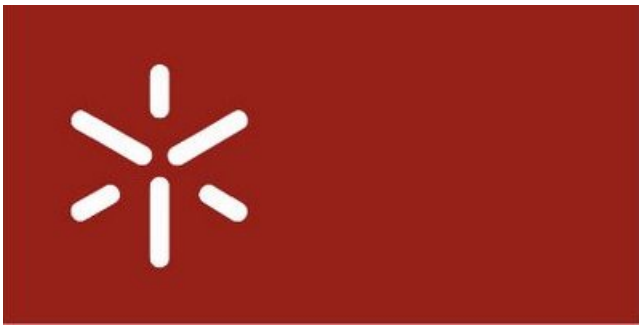


Desenvolvimento de Linguagem e Compilador

Mestrado Integrado em Engenharia Informática

Processamento de Linguagens
2º Semestre\2016-2017

A70676 Marcos de Morais Luís
A71625 Nelson Arieira Parente

11 de Junho de 2017
Braga

Processamento de Linguagens

Resumo

Este relatório tem como objetivo a exposição da estratégia adotada, código escrito e documentação para definição de uma linguagem iterativa simples e a criação do seu compilador - com recurso às ferramentas Flex e Yacc - que gera código para uma máquina de stack virtual. Os objectivos de aprendizagem foram alcançados, havendo agora uma muita maior compreensão das gramáticas independentes de contexto com condição LR().

Conteúdo

Capítulo 1

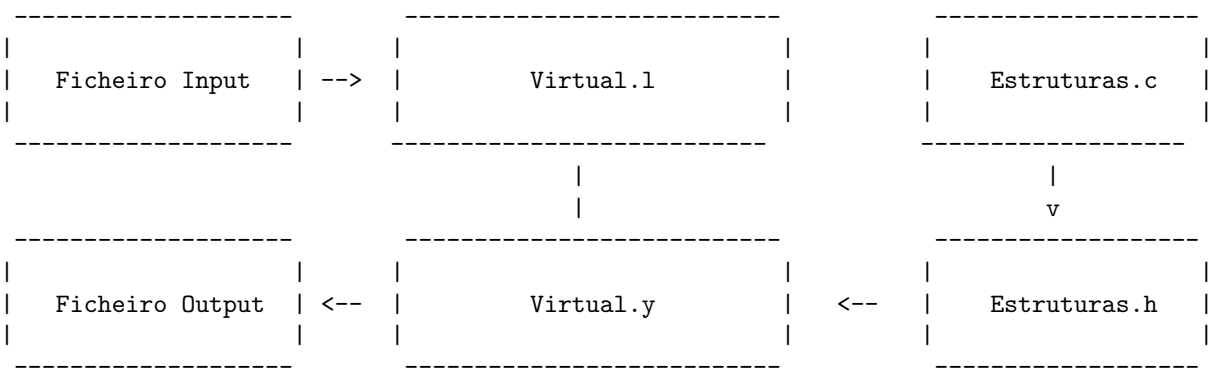
Introdução

Neste trabalho é nos proposto o desenvolvimento um compilador gerando código para uma máquina de stack virtual utilizando geradores de compiladores baseados em gramáticas tradutoras, sendo neste caso o Yacc. Tendo como objetivo geral, definido coerentemente ao longo do semestre e na realização dos 3 trabalhos associados à cadeira de Processamento de Linguagens, foi o aumento a experiência de uso do ambiente Linux, da linguagem imperativa C e de algumas ferramentas de apoio à programação.

Capítulo 2

Contextualização

Um compilador é um programa que nos permite transformar código que está num primeiro estado numa linguagem normalmente de nível superior para outra linguagem , servindo assim como um tradutor, é desta maneira que existe a interação entre as linguagens mais percetíveis ao dialeto humano com a linguagem de computador. Na transformação entre Assembly e código-maquina a correspondência de comandos é de 1 para 1 ou seja cada instrução em Assembly é transformada em uma instrução em código -maquina, o mesmo não acontece na transformação de uma linguagem de alto nível para Assembly, sendo este o caso que iremos abordar na realização deste projeto. Podemos então dividir o projeto em duas fases numa primeira um código flex que faz o reconhecimento de cada simbolo encontrado no ficheiro e posteriormente o yacc dependendo do que é passado no flex faz uma acção, esta acção terá funções e estruturas que serão declaradas num modo auxiliar. Vamos entao ter 3 modulos diferentes , em baixo puderemos visualizar o encadeamento deles



Capítulo 3

Análise e Especificação da Linguagem

Foi solicitado a criação de uma linguagem imperativa ao nosso gosto que permite-se e fosse possível uma serie de comandos , de seguida encontram-se descritos as diversas funcionalidades que era necessário que a nossa linguagem reconhece-se e interpreta-se.

- Declaração e manuseamento de variáveis atómicas do tipo inteiro.
- Possibilidade das habituais operações aritméticas, relacionais e lógicas com as variáveis declaras no ponto acima.
- Declaração e manuseamento de variáveis do tipo array de inteiros com 1 ou 2 dimensões.
- Efetuar instruções algoritmicas básicas como a atribuição de expressões a variáveis.
- Leitura do standard input e escrita no standard output
- Efetuar instruções para controlo do fluxo de execução—condicional e cíclica que possam ser aninhadas
- Definir e invocar subprogramas sem parâmetros mas que possam retornar um resultado atómico

Iremos de seguida especificar a linguagem.

