

Processamento de Linguagens  
**Trabalho Prático 2**  
Relatório de Desenvolvimento

Diogo Machado  
(A75399)

Lisandra Silva  
(A73559)

Rui Leite  
(A75551)

12 de Junho de 2017

## Resumo

Este trabalho consiste no desenvolvimento de um compilador para uma máquina de *stack* virtual. O compilador criado recebe como *input* um ficheiro de texto correspondente a um programa escrito numa linguagem definida e tem como *output* um ficheiro de texto com instruções que podem ser executadas numa máquina de *stack* virtual.

A linguagem usada nos programas de input é uma linguagem semelhante à linguagem C. Por sua vez, o código gerado também se assemelha a código *assembly*. Desta forma, o compilador desenvolvido pretende simular, ainda que de forma simplificada, o trabalho realizado por um compilador gcc.

# Conteúdo

<b>Introdução</b>	<b>3</b>
<b>1 Implementação da Solução</b>	<b>4</b>
1.1 Descrição do Problema . . . . .	4
1.2 Especificação da linguagem . . . . .	5
1.2.1 Ortografia . . . . .	5
1.3 Sintaxe . . . . .	7
1.4 Semântica estática . . . . .	9
1.4.1 Árvore de identificadores . . . . .	9
<b>2 Geração de código</b>	<b>11</b>
2.1 Início do Programa . . . . .	11
2.2 Declaração de variáveis . . . . .	11
2.3 Declaração de funções . . . . .	11
2.4 Retorno . . . . .	12
2.5 Chamada de funções . . . . .	12
2.6 Atribuições . . . . .	12
2.7 Estrutura <b>if-then-else</b> . . . . .	13
2.8 Estrutura <b>while</b> . . . . .	13
2.9 Leitura do <i>standard input</i> . . . . .	14
2.10 Impressão no ecrã . . . . .	14
2.11 Expressões e condições . . . . .	14
<b>3 Resultados e testes</b>	<b>15</b>
3.1 Exemplo 1 - quadrado . . . . .	15
3.2 Exemplo 2 - menor . . . . .	17
3.3 Exemplo 3 - produtório . . . . .	19
3.4 Exemplo 4 - Números ímpares . . . . .	20
3.5 Exemplo 5 - Ordenação do array . . . . .	22
3.6 Exemplo 6 - Ordem Inversa . . . . .	25
3.7 Exemplo 7 - Multiplicação de vetores e matrizes . . . . .	28

3.8 Exemplo 8 - Calcular o quadrado de um numero . . . . .	31
<b>Conclusão e aspetos a melhorar</b>	<b>33</b>
<b>A Código</b>	<b>34</b>

# Introdução

Este trabalho prático tem como principal objetivo o desenvolvimento de um compilador que permita gerar código para uma máquina de *stack* virtual (VM). Este processo consiste na criação de um processador de linguagem segundo o método da tradução dirigida pela sintaxe, suportado por uma gramática tradutora. Neste relatório são apresentados os passos seguidos até a implementação da solução, onde é essencial a utilização do FLEX para fazer o reconhecimento de símbolos terminais no ficheiro de entrada e do YACC para gerar o compilador baseado na gramática tradutora.

## Estrutura do Relatório

O presente relatório encontra-se dividido em três capítulos. No primeiro é especificada a linguagem escolhida, a sua gramática (sintaxe) e toda a semântica associada, nomeadamente, as regras definidas inicialmente. No segundo capítulo é discutida a forma como o compilador gera o código *Assembly*, para cada instrução. Por fim, no terceiro capítulo são apresentados alguns exemplos de programas escritos na linguagem definida, capaz de ser lida pelo compilador, acompanhados do respetivo código *Assembly* gerado e do resultado obtido na máquina virtual.

# Capítulo 1

## Implementação da Solução

Neste capítulo são apresentadas todas as tomadas de decisão e o processo que foi seguido para a implementação de uma solução que satisfaz os objetivos.

### 1.1 Descrição do Problema

Neste trabalho prático pretende-se implementar uma solução que permita a geração de código *Assembly* a partir de código escrito numa linguagem bem definida pelo grupo. Desta forma, começou-se então por definir uma linguagem que, no seu todo, permite as seguintes funcionalidades:

- declarar e manusear variáveis atômicas do tipo inteiro, com os quais se podem realizar operações aritméticas, relacionais e lógicas;
- declarar e manusear variáveis estruturadas do tipo *array* de inteiros, de 1 ou 2 dimensões, em relação aos quais é apenas permitida a operação de indexação (índice inteiro);
- efetuar instruções algorítmicas básicas como 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 com/sem parâmetros mas que possa retornar um resultado atómico.

Pretende-se que as declarações das variáveis sejam feitas antes das funções, no caso de se tratar de variáveis globais, ou no início do código, i.e., antes das instruções, e que não sejam permitidas re-declarações, nem utilizações sem uma declaração prévia. Se não houver qualquer atribuição na declaração da variável, o seu valor é posto a zero.

Depois de bem definida a linguagem passou-se à implementação de um compilador capaz de gerar código *Assembly* para a VM, com o recurso ao Gerador YACC/FLEX.

## 1.2 Especificação da linguagem

Tal como solicitado no enunciado do trabalho prático, foi necessário definir uma linguagem de programação imperativa simples. Optou-se por criar um linguagem semelhante à linguagem C dado tratar-se de uma linguagem com a qual estamos familiarizados. Nesta secção será apresentada a ortografia e sintaxe da linguagem bem como a gramática correspondente.

### 1.2.1 Ortografia

De seguida são apresentados exemplos de programas escritos na linguagem criada. Tratam-se de exemplos classificativos, criados com o objetivo de ilustrar como se deve escrever um programa na nossa linguagem.

#### Estrutura do programa

O programa é constituído essencialmente por 3 partes: Variáveis Globais, Funções e Função Principal (*Main*). Destas 3 componentes a única que é obrigatória é a Main, embora possa não conter qualquer declaração de variáveis ou instrução. Esta deve ser declarada recorrendo à palavra reservada **MAIN** conforme se ilustra de seguida:

```
MAIN{  
    // Declarações de Variáveis  
    // Instruções  
}
```

#### Declarações de variáveis

A declaração de variáveis deve ser feita recorrendo à palavra reservada **INT**. É possível declarar variáveis inteiras, *arrays* e matrizes (*arrays* bidimensionais). Para as variáveis inteiras é ainda possível atribuir um valor na inicialização.

Na mesma linha é possível declarar várias variáveis desde que estas estejam separadas por um vírgula (','). A todas as variáveis não inicializadas é atribuído, por defeito, o valor 0 (zero).

```
INT x;  
INT y, z;  
INT a = 0, b = 2;  
INT v[5], m[2][3];
```

#### Atribuição

A atribuição faz-se, tal como em C, recorrendo ao sinal de '='. No exemplo abaixo é declarada uma variável **a** e é-lhe atribuído o valor 2.

```
INT a;  
a = 3;
```

## Estrutura if-then-else

Para permitir instruções de controlo do fluxo de execução condicional foi considerada a estrutura de controlo **if-then-else**. Esta escreve-se da mesma forma que em C, sendo permitidos **if-then-else** sem a cláusula **else**. O seguinte excerto retrata um exemplo de utilização desta estrutura:

```
INT a, b, c;
a = 1;
IF (a == 1) {
    b = 2;
} ELSE {
    b = 3;
}
IF (b == 2){
    c = 1;
}
```

## Estrutura while

Para além de mecanismos de controlo de fluxo do tipo condicional foram considerados mecanismos de controlo de fluxo de execução cíclicos. Este tipo de mecanismo foi conseguido graças ao uso da estrutura **while**, semelhante à da linguagem C.

```
INT i = 0;
INT a[5];
WHILE (i < 5){
    a[i] = i * 2;
    i = i + 1;
}
```

## Funções

A linguagem especificada permite a implementação e invocação de funções. As funções podem receber 1 argumento e devem ter, obrigatoriamente, uma instrução de retorno no seu final, i.e., devem fazer o **return** de um resultado atómico. Estas devem ser declaradas com a palavra reservada **FUNCAO**.

```
FUNCAO exemplo() {
    RETURN 1;
}

FUNCAO exemplo2(INT x) {
    RETURN x+1;
}
```



## Impressão e Leitura

A linguagem definida permite a impressão no ecrã de uma *string*, valor ou expressão e ainda a leitura de um valor do *standard input* para uma variável declarada. A impressão usa a palavra reservada `PRINT` e a leitura a palavra reservada `READ`.

