

Processamento de Linguagens e Compiladores (3º ano de LCC/4º ano de MIEFis)

Compilador de uma Linguagem Imperativa

Jaime Santos (A71739)

Hugo Sousa (A76257)

Luís Bigas (A76964)

6 de Janeiro de 2019

Resumo

Neste relatório irá ser analisada a implementação, e os desafios associados, de um compilador de uma linguagem imperativa original através do uso do conjunto de ferramentas *Yacc* e *Flex*.

Conteúdo

1	Introdução	3
1.1	Estrutura do Relatório	3
2	Objetivo do Trabalho	4
3	Resolução	5
3.1	Análise e Especificação	5
3.1.1	Bison/Yacc	5
3.2	Flex	8
3.3	Biblioteca C	9
3.3.1	Estrutura de dados	9
3.3.2	Funções	9
4	Exemplos de Execução	10
4.1	Soma de inteiros	10
4.1.1	Input:	10
4.1.2	Output:	11
4.2	Soma de Floats	11
4.2.1	Input:	11
4.2.2	Output:	12
4.3	Variável não declarada	12
4.3.1	Input:	12
4.3.2	Output:	13
4.4	Variável já declarada	13
4.4.1	Input:	13
4.4.2	Output:	14
4.5	Condicionais e Ciclos	14
4.5.1	Input:	14
4.5.2	Output:	15
4.6	Leitura de variáveis	16
4.6.1	Input:	16
4.6.2	Output:	17
5	Conclusão	18
A	Código Bison:	19
B	Código Flex:	26

Capítulo 1

Introdução

Para este trabalho pretende-se demonstrar os conhecimentos adquiridos pelo grupo na aplicação de GIC's (Gramáticas Independentes de Contexto).

Para tal, tem-se como objetivo a criação de um compilador, para uma simples linguagem de programação imperativa, constituído pelo analisador sintático em *Flex* que irá reconhecer e devolver os símbolos terminais como tokens ao analisador léxico em *Yacc*. O *Yacc* irá então avaliar a validade das frases fornecidas e executar uma dada ação, como a geração de erros de compilação e código Assembly para a VM.

1.1 Estrutura do Relatório

No seguinte capítulo, irão ser analisados os requisitos para a resolução do trabalho proposto pelo professor, que se baseiam nas capacidades que a linguagem criada terá que apresentar.

No capítulo 3 apresenta-se primeiramente uma visão geral da implementação da solução para este trabalho, seguida da especificação em detalhe desta solução.

No capítulo 4 demonstra-se o funcionamento da linguagem criada através do uso de exemplos de código e o respetivo output na VM fornecida pelo professor seguido de uma breve reflexão dos resultados obtidos.

No 5º e último capítulo encontra-se a conclusão, onde irão ser discutidas as limitações da implementação, desafios encontrados e uma perspetiva geral da linguagem criada.

Capítulo 2

Objetivo do Trabalho

Na criação deste compilador de uma linguagem imperativa, temos como objetivo a criação de uma Gramática Independente de Contexto com as seguintes capacidades:

1. Declaração de variáveis de tipo Inteiro, Real e Booleano.
2. Atribuição de valores a variáveis.
3. Capacidade de execução de operações aritméticas e comparação.
4. Execução de condições (*IF THEN ELSE*) e ciclos (*REPETIR...ATÉ*)
5. Escrita de Inteiros, Reais, Booleanos e *Strings*.
6. Leitura de input (Inteiros e Reais) por parte do utilizador.

Para além dos requisitos mínimos, foi decidido implementar as seguintes funcionalidades:

1. Incrementar e Decrementar uma variável Inteira.
2. Calcular o seno e cosseno de uma variável Real.
3. Escrita da concatenação de duas *Strings*.

Após estes requisitos serem cumpridos, é necessário verificar o funcionamento da linguagem fornecendo códigos exemplo ao executável que irá imprimir, para um ficheiro do tipo *vm*, as instruções necessárias para o correto funcionamento deste na Máquina Virtual. Estes códigos encontram-se no capítulo 4.

Capítulo 3

Resolução

Para atingir os objetivos pretendidos neste trabalho, foi realizado em três partes que em conjunto realizam a compilação de uma linguagem imperativa:

1. A gramática independente de contexto e as suas ações em Bison.
2. Gerador léxico para o reconhecimento do programa a compilar em Flex.
3. Uma biblioteca sobre uma estrutura de dados na linguagem C, criada pelo grupo para facilitar a interpretação e processamento de dados do programa. Esta estrutura irá permitir a indexação de variáveis de modo a impedir que existam múltiplas declarações da mesma, bem como a utilização de uma variável nunca declarada.

A Gramática interpreta o padrão do código da linguagem criada, através do gerador léxico, e gera o código assembly lido pela máquina virtual, assim como tratamento de erros.

Graças à flexibilidade das ferramentas, o programa Bison utiliza a biblioteca C para complementar a interpretação dos dados recebidos.

No próximo capítulo procedemos à demonstração e análise do nosso trabalho.

O código utilizado nestas três ferramentas encontra-se em anexo.

3.1 Análise e Especificação

3.1.1 Bison/Yacc

Gramática

1. A gramática tem início com o padrão:

```
Program : START Tasks END
```

A palavra **START** indica o início do código do programa, a palavra **END** indica o seu final.

2. Tasks:

```
Tasks : Tasks Task ';' ;
```

```
Tasks : Task ';' ;
```

Task define cada uma das linhas/instruções do programa principal, cada uma das linhas termina com o símbolo terminal ponto e vírgula ”;”.

Tasks é a chamada recursiva do padrão, de modo a que seja possível reconhecer mais do que uma **Task**.

