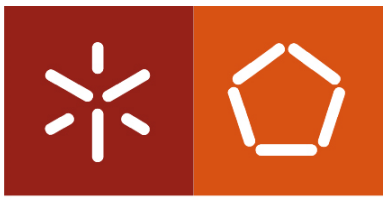

Escola de Engenharia

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Processamento de Linguagens

Criação de um Compilador em Yacc

Octávio Maia a71369
João Silva a72023
Rui Freitas a72399

31 de Maio de 2016

Conteúdo

Lista de Figuras

Resumo

Este relatório tem como objetivo descrever o processo de desenvolvimento, estruturação e tomadas de decisão resultantes da segunda fase do trabalho prático da Unidade Curricular de Processamento de Linguagens.

Nesta fase, foi abordado o desenvolvimento de *processadores de linguagens*, ou seja, o desenvolvimento de um *compilador* gerando código para uma *máquina de stack virtual*.

Palavras chave: Sintaxe, Compilador, Stack, Gramáticas Tradutoras, Yacc, Estruturas de Dados, Algoritmos de manipulação.

Capítulo 1

Introdução

A realização deste trabalho prático, tem como objetivo o desenvolvimento de uma linguagem de programação imperativa simples *LPIS*[1]. Será também ser desenvolvido um compilador para a linguagem definida de modo a gerar código *assembly* para uma máquina de stack virtual que interpretará o código.

Tal como uma grande parte das linguagens imperativas esta linguagem deve a realização de operações lógicas e matemáticas para isso lidando apenas com valores do tipo inteiro. Para finalizar é também fornecido a implementação de ciclos e blocos condicionais, podendo estes ser aninhados.

1.1 Descrição do Problema e Implementação

A nossa primeira tarefa teve como base a definição da linguagem que irá ser utilizada no âmbito desta segunda fase do trabalho, de modo a serem capazes de processar a informação requerida.

Após a finalização da linguagem, tivemos como objetivo o desenvolvimento do código de forma a responder a todas as tarefas requeridas.

[1]Linguagem de programação imperativa simples.

Capítulo 2

Definição da Linguagem

Neste capitulo, será abordado ao pormenor as regras estabelecidas pelo grupo durante o desenvolvimento da linguagem. Explicitando a documentação da linguagem, bem como a forma de construção de um programa em *LIPS* e posteriormente apresentado exemplos das regras definidas.

2.1 Estrutura geral de um programa em *LIPS*

Um programa definido em *LIPS* tem, obrigatoriamente, de ser iniciado recorrendo à palavra reservada *INICIO* e terminado pela palavra *FIM*. Entre estas duas palavras reservadas encontra-se todo o corpo do programa.

```
INICIO
    corpo do programa
    ...
    ...
    ...
FIM
```

2.1.1 Definição de variáveis

O campo *corpo do programa* é constituído por uma parte relativa à declaração de variáveis, apenas do tipo *inteiro*.

Estas variáveis podem ser simples, *arrays* uni-dimensionais ou bidimensionais. A declaração de uma variável deve ser feita recorrendo à palavra reservada *VAR*.

- Declaração de variável simples com o identificador "a".

```
VAR a;
```

- Declaração de variável simples com o identificador "b" com valor 5.

```
VAR b=5;
```

- Declaração de um *array* uni-dimensional com o identificador "c" e tamanho 10.

```
VAR c[10];
```

- Declaração de um *array* bidimensional com o identificador "d" e tamanho 10 por 20.

```
VAR d[10][20];
```

A alocação de memória para *Arrays* segue o principio de *Row Major*, ou seja, as linhas são alocadas em blocos contínuos.

A referencia a uma posição de um *Array* começa em 0 e termina em $tamanho - 1$. Assim sendo, para o *Array* uni-dimensional "a" exemplificado anteriormente, as posições válidas de acesso variam entre *a[0]* até *a[9]* inclusive.

Todas a variáveis simples definidas sem valor predefinido, tal como os *arrays* é atribuído o valor 0.

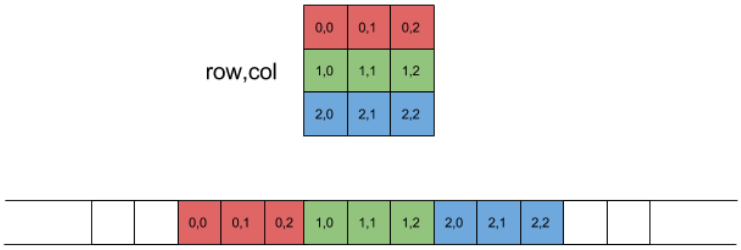

Figura 2.1: Representação de alocação *Row Major*

Seguida à declaração de variáveis, é iniciada a parte do programa que contém as declarações das funções auxiliares que se encontram no programa. instruções permitidas da linguagem, sendo este bloco iniciado pela palavra reservada *INSTINICIO* e terminado pela palavra reservada *INSTFIM*.

```
INICIO
    declaração variáveis
    declaração de funções
    INSTINICIO (main)

        conjunto de instruções disponíveis

    INSTFIM
FIM
```

2.1.2 Declaração de Funções

A linguagem *LPIS* desenvolvida, permite ainda a declaração de funções auxiliares à *main* para que sejam invocadas quando desejado.

