

Teoria de Linguagens e Compilação
1º ano - MEFis

Trabalho Prático nº 2 (Gramáticas, Compiladores)

Grupo 102

João Gomes
PG47339

Alexandre Silva
PG46949

Maria Inês Dias
PG47474

Ricardo Correia
PG47607

21 de janeiro de 2022

Resumo

Este documento é relativo ao Trabalho Prático nº2 da U.C. Teoria das Linguagens e Compilação. Este foca-se na criação de uma linguagem imperativa, à qual chamamos de ***Suricata***, e de um compilador para essa mesma linguagem. Este último deve gerar código para uma máquina de stack virtual.

Para a elaboração do projeto começou-se por criar a gramática independente do contexto, de seguida o analisador léxico, a conversão da GIC para Python e por fim o compilador yacc.

Neste relatório explica-se detalhadamente todos os passos do projeto e as decisões tomadas.

Conteúdo

1	Introdução	4
1.1	Enquadramento e Contexto	4
1.2	Problema e Objetivo	4
1.3	Abordagem e Decisões	5
1.4	Estrutura do Relatório	6
2	Análise e Especificação	7
2.1	Descrição informal do problema	7
2.2	Especificação de Requisitos	7
3	Gramática independente do contexto (GIC)	8
4	Analizador Léxico	13
5	Conversão da GIC para Python	17
6	Compilador Yacc	22
6.1	Declarações:	22
6.2	Funções auxiliares:	25
6.3	Instruções:	26
6.3.1	Atribuições:	27
6.3.2	Estruturas condicionais:	32
6.3.3	Instruções das estruturas condicionais e cíclicas:	33
6.3.4	Ciclo:	33
6.3.5	Imprimir:	34
6.3.6	Expressões:	34
6.3.7	Condições:	40
7	Testes	41
7.1	Exemplos propostos no enunciado do trabalho	41
7.1.1	Exemplo 1:	41
7.1.2	Exemplo 2:	44
7.1.3	Exemplo 3:	47
7.1.4	Exemplo 4:	49

7.1.5	Exemplo 5:	54
7.2	Exemplos de casos de erro	57
7.2.1	Variável não declarada	57
7.2.2	Redeclaração de uma variável	57
7.2.3	Variável do tipo VarInt com indexação	59
7.2.4	Variável do tipo ArrayInt sem indexação	60
7.2.5	ArrayInt unidimensional com 2 índices	61
7.2.6	Erro léxico:	62
7.2.7	Erro sintático:	64
8	Conclusão	66
9	Código	67
9.1	Analizador Léxico	67
9.2	Compilador Yacc	70

Lista de Figuras

7.1	<i>Aviso no terminal (Exemplo 1).</i>	42
7.2	<i>Exemplo de um caso em que são lados de um quadrado</i>	44
7.3	<i>Exemplo de um caso em que não são lados de um quadrado</i>	44
7.4	<i>Exemplo onde comparamos apenas 2 números</i>	47
7.5	<i>Exemplo onde comparamos 4 números</i>	47
7.6	<i>Multiplicação de 3 números(1,2,3)</i>	49
7.7	<i>Aviso no terminal (Exemplo 4).</i>	50
7.8	<i>Resultado obtido na VM para uma sequência de 5 números.</i>	53
7.9	<i>Resultado obtido na VM para uma sequência de 0 números.</i>	53
7.10	<i>Resultado obtido na VM para uma sequência de 1 número.</i>	54
7.11	<i>Aviso no terminal (Exemplo 5).</i>	55
7.12	<i>Resultado obtido na VM para o caso de $N = 4$ (a figura 7.12b é a continuação da figura 7.12a).</i>	56
7.13	<i>Resultado obtido na VM para o caso de $N = 3$ (a figura 7.13b é a continuação da figura 7.13a).</i>	57
7.14	<i>Resultado apresentado no terminal</i>	58
7.15	<i>Erro obtido no terminal, proveniente da redeclaração de uma variável VarInt.</i>	58
7.16	<i>Erro obtido no terminal, proveniente da indexação de uma variável VarInt.</i>	59
7.17	<i>Erro obtido no terminal, proveniente de não indexar uma variável ArrayInt.</i>	61
7.18	<i>Erro obtido no terminal, proveniente de usar dois índices para indexar um array unidimensional.</i>	62
7.19	<i>Erros e informações obtidas no terminal, proveniente de um programa com erro léxico.</i>	63
7.20	<i>Erros e informações obtidas no terminal, proveniente de um programa com erro sintático.</i>	65

Capítulo 1

Introdução

1.1 Enquadramento e Contexto

Área: Teoria das Linguagens e Compilação

O tema central deste trabalho prático, Gramáticas e Compiladores, enquadra-se na U.C. Teoria das Linguagens e Compilação. Tal como referido antes, este trabalho tem como objetivo elaborar uma linguagem imperativa simples, e também um compilador responsável por gerar código para uma máquina de stack virtual.

