%sn", tip, loadtype(tip, 1),
tip2, loadtype(tip2, 1));}
| PUT STRING WORD {tip=loadIndex($3, pop); asprintf(&$$, "pushs \"%s\"\n
writes\npushg %dn%sn", $2, tip, loadtype(loadIndex($3, pop), 1));}
| PUT WORD {'[' exp ']' {asprintf(&$$, "pushgp\n%snpushi %d\nadd\npadd
\nload 0\nwrite\n", $4, loadIndex($2, pop));}
| PUT STRING WORD {'[' exp ']' {asprintf(&$$, "pushs %s\nwrites\npushgp
\n%snpushi %d\nadd\npadd\nload 0\nwrite\n", $2, $5, loadIndex($3,
pop));}
| NUM TO WORD {asprintf(&$$, "pushi %d\nstore %dn", $1, loadIndex($3,
pop));}
| WORD TO WORD {asprintf(&$$, "%s\nstore %dn", loadVar($1, pop),
loadIndex($3, pop));}
| WORD BTO EXP {asprintf(&$$, "%sstore %dn", $3, loadIndex($1, pop));}
| INC {asprintf(&$$, "%s\npushi 1\nadd\nstore %dn", loadVar($1, pop),
loadIndex($1, pop));}
| DEC {asprintf(&$$, "%s\npushi 1\nsub\nstore %dn", loadVar($1, pop),
loadIndex($1, pop));}
| BREAK {asprintf(&$$, "jump endDo%dn", h);}
| WORD {'[' exp ']' BTO EXP {asprintf(&$$, "pushgp\n%snpushi %d\nadd\n
padd\n%snstore 0\n", $3, loadIndex($1, pop), $6);}
| SWAP WORD WORD {tip=loadIndex($2, pop); tip2=loadIndex($3, pop);
asprintf(&$$, "pushg %d\npushg %d\nstore %d\nstore %dn", tip, tip2,
tip, tip2);}
| SWAP WORD {'[' exp ']' WORD {'[' exp ']' {asprintf(&$$, "pushgp\n%sn
pushi %d\nadd\npadd\npushgp\n%snpushi %d\nadd\npadd\nload 0\n
pushgp\n%snpushi %d\nadd\npadd\npushgp\n%snpushi %d\nadd\npadd\n
load 0\nstore 0\nstore 0\n", $4, loadIndex($2, pop), $8, loadIndex($6,
pop), $8, loadIndex($6, pop), $4, loadIndex($2, pop));}
| OUT value {asprintf(&$$, "%s\nstore %dn", $2, -t-1);}
| EXP DO ':' LInstr {asprintf(&$$, "do%dn%sjz endDo%dn jump do%dn
endDo%dn", l, $4, $1, l, l); l++;}
| IF EXP ':' LInstr ':' Else {asprintf(&$$, "%sjz more%dn%sjump more2%dn
more%dn%sn%smore2%dn", $2, h, $4, h, h, $6, h); h++;}

```

Figura 3.6: Atribuições, instruções e controlo de fluxo

Operações Aritméticas

Esta é uma parte importante pois para fazermos operações matemáticas temos de ter em conta a precedência de operadores. Dito isto, na figura 3.7 podemos visualizar a sua implementação.

```

Else: {$$=""};
| ELSE ':' LInstr ':' {asprintf(&$$, "%s\n", $3);}
;

EXP: exp '+' exp {asprintf(&$$, "%s\n%snadd\n", $1, $3);}
| exp '-' exp {asprintf(&$$, "%s\n%snsub\n", $1, $3);}
| exp GREATER exp {asprintf(&$$, "%s\n%sn>\n", $1, $3);}
| exp SMALLER exp {asprintf(&$$, "%s\n%sn<\n", $1, $3);}
| exp GREATER exp {asprintf(&$$, "%s\npushi 1\nadd\n%snpush\n", $1, $3);}
| exp EQUAL exp {asprintf(&$$, "%s\n%sn==\n", $1, $3);}
| exp DIFF exp {asprintf(&$$, "%s\n%sn-=\n", $1, $3);}
| exp {$$=$1};

exp: exp '*' value {asprintf(&$$, "%s\n%snmul\n", $1, $3);}
| exp '/' value {asprintf(&$$, "%s\n%snmdiv\n", $1, $3);}
| exp '%' value {asprintf(&$$, "%s\n%snmod\n", $1, $3);}
| value {asprintf(&$$, "%s\n", $1);}
;

