

Processamento de Linguagens e Compiladores (3º ano de LCC)

Trabalho Prático 2

Relatório de Desenvolvimento do Grupo 2

Bruna Araújo
(a84408)

Daniel Ferreira
(a85670)

Ricardo Cruz
(a86789)

4 de fevereiro de 2021

Resumo

Numa sociedade contemporânea, acentuada pelo constante desenvolvimento das tecnologias, aprender a programar, além de ser um exercício excelente para o cérebro e para o desenvolvimento do raciocínio lógico, pode ajudar com diversas tarefas do dia a dia, em diversas profissões e atividades.

O presente trabalho surge com o intuito de explorar a definição de uma linguagem de programação imperativa e a construção do seu compilador.

Para o desenvolvimento deste trabalho foi utilizado um *parser YACC* para reconhecer e converter a sintaxe da linguagem imperativa em código **VM**, com o auxílio de um analisador léxico *FLex* para identificar as palavras reservadas da mesma.

O principal desafio foi perceber que ações semânticas realizar para geração de código **VM**.

Este estudo permitiu-nos trabalhar pela primeira vez com um *parser*, e perceber a forma de como funciona a ligação entre uma linguagem de programação e um compilador.

Palavras-Chave: Filtro de texto, gerador *FLex*, *parser*, expressões regulares, *YACC*, *VM*, ação semântica, linguagem imperativa, compilador

Conteúdo

1	Introdução	3
1.1	Contextualização	3
1.2	Desafios	3
2	Concepção/desenho da Resolução	4
2.1	Variáveis Globais	4
2.2	Funções Auxiliares	6
2.3	Gramática Independente de Contexto	8
2.4	Conversão para linguagem VM	11
3	Testes	12
3.1	Testes realizados e Resultados	12
4	Conclusão	14
5	Anexos	15
5.1	Anexo A	15
5.1.1	Parser	15
5.2	Anexo B	16
5.2.1	Analisador léxico	16

Lista de Figuras

2.1	Variáveis Globais	4
2.2	Função existe	6
2.3	Função declara	7
2.4	Declaração dos <i>token's</i>	8
2.5	Gramática Parte I	8
2.6	Gramática Parte II	9
2.7	Gramática Parte III	9
2.8	Gramática Parte IV	10
2.9	Gramática Parte V	10
2.10	Gramática Parte VI	11
3.1	Primeiro exemplo	12
3.2	Segundo exemplo	13

Capítulo 1

Introdução

1.1 Contextualização

Este trabalho prático foi realizado no âmbito da UC de Processamento de Linguagens e Compiladores, no 3º ano. Os principais *objetivos*:

- aumentar a experiência em *engenharia de linguagens* e em *programação generativa* (gramatical);
- desenvolver processadores de linguagens segundo o método da *tradução dirigida pela sintaxe*, suportado numa gramática tradutora;
- desenvolver um **compilador** gerando código para uma **máquina de stack virtual**, no ano corrente será usada a **VM, Virtual Machine**;
- utilizar *geradores de compiladores baseados em gramáticas tradutoras*, concretamente o **YACC**, completado pelo *gerador de analisadores léxicos* **Flex**.

1.2 Desafios

Numa primeira fase, idealizamos o "aspeto" da sintaxe que a nossa linguagem iria ter, escrevendo alguns programas com a mesma.

De seguida, desenvolvemos uma Gramática Independente de Contexto (**YACC**) e um analisador léxico (**Flex**), de maneira a que reconhecesse a linguagem estabelecida.

Por último, associamos ações semânticas, na GIC, de modo a transformar os programas da nossa linguagem em código da **VM**.

