Universidade do Minho

LEI 3ºAno 2ºSemestre

Processamento de Linguagens

TP2 - Compilador para a LPIS Fruit & Juice

Daniel Caldas a67691
Marcelo Gonçalves a67736
Ricardo Silva a67728

6 de Junho de 2015

Resumo

No âmbito da UC de *Processamento de Linguagens*, foi proposto ao grupo de trabalho que desenvolvesse uma simples **LPIS** (Linguagem de Programação Imperativa Simples) e respetivo compilador que deverá gerar código para uma máquina virtual de stack. O presente relatório servirá então para apresentar uma proposta de solução, a LPIS ***FRUIT & JUICE***. Iremos apresentar os passos de desenvolvimento da linguagem desde a sua especificação formal (através de uma gramática), até à sua implementação (o *compilador* em Yacc e o analisador léxico em Flex). Finalmente iremos demonstrar os resultados obtidos e apresentar as respetivas conclusões.

Conteúdo

Capítulo 1

Introdução

Nesta introdução iremos apresentar os passos para resolução do problema, ou seja, uma espécie de guião das etapas executadas para desenvolvimento do projeto.

1.1 Abordagem ao problema

De seguida listamos os passos executados para desenvolvimento do projeto:

- Desenhar a linguagem;
- Formalizar a linguagem (verificar se conseguimos responder à pergunta: "*Esta gramática define rigorosamente a nossa linguagem?*");
- Validar a gramática junto dos docentes;
- Especificar o processador;
- Análise semântica (estudo das acções semânticas);
- Implementação do Analisador Léxico (AL em **Flex**);
- Implementação do (compilador em **Yacc**).

Capítulo 2

Linguagem

A linguagem FRUIT & JUICE [1] foi desenhada de forma simplista, portanto daí resulta a estrutura da linguagem nos seguintes blocos que passamos a apresentar:

- **Bloco de declarações**: Neste bloco deverão ser declaradas as variáveis a utilizadar na codificação do programa, à **semelhança** de linguagems como o ***Pascal*** por exemplo.
- **Bloco de instruções**: Bloco onde está codificado o programa propriamente dito, ou seja, as instruções que queremos interpretar e transformar em código para a máquina virtual.

[1] FRUIT & JUICE - O nome surgiu de uma analogia da preparação de um sumo de frutas à programação, a fruta são as variáveis, o que o programa criado produz é o sumo.

2.1 Sintaxe

Segundo a linguagem que desenhamos todo o programa deve começar com a marca **FJBEGIN** [2], de seguida **FRUIT**, é então neste momento que declaramos as nossas variáveis, este bloco termina a quando da marca **JUICE** que marca o início das intruções (ou linhas de código propriamente ditas se preferirmos). Após escrito o programa este deve terminar com a marca **FJEND**.

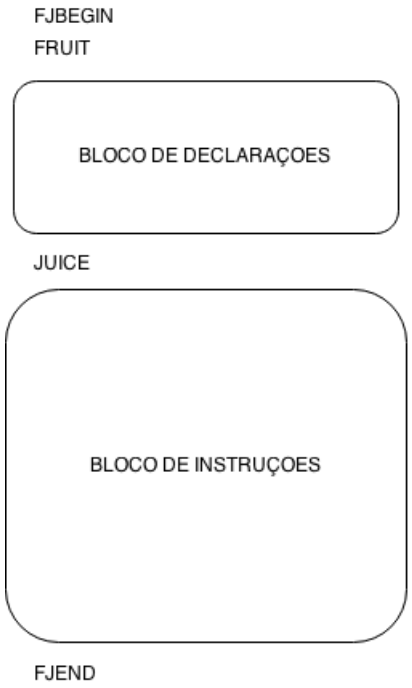

Figura 2.1: Estrutura de um programa escrito na nossa linguagem.

No esquema da figura 2.2 em cima pode ser vista a estrutura base típica de um programa escrito numa qualquer linguagem de programação, neste caso específico na liguagem que propomos implementar.

Usamos os operadores aritméticos e lógicos habituais. Relativamente ao controlo de fluxo, usamos tal como na linguagem **C**, **IF ELSE** para exprimir condições, **WHILE** e **FOR** para fluxo cíclico.

Relativamente a arrays, podem ser declarados de acordo com o exemplo:

```
v[10]; array[5]={1,2,3};
```

. As operações de input e output são executadas através de:

```
WOUT(nome_var); // Para escrever para o standard output
RINP(nome_var) // Para ler do standard input.
```

[2] Iremos justificar mais à frente neste relatório que as palavras reservadas podem ser escritas em letras maiúsculas ou minúsculas, mas nesta secção fica apenas a nota

Agora que já introduzimos verbalmente a nossa sintaxe aqui fica um exemplo de um simples programa que faz apenas atribuições de valores numéricos e operações aritméticas sobre os mesmo.

```
1  FJBEGIN
2  FRUIT
3      x=1;
4      y=2;
5      w;
6  JUICE
7      w=x+y;
8      w=x+5;
9  FJEND
```

Figura 2.2: Exemplo de um simples programa escrito em FRUIT & JUICE.

Uma vez tendo nós definido a sintaxe definindo algumas das palavras reservadas e tendo ideia de como queremos ver refletidas no código as diversas operações, vamos fazer a ponte deste capítulo mais informal e entrar num contexto mais formal do problema, o próximo capítulo é então a definição formal do problema através da escrita de uma gramática, que **consiga espelhar estas características**, por nós definidas nesta secção.

Capítulo 3

Gramática

Neste capítulo do relatório vamos apresentar formalmente a gramática desenvolvida no processo da criação da linguagem de programação imperativa simples *FRUIT & JUICE*.

3.1 T, N e S

De seguida especificamos os símbolos terminais (T), não terminais (N) e axioma da linguage (S).

T = { NOME NUMERO '{' '}' '(' ')' '[' ']' '=' '>' '<' ',' ';' '+' '-' '*' '/' '%' '&' '|' '!' FJBEGIN FJEND FRUIT JUICE IF ELSE WHILE FOR WOUT RINP }

N = { Programa, Linhas, Declaracoes, Codigo, ListaDeclaracoes, ListaVar, Var, ListaNum, ListaNums, Print, Atribuicao, Expr, If, Ciclo, Expr, Oper }

S = Programa

3.2 Produções

P = {

Programa $\rightarrow FJBEGIN\ Linhas\ FJEND$

Linhas $\rightarrow \epsilon$
$\quad | \ FRUIT\ Declaracoes\ JUICE\ Codigo$

Declaracoes $\rightarrow \epsilon$
$\quad | \ ListaDeclaracoes$

ListaDeclaracoes $\rightarrow \ ListaDeclaracoes\ ListaVar\ ';'$
$\quad | \ ListaVar\ ';'$

ListaVar $\rightarrow Var$
$\quad | \ ListaVar\ ','\ Var$

Var $\rightarrow NOME$
$\quad | \ NOME\ '='\ NUMERO$
$\quad | \ NOME\ '['\ NUMERO\ ']'$
$\quad | \ NOME\ '['\ NUMERO\ ']'\ '='\ ListaNum$

ListaNum $\rightarrow '\{'\ '\}'$
$\quad | \ '\{'\ ListaNums\ '\}'$

ListaNums $\rightarrow NUMERO$
$\quad | \ ListaNums\ ','\ NUMERO$

Codigo $\rightarrow \epsilon$
$\quad | \ Lines$

Lines $\rightarrow \ Lines\ Linha$
$\quad | \ Linha$

Linha $\rightarrow \epsilon$
$\quad | \; Atribuicao \; ';'$
$\quad | \; If$
$\quad | \; Ciclo$
$\quad | \; Print \; ';'$

Atribuicao $\rightarrow NOME \; '=' \; Expr$
$\quad | \; NOME \; '[' \; NUMERO \; ']' \; '=' \; Expr$
$\quad | \; RINP \; '(' \; NOME \; ')'$
$\quad | \; RINP \; '(' \; NOME \; '[' \; NUMERO \; ']' \; ')'$

If $\rightarrow \; IF \; '(' \; Condicao \; ')' \; '\{' \; Codigo \; '\}'$
$\quad | \; IF \; '(' \; Condicao \; ')' \; '\{' \; Codigo \; '\}' \; ELSE \; '\{' \; Codigo \; '\}'$

Condicao $\rightarrow \; Expr \; '<' \; Expr$
$\quad | \; Expr \; '=' \; '<' \; Expr$
$\quad | \; Expr \; '>' \; Expr$
$\quad | \; Expr \; '>' \; '=' \; Expr$
$\quad | \; Expr \; '=' \; '=' \; Expr$
$\quad | \; '!' \; '(' \; Condicao \; ')'$

Ciclo $\rightarrow WHILE \; '(' \; Condicao \; ')' \; '\{' \; Codigo \; '\}'$
$\quad | \; FOR \; '(' \; Atribuicao \; ';' \; Condicao \; ';' \; Atribuicao \; ')' \; '\{' \; Codigo \; '\}'$

Expr $\rightarrow \; Valor$
$\quad | \; Expr \; Oper \; Valor$

Valor $\rightarrow \; NUMERO$
$\quad | \; NOME$
$\quad | \; NOME \; '[' \; NUMERO \; ']'$

Oper $\rightarrow OperAdicao$
$\quad | OperMultiplicacao$

OperAdicao $\rightarrow \; '+' \; | \; '-' \; | \; '|'$

OperMultiplicacao $\rightarrow \; '*' \; | \; '/' \; | \; '\%' \; | \; ''$

Print $\rightarrow \; WOUT \; '(' \; NUMERO \; ')'$
$\quad | \; WOUT \; '(' \; NOME \; ')'$
$\quad | \; WOUT \; '(' \; NOME \; '[' \; NUMERO \; ']' \; ')'$

}

Validada a gramática com os docentes, e feitos um reajustes finais podemos então passar para etapa seguinte, já muito próxima da fase de implementação.