3.1 Gramática

Nesta secção iremos definir a nossa linguagem começando por analisar em primeiro lugar os símbolos terminais e não-terminais partindo depois para as produções. Como estudado previamente nas aulas lecionadas uma gramática definidas seguinte maneira, sendo G um gramática que representa uma linguagem imperativa teremos ,

- T o conjunto de simbolos Terminais
- N o conjunto de simbolos nao-terminais
- S representando o axioma da gramática
- P o conjunto de regras de produções

Sendo assim a gramatica descrita pela igualdade, G = <T,N,S,P>. Antes de se passar à secção relativa de cada um dos termos da gramática é de notar que foram declaradas expressões regulares mais concretamente as seguintes:

```
espaco      [ \t]
inteiro     [0-9]+
var         [a-zA-Z_][a-zA-Z0-9_]*
vars        (?i:VARS)
start       (?i:START)
end         (?i:END)
inc         \+\+
dec         \-\-
addass      \+\=
subass      \-\=
mulass      \*\=
divass      \/\=
modass      \%\=
scan        (?i:SCAN)
print       (?i:PRINT)
string      \"([^"]|\\\")*\"
if          (?i:IF)
else        (?i:ELSE)
while       (?i:WHILE)
```

3.1.1 Terminais

Os terminais são símbolos que fazem parte de uma gramática podendo aparecer quer na entrada quer na saída de uma produção , estes mesmo não podem ser derivados em "unidades menores", por decisão do grupo os terminais designados por caracteres terão toda a sua nomenclatura em maiúsculas. Procedeu-se então à descrição dos terminais sendo estes todas as "tags"de uma ordem como ordens de começo , fim, de impressão, de leitura , de inicio de condições de controlo de fluxo , entre outras, e também todos os símbolos quer algébricos referentes a operações aritméticas como símbolos de controlo de dimensões de arrays.

```
T = { 'VARS', 'START', 'END', 'SCAN', 'PRINT', 'IF', 'ELSE', 'WHILE' ,
    STRING, 'ERRO', '(', ')', '{', '}', '[', ']', ';', '"', '"', '+' ,
    '-','*', '/', '%', '<', '>','=', '!', '&', '|', '\n','\$',VAR,INT}
```

3.1.2 Não-Terminais

Os não-terminais são símbolos que fazem parte de uma gramática podendo aparecer apenas na saída de uma produção , estes mesmo podem ser derivados em "unidades menores"podendo estas unidades ser compostas por símbolos terminais e não-terminais, por decisão do grupo os não-terminais designados por caracteres terão toda a sua nomenclatura minúscula podendo ter ou não o primeiro carácter maiúsculo, em termos de não terminais não iremos ter caracteres especiais.

```
N = { Fonte, Cabeca, Corpo, Declaracoes, Declaracao, Instrucoes, instrucao,
      Atribuicao, Escrita, Leitura, Condicional, Else, Ciclo, Exp }
```

3.1.3 Axioma

O axioma é a raiz da árvore de derivação e por onde começa a primeira produção neste caso acabou-se por designar o nosso axioma como Fonte no qual se irá derivar nas duas parte de um ficheiro da nossa linguagem sendo estas a Cabeça e o Corpo. Podemos de seguida visualizar a primeira derivação que existe na nossa gramática , juntamente com a sua produção inicial.

```
S = {Fonte}

 --------------------
|                    |
|       Cabeça       |
|--------------------|
|                    |
|       Corpo        |
|                    |
 --------------------

 Fonte -> Cabeca Corpo
```