3. Task:

Task : startVars

Task : Inst

startVars: contém as produções necessárias à inicialização de variáveis.

Inst: representa a derivação que permitirá fazer as restantes ações disponíveis na linguagem.

Optou-se por esta estrutura de modo a que seja possível declarar variáveis e executar ações intercaladamente.

4. startVars:

startVars : INT Var

startVars : REAL Var

startVars : REAL Var2

startVars : BOOL Var

Var: Nome da Variável(inteiros e bools)

Var2: Nome da Variável(floats)

É de notar que, devido à natureza das produções criadas, estas duas derivações foram a estratégia encontrada pelo grupo de modo conseguir distinguir entre variáveis inteiras e reais, sendo esta uma das limitações do trabalho, visto que **Var** e **Var2** irão derivar em dois símbolos terminais diferentes de modo a evitar conflitos.

INT, REAL, BOOL: Símbolos terminais que determinam o tipo da variável a definir.

5. Var:

Var : STR

STR: Símbolo terminal de reconhecimento do nome de uma variável.

6. Var2:

Var2 : nome

nome: Símbolo terminal de reconhecimento do nome de uma variável float.

7. Insts:

Inst : READ Var

Inst : READ Var2

Inst : WRITE Var

Inst : WRITE Var2

Inst : WRITE STRING

Inst : Var INC

Inst : Var DEC

Inst : IF '(' Cond ')' THEN " Tasks " ELSE

Inst : UNTIL '('Cond')' DO "Tasks"

Insts : Atrib

READ: Leitura de input do utilizador a partir do standard input.

WRITE: Escrita do valor de uma variável no standard output

Note-se que tanto no READ como no WRITE, a derivação escolhida irá depender do tipo da variável.

Caso esta seja um Inteiro ou Booleano, irá derivar pela **Var**, caso seja um Real, derivará por **Var2**, e caso seja pretendido escrever uma String, a derivação escolhida será a da **STRING**.

INC, DEC: Incremento/decremento do valor de uma variável inteira por 1.

IF '(' Cond ')' THEN " Tasks " ELSE Reconhecimento de uma condição if(), else(). **COND** representa a condição que, se aceite, é executado o código contido entre as chavetas, que irá derivar em **Tasks**. **ELSE** reconhece a alternativa de execução quando falha a condição if(), e é não terminal de modo a ser possível derivar em vazio.

UNTIL '('Cond')' DO "Tasks": Ciclo REPETIR...ATÉ, que enquanto a condição presente em **Cond** é falsa, repete o código entre chavetas que deriva em **Tasks**.

Atrib: Define que uma instrução irá poder derivar numa atribuição, que irá ser descrita de seguida.

8. STRINGS:

STRINGS : STRING STRINGS

Derivação utilizada quando é pretendido fazer a escrita de Strings, podendo esta ser vazia, ou várias strings delimitadas por , que irão ser concatenadas pela Máquina Virtual.

9. ELSES:

ELSES : &

ELSES : ELSE '{ Tasks }'

&: Reconhecimento de uma produção vazia caso não seja utilizada a condição else.

ELSE '{ Tasks }': Reconhecimento do padrão **ELSE** e execução do código contido nas chavetas que deriva em **Tasks**.

10. Atribuições:

Atrib : Var '=' Calc

Atrib : Var2 '=' Calc2

Caso seja derivada uma variável Inteira ou Booleana, a atribuição feita a esta será feita através de **Calc**. Caso seja um Real, a atribuição será por **Calc2**.

11. Condições:

Cond : Calc EQ Calc

Cond : Calc NE Calc

Cond : Calc LT Calc

Cond : Calc LE Calc

Cond : Calc GT Calc

Cond : Calc GE Calc

Cond : Calc2 EQ Calc2

Cond : Calc2 NE Calc2

Cond : Calc2 LT Calc2

Cond : Calc2 LE Calc2

Cond : Calc2 GT Calc2

Cond : Calc2 GE Calc2

Cond: Reconhece a comparação entre duas operações aritméticas derivadas ou por **Calc** ou **Calc2**.

EQ, NE, LT, LE, GT, GE: Representam as noções de Igualdade(==), Diferença(!=), Menor(<), Menor ou igual(<=), Maior(>), Maior ou igual(>=), respetivamente.

12. Operações Aritméticas

Através das derivações a seguir definidas podemos realizar operações aritméticas básicas.

Calc : Exp

Calc : Calc '+' Exp

Calc : Calc '-' Exp

Exp : Fat

Exp : Exp '*' Fat

Exp : Exp '/' Fat

Exp : Exp '%' Fat

Fat : '(' Calc ')'

Fat : cos '(' Calc ')'

Fat : sin '(' Calc '='

Fat : typeVar

Calc '+' ou '-' Exp: Definição das somas e subtrações de modo a que sejam as ultimas operações aritméticas a ser feitas (caso não existam parênteses).

Exp: Derivação que irá reconhecer operações como multiplicação, divisão e módulo (no caso dos Inteiros).

Fat: faz o reconhecimento de operações com parênteses e as funções da máquina virtual **cos** e **sin**. Irá também receber o valor das variáveis derivadas por **typeVar**.

Desta forma as operações respeitam a prioridade das operações aritméticas, uma vez que o reconhecimento é feito *Bottom Up*.

MUITO IMPORTANTE

Todas as operações aritméticas derivadas por **Calc** são feitas sobre inteiros. Para a diferenciação dos Reais, optou-se por fazer a derivação **Calc2** que irá ser omitida desta explicação, uma vez que tem uma estrutura semelhante à supracitada, à exceção da operação módulo. Para reconhecimento da variável Real, **Fat2** irá ter uma produção em **typeVar2** que contém informação sobre os Reais.