Capítulo 4

Arquitetura do compilador

Nesta etapa iremos, explicitar de modo sucinto como foi pensado o compilador, ou seja vamos apresentar cada componente que no seu todo forma o compilador da nossa LPIS.

4.1 Arquitetura

Relativamente às componentes que constituem o compilador e suas funções, segue-se uma breve descrição geral acerca de cada uma delas:

- **fruitjuice.l** - o analisador léxico que faz o reconhecimento dos tokens definidos em y.tab.h. É também responsável por passar valores para o yacc através da union por nós definida;
- **fruitjuice.y** - parser em yacc. Implementa a gramática, faz análise sintática e semântica, gera código para a máquina virtual;
- **hash.c** - contém uma tabela de hash implementada pelo grupo. Esta tabela encontra-se encapsulada no módulo hash.c, representa a tabela de símbolos (variáveis de um programa);
- **const.h** - onde se encontram definidas algumas constantes, como o tipo de dados ou mensagens de erro. Neste ficheiro definimos as duas constantes mais importantes neste contexto, que são os tipos inteiro e array de inteiros. Portanto definimos a **macro INT** para inteiros e a **macro INT_ARRAY** para arrays de inteiros.

No esquema da página seguinte, podemos observar como estas componentes interagem por forma a no final obtermos um executável (**fj.out**), capaz interpretar código na linguagem **FRUIT & JUICE**, e gerar código assembly para a máquina de stack virtual.

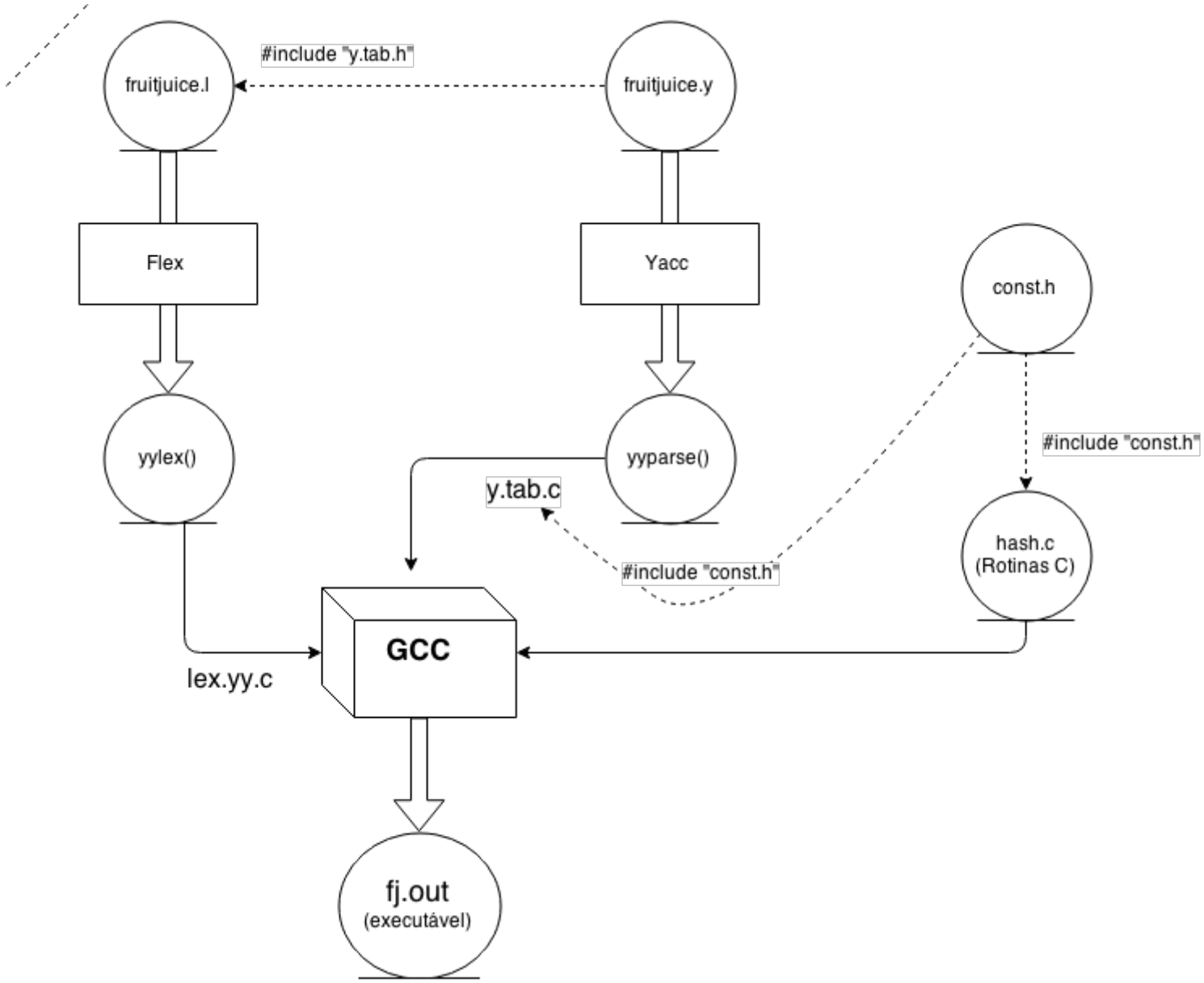

Figura 4.1: Diagrama que retrata a aquitetura/passos de compilação do compilador

Capítulo 5

Tabela de Símbolos

Tendo em vista o armazenamento das variáveis declaradas, para isso, implementamos uma tabela de hash, que nos permitisse, além de armazenar as informações das variáveis, um tempo de acesso constante.

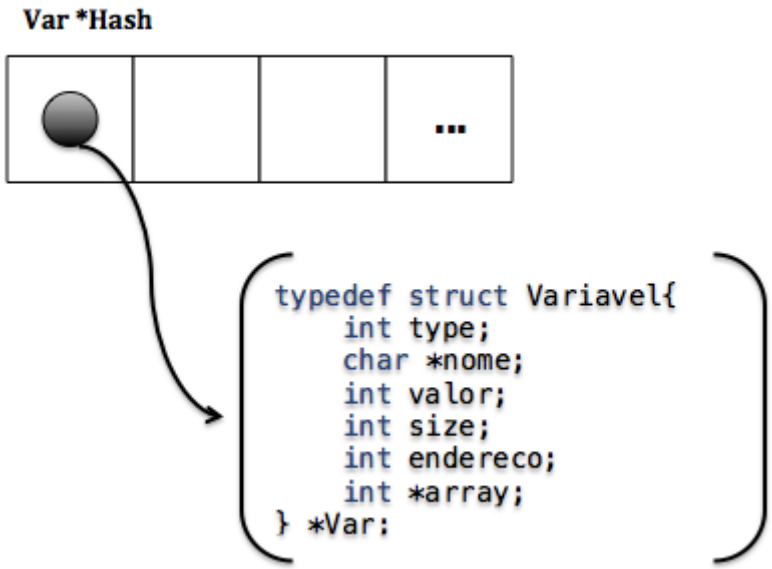

Figura 5.1: Diagrama que retrata a nossa tabela de hash

Como podemos observar na Figura 5.1, cada posição da hash desenvolvida possui:

- **int type** - representa o tipo da variável;
- **char *nome** - representa o nome da variável;
- **int valor** - representa o valor da variável, quando se trata de uma variável simples;

- **int size** - representa o tamanho da variável, 1 quando se trata de uma variável simples, n quanto for um array;
- **int endereco** - representa o endereço onde se encontra a variável;
- **int *array** - quando se trata de uma variável do tipo array, representa os valores presentes nesse array;