value: {$$=""};
| '-' WORD {asprintf(&$$, "%s\nwrite\n", loadVar($2, pop));}
| WORD {asprintf(&$$, "%s", loadVar($1, pop));}
| NUM {asprintf(&$$, "pushi %d", $1);}
| WORD {'[' exp ']' {asprintf(&$$, "pushgp\n%snpushi %d\nadd\npadd\nload 0\n
%snloadIndex($1, pop));}
;

```

Figura 3.7: Operações Aritméticas

3.2.4 Funções implementadas em C

Como referido no subcapítulo 3.1.2 as seguintes funções foram implementadas com o intuito não só de saber a posição de cada variável na stack, mas também saber se a variável pertence a uma função auxiliar ou ao main e saber o tipo da variável, de forma a produzir o pseudo-código Assembly correto. Dito isto na figura 3.8 está representada a implementação da função em C responsável por saber se a variável pertence a função ou ao main. Na figura 3.9 está apresentada a função responsável por saber o índice de cada variável na stack. Na figura 3.10 apresenta-se a função responsável por saber o tipo da variável. É importante referir que esta função, tem função de gerar pseudo-código assembly de funções de escrita ou de leitura, ou seja se chamar-mos a função com o parametro inteiro ela sabe que tem de devolver código de leitura em Assembly, enquanto que se passarmos o parametro inteiro 1, ela sabe que tem de devolver código de escrita em Assembly.

```

char* loadVar(char *a,int x){
    int i;
    char *str;
    for(i = 0;i<x;i++){
        if(strcmp(varFunc[i],a) == 0){
            asprintf(&str,"pushl %d",-x+i);
            return str;
        }
    }
    for(i = 0;i<10;i++){
        if(strcmp(var[i],a) == 0){
            asprintf(&str,"pushg %d",i);
            return str;
        }
    }
}

```

Figura 3.8: loadVar - Saber se variável pertence a Função ou Main

```

int loadIndice(char *a,int x){
    int i;
    for(i=0;i<10;i++){
        if((strcmp(var[i],a) == 0)) return i;
    }
    for(i=0;i<x;i++){
        if((strcmp(varFunc[i],a) == 0)) return i;
    }
}

```

Figura 3.9: loadIndice - Índice da variável na stack

```

char * loadtype(int a,int x){ // x==0 R | x==1 W
    char *b = "int";
    char *c = "string";
    if(strcmp(types[a],b) == 0){
        if(x == 0){ return "read\atoi\nstoreg";}
        else{ return "writei";}}
    if(strcmp(types[a],c) == 0){
        if(x == 0){ return "read\natof\nstoreg";}
        else{ return "writes";}}
}

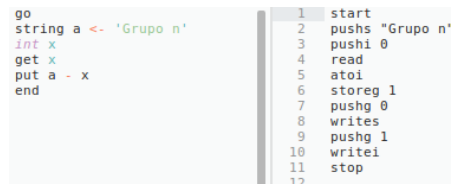
```

Figura 3.10: loadType - Tipo da variável , e retorno de pseudo-código Assembly de escrita/leitura

Capítulo 4

Codificação e Testes

Neste capítulo, mostraremos alguns exemplos feitos, por iniciativa própria, para testar a linguagem. É importante referir, que no subcapítulo 4.2, apresentaremos mais exemplos implementados, propostos pelos docentes da disciplina, sendo esses exemplos de cariz obrigatório. A figura 4.1 ilustra um simples leitura de um valor inteiro do utilizador e impressão. Na figura 4.2, está presente a implementação de um array com cada componente indexada a ser lida pelo utilizador e posteriormente é criado um novo array com os elementos pares do array inicial, recorrendo ao uso de uma função auxiliar. Na figura 4.3, está presente a implementação de uma multiplicação seguida de uma soma para testar a precedência de operadores. Na figura 4.4, está presente a implementação de uma função que recebe como parametro um inteiro, inteiro este responsável por definir o tamanho de uma matriz criada a partir de valores obtidos do utilizador, recorrendo ao uso de uma função auxiliar. Na figura 4.5, está representado, um exemplo que demonstra a versatilidade da nossa linguagem, e a rapidez com que se pode implementar algo. Um pormenor que não gostamos na linguagem C, é nomeadamente quando temos de mudar os valores de duas variáveis, ter de chamar uma variável temporária que guarda um dos valores para posteriormente se proceder a troca e isto são três instruções. Em Pickle, conseguimos o mesmo apenas com uma instrução, dando ao compilador o trabalho de fazer as trocas por nós, e dando ao programador a liberdade de escrever apenas "swap x y" no seu código. É importante referir que a "função de ordem superior" SWAP, funciona para qualquer tipo na linguagem Pickle, sejam variáveis inteiras ou arrays. Na figura 4.6 mostra-se a chamada de uma função com dois parametros de tipo inteiro, que retorna o valor de uma simples comparação entre os dois.