O bloco relativo a estas funções auxiliares poderá ser vazio, caso não seja necessário, ou então poderá ter quantas funções deseje.

As funções podem ser de dois tipos, sem retorno, sendo iniciadas com a palavra reservada *VOID* ou então contendo retorno, não sendo especificado nada anteriormente à sua declaração. Tais funções, contêm ainda a possibilidade de possuírem variáveis locais com um mesmo identificador existente nas variáveis globais, mas não podem aceder a variáveis globais.

Como se trata de um linguagem simples, não existe a possibilidade de chamada de funções numa função.

A declaração de uma função sem retorno pode ser feita da seguinte forma:

```
VOID FUNCTION teste ( )
    conjunto de instruções disponíveis
FIMFUNCTION
```

Já a declaração de uma função com retorno pode ser feita da seguinte forma:

```
FUNCTION teste ( )
    VAR x;
    conjunto de instruções disponíveis
    RETURN x;
FIMFUNCTION
```

Após a declaração de funções auxiliares, é iniciada a parte do programa que dirá respeito à *Main*, sendo este bloco iniciado pela palavra reservada *INSTINICIO* e terminado pela palavra reservada *INSTFIM*.

2.1.3 Conjunto de Instruções Disponíveis

Nesta parte do programa, tal como na parte relativa à declaração de variáveis, é permitido a atribuição de valores a varáveis simples. Mas apenas aqui é possível atribuir valores a *arrays*, sendo a única forma de atribuição feita individualmente a cada uma das posições.

Exemplos utilização:

```
a = 5;
c[5] = a;
d[2][2] = 1;
d[2][2] = d[0][0];
c[a] = 10;
c[a-1] = 4;
```

Operações aritméticas e lógicas

O conjunto de operações aritméticas disponíveis na linguagem a ser definida é restrito a

- Somas representado pelo sinal *+*
- Subtrações representado pelo sinal -
- Multiplicações representado pelo sinal ***
- Divisões representado pelo sinal */*
- Modulo representado pelo sinal *%*

Apesar de se tratar de uma linguagem de programação muito simples, as operações aritméticas possuem em consideração as prioridades do operadores. Posto isto, as multiplicações e divisões têm prioridade relativamente às somas e subtrações, logo esta terá de ser sempre forçada através do uso de parênteses.

O uso de parênteses permite que estas ordens sejam modificadas, tendo os operandos entre parênteses prioridade.

Exemplos de utilização:

```
a = 5 + (4 + 1); resultado será 10
a = (5 + 4) + 1; resultado será 10
a = 5 * (4 + 1); resultado será 25
a = 5 + (4 * 1); resultado será 9
a = 5 + 4 * 1; resultado será 9
a = 1 + a; forma de incrementar variável
```

Estando apresentadas as operações aritméticas disponíveis, serão apresentadas as operações lógicas consideradas. Tal com acontece na linguagem *C* os valores lógicos são representados em inteiros, sendo que *0* representa o valor de *Falso*, e qualquer valor diferente de *0* representa o valor de *Verdade*.

O conjunto de operações lógicas disponíveis na linguagem a ser definida é restrito a:

- $>$ representado pela palavra reservada *GG*
- $<$ representado pela palavra reservada *LL*
- $=$ representado pela palavra reservada *EQ*
- $\neq$ representado pela palavra reservada *DIFF*
- $\geq$ representado pela palavra reservada *GE*
- $\leq$ representado pela palavra reservada *LE*
- $\wedge$ representado pela palavra reservada *AND*
- $\vee$ representado pela palavra reservada *OR*
- $\neg$ representado pela palavra reservada *NOT*

As operações lógicas permitem várias operações recursivas, bem como o seu resultado ser guardado numa variável para que depois possa ser utilizado.

Exemplos de utilização:

```
a AND a; resultado será Verdade
a OR a; resultado será Verdade
NOT a; resultado será Falso
a LL a; resultado será Falso
a GE a; resultado será Verdade
```

Instruções condicionais

Tal como em todas as linguagens, a *LPIS* permite a utilização de blocos de código condicionais, de modo a ter controlo de fluxo para o código a utilizar. As estruturas condicionais utilizadas são constituídas pelo conjunto de instruções a executar, caso a condição seja verdadeira, mas também caso a condição seja falsa, contudo estes conjuntos de instruções podem ser vazios.

- Estrutura geral de um bloco condicional

```
SE (condição) ENTAO
    conjunto instruções a realizar na verdade da condição
SENAO
    conjunto instruções a realizar na falsidade da condição
FIMSE
```

A condição referida acima poderá ser

- expressão lógica.
- variável.