Este módulo, além de um conjunto de funções para inserir novas variáveis na tabela, possui também uma série de funções auxiliares que nos permitem obter a informação necessária para gerar o código da máquina virtual.
À a destacar as funções "containsValue", que nos permite verificar se uma dada variável está presente na tabela de hash, ou seja, se a variável está declarada. Também à a destacar as funções "findEnderecoA"e "findEnderecoV", que nos permitem dado o nome de uma variável obter o seu endereço na pilha. No caso da função "findEnderecoA", obtemos o endereço na pilha. de uma dada posição do array.

```
int findEnderecoA(char* nome, int pos){
    int key = getPos(nome);

    while(key < size && hash[key] != NULL){
        if(strcmp(hash[key]->nome,nome)==0)
            return (hash[key]->endereco + pos - 1);
        key++;
    }
    return 0;
}
```

Figura 5.2: Função que obtem o endereço de uma posição de um variável array

```
int findEnderecoV(char *nome){
    int key = getPos(nome);

    while(key < size && hash[key] != NULL){
        if(strcmp(hash[key]->nome,nome)==0)
            return hash[key]->endereco;
        key++;
    }
    return 0;
}
```

Figura 5.3: Função que obtem o endereço de uma variável simples

Capítulo 6

Análise Semântica

Neste capítulo iremos tentar particionar a nossa gramática, por forma a podermos na fase de implementação, dividir a mesma em pedaços pequenos, para tal, temos primeiro de compreender a dinâmica de gerar código para a máquina virtual. É neste momento que **especificamos e definimos o processamento de frases da linguagem**.

6.1 Inicialiação da tabela de símbolos

É necessário definir o instante em que se inicializa a estrutura de dados. Poderia ser na função **main()**, mas o grupo optou por inicializar a estrutura de dados a quando do reconhecimento do símbolo terminal FRUIT, como podemos ver no seguinte pedaço de código, a chamada de **initHash()**, é também imprimido para o ficheiro resposta [1] a string Start.

```
Linhas   : FRUIT { initHash(); } Declaracoes JUICE { fprintf(f, "START\n"); } Codigo
   |
   ;
```

6.2 Declarações & Atribuições

Quanto a declarações e atribuições, podem ocorrer sensivelmente duas coisas, ou a variável não é inicializada, ou é atribuído à partida pelo programador (tratemos por programador a entidade que interage com a linguagem) um determinado valor. O procedimento para declarações de variáveis é semelhante independentemente de a mesma ser simples, ou um array. Primeiro verificamos se a variável está declarada na tabela de símbolos,

```
Var : NOME {
if(containsValue($1)==1){
fprintf(stderr, "%s %s\n", ERR_REDECLARED, $1);
}
```

a função **containsValue(char* var)**, verifica se a variável já foi declarada e caso tenha sido, lança o erro de variável redeclarada, caso a variável ainda não exista no contexto de execução do programa,

[1] **a partir deste ponto iremos referir-nos ao ficheiro com o código gerado pelo compilador como ficheiro resposta**

teremos de inseri-la na tabela, e fazer um **PUSH 0** para a stack, ora **0** representa "lixo" neste caso, pois a variável não foi inicializada

```
else {
insertHashVSI(INT,$1);
fprintf(f, "PUSHI 0\n");
}
```

As restantes atribuições não passam agora de variações deste simples exemplo, mas destacamos a leitura de valores para um array.

6.2.1 Atribuições em arrays

Este é o processo de atribuição de valores mais complexo pelo que, um número é um símbolo terminal e terão por isso os elementos de arrays, ser processados um a um, o mesmo é dizer que temos de adoptar uma estratégia para inserção desses valores na nossa tabela de símbolos, associados a uma dada variável do tipo INT_ARRAY.

Declaramos então no topo do ficheiro algumas variáveis globais que nos ajudam nesse processo.

```
   char *array_nome; // Nome do array
   int *array; // Decl de um array com elementos, guarda elementos para posterior insercao na hash
   int nelems; // Decl de um array com elementos, representa o numero de elementos no array
   int size_array; // Init um array com valores, size_array fica com o tamanho desse array
```

De seguida o processo é o seguinte,

```
ListaNum : '{''}'
   | '{' ListaNums '}'
   ;
  ListaNums : NUMERO {
array = (int*) malloc(MAX_SIZE*sizeof(int));
nelems=0;
```

alocamos memória para um array de inteiros quando reconhecemos o primeiro número do vetor de elementos,

```
   (...)
array[nelems] = $1;
nelems++;
fprintf(f, "PUSHI %d\n", $1);
   }
   | ListaNums ',' NUMERO {
array[nelems] = $3;
nelems++;
fprintf(f, "PUSHI %d\n", $3);
   }
   ;
```

para os restantes elementos iremos apenas fazer a inserção do respetivo elemento no array, incrementar o número de elementos do array e finalmente escrever no ficheiro resposta o PUSH do respetivo valor.
Quando pararmos de de reconhecer números, iremos então proceder à inserção da variável do tipo INT_ARRAY na estrutura de dados, cujo o valor é aquele acumulado em **array**.

```
(...)
if(nelems>size_array){
   fprintf(stderr, "%s %d %s %d\n", ERR_ARRAY_OVERFLOW_1, size_array, ERR_ARRAY_OVERFLOW_2, nelems
}
else{
if(nelems==0){
   insertHashASI(INT_ARRAY,array_nome,size_array);
}
else {

   insertHashACI(INT_ARRAY,array_nome,size_array,array,nelems);

   free(array);
      }
}
// (Re)inicializacao de variaveis auxiliares
array=NULL;
array_nome=NULL;
size_array=0;
nelems=0;
(...)
```

Também de referir que quando o número de elementos inseridos ultrapassa o tamanho alocado ao array em questão, o compilador é "inteligente" o suficente para lançar o erro *stackoverflow*, devido a **tentativa de empilhamento em pilha cheia**.

6.2.2 Condições

Para as ações semânticas que permitem à liguagem exprimir condição, definimos uma estrutura de dados (**lista ligada**), que nos permita ter a noção de aninhamento e encadeamente à medida que reconhecemos instruções. O mecanismo é simples, baseia-se numa lista *FIFO* (**First in First Out**), ou seja quando é reconhecida uma condição é inserida nessa lista (**SC stackIf**) um novo elemento à cabeça. Quando estamos no bloco **else** ou num bloco **else if** removemos um elemento da lista ligada. A estrutra de dados que suporta as funcionalidades descritas é a seguinte:

```
typedef struct stackCondicao{
   int valor;
   struct stackCondicao *prox;
   } *SC;

   //auxiliares para contagem dos if's
```

```
   int prof = 0; // Variável auxiliar de contagem
   SC stackIf;
```

Aqui podemos observar as ações semânticas de geração de código associadas à **if else clause**:

```
If : IF { fprintf(f,"\n\\\\if then else\n"); insertCond(); }
   '(' Condicao ')'{ fprintf(f,"JZ senao%d \n",getHead());}
   '{' Codigo '}'{ fprintf(f,"JUMP fse%d \n",getHead());
   fprintf(f,"senao%d: NOP \n",getHead()); }
   ELSE '{' Codigo '}' {
   fprintf(f,"fse%d: NOP \n",getHead());
   removeHead();
   }
```

6.2.3 Ciclos

Por norma um ciclo é um bloco de instruções que deve ser executado um determinado número de vezes, até que se verifique um certo critério de paragem ou condição de paragem. Traduzido para assembly, teremos um dados bloco de instruções a executar, em que o respetivo bloco contém uma marca (identificador), para que no final desse bloco se posso regregressar ao início (JUMP). Passando à prática da implementação das acções semânticas, obtemos:

```
Ciclo : { fprintf(f,"\n\\\\while\n");
        fprintf(f,"ciclo%d: NOP \n",ciclo);
      } WHILE '(' Condicao ')' {
fprintf(f,"JZ fciclo%d \n",fciclo);
}
'{' Codigo '}' {
fprintf(f,"JUMP ciclo%d \n",ciclo);
ciclo++;
fprintf(f,"fciclo%d: NOP \n",fciclo);
fciclo++;
}
```

6.2.4 I/O

Nesta secção apresentamos de que forma foram definidas as ações semâticas de **I/O** ***Input Output***. As possibilidades e respetivas descrições das ações semâticas tanto para escrita como para leitura são semelhantes, passamos a explicitar as de escrita. Pode portanto escrita de:

- **Constante** - é feito apenas o push para a pilha, e escrita a instrução WRITEI;
- **Variável simples** - para estes casos é preciso invocar **findEnderecoV**(nome_da_variavel) fazer push e respetivas instruções de escrita;
- **Variável array** - ao detatarmos o tipo de variável array, invocamos a função **generateCodeA**(nome_variavel, descritor_ficheiro_resposta). Esta função foi implementada no módulo **hash.c** devido à sua dimensão. Essa função apenas percorre os valores desse array e gera as intruções de escrita em conformidade.