13. typeVar

typeVar : num

typeVar : rial

typeVar : booleans

typeVar : Var

num, booleans: Reconhecimento literal de um valor Inteiro ou Booleano (símbolo terminal).

Var: Usada para atribuir o valor de uma variável Inteira ou Booleana já definida a outra variável Inteira ou Booleana.

14. typeVar2

rial: Reconhecimento literal de um valor Real (símbolo terminal).

Var2: Usada para atribuir o valor de uma variável Real já definida a outra variável Real.

3.2 Flex

A ferramenta Flex faz a análise léxica para possibilitar o reconhecimento da gramática.

Cada uma das instruções do programa Flex faz com que cada um dos símbolos terminais seja reconhecido pela gramática.

Por exemplo:

```
"START:" { return START; }
```

Reconhece o padrão literal "START:" e retorna-o para ser interpretado pela gramática definida no Bison.

```
[a-zA-Z]+ { yylval.terms=strdup(yytext); return STR; }
```

Reconhece strings, e é copiado o seu conteúdo para a estrutura `yylval`.

3.3 Biblioteca C

3.3.1 Estrutura de dados

```
typedef struct pos{  
char* nome;  
char estado;  
char* type;  
}POS[MAX];
```

nome: uma string com o nome da variável.

estado: estado na posição da estrutura de dados(livre ou ocupado).

type: tipo de dados guardado em cada uma das posições da estrutura.

3.3.2 Funções

1. `void _initVars (POS p);`
Inicia todas as posições como vazias.
2. `int _isDeclared (POS p , char* nome);`
Verifica se uma variável "nome" existe na estrutura de dados, retorna 0 caso não exista e 1 caso contrário.
3. `int _getposi (POS p, char *nome);`
Retorna a posição da variável(nome) na estrutura que é também a posição desta na stack da máquina virtual.
4. `void _addVar (POS p, char* nome, char* tipo);`
Adiciona uma variável na estrutura.
5. `char* _getType (POS p, char* nome);`
Retorna o tipo da variável(nome).

Capítulo 4

Exemplos de Execução

Neste capítulo seguem os vários exemplos da execução de códigos, que variam de simples somas e atribuições, a códigos com condições *IF THEN ELSE* com ciclos. Também serão testados os erros de múltiplas declarações de variáveis e a atribuição a variáveis nunca declaradas.

4.1 Soma de inteiros

Neste exemplo vamos testar a declaração e a soma de dois inteiros. Declaramos o inteiro "a" e escrevemos o valor dele, declaramos o inteiro "b" e atribuímos a "b" a soma dos dois inteiros imprimimos o resultado.

4.1.1 Input:

START:

```
INT a;  
a=10;  
W: a;
```

```
INT b;  
b=a+1;  
W: b;
```

:END

Podemos ver que o código foi compilado com sucesso e os valores foram gravados nas posições pretendidas.

4.1.2 Output:

The screenshot shows the VMS-Projeto IDE interface. The main window is divided into several panes:

- Code:** A list of instructions from 0 to 26. Instruction 26, 'STOP', is highlighted in blue.
- OPStack:** A table showing the current state of the operand stack.
- Heap:** A table showing the current state of the heap memory.
- Call Stack:** A table showing the current state of the call stack.
- Execution Controls:** Buttons for 'Execute 1', 'Execute N:', 'Load Program File', 'Reload Program File', and 'Load Input File'.
- Registers:** A section showing the values of registers PC, FP, SP, and GP.

#--	Instruction	ValueA	TypeA	ValueB	Ty
0	START	-	-	-	-
1	PUSHI	0	INT	-	-
2	PUSHI	10	INT	-	-
3	STOREG	0	INT	-	-
4	PUSHGP	-	-	-	-
5	PUSHI	0	INT	-	-
6	PADD	-	-	-	-
7	LOAD	0	INT	-	-
8	WRITEI	-	-	-	-
9	PUSHS	-	STRING	-	-
10	WRITES	-	-	-	-
11	PUSHI	0	INT	-	-
12	PUSHGP	-	-	-	-
13	PUSHI	0	INT	-	-
14	PADD	-	-	-	-
15	LOAD	0	INT	-	-
16	PUSHI	1	INT	-	-
17	ADD	-	-	-	-
18	STOREG	1	INT	-	-
19	PUSHGP	-	-	-	-
20	PUSHI	1	INT	-	-
21	PADD	-	-	-	-
22	LOAD	0	INT	-	-
23	WRITEI	-	-	-	-
24	PUSHS	-	STRING	-	-
25	WRITES	-	-	-	-
26	STOP	-	-	-	-

4.2 Soma de Floats

Neste exemplo à semelhança do que tínhamos no exemplo anterior fazemos a declaração, soma e escrita das variáveis mas neste caso em vez de inteiros estamos a usar reais.

4.2.1 Input:

START:

```
REAL a.;  
REAL b.;  
REAL c.;
```

```
a.=5.0+1.0;  
b.=6.0+1.5;  
c.=a.+b.;
```

W: c.;

:END

Podemos ver que o código foi compilado com sucesso e os valores foram gravados nas posições pretendidas.

4.2.2 Output:

The screenshot shows the VMS-Projeto IDE interface. The main window is divided into several panes:

- Code:** A list of assembly instructions with their values and types. The instructions are numbered 1 to 29. The last instruction is `STOP`.
- OPStack:** A table showing the current state of the operand stack. It contains three entries: `0 6.000000 FLOAT`, `1 7.500000 FLOAT`, and `2 13.500000 FLOAT`.
- Heap:** A table showing the current state of the heap. It contains one entry: `0`.
- Call Stack:** A table showing the current state of the call stack. It is currently empty.