O exemplo seguinte consiste na impressão da *string* "Insira um número:", leitura do número lido para a variável `x` e a impressão do valor lido incrementado:

```
INT x;
PRINT "Insira um número";
READ x;
PRINT x + 1;
```

## 1.3 Sintaxe

A sintaxe da linguagem é capturada pela gramática que se apresenta de seguida:

```
Programa : Decls Instrucoes Funcoes MainFunc
          ;

Funcoes  : Funcoes Funcao
          |
          ;

Funcao   : FUNCAO DESIGNACAO '(' ')' '{' Decls Instrucoes Return '}'
          | FUNCAO DESIGNACAO '('('DESIGNACAO')' '{' Decls Instrucoes Return '}'
          ;

Return   : RETURN Expr ';'
          ;

MainFunc : MAIN '{' Decls Instrucoes '}'
          ;

Decls    : Decls Decl
          |
          ;

Decl     : INT DeclsVar ';'
          ;

DeclsVar : DeclVar
          | DeclVar ',' DeclsVar
          ;
```

```

DeclVar : DESIGNACAO Init
        | DESIGNACAO '[' VALOR ']'
        | DESIGNACAO '[' VALOR ']' '[' VALOR ']'
        ;

Init : '=' Expr
      |
      ;

Instrucoes : Instrucoes Instrucao
            |
            ;

Instrucao : Var '=' Expr ';'
           | IF '(' Cond ')' '{' Instrucoes '}' ELSE '{' Instrucoes '}'
           | WHILE '(' Cond ')' '{' Instrucoes '}' }
           | PRINT Expr ';'
           | PRINT STRING ';'
           | READ Var ';'
           | DESIGNACAO '(' ')' ';'
           | DESIGNACAO '(' Expr ')' ';'
           ;

Cond : Expr '>' '=' Expr
      | Expr '<' '=' Expr
      | Expr '=' '=' Expr
      | Expr '!' '=' Expr
      | Expr '<' Expr
      | Expr '>' Expr
      | Expr
      ;

Expr : Termo
      | Expr '+' Termo
      | Expr '-' Termo
      | Expr '|' Termo
      ;

Termo : Fator
       | Termo '*' Fator
       | Termo '/' Fator
       | Termo '%' Fator
       | Termo '&' Fator
       ;

```

```

Fator : VALOR
      | Var
      | '(' Cond ')'
      | DESIGNACAO '(' ')'
      | DESIGNACAO '(' Expr ')'

Var : DESIGNACAO
    | DESIGNACAO '[' Expr ']'
    | DESIGNACAO '[' Expr ']' '[' Expr ']'
    ;

```

## 1.4 Semântica estática

A semântica estática estabelece as restrições que um programa sintaticamente correto deve satisfazer para que seja possível estabelecer um significado para ele. Desta forma, na verificação da semântica estática, teve-se o cuidado de garantir que:

- O uso de uma variável só pode ser validado se a variável tiver sido declarada;
- O uso de um função também só poderá ser autorizado se esta tiver sido declarada e implementada previamente;
- O uso de um função é validado se o número de argumentos estiver correto;
- Não é validada a declaração de uma variável global se existir outra global com a mesma identificação;
- Não é validada a declaração de uma variável local a uma função se existir outra local nessa mesma função com a mesma identificação;
- Caso seja declarada uma variável local quando exista outra global com a mesma identificação, a declaração que prevalece é a da variável local;
- As dimensões dos *arrays* devem ser números inteiros maiores do que 1.

### 1.4.1 Árvore de identificadores

Para a implementação das regras de verificação da semântica estática enunciadas anteriormente, foram declaradas três árvores de identificadores, **variaveis**, **varGlobais** e **funcoes**, com o tipo de dados **Gtree** da biblioteca **GLIB**. A árvore **varGlobais** tem informação de todas as variáveis globais declaradas ao longo do código e a árvore **variaveis** contém todas as variáveis locais, na função atual. Para o efeito foi criada uma estrutura de dados **Nodo** cujo conteúdo pode ser visto de seguida:

```
typedef struct nodo {
    int indice;
    int colunas;
} *Nodo;
```

Esta estrutura contém o índice da variável, que é um valor inteiro, iniciado a zero (0) e incrementado sempre que ocorre uma declaração válida de uma variável. Este valor é útil, por exemplo, quando se pretende fazer **push** da variável para a *stack* e nas atribuições, quando se pretende fazer **storeg**. O valor de **colunas** serve para identificar se a variável é um *array* bidimensional e, caso seja, quantas “colunas” possui. Se não for um *array* bidimensional, esta variável tomará o valor 0.

Para identificar se a declaração é local a uma função ou se é global, fez-se uso de uma *flag local*, iniciada a 0, que toma o valor 1 quando é encontrada a declaração de uma função ou da **MAIN** do programa e volta ao valor 0 sempre que se encontra o fim de uma função.

Sempre que é encontrada a declaração de uma variável num contexto global, é testada a existência dessa variável na árvore **varGlobais**. Caso não exista, é inserida uma nova estrutura **Nodo** na árvore. Se a variável já existir na árvore é imprimido erro.

Sempre que é encontrada a declaração de uma variável num contexto local, é testada a existência dessa variável na árvore **variaveis**. Caso não exista, é inserida uma nova estrutura **Nodo** na árvore. Se a variável já existir na árvore é imprimido erro. Nesta situação não se testa a existência desta variável na árvore das variáveis globais, pois é permitida a re-declaração de variáveis locais mesmo que já existam globais e é dada prioridade à declaração local.

A árvore **funcoes** contém informação do número de argumentos que cada um das funções encontradas contém. Sempre que é chamada uma função, testa-se a existência dessa mesma função, dando erro caso não esteja declarada, e testa-se também se o número de argumentos está correto.

# Capítulo 2

## Geração de código

Neste capítulo é explicado, sucintamente, a forma como o YACC gera o código *Assembly* para a VM. De um modo geral, a estratégia usada foi a de guardar *strings* com todo código *Assembly* que poderá ser usado e que está associado aos símbolos não-terminais encontrados. O código do programa é construído com a composição das *strings* que sejam necessárias em cada contexto.

### 2.1 Início do Programa

No início do programa existem sempre duas instruções `pushi 0`. A primeira instrução destina-se a criar uma variável “global” cujo objetivo é guardar o resultado de retorno das funções. A segunda tem como objetivo guardar o argumento passado a uma função que receba argumentos.

### 2.2 Declaração de variáveis

Por forma a saber a cada momento qual o próximo endereço disponível para o armazenamento de variáveis na *stack*, foi criado um contador, `count`. Quando se declara uma variável é gerada a instrução `pushi n`, onde `n` corresponde ao valor de inicialização, e é armazenada a informação na árvore de identificadores, de acordo com o contexto, e incrementado o contador (`count++`). No caso dos vetores e das matrizes são geradas instruções `pushn` e incrementado o contador do endereço num valor igual ao tamanho do vetor/matriz.

### 2.3 Declaração de funções

No momento de declaração de uma função é verificado se esta foi declarada anteriormente e, caso não tenha sido, é gerada uma *label* `func_<nome_funcao>` a marcar o início da mesma. De seguida, se a função não receber argumentos, é adicionado à string que contém a *label* o código das declarações e instruções da função. No final, e antes da instrução de retorno, o valor a devolver pela função é colocado em `gp[0]`. Caso a função receba argumentos, antes do código das declarações e das instruções, i.e., do corpo da função, é feito um `pushg 1` para colocar na

*stack* o valor recebido no argumento. De seguida, é gerado o código do seu corpo, tal como descrito anteriormente para as funções sem argumentos.