Exemplos de utilização:

```
SE (NOT a) ENTAO
    a=1;
SENAO
    a=0;
FIMSE
```

```
SE (a) ENTAO
    a=0;
SENAO
    a=1;
SIMSE
```

Instruções cíclicas

Tal como em todas as linguagens, a *LPIS* permite a utilização de blocos de código cíclico, de modo a ter controlo de fluxo para o código a utilizar e efetuar repetições.

As estruturas cíclicas utilizadas são constituídas pelo conjunto de instruções a executar. Caso a condição seja verdadeira este conjuntos de instruções poderá ser vazio.

- Estrutura geral de um bloco condicional

```
ENQUANTO (condição) ENTAO
    conjunto instruções a realizar na verdade da condição
FIMENQUANTO
```

A condição referida acima poderá ser

- expressão lógica.
- variável.

Exemplos utilização:

```
ENQUANTO (a LE 10) ENTAO
    a=a+1;
FIMENQUANTO


ENQUANTO (a) ENTAO
    a=0;
FIMENQUANTO
```

Para as instruções cíclicas e condicionais, o corpo de instruções das mesmas poderá ser novamente uma instrução cíclica ou uma instrução condicional, permitindo que estas instruções sejam aninhadas.

2.1.4 Interação com o utilizador

A interação com o utilizador é feita recorrendo as instruções *LER var;* e *IMPRIMIR var;* onde *var* é uma variável simples ou uma posição de um *array*.

A instrução *LER x;* permite ler do *stdin* um valor inteiro e guarda-lo na posição de memória correspondente a *x*.

A instrução *IMPRIMIR x;* permitem escrever no *stdout* um valor inteiro que se encontra na posição de memória correspondente a *x*.

Exemplos de utilização:

```
LER a;
LER c[1];
IMPRIMIR d[0][0];
```

Estando apresentado o conjunto de instruções que estão disponíveis na linguagem definida e a forma como estas podem ser utilizadas, consideramos esta documentada.

Encontrasse ainda disponível em 5.3 um conjunto de programas definidos nesta linguagem.

2.2 Apresentação da Gramática Independente de Contexto

Uma vez definida a estrutura da nossa linguagem e as palavras reservadas, tornou-se necessário desenvolver uma GIC[1] que suporte a linguagem apresentada.

$Programa \quad \rightarrow \quad 'INICIO' \quad Declaracoes \quad Functions \quad Body \quad 'FIM'$

Esta primeira regra explicita que um programa definido na linguagem especificada tem necessariamente de ser iniciado com a palavra reservada "INICIO" e terminada pela palavra reservada "FIM". Entre estas duas palavras existe um bloco de declarações de variáveis seguidas pelo corpo do programa.

$Declaracoes \quad \rightarrow \quad \varepsilon$
$| \quad Declaracoes \quad Declaracao$

O Bloco correspondente às declarações pode ser vazio, dai existir a derivação em ε ou entao ser composto por uma declaração e recursivamente declarações.

$Declaracao \quad \rightarrow \quad 'VAR' \quad id \quad ';'$
$| \quad 'VAR' \quad id \quad '=' \quad numero \quad ';'$
$| \quad 'VAR' \quad id \quad '['num']' \quad ';'$
$| \quad 'VAR' \quad id \quad '['num']' \quad '['num']' \quad ';'$

A declaração de variáveis é feita obedecendo à regra acima apresentada. A declaração de uma variável é feita recorrendo à palavra reservada "VAR" seguida do seu nome identificador. Esta variável poderá ser inicializada ou não a quando da sua declaração.

Existe a possibilidade de declarar 3 tipos de variáveis, simples, *array* uni-dimensional e bidimensional com indicação dos tamanhos que estes possuem.

$Functions \quad \rightarrow \quad \varepsilon$
$| \quad Functions \quad Function$

$Function \quad \rightarrow \quad 'FUNCTION' \quad id \quad '('')' \quad Declaracoes \quad Instrucoes$
$'RETURN' \quad Expressao \quad 'FIMFUNCTION'$
$| \quad 'VOID' \quad 'FUNCTION' \quad id \quad '('')' \quad Declaracoes \quad Instrucoes \quad 'FIMFUNCTION'$
$| \quad 'FUNCTION' \quad id \quad '(' \quad 'VAR' \quad id \quad ',' \quad 'VAR' \quad id \quad ')' \quad 'FIMFUNCTION'$
$| \quad 'VOID' \quad 'FUNCTION' \quad id \quad '(' \quad 'VAR' \quad id \quad ',' \quad 'VAR' \quad id \quad ')' \quad 'FIMFUNCTION'$