Capítulo 2

Concepção/desenho da Resolução

2.1 Variáveis Globais

No ficheiro **YACC** introduzimos as variáveis globais necessárias para o bom funcionamento do sistema, como mostra a Figura 2.1.

```
#define TAM 500

void yyerror(char* s);

int yylex();

FILE* fp;
char * mensagem;
char * erroMensagem = "tentou usar uma variavel que nao esta declarada ou nao tem um valor atribuido";

typedef struct variavel{
    GString * nome;
    int valor;
}Variavel;

Variavel lista[TAM];
int erro = 0;
int posicao;
int posicao2;
int counter = 0; //variavel que nos diz a ultima posição ocupada no array com todas as variaveis
int contaIF = 0;
int contaFOR = 0;
int acc;
```

Figura 2.1: Variáveis Globais

De seguida, explicamos o propósito de cada variável:

1. ***FILE* fp*** - é o apontador para o ficheiro onde vamos escrever as instruções máquina;
2. ***char* mensagem*** - string que utilizamos armazenar a mensagem, em caso de erro;
3. ***char* erroMensagem*** - string que contém a mensagem para um erro específico nos fatores;
4. ***Variavel lista[TAM]*** - array de tamanho *TAM* composto por elementos do tipo variável, que nos vai ajudar a perceber as posições das variáveis no *gp*. Quanto ao tipo Variável é uma *struct* com dois campos, (*GString* nome*, que representa o nome da variável e *int valor*;; pode ser 0 ou 1, caso a variável tenha ou não algum valor atribuído);
5. ***int erro = 0*** - variável que nos ajuda a perceber se ocorreu erro (fica com o valor 1) ou não (continua com o valor 0) na execução do programa;
6. ***int posicao, posicao2*** - vão ajudar a perceber que valores colocar em conjunto com os comandos *pushg, storeg, etc.*
7. ***int contaIF = 0*** - vai nos ajudar a escrever as *labels* para as expressões condicionais;
8. ***int contaFOR = 0*** - vai nos ajudar a escrever as *labels* para os ciclos;
9. ***int acc*** - colocamos o *return* dos vários *asprintf*.

2.2 Funções Auxiliares

A seguinte função, Figura 2.2, tem como propósito saber se uma variável já foi declarada, retornando a sua posição, ou não, retornando -1 .

```
int existe(char * var){
    int i = 0;
    int pos = -1;
    GString * aux = g_string_new(var);
    while (i<TAM && pos==-1){
        if(g_string_equal(aux, lista[i].nome) == TRUE){
            pos = i;
        }
        i++;
    }
    return pos;
}
```

Figura 2.2: Função **existe**

A função **declara** é chamada quando está ser lida a parte das declarações, atualizando a variável global lista ou sinalizando erro caso a declaração não seja validada.

```
void declara(char * nome, int x){
    int res = existe(nome);
    int r;
    if(res == -1){
        if(x == 0){
            g_string_append(lista[counter++].nome,nome);
        }
        else{
            if(x > 0){
                int j = 0;
                while(j<x){
                    g_string_append(lista[counter++].nome,nome);
                    j++;
                }
            }
            else{
                acc = asprintf(&mensagem,"O array %s esta a ser declarado com tamanho negativo",nome);
                erro = 1;
            }
        }
    }
    else{
        acc = asprintf(&mensagem,"A variavel %s ja foi declarada",nome);
        erro = 1;
    }
}
```

Figura 2.3: Função **declara**

2.3 Gramática Independente de Contexto

Depois de definida a estrutura da nossa linguagem, elaboramos uma gramática para que a pudesse reconhecer, respeitando os padrões específicos da mesma. Esta gramática tem várias palavras reservadas (definidas com o *%token*) que, em conjunto com o analisador léxico, Anexo B, percorre todo o ficheiro de texto com um programa escrito na nossa linguagem. Tem, também, vários símbolos que delimitam as produções, como por exemplo, "{}" que delimitam as declarações ou então, a lista de instruções, nas condicionais e nos ciclos.

```
%union{
    int valI;
    char * valS;
}
%token <valI>NUM
%token <valS>ID
%token <valS>FRASE
%token <valI> VERDADEIRO FALSO
%token DECLARACOES INTEIRO ARRINTEIRO INICIO FIM SE PARA FAZER SENAO ATE
%token ESCREVER LER
%token EQ NE GE GT LE LT
%token E OU

%type <valS> Expr Termo Fator ExprR Condicao Condicional ListaInstrucoes
%type <valS> Programa ListaDecls Decl Variaveis Variavel Instrucao Atrib Funcao Ciclo
```