## 2.4 Retorno

Para fazer o *return* do valor de uma função é necessário em primeiro lugar gerar instruções *assembly* para colocar no topo da *stack* o valor da expressão a ser retornada. De seguida é feito um **storeg 0** para armazenar em **gp[0]** esse mesmo valor. Por fim, é gerada a instrução **return** que permite a continuação do programa no ponto onde foi invocada a função.

## 2.5 Chamada de funções

Na chamada de funções sem argumentos em primeiro lugar é gerada a instrução **pusha**, que coloca o endereço da função no topo da *stack*, seguida da instrução **call**. Caso a chamada à função seja feita numa expressão, em que o valor de retorno é necessário, a seguir à instrução **call** é feito um **push** do valor de retorno para a *stack*, com a instrução **pushg 0**.

Caso a função receba um valor como argumento, tendo em conta que esse valor é uma expressão, antes de tudo são geradas as instruções capazes de obter o resultado da expressão. De seguida é feito um **storeg 1** para guardar no registo o valor a passar como argumento. De seguida é tomado o procedimento já descrito em cima.

## 2.6 Atribuições

O código que é gerado para conseguir a atribuição depende do tipo de dados da variável que está a ser sujeita à atribuição. Se for uma variável atómica, i.e., um inteiro, **INT**, são geradas as instruções necessárias para colocar no topo da *stack* o resultado da expressão do lado direito da atribuição. Por fim, caso se esteja num contexto global, tratando-se certamente de uma variável global, é feito um **storeg i**, onde **i** é o índice da variável. Caso se esteja num contexto local, uma de duas situações podem ocorrer: a variável é global ou é local à função atual. No caso de se tratar de uma variável global o processo é exatamente o mesmo ao descrito no contexto global. Caso se trate de uma variável local, é feito **storel i**.

No caso em que a variável sujeita à atribuição seja de um tipo estruturado de dados, i.e., um *array* (bidimensional, ou não), são geradas, em primeiro lugar, as instruções necessárias para colocar na *stack* o endereço onde se pretende armazenar o valor da atribuição; depois, as instruções que colocam na *stack* o resultado da expressão do lado direito da atribuição e, por fim, é feito um **storen** que guarda o valor da expressão no endereço. A forma como se obtém o endereço onde se pretende armazenar o valor da atribuição depende de se a variável é global ou local, mas pode ser descrito da seguinte forma, supondo que se pretende armazenar em **a[Exp]**, onde **a** é uma variável do tipo *array* e **Exp** é uma expressão.

```

pushgp se global          ou          pushfp se local
pushi indice
padd
Instruções para o resultado de Exp
loadn

```

Se se pretender armazenar um dado valor em `a[Exp1][Exp2]`, onde `a` é uma variável do tipo *array* bidimensional e `Exp1` e `Exp2` são expressões:

```

pushgp se global          ou          pushfp se local
pushi indice
padd
Instruções para o resultado de Exp1
pushi colunas
mul
Instruções para o resultado de Exp2
add
loadn

```

## 2.7 Estrutura if-then-else

Para o código da estrutura **if-then-else**, em primeiro lugar é gerado o respetivo código capaz de colocar na *stack* o resultado da condição. De seguida é feito um salto condicional **jz** para uma *label* a indicar o início do código do bloco **else**, caso exista. Depois da instrução de salto, é inserido o código do bloco **then**. No final deste bloco é ainda inserido um salto não condicional para o fim da estrutura de controlo, por forma a impedir que a execução progrida para o código do bloco **else**. Caso não seja especificada a cláusula **else**, o salto condicional **jz** é associado à *label* do final da estrutura de controlo.

Cada uma das *labels* criadas, são geradas através de um contador, sendo todas diferentes, para todas as estruturas condicionais.

## 2.8 Estrutura while

O código *Assembly* para a estrutura **while** começa com a definição de uma *label* que identifica o seu início. De seguida, é gerado o código que coloca na *stack* o resultado da condição, seguido de um salto condicional **jz** para o final do ciclo. A seguir a esta instrução é inserido o código relativo ao bloco de instruções do **while**, seguido de um salto não condicional para o início da estrutura, por forma a testar novamente a condição.

## 2.9 Leitura do *standard input*

Para a leitura de um valor do *standard input* é gerado o código que coloca no topo da *stack* o endereço da variável onde se pretende armazenar o valor lido, seguido das instruções `read` e `atoi` para a leitura e conversão para inteiro da *string* lida. Por fim é armazenado no endereço da variável, como se tratasse de uma atribuição.

## 2.10 Impressão no ecrã

Para imprimir uma expressão no ecrã é necessário em primeiro lugar gerar código para colocar o valor da expressão na *stack*. De seguida, esse valor é escrito com a instrução `writeln`. No caso de se tratar de uma string, o endereço da string lida é colocado na *stack* através da instrução `pushs` e escrita no ecrã com a instrução `writes`.

## 2.11 Expressões e condições

O código *Assembly* gerado para as expressões e para as condições, tem apenas em conta o tipo de operações em uso, conforme o que é apresentado de seguida.

- Condições:

Expr1 >= Expr2	---->	Expr1 Expr2 supeq
Expr1 <= Expr2	---->	Expr1 Expr2 infeq
Expr1 == Expr2	---->	Expr1 Expr2 equal
Expr1 != Expr2	---->	Expr1 Expr2 equal not
Expr1 < Expr2	---->	Expr1 Expr2 inf
Expr1 > Expr2	---->	Expr1 Expr2 sup

- Expressões:

Expr + Termo	---->	Expr Termo add
Expr - Termo	---->	Expr Termo sub
Expr   Termo	---->	Expr Termo add

- Termos:

Termo * Fator	---->	Termo Fator mul
Termo / Fator	---->	Termo Fator div
Termo % Fator	---->	Termo Fator mod
Termo & Fator	---->	Termo Fator mul

Para os não terminais relacionados com expressões e condições, o código assembly gerado pelas suas acções coloca na *stack* o resultado a expressão/condição.



# Capítulo 3

## Resultados e testes

Neste capítulo do relatório são apresentados os exemplos pedidos no enunciado, nomeadamente, o código do programa escrito na linguagem para cada um exemplos e o código *Assembly* obtido com o compilador implementado.

### 3.1 Exemplo 1 - quadrado

**Descrição:** ler quatro números e dizer se podem ser os lados de um quadrado.

---

LADOSQUAD.MC

---

```
MAIN {
  INT a,b,c,d;

  print "Insira um valor: \n";
  read a;
  print "Insira um valor: \n";
  read b;
  print "Insira um valor: \n";
  read c;
  print "Insira um valor: \n";
  read d;

  IF ((a == b) & (b == c) & (c == d)) {
    PRINT "É um Quadrado\n";
  }
  ELSE {
    PRINT "Os lados não correspondem a um quadrado\n";
  }
}
```

```

pushi 0          read
pushi 0          atoi
start            storel 3
pushi 0          pushl 0
pushi 0          pushl 1
pushi 0          equal
pushi 0          pushl 1
pushs "Insira um valor: \n" pushl 2
writes          equal
read            mul
atoi          pushl 2
storel 0        pushl 3
pushs "Insira um valor: \n" equal
writes          mul
read            jz else_0
atoi          pushs "É um Quadrado\n"
storel 1        writes
pushs "Insira um valor: \n" jump fim_if_0:
writes          else_0:
read            pushs "Os lados não correspondem a um quadrado\n"
atoi          writes
storel 2        fim_if_0:
pushs "Insira um valor: \n" stop
writes

```

### Exemplo da execução

No exemplo apresentado de seguida foi inserido 4 vezes o valor 4. Como era esperado, a resposta obtida foi: “É um Quadrado”

```

Insira um valor:
Insira um valor:
Insira um valor:
Insira um valor:
É um Quadrado

```

(a) Exemplo em que os lados correspondem a um quadrado

OPStack		
Index	Value	Type
0	0	integer
1	0	integer
2	4	integer
3	4	integer
4	4	integer
5	4	integer

(b) Estado final da *Stack*

Figura 3.1: Exemplo de teste

Neste segundo exemplo foram inseridos os seguintes valores: 3, 4, 5, 6. Como era esperado, a resposta obtida foi: “Os lados não correspondem a um quadrado.”