On the right side of the IDE, there are several buttons and a text area:

- Execute 1:** A button to execute the first instruction.
- Execute N:** A button to execute the instruction at index N.
- Load Program File:** A button to load a program file.
- Reload Program File:** A button to reload the program file.
- Load Input File:** A button to load an input file.
- Text Area:** A text area showing the output of the program. It contains the value `13.500000`.

4.3 Variável não declarada

Vamos agora testar detecções de erro, declaramos um inteiro "b" e atribuímos a uma variável não declarada "a" o valor de b.

4.3.1 Input:

START:

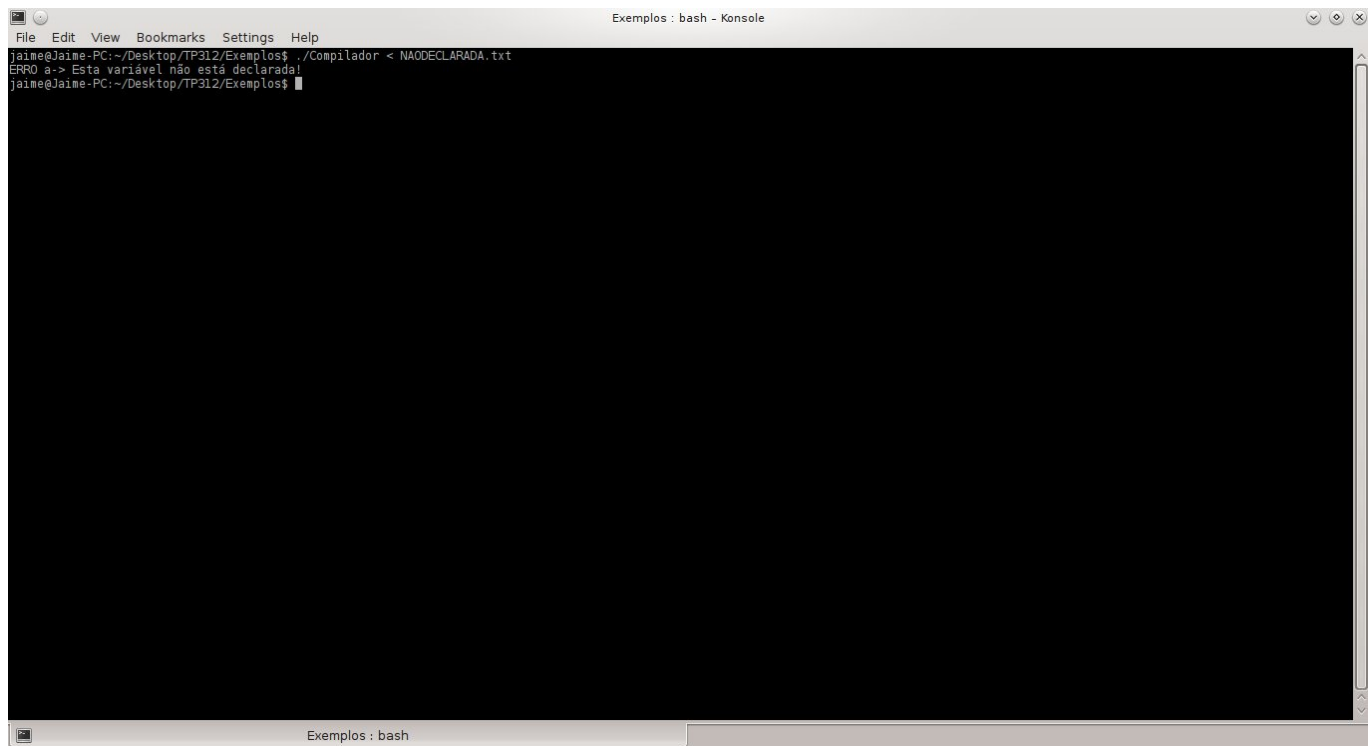
INT b;

a=b;

:END

A mensagem de erro "Variável não declarada" foi escrita corretamente no momento de compilação.

4.3.2 Output:



```
jaime@Jaime-PC:~/Desktop/TP312/Exemplos$ ./Compilador < NAODECLARADA.txt
ERRO a-> Esta variável não está declarada!
jaime@Jaime-PC:~/Desktop/TP312/Exemplos$
```

4.4 Variável já declarada

Neste exemplo vamos testar outra detecção de erro, neste caso a redeclaração de variáveis, primeiro declaramos um inteiro "a" e de seguida um booleano também chamado "a".

4.4.1 Input:

START:

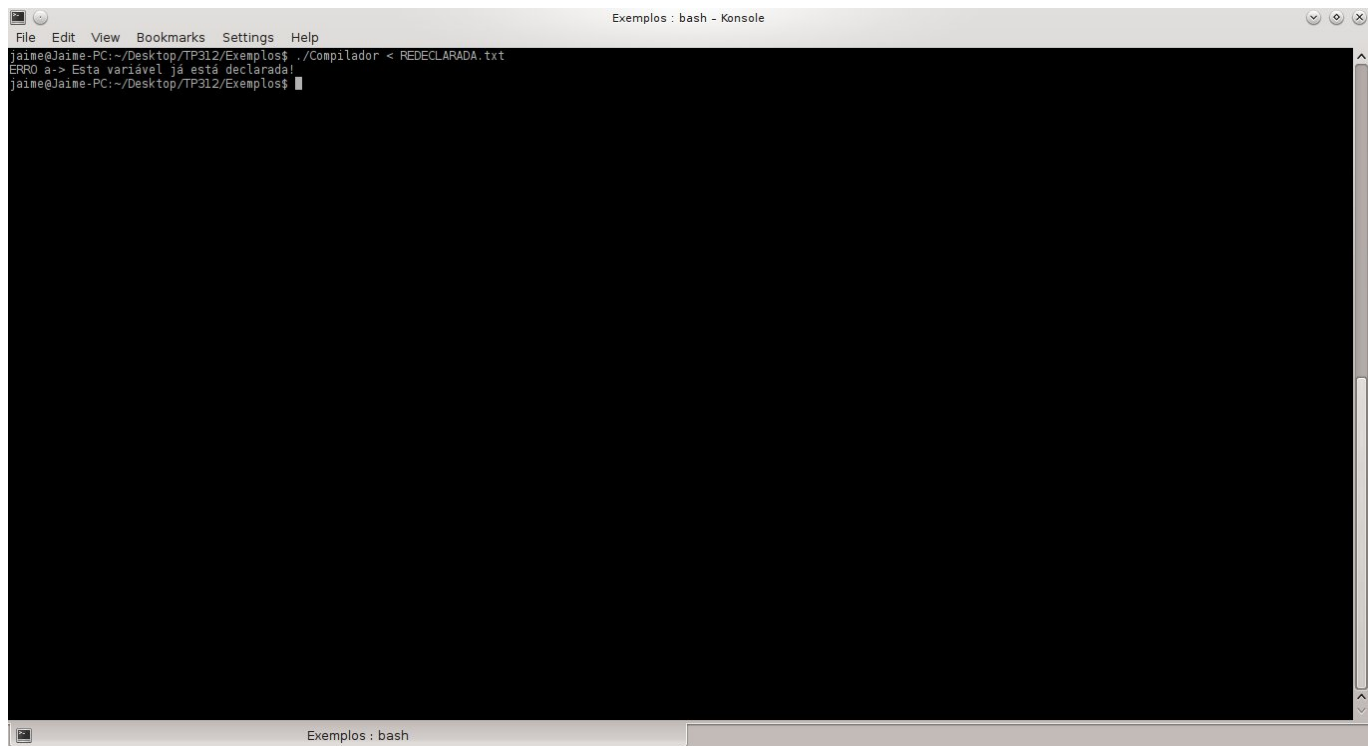
INT a;

BOOL a;

:END

Podemos ver que a mensagem de erro foi escrita no momento de compilação com a mensagem de erro adequada.

4.4.2 Output:



```
jaime@Jaime-PC:~/Desktop/TP312/Exemplos$ ./Compilador < REDECLARADA.txt
ERRO a-> Esta variável já está declarada!
jaime@Jaime-PC:~/Desktop/TP312/Exemplos$
```

4.5 Condicionais e Ciclos

Neste exemplo vamos testar ciclos e condições "IF THEN ELSE". A variável "c" é inicializada a TRUE por isso no primeiro IF vamos derivar pelo "Tasks" do THEN one vai ser escrita uma string com o valor do "a" durante o ciclo e o valor de "a" vai ser decrementado. De seguida, vamos executar outro IF mas desta vez como "d" foi inicializado como FALSE vai derivar pelo ELSE que irá entrar num ciclo que irá printar o valor de "b" e incrementar o mesmo.

4.5.1 Input:

START:

```
INT a;
```

```
a=3;
```

```
INT b;
```

```
b=0;
```

```
BOOL c;
```

```
BOOL d;
```

```
c=TRUE;
```

```
d=FALSE;
```

```
IF(c==TRUE) THEN {
```

```
    UNTIL(a==0) DO {
```

```
        W: "0 valor de a durante o ciclo é:";
```

```
        W: a;
```

```
        a--;
```

```
    };
```



```

        W: "O valor de a no fim do ciclo é:";
        W: a;
    };

IF (d==TRUE) THEN {
    d=FALSE;
}
ELSE { IF (d==FALSE) THEN {
    UNTIL(b==3) DO {
        W: "O valor de b durante o ciclo é:";
        W: b;
        b++;
    };

    W: "O valor de b no fim do ciclo é:";
    W: b;
    };

};

:END

```

Podemos ver na vm que o valor de "a" é corretamente escrito ao longo do ciclo o que indica que está a ser decrementado, e o mesmo se verifica para o "b" que vai ser incrementado.

4.5.2 Output:

The screenshot displays the VMS-Projeto VM monitor interface. The main window is divided into several panels:

- Code:** A list of instructions with their addresses and values. Instruction 107 is highlighted as 'STOP'.
- OPStack:** A table showing the current state of the operand stack. It contains three entries: 0 (0, INT), 1 (3, INT), and 2 (1, INT).
- Heap:** A table showing the current state of the heap. It contains 17 entries, with the value 'a' appearing at addresses 11, 12, 13, 14, 15, and 16.
- Call Stack:** A table showing the current state of the call stack. It is currently empty.
- Right Panel:** Contains controls for executing the program (Execute 1, Execute N: 10), loading program files, and a text area showing the output of the program. The output shows the values of 'a' and 'b' during and at the end of the cycle.

The output text in the right panel is as follows:

```

O valor de a durante o ciclo é:3
O valor de a durante o ciclo é:2
O valor de a durante o ciclo é:1
O valor de a no fim do ciclo é:0
O valor de b durante o ciclo é:0
O valor de b durante o ciclo é:1
O valor de b durante o ciclo é:2
O valor de b no fim do ciclo é:3

```

4.6 Leitura de variáveis

Este exemplo serve para demonstrar a capacidade de leitura de variáveis tanto reais como inteiras. Começa-se por fazer a escrita da concatenação de duas strings e a leitura de uma variável booleana. Dependendo do valor desta variável, o ramo do if a ser percorrido vai encontrar uma leitura de um real ou inteiro.