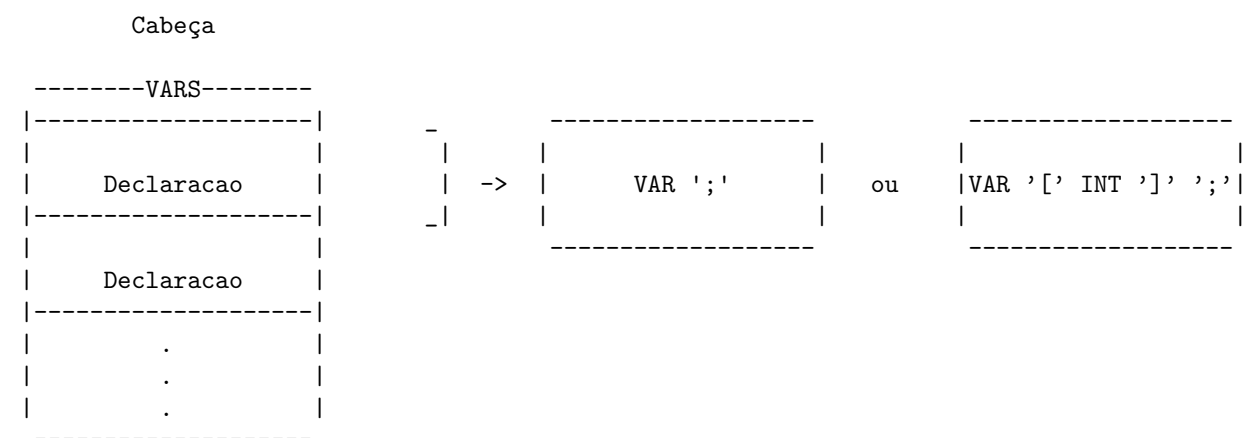
3.1.4 Produções

Após analisados, descritos e declarados todos os conjuntos acima referenciados, vamos agora relacionar os vários conjuntos de símbolos usando produções, as produções são simplesmente regras de derivação em que se parte de algo, z0 e se chega a z1 , ou seja se tivermos a produção z0 -¿ z1 , estamos a declarar que z0 pode-se transformar em z1. Com as produções podemos originar frases gramaticais , através dos resultados que queremos obter criamos as nossas produções , em primeiro lugar iremos expôr todas as produções definidas usando posteriormente a nomenclatura p[nº da produção] para explicar a estratégia e raciocínio.

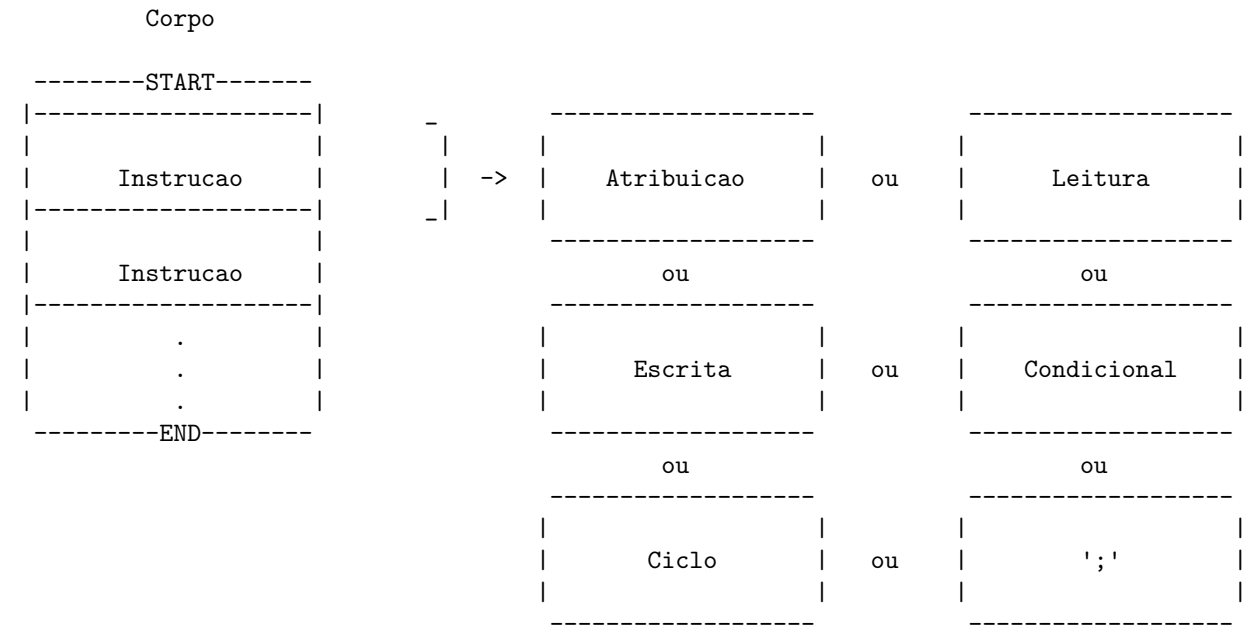
```
P = {
p0:  Cabeca -> VARS Declaracoes
p1:  Declaracoes -> Declaracoes Declaração
p2:  Declaracoes -> Declaracao
p3:  Declaracao -> VAR ';'
p4:  Declaracao -> VAR '[' INT ']' ';'
p5:  Corpo -> Start Instrucoes END
p6:  Instrucoes -> Instrucoes Instrucao
p7:  Instrucoes -> Instrucao
p8:  Instrucao -> Atribuicao
p9:  Instrucao -> Escrita
p10: Instrucao -> Leitura
p11: Instrucao -> Condicional
p12: Instrucao -> Ciclo
p13: Instrucao -> ';'
p14: Atribuicao -> VAR '[' Exp ']' '=' Exp ';'
p15: Atribuicao -> VAR '=' Exp ';'
p16: Atribuicao -> VAR OPINCDEC ';'
p17: Atribuicao -> VAR OPASS Exp ';'
p18: Leitura -> SCAN VAR ';' INPUT
p19: Leitura -> SCAN VAR '[' Exp ']' ';' INPUT
p20: Escrita -> PRINT Exp ';'
p21: Escrita -> PRINT STRING ';'
p22: Condicional -> IF '(' Exp ')' '{' Instrucoes '}' Else
p23: Else -> ELSE '{' Instrucoes '}'
p24: Else -> &
p25: Ciclo -> WHILE '(' Exp ')' '{' Instrucoes '}'
p26: Exp -> NOT Exp
p27: Exp -> Exp OPREL Exp
p28: Exp -> Exp OPRELEQ Exp
p29: Exp -> Exp OPLOG Exp
p30: Exp -> Exp OPADD Exp
p31: Exp -> Exp OPMUL Exp
p32: Exp -> VAR
p33: Exp -> VAR '[' Exp ']'
p34: Exp -> INT
p35: Exp -> '(' Exp ')'
}
```

Como já antes referenciado temos a fonte que é composta pela cabeça e pelo corpo, numa primeira instância relativamente à cabeça é o local onde estarão todas as declarações de variáveis. Teremos então para estar a par dos requisitos do trabalho dois tipo de declaração podendo estes ser quer do tipo variável quer do tipo array.