Insira um valor:  
 Insira um valor:  
 Insira um valor:  
 Insira um valor:  
 Os lados não correspondem a um quadrado

(a) Exemplo em que os lados não correspondem a um quadrado

OPStack		
Index	Value	Type
0	0	integer
1	0	integer
2	3	integer
3	4	integer
4	5	integer
5	6	integer

(b) Estado final da *Stack*

Figura 3.2: Exemplo de teste

## 3.2 Exemplo 2 - menor

**Descrição:** Ler um inteiro  $N$ , depois ler  $N$  números e escrever o menor deles.

---

MENORNUM.MC

---

```

INT n;

MAIN {
    INT numLido, menor;

    print "Insira quantos quer ler: \n";
    read n;

    print "De seguida insira os valores\n";
    read numLido;
    menor = numLido;

    WHILE (n > 1) {
        read numLido;
        if(numLido < menor) {
            menor = numLido;
        }
        n = n - 1;
    }

    print "O menor numero introduzido foi: ";
    print menor;
}

```

```

pushi 0                jz fim_while_1
pushi 0                read
pushi 0                atoi
start                 storel 0
pushi 0                pushl 0
pushi 0                pushl 1
pushs "Insira quantos quer ler: \n" inf
writes                jz fim_if_0
read                  pushl 0
atoi                 storel 1
storeg 2              fim_if_0:
pushs "De seguida insira os valores\n" pushg 2
writes                pushi 1
read                  sub
atoi                 storeg 2
storel 0              jump while_1
pushl 0               fim_while_1:
storel 1              pushs "0 menor numero introduzido foi: "
while_1:              writes
pushg 2               pushl 1
pushi 1               writei
sup                   stop

```

### Exemplo da execução

No exemplo apresentado de seguida foi inserido o valor 5 como o número de valores a ler e os seguintes valores para calcular o menor: 4,3,1,2,5. O menor valor encontrado foi o um(1).

**Insira quantos quer ler:**  
**De seguida insira os valores**  
**O menor numero introduzido foi:1**

(a) Menor número da sequência:  
 4,3,1,2,5

OPStack		
Index	Value	Type
0	0	integer
1	0	integer
2	1	integer
3	5	integer
4	1	integer

(b) Estado final da *Stack*

Figura 3.3: Exemplo de teste

### 3.3 Exemplo 3 - produtório

**Descrição:** Ler  $N$  (constante do programa) números e calcular o seu produtório.

---

PRODUTORIO.MC

---

```
Int n = 5;

Main {
    Int resultado, numLido;
    Int i = 0;

    resultado = 1;

    print "Insira 1 a 1 os numeros que quer multiplicar\n";

    while (i < n) {
        read numLido;
        resultado = resultado * numLido;
        i = i + 1;
    }

    print "O produtorio é: ";
    print resultado;
}
```

---

PRODUTORIO.VM

---

pushi 0	atoi
pushi 0	storel 1
pushi 5	pushl 0
start	pushl 1
pushi 0	mul
pushi 0	storel 0
pushi 0	pushl 2
pushi 1	pushi 1
storel 0	add
pushs "Insira 1 a 1 os numeros	storel 2
que quer multiplicar\n"	jump while_0
writes	fim_while_0:
while_0:	pushs "O produtorio é: "
pushl 2	writes
pushg 2	pushl 0
inf	writei
jz fim_while_0	stop
read	

### Exemplo da execução

No exemplo apresentado de seguida foi inserida a seguinte sequência de valores por forma a calcular o seu produtório: 1,2,3,4,5. O produtório resultante desta sequência de valores é 120.

Insira 1 a 1 a sequência de números  
O numero 9 é impar  
O numero 8 não é impar  
O numero 5 é impar  
O numero 3 é impar  
O numero 2 não é impar  
O numero 1 é impar  
Inseriu 4 números ímpares

(a) Produtório da sequência:  
1,2,3,4,5

OPStack		
Index	Value	Type
0	0	integer
1	5	integer
2	120	integer
3	5	integer
4	5	integer

(b) Estado final da *Stack*

Figura 3.4: Exemplo de teste

## 3.4 Exemplo 4 - Números ímpares

**Descrição:** Contar e imprimir os números ímpares de uma sequência de números naturais, terminada por zero (0).

---

IMPARES.MC

---

```
Main {  
  
    Int quantos = 0;  
    Int numLido;  
  
    print "Insira 1 a 1 a sequência de números\n";  
    read numLido;  
  
    while (numLido > 0) {  
  
        if((numLido%2) == 0){  
            print "O numero ";  
            print numLido;  
            print " não é ímpar\n";  
        }  
        else {  
            print "O número ";  
            print numLido;  
            print " é ímpar\n";  
            quantos = quantos + 1;  
        }  
        read numLido;  
    }  
}
```

```

    print "Inseriu ";
    print quantos;
    print " números ímpares";
}

```

---

IMPARES.VM

---

<pre> pushi 0 pushi 0 start pushi 0 pushi 0 pushs "Insira 1 a 1 a sequência       de números\n" writes read atoi storel 1 while_1: pushl 1 pushi 0 sup jz fim_while_1 pushl 1 pushi 2 mod pushi 0 equal jz else_0 pushs "0 numero " writes pushl 1 writei pushs " não é ímpar\n" </pre>	<pre> writes jump fim_if_0: else_0: pushs "0 número " writes pushl 1 writei pushs " é ímpar\n" writes pushl 0 pushi 1 add storel 0 fim_if_0: read atoi storel 1 jump while_1 fim_while_1: pushs "Inseriu " writes pushl 0 writei pushs " números ímpares" writes stop </pre>
---	--

## Exemplo da execução

No exemplo apresentado de seguida foi inserida a seguinte sequência de valores: 9,8,5,3,2,1,0.

Insira 1 a 1 a sequência de números  
O numero 9 é ímpar  
O numero 8 não é ímpar  
O numero 5 é ímpar  
O numero 3 é ímpar  
O numero 2 não é ímpar  
O numero 1 é ímpar  
Inseriu 4 números ímpares

(a) Quantidade de números ímpares existentes  
na sequência: 9,8,5,3,2,1,0

OPStack		
Index	Value	Type
0	0	integer
1	0	integer
2	4	integer
3	0	integer

(b) Estado final da *Stack*

Figura 3.5: Exemplo de teste

## 3.5 Exemplo 5 - Ordenação do array

**Descrição:** 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.

---

ORDENAARRAY.MC

---

```
Int array[5];
```

```
FUNCAO ordena() {  
    int i = 0;  
    int j, menor, tmp;  
    WHILE (i < 4) {  
        j = i + 1;  
        menor = i;  
        WHILE (j < 5) {  
            if (array[j] < array[menor]) {  
                menor = j;  
            }  
            j = j + 1;  
        }  
        tmp = array[i];  
        array[i] = array[menor];  
        array[menor] = tmp;  
        i = i + 1;  
    }  
    return 0;  
}
```



```

Main {
    Int i = 0;
    print "Insira 1 a 1 os 5 numeros que pretende ordenar\n";

    while (i < 5) {
        read array[i];
        i = i + 1;
    }

    ordena();
    print array[0];
    i = 1;
    while (i < 5) {
        print "    -> ";
        print array[i];
        i = i + 1;
    }
}