1.2 Problema e Objetivo

A elaboração deste trabalho consistia na criação de uma linguagem imperativa a nosso gosto, a qual fosse capaz de cumprir os seguintes requisitos:

- Declarar variáveis do tipo *inteiro*.
- Efetuar instruções algorítmicas básicas como a atribuição do valor de expressões numéricas a variáveis.
- Ler do *standard input* e escrever no *standard output*.
- Efetuar *instruções condicionais* para controlo do fluxo de execução.
- Efetuar *instruções cíclicas* para controlo do fluxo de execução (*repeat-until*).
- *Opcional*: declarar e manusear variáveis estruturadas do tipo *array* (a 1 ou 2 dimensões) de inteiros, em relação aos quais é apenas permitida a operação de *indexação* (índice inteiro).
- As variáveis são declaradas no **início do programa** e não pode haver **re-declarações**.
- As variáveis não podem ser utilizadas sem declaração prévia.
- Na declaração, se nada for explicitado, a variável toma o valor **0**.

Este trabalho tinha como objetivos:

- Aumentar a experiência em engenharia de linguagens e em programação generativa (gramatical), reforçando a capacidade de escrever gramáticas, quer independentes de contexto (GIC), quer tradutoras (GT);
- Desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe;
- Desenvolver um compilador gerando código para uma máquina de stack virtual;
- Utilizar geradores de compiladores baseados em gramáticas tradutoras, concretamente o Yacc, versão PLY do Python, completado pelo gerador de analisadores léxicos Lex, também versão PLY do Python.

1.3 Abordagem e Decisões

Durante a realização deste trabalho, procuramos sempre utilizar conhecimentos adquiridos nas aulas de forma a tornar a elaboração deste projecto mais simples.

Uma das estratégias adoptadas foi começar por definir o aspecto que gostaríamos de dar à nossa linguagem. Tendo o aspecto definido, partiu-se para a elaboração da **gramática independente de contexto (GIC)**, a qual foi alvo de várias alterações com o decorrer do projecto, pois havia sempre pormenores a ter em conta. No final de realizar a GIC elaboramos o **analisador léxico** que melhor se adequava à nossa gramática, partindo logo de seguida para a **conversão** da nossa GIC para código **Python**.

Tendo a conversão realizada, dedicamos grande parte do tempo a elaborar o **compilador yacc**, pois era necessário **detectar erros** e gerar **código máquina** capaz de fazer o pretendido na máquina de stack virtual.

Terminado isto tudo, realizamos vários testes e fomos aperfeiçoando vários pontos que achamos que podiam ser melhorados até obter o resultado pretendido.

Outros aspectos que tivemos em conta foi a separação entre as **declarações** e as **instruções**. Na nossa linguagem existe um bloco na qual se faz as declarações sem qualquer atribuição e durante as instruções não é permitido declarar nenhuma variável. Esta ideia de separar as declarações e as instruções foi um dos aspectos aconselhados pelos docentes da UC, a qual executamos com êxito. Para além desses aspectos, permitimos que a nossa linguagem não contenha declarações, uma vez que podem não ser utilizadas para programas mais elementares. No entanto, não permitimos que não haja instruções, uma vez que para nós não faz sentido um utilizador escrever um programa sem qualquer instrução.

Tivemos também um especial cuidado na **detecção de erros**, como **variáveis não declaradas**, **erros de tipo** e **variáveis redeclaradas**. Sempre que o programa escrito pelo utilizador contivesse erros léxicos, ou sintáticos, ou até mesmo semânticos, decidimos não permitir a geração de código **Assembly** para um ficheiro .vm, uma vez que esse tipo de erros em "compile time" devem ser detetados pelo compilador e impressas informações referentes a eles, e consideramos que não devia ser gerado um ficheiro com código Assembly que permitisse executar o programa com erros na máquina virtual.

Outro cuidado especial que tivemos foi na elaboração das **expressões**, uma vez que pretendíamos desde o início realizar operações aritméticas, relacionais, lógicas, introduzir inteiros, variáveis e uma combinação de todos estes pontos de forma a tornar as expressões mais dinâmicas