$Body \quad \rightarrow \quad 'INSTINICIO' \quad Instrucoes \quad 'INSTFI'$

[1] Gramática Independente de Contexto

$Instrucoes \rightarrow \varepsilon$
$\mid\ Instrucoes\ \ Instrucao$

$Instrucao \rightarrow Atribuicao$
$\mid\ Condicional$
$\mid\ Input$
$\mid\ Output$
$\mid\ Ciclo$
$\mid\ id\ \ '('\ \ Args\ \ ')'\ \ ';'$

$Args \rightarrow \varepsilon$
$\mid\ Args\ \ Arg$

$Arg \rightarrow Expressao$

$Atribuicao \rightarrow Endereco\ \ '='\ \ Condicao\ \ ';'$

$Condicao \rightarrow Expressao$
$\mid\ Expressao\ \ OpRel\ \ Expressao$
$\mid\ 'NOT'\ \ Expressao$

$OpLog \rightarrow 'AND'$
$\mid\ 'OR'$

$Expressao \rightarrow Termo$
$\mid\ Expressao\ \ OpAdd\ \ Termo$

$OpAdd \rightarrow '+'$
$\mid\ '-'$
$\mid\ 'DIFF'$
$\mid\ 'GG'$
$\mid\ 'LL'$
$\mid\ 'GE'$
$\mid\ 'LE'$
$\mid\ 'EQ'$

$Termo \rightarrow Fator$
$\mid\ Termo\ \ OpMul\ \ Forma$

$Fator \rightarrow Valor$
$\mid\ '('\ \ Expressao\ \ ')'$
$\mid\ id\ \ '('\ \ Args\ \ ')'$

$OpMul \rightarrow\ '*'$
$\mid\ '/'$
$\mid\ '\%'$

$Condicional \rightarrow\ 'SE'\ \ '('Condicao')'\ \ 'ENTAO'$
$Instrucoes\ \ 'SENAO'\ \ Instrucoes\ \ 'FIMSE'$

$Ciclo \rightarrow\ 'ENQUANTO'\ \ '('Condicao')'\ \ 'ENTAO'$
$Instrucoes\ \ 'FIMENQUANTO'$

$Input \rightarrow\ 'LER'\ \ Endereco\ \ ';'$

$Output \rightarrow\ 'IMPRIMIR'\ \ Valor\ \ ';'$

$numero \rightarrow\ num$
$\mid\ '-'\ \ num$

$Valor \rightarrow\ numero$
$\mid\ Variavel$

$Endereco \rightarrow\ id$
$\mid\ id\ \ '['Expressao']'$
$\mid\ id\ \ '['Expressao']'\ \ '['Expressao']'$

$Variavel \rightarrow\ id$
$\mid\ id\ \ '['Expressao']'$
$\mid\ id\ \ '['Expressao']'\ \ '['Expressao']'$

Capítulo 3

Desenvolvimento

Neste capitulo será abordado o desenvolvimento do compilador responsável por efetuar a compilação de um programa na linguagem *LPIS*, com a gramática definida em 2 de modo a este ser executável na *VM*[1].

A estruturação deste capitulo será inicialmente a apresentação das estrutura de dados onde o compilador a ser desenvolvido se irá basear de forma a que a compilação seja possível.

Numa segunda parte apresentaremos as ações a executar quando uma produção da gramática for reconhecida.

3.1 Estrutura de Dados de Suporte

Como estrutura de dados de suporte ao compilador a ser desenvolvido, e de forma a guardar informação relativa às variáveis definidas num programa na linguagem *LPIS*, foi utilizada uma *Tabela de HASH*. A escolha desta estrutura de dados foi devido a ser a que é utilizada em compiladores reais e também ser a indicada pelos docentes.

A *Tabela de HASH* deve proporcionar uma associação entre um identificador de uma variável, ou função (*string*), e a informação associada a esse identificador (*Definition*). *Definition* foi a estrutura de dados que o grupo pensou, onde é possível guardar o identificador da variável, ou função no campo *nome*, e o seu tipo onde é distinguindo uma variável de uma função.

```
typedef struct definition{
  char* name;
  int type;
  union {
    Variavel var;
    Function func;
  };
}*Definition;
```

No caso das variáveis a informação a guardar visto estas poderem ser simples, *Arrays* uni-dimensionais e bidimensionais é necessário guardar o endereço de memória

[1]Máquina virtual disponibilizada

onde estas se encontram, o tamanho ocupado por elas e a dimensão de cada "linha" no caso dos *Arrays* bidimensionais.

```
typedef struct variavel {
  int addr;
  int dim;
  int size;
}*Variavel;
```

Para os identificadores de funções é guardada o seu identificador, se permite retorno ou não e ainda o numero de argumentos que possui.

```
typedef struct function {
  char* label;
  int enableReturn;
  int argc;
}*Function;
```

Para gestão das variáveis internas às funções, foi utilizada uma outra instância da estrutura apresentada, com existência apenas aquando da definição da função, uma que vez que quando terminamos uma definição a informação relativa às variáveis internas não é mais necessária.