```
go
string a <- 'Grupo n'
int x
get x
put a - x
end
```

```
1 start
2 pushs "Grupo n"
3 pushi 0
4 read
5 atoi
6 storeg 1
7 pushg 0
8 writes
9 pushg 1
10 writei
11 stop
12
```

Figura 4.1: Ler variáveis do utilizador e imprimir

go	1	start	1	add	1	pad
int result	2	pushi 0	2	pad	2	load 0
result <- newMat(5)	3	pushi 5	3	read	3	store 0
end	4	pusha newMat	4	atoi	4	pushi 4
def newMat(i):	5	call	5	store 0	5	pushi 1
int i <- 0	6	nop	6	pushi 3	6	add
int acum <- 0	7	pop 1	7	pushi 1	7	store 4
int new2[i]	8	stop	8	add	8	jump more21
int new[i]	9		9	store 3	9	more1:
i <- y do:	10	newMat:	10	pushi 2	10	
get new[i]	11	nop	11	pushi 1	11	more21:
acum =	12	pushi 0	12	add	12	pushi 2
i++	13	pushi 0	13	store 2	13	pushi 1
0 >= 1	14	pushi 0	14	pushi 2	14	add
i < y do:	15	pushi 1	15	pushi -1	15	store 2
if new[i] % 2 = 0:	16	array1:	16	pushi -1	16	
new2[acum] <- new[i]	17	pushi 5	17	inf	17	
acum++	18	pushi 1	18	jz endDo1	18	pushi -1
0 >= 1	19	jz array2	19	endDo1:	19	inf
i < acum2 do:	20	pushi 5	20	pushi 0	20	jz endDo2
put new2[i]	21	pushi 1	21	store 2	21	endDo2:
i++	22	store 5	22	do2:pushgp	22	pushi 0
0 >= 1	23	sub	23	pushi 2	23	store 2
i < acum2 do:	24	pushi 0	24	do1:pushgp	24	pushi 0
put new2[i]	25	jump array1	25	pushi 2	25	do1:pushgp
i++	26	array2:	26	add	26	pushi 5
out 0:	27	pushi 0	27	pad	27	add
	28	pushi 1	28	load 0	28	pad
	29	array1:	29	pushi 2	29	load 0
	30	pushi 6	30	add	30	writei
	31	pushi 1	31	pushi 0	31	pushi 2
	32	sub	32	pushi 1	32	pushi 1
	33	jz array2	33	equal	33	add
	34	pushi 6	34	jz more1	34	store 2
	35	pushi 1	35	pushgp	35	pushi 2
	36	sub	36	pushi 4	36	pushi 4
	37	store 6	37	pushi 5	37	inf
	38	pushi 0	38	add	38	jz endDo3
	39	jump array1	39	add	39	endDo3:
	40	array2:	40	pushgp	40	jump do3
	41	pushi 0	41	pushi 2	41	endDo3:
	42	pushi 0	42	pushi 0	42	pushi 0
	43	do1:pushgp	43	pushi 6	43	storei -2
	44	pushi 2	44	add	44	return
	45	pushi 6	45		45	
	46		46		46	
	47		47		47	
	48		48		48	
	49		49		49	

Figura 4.2: Implementação de um array

go	1	start
int y <- 2	2	pushi 2
int x <- 3	3	pushi 3
int t	4	pushi 0
t <- x * 2 + y	5	pushi 1
put 'resultado - ' t	6	pushi 2
end	7	mul
	8	pushgp
	9	pushi 0
	10	add
	11	store 2
	12	pushs "resultado - "
	13	writei
	14	pushi 2
	15	writei
	16	stop
	17	

Figura 4.3: Precedência de operadores

go	1	start	1	jump do1
int h	2	pushi 0	2	endDo1:
h <- array(5)	3	pushi 5	3	pushi 0
put h	4	pusha array	4	store 3
end	5	call	5	do2:pushgp
	6	nop	6	pushi 3
def array(i):	7	pop 1	7	
int i <- 0	8	pushi 0	8	pushi 2
int x[i]	9	writei	9	add
i <- y do:	10	stop	10	pad
get x[i]	11		11	load 0
i++	12	array:	12	writei
0 >= 1	13	nop	13	pushi 3
i < 5 do:	14		14	add
put x[i]	15	pushi 1	15	store 3
i++	16	array1:	16	store 3
out 0:	17	pushi 2	17	pushi 3
	18	pushi 1	18	pushi 5
	19	sub	19	inf
	20	jz array2	20	jz endDo2
	21	pushi 2	21	jump do2
	22	pushi 1	22	endDo2:
	23	sub	23	pushi 0
	24	store 2	24	storei -2
	25	pushi 0	25	return
	26	jump array1	26	
	27	array2:	27	
	28	pushi 0	28	
	29	pushi 0	29	
	30	do1:pushgp	30	
	31	pushi 3	31	
	32	add	32	
	33	pushi 2	33	
	34	add	34	
	35	pad	35	
	36	read	36	
	37	atoi	37	
	38	store 0	38	
	39	pushi 3	39	
	40	pushi 1	40	
	41	add	41	
	42	store 3	42	
	43	pushi 3	43	
	44	pushi -1	44	
	45	inf	45	
	46	jz endDo1	46	
	47		47	
	48		48	
	49		49	

Figura 4.4: Função sobre arrays

go	1	start	1	add
int i <- 0	2	pushi 0	2	padd
int x[i]	3	pushi 5	3	pushgp
i <- 5; do:	4	do1:pushgp	4	pushgp 0
