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

Trabalho Prático Nº2
Compilador Para uma Linguagem Imperativa

Relatório de Desenvolvimento

Bruno Alves (85481)

João Marques (84684)

22 de janeiro de 2021

Resumo

Neste projeto teremos de definir uma linguagem de programação imperativa, à nossa escolha. Após a definição desta linguagem, com recurso ás ferramentas **Yacc e Flex** teremos de gerar um compilador que terá de traduzir a linguagem imperativa para pseudo-código Assembly da Máquina Virtual VM.

Conteúdo

Capítulo 1

Introdução

1.1 Compilador Rattle

Área: Compiladores

Enquadramento Construção de Compiladores

Contexto Construção de um compilador de linguagens usando os geradores Flex e Yacc

Problema Pretendemos criar um compilador capaz de ler um ficheiro numa linguagem de programação definida por nós, e de traduzir este ficheiro para código-máquina da máquina virtual VM.

Objetivo O objetivo deste relatório é demonstrar o método utilizado para resolver o problema acima referido, explicando cada passo e decisão tomada para chegarmos a uma solução.

Estrutura do Relatório

Este relatório é iniciado por uma introdução no capítulo 1. No capítulo 1 faz-se uma análise detalhada do problema proposto.
No capitulo 2 é feita uma descrição informal do problema que é proposto no enunciado do trabalho prático assim como a especificação dos seus requisitos.
No capítulo 3 encontra-se a explicação da resolução. Primeiro de uma forma resumida em 3.1 seguido depois por um aprofundamento nas decisões tomadas relativamente a estruturas de dados e algoritmos assim como a geração do código em Assembly.
No capítulo 4 encontra-se um resumo das decisões que tiveram que ser tomadas ao longo do trabalho e problemas que foram surgindo. Na segunda parte deste capítulo encontra-se alguns dos testes realizados pelo grupo durante o trabalho e como estes foram feitos.
Por fim no capítulo 5, uma breve conclusão sobre o trabalho e uma auto-avaliação do trabalho realizado.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

O problema apresentado neste trabalho prático consiste na criação de um compilador para uma linguagem de programação imperativa. A linguagem imperativa em questão deverá ser definida pelo grupo de acordo com as suas preferências. No entanto, a linguagem terá de cumprir alguns requisitos que são enunciados posteriormente no relatório. Após definida a linguagem, é também necessário a construção de um compilador para a mesma. Para o desenvolvimento do compilador será usado a ferramenta Flex para a análise léxica e a ferramenta Yacc para a análise sintática.

2.2 Especificação dos Requisitos

A linguagem de programação definida para este trabalho deverá ser imperativa e terá de permitir:

- Declarar variáveis atómicas do tipo inteiro, com os quais se podem realizar as operaçõoes aritméticas, relacionais e lógicas.
- Efetuar instruções algorítmicas básicas como a atribuição do valor de expressões numéricas a variáveis.
- Ler do standard input
- Escrever no standard output
- Efetuar instruções condicionais
- Efetuar instruções cíclicas (for-do)

As variáveis deverão ser declaradas no início do programa e não pode haver re-declarações nem utilizações sem declaração prévia. Estas variáveis são iniciadas com o valor 0(zero).
Deverá ser criada uma GIC e posteriormente um compilador com base na mesma com recurso Gerador Yacc/Flex. Por fim, deve ser ainda criado um conjunto de testes escritos na linguagem criada para que seja possível observar o respetivo código Assembly gerado e o programa a correr na máquina virtual VM.

Capítulo 3

Concepção/desenho da Resolução

3.1 Resumo da resolução

O primeiro passo tomado foi o de definir uma linguagem através da escrita de uma GIC. A linguagem produzida recebeu o nome de Rattle. Após definida a GIC, foi necessário escrever um analisador léxico usando a ferramenta Flex. Por fim, foi necessário construir-se um analisador sintático usando o Yacc. Neste último passo, tivemos que fazer a conversão da GIC para um Gramática tradutora. Durante a construção do analisador sintático o grupo apercebeu-se da necessidade de se criar estruturas (em C) de maneira a guardar alguns valores e funções para aceder e alterar as estruturas. Todo este processo foi acompanhado por vários testes de maneira a detetar erros e a perceber as limitações da linguagem e ou compilador.