4.6.1 Input:

START:

```
BOOL a;
REAL c.;
INT d;
INT e;
INT f;
e=10;
```

```
W:"Prentende jogar este jogo?" " ( 0 ou 1)";
R: a;
```

```
IF (a==TRUE) THEN {
```

```
W: "Insira um valor Real";
R: c.;
REAL b.;
b.*cos(c.);
W:"Voce perdeu:";
W:c.;
W:"*";
}
```

```
ELSE { W: "Insira um valor Inteiro";
```

```
R: d;
```

```
UNTIL (e<=5) DO {
```

```
f= d* ((1+2) - 1)*e;
W:"JACKPOT! Voce ganhou:";
W: f;
W: "*";
e--;
```

```
};
```

```
};
```

```
:END
```

Neste caso o "a" foi lido como FALSE como é possível ver pelas múltiplas escritas devido ao ciclo. *Certos simbolos foram substituidos por * devido a conflito no latex.*

4.6.2 Output:

The screenshot displays the VMS-Projeto debugger interface. The main window is divided into several panes:

- Code:** A list of instructions with their addresses and values. Instruction 108 (STOP) is highlighted in blue.
- OPStack:** A table showing the current state of the operand stack.
- Heap:** A table showing the current state of the heap memory.
- Call Stack:** A table showing the current state of the call stack.
- Right Panel:** Contains execution controls and a text area for output.

Code Pane:

#--	Instruction	ValueA
85	PUSHS	"
86	WRITES	JACKPOT! Voce ganhou:"
87	PUSHS	"
88	WRITES	—
89	PUSHGP	—
90	PUSHI	4
91	PADD	—
92	LOAD	0
93	WRITEI	—
94	PUSHS	"
95	WRITES	—
96	PUSHS	€*
97	WRITES	—
98	PUSHS	"
99	WRITES	—
100	PUSHGP	—
101	PUSHI	3
102	PADD	—
103	LOAD	0
104	PUSHI	1
105	SUB	—
106	STOREG	3
107	JUMP	Ali0*
108	STOP	—

OPStack Pane:

#--	Value	Type
0	0	INT
1	0	INT
2	1	INT
3	5	INT
4	12	INT

Heap Pane:

#--	Value
0	P
1	r
2	e
3	n
4	t
5	e
6	n
7	d
8	e
9	
10	j
11	o
12	g
13	a
14	r
15	e
16	e

Call Stack Pane:

#--	PcValue	FpValue

Right Panel:

Execute 1

Execute N:

Load Program File Reload Program File

Load Input File

Prentende jogar este jogo? (0 ou 1)
0
Insira um valor inteiro
1
JACKPOT! Voce ganhou:
20
€
JACKPOT! Voce ganhou:
18
€
JACKPOT! Voce ganhou:
16
€
JACKPOT! Voce ganhou:
14
€

Capítulo 5

Conclusão

Através deste trabalho, conseguimos entender a utilidade e importância de ferramentas com a capacidade de interpretação de Gramáticas Independentes de contexto na análise léxica.

Com o uso destas ferramentas obtemos uma linguagem de programação bastante simples, mas entendemos a potencialidade de aplicação destas no mundo real.

Em termos de desafios, o principal foi a distinção entre variáveis Inteiras e Reais de modo a enviar a instrução correta para a Máquina Virtual. Daqui surge a principal limitação da nossa implementação, que assenta no uso de expressões regulares distintas para a diferenciação entre Reais e Inteiros, devido à existência de derivações diferentes para ambos os tipos de variáveis em determinadas operações.

Concluindo, somos da opinião que este trabalho foi razoavelmente bem conseguido, tendo em conta as limitações mencionadas, uma vez que conseguimos obter resultados satisfatórios dos diferentes testes que fizemos.

Apêndice A

Código Bison:

```
%{
#include <stdio.h>
#include <strings.h>
#include "aux.h"
int asprintf(char **strp,const char *fmt,...);
void yyerror(char *s);
int yylex();
char erro[100];
char aux[100];
POS pos;
char* type;
int jmp=0;
int test=0;
FILE* file;
%}

%union
{
    int inteiro;
    float real;
    int boolconst;
    char* others;
    char* terms;
    char* increments;
}

%token<terms> START END READ WRITE IF THEN ELSE UNTIL DO RETURN VAR REAL BOOL STR INT
%token<terms> STRING EQ LE GE LT GT NE nome sin cos

%token<increments>INC DEC
%token<inteiro> num
%token<real> rial
%token<boolconst> booleans

%type<others>Program Tasks Task startVars Var Inst Atrib Calc Exp Fat typeVar ELSES Cond
%type<others>Calc2 Exp2 Fat2 typeVar2 Var2 STRINGS

%%
```

```

Program : START Tasks END          {fprintf(file,"start\n%s\nstop\n",$2);}
;

Tasks : Tasks Task ';'           {asprintf(&$$,"%s%s",$1,$2);}
    | Task ';'                   {asprintf(&$$,"%s",$1);}
;

Task : startVars                 {asprintf(&$$,"%s",$1);}
    | Inst                      {asprintf(&$$,"%s",$1);}
;

startVars : INT Var      {
    if(!_isDeclared(pos,$2))
    {
        strcat(erro,$2);
        strcat(erro," -> Esta variável já está declarada!");
        yyerror(erro);
    }
    _addVar(pos,$2,"INT");
    asprintf(&$$,"\tpushi 0\n",$1);
}

| REAL Var2      {
    if(!_isDeclared(pos,$2))
    {
        strcat(erro,$2);
        strcat(erro,"-> Esta variável já está declarada!");
        yyerror(erro);
    }
    _addVar(pos,$2,"REAL");
    asprintf(&$$,"\tpushi 0\n");
}

| BOOL Var      {
    if(!_isDeclared(pos,$2))
    {
        strcat(erro,$2);
        strcat(erro,"-> Esta variável já está declarada!");
        yyerror(erro);
    }
    _addVar(pos,$2,"BOOL");
    asprintf(&$$,"\tpushi 0\n");
}

;