get x[i]	5	pushg 0	5	pushi 1
i++	6	pushi 1	6	add
0 -> i	7	add	7	padd
swap x[i] x[i+1]	8	padd	8	load 0
	9	read	9	store 0
	10	atoi	10	store 0
0->1	11	store 0	11	pushi 0
	12	pushg 0	12	store 0
i<5 do:	13	pushi 1	13	do2:pushgp
put x[i]	14	add	14	pushg 0
i++	15	store 0	15	pushi 1
end	16	pushg 0	16	add
	17	pushi 5	17	padd
	18	jz end01	18	load 0
	19	jump do1	19	writei
	20	end01:	20	pushg 0
	21	pushi 0	21	pushi 1
	22	store 0	22	add
	23	pushgp	23	store 0
	24	pushg 0	24	pushi 5
	25	pushi 1	25	inf
	26	add	26	jz end02
	27	padd	27	jump do2
	28	load 0	28	end02:
	29	store 0	29	stop
	30	pushi 1		
	31	add		
	32	padd		
	33	pushgp		
	34	pushg 0		
	35	pushi 1		
	36	add		
	37	pushi 1		
	38	add		
	39	padd		
	40	load 0		
	41	pushgp		
	42	pushg 0		
	43	pushi 1		
	44	add		
	45	pushi 1		
	46	add		
	47	pushi 1		
	48	add		
	49	pushi 1		

Figura 4.5: Swap com elementos de matrizes

go	1	start
int h	2	pushi 0
h <- funcao(5 5)	3	pushi 5
put h	4	pushi 5
end	5	pusha funcao
	6	call
def funcao(x y):	7	nop
if y ~ x : out 1:	8	pop 2
ahoy:out 0:	9	pushg 0
	10	writei
	11	stop
	12	funcao:
	13	nop
	14	pushi -1
	15	pushi -2
	16	sub
	17	jz more1
	18	pushi 1
	19	storei -3
	20	add
	21	jump more21
	22	more1:
	23	pushi 0
	24	storei -3
	25	:
	26	more21:
	27	return

Figura 4.6: Função com comparações que retorna valor

4.1 Alternativas, Decisões e Problemas de Implementação

Uma das ideias propostas pelo grupo de trabalho, para a resolução da alinea D, a construção do grafo de relações, foi a partir de uma página HTML, o utilizador escrever um autor à sua escolha numa caixa de diálogo, e de seguida seria reenviado para uma página que conteria o grafo dos autores com quem trabalhou. Após enumeras tentativas de implementação sem sucesso, achamos melhor gerar um grafo de relações de todos os autores do ficheiro. Esta seria uma alternativa mais atrativa para o utilizador.

Um dos problemas recorrentes, foram as diversas representações usadas para o mesmo autor ao longo do ficheiro, dos quais todos, excepto um, foram ultrapassadas. O caso para o qual não nos foi possível arranjar uma solução, foi o caso em que o primeiro ou segundo nome do Autor é representado pela sua primeira letra seguida de um ponto. Ex: Pedro R. Henriques. Isto acaba por ser um erro grotesco, pois para a implementação de um grafo de relações, aparece diferentes nodos que representam o mesmo autor.

Outro dos problemas de implementação surgiu devido às enumeras inconsistências no ficheiro original, o que nos obrigou a redrobar as atenções nas expressões regulares utilizadas de forma a que estes detalhes não prejudicassem a performance do resto do trabalho.

4.2 Testes realizados e Resultados

Nesta secção, mostraremos a implementação de exemplos mais alguns exemplos e respetivos resultados.

- ler 4 números e dizer se podem ser os lados de um quadrado - figura 4.7
- ler um inteiro N, depois ler N números e escrever o menor deles - figura 4.8
- ler N (constante do programa) números e calcular e imprimir o seu produto - figura 4.9
- contar e imprimir os números impares de uma sequência de números naturais - figura 4.10
- ler e armazenar os elementos de um vetor de comprimento N; imprimir os valores por ordem decrescente após fazer a ordenação do array por trocas diretas - figura 4.11
- ler e armazenar N números num array; imprimir os valores por ordem inversa - figura 4.12