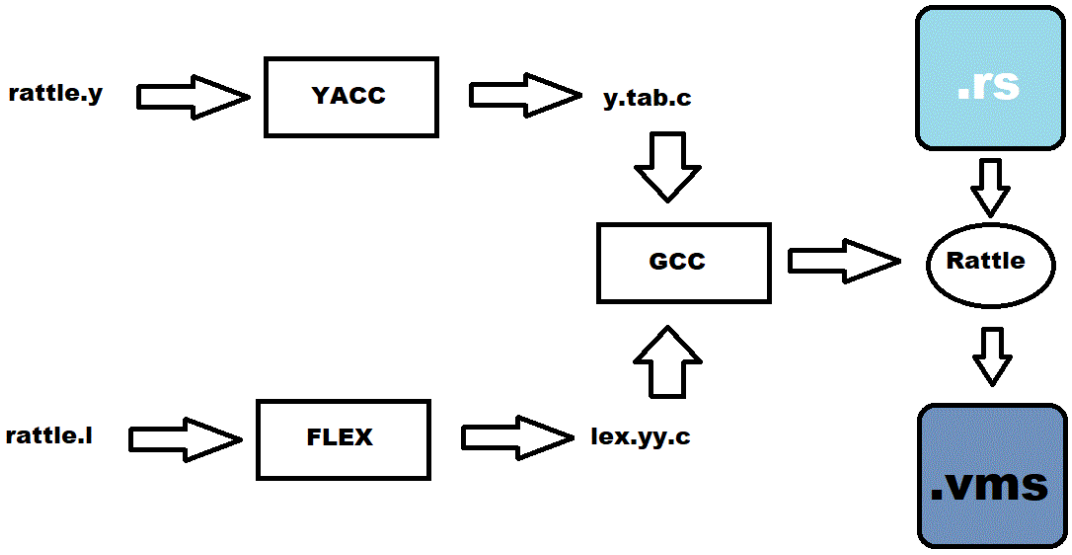

Figura 3.1: Esquema de Resolução

3.2 Linguagem Rattle

Como já referido anteriormente, foi necessário definir-se uma linguagem. Rattle é uma linguagem imperativa bastante simples. As variáveis nesta linguagem devem ser todas definidas no começo do programa. É possível com ela realizar operações matemáticas, escrever para o standard output e ler do standard input, efetuar a instrução cíclica for-do e a instrução condicional if/else para controlo de fluxo e mais algumas ações características de linguagens de programação imperativa simples.

Foi escrita a seguinte gramática para definir a linguagem Rattle:

```
Rattle: Decls Cmds
      | Cmds
      ;

Decls : Decls Decl
      | Decl
      ;

Decl  : VAR ID ';'
      | VAR ID '[' NUM ']' ';'
      ;

Cmds  : Cmds Rat
      | Rat
      ;

Rat   : Atrib
      | If
      | For
      | Inp
      | Out
      ;

Atrib : ID '=' Expr ';'
      | ID '[' Expr ']' '=' Expr ';'
      ;

If    : IF '(' Cond ')' '{' Cmds '}'
      | IF '(' Cond ')' '{' Cmds '}' ELSE '{' Cmds '}'
      ;

For   : FOR '(' ID '|' Cond ')' DO '{' Cmds '}'
      | FOR '(' Atrib '|' Cond ')' DO '{' Cmds '}'
      ;

Inp   : INPUT '(' ID ')' ';'
      | INPUT '(' ID '[' Expr ']' ')' ';'
      ;
```

```
Out   : OUTPUT '(' Expr ')' ';'
      ;

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

Expr  : Fator
      | Expr '+' Expr
      | Expr '-' Expr
      | Expr '*' Expr
      | Expr '/' Expr
      | Expr '%' Expr
      ;

Fator : NUM
      | '-' NUM
      | ID
      | ID '[' Expr ']'
      | TRUE
      | FALSE
      ;
```

3.3 Analisador Léxico em Flex

A etapa de construção do analisador léxico foi relativamente simples sendo que a única tarefa que este desempenha é filtrar o texto de entrada e retornar os simbolos terminais respetivos da gramática acima indicada. Há medida que se foram fazendo testes e com a gramática a ficar cada vez mais complexa foram feitas várias atualizações a este analisador consoante as necessidades.