- **Valor de posição de array** - neste caso teremos de pesquisar o endereço associado à posição do array em questão e fazer PUSHI desse mesmo endereço. Fazemos a pesquisa do endereço a função através do método **findEnderecoA**(array, pos).

Eis o exemplo de gereção do código para escrita de uma variável array ou simples (o tipo de dados é testado internamente à acção semântica):

```
(...)
| WOUT '(' NOME ')'{
if(varSimple($3)==1 && containsValue($3)){
fprintf(f,"PUSHG %d\n",findEnderecoV($3));
fprintf(f,"WRITEI\n");
fprintf(f,"PUSHS \"\\n\"\n");
fprintf(f,"WRITES\n");
} else if (varSimple($3)==0 && containsValue($3)){
generateCodeA($3,f);
} else printf("ERRO : Variável %s não declarada\n",$3);
(...)
```

Capítulo 7

Analisador Léxico

É necessário implementar um analisador léxico que reconheça os símbolos terminais que definimos para nossa gramática, portanto implementamos em flex tal ferramenta. Relativamente ao uso das macros para símbolos terminais, é feito a inclusão do ficheiro **y.tab.h**, como já podemos observar pelo esquema da arquitetura do compilador neste relatório, é feito também o include do ficheiro **const.h** para uso do token de erro.

```
%{
    /*----------------------------------------
    Analisador L xico (AL) para a
    linguagem de programa  o Fruit & Juice
    ----------------------------------------*/
    #include <string.h>
    #include "y.tab.h"
%}

%%

[ \t\n] { ; }

(?i:FJBEGIN) { return (FJBEGIN); }
(?i:FRUIT) { return(FRUIT); }
(?i:JUICE) { return(JUICE); }
(?i:FJEND) { return(FJEND); }
(?i:IF) { return(IF); }
(?i:ELSE) { return(ELSE); }
(?i:WHILE) { return(WHILE); }
(?i:FOR) { return(FOR); }
(?i:WOUT) { return(WOUT); }
(?i:RINP) { return(RINP); }

"\{"     {return yytext[0];}
"\}"     {return yytext[0];}
"\("     {return yytext[0];}
```

```
"\)"      {return yytext[0];}
"\["      {return yytext[0];}
"\]"      {return yytext[0];}
"\="      {return yytext[0];}
"\<"      {return yytext[0];}
"\>"      {return yytext[0];}
"\,"      {return yytext[0];}
"\;"      {return yytext[0];}
"\""      {return yytext[0];}
"\+"      {return yytext[0];}
"\-"      {return yytext[0];}
"\*"      {return yytext[0];}
"\/"      {return yytext[0];}
"\%"      {return yytext[0];}
"\&"      {return yytext[0];}
"\|"      {return yytext[0];}
"\!"      {return yytext[0];}


[a-zA-Z][a-zA-Z0-9]* { yylval.vstr = strdup(yytext); return(NOME); }
[0-9]+ { yylval.vint = atoi(yytext); return(NUMERO); }

. { return(ERRO); }

%%

int yywrap() {
   return(1);
}
```

Relativamente à passagem de valores do flex para o yacc, doi declarada um union a seguir ao bloco de inicializações do ficheiro **fruitjuice.y**, a mesma union tem a seguinte estrutura, **e aproveitando desde já para fazer referência aos tokens e sua tipagem**:

```
%union{
int vint;
char* vstr;
char vchar;
}
%token NOME NUMERO FJBEGIN FJEND FRUIT JUICE IF ELSE WHILE FOR WOUT RINP ERRO
%type <vstr> NOME
%type <vint> NUMERO
%type <vchar> Oper
%type <vchar> OperAdicao
%type <vchar> OperMultiplicacao
%type <vint> Valor
```

Capítulo 8

Apresentação de resultados

Nesta secção do relatório iremos apresentar uma série de resultados obtidos, apresentando uma série de programas exemplos na linguagem criada, e o respetico código assembly. Os programas serão expostos por ordem crescente de complexidade.

8.1 Teste 1

Codigo

```
FJBEGIN
FRUIT
x=1;
y=0;
w;
JUICE
w=x+y;
w=x/y;
FJEND
```

Resultado

```
PUSHI 1
PUSHI 0
PUSHI 0
START
PUSHG 0
PUSHG 1
ADD
STOREG 2
PUSHG 0
PUSHG 1
DIV
```

```
STOREG 2
STOP
```

8.2 Teste 2

Codigo

```
FJBEGIN
FRUIT
x[13]={12,14,16,18,20,2,3,1,4};
b=0;
JUICE
IF(x[4] == 20) {WOUT(x);} ELSE{RINP(b);}
IF(!(12==13)) {WOUT(x[2]);} ELSE {b=x[3];}
FJEND
```

Resultado

```
PUSHI 12
PUSHI 14
PUSHI 16
PUSHI 18
PUSHI 20
PUSHI 2
PUSHI 3
PUSHI 1
PUSHI 4
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START

\\if then else
PUSHGP
PUSHI 3
PADD
PUSHI 4
LOADN
PUSHI 20
EQUAL
JZ senao1
PUSHS "]"
PUSHG 12
STRI
PUSHS ","
```

```
PUSHG 11
STRI
PUSHS ","
PUSHG 10
STRI
PUSHS ","
PUSHG 9
STRI
PUSHS ","
PUSHG 8
STRI
PUSHS ","
PUSHG 7
STRI
PUSHS ","
PUSHG 6
STRI
PUSHS ","
PUSHG 5
STRI
PUSHS ","
PUSHG 4
STRI
PUSHS ","
PUSHG 3
STRI
PUSHS ","
PUSHG 2
STRI
PUSHS ","
PUSHG 1
STRI
PUSHS ","
PUSHG 0
STRI
PUSHS "["
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
```

```
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
WRITES
JUMP fse1
senao1: NOP
READ
ATOI
STOREG 13
fse1: NOP

\\if then else
PUSHI 12
PUSHI 13
EQUAL
NOT
JZ senao2
PUSHGP
PUSHI 1
PADD
PUSHI 2
LOADN
WRITEI
PUSHS "\n"
WRITES
JUMP fse2
senao2: NOP
PUSHGP
PUSHI 2
PADD
PUSHI 3
LOADN
STOREG 13
fse2: NOP
STOP
```

8.3 Teste 3

Codigo

```
FJBEGIN
FRUIT
asa[5]={1,2,4,5}, b, c=4, d[122];
w;
a=20;
x[10]={1,2,3,4,5};
e=10;
JUICE
IF(x[5]>2)
          {WHILE(1+1==2) {a=a+20;} a=a+1;}
          ELSE {IF(e>=2)
                              {a=3;}
                ELSE {WOUT(asa);}
          }
RINP(b);
e=5;
d[12]=5;
WOUT(asa);
asa[2] = 1+1 ;
c= c * 4;
e= 44 + 2;
FJEND
```

Resultado

```
PUSHI 1
PUSHI 2
PUSHI 4
PUSHI 5
PUSHI 0
PUSHI 0
PUSHI 4
PUSHN 122
PUSHI 0
PUSHI 20
PUSHI 1
PUSHI 2
PUSHI 3
PUSHI 4
PUSHI 5
PUSHI 0
PUSHI 0
PUSHI 0
```

```
PUSHI 0
PUSHI 0
PUSHI 10
START

\\if then else
PUSHGP
PUSHI 135
PADD
PUSHI 5
LOADN
PUSHI 2
SUP
JZ senao1

\\while
ciclo1: NOP
PUSHI 1
PUSHI 1
ADD
PUSHI 2
EQUAL
JZ fciclo1
PUSHG 130
PUSHI 20
ADD
STOREG 130
JUMP ciclo1
fciclo1: NOP
PUSHG 130
PUSHI 1
ADD
STOREG 130
JUMP fse1
senao1: NOP

\\if then else
PUSHG 141
PUSHI 2
SUPEQ
JZ senao2
PUSHI 3
STOREG 130
JUMP fse2
senao2: NOP
PUSHS "]"
PUSHG 4
STRI
```

```
PUSHS ","
PUSHG 3
STRI
PUSHS ","
PUSHG 2
STRI
PUSHS ","
PUSHG 1
STRI
PUSHS ","
PUSHG 0
STRI
PUSHS "["
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
WRITES
fse2: NOP
fse1: NOP
READ
ATOI
STOREG 5
PUSHI 5
STOREG 141
PUSHGP
PUSHI 18
PADD
PUSHI 12
LOADN
PUSHI 5
STOREN
PUSHS "]"
PUSHG 4
STRI
PUSHS ","
PUSHG 3
STRI
PUSHS ","
PUSHG 2
STRI
PUSHS ","
```

```
PUSHG 1
STRI
PUSHS ","
PUSHG 0
STRI
PUSHS "["
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
CONCAT
WRITES
PUSHGP
PUSHI 1
PADD
PUSHI 2
LOADN
PUSHI 1
PUSHI 1
ADD
STOREN
PUSHG 6
PUSHI 4
MUL
STOREG 6
PUSHI 44
PUSHI 2
ADD
STOREG 141
STOP
```