```
go
int i <- 1
int x
int y
int result <- 0
get x
i <- 4
do:
  get y
  if x < y: 1->result over:
  i++
put result
end

1 start
2 pushl 1
3 pushl 0
4 pushl 0
5 pushl 0
6 read
7 atoi
8 storeg 1
9 do:read
10 atoi
11 storeg 2
12 pushg 2
13 pushg 1
14 sub
15 jz more
16 pushl 1
17 storeg 3
18 jmp endDo
19 jump more2
20 more:
21
22 more2:
23 pushg 0
24 pushl 1
25 add
26 storeg 0
27 pushl 4
28 pushg 0
29
30 sup
31
32 jz endDo
33 jump do
34 endDo:pushg 3
35 writel
36 stop
---
```

Figura 4.7: Testa lados de um quadrado

```
go
int N
int x
int menor
get N
get x
x <- menor
N <- 0
do:
  get x
  if x < menor:
    x <- menor
  N++
put "o menor numero e: " . menor
end

1 start
2 pushl 0
3 pushl 0
4 pushl 0
5 read
6 atoi
7 storeg 0
8 read
9 atoi
10 storeg 1
11 pushg 1
12 storeg 2
13 pushg 0
14 pushl 1
15 sub
16 jz endDo1
17 do:read
18 atoi
19 storeg 1
20 pushg 1
21 pushg 2
22
23 inf jz more1
24 pushg 1
25 storeg 2
26 jump more21
27 more1:
28
29 more21:
30 pushg 0
31 pushl 1
32 sub
33 storeg 0
34 pushg 0
35 pushl 0
36
37 sup jz endDo1
38
39 jump dol
40 endDo1
41
42 pushs "o menor numero e: "
43 writel
44 pushg 2
45 writel
46 stop
---
```

Figura 4.8: Menor numero de uma lista de numeros

```

go
int i <- 5
int num
int prod <- 1
do:
  get num
  prod <- prod * num
  x--
put prod
end

1 start
2 pushi 5
3 pushi 0
4 pushi 1
5 dol:read
6 atoi
7 storeg 1
8 pushg 2
9
10 pushg lmul
11 storeg 2
12 pushg 0
13 pushi 1
14 sub
15 storeg 0
16 pushg 0
17
18 pushi 1
19
20 sup
21 jz endDo1
22 jump do1
23 endDo1:
24 pushg 2
25 writel
26 stop

```

Figura 4.9: Produtório de sequência de numeros

```

go
int i <- 5
int num
int contagem <- 0
i > 0 do:
  get num
  if num % 2 = 0:
    contagem++
    put "par"
  num:
  i--
put "numeros impares - " contagem
end

1 start
2 pushi 5
3 pushi 0
4 pushi 0
5 dol:read
6 atoi
7 storeg 1
8 pushg 1
9 storeg 2
10 add
11 pushi 0
12 sub
13 jz more1
14 pushg 2
15 pushi 1
16 add
17 storeg 2
18 pushg "par - "
19 writes
20 pushg 1
21 writel
22 jump more21
23 more1:
24
25 more21:
26 pushg 0
27 pushi 1
28 sub
29 storeg 0
30 pushg 0
31 pushi 0
32 sup
33 jz endDo1
34 jump do1
35 endDo1:
36 pushg "numeros impares - "
37 writes
38 pushg 2
39 writel
40 stop

```

Figura 4.10: Contagem e impressão de numeros impares de sequência de numeros

```

go
int i <- 0
int c <- 0
int d <- 0
int t
int x[5]
i < 5 do:
  get x[i]
  i++
0 -> i
c < 4 do:
  d < 4 - c do:
    if x[d] > x[d+1]:
      swap x[d] x[d+1]
  d++
c++
d <- 0:
  put x[i]
  i++
end

1 start
2 pushi 0
3 pushi 0
4 pushi 0
5 pushi 0
6 pushn 5
7 dol:pushgp
8 pushg 0
9
10 pushi 4
11 add
12 padd
13 read
14 atoi
15 store 0
16 pushg 0
17 pushi 1
18 add
19 storeg 0
20 pushg 0
21
22 pushi 5
23
24 inf
25 jz endDo1
26 jump do1
27 endDo1:
28 pushi 0
29 storeg 0
30 do3:do2:pushgp
31 pushg 2
32
33 pushi 4
34 add
35 padd
36 load 0
37
38 pushgp
39 pushg 2
40
41 pushi 1
42 add
43
44 pushi 4
45 add
46 padd
47 load 0
48
49 sup
50 jz more1
51 pushgp
52 pushg 2
53
54 pushi 4
55 add
56 padd
57 pushgp
58 pushg 2
59
60 pushi 1
61 add
62
63 pushi 4
64 add
65 padd
66 load 0
67 pushgp
68 pushg 2
69
70 pushi 1
71 add
72
73 pushi 4