3.4 Analisador Sintático em Yacc

O analisador sintático feito com utilização da ferramenta Yacc foi o maior foco deste trabalho. Fez-se uma transformação da GIC para uma GT através da implementação de pares Condição-Reação que utilizam a função printf e a função asprintf das bibliotecas do C para produzir o output desejado, ou seja, o código em assembly. Foi também necessário a construção de algumas estruturas de dados e funções em C.

3.4.1 Estruturas dados

```
int spot = 0;
int fors = 0;
int erro = 0;
int gp = 0;


typedef struct variable{
      char* id;
      int posStack;
} *Variable;

Variable v[MAX] = {0};
```

Figura 3.2: Estruturas e Variáveis Auxiliares

Em termos de estruturas de dados usadas, primeiro foi definida uma estrutura *variable* para que fosse possível guardar o id e o endereço base de uma variável global

Após a *struct variable* ter sido criada, foi então criado um array do tipo *variable* de maneira a conseguirmos guardar os endereços base e os ids de várias variáveis. Foram também criadas funções como a *void createVar(char* id)* e a *void createArray(charid, int N)*; que servem para conseguirmos guardar os ids dados pelo nosso código em **rattlesnake** juntamente com os respetivos endereços base a serem usados pelo código **Assembly** da **VM**; a *int inArray(char id* que serve para percorrer o nosso array de structs e verificar se temos guardado o endereço base de um determinado array, e a *int getPos(char *id* que dando o id de uma váriavel devolve-nos a sua posição no registo **gp**

Adicionamos também quatro variáveis locais, *int spot, int fors, int erro, int gp* que servem respetivamente de:

1. marcador para as condições **if**;
2. marcador para os ciclos **for**;
3. flag de **erro**;
4. contador da posição do nosso array do tipo *variable*.

3.4.2 Tradução para Assembly

Para se realizar a conversão do ficheiro de texto que o compilador vai receber para o código Assembly que deverá produzir, á medida que o programa vai derivando, é feita um asprintf do respetivo comando de modo a ir acumulando o código. No final da derivação é executado um printf que irá imprimir todo o código resultante esperado

Capítulo 4

Codificação e Testes

4.1 Alternativas, Decisões e Problemas de Implementação

Uma implementação inicial da gramática que usaríamos para criar o nosso compilador foi relativamente fácil, mas à medida que fomos executando encontrámos problemas que tiveram de ser alterados, como por exemplo um problema em que só conseguiamos aceder ao valor de uma posição de um array se o índice dado fosse um inteiro e não aceitava uma variável com o mesmo valor, sendo estes pequenos problemas numerosos . Pouco depois apercebemo-nos que teriamos de criar uma struct para que fosse possível controlar que endereços base seriam atribuidos a cada variável declarada, esta era inicialmente composta por três variaveis, *char* id* usado para guardar o id da variável, *int posStack* usado para guardar a posição da variável no registo gp, e *int valor* que serviria para guardar o valor atribuido a cada variável, rapidamente reparamos que esta variável seria inútil uma vez que a maquina virtual consegue guardar estes valores e acedê-los quando quiser.

4.2 Testes

Os testes realizados foram inicialmente com inputs simples de maneira a testar a leitura de variáveis. Com o passar do tempo a complexidade dos testes foram aumentando sendo os últimos já com programas como foram pedidos no enunciado do trabalho. Por exemplo, ler quatro inteiros e verificar se podem ser os lados de um quadrado.
Para se realizar os testes primeiro é necessário executar executar os seguintes comandos:

1. lex rattle.l
2. yacc -d rattle.y
3. gcc -o rattle y.tab.c

Depois de gerar o ficheiro em Assembly bastava utilizar a máquina virtual fornecida e observar as suas operações.