8.4 Teste 4

Codigo

```
FJBEGIN
FRUIT
x=10;
y=0;
i,z;
JUICE
for(i=0;i<x;i=i+1) {
```

```
        for (z=x; z>0; z=z-1){
                y=y+1;
        }
}
FJEND


Resultado

PUSHI 10
PUSHI 0
PUSHI 0
PUSHI 0
START

\\for
ciclo1: NOP
PUSHI 0
STOREG 2
PUSHG 2
PUSHG 0
INF
JZ fciclo1
PUSHG 2
PUSHI 1
ADD
STOREG 2

\\for
ciclo1: NOP
PUSHG 0
STOREG 3
PUSHG 3
PUSHI 0
SUP
JZ fciclo1
PUSHG 3
PUSHI 1
SUB
STOREG 3
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP ciclo1
fciclo1: NOP
JUMP ciclo2
fciclo2: NOP
```

STOP

Capítulo 9

Extra - Desenho da stack

O **desenhado das stacks de inialização** que foi visto no primeiro exemplo do capítulo anterior, trata-se de uma funcionalidade acrescentada pelo grupo ao compilador. Criamos uma função que executa o seguinte algoritmo para desenho da **stack**:

- Cópia da tabela de símbolos (hash) para um array sem posições nulas;
- Ordenação do array obtido, usando o algoritmo bubble sort;
- Criar a stack dinamicamento com o package **drawstack** do LaTeX;

As funções que perforam as tarefas descritas, **OrderHashEnd()** é invocada pelo exterior, **void writeLatexFancyStack(Var* v)**, podem ser consultadas no código em anexo a este relatório, mais concretamente no módulo **hash.c**.

O programa com as inicializações:
v[4]=1,2; x=3; y=1;
gera o desenho da stack na seguinte imagem.

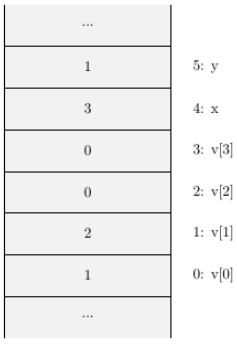

Figura 9.1: Exemplo de uma stack de inicialização gerada.

Capítulo 10

Conclusões

Com último trabalho prático da UC de Processamento de Linguagens conseguimos obter a noção da implementação de um compilador para uma linguagem de programação bem como a definição formal de uma linguagem e a geração de código assembly respetiva. Esta foi uma abordagem simples mas de certa forma genérica que se pode transportar para outras realidades complexas. Novamente sentimos uma forte consolidação da implementação de programas na linguagem C, programação modular, e uso de ferramentas em ambiente Linux.

Capítulo 11

Anexos

11.1 Analisador Léxico

O analisador léxico já se encontra no presente relatório no capítulo **7. Analisador Léxico**.

11.2 fruitjuice.y

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

    extern int yylex();
    int yyerror(char* s);

    // Vari veis globais
    FILE *f; // Ficheiro que ir  conter o resultado final do programa, o assembly gera

    char *array_nome; // Nome do array

    /* A quando da declaracao de um array com elementos,
         esta variavel faz cache desses elementos para posterior
         insercao na tabela de simbolos */
    int *array;

    /* A quando da declaracao de um array com elementos,
         representa o numero de elementos no array */
    int nelems;

    /* Sempre que    inicializado um array com valores,
         size_array fica com o tamanho desse array */
```

```
    int size_array;

    typedef struct stackCondicao{
            int valor;
            struct stackCondicao *prox;
    } *SC;

    //auxiliares para contagem dos if's
    int prof = 0;
    SC stackIf;

    //auxiliares para contagem dos ciclos
    int ciclo=1;
    int fciclo=1;

    //insere o nivel atual na cabe a da lista auxif
    void insertCond(){
            prof++;
            SC novo;
            novo = (SC)malloc(sizeof(struct stackCondicao));
            novo->valor = prof;
            novo->prox = NULL;

            if( stackIf == NULL) stackIf = novo;
            else {
                    novo->prox = stackIf;
                    stackIf = novo;
            }

    }

    // Remove a cabe a da stack de condicoes
    void removeHead(){
            SC novo;
            novo = stackIf->prox;
            free(stackIf);
            stackIf = novo;
    }

    // Obtem o nivel que estiver na cabe a da stack de condicoes
    int getHead(){
            if(stackIf == NULL) return -1;
            else return(stackIf->valor);
    }

%}

%union{
```

```
        int vint;
        char* vstr;
        char vchar;
}
%token NOME NUMERO FJBEGIN FJEND FRUIT JUICE IF ELSE WHILE FOR WOUT RINP ERRO
%type <vstr> NOME
%type <vint> NUMERO
%type <vchar> Oper
%type <vchar> OperAdicao
%type <vchar> OperMultiplicacao
%type <vint> Valor

%%
  Programa : FJBEGIN Linhas FJEND
           ;
  Linhas : FRUIT { initHash(); } Declaracoes JUICE
                        { fprintf(f, "START\n"); } Codigo
           |
           ;
  Declaracoes :
           | ListaDeclaracoes
           ;
  ListaDeclaracoes : ListaDeclaracoes ListaVar ';'
           | ListaVar ';'
           ;
  ListaVar : Var
           | ListaVar ',' Var
           ;
  Var : NOME {
                printf("containsValue(\"%s\")\n", $1);
                if(containsValue($1)==1){
                        fprintf(stderr, "%s %s\n", ERR_REDECLARED, $1);
                }
                else {
                        insertHashVSI(INT,$1);
                        fprintf(f, "PUSHI 0\n");
                }
           }
                | NOME '=' NUMERO {
                printf("containsValue(\"%s\")\n", $1);
                if(containsValue($1)==1){
                        fprintf(stderr, "%s %s\n", ERR_REDECLARED, $1);
                }
                else {
                        insertHashVCI(INT,$1,$3);
                        fprintf(f, "PUSHI %d\n", $3);
                }
           }
```

```
        | NOME '[' NUMERO ']' {
                printf("containsValue(\"%s\")\n", $1);
            if(containsValue($1)){
                    fprintf(stderr, "%s %s\n", ERR_REDECLARED, $1);
            }
            else {
                    insertHashASI(INT_ARRAY,$1,$3);
                    fprintf(f, "PUSHN %d\n", $3);
            }
        }
        | NOME '[' NUMERO ']' '=' ListaNum {
            if(containsValue($1)){
                    fprintf(stderr, "%s %s\n", ERR_REDECLARED, $1);
            }
            else {
              array_nome=$1;
              size_array=$3;
              printf("containsValue(\"%s\")\n", $1);

              int i;
              for(i=0; i<(size_array-nelems); i++){
                    fprintf(f, "PUSHI 0\n");
                    // Alocar espa o na stack para o array
              }

              if(nelems>size_array){
                    fprintf(stderr, "%s %d %s %d\n", ERR_ARRAY_OVERFLOW_1, size_ar
              }
              else{
                    if(nelems==0){
                            insertHashASI(INT_ARRAY,array_nome,size_array);
                    }
                    else {
                            insertHashACI(INT_ARRAY,array_nome,size_array,array,ne
                            free(array);
                            }
                    }
                    // (Re)inicializacao de variaveis auxiliares
                    array=NULL;
                    array_nome=NULL;
                    size_array=0;
                    nelems=0;
            }
        }
        ;
ListaNum : '{''}'
        | '{' ListaNums '}'
        ;
```

```
ListaNums : NUMERO {
        // Alocar array para armazenar valores na produ  o ListaNums
              array = (int*) malloc(MAX_SIZE*sizeof(int));
              nelems=0;

              array[nelems] = $1;
              nelems++;
              fprintf(f, "PUSHI %d\n", $1);
         }
         | ListaNums ',' NUMERO {
              array[nelems] = $3;
              nelems++;
              fprintf(f, "PUSHI %d\n", $3);
         }
         ;
Codigo : Lines
               |
               ;
Lines : Linha
              | Lines Linha
              ;
Linha :Print ';'
              |If
              |Ciclo
              |Atribuicao ';'
         ;
Atribuicao : NOME '=' Expr     {
      if(containsValue($1))
              fprintf(f,"STOREG %d\n",findEnderecoV($1));
      else
              fprintf(stderr, "%s %s\n", ERR_REDECLARED, $1);
      }
      | NOME '[' NUMERO ']' '=' {
      if(containsValue($1) && validPos($1,$3)){
              fprintf(f,"PUSHGP\n");
              fprintf(f,"PUSHI %d \n",findEnderecoA($1,$3));
              fprintf(f,"PADD\n");
              fprintf(f,"PUSHI %d\n",$3);
              fprintf(f,"LOADN\n");
      } else if(!containsValue($1)) fprintf(stderr, "%s %s\n", ERR_REDECLARED, $1);
      else fprintf(stderr, "erro. posi  o %d n o existe no array %s\n",$3,$1);
      } Expr {fprintf(f,"STOREN\n");}
      | RINP '(' NOME ')'     {
      if(containsValue($3)) {
              fprintf(f,"READ\n");
              fprintf(f,"ATOI\n");
              fprintf(f,"STOREG %d\n",findEnderecoV($3));
      } else printf("ERRO : Variavel %s n o declarada\n",$3);}
```