1 add
2 padd
3 pushgp
4 pushg 2
5
6 pushi 4
7 add
8 padd
9 load 0
10 store 0
11 store 0
12 jump more21
13 more1:
14
15 more21:
16 pushg 2
17 pushi 1
18 add
19 storeg 2
20 pushg 2
21
22 pushi 4
23
24 pushg 1
25 sub
26
27 inf
28 jz endDo2
29 jump do2
30 endDo2:
31 pushg 1
32 pushi 1
33 add
34 storeg 1
35 pushi 0
36 storeg 2
37 pushg 1
38
39 pushi 4
40
41 inf
42 jz endDo3
43 jump do3
44 endDo3:
45 do4:pushgp
46 pushg 0
47
48 pushi 4
49 add
50 padd
51 load 0
52 writel
53 pushg 0
54 pushi 1
55 add
56 storeg 0
57 pushg 0
58
59 pushi 5
60
61 inf
62 jz endDo4
63 jump do4
64 endDo4:
65 stop

```

Figura 4.11: Ordenação de um array por ordem crescente

go	1	start
int i <- 0	2	pushi 0
int x[5]	3	pushn 5
i < 5 do:	4	dol:pushgp
get x[i]	5	pushg 0
i++:	6	
4 -> i	7	pushi 1
i >= 0 do:	8	add
put x[i]	9	padd
i--:	10	read
end	11	atoi
	12	store 0
	13	pushg 0
	14	pushi 1
	15	add
	16	storeg 0
	17	pushg 0
	18	
	19	pushi 5
	20	inf
	21	jz endDo1
	22	jump dol
	23	endDo1:
	24	pushi 4
	25	storeg 0
	26	do2:pushgp
	27	pushg 0
	28	
	29	pushi 1
	30	add
	31	padd
	32	load 0
	33	writel
	34	pushg 0
	35	pushi 1
	36	sub
	37	storeg 0
	38	pushg 0
	39	
	40	pushi 1
	41	add
	42	pushi 0
	43	
	44	sup
	45	jz endDo2
	46	jump do2
	47	endDo2:
	48	stop
	49	
	50	

Figura 4.12: Ler elementos de um array e imprimir array por ordem inversa

Capítulo 5

Conclusão

Ao longo desta jornada, que nos levou à realização do trabalho, observamos um aumento significativo nas nossas competências relativamente à utilização de gramáticas independentes de contexto(GIC) e percebemos o papel que estas representam não só no processamento de linguagens e compilação destas mas também na resolução de diversos problemas do quotidiano virtual. De facto, é de extrema relevância, referir que ao longo desta disciplina evoluímos a passos largos no sentido de nos tornar-mos mais capazes em todo o paradigma computacional. Apesar de não termos conseguido implementar tudo o que tinha-mos em mente, achamos que os resultados obtidos são bastante satisfatórios. No futuro, logo que seja possível, seria do nosso agrado implementar algumas das alternativas propostas, que ao longo do trabalho não nos foi possível investir o tempo necessário para a sua execução.

Apêndice A

Código do Programa

```
%{
#include <stdio.h>
#include <strings.h>
int yyerror(char *s);
int yylex();
int loadIndice(char *a,int x);
char * loadtype(int a , int x);
char* loadVar(char *a,int x);
char *var[100];
char *varFunc[100];
char *types[10];
int ind=0;
int i=0;
int pop=0;
int tip;
int tip2;
int tam;
int t=0;
int j=0;
int l=1;
int h=1;
int error=1;

%}

%union {
char *pal;
int num;
}

%token <pal> TYPE WORD STRING INC DEC FUNC
%token <num> NUM
%token GO END SWAP TO BTO GET PUT DO SMALLER GREATER GREATEQ SMALEQ EQUAL DIF IF ELSE DEF OUT BREAK
%type <pal> prog
%type <pal> decl
%type <pal> Instr
%type <pal> LInstr
%type <pal> func
```

```

%type <pal> exp
%type <pal> EXP
%type <pal> value
%type <pal> Else
%type <pal> seq

%%

prog: GO decl LInstr END func {printf("start\n%s\stop\n\n%s", $2, $3, $5);}
;

func: {$$="";}
    | func DEF FUNC '(' seq ')' ':' decl LInstr ':' {asprintf(&$$, "%s%s:\nnop\n%s\n%s\nreturn", $1, $3, $8, $9);}
;

seq: WORD {varFunc[t++]=$1;}
    | seq WORD {varFunc[t++]=$2;}
    | NUM {pop++;var[ind++]="FuncParam";asprintf(&$$, "pushi %d\n", $1);j++;}
    | seq NUM {pop++;var[ind++]="FuncParam";asprintf(&$$, "%spushi %d\n", $1, $2);j++;}
    | WORD '[' EXP ']' {asprintf(&$$, "pushgp\n%s\npushi %d\nadd\npadd\nload 0", $3, loadIndice($1, pop));}
    | seq WORD '[' EXP ']' {asprintf(&$$, "%spushgp\n%s\npushi %d\nadd\npadd\nload 0", $1, $4, loadIndice($2, pop))}
;