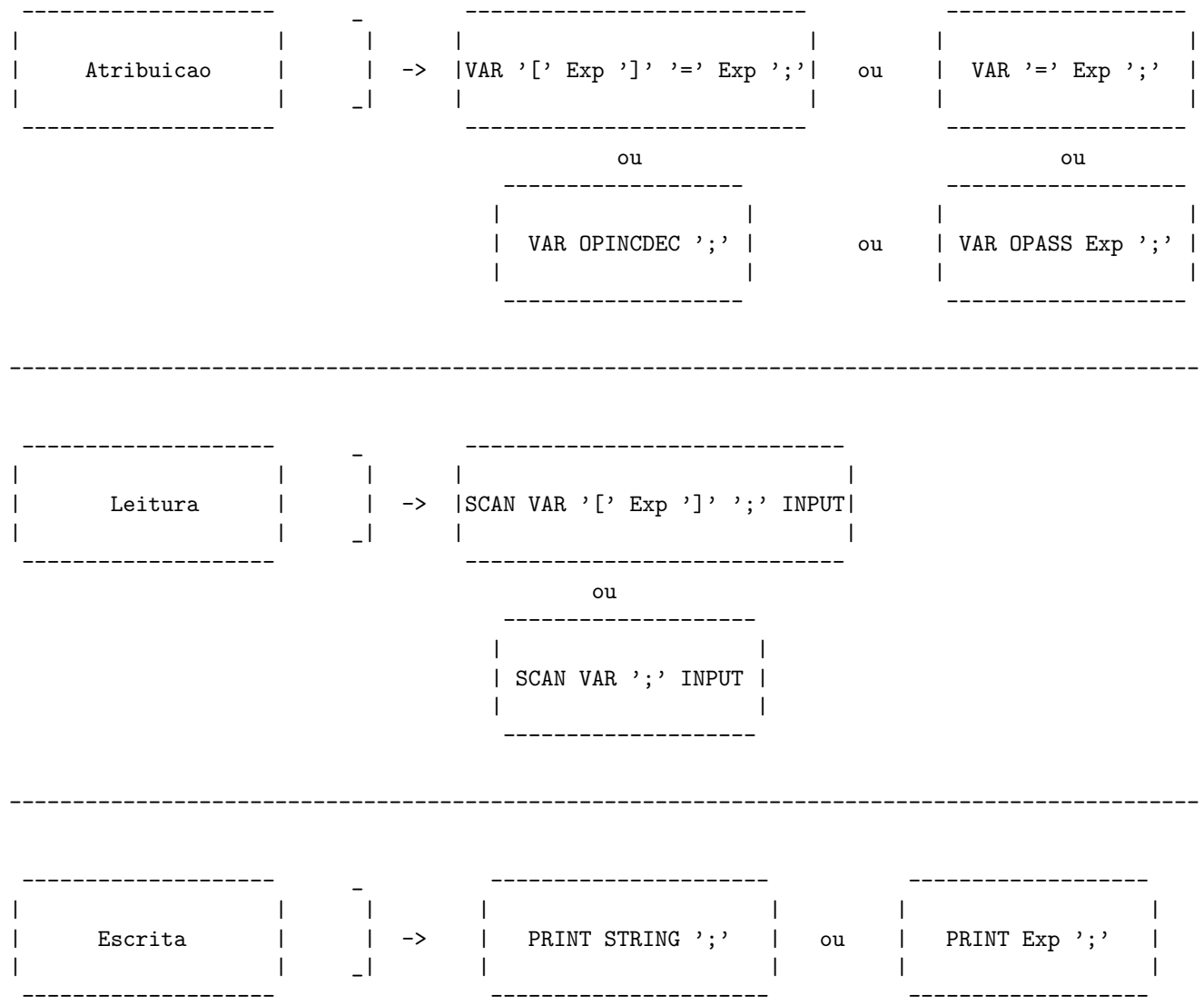
A primeira estratégia aqui apresentada é relativamente as produções p0,p1,p2,p3 e p4.



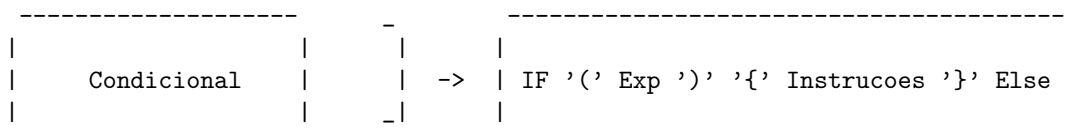
Em segunda instância temos agora o corpo em que utilizamos um técnica em que tem duas limitações uma inicial sendo o START e uma final designada pelo END , entre estas temos um conjuntos de instruções, na nossa linguagem imperativa uma instrução pode ser uma atribuição , escrita leitura, condicional ,ciclo ou o simbolo terminal ';'. Temos agora um esquema em que se ilustra as produções , p5, p6, p7, p8, p9, p10, p11, p12 e p13.