```

```

Var : STR          {asprintf(&$$,"%s",$1);}
;

Var2 : nome        {asprintf(&$$,"%s",$1);}
;

Inst : READ Var
      {
        if(!_isDeclared(pos,$2))
        {
          strcat(erro,$2);
          strcat(erro,"-> Esta variável não está declarada!");
          yyerror(erro);
        }

        asprintf(&$$,"\tread\n\tatoi\n\tstoreg %d\n",_getposi(pos,$2));

      }

| READ Var2
      {
        if(!_isDeclared(pos,$2))
        {
          strcat(erro,$2);
          strcat(erro,"-> Esta variável não está declarada!");
          yyerror(erro);
        }

        asprintf(&$$,"\tread\n\tatof\n\tstoreg %d\n",_getposi(pos,$2));

      }

| WRITE Var
      {
        if(!strcmp(_getType(pos,$2),"INT"))
          asprintf(&$$,
            "\tpushgp\n\tpushi %d\n\tpadd\n\tload 0\n\twritei\n\tpushs %s\n\twrites\n",
            _getposi(pos,$2),"\n\n");
        if(!strcmp(_getType(pos,$2),"BOOL"))
          asprintf(&$$,
            "\tpushgp\n\tpushi %d\n\tpadd\n\tload 0\n\twritei\n\tpushs %s\n\twrites\n",
            _getposi(pos,$2),"\n\n");

      }

| WRITE Var2
      {

        asprintf(&$$,
          "\tpushgp\n\tpushi %d\n\tpadd\n\tload 0\n\twritef\n\tpushs %s\n\twrites\n",
          _getposi(pos,$2),"\n\n");

      }

| WRITE STRINGS
      {
        asprintf(&$$,"%s",$2);
      }

```

```

| Var INC {
    asprintf(&$$,
        "\tpushgp\n\tpushi %d\n\tpadd\n\tload 0\n\tpushi 1\n\tadd\n\tstoreg %d",
        _getposi(pos,$1),_getposi(pos,$1));

    }

| Var DEC {
    asprintf(&$$,
        "\tpushgp\n\tpushi %d\n\tpadd\n\tload 0\n\tpushi 1\n\tsub\n\tstoreg %d",
        _getposi(pos,$1),_getposi(pos,$1));

    }

| IF '(' Cond ')' THEN '{' Tasks '}' ELSE {

    asprintf(&$$,
        "%s\tjz Aqui%d\n%s\njump Ali%d\nAqui%d\n:\n%s\nAli%d:"
        , $3, jmp, $7, jmp, jmp, $9, jmp);
    jmp++;

    }

| UNTIL '(' Cond ')' DO '{' Tasks '}' {
    asprintf(&$$,
        "Ali%d:\n%s\tjz Aqui%d\n\tjump Fora%d\nAqui%d:\n%s\tjump Ali%d\nFora%d:",
        jmp, $3, jmp, jmp, jmp, $7, jmp, jmp);
    jmp++;

    }

| Atrib {asprintf(&$$,"%s",$1);}
;

ELSE : {asprintf(&$$,"");}
| ELSE '{' Tasks '}' {asprintf(&$$, "%s",$3);}
;

STRINGS: {asprintf(&$$,"\n\tpushs %s\n\twrites","\n\n");}
| STRING STRINGS {asprintf(&$$,"\n\tpushs %s\n\twrites%s",$1,$2);}

```



```

Atrib : Var '=' Calc
        {
            if(!_isDeclared(pos,$1))
            {
                strcat(erro,$1);
                strcat(erro,"-> Esta variável não está declarada!");
                yyerror(erro);
            }
            asprintf(&$$,"%s\tstoreg %d\n",$3,_getposi(pos,$1));
        }

| Var2 '=' Calc2 {
        if(!_isDeclared(pos,$1))
        {
            strcat(erro,$1);
            strcat(erro,"-> Esta variável não está declarada!");
            yyerror(erro);
        }
        asprintf(&$$,"%s\tstoreg %d\n",$3,_getposi(pos,$1));
    }

;

Calc : Exp
      | Calc '+' Exp
      | Calc '-' Exp
      ;

Calc2 : Exp2
       | Calc2 '+' Exp2
       | Calc2 '-' Exp2
       ;

Exp : Fat
    | Exp '*' Fat
    | Exp '/' Fat
    | Exp '%' Fat
    ;

Exp2 : Fat2
     | Exp2 '*' Fat2
     | Exp2 '/' Fat2
     {if (test>0) {
        strcat(erro,"Impossível dividir por 0!");
        yyerror(erro);
    }
    asprintf(&$$,"%s\tdiv\n",$1,$3);}
     {asprintf(&$$,"%s\tmod\n",$1,$3);}