```

---

ORDENAARRAY.VM

---

<pre> pushi 0 pushi 0 pushn 5 start pushi 0 pushs "Insira 1 a 1 os 5 numeros       que pretende ordenar\n" writes while_3: pushl 0 pushi 5 inf jz fim_while_3 pushgp pushi 2 padd pushl 0 read atoi storen pushl 0 pushi 1 add storel 0 </pre>	<pre> jump while_3 fim_while_3: pusha func_ordena call nop pushgp pushi 2 padd pushi 0 loadn writei pushi 1 storel 0 while_4: pushl 0 pushi 5 inf jz fim_while_4 pushs "    -&gt; " writes pushgp pushi 2 padd pushl 0 </pre>
--	---

```

loadn
writei
pushl 0
pushi 1
add
storel 0
jump while_4
fim_while_4:
stop
func_ordena:
nop
pushi 0
pushi 0
pushi 0
pushi 0
while_2:
pushl 0
pushi 4
inf
jz fim_while_2
pushl 0
pushi 1
add
storel 1
pushl 0
storel 2
while_1:
pushl 1
pushi 5
inf
jz fim_while_1
pushgp
pushi 2
padd
pushl 1
loadn
pushgp
pushi 2
padd
pushl 2
loadn
inf

```

```

jz fim_if_0
pushl 1
storel 2
fim_if_0:
pushl 1
pushi 1
add
storel 1
jump while_1
fim_while_1:
pushgp
pushi 2
padd
pushl 0
loadn
storel 3
pushgp
pushi 2
padd
pushl 0
pushgp
pushi 2
padd
pushl 2
loadn
storen
pushgp
pushi 2
padd
pushl 2
pushl 3
storen
pushl 0
pushi 1
add
storel 0
jump while_2
fim_while_2:
pushi 0
storeg 0
return

```

### Exemplo da execução

No exemplo apresentado de seguida foi inserida a seguinte sequência de valores para ordenar de forma crescente: 10,2,8,6,1.

Insira 1 a 1 os 5 números que pretende ordenar  
1 -> 2 -> 6 -> 8 -> 10

(a) Resulta da ordenação da sequência: 10,2,8,6,1

OPStack		
Index	Value	Type
0	0	integer
1	0	integer
2	1	integer
3	2	integer
4	6	integer
5	8	integer
6	10	integer
7	5	integer

(b) Estado final da *Stack*

Figura 3.6: Exemplo de teste

## 3.6 Exemplo 6 - Ordem Inversa

**Descrição:** Ler e armazenar  $N$  números num *array*. Imprimir os valores por ordem inversa.

---

INVERTEARRAY.MC

---

```
Int n = 5;  
Int array[5];
```

```
FUNCAO inverte() {  
    int i = 0;  
    int j = n - 1;  
    int tmp;  
  
    WHILE (i < n/2) {  
        tmp = array[i];  
        array[i] = array[j];  
        array[j] = tmp;  
        i = i + 1;  
        j = j - 1;  
    }  
    return 0;  
}
```

```
Main {  
    Int i = 0;  
    print "Insira 1 a 1 os 5 numeros que pretende inverter\n";
```

```

while (i < 5) {
    read array[i];
    i = i + 1;
}

inverte();
print array[0];
i = 1;
while (i < 5) {
    print " -> ";
    print array[i];
    i = i + 1;
}
}

```

---

INVERTEARRAY.VM

---

pushi 0	call
pushi 0	nop
pushi 5	pushgp
pushn 5	pushi 3
start	padd
pushi 0	pushi 0
pushs "Insira 1 a 1 os 5 numeros que pretende inverter\n"	writei
writes	pushi 1
while_1:	storel 0
pushl 0	while_2:
pushi 5	pushl 0
inf	pushi 5
jz fim_while_1	inf
pushgp	jz fim_while_2
pushi 3	pushs " -> "
padd	writes
pushl 0	pushgp
read	pushi 3
atoi	padd
storen	pushl 0
pushl 0	loadn
pushi 1	writei
add	pushl 0
storel 0	pushi 1
jump while_1	add
fim_while_1:	storel 0
pusha func_inverte	

```

jump while_2
fim_while_2:
stop
func_inverte:
nop
pushi 0
pushg 2
pushi 1
sub
pushi 0
while_0:
pushl 0
pushg 2
pushi 2
div
inf
jz fim_while_0
pushgp
pushi 3
padd
pushl 0
loadn
storel 2
pushgp
pushi 3
padd
pushl 0
pushgp
pushi 3
padd
pushl 1
pushl 2
storen
pushl 0
pushi 1
add
storel 0
pushl 1
pushi 1
sub
storel 1
jump while_0
fim_while_0:
pushi 0
storeg 0
return

```

### Exemplo da execução

No exemplo apresentado de seguida foi inserida a seguinte sequência de valores para inverter: 1,2,3,4,5.

Insira 1 a 1 os 5 números que pretende inverter  
5 -> 4 -> 3 -> 2 -> 1

(a) Resulta da inversão da sequência de: 1,2,3,4,5

OPStack		
Index	Value	Type
0	0	integer
1	0	integer
2	5	integer
3	5	integer
4	0	integer
5	3	integer
6	2	integer
7	1	integer
8	5	integer

(b) Estado final da Stack

Figura 3.7: Exemplo de teste

## 3.7 Exemplo 7 - Multiplicação de vetores e matrizes

**Descrição:** Ler e fazer a multiplicação de uma matriz (*array* bidimensional) com um vetor (*array*).

---

MATRIZ.MC

---

```
main {
    Int matriz[2][2];
    Int vetor[2];
    Int resultado[2];
    Int i, j, soma;

    print "Insira os valores da matriz\n";
    read matriz[0][0];
    read matriz[0][1];
    read matriz[1][0];
    read matriz[1][1];

    print "Insira os valores do vetor\n";
    read vetor[0];
    read vetor[1];

    i = 0;
    While (i < 2) {
        soma = 0;
        j = 0;
        While (j < 2) {
            soma = soma + matriz[i][j] * vetor[j];
            j = j + 1;
        }
        resultado[i] = soma;
        i = i + 1;
    }

    print "O resultado é: \n";
    print "[";
    print resultado[0];
    print "]\n";
    print "[";
    print resultado[1];
    print "]\n";
}
```

pushi 0	storen
pushi 0	pushfp
start	pushi 0
pushn 4	padd
pushn 2	pushi 1
pushn 2	pushi 2
pushi 0	mul
pushi 0	pushi 1
pushi 0	add
pushs "Insira os valores da matriz\n"	read
writes	atoi
pushfp	storen
pushi 0	pushs "Insira os valores do vetor\n"
padd	writes
pushi 0	pushfp
pushi 2	pushi 4
mul	padd
pushi 0	pushi 0
add	read
read	atoi
atoi	storen
storen	pushfp
pushfp	pushi 4
pushi 0	padd
padd	pushi 1
pushi 0	read
pushi 2	atoi
mul	storen
pushi 1	pushi 0
add	storel 8
read	while_1:
atoi	pushl 8
storen	pushi 2
pushfp	inf
pushi 0	jz fim_while_1
padd	pushi 0
pushi 1	storel 10
pushi 2	pushi 0
mul	storel 9
pushi 0	while_0:
add	pushl 9
read	pushi 2
atoi	inf

jz fim_while_0	storen
pushl 10	pushl 8
pushfp	pushi 1
pushi 0	add
padd	storel 8
pushl 8	jump while_1
pushi 2	fim_while_1:
mul	pushs "0 resultado é: \n"
pushl 9	writes
add	pushs "["
loadn	writes
pushfp	pushfp
pushi 4	pushi 6
padd	padd
pushl 9	pushi 0
loadn	loadn
mul	writei
add	pushs "]\n"
storel 10	writes
pushl 9	pushs "["
pushi 1	writes
add	pushfp
storel 9	pushi 6
jump while_0	padd
fim_while_0:	pushi 1
pushfp	loadn
pushi 6	writei
padd	pushs "]"
pushl 8	writes
pushl 10	stop