```
        | RINP '(' NOME '[' NUMERO ']' ')'       {
        if(containsValue($3) && validPos($3,$5)){
                fprintf(f,"PUSHGP\n");
                fprintf(f,"PUSHI %d\n",findEnderecoA($3,$5));
                fprintf(f,"PADD\n");
                fprintf(f,"PUSHI %d\n",$5);
                fprintf(f,"READ\n");
                fprintf(f,"ATOI\n");
                fprintf(f,"STOREN\n");}
        else if(!containsValue($3))
                printf("ERRO : Vari vel %s n o declarada\n",$3);
        else
                printf("ERRO : Posi   o %d n o existe no array %s\n",$5,$3);
        }
        ;
If : IF { fprintf(f,"\n\\\\if then else\n"); insertCond(); }
           '(' Condicao ')'{ fprintf(f,"JZ senao%d \n",getHead());}
           '{' Codigo '}'{ fprintf(f,"JUMP fse%d \n",getHead());
                   fprintf(f,"senao%d: NOP \n",getHead()); }
           ELSE '{' Codigo '}' {
                   fprintf(f,"fse%d: NOP \n",getHead());
                   removeHead();
           }
         | IF {fprintf(f,"\n\\\\if then else\n"); insertCond();}
                '(' Condicao ')'{ fprintf(f,"JZ senao%d \n",getHead());}
                '{' Codigo '}' { fprintf(f,"JUMP fse%d \n",getHead());
                        fprintf(f,"senao%d: NOP \n",getHead()); removeHead(); }
        ;
Condicao : Expr '<' Expr                {fprintf(f,"INF\n");}
         | Expr '=' '<' Expr            {fprintf(f,"INFEQ\n");}
         | Expr '>' Expr                        {fprintf(f,"SUP\n");}
         | Expr '>' '=' Expr            {fprintf(f,"SUPEQ\n");}
         | Expr '=' '=' Expr            {fprintf(f,"EQUAL\n");}
         | Expr '!' '=' Expr            {fprintf(f,"NOT EQUAL\n");}
         | '!' '(' Condicao ')'         {fprintf(f,"NOT\n");}
         ;
Ciclo : { fprintf(f,"\n\\\\while\n");
                fprintf(f,"ciclo%d: NOP \n",ciclo);
             } WHILE '(' Condicao ')' {
                                fprintf(f,"JZ fciclo%d \n",fciclo);
                        }
                        '{' Codigo '}' {
                                fprintf(f,"JUMP ciclo%d \n",ciclo);
                                ciclo++;
                                fprintf(f,"fciclo%d: NOP \n",fciclo);
                                fciclo++;
                        }
               |{ fprintf(f,"\n\\\\for\n");
```

```
                        fprintf(f,"ciclo%d: NOP \n",ciclo);
                } FOR '(' Atribuicao ';' Condicao {
                        fprintf(f,"JZ fciclo%d \n",fciclo);
                }';' Atribuicao ')' '{' Codigo '}' {
                        fprintf(f,"JUMP ciclo%d \n",ciclo);
                        ciclo++;
                        fprintf(f,"fciclo%d: NOP \n",fciclo);
                        fciclo++;
                }
        ;
Expr : Valor
        | Expr Oper Valor {
                switch($2){
                        case '+' :
                                fprintf(f,"ADD\n");
                                break;
                        case '-' :
                                fprintf(f,"SUB\n");
                                break;
                        case '*' :
                                fprintf(f,"MUL\n");
                                break;
                        case '/' :
                                // Verificar tentativa de divis o por 0
                                if($3==0){
                                        fprintf(stderr, "%s\n", ERR_DIVISION_BY_0);
                                }
                                else{
                                   fprintf(f,"DIV\n");
                                }
                                break;
                        case '%' :
                                fprintf(f,"MOD\n");
                                break;
                        case '|' :
                                fprintf(f,"OR\n");
                                break;
                        case '&' :
                                fprintf(f,"AND\n");
                                break;
                }
        }
        ;
Valor : NUMERO { $$=$1;fprintf(f,"PUSHI %d\n",$1); }
        | NOME      {
                if(containsValue($1)){
                        fprintf(f,"PUSHG %d\n",findEnderecoV($1));
                } else printf("ERRO : Vari vel %s n o declarada\n",$1);}
```

```
            | NOME '[' NUMERO ']'         {
                    if(containsValue($1) && validPos($1,$3)){
                            fprintf(f,"PUSHGP\n");
                            fprintf(f,"PUSHI %d\n",findEnderecoA($1,$3));
                            fprintf(f,"PADD\n");
                            fprintf(f,"PUSHI %d\n",$3);
                            fprintf(f,"LOADN\n");
                    } else if(!containsValue($1))
                            printf("ERRO : Vari vel %s n o declarada\n",$1);
                    else
                        printf("ERRO : Posi  o %d n o existe no array %s\n",$3,$1);
            }
            ;
  Oper : OperAdicao                         {$$=$1;}
            | OperMultiplicacao  {$$=$1;}
            ;
  OperAdicao : '+'      {$$='+';}
            | '-'                   {$$='-';}
            | '|'                   {$$='|';}
            ;
  OperMultiplicacao : '*'       {$$='*';}
            | '/'                                       {$$='/';}
            | '%'                                       {$$='%';}
            | '&'                                       {$$='&';}
            ;
Print : WOUT '(' NUMERO ')' {
                        fprintf(f,"PUSHI %d\n",$3);
                        fprintf(f,"WRITEI\n");
                        fprintf(f,"PUSHS \"\\n\"\n");
                        fprintf(f,"WRITES\n");
                }
                | WOUT '(' NOME ')'     {
                        if(varSimple($3)==1 && containsValue($3)){
                                fprintf(f,"PUSHG %d\n",findEnderecoV($3));
                                fprintf(f,"WRITEI\n");
                                fprintf(f,"PUSHS \"\\n\"\n");
                                fprintf(f,"WRITES\n");
                        } else if (varSimple($3)==0 && containsValue($3)){
                                generateCodeA($3,f);
                        } else printf("ERRO : Vari vel %s n o declarada\n",$3);
                }
                | WOUT '(' NOME '[' NUMERO ']' ')'      {
                        if(containsValue($3) && validPos($3,$5)){
                                fprintf(f,"PUSHGP\n");
                                fprintf(f,"PUSHI %d\n",findEnderecoA($3,$5));
                                fprintf(f,"PADD\n");
                                fprintf(f,"PUSHI %d\n",$5);
                                fprintf(f,"LOADN\n");
```

```
                                fprintf(f,"WRITEI\n");
                                fprintf(f,"PUSHS \"\\n\"\n");
                                fprintf(f,"WRITES\n");
                        } else if(!containsValue($3))
                                printf("ERRO : Vari vel %s n o declarada\n",$3);
                        else printf("ERRO : Posi  o %d n o existe no array %s\n",$5
                }
        ;

%%

int yyerror(char *s)
{
   fprintf(stderr, "ERRO: %s\n", s);
   return 0;
}

int main()
{
   f = fopen("assembly.vm", "w");

   yyparse();

   fprintf(f,"STOP\n");
   fclose(f);

   printHash();

   orderHashEnd();
   system("pdflatex stack.tex");
   system("open stack.pdf");
   system("clear");

   return(0);
}
```

11.3 hash.h

```
# ifndef HASH
# define HASH

/**Imprimir tabela de s mbolos para stdout*/
void printHash();

/** Inicia uma hash com 10 posicoes */
void initHash();

/** Calcula a posicao na hash */
```

```
int getPos(char *nome);

/** Insere na Hash uma variavel array sem inicializacao */
void insertHashASI(int type, char *nome, int size);

/** Insere na Hash uma variavel array com inicializacao */
void insertHashACI(int type,char *nome, int size, int *a, int nelems);

/** Insere uma varivel sem inicializacao */
void insertHashVSI(int type, char * nome );

/* Insere uma varivel com inicializacao */
void insertHashVCI(int type, char *nome, int valor);

/** Gera codigo assembly para uma variavel array*/
void generateCodeA(char *nome, FILE *fp);

/** Verifica se uma dada variavel esta declarada na hash */
int containsValue(char *nome);