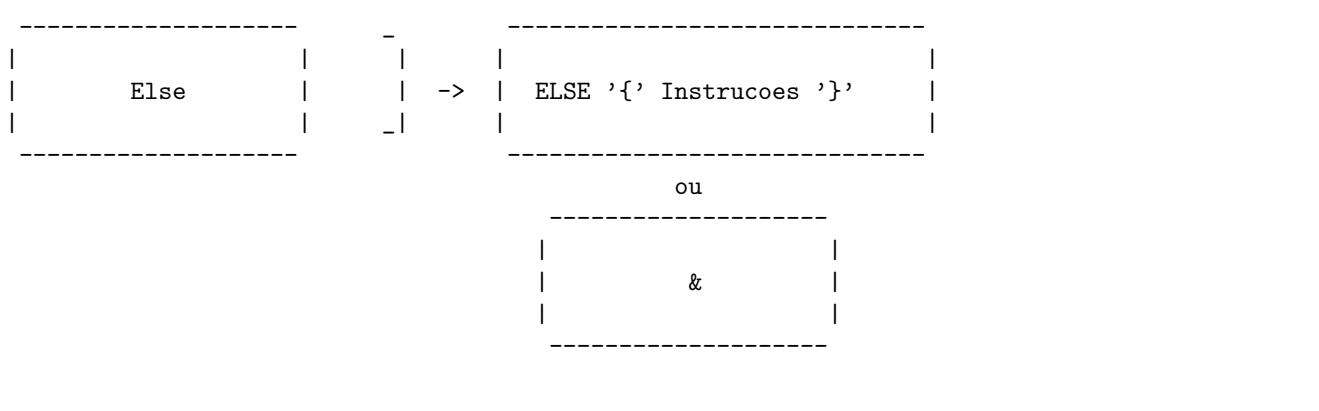


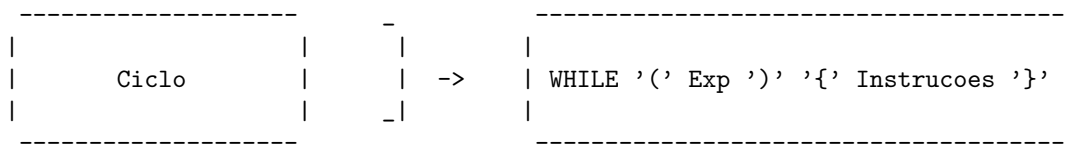
Como podemos ver no esquema acima apresentado podemos ver que das 6 produções geradas a partir da instrução , 5 delas ainda tem simbolos não-terminais , em primeiro temos a atribuicao, que se encontra como simbolo de entrada em 4 produções diferentes sendos estas p14,p15, p16 e p17. Em relação à leitura e escrita cada uma tem duas produções sendo estas a p18, p19, p20 e p21.



De seguida temos a ilustração das reproduções relativamente as expressões de controlo de fluxo sendo esta o condicional e o ciclo while.







Após a exposição de ilustrações que demonstram a linha de pensamento na construção de produções que caracterizam a nossa gramática , ainda falta discutir a estratégia relativamente as ultimas produções que constam no intervalo de p[26,35] , todas estas são relativas ao símbolo Exp que caracteriza uma expressão tendo sido declaradas para as operações aritméticas, manipulação de conhecimento negativo e declaração de variáveis.

Capítulo 4

Implementação

Após a exposição relativamente à linguagem definida e as suas regras passou-se à elaboração de um modulo , ao qual se deu o nome de "Estruturas", onde se definiu as estruturas e as funções que que auxiliam e acompanham o processo de geração de código Assembly. Iremos de seguida expor e explicar as API deste modulo que contem as estruturas que foram necessidades, num total de 4 e as funções que as acompanham. Como estruturas foi preciso definir em primeiro lugar uma estrutura que guardasse as variáveis e as suas características para isso então começou se por definir uma lista ligada em que cada elemento da lista contem informação sobre a variável conseguindo assim ser fiável aquando uma ação é feita sobre uma variável tendo informação sobre o estado das variáveis em relação a sua inicialização , o tipo, o nome e o endereço.

- Variável

```
struct variavel{
char* nome ;
char* tipo ;
int endereco ;
int inicializado ;
 struct variavel *next ;
};
```

A mesma estratégia foi definida também em termos de instruções ao ter o encadeamento de instruções é necessário guardar o endereço de cada uma delas para no final do encadeamento sendo este condicional ou cíclico será necessário ter o endereço de cada uma desta instruções para então saltar corretamente.

- Instrução

```
struct instrucao{
char* instrucao ;
int endereco ;
struct instrucao *next ;
};
```

Como foi requisitado também é necessário que o compilador interprete e gere de forma correta o código Assembly relativamente a aninhamentos como exemplo ilustrativo poderemos visualizar a situação de um simples if dentro de outro if. Para resolver esta situação foi então necessário a criação de duas estruturas auxiliares uma para o condicional e outra para o cíclico que resolve esta situação guardando o endereço para quando da terminação do seu ciclo este saltar corretamente.