Já aquando do desenvolvimento, de modo a conseguir a gestão das estruturas cíclicas e condicionais, o grupo implementou uma *Stack* para gerir as *Labels* destas estruturas, permitindo assim a existência de múltiplas estruturas existentes no programa ao mesmo nível ou aninhadas.

Apresentada a estrutura de dados de suporte, resta definir as ações a executar quando uma produção for reconhecida.

De modo a utilizar a tabela de HASH foram criados os seguintes métodos:

HashTable createHashTable()
Responsável pela criação da tabela de HASH
void insertDefinition(HashTable h, Definition var)
Responsável pela inserção de uma *Definition*
Definition getDefinition(HashTable h, char* name, int type)
Responsável pela obtenção de uma *Definition*

3.2 Ações a executar

Pela razão do programa a ser desenvolvido se tratar um compilador, as ações a executar quando uma produção *Pi* for reconhecida a ação a executar deverá ser gerar o código *Assembly* associado à "instrução" reconhecida para que este seja executado na *VM*, ou então imprimir o código gerado das produções reconhecidas anteriormente para o ficheiro com código compilado.

- ***Declaração de variáveis***

Quando é reconhecida uma declaração de uma variável é chamada a função.

```
void declaracao(int tamanho, char* identficador, int value, int dim);
```

Tal função é responsável pela verificação da não repetição de identificadores na *Tabela de Hash*, caso seja detetada um repetição de um identificador o utilizador é informado e o processo de compilação é terminado. Caso não seja detetado uma declaração repetida, a informação relativa à variável é inserida na tabela, o contador do próximo endereço de memória a utilizar é atualizado e ainda é imprimido para o ficheiro compilado o código correspondente à declaração.

Esta é também a função invocada para qualquer tipo de declaração. Posto isto, torna-se necessário distinguir os tipos de instruções de compilação para cada tipo de variável, para que sejam escritas instruções diferentes para cada tipo de declaração.

- ***Declaração de funções***

As declarações de funções são um caso especial do programa, pois cada função tem as suas próprias variáveis locais. Ou seja, tem de ser criada uma nova estrutura de dados para cada função que é declarada, de modo a que sejam verificadas as variáveis que nela são declaradas e se todas as variáveis utilizadas na função estão declaradas.

Quando o *parser* fizer *match* num cabeçalho de declaração de uma função, vai colocar a variável global do programa *currPointer*(variável que controla o endereços de memória das variáveis) a zero, visto que cada função ao ser chamada tem a sua zona de memória, e as variáveis nela declarada são guardadas apenas na sua memória e deixam de existir quando a função acaba a sua execução. De seguida é chamada a função:

```
void declaracaoFunction(char* nameFuntion, int retorno);
```

Esta função recebe como parâmetros o nome da função a declarar e se esta tem retorno ou não (argumento retorna 1 se a função tiver retorno; 0 caso contrário). Após isto verifica na tabela de hash principal se a função que está a ser declarada já se encontra declarada. Neste caso, é libertado um erro *"Multiple functions definitions"* e é terminado o processo de compilação. Caso contrário, a função a declarar é guardada na estrutura de dados com todos os dados necessários a ser guardado tal como:

- *possibilidade de retorno* Para distinguir os casos em que podem ser posteriormente chamadas as funções. Uma função com retorno apenas pode ser chamada quando é identificada numa expressão. Já as funções sem return podem ser chamadas em qualquer altura do programa.

– *label da função* Necessário para que seja conhecida o program counter em que se encontra o código da função para ser utilizado quando a função for chamada.

Quando for encontrado o final da declaração da função é chamada a instrução *return*, para voltar ao estado antes da função ser chamada. Deste modo, a estrutura secundária utilizada para controlar as variáveis de cada função é regenerada para que numa próxima declaração de uma função seja reutilizada.

- ***Bloco de Instruções***

Quando a produção relativa ao bloco de instruções é reconhecida é escrito inicialmente para o ficheiro de código compilado a instrução *start*, que indica o inicio do bloco relativo a instruções do programa. Após esta impressão são impressas todas as instruções do programa (que já haviam sido reconhecido e traduzidas), por fim a instrução *stop* é impressa de modo a marcar o termino do programa.

- ***Reconhecimento de operadores***

Quando é reconhecida uma instrução de relativa a uma operação que esteja no "saco" das

- OpRel;
- OpAdd;
- OpMul.

a ação a executar é simplesmente a atribuição do código correspondente à operação na *VM* do símbolo terminal reconhecido ao símbolo não terminal.

Contudo dado que na *VM* não existe um instrução correspondente à diferença lógica, quando reconhecemos uma diferença lógica o código gerado é a instrução *NOT* seguida da instrução *EQUAL*

- ***Reconhecimento de uma Instrução***

Devido a uma *Instrucao* apenas derivar em um símbolo não terminal em cada produção e sem mais nenhum filho na árvore, quando uma instrução é reconhecida é apenas atribuído ao *$$* o valor do que foi reconhecido anteriormente.