decl: {$$="";}
    | decl TYPE WORD BTO STRING {var[ind++]=$3;types[i++]=$2;asprintf(&$$, "%spushs \"%s\"\\n", $1, $5);}
    | decl TYPE WORD {var[ind++]=$3;types[i++]=$2;asprintf(&$$, "%spushi 0\\n", $1);}
    | decl TYPE WORD BTO NUM {var[ind++]=$3;types[i++]=$2;asprintf(&$$, "%spushi %d\\n", $1, $5);}
    | decl TYPE WORD BTO WORD {var[ind++]=$3;types[i++]=$2;asprintf(&$$, "%s%s\\n", $1, loadVar($5, pop));}
    | decl TYPE WORD '[' NUM ']' {var[ind]= $3;types[i++]=$2;asprintf(&$$, "%spushn %d\\n", $1, $5);ind=ind+$5;}
    | decl TYPE WORD '[' WORD ']' {var[ind++]=$3;types[i++]=$2;asprintf(&$$, "%s\npushg %d\\narray1:\\n
pushg %d\\npushi 1\\nsub\\njz array2\\npushg %d\\n
pushi 1\\nsub\\nstoreg %d\\npushi 0\\njump array1\\narray2:\\npushi 0\\n", $1, loadIndice("FuncParam", pop),
loadIndice($3, pop), loadIndice($3, pop), loadIndice($3, pop));}
    | decl TYPE WORD '[' NUM ']' BTO '{' seq '}' {tam=$5;var[ind]=$3;types[i++]=$2;
asprintf(&$$, "%s\\n %s\\n", $1, $9);ind=ind+j;}
;

LInstr: Instr {asprintf(&$$, "%s", $1);}
    | LInstr Instr {asprintf(&$$, "%s%s", $1, $2);}
;

Instr: decl {yyerror("Redeclaração");}
    | GET WORD {tip=loadIndice($2, pop);asprintf(&$$, "%s %d\\n", loadtype(tip-pop, 0), tip);}
    | GET WORD '[' exp ']' {asprintf(&$$, "pushgp\n%s\npushi %d\nadd\npadd\nread\natoi\\nstore 0\\n",
$4, loadIndice($2, pop));}
    | WORD BTO FUNC '(' seq ')' {asprintf(&$$, "%spusha %s\\ncall\\nnop\\npop %d\\n", $5, $3, pop);}
    | PUT NUM {asprintf(&$$, "pushi %d\\nwritei\\n", $2);}
    | PUT WORD {tip=loadIndice($2, pop);asprintf(&$$, "pushg %d\\n%s\\n", loadIndice($2, pop), loadtype(tip, 1));}
    | PUT WORD '-' WORD {tip=loadIndice($2, pop);tip2=loadIndice($4, pop);asprintf(&$$, "pushg %d\\n%s\\n
pushg %d\\n%s\\n", tip, loadtype(tip, 1), tip2, loadtype(tip2, 1));}
    | PUT STRING WORD {tip=loadIndice($3, pop);asprintf(&$$, "pushs \"%s\"\\nwrites\\npushg %d\\n%s\\n",
$2, tip, loadtype(loadIndice($3, pop), 1));}
    | PUT WORD '[' exp ']' {asprintf(&$$, "pushgp\n%s\npushi %d\nadd\npadd\nload 0\\nwritei\\n", $4,

```