- If & While

```
struct ifAddr{
int jz ;
```

```
struct ifAddr *next ;
    };

struct whileAddr{
int jz ;
int jump ;
struct whileAddr *next ;
};
```

Vamos agora explicar e declarar a API de funções que se encontra no ficheiro Estruturas.h juntamente com as estruturas já acima declaradas. Estas funções tais como as estruturas em cima são depois utilizadas no ficheiro yacc dependendo do que é necessário executar conforme cada produção.

- API

  `struct variavel* insertV(struct variavel* variavel , struct variavel *listaV) ;`

- Insere a nova estrutura variável contendo a informação sobre uma variável , na lista ligada passada no segundo argumento da função.

  `int existeV(char* variavel , struct variavel * listaV)`

- Verifica se existe um certa variável numa dada lista de variáveis retornando o valor 0 ou 1.

  `int enderecoV(char* variavel , struct variavel * listaV)`

- Retorna o endereço correspondente ao nome da variável que é passada no primeiro argumento.

  `char* tipoV(char* variavel , struct variavel * listaV)`

- Retorna o tipo correspondente ao nome da variável que é passada no primeiro argumento.

  `void inicializaV(char* variavel , struct variavel * listaV)`

- Altera na lista de variaveis o int inicializa para um caso exista uma variável com o nome que é passado no primeiro argumento.

  `int variavelInicializada(char* variavel , struct variavel * listaV)`

- Retorna o o valor da inicialização correspondente ao nome da variável que é passada no primeiro argumento.

  `struct instrucao* insertI(int endereco , char*instrucao , struct instrucao *listaI)`

- Insere a nova estrutura instruçao contendo o endereço passado no primeiro argumento e o nome passado no segundo argumento , na lista ligada passada no terceiro argumento da função.

  `struct ifAddr* pushIfAddr(int endereco , struct ifAddr *stackIfAddr)`

- Insere na cabeça de uma lista ligada uma nova estrutura do tipo ifAddr contendo o endereço passado no primeiro argumento e apontando para o primeiro elemento da lista ligada passada no segundo argumento.

  `struct ifAddr* popIfAddr(struct ifAddr *stackIfAddr)`

- Passa o apontador da lista para o segundo elemento e liberta o primeiro elemento da lista ligada.

  ```
  void ifJump(int endereco , struct ifAddr *stackIfAddr , struct instrucao *listaI)
  ```

- Procura na lista de instruções passada como terceiro argumento por um endereço que retirou da pilha e lhe coloca o endereço onde se encontra actualmente, mais uma unidade.

  ```
  void elseJump(int endereco , struct ifAddr *stackIfAddr , struct instrucao *listaI)
  ```

- Igual À função descrita anteriormente mudando apenas a tag impressa.

  ```
  struct whileAddr* pushWhileAddr(int endereco , int whileAddr , struct whileAddr *stackWhileAddr)
  ```

- Insere na cabeça de uma lista ligada uma nova estrutura do tipo whileAddr contendo o endereço passado no primeiro argumento e apontando para o primeiro elemento da lista ligada passada no segundo argumento.

  ```
  struct whileAddr* popWhileAddr(struct whileAddr *stackWhileAddr)
  ```

- Passa o apontador da lista para o segundo elemento e liberta o primeiro elemento da lista ligada.

  ```
  int whileJump(int endereco , struct whileAddr *stackWhileAddr , struct instrucao *listaI)
  ```

- Parecida com as funções ja analisadas, ifJump e elseJump mas retorna o outro endereço que se encontra na pilha.

Capítulo 5

Testes e Resultados

Mostram-se a seguir o conjunto testes e respectivos resultados obtidos:

- Ler 4 números e dizer se podem ser os lados de um quadrado.
- Ler um inteiro N, depois ler N números e escrever o menor deles.
- Ler N (constante do programa) números e calcular e imprimir o seu produtório.
- Contar e imprimir os números impares de uma sequência de números naturais.
- Ler e armazenar os elementos de um vetor de comprimento N; imprimir os valores por ordem decrescente após fazer a ordenação do array por trocas diretas.
- Contar e imprimir os números impares de uma sequência de números naturais.
- ler e armazenar N números num array; imprimir os valores por ordem inversa.

5.1 quadrado.c

```
                // --- quadrado.c -- //

VARS
a[4] ;
aux ;
res ;
i ;
START

PRINT "Insira 4 números ";

i=0;
while(i < 4)
{
PRINT "Insira número :";
SCAN a[i];
>4
i++;
}

aux=a[0];
```

```
i=0 ;
res=1;
while(i < 4)
{
i++;
if(a[i] == aux)
{
res = 0;
}
}

if (res == 1)
{
PRINT "É QUADRADO!";
}
else
{
PRINT "NAO É QUADRADO!";
}

END

_000000: PUSHN 4
_000001: PUSHI 0
_000002: PUSHI 0
_000003: PUSHI 0
_000004: START
_000005: PUSHS "Insira 4 números "
_000006: WRITES
_000007: PUSHI 0
_000008: STOREG 6
_000009: PUSHG 6
_000010: PUSHI 4
_000011: INF
_000012: JZ_000026
_000013: PUSHS "Insira número :"
_000014: WRITES
_000015: PUSHI 0
_000016: PUSHG 6
_000017: READ "4"
_000018: ATOI
_000019: STOREN
_000020: PUSHG 6
_000021: PUSHI 1
_000022: ADD
_000023: SUB
_000024: STOREG 6
_000025: JUMP 9
_000026: PUSHI 0
_000027: PUSHI 0
_000028: LOADN
_000029: STOREG 4
_000030: PUSHI 0
_000031: STOREG 6
```

```
_000032: PUSHI 1
_000033: STOREG 5
_000034: PUSHG 6
_000035: PUSHI 4
_000036: INF
_000037: JZ_000052
_000038: PUSHG 6
_000039: PUSHI 1
_000040: ADD
_000041: SUB
_000042: STOREG 6
_000043: PUSHI 0
_000044: PUSHG 6
_000045: LOADN
_000046: PUSHG 4
_000047: EQUAL
_000048: JZ _000052
_000049: PUSHI 0
_000050: STOREG 5
_000051: JUMP 34
_000052: PUSHG 5
_000053: PUSHI 1
_000054: EQUAL
_000055: JZ _000059
_000056: PUSHS "É QUADRADO!"
_000057: WRITES
_000058: JUMP _000061
_000059: PUSHS "NAO É QUADRADO!"
_000060: WRITES
_000061: STOP
```