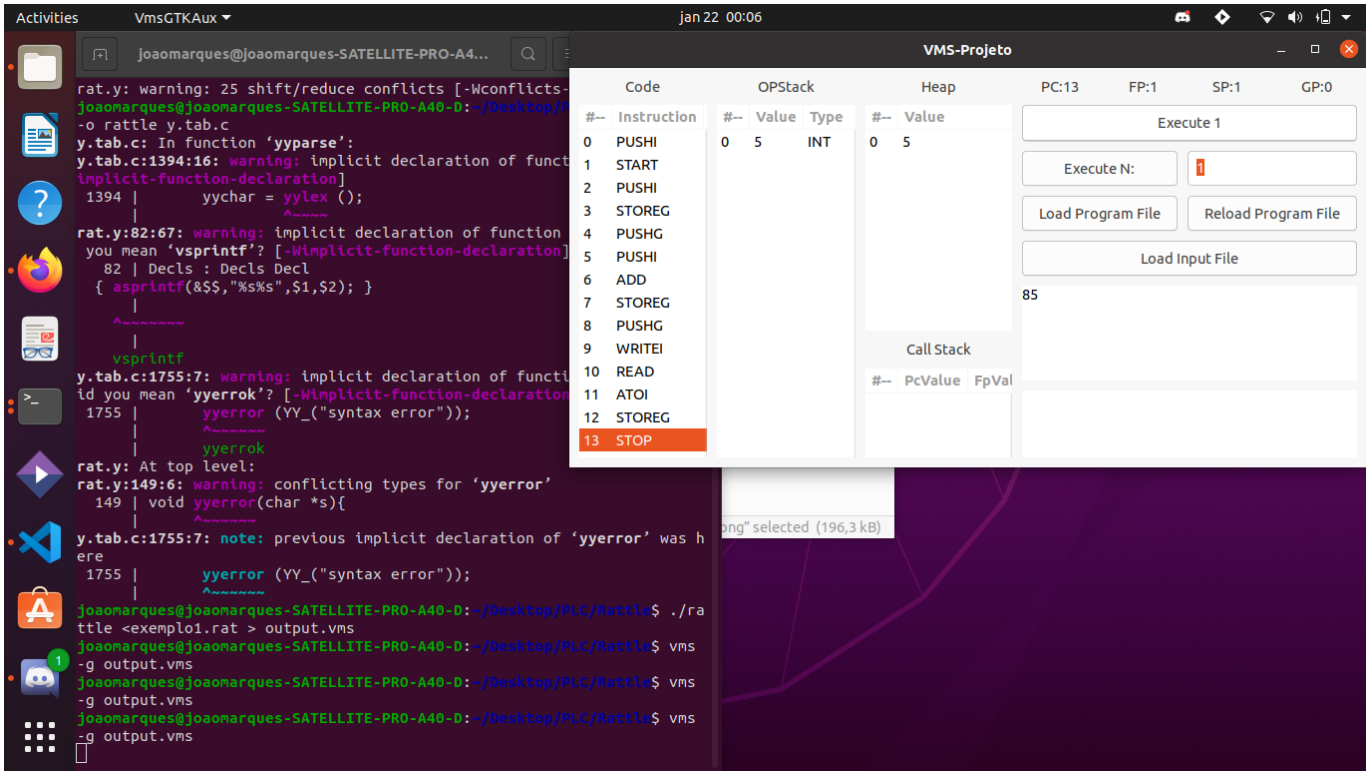

Figura 4.1: Execução de Testes

Capítulo 5

Conclusão

Após a realização deste trabalho foi possível entender um pouco melhor como funcionam os compiladores que usamos no dia-a-dia e permitiu-nos aumentar a nossa experiência de utilização da ferramenta Flex e Yacc assim como escrever GIC´s. Em síntese, o trabalho consistiu em definir uma linguagem de programação imperativa e construir um compilador que gerasse código em Assembly para a máquina virtual fornecida pelo professor.
Apesar das dificuldades encontradas ao longo do trabalho e do resultado produzido ter ainda margem para melhorar de forma a tornar mais eficiente e mais complexa a linguagem Rattle, o grupo sente que o trabalho realizado foi positivo.