### Exemplo da execução

No exemplo apresentado de seguida foi inserido o vetor  $v$  e a matriz  $A$  para realizar a multiplicação da matriz pelo vetor:

$$v = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$



O resultado é:

[5]

[11]

(a) Resultado da  
multiplicação da  
matriz com o vetor

OPStack		
Index	Value	Type
0	0	integer
1	0	integer
2	1	integer
3	2	integer
4	3	integer
5	4	integer
6	1	integer
7	2	integer
8	5	integer
9	11	integer
10	2	integer
11	2	integer
12	11	integer

(b) Estado final da *Stack*

Figura 3.8: Exemplo de teste

## 3.8 Exemplo 8 - Calcular o quadrado de um numero

**Descrição:** Ler um número e invocar uma função que calcula o seu quadrado

---

POWER.MC

```
Funcao power(x) {  
    int res;  
  
    res = x * x;  
  
    return res;  
}  
  
Main {  
    int numLido, res;  
  
    print "Insira um número\n";  
    read numLido;  
  
    res = power(numLido);  
  
    print "O quadrado do número é: ";  
    print res;  
}
```

---

POWER.VM

```

pushi 0
pushi 0
start
pushi 0
pushi 0
pushs "Insira um número\n"
writes
read
atoi
storel 0
pushl 0
storeg 1
pusha func_power
call
nop
pushg 0
storel 1

```

```

pushs "O quadrado do número é: "
writes
pushl 1
writei
stop
func_power:
nop
pushg 1
pushi 0
pushl 0
pushl 0
mul
storel 1
pushl 1
storeg 0
return

```

### Exemplo da execução

No exemplo apresentado de seguida foi inserido o número 50 e realizada a operação `power(50)`.

Insira um número  
O quadrado do número é: 2500

(a) Resultado  
da operação  
`power(50)`

OPStack		
Index	Value	Type
0	2500	integer
1	50	integer
2	50	integer
3	2500	integer

(b) Estado final da *Stack*

Figura 3.9: Exemplo de teste

# Conclusão e aspetos a melhorar

Consideramos que, de um modo geral, o balanço do trabalho desenvolvido é positivo. Os objetivos que nos foram propostos pela equipa docente foram cumpridos.

Foi desenvolvido um compilador para a máquina de *stack* virtual, capaz de suportar as mais básicas operações em linguagens de programação imperativa, ainda que apenas com o tipo de dados inteiro (incluindo *arrays*).

Em termos de aspetos a melhorar consideramos a hipótese de dar possibilidade ao programador de criar funções que recebam mais do que um argumento, a qual não implementamos apenas porque o modo de execução seria similar ao das funções que recebem um argumento. Além disso, seria interessante, dar a possibilidade ao programador de passar como argumento de funções endereços de variáveis, o que por uma questão de tempo não nos foi possível implementar.

Outro aspeto que mereceria ser alvo de interesse num desenvolvimento futuro, é a possibilidade de fazer declarações de variáveis de outros tipos de dados, como caracteres ou mesmo *strings*. E por ultimo, que ficará certamente como um desafio pessoal para todos os membros do grupo a possibilidade de declarar estruturas de variáveis.

# Apêndice A

## Código

---

COMPILADOR.Y

---

```
%{  
    #include <stdio.h>  
    #include <unistd.h>  
    #include <string.h>  
    #include <glib.h>  
    #include <stdlib.h>  
  
    typedef struct infovar {  
        char *instrucoes;  
        char *atribuicoes;  
        char *prep_atribuicoes;  
    } InfoVar;  
  
    typedef struct nodo {  
        int indice;  
        int colunas;  
    } *Nodo;  
  
    GTree *variaveis;  
    GTree *varGlobais;  
    GTree *funcoes;  
    char *erro;  
    Nodo aux = NULL;  
    Nodo e;  
    int count;  
    int local;  
    int countCond = 0;  
    char *funcaoAtual;  
    int * auxFuncoes;  
}%}
```

```

%union {
    char *valString;
    int valInt;
    InfoVar infoV;
}

%token <valString> MAIN IF ELSE WHILE READ PRINT RETURN FUNCAO INT VALOR
    ↪ DESIGNACAO STRING

%type <valString> Programa MainFunc Funcoes Instrucoes Instrucao Decls Decl
    ↪ DeclsVar DeclVar Init Expr Termo Fator Cond Funcao Return
%type <infoV> Var

%%
Programa : Decls Instrucoes Funcoes MainFunc
        {   asprintf(&$$, "pushi 0\n"
                    "pushi 0\n"
                    "%s%s%s%s", $1, $2, $4, $3);
            printf("%s", $$);
        }
        ;

Funcoes : Funcoes Funcao
        {   asprintf(&$$, "%s"
                    "%s", $1, $2);   }
        |
        {   $$ = "";   }
        ;

Funcao : FUNCAO DESIGNACAO '(' ')'
        {   count = 0;
            local = 1;
            funcaoAtual = strdup($2);
            variaveis = g_tree_new((GCompareFunc)strcmp);
            int *func = (int *)malloc(sizeof(int));
            *func = 0;
            g_tree_insert(funcoes, $2, func);
        }
        '{' Decls Instrucoes Return '}'
        {   asprintf(&$$, "func_%s:\n"
                    "nop\n"

```

```

        "%s"
        "%s"
        "%s", $2, $7, $8, $9);
    }

| FUNCAO DESIGNACAO '(' DESIGNACAO ')'
{
    count = 0;
    local = 1;
    funcaoAtual = strdup($2);
    variaveis = g_tree_new((GCompareFunc)strcmp);
    int *func = (int *)malloc(sizeof(int));
    *func = 1;
    g_tree_insert(funcoes, $2, func);
    Nodo n = (Nodo)malloc(sizeof(struct nodo));
    n->indice = count;
    n->colunas = 0;
    g_tree_insert(variaveis, $4, n);
    count++;
}

'{' Decls Instrucoes Return '}'
{
    asprintf(&$$, "func_%s:\n"
        "nop\n"
        "pushg 1\n"
        "%s"
        "%s"
        "%s", $2, $8, $9, $10);
}

;

Return : RETURN Expr ';'
{
    asprintf(&$$, "%s"
        "storeg 0\n"
        "return", $2);
}

;

MainFunc : MAIN
{
    variaveis = g_tree_new((GCompareFunc)strcmp);
    count = 0;
    local = 1;
    funcaoAtual = strdup("MAIN");
}

'{' Decls Instrucoes '}'

```

```

        {    asprintf(&$$, "start\n"
                        "%s"
                        "%s"
                        "stop\n", $4, $5);
        }
    }
;

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

Decl : INT DeclVar ';'
    {    $$ = $2;    }
;

DeclVar : DeclVar
    {    $$ = $1;    }
| DeclVar ',' DeclVar
    {    asprintf(&$$, "%s%s", $1, $3);    }
;

DeclVar : DESIGNACAO Init
    {    if (local == 0) {                                // contexto global
        if ((aux = (Nodo)g_tree_lookup(varGlobais,$1)) != NULL) {
            asprintf(&erro, "Variável %s já declarada", $1);
            yyerror(erro);
        } else {
            Nodo n = (Nodo)malloc(sizeof(struct nodo));
            n->indice = count;
            n->colunas = 0;
            g_tree_insert(varGlobais, $1, n);
            count++;
            asprintf(&$$, "%s", $2);
        }
    } else {
        if (g_tree_lookup(variaveis, $1) != NULL) {
            asprintf(&erro, "Variável %s já declarada na função
            ↪ %s", $1, funcaoAtual);
            yyerror(erro);
        } else {
            Nodo n = (Nodo)malloc(sizeof(struct nodo));