5.2 menor.c

```
// --- menor.c -- //

VARS

N;
min;
a;
i;

START

min=20;

PRINT "Insira N";
SCAN N;
>3

i=0;
while(i < N)
{
```

```
PRINT "Insira número :";
SCAN a;
>X
if (a < min)
{
min=a;
}
i++;
}

PRINT "O MINIMO INSERIDO É";
PRINT min;

END


_000000: PUSHI 0
_000001: PUSHI 0
_000002: PUSHI 0
_000003: PUSHI 0
_000004: START
_000005: PUSHI 20
_000006: STOREG 1
_000007: PUSHS "Insira N"
_000008: WRITES
_000009: READ "3"
_000010: ATOI
_000011: STOREG 0
_000012: PUSHI 0
_000013: STOREG 3
_000014: PUSHG 3
_000015: PUSHG 0
_000016: INF
_000017: JZ_000035
_000018: PUSHS "Insira número :"
_000019: WRITES
_000020: READ "4"
_000021: ATOI
_000022: STOREG 2
_000023: PUSHG 2
_000024: PUSHG 1
_000025: INF
_000026: JZ _000030
_000027: PUSHG 2
_000028: STOREG 1
_000029: PUSHG 3
_000030: PUSHI 1
_000031: ADD
_000032: SUB
_000033: STOREG 3
_000034: JUMP 14
_000035: PUSHS "O MINIMO INSERIDO É"
_000036: WRITES
_000037: PUSHG 1
```

```
_000038: WRITEI
_000039: STOP
```

5.3 produtorio.c

```
// --- produtorio.c -- //

VARS

N;
total;
i;
a;

START

total=1;
PRINT "Insira N";
SCAN N;
>10

i=0;
while(i < N)
{
PRINT "Insira número :";
SCAN a;
>X
total*=a;
}

PRINT "O PRODUTORIO É";
PRINT total;

END


_000000: PUSHI 0
_000001: PUSHI 0
_000002: PUSHI 0
_000003: PUSHI 0
_000004: START
_000005: PUSHI 1
_000006: STOREG 1
_000007: PUSHS "Insira N"
_000008: WRITES
_000009: READ "10"
_000010: ATOI
_000011: STOREG 0
_000012: PUSHI 0
_000013: STOREG 2
_000014: PUSHG 2
_000015: PUSHG 0
_000016: INF
```

```
_000017: JZ_000028
_000018: PUSHS "Insira número :"
_000019: WRITES
_000020: READ "X"
_000021: ATOI
_000022: STOREG 3
_000023: PUSHG 1
_000024: PUSHG 3
_000025: MUL
_000026: STOREG 1
_000027: JUMP 14
_000028: PUSHS "O PRODUTORIO É"
_000029: WRITES
_000030: PUSHG 1
_000031: WRITEI
_000032: STOP
```

5.4 impares.c

```
// --- impares.c -- //

VARS

N;
total;
i;
a;

START

total=0;
PRINT "Insira N";
SCAN N;
>10

i=0;
while(i < N)
{
PRINT "Insira número :";
SCAN a;
>X
if( a % 2 != 0)
{
PRINT a;
PRINT "É ÍMPAR!";
}
i++;
total++;
}

PRINT "O TOTAL DE IMPARES É";
PRINT total;
```

```
END


_000000: PUSHI 0
_000001: PUSHI 0
_000002: PUSHI 0
_000003: PUSHI 0
_000004: START
_000005: PUSHI 0
_000006: STOREG 1
_000007: PUSHS "Insira N"
_000008: WRITES
_000009: READ "10"
_000010: ATOI
_000011: STOREG 0
_000012: PUSHI 0
_000013: STOREG 2
_000014: PUSHG 2
_000015: PUSHG 0
_000016: INF
_000017: JZ_000045
_000018: PUSHS "Insira número :"
_000019: WRITES
_000020: READ "X"
_000021: ATOI
_000022: STOREG 3
_000023: PUSHG 3
_000024: PUSHI 2
_000025: MOD
_000026: PUSHI 0
_000027: EQUAL
_000028: NOT
_000029: JZ _000035
_000030: PUSHG 3
_000031: WRITEI
_000032: PUSHS "É ÍMPAR!"
_000033: WRITES
_000034: PUSHG 2
_000035: PUSHI 1
_000036: ADD
_000037: SUB
_000038: STOREG 2
_000039: PUSHG 1
_000040: PUSHI 1
_000041: ADD
_000042: SUB
_000043: STOREG 1
_000044: JUMP 14
_000045: PUSHS "O TOTAL DE IMPARES É"
_000046: WRITES
_000047: PUSHG 1
_000048: WRITEI
_000049: STOP
```