Capítulo 6

Anexos

6.1 rattle.l

```
%option noyywrap

%%
(?i:VAR)                    { return(VAR); }
(?i:TRUE)                   { return(TRUE); }
(?i:FALSE)                  { return(FALSE); }
\=\=                        { return(EQ); }
\!\=                        { return(NE); }
\<                          { return(LT); }
\<\=                        { return(LE); }
\>                          { return(GT); }
\>\=                        { return(GE); }
(?i:IF)                     { return(IF); }
(?i:ELSE)                   { return(ELSE); }
(?i:FOR)                    { return(FOR); }
(?i:DO)                     { return(DO); }
(?i:INPUT)                  { return(INPUT); }
(?i:OUTPUT)                 { return(OUTPUT); }
[\=\(\)\{\}\;\+\-\*\/\|\%\[\]]  { return yytext[0]; }
[a-zA-Z][0-9a-zA-Z]*        { yylval.valS=strdup(yytext); return(ID); }
[0-9]+                      { yylval.valN=atoi(yytext); return(NUM); }
[ \t\n]                     { ; }
.                           { printf("Ficheiro Rattle contém caracter inválido [%c]\n",
                            yytext[0]); }
%%
```

6.2 rattle.y

```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 1024

int spot = 0;                       //contador para marcar ifs
int fors = 0;                       //contador para marcar ciclos
int erro = 0;                       //flag de erro
int gp = 0;                         //contador para gp


typedef struct variable{
      char* id;
      int posStack;
} *Variable;

Variable v[MAX] = {0};


void createVar (char* id){
      Variable variavel = (Variable)malloc(sizeof(struct variable));
      variavel->id = id;
      variavel->posStack = gp;
      if(gp>=MAX){
            erro = 1;
        }
      v[gp]=variavel;
      gp++;
}

void createArray(char* id, int N){
      Variable variavel = (Variable)malloc(sizeof(struct variable));
      variavel->id = id;
      variavel->posStack = gp;
      if(gp>=MAX){
            erro = 1;
        }
      int i=gp+N;
      while(gp<i){
            v[gp]=variavel;
            gp++;
      }
}
int inArray(char* id){
```

```
      int i;
      for(i=0; i<MAX; i++){
            if(strcmp(v[i]->id,id)==0){
                return 1;
            }
      }
      return 0;
}

int getPos(char* id){
      int i;
      for(i=0; i<MAX; i++){
            if(strcmp(v[i]->id,id)==0){ return v[i]->posStack; }
      }
}

%}

%union { int valN; char* valS; }
%token <valN>NUM
%token <valS>ID
%token VAR
%token IF ELSE
%token FOR DO
%token INPUT
%token OUTPUT
%token EQ NE LT LE GT GE
%token TRUE FALSE
%type  <valS> Decls Decl Cmds Rat Atrib If For Inp Out Cond Expr Fator

%%
Rattle: Decls Cmds                                              { printf("%sstart\n%sstop\n
",$1,$2); }
      | Cmds                                                    { printf("start\n%sstop\n"
      ,$1); }
      ;

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

Decl  : VAR ID ';'                                              { asprintf(&$$,"pushi 0\n");
createVar($2); }
      | VAR ID '[' NUM ']' ';'                                  { asprintf(&$$,"pushn %d\n",$4)
      ; createArray($2,$4); }
      ;
```

```
Cmds  : Cmds Rat                                                { asprintf(&$$,"%s%s",$1,$2);}
      | Rat                                                     { asprintf(&$$,"%s",$1); }
      ;

Rat   : Atrib                                                   { asprintf(&$$,"%s",$1); }
      | If                                                      { asprintf(&$$,"%s",$1); }
      | For                                                     { asprintf(&$$,"%s",$1); }