Figura 2.4: Declaração dos *token's*

```
ListaProgs : Programa
           | ListaProgs Programa
           ;

Programa : DECLARACOES '{' ListaDecls '}' INICIO ListaInstrucoes FIM
         ;
```

Figura 2.5: Gramática Parte I

```

ListaDecls : Decl
           | Decl ListaDecls
           ;

Decl : INTEIRO Variaveis ';'
     | ARRINTEIRO Variaveis ';'
     ;

Variaveis : Variavel
          | Variaveis ',' Variavel
          ;

Variavel : ID
         | ID '<' NUM '>'
         ;

```

Figura 2.6: Gramática Parte II

```

ListaInstrucoes : Instrucao
                | ListaInstrucoes Instrucao
                ;

Instrucao : Atrib
          | Funcao
          | Condicional
          | Ciclo
          ;

```

Figura 2.7: Gramática Parte III

```

Ciclo : PARA '(' ExprR ATE ExprR ')' FAZER '{' ListaInstrucoes '}'

;

Atrib : ID '<' '-' ExprR ';'

      | ID '<' ExprR '>' '<' '-' ExprR ';'

;

Funcao : ESCREVER '(' ExprR ')' ';'
       | ESCREVER '(' FRASE ')' ';'
;

```

Figura 2.8: Gramática Parte IV

```

Condicional : SE '(' Condicao ')' '{' ListaInstrucoes '}' SENAO '{' ListaInstrucoes '}'

;

Condicao : ExprR
        | ExprR E Condicao
        | ExprR OU Condicao
;

ExprR : Expr
      | Expr EQ Expr
      | Expr NE Expr
      | Expr GE Expr
      | Expr GT Expr
      | Expr LE Expr
      | Expr LT Expr
;

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

```

Figura 2.9: Gramática Parte V

```

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

Fator : NUM
      | '-' NUM
      | ID

      | ID '<' NUM '>'

      | ID '<' ID '>'

      | VERDADEIRO
      | FALSO
      | '(' Expr ')'
      | LER '(' ' ' ')'
      ;

```

Figura 2.10: Gramática Parte VI

2.4 Conversão para linguagem VM

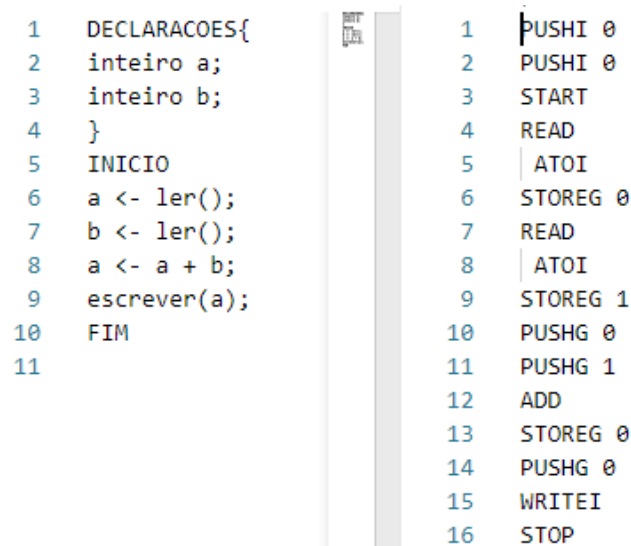
A transição de um programa, escrito com a nossa sintaxe, para linguagem da **VM** é feita através da concatenação dos comandos **VM** resultantes da análise de cada situação específica para, por fim, ser escrito o texto completo, de uma só vez, no ficheiro. Esta metodologia é possível pois definimos os símbolos não terminais, como sendo, do tipo *char** e recorrendo à função *asprintf* da linguagem *C*.

Capítulo 3

Testes

3.1 Testes realizados e Resultados

Nesta secção apresentamos alguns testes. Do lado esquerdo está o programa escrito com a nossa sintaxe e no lado direito o código **VM** correspondente.