```

loadIndice($2,pop));}
| PUT STRING WORD '[' exp ']' {asprintf(&$$,"pushs %s\nwrites\npushgp\n%s\npushi %d\nadd\n
padd\nload 0\nwritei\n",$2,$5,loadIndice($3,pop));}
| NUM TO WORD {asprintf(&$$,"pushi %d\nstoreg %d\n",$1,loadIndice($3,pop));}
| WORD TO WORD {asprintf(&$$,"%s\nstoreg %d\n",loadVar($1,pop),loadIndice($3,pop));}
| WORD BTO EXP {asprintf(&$$,"%sstoreg %d\n",$3,loadIndice($1,pop));}
| INC {asprintf(&$$,"%s\npushi 1\nadd\nstoreg %d\n",loadVar($1,pop),loadIndice($1,pop));}
| DEC {asprintf(&$$,"%s\npushi 1\nsub\nstoreg %d\n",loadVar($1,pop),loadIndice($1,pop));}
| BREAK {asprintf(&$$,"jump endDo%d\n",h);}
| WORD '[' exp ']' BTO EXP {asprintf(&$$,"pushgp\n%s\npushi %d\nadd\npadd\nload 0\n%s\nstore 0\n",
$3,loadIndice($1,pop),$6);}
| SWAP WORD WORD {tip=loadIndice($2,pop);tip2=loadIndice($3,pop);asprintf(&$$,"pushg %d\n
pushg %d\nstoreg %d\nstoreg %d\n",tip,tip2,tip,tip2);}
| SWAP WORD '[' exp ']' WORD '[' exp ']' {asprintf(&$$,"pushgp\n%s\npushi %d\nadd\npadd\n
pushgp\n%s\npushi %d\nadd\npadd\nload 0\npushgp\n%s\npushi %d\nadd\n
padd\npushgp\n%s\npushi %d\nadd\npadd\nload 0\nstore 0\nstore 0\n",$4,loadIndice($2,pop),
$8,loadIndice($6,pop),$8,loadIndice($6,pop),$4,loadIndice($2,pop));}
| OUT value {asprintf(&$$,"%s\nstorel %d\n",$2,-t-1);}
| EXP DO ':' LInstr ':' {asprintf(&$$,"do%d:%s%sjz endDo%d\n jump do%d\nendDo%d:\n",1,$4,$1,1,1,1);1+
| IF EXP ':' LInstr ':' Else {asprintf(&$$,"%s%sjz more%d\n%sjump more2%d\nmore%d:\n%s\nmore2%d:\n",$2,
h,$4,h,h,$6,h);h++;}

Else: {$$="";}
| ELSE ':' LInstr ':' {asprintf(&$$,"%s\n:",$3);}
;
EXP : exp '+' exp {asprintf(&$$,"%s\n%s\nadd\n",$1,$3);}
| exp '-' exp {asprintf(&$$,"%s\n%s\nsub\n",$1,$3);}
| exp GREATER exp {asprintf(&$$,"%s\n%s\nsup\n",$1,$3);}
| exp SMALLER exp {asprintf(&$$,"%s\n%s\ninf\n",$1,$3);}
| exp GREATEREQ exp {asprintf(&$$,"%s\npushi 1\nadd\n%s\nsup\n",$1,$3);}
| exp EQUAL exp {asprintf(&$$,"%s\n%s\nnequal\n",$1,$3);}
| exp DIF exp {asprintf(&$$,"%s\n%s\nsub\n",$1,$3);}
| exp {$$=$1;}

exp : exp '+' value {asprintf(&$$,"%s\n%s\nadd\n",$1,$3);}
| exp '-' value {asprintf(&$$,"%s\n%s\nsub\n",$1,$3);}
| exp '*' value {asprintf(&$$,"%s\n%s\nmul\n",$1,$3);}
| exp '/' value {asprintf(&$$,"%s\n%s\ndiv\n",$1,$3);}
| exp '%' value {asprintf(&$$,"%s\n%s\nmod\n",$1,$3);}
| value {asprintf(&$$,"%s\n",$1);}
;

value: {$$="";}
| '-' WORD {asprintf(&$$,"%s\nwritei\n",loadVar($2,pop));}
| WORD {asprintf(&$$,"%s",loadVar($1,pop));}
| NUM {asprintf(&$$,"pushi %d",$1);}
| WORD '[' exp ']' {asprintf(&$$,"pushgp\n%s\npushi %d\nadd\npadd\nload 0",$3,loadIndice($1,pop));}
;

%%

#include "lex.yy.c"

```

```

int yyerror(char *s)
{
    fprintf(stderr, "ERRO SINTATICO: %s \n", s);
    exit(-1);
}

int strcicmp(char const *a, char const *b)
{
    int d;
    for (; a++, b++) {
        d = tolower(*a) - tolower(*b);
        if (d != 0 || !*a)
            return d;
    }
}

char* loadVar(char *a,int x){
    int i;
    char *str;
    for(i = 0;i<x;i++){
        if(strcicmp(varFunc[i],a) == 0){
            asprintf(&str,"pushl %d",-x+i);
            return str;
        }
    }
    for(i = 0;i<10;i++){
        if(strcicmp(var[i],a) == 0){
            asprintf(&str,"pushg %d",i);
            return str;
        }
    }
}

int loadIndice(char *a,int x){
    int i;
    for(i=0;i<10;i++){
        if((strcicmp(var[i],a) == 0)) return i;
    }
    for(i=0;i<x;i++){
        if((strcicmp(varFunc[i],a) == 0)) return i;
    }
}

char * loadtype(int a,int x){ // x==0 R | x==1 W
    char *b = "int";
    char *c = "string";
    if(strcicmp(types[a],b) == 0){
        if(x == 0){ return "read\natoi\nstoreg";}
        else{ return "writei";}}
    if(strcicmp(types[a],c) == 0){
        if(x == 0){ return "read\natof\nstoreg";}
        else{ return "writes";}}
}

```

```
}}
```

```
int main()
{
    yyparse();
    return(0);
}
```