- ***Leitura de valor***

Quando ocorre uma leitura o valor este é sempre um valor inteiro, o conjunto de instruções a realizar será ler com a instrução *READ* da *VM*. Após isso é necessário converter a *string* lida para para inteiro. Por fim quadrar o valor convertido no endereço de memória correspondente é variável indicada. Uma vez que este endereço foi já calculado na produção correspondente ao *Endereço* e colocado na *Stack* é utilizada a instrução *STORE 0* de forma a que o valor lido seja guardado no endereço que se encontra imediatamente a baixo na *Stack*.

- ***Reconhecimento de* Numero**

Devido às complicações com o símbolo "-" ser poder significar a diferença ou um valor negativo, este reconhecimento não pode ser feito no analisador léxico, sendo que o analisador apenas reconhece "Inteiros". Assim sendo, o tratamento é feito no *parser*. Caso seja reconhecido "-" seguido de um "num" corresponderá a um numero negativo, logo nesse caso *$$* será igual ao oposto do número reconhecido.

- ***Impressão de valor***

Quando ocorre a impressão de algo, esse ou é um valor direto (por exemplo 5) ou então será para imprimir o que se encontra no endereço de memória correspondente ao argumento. Assumindo que o calculo do valor direto a imprimir ou da posição de memória onde se encontra a "variavel" a imprimir já se encontra calculado em *$1*. O conjunto de instruções a realizar será apenas imprimir o que ai se encontra.

- ***Reconhecimento de valor***

Sendo valor ou um numero ou então uma variável, caso seja um numero então o código a retornar em *$$* será apenas fazer o *PUSHI* do numero especificado. Caso seja reconhecido como variável uma vez que o endereço de memória correspondente a essa variável já se encontra calculado pela produção que reconhece variáveis, o valor a retornar em *$$* será igual ao produzindo pelo reconhecimento da variável.

- ***Reconhecimento de* Variavel**

Para obter o valor de uma variável (por exemplo quando esta é um operando), é necessário sempre conhecer o endereço onde esta se encontra alocada.

Para as variáveis simples, como o local a aceder é exatamente o que se encontra referido na tabela de identificadores apenas é necessário fazer um *PUSHG* desse endereço.

Para variáveis associadas a *Arrays*, visto estes terem indexação a uma ou duas dimensões, antes de fazer o *PUSHG* da posição referenciada por essas expressões é necessário efetuar os cálculos dessas expressões para depois fazer operações com esses valores, de modo a aceder ao endereço correto segundo a alocação *Row Major*.

Para *Arrays* de duas dimensões, inicialmente é feito um *PUSHGP* para que os cálculos seguintes sejam interpretados como endereços. De seguida, é calculada a excreção correspondente à primeira dimensão executando o código gerado correspondente a essa expressão. Após isso é feito um *PUSHI* da dimensão de cada linha para de seguida ser efetuada a sua multiplicarão de seguida é somado ao código da expressão que nos dá o segundo índice. Após todo este processo é feito o *LOAD* ao endereço que se encontra no topo da *Stack*.

- ***Reconhecimento de* Endereco**

Após a explicação de como é carregado um valor de uma variável em tempo de execução em 3.2, o carregamento de endereços para *Arrays* é semelhante, sendo que a ultima expressão, em vez de ser um *LOAD* será um *PADD*.

A grande diferença será nas variáveis simples, devido a que não poderá ser um simples *PUSH* do endereço dela. Sendo necessário obter um endereço é anteriormente feito o *PUSHGP*, para depois ser adicionado em modo endereço *PADD* com o endereço onde a variável foi alocada.

- ***Reconhecimento Bloco Condicional***

Durante o reconhecimento de um bloco condicional, sendo este iniciado sempre pela palavra reservada *SE*, é introduzido na *stack* que controla as *labels* destes blocos a informação relativa à existência do novo bloco.

Uma vez reconhecida, é impressa a Condição correspondente ao teste deste bloco. Uma vez impressa a condição para teste, é marcado o inicio do corpo do *IF* com a *label* correspondente. Terminado o reconhecimento das instruções que fazem parte do bloco *IF* é impressas a instrução que permite o salto para o final do bloco condicional. Seguidamente é impressa a *label* que identifica o bloco do *ELSE*. Após este passo são reconhecidas as instruções que fazem parte do bloco alternativo. Por fim reconhecido o fim do ciclo e impressa a *label* que o identifica este final.

- ***Reconhecimento Bloco Ciclico***

Durante o reconhecimento de um bloco cíclico, sendo este iniciado sempre pela palavra reservada *ENQUANTO*, é introduzido na *stack* que controla as *labels* destes blocos a informação relativa à existência do novo bloco, sendo a mesma impressa.