      | Inp                                                     { asprintf(&$$,"%s",$1); }
      | Out                                                     { asprintf(&$$,"%s",$1); }
      ;

Atrib : ID '=' Expr ';'                                         { asprintf(&$$,"%sstoreg %d\n"
,$3,getPos($1)); }
      | ID '[' Expr ']' '=' Expr ';'                            { asprintf(&$$,"pushgp\npushi
      %d\n%sadd\n%sstoren\n",getPos($1),$3,$6); }
      ;

If    : IF '(' Cond ')' '{' Cmds '}'                            { asprintf(&$$,"%sjz spot%d\n
%sspot%d:\n",$3,spot,$6,spot); spot++; }
      | IF '(' Cond ')' '{' Cmds '}' ELSE '{' Cmds '}'          { asprintf(&$$,"%sjz spot%d
      \n%sjump spot%d\nspot%d:\n%sspot%d:\n",$3,spot,$6,spot+1,spot,$10,spot+1); spot++; spot++;
      ;

For   : FOR '(' ID '|' Cond ')' DO '{' Cmds '}'                  { asprintf(&$$, "for%d:\n%sjz
endfor%d\n%sjump for%d\nendfor%d:\n",fors,$5,fors,$9,fors,fors); fors++;}
      | FOR '(' Atrib '|' Cond ')' DO '{' Cmds '}'               { asprintf(&$$, "for%d:\n%s%sjz
      enfor%d\n%sjump for%d\nendfor%d:\n",fors,$3,$5,fors,$9,fors,fors);fors++;}
      ;

Inp   : INPUT '(' ID ')' ';'                                    { asprintf(&$$,"read\natoi\nstore
getPos($3)); }
      | INPUT '(' ID '[' Expr ']' ')' ';'                       { asprintf(&$$,"read\natoi\npushg
      %d\n%sadd\nstoren\n",getPos($3),$5); }
      ;

Out   : OUTPUT '(' Expr ')' ';'                                 { asprintf(&$$,"%swritei\n",$3);
      ;

Cond  : Expr EQ Expr                                            { asprintf(&$$,"%s%sequal\n", $1
      | Expr NE Expr                                            { asprintf(&$$,"%s%sequal\nnot\n"
      | Expr LT Expr                                            { asprintf(&$$,"%s%sinf\n", $1, $
      | Expr LE Expr                                            { asprintf(&$$,"%s%sinfeq\n", $1
      | Expr GT Expr                                            { asprintf(&$$,"%s%ssup\n", $1, $
      | Expr GE Expr                                            { asprintf(&$$,"%s%ssupeq\n", $1
      ;

Expr  : Fator
      | Expr '+' Expr                                           { asprintf(&$$,"%s%sadd\n",$1,$3)
```

```
      | Expr '-' Expr                                                { asprintf(&$$,"%s%ssub\n",$1,$3)
      | Expr '*' Expr                                                { asprintf(&$$,"%s%smul\n",$1,$3)
      | Expr '/' Expr                                                { if($3){ asprintf(&$$,"%s%sdiv\n
      }else{printf("Erro: Divisao por 0"); $$=0; erro=1;} }
      | Expr '%' Expr                                                { asprintf(&$$,"%s%smod\n",$1,$3)
      ;

Fator : NUM                                                          { asprintf(&$$,"pushi %d\n",$1);
      | '-' NUM                                                      { asprintf(&$$,"pushi %d\n",(-1)*
      | ID                                                           { if(inArray($1)==1){ asprintf(&$
      "pushg %d\n",getPos($1)); }else{printf("Erro: Variavel %s não existe",$1); $$=0; erro=1;} }
      | ID '[' Expr ']'                                              { if(inArray($1)==1){ asprintf(&$
      ,"pushgp\npushi %d\n%sadd\nloadn\n",getPos($1),$3); }else{printf("Erro: Array %s não existe
      | TRUE                                                         { asprintf(&$$,"pushi %d\n",1); }
      | FALSE                                                        { asprintf(&$$,"pushi %d\n",0); }
      ;


%%

#include "lex.yy.c"

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

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