```

```

        n->indice = count;
        n->colunas = 0;
        g_tree_insert(variaveis, $1, n);
        count++;
        asprintf(&$$, "%s", $2);
    }
}
}
| DESIGNACAO '[' VALOR ']'
{
    if (local == 0) {
        // contexto Global
        if((aux = (Nodo)g_tree_lookup(varGlobais,$1)) != NULL){
            asprintf(&erro, "Variável %s já declarada", $1);
            yyerror(erro);
        } else {
            if (atoi($3) < 1) {
                asprintf(&erro, "Dimensão %s do array %s inválida",
                    ↪ $3, $1);
                yyerror(erro);
            } else {
                Nodo n = (Nodo)malloc(sizeof(struct nodo));
                n->indice = count;
                n->colunas = 0;
                g_tree_insert(varGlobais, $1, n);
                count += atoi($3);
                asprintf(&$$, "pushn %s\n", $3);
            }
        }
    } else {
        // Está numa função
        if (g_tree_lookup(variaveis, $1) != NULL) {
            asprintf(&erro, "Variável %s já declarada na função %s",
                ↪ $1, funcaoAtual);
            yyerror(erro);
        } else {
            Nodo n = (Nodo)malloc(sizeof(struct nodo));
            n->indice = count;
            n->colunas = 0;
            g_tree_insert(variaveis, $1, n);
            count += atoi($3);
            asprintf(&$$, "pushn %s\n", $3);
        }
    }
}
| DESIGNACAO '[' VALOR ']' '[' VALOR ']'

```



```

{   if (local == 0) {                                     // Contexto Global
    if((aux = (Nodo)g_tree_lookup(varGlobais,$1)) != NULL){
        asprintf(&erro, "Variável %s já declarada", $1);
        yyerror(erro);
    } else {
        if (atoi($3) < 1 || atoi($6) < 1) {
            asprintf(&erro, "Dimensões do array bidimensional
            ↪ %s inválidas", $1);
            yyerror(erro);
        } else {
            Nodo n = (Nodo)malloc(sizeof(struct nodo));
            n->indice = count;
            n -> colunas = atoi($6);
            g_tree_insert(varGlobais, $1, n);
            count += (atoi($3) * atoi($6));
            asprintf(&$$, "pushn %d\n", atoi($3) * atoi($6));
        }
    }
} else {                                                   // Está numa função
    if (g_tree_lookup(variaveis, $1) != NULL) {
        asprintf(&erro, "Variável %s já declarada na função
        ↪ %s", $1, funcaoAtual);
        yyerror(erro);
    } else {
        Nodo n = (Nodo)malloc(sizeof(struct nodo));
        n->indice = count;
        n -> colunas = atoi($6);
        g_tree_insert(variaveis, $1, n);
        count += (atoi($3) * atoi($6));
        asprintf(&$$, "pushn %d\n", atoi($3) * atoi($6));
    }
}
}

;

Init : '=' Expr
    {   $$ = $2;    }
    |
    {   $$ = "pushi 0\n";    }
    ;

Instrucoes : Instrucoes Instrucao
    {   asprintf(&$$, "%s%s", $1, $2);    }

```

```

|
    {    $$ = "";    }
;

Instrucao : Var '=' Expr ';'
    {    asprintf(&$$, "%s%s%s", $1.prep_atribuicoes, $3,
        ↪ $1.atribuicoes);    }
| IF '(' Cond ')' '{' Instrucoes '}'
    {    asprintf(&$$, "%s"
        "jz fim_if_%d\n"
        "%s"
        "fim_if_%d: \n", $3, countCond, $6,
        ↪ countCond);
        countCond++;
    }
| IF '(' Cond ')' '{' Instrucoes '}' ELSE '{' Instrucoes '}'
    {    asprintf(&$$, "%s"
        "jz else_%d\n"
        "%s"
        "jump fim_if_%d: \n"
        "else_%d:\n"
        "%s"
        "fim_if_%d: \n", $3, countCond, $6,
        ↪ countCond, countCond, $10, countCond);
        countCond++;
    }
| WHILE '(' Cond ')' '{' Instrucoes '}'
    {    asprintf(&$$, "while_%d: \n"
        "%s"
        "jz fim_while_%d\n"
        "%s"
        "jump while_%d\n"
        "fim_while_%d: \n", countCond, $3,
        ↪ countCond, $6, countCond, countCond);
        countCond++;
    }
| PRINT Expr ';'
    {    asprintf(&$$, "%s"
        "writei\n", $2);
    }
| PRINT STRING ';'
    {    asprintf(&$$, "pushs %s\n"
        "writes\n", $2);
    }

```

```

| READ Var ';'
{
    asprintf(&$$, "%s"
              "read\n"
              "atoi\n"
              "%s", $2.prep_atribuicoes, $2.atribuicoes);
}

| DESIGNACAO '(' ')' ';'
{
    if((auxFuncoes = (int *)g_tree_lookup(funcoes, $1)) !=
    ↪ NULL){
        if(*auxFuncoes == 0){
            asprintf(&$$, "pusha func_%s\n"
                          "call\n"
                          "nop\n", $1);
        }
        else {
            asprintf(&erro, "Número de argumentos da função
            ↪ %s inválido", $1);
            yyerror(erro);
        }
    } else {
        asprintf(&erro, "Função %s não declarada", $1);
        yyerror(erro);
    }
}

| DESIGNACAO '(' Expr ')' ';'
{
    if((auxFuncoes = (int *)g_tree_lookup(funcoes, $1)) !=
    ↪ NULL){
        if (*auxFuncoes == 1){
            asprintf(&$$, "%s"
                          "storeg 1\n"
                          "pusha func_%s\n"
                          "call\n"
                          "nop\n", $3, $1);
        } else {
            asprintf(&erro, "Número de argumentos da função %s
            ↪ inválido", $1);
            yyerror(erro);
        }
    } else {
        asprintf(&erro, "Função %s não declarada", $1);
        yyerror(erro);
    }
}

```

```

;

Cond : Expr '>' '=' Expr { asprintf(&$$, "%s%ssupeq\n", $1, $4);}
    | Expr '<' '=' Expr { asprintf(&$$, "%s%sinfeq\n", $1, $4); }
    | Expr '=' '=' Expr { asprintf(&$$, "%s%sequal\n", $1, $4); }
    | Expr '!' '=' Expr { asprintf(&$$, "%s%sequal\nnot\n", $1, $4); }
    | Expr '<' Expr { asprintf(&$$, "%s%sinf\n", $1, $3); }
    | Expr '>' Expr { asprintf(&$$, "%s%ssup\n", $1, $3); }
    | Expr
;

Expr : Termo { $$ = $1; }
    | Expr '+' Termo { asprintf(&$$, "%s%sadd\n", $1, $3); }
    | Expr '-' Termo { asprintf(&$$, "%s%ssub\n", $1, $3); }
    | Expr '|' Termo { asprintf(&$$, "%s%sadd\n", $1, $3); }
;

Termo : Fator { $$ = $1; }
    | Termo '*' Fator { asprintf(&$$, "%s%smul\n", $1, $3); }
    | Termo '/' Fator { asprintf(&$$, "%s%sdiv\n", $1, $3); }
    | Termo '%' Fator { asprintf(&$$, "%s%smod\n", $1, $3); }
    | Termo '&' Fator { asprintf(&$$, "%s%smul\n", $1, $3); }
;

Fator : VALOR { asprintf(&$$, "pushi %s\n", $1); }
    | Var { $$ = $1.instrucoes; }
    | '(' Cond ')' { $$ = $2; }
    | DESIGNACAO '(' ')'
    {
        if((auxFuncoes = (int *)g_tree_lookup(funcoes,$1)) != NULL){
            if(*auxFuncoes == 0){
                asprintf(&$$,"pusha func_%s\n"
                    "call\n"
                    "nop\n"
                    "pushg 0\n", $1);
            }
            else {
                asprintf(&erro, "Argumento da função %s não é válido",
                    ↪ $1);
                yyerror(erro);
            }
        }
        else {
            asprintf(&erro, "Função %s não declarada", $1);
            yyerror(erro);
        }
    }
;