Uma vez reconhecida e impressa a Condição correspondente ao teste deste bloco, é feita a impressão do instrução que permite o salto para o final do ciclo caso a condição seja falsa.

De seguida são impressas todas as instruções presentes no corpo do ciclo. Quando o final do ciclo é reconhecido devido à palavra reservada *FIMENQUANTO*, é impressa a instrução para voltar à instrução de inicio do ciclo, e após isso é impressa a *label* que identifica o final do bloco cíclico.

- ***Reconhecimento chamadas de funções***

Existem duas formas/locais em que as funções podem ser chamadas. Como uma instrução normal, para as funções que não tem um valor de retorno e também podem ser utilizadas nas expressões onde é necessário que a função tenha um valor de retorno.

As funções podem ter argumentos, que necessitam de ser colocados na stack antes das ações da função sejam chamadas, para que os argumentos sejam colocados em posições negativas em relação ao frame point para posteriormente serem acedidos pelas instruções das funções.

Os argumentos das funções, aquando da sua chamada, podem ser *Expressoes* e é realizado o push das variáveis para a stack.

Para serem realizados todos este procedimentos relativos á chamada de uma função é utilizada a seguinte função.

```
void functionCall(char* nameFuntion, int needReturn);
```

Esta função, para além de realizar as ações expressas acima verifica se a função está declarada (caso não esteja é libertado o erro "Function not exist. Need to be declared") e verifica se o argumento *needReturn* está de acordo com o valor contido na estrutura de dados sobre o retorno da função.

O valor *needReturn* pode tomar o valor 1 caso necessite de retorno ou 0 caso contrário. Caso este valor não correspondam é libertado um erro "Function return a value. Call error" ou "Function not return a value. Call error".

Capítulo 4

Conclusão

Na fase final do trabalho deparamo-nos que podiamos ter estruturado melhor na nossa estrutura de dados. Para que de forma mais facil fosse permitida a utilização de variaveis globais e a

Capítulo 5

Anexos

5.1 Analisador Léxico

```
%{

%}

letra [A-Za-z]
digito [0-9]
lixo .|\n

%%

[\+\-\*\/=;()\[\]\%]    { return yytext[0]; }
INICIO                  { return INICIO; }
FIM                     { return FIM; }
INSTINICIO              { return INSTINICIO; }
INSTFIM                 { return INSTFIM; }
VOID                    { return VOID; }
FUNCTION                { return FUNCTION; }
FIMFUNCTION             { return FIMFUNCTION; }
RETURN                  { return RETURN; }
VAR                     { return VAR; }
SE                      { return SE; }
ENTAO                   { return ENTAO; }
SENAO                   { return SENAO; }
FIMSE                   { return FIMSE; }
ENQUANTO                { return ENQUANTO; }
FIMENQUANTO             { return FIMENQUANTO; }
IMPRIMIR                { return IMPRIMIR; }
LER                     { return LER; }
DIFF                    { return DIFF; }
GG                      { return GG; }
```

```
LL                      { return LL; }
GE                      { return GE; }
LE                      { return LE; }
EQ                      { return EQ; }
AND                     { return AND; }
NOT                     { return NOT; }


{letra}+    { yylval.var = strdup(yytext); return id; }
{digito}+   { yylval.val = atoi(yytext); return num; }
{lixo}      { ; }

%%

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

5.2 *GIC* em *YACC*

```
Programa    : INICIO Declaracoes Functions Body FIM
            ;

Declaracoes :
            | Declaracoes Declaracao
            ;

Declaracao : VAR id ';'
            | VAR id '=' numero ';'
            | VAR id '[' num ']' ';'
            | VAR id '[' num ']''[' num ']'  ';'
            ;

Functions   :
            | Functions Function
            ;

Function    : FUNCTION id '('')' Declaracoes Instrucoes RETURN Expressao FIMFUNCTIO
            | VOID FUNCTION id '('')'  Declaracoes Instrucoes FIMFUNCTION
            | FUNCTION id '(' VAR id ',' VAR id ')' FIMFUNCTION
            | VOID FUNCTION id '(' VAR id ',' VAR id ')' FIMFUNCTION
            ;

Body        : INSTINICIO Instrucoes INSTFIM
```

```
    ;

Instrucoes :
            | Instrucoes Instrucao
            ;

Instrucao   : Atribuicao
            | Condicional
            | Input
            | Output
            | Ciclo
            | id '(' Args ')' ';'
            ;

Args        :
            | Args Arg
            ;

Arg         : Expressao
    ;

Atribuicao  : Endereco '=' Condicao ';'
            ;

Condicao    : Expressao
        | Expressao OpLog Expressao
    | NOT Expressao
    ;

OpLog       : AND
    | OR
     ;

Expressao   : Termo
            | Expressao OpAdd Termo
            ;

OpAdd     : '+'
            | '-'
            | DIFF
            | GG
            | LL
            | GE
            | LE
            | EQ
```