/** Verifica se a variavel e simples */
int varSimple(char *nome);

/** Dado o nome de uma variavel array vai buscar a posicao de um elemento seu na pilha
int findEnderecoA(char* nome, int pos);

/** Dado o nome de uma variavel vai buscar o seu tamanho */
int getVarSize(char *nome);

/** Dado o nome de uma variavel vai buscar a sua posicao na pilha */
int findEnderecoV(char *nome);

/** Sabendo que a variavel existe e que a posi ao que estamos a tentar aceder existe
int getArrValue(char *nome, int posicao);

/** Verifica se a posicao do array que estamos a tentar aceder e valida */
int validPos(char *nome, int posicao);

/** Sabendo que a variavel existe vai buscar o valor da variavel */
int getVarValue(char *nome);

void orderHashEnd();

# endif
```

11.4 hash.c

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

```
#include <stdlib.h>
#include <string.h>
#include "hash.h"
#include "const.h"

#define MIN_SIZE 10
#define BASE 10
#define FALSE 0
#define TRUE 1

// Estrutura de dados para a tabela de simbolos
typedef struct Variavel{
        int type;
        char *nome;
        int valor;
        int size;
        int endereco;
        int *array;
} *Var;


Var *hash; // Variavel global da tabela de simbolos
int size = 0; // Tamanho da hash ou numero de simbolos na tabela de simbolos
int sp = 0; // Apontador para o proximo endereco livre da stack

void printHash()
{
   printf("\n\n## Tabela de S mbolos ##\n");
   int i=0;
   int j;
   for(; i<size; i++){
        if(hash[i]!=NULL){
                if(hash[i]->type==INT){
                        printf("[%s] [%d]\n", hash[i]->nome, hash[i]->valor);
                }
                else if(hash[i]->type==INT_ARRAY){
                        printf("[%s] [size:%d]: ", hash[i]->nome, hash[i]->size);
                        printf(" [ ");
                        for(j=0; j<hash[i]->size && hash[i]->array[j]!=-1; j++){
                           printf("%d ", hash[i]->array[j]);
                        }
                        printf("]\n");
                }
        }
   }
}

/**
```

```
* Fun  o de inicializa  o da tabela de s mbolos
*/
void initHash() {
        int i;

        size = MIN_SIZE;

        hash = (Var*) malloc(MIN_SIZE * sizeof(Var));

        for (i = 0; i<MIN_SIZE; i++){
                hash[i] = NULL;
        }
}

/**
* Fun  o que calcula posicao de variavel na hash e devolve inteiro
* @param nome, string com nome da variavel
* @return inteiro, posicao da variavel na hash
*/
int getPos(char *nome) {

        int n = strlen(nome);
        int i, hash = 0;

        for(i = 0; i < n; i++){
                hash += nome[i] * BASE^(n-1-i);
        }

        return hash;
}

static void hashAlloc (int tam) {
        Var *aux = hash;
        hash = (Var *)malloc(sizeof(Var *)*(tam+1));
        int i;
        for (i=0;i<size;i++) hash[i] = aux[i];
        free(aux);
}

/**
* Insere na hash uma variavel array sem inicializacao
* @param type, c digo que representa o tipo da vari vel
* @param nome, nome da varivel (a ser)
* @param tam, tamanho com que foi declarado o array
*/
void insertHashASI(int type, char *nome, int tam) {
        int i=0;
        int p;
```

```
        p = getPos(nome);

        if(p < size){
                if(hash[p] == NULL){
                        hash[p] = (Var)malloc(sizeof(struct Variavel));
                } else {
                        while (p < size && hash[p] != NULL) {
                                p++;
                        }
                        if (p>size) {
                                /*
                                hash = (Var *)realloc(hash, (p+1) * sizeof(Var));*/
                                hashAlloc(p);
                                for (i = size+1; i <= (p+1); i++){
                                        hash[i] = NULL;
                                }
                                hash[p] = (Var)malloc(sizeof(struct Variavel));
                                size = p+1;
                        } else {
                                hash[p] = (Var)malloc(sizeof(struct Variavel));
                        }
                }
        } else {
                //hash = (Var *)realloc(hash, (p+1) * sizeof(Var));
                hashAlloc(p);
                for (i = size+1; i <= (p+1); i++){
                        hash[i] = NULL;
                }
                hash[p] = (Var)malloc(sizeof(struct Variavel));
                size = p+1;
        }

        hash[p]->array = (int *)malloc(tam * sizeof(int));

        hash[p]->type = type;
        hash[p]->nome = nome;
        hash[p]->size = tam;
        hash[p]->endereco = sp;
        sp = sp + hash[p]->size;
        for(i = 0; i < tam; i++){
                hash[p]->array[i] = -1;
        }
}

/*
* Insere na Hash uma variavel array com inicializacao
* @param type, tipo da variavel
```

```
* @param nome, nome da variavel
* @param tam, tamanho com que foi declarado o array (porque pode diferir de nelems!)
* @param a, o array
* @param nelems, numero de elementos inseridos no array
*/
void insertHashACI(int type, char *nome, int tam, int *a, int nelems)
{
        int i = 0;
        int p;

        p = getPos(nome);
        if(p < size){
                if(hash[p] == NULL){
                        hash[p] = (Var)malloc(sizeof(struct Variavel));
                } else {
                        while (p < size && hash[p] != NULL) {
                                p++;
                        }
                        if (p>size) {
                                //hash = (Var *)realloc(hash, (p+1) * sizeof(Var));
                                hashAlloc(p);
                                for (i = size+1; i <= (p+1); i++){
                                        hash[i] = NULL;
                                }
                                hash[p] = (Var)malloc(sizeof(struct Variavel));
                                size = p+1;
                        } else {
                                hash[p] = (Var)malloc(sizeof(struct Variavel));
                        }
                }
        } else {
                //hash = (Var *)realloc(hash, (p+1) * sizeof(Var));
                hashAlloc(p);
                for (i = size+1; i <= (p+1); i++){
                        hash[i] = NULL;
                }
                hash[p] = (Var)malloc(sizeof(struct Variavel));
                size = p+1;
        }

        hash[p]->type = type;
        hash[p]->nome = nome;
        hash[p]->size = tam;
        hash[p]->endereco = sp;
        sp = sp + hash[p]->size;

        hash[p]->array = (int *)malloc(tam * sizeof(int));
        for(i = 0; i < nelems; i++){
```

```
                hash[p]->array[i] = a[i];
        }
        for(i = nelems; i < size; i++){
                hash[p]->array[i] = -1;
        }
}

/**
 * Insere uma varivel sem inicializacao
 * @param type, o tipo de dados
 * @param nome, nome da variavel
 */
void insertHashVSI(int type, char * nome) {
        int i = 0;
        int p = getPos(nome);

        if(p < size){
                if(hash[p] == NULL){
                        hash[p] = (Var)malloc(sizeof(struct Variavel));
                } else {
                        while (p < size && hash[p] != NULL) {
                                p++;
                        }
                        if (p>size) {
                                //hash = (Var *)realloc(hash, (p+1) * sizeof(Var));
                                hashAlloc(p);
                                for (i = size+1; i <= (p+1); i++){
                                        hash[i] = NULL;
                                }
                                hash[p] = (Var)malloc(sizeof(struct Variavel));
                                size = p+1;
                        } else {
                                hash[p] = (Var)malloc(sizeof(struct Variavel));
                        }
                }
        } else {
                //hash = (Var *)realloc(hash, (p+1) * sizeof(Var));
                hashAlloc(p);
                for (i = size+1; i <= (p+1); i++){
                        hash[i] = NULL;
                }
                hash[p] = (Var)malloc(sizeof(struct Variavel));
                size = p+1;
        }

        hash[p]->type = type;
        hash[p]->nome = nome;
        hash[p]->valor = 0;
```

```
        hash[p]->size = 1;
        hash[p]->endereco = sp;
        hash[p]->array = NULL;

        sp = sp + hash[p]->size;
}

/**
 * Insere uma varivel com inicializacao
 * @param type, tipo de dados
 * @param nome, o nome da variavel
 * @param valor, o valor atribuido a variavel
*/
void insertHashVCI(int type, char *nome, int valor) {

        int i = 0;
        int p = getPos(nome);

        if(p < size){
                if(hash[p] == NULL){
                        hash[p] = (Var)malloc(sizeof(struct Variavel));
                } else {
                        while (p < size && hash[p] != NULL) {
                                p++;
                        }
                        if (p>size) {
                                //hash = (Var *)realloc(hash, (p+1) * sizeof(Var));
                                hashAlloc(p);
                                for (i = size+1; i <= (p+1); i++){
                                        hash[i] = NULL;
                                }
                                hash[p] = (Var)malloc(sizeof(struct Variavel));
                                size = p+1;
                        } else {
                                hash[p] = (Var)malloc(sizeof(struct Variavel));
                        }
                }
        } else {
                //hash = (Var *)realloc(hash, (p+1) * sizeof(Var));
                hashAlloc(p);
                for (i = size+1; i <= (p+1); i++){
                        hash[i] = NULL;
                }
                hash[p] = (Var)malloc(sizeof(struct Variavel));
                size = p+1;
        }

        hash[p]->type = type;
```

```
        hash[p]->nome = nome;
        hash[p]->valor = valor;
        hash[p]->size = 1;
        hash[p]->endereco = sp;
        hash[p]->array = NULL;
        sp = sp + hash[p]->size;
}