```

```

        strcat(erro,"Impossível dividir por 0!");
        yyerror(erro);

        } asprintf(&$$,"%s%s\tdiv\n",$1,$3);}

;

Fat : '('Calc')'      {asprintf(&$$,"%s",$2);}
    | typeVar          {asprintf(&$$,"%s",$1);}
;

Fat2 : '('Calc2')'    {asprintf(&$$,"%s",$2);}

    | cos'('Calc2')'  {
                        asprintf(&$$,"%s\tfcos\n",$3);

                        }

    | sin'('Calc2')'  {
                        asprintf(&$$,"%s\tfsin\n",$3); }

    | typeVar2         {asprintf(&$$,"%s",$1);}
;

Cond : Calc EQ Calc   {asprintf(&$$,"%s%s\tequal\n",$1,$3);}
    | Calc NE Calc    {asprintf(&$$,"%s%s\tsub\n",$1,$3);}
    | Calc GE Calc    {asprintf(&$$,"%s%s\tsupeq\n",$1,$3);}
    | Calc LE Calc    {asprintf(&$$,"%s%s\tsinfeq\n",$1,$3);}
    | Calc GT Calc    {asprintf(&$$,"%s%s\tsup\n",$1,$3);}
    | Calc LT Calc    {asprintf(&$$,"%s%s\tsinf\n",$1,$3);}
    | Calc2 EQ Calc2  {asprintf(&$$,"%s%s\tequal\n",$1,$3);}
    | Calc2 NE Calc2  {asprintf(&$$,"%s%s\tsfsub\n",$1,$3);}
    | Calc2 GE Calc2  {asprintf(&$$,"%s%s\tsfsupeq\n",$1,$3);}
    | Calc2 LE Calc2  {asprintf(&$$,"%s%s\tsfinfeq\n",$1,$3);}
    | Calc2 GT Calc2  {asprintf(&$$,"%s%s\tsfsup\n",$1,$3);}
    | Calc2 LT Calc2  {asprintf(&$$,"%s%s\tsfinf\n",$1,$3);}
;

typeVar : num          { if ($1==0) test++; asprintf(&$$,"\tpushi %d\n", $1);}
    | booleans         { if ($1==0) test++;asprintf(&$$,"\tpushi %d\n", $1);}
    | Var              {
                        if(!_isDeclared(pos,$1))
                        {
                            strcat(erro,$1);
                            strcat(erro," -> Variável não declarada!");
                            yyerror(erro);
                        }
                        asprintf(&$$,"\tpushgp\n\tpushi %d\n\tpadd\n\tload 0\n",_getposi(pos,$1));

```

```

    }
;

typeVar2 : rial          { if ($1==0.0) test++ ;asprintf(&$$,"\tpushf %f \n", $1);}
      | Var2            {
      if(!_isDeclared(pos,$1))
      {
        strcat(erro,$1);
        strcat(erro," -> Variável não declarada!");
        yyerror(erro);
      }
      asprintf(&$$,"\tpushgp\n\tpushi %d\n\tpadd\n\tload 0\n",_getposi(pos,$1));
    }
;

%%

#include "lex.yy.c"

void yyerror(char *s)
{
    printf("ERRO %s\n",s);
    exit(-1);
}

int main()
{
    file=fopen("cenas.vm","w+");
    _initVars(pos);
    yyparse();
    fclose(file);
    return 0;
}

```

Apêndice B

Código Flex:

Este ficheiro nao é o utilizado na compilação, uma das linhas teve que ser ocultada devido a conflito com latex, verificar proc.l em anexo.

```
%option noyywrap

%%
"START:"          { return START;          }
":END"            { return END; }
"R:"              { return READ; }
"W:"              { return WRITE; }
"IF"              { return IF; }
"THEN"            { return THEN; }
"ELSE"            { return ELSE; }
"UNTIL"           { return UNTIL; }
"DO"              { return DO; }
"INT"             { return INT; }
"REAL"            { return REAL; }
"BOOL"            { return BOOL; }
"RETURN"          { return RETURN; }
(OCULTADO DEVIDO A ERRO COM LATEX) { yylval.terms=strdup(yytext); return STRING; }
[0-9]+            { yylval.inteiro=atoi(yytext); }
[0-9]+\.\?[0-9]+  { yylval.real=atof(yytext); return rial; }
"TRUE"            { yylval.boolconst = 1; return booleans; }
"FALSE"           { yylval.boolconst = 0; return booleans; }
[a-zA-Z]+\.\.     { yylval.terms=strdup(yytext); return nome; }
[a-zA-Z]+\.\.     { yylval.terms=strdup(yytext); return STR; }
[(){}]=\+="\-\*/%\\.\\[\\]|] { return yytext[0]; }
";"              { return yytext[0]; }
"=="             { return EQ; }
">="            { return GE; }
"<="            { return LE; }
">"             { return GT; }
"<"             { return LT; }
"!="            { return NE; }
\\"cos"          {return cos;}
\\"sin"          {return sin;}
"++"            {return INC;}
"--"            {return DEC;}
[&].+           { ; }
```

```
[ \t\n]*  
.  
%%
```

```
{ ; }  
{ printf("erro %c\n",yytext[0]); }
```

Apêndice C

Biblioteca em C:

```
#include<stdio.h>
#include<string.h>

#define MAX 50
#define ocup 1
#define livre 0

typedef struct pos{
    char* nome;
    char estado;
    char* type;
}POS[MAX];

void _initVars(POS p)
{
    int i = 0;
    for(i=0;i<MAX;i++)
    {
        p[i].estado = livre;
        p[i].nome = "";
        p[i].type = "";
    }
}

int _isDeclared(POS p, char* nome)
{
    int i;
    int r = 0;
    for(i=0;i<MAX;i++)
    {
        if(!strcmp(nome,p[i].nome)) r=1;
    }
    return r;
}

int _getposi(POS p, char *nome)
{

```

```

        int i;
        for(i=0;i<MAX;i++)
        {
            if(!strcmp(nome,p[i].nome)) return i;
        }
    }

void _addVar(POS p, char* nome, char* tipo)
{
    int i = 0;
    while(p[i].estado == ocup) i++;
    p[i].nome = nome;
    p[i].estado = ocup;
    p[i].type = tipo;
}

char* _getType(POS p, char* nome)
{
    int i;
    char* tipo;
    for(i=0;i<MAX;i++)
    {
        if(!strcmp(nome,p[i].nome)) tipo = p[i].type;
    }
    return tipo;
}

```