```
            ;

Termo       : Fator
            | Termo OpMul Fator
            ;

Fator       : Valor
            | '(' Expressao ')'
            | id '(' Args ')'
            ;

OpMul       : '*'
            | '/'
            | '%'
            ;

Condicional : SE  '(' Condicao ')' ENTAO Instrucoes SENAO Instrucoes FIMSE
;

Ciclo       : ENQUANTO '(' Condicao ')'ENTAO Instrucoes FIMENQUANTO
;

Input       : LER Endereco ';'
            ;

Output      : IMPRIMIR Valor ';'
            ;

numero      : num
            | '-' num
            ;

Valor       : numero
            | Variavel
            ;

Endereco    : id
            | id '[' Expressao ']'
            | id '[' Expressao ']''[' Expressao ']'
            ;

Variavel    : id
            | id '[' Expressao ']'
            | id '[' Expressao ']''[' Expressao ']'
```

5.3 Programas Exemplo

- Lidos 3 números, escrever o maior deles:

```
INICIO
    VAR a;
    VAR b;
    VAR c;
    VAR g;
INSTINICIO
    g=0;
    LER a;
    g=a;
    LER b;
    SE (b GG g) ENTAO
        g=b;
    SENAO
    FIMSE
    LER c;
    SE (c GG g) ENTAO
        g=c;
    SENAO
    FIMSE
    IMPRIMIR g;
INSTFIM
FIM
```

- Ler N (valor dado) números e calcular e imprimir o seu somatório:

```
INICIO
    VAR n;
    VAR na;
    VAR read;
    VAR sum;
INSTINICIO
    sum = 0;
    na=1;
    LER n;
    ENQUANTO (na LE n) ENTAO
        LER read;
        sum = sum + read;
        na = na +1;
    FIMENQUANTO
    IMPRIMIR sum;
INSTFIM
FIM
```

- Contar e imprimir os números pares de uma sequência de N números dados:

```
INICIO
    VAR n;
    VAR na;
    VAR read;
    VAR tot;
    VAR mod;
INSTINICIO
    tot = 0;
    na=0;
    LER n;
    ENQUANTO (na LL n) ENTAO
        LER read;
        mod = read % 2;
        SE (mod EQ 0) ENTAO
            IMPRIMIR read;
            tot=tot+1;
        SENAO
        FIMSE
        na = na +1;
    FIMENQUANTO
    IMPRIMIR tot;
INSTFIM
FIM
```

- Ler e armazenar os elementos de um vetor de comprimento N, imprimir os valores por ordem crescente após fazer a ordenação do *array* por trocas diretas:

```
INICIO
    VAR n;
    VAR na;
    VAR read;
    VAR arr[10];
    VAR i;
    VAR j;
    VAR swap;
    VAR aux;
    VAR jone;
INSTINICIO
    tot = 0;
    na=0;
    n=10;
    ENQUANTO (na LL n) ENTAO
        LER read;
        arr[na]=read;
```

```
        na = na +1;
    FIMENQUANTO
    i=0;
    n=n-1;
    ENQUANTO (i LL n) ENTAO
        j=0;
        aux=n-i;
        ENQUANTO (j LL aux) ENTAO
            jone=j+1;
            SE (arr[j] GG arr[jone]) ENTAO
                swap = arr[j];
                arr[j]=arr[jone];
                arr[jone]=swap;
            SENAO
            FIMSE
            j=j+1;
        FIMENQUANTO
        i = i +1;
    FIMENQUANTO
    i=0;
    ENQUANTO (i LL n) ENTAO
        IMPRIMIR arr[na];
        i = i +1;
    FIMENQUANTO
INSTFIM
FIM
```

- Ler e armazenar os elementos de uma matriz NxM, calcular e imprimir a média e máximo dessa matriz:

```
INICIO
    VAR i;
    VAR j;
    VAR read;
    VAR max=-999;
    VAR sum=0;
    VAR matrix[10][10];
    VAR atual;
    VAR media;
INSTINICIO
    ENQUANTO (i LL 10) ENTAO
        ENQUANTO (j LL 10) ENTAO
            LER read;
            matrix[i][j]=read;
            j = j +1;
        FIMENQUANTO
```

```
        i = i +1;
    FIMENQUANTO

    i=0;
    j=0;
    ENQUANTO (i LL 10) ENTAO
        ENQUANTO (j LL 10) ENTAO
            atual=matrix[i][j];
            sum=sum+atual;
            SE (atual GE max) ENTAO
                max=atual;
            SENAO
            FIMSE
        FIMENQUANTO
        i = i +1;
    FIMENQUANTO
    IMPRIMIR max;
    media=sum / (i*j);
    IMPRIMIR media;
INSTFIM
FIM
```

- Invocar e usar num programa seu uma função "potencia(Base,Exp)":

```
FAZER
```