```

```

    }
}
| DESIGNACAO '(' Expr ')'
{
    if((auxFuncoes = (int *)g_tree_lookup(funcoes, $1)) != NULL){
        if(*auxFuncoes == 1){
            asprintf(&$$, "%s"
                    "storeg 1\n"
                    "pusha func_%s\n"
                    "call\n"
                    "nop\n"
                    "pushg 0\n", $3, $1);
        }
        else {
            asprintf(&erro, "Argumento da função %s não é válido",
                    ↪ $1);
            yyerror(erro);
        }
    } else {
        asprintf(&erro, "Função %s não declarada", $1);
        yyerror(erro);
    }
}

;

Var : DESIGNACAO
{
    if(local == 0) {
        // contexto Global
        if ((aux = (Nodo)g_tree_lookup(varGlobais, $1)) == NULL){
            asprintf(&erro, "Variável %s não declarada", $1);
            yyerror(erro);
        }
        else {
            asprintf(&$$, "instrucoes, \"pushg %d\n", aux->indice);
            asprintf(&$$, "atribuicoes, \"storeg %d\n", aux->indice);
            $$, "prep_atribuicoes = \"";
        }
    }
    else {
        // contexto local
        if ((aux = (Nodo)g_tree_lookup(variaveis, $1)) != NULL){
            asprintf(&$$, "instrucoes, \"pushl %d\n", aux->indice);
            asprintf(&$$, "atribuicoes, \"storel %d\n", aux->indice);
            $$, "prep_atribuicoes = \"";
        } else {
            if((aux = (Nodo)g_tree_lookup(varGlobais, $1)) != NULL){

```

```

        asprintf(&$$$.instrucoes, "pushg %d\n", aux->indice);
        asprintf(&$$$.atribuicoes, "storeg %d\n", aux->indice);
        $$$$.prep_atribuicoes = "";
    } else {
        asprintf(&erro, "Variável %s não declarada", $1);
        yyerror(erro);
    }
}
}
}

| DESIGNACAO '[' Expr ']'
{
    if(local == 0) {
        if ((aux = (Nodo)g_tree_lookup(varGlobais, $1)) == NULL){
            asprintf(&erro, "Variável %s não declarada", $1);
            yyerror(erro);
        }
        else {
            asprintf(&$$$.instrucoes, "pushgp\n"
                                "pushi %d\n"
                                "padd\n"
                                "%s"
                                "loadn\n", aux->indice, $3);
            asprintf(&$$$.atribuicoes, "storen\n");
            asprintf(&$$$.prep_atribuicoes, "pushgp\n"
                                "pushi %d\n"
                                "padd\n"
                                "%s", aux->indice, $3);
        }
    }
    else {
        if ((aux = (Nodo)g_tree_lookup(variaveis, $1)) != NULL){
            asprintf(&$$$.instrucoes, "pushfp\n"
                                "pushi %d\n"
                                "padd\n"
                                "%s"
                                "loadn\n", aux->indice, $3);
            asprintf(&$$$.atribuicoes, "storen\n");
            asprintf(&$$$.prep_atribuicoes, "pushfp\n"
                                "pushi %d\n"
                                "padd\n"
                                "%s", aux->indice, $3);
        }
        else {
            if((aux = (Nodo)g_tree_lookup(varGlobais, $1)) != NULL){

```

```

        asprintf(&$$$.instrucoes, "pushgp\n"
                                "pushi %d\n"
                                "padd\n"
                                "%s"
                                "loadn\n", aux->indice, $3);
        asprintf(&$$$.atribuicoes, "storen\n");
        asprintf(&$$$.prep_atribuicoes, "pushgp\n"
                                "pushi %d\n"
                                "padd\n"
                                "%s", aux->indice, $3);
    } else {
        asprintf(&erro, "Variável %s não declarada", $1);
        yyerror(erro);
    }
}
}
}

| DESIGNACAO '[' Expr ']' '[' Expr ']'
{
    if(local == 0) {
        if ((aux = (Nodo)g_tree_lookup(varGlobais, $1)) == NULL){
            asprintf(&erro, "Variável %s não declarada", $1);
            yyerror(erro);
        }
        else {
            asprintf(&$$$.instrucoes, "pushgp\n"
                                "pushi %d\n"
                                "padd\n"
                                "%s"
                                "pushi %d\n"
                                "mul\n"
                                "%s"
                                "add\n"
                                "loadn\n", aux->indice, $3,
                                ↪ aux->colunas, $6);
            asprintf(&$$$.atribuicoes, "storen\n");
            asprintf(&$$$.prep_atribuicoes, "pushgp\n"
                                "pushi %d\n"
                                "padd\n"
                                "%s"
                                "pushi %d\n"
                                "mul\n"
                                "%s"

```

```

        "add\n", aux->indice, $3,
        ↪ aux->colunas, $6);
    }
}
else {
    if ((aux = (Nodo)g_tree_lookup(variaveis, $1)) != NULL){
        asprintf(&$$instrucoes, "pushfp\n"
                                "pushi %d\n"
                                "padd\n"
                                "%s"
                                "pushi %d\n"
                                "mul\n"
                                "%s"
                                "add\n"
                                "loadn\n", aux->indice, $3,
                                ↪ aux->colunas, $6);
        asprintf(&$$atribuicoes, "storen\n");
        asprintf(&$$prep_atribuicoes, "pushfp\n"
                                        "pushi %d\n"
                                        "padd\n"
                                        "%s"
                                        "pushi %d\n"
                                        "mul\n"
                                        "%s"
                                        "add\n", aux->indice, $3,
                                        ↪ aux->colunas, $6);
    } else {
        if ((aux = (Nodo)g_tree_lookup(varGlobais, $1)) != NULL){
            asprintf(&$$instrucoes, "pushgp\n"
                                    "pushi %d\n"
                                    "padd\n"
                                    "%s"
                                    "pushi %d\n"
                                    "mul\n"
                                    "%s"
                                    "add\n"
                                    "loadn\n", aux->indice, $3,
                                    ↪ aux->colunas, $6);
            asprintf(&$$atribuicoes, "storen\n");
            asprintf(&$$prep_atribuicoes, "pushgp\n"
                                            "pushi %d\n"
                                            "padd\n"
                                            "%s"
                                            "pushi %d\n"

```



```

        "mul\n"
        "%s"
        "add\n", aux->indice, $3,
        ↪ aux->colunas, $6);
    } else {
        asprintf(&erro, "Variável %s não declarada", $1);
        yyerror(erro);
    }
}
}
}
;
%%

```

```

#include "lex.yy.c"

```

```

int yyerror (char *s) {
    fprintf(stderr, "ERRO SINTÁTICO %s (Linha %d: | yychar: %d)\n", s,
        ↪ yylineno, yychar);
    return 1;
}

int main() {
    funcaoAtual = strdup("GLOBAL");
    local = 0;
    count = 2;
    varGlobais = g_tree_new((GCompareFunc)strcmp);
    funcoes = g_tree_new((GCompareFunc)strcmp);
    yyparse();
    return 0;
}

```

---

COMPILADOR.L

```

%{
%}

```

```

%option noyywrap
%option yylineno

```

```

%%

```

```

(?i:main)                {    return MAIN;        }

(?i:if)                   {    return IF;          }

```

(?i:else)	{ return ELSE; }
(?i:while)	{ return WHILE; }
(?i:read)	{ return READ; }
(?i:print)	{ return PRINT; }
(?i:return)	{ return RETURN; }
(?i:funcao)	{ return FUNCAO; }
(?i:int)	{ return INT; }
[ - ] ? [ 0 - 9 ] +	{ yylval.valString = strdup(yytext); return VALOR; }
[ - * / ; \ [ \ ] , = > < ! + \ ( \ ) \ { \ } % \   & ]	{ return yytext[0]; }
[ _ a - z A - Z ] [ _ A - Z a - z 0 - 9 ] *	{ yylval.valString = strdup(yytext); return → DESIGNACAO; }
\ " [ ^ " ] * \ "	{ yylval.valString = strdup(yytext); return → STRING; }
( .   \ n ) %%	{ ; }