5.5 array1.c

```
// --- array1.c -- //

VARS

i;
a[10] ;
aux;
N;

START

N=10;
i=0;
while(i < N)
{
PRINT "Insira número :";
SCAN a[i];
>X
i++;
}

i=N-1;
while(i>=0){
if(a[i+1]<a[i]){
aux=a[i+1];
a[i+1]=a[i];
a[i]=aux;
}
}

i=0;
while(i<N)
{
PRINT a[i];
i++;
}


END


_000000: PUSHI 0
_000001: PUSHN 10
_000002: PUSHI 0
_000003: PUSHI 0
_000004: START
_000005: PUSHI 10
_000006: STOREG 12
```

```
_000007: PUSHI 0
_000008: STOREG 0
_000009: PUSHG 0
_000010: PUSHG 12
_000011: INF
_000012: JZ_000026
_000013: PUSHS "Insira número :"
_000014: WRITES
_000015: PUSHI 1
_000016: PUSHG 0
_000017: READ "X"
_000018: ATOI
_000019: STOREN
_000020: PUSHG 0
_000021: PUSHI 1
_000022: ADD
_000023: SUB
_000024: STOREG 0
_000025: JUMP 9
_000026: PUSHG 12
_000027: PUSHI 1
_000028: SUB
_000029: STOREG 0
_000030: PUSHG 0
_000031: PUSHI 0
_000032: SUPEQ
_000033: JZ_000063
_000034: PUSHI 1
_000035: PUSHG 0
_000036: PUSHI 1
_000037: ADD
_000038: LOADN
_000039: PUSHI 1
_000040: PUSHG 0
_000041: LOADN
_000042: INF
_000043: JZ _000063
_000044: PUSHI 1
_000045: PUSHG 0
_000046: PUSHI 1
_000047: ADD
_000048: LOADN
_000049: STOREG 11
_000050: PUSHI 1
_000051: PUSHG 0
_000052: PUSHI 1
_000053: ADD
_000054: PUSHI 1
_000055: PUSHG 0
_000056: LOADN
_000057: STOREN
_000058: PUSHI 1
_000059: PUSHG 0
_000060: PUSHG 11
```

```
_000061: STOREN
_000062: JUMP 30
_000063: PUSHI 0
_000064: STOREG 0
_000065: PUSHG 0
_000066: PUSHG 12
_000067: INF
_000068: JZ_000079
_000069: PUSHI 1
_000070: PUSHG 0
_000071: LOADN
_000072: WRITEI
_000073: PUSHG 0
_000074: PUSHI 1
_000075: ADD
_000076: SUB
_000077: STOREG 0
_000078: JUMP 65
_000079: STOP
```

5.6 array2.c

```
// --- array2.c -- //

VARS

i;
a[10] ;
N;

START

N=10;
i=0;
while(i < N)
{
PRINT "Insira número :";
SCAN a[i];
>X
i++;
}


i=N;
while(i>0)
{
PRINT a[i];
i--;
}
```

```
END


_000000: PUSHI 0
_000001: PUSHN 10
_000002: PUSHI 0
_000003: START
_000004: PUSHI 10
_000005: STOREG 11
_000006: PUSHI 0
_000007: STOREG 0
_000008: PUSHG 0
_000009: PUSHG 11
_000010: INF
_000011: JZ_000025
_000012: PUSHS "Insira número :"
_000013: WRITES
_000014: PUSHI 1
_000015: PUSHG 0
_000016: READ "X"
_000017: ATOI
_000018: STOREN
_000019: PUSHG 0
_000020: PUSHI 1
_000021: ADD
_000022: SUB
_000023: STOREG 0
_000024: JUMP 8
_000025: PUSHG 11
_000026: STOREG 0
_000027: PUSHG 0
_000028: PUSHI 0
_000029: SUP
_000030: JZ_000039
_000031: PUSHI 1
_000032: PUSHG 0
_000033: LOADN
_000034: WRITEI
_000035: PUSHG 0
_000036: PUSHI 1
_000037: STOREG 0
_000038: JUMP 27
_000039: STOP
```

Capítulo 6

Conclusão

Após a realização deste trabalho consolidou-se a matéria referente a yacc, flex e consequentemente implementação de uma gramática tendo sido cumprido todos os objetivos referidos no enunciado do trabalho pratico. Em termos de dificuldades existiram algumas tendo sido ultrapassadas com a cooperação do grupo tendo sido uma delas por exemplo a importância da ordem das operações algébricas sobre variáveis. Neste ultimo trabalho ao contrario aos outros dois não se realizou nenhuma tarefa ou objetivo extra. Com este trabalho completou-se assim um ciclo de três trabalhos realizados no âmbito da cadeira de Processamento de Linguagens.