/**
 * Dado um array gera o c digo para a VM
 * @param lista, array de valores
 * @param posicao, endereco base do array
 * @param size, tamanho do array
 * @param fp, descritor do ficheiro de destino
*/
void writeToFile(int *lista, int posicao, int size, FILE *fp){
        int nconcats = 0;
        fprintf(fp,"PUSHS \"]\"\n"); nconcats++;
        int i = posicao + size-1;

        while(i > posicao){
                fprintf(fp,"PUSHG %d\n",i); nconcats++;
                fprintf(fp,"STRI\n");
                fprintf(fp,"PUSHS \",\"\n"); nconcats++;
                i--;
        }
        fprintf(fp,"PUSHG %d\n",i); nconcats++;
        fprintf(fp,"STRI\n");
        fprintf(fp,"PUSHS \"[\"\n"); nconcats++;

        nconcats--;
        while(nconcats>0){
                fprintf(fp,"CONCAT\n");
                nconcats--;
        }
        fprintf(fp,"WRITES\n");
}

/*
 * Gera as instru  es relativas    constru  o do array para a VM
 * @param nome, nome da variavel array
 * @param fp, descritor do ficheiro destino
*/
void generateCodeA(char *nome, FILE *fp){
        int key = getPos(nome);

        while(key < size && hash[key] != NULL) {
                if(strcmp(hash[key]->nome,nome) == 0){
```

```
                        writeToFile(hash[key]->array, hash[key]->endereco, hash[key]->
                }
                key++;
        }
}

/**
 * Verifica se uma dada variavel esta declarada na hash
 * @param nome, nome da variavel
 * @return FALSE se n o existir TRUE se existir
*/
int containsValue(char *nome){
        int key = getPos(nome);
        int res = FALSE;

        if(hash[key] == NULL) { return FALSE; }
        else {
                while( res == 0 && key < size && hash[key] != NULL) {
                        if(strcmp(hash[key]->nome,nome) == 0)
                                res = TRUE;
                        key++;
                }
        }
        return res;
}

/*
 * Verifica se a variavel e simples
 * @param nome, nome da variavel
 * @return TRUE caso seja simples, FALSE caso contrario
*/
int varSimple(char *nome){
        int key = getPos(nome);

        while(key < size && hash[key] != NULL){
                if(strcmp(hash[key]->nome,nome)==0)
                        return (hash[key]->array == NULL);
                key++;
        }
        return FALSE;
}

/*
 * Dado o nome de uma variavel array vai buscar o endereco de um elemento seu,
na pilha
 * @param nome, nome da variavel array
 * @param pos, posicao sobre a qual queremos obter endereco
 * @return endereco da variavel
```

```
 */
int findEnderecoA(char* nome, int pos){
        int key = getPos(nome);

        while(key < size && hash[key] != NULL){
                if(strcmp(hash[key]->nome,nome)==0)
                        return (hash[key]->endereco + pos - 1);
                key++;
        }
        return 0;
}

/*
 * Dado o nome de uma variavel vai buscar o seu tamanho
 * @param nome, nome da variavel
 * @return tamanho da variavel (1 caso seja variavel simples, o tamanho do array caso
 */
int getVarSize(char *nome){
        int key = getPos(nome);
        int res = 0;

        while(res == 0 && key < size && hash[key] != NULL){
                if(strcmp(hash[key]->nome,nome)==0)
                        res = hash[key]->size;
                key++;
        }
        return res;
}

/*
 * Dado o nome de uma variavel vai buscar o seu endereco na pilha
 * @param nome, nome da variavel
 * @return endereco da variavel
 */
int findEnderecoV(char *nome){
        int key = getPos(nome);

        while(key < size && hash[key] != NULL){
                if(strcmp(hash[key]->nome,nome)==0)
                        return hash[key]->endereco;
                key++;
        }
        return 0;
}

/*
 * Sabendo que a variavel existe e que a posicao que estamos a tentar aceder existe va
 * @param nome, nome do array
```

```
 * @param posicao, posicao do array sobre a qual queremos obter o valor
*/
int getArrValue(char *nome, int posicao) {
        int key = getPos(nome);

        while(key < size && hash[key] != NULL){
                if(strcmp(hash[key]->nome,nome)==0)
                        return hash[key]->array[posicao];
                key++;
        }
        return 0;
}

/*
 * Verifica se a posicao do array que estamos a tentar aceder e valida
 * @param nome, nome da variavel
 * @param TRUE, caso posicao seja valida FALSE caso contrario
 */
int validPos(char *nome, int posicao) {
        int key = getPos(nome);

        while(key < size && hash[key] != NULL){
                if(strcmp(hash[key]->nome,nome)==0)
                        return (posicao < hash[key]->size);
                key++;
        }
        return FALSE;
}

/*
 * Sabendo que a variavel existe vai buscar o valor da variavel
 * @param nome, nome da variavel
 * @return valor da variavel
 */
int getVarValue(char *nome) {
        int key = getPos(nome);

        while(key < size && hash[key] != NULL){
                if(strcmp(hash[key]->nome,nome)==0)
                        return hash[key]->valor;
                key++;
        }
        return 0;
}
void writeLatexFancyStack(Var* v);


void orderHashEnd() {
```

```
        Var *aux;
        aux = (Var*) malloc(size * sizeof(Var));
        Var a;
        int k=0, j;
        for(j=0; j < size ; j++) {
                if (hash[j]!= NULL) {
                        aux[k] = hash[j];
                        k++;
                }
        }
        int i;
        for(i=k; i >= 1; i--) {
                for(j=0; j < i ; j++) {
                        if(aux[j+1]!=NULL && aux[j]->endereco > aux[j+1]->endereco) {
                                a = aux[j];
                                aux[j] = aux[j+1];
                                aux[j+1] = a;
                        }
                }
        }
        int r;
        for(r = 0; r < k; r++){
                printf("%d\n",aux[r]->endereco);
        }
        writeLatexFancyStack(aux);
}

void writeLatexFancyStack(Var* v)
{
   FILE *f = fopen("stack.tex", "w");

   fprintf(f, "\\documentclass{article}\n");
   fprintf(f, "\\usepackage[nocolor]{drawstack}\n");
   fprintf(f, "\\begin{document}\n");

   fprintf(f, "\\begin{drawstack}\n");

   // Percorrer estrutura de dados e criar uma stack bonita em LaTeX
   int i=size;
   int j;
   for(; i>=0; i--){
        if(hash[i]!=NULL){
                if(hash[i]->type==INT){
                        fprintf(f, "\\cell{%d} \\cellcom{%d: %s}\n", hash[i]->valor, h
                }
                else if(hash[i]->type==INT_ARRAY){
                        for(j=0; j<hash[i]->size && hash[i]->array[j]!=-1; j++){
                           fprintf(f,"\\cell{%d} \\cellcom{%d: %s[%d]}\n", hash[i]->ar
```

```
                        }
                    }
                }
            }

            fprintf(f, "\\end{drawstack}\n");
            fprintf(f, "\\end{document}\n");

            fclose(f);
}
```

11.5 const.h

```
#ifndef CONST
#define CONST

#define MAX_SIZE 1000

/*Tipos de dados*/
#define INT 3000
#define INT_ARRAY 3001

#define ERR_REDECLARED "erro. variavel redeclarada:"
#define ERR_ARRAY_OVERFLOW_1 "erro. stack overflow tamanho do array:"
#define ERR_ARRAY_OVERFLOW_2 "mas numero de elementos do array e:"
#define ERR_DIVISION_BY_0 "erro. opera  o aritm tica inv lida. tentativa de divis

#endif
```

11.6 makefile

```
fj: y.tab.o lex.yy.o
gcc -o fj y.tab.o lex.yy.o hash.o -ll

y.tab.o: y.tab.c
gcc -c y.tab.c

y.tab.c: hash.o fruitjuice.y
yacc -d fruitjuice.y

lex.yy.o: lex.yy.c
gcc -c lex.yy.c

lex.yy.c: fruitjuice.l
flex fruitjuice.l

hash.o: hash.h hash.c
gcc -c hash.c
```

```
clean:
-rm *.tex
-rm *.aux
-rm *.log
-rm *.pdf
-rm *.o
-rm y.tab.c
-rm y.tab.h
-rm lex.yy.c
-rm fj
```