1	DECLARACOES{	1	PUSHI 0
2	inteiro a;	2	PUSHI 0
3	inteiro b;	3	START
4	}	4	READ
5	INICIO	5	ATOI
6	a <- ler();	6	STOREG 0
7	b <- ler();	7	READ
8	a <- a + b;	8	ATOI
9	escrever(a);	9	STOREG 1
10	FIM	10	PUSHG 0
11		11	PUSHG 1
		12	ADD
		13	STOREG 0
		14	PUSHG 0
		15	WRITEI
		16	STOP

Figura 3.1: Primeiro exemplo

DECLARACOES{	1	PUSHI 0
inteiro a,b,c,d;	2	PUSHI 0
}	3	PUSHI 0
INICIO	4	PUSHI 0
a <- 1;	5	START
b <- 2;	6	PUSHI 1
c <- 3;	7	STOREG 0
d <- 1;	8	PUSHI 2
se (a == b E a == c E a == d) { escrever("PODE SER"); } senao { escrever("NAO PODE SER");	9	STOREG 1
FIM	10	PUSHI 3
	11	STOREG 2
	12	PUSHI 1
	13	STOREG 3
	14	PUSHG 0
	15	PUSHG 1
	16	EQUAL
	17	PUSHG 0
	18	PUSHG 2
	19	EQUAL
	20	PUSHG 0
	21	PUSHG 3
	22	EQUAL
	23	MUL
	24	MUL
	25	JZ ELSE1
	26	PUSHS "PODE SER"
	27	WRITES
	28	JUMP FIM1
	29	ELSE1 :
	30	PUSHS "NAO PODE SER"
	31	WRITES
	32	FIM1 :
	33	STOP

Figura 3.2: Segundo exemplo

Capítulo 4

Conclusão

Após a realização deste trabalho, conseguimos desenvolver as nossas capacidades de trabalhar com expressões regulares, *FLex* e, também ficamos a compreender melhor o funcionamento de um *parser*.

Apesar das dificuldades iniciais em perceber o que era pedido no enunciado e, posteriormente, no tipo de ações semânticas que tínhamos que fazer para transformar os nossos programas em linguagem **VM**, consideramos que o trabalho desenvolvido cumpre o seu propósito.

A definição da nossa sintaxe "em papel" e posterior transição para linguagem **VM**, "à mão", foram fundamentais para percebermos o rumo que devíamos tomar.

Para concluir, um dos aspetos, que poderíamos ter melhorado, era permitir a definição e invocação de subprogramas sem parâmetros que possam retornar um resultado do tipo inteiro.

Capítulo 5

Anexos

5.1 Anexo A

5.1.1 Parser

Este pin contém o ficheiro com o parser (linguagem.y).



5.2 Anexo B

5.2.1 Analisador léxico

```
1
2 %{
3
4 %}
5 %option noyywrap
6
7 %%
8 [{\}+|-*/%;<>()] { return(yytext[0]); }
9
10 (?i:declaracoes) { return(DECLARACOES); }
11
12 (?i:inteiro) { return(INTEIRO); }
13 ((?i:arr)\-(?i:inteiro)) { return(ARRINTEIRO); }
14
15 (?i:verdadeiro) { yylval.valI = atoi(yytext); return(TRUE); }
16 (?i:falso) { yylval.valI = atoi(yytext); return(FALSE); }
17
18 (?i:inicio) { return(INICIO); }
19
20 (?i:escrever) { return(ESCREVER); }
21 (?i:ler) { return(LER); }
22
23 (?i:se) { contaIF++; return(SE); }
24 (?i:senao) { return(SENAO); }
25
26 (?i:para) { contaFOR++; return(PARA); }
27 (?i:ate) { return(ATE); }
28 (?i:fazer) { return(FAZER); }
29
30 ">>" { return(GT); }
31 "<<" { return(LT); }
32 "==" { return(EQ); }
33 ">=" { return(GE); }
34 "<=" { return(LE); }
35 "#" { return(NE); }
36
37 (?i:e) { return(E); }
38 (?i:ou) { return(OU); }
39
40 (?i:fim) { return(FIM); }
41
42 \ "[a-zA-Z0-9 =:\n,]+\\" { yylval.valS = strdup(yytext); return(FRASE)
43 ; }
44
45 [a-zA-z]+ { yylval.valS = strdup(yytext); return(ID); }
46 [0-9]+ { yylval.valI = atoi(yytext); return(NUM); }
47
48 .|\n { ; }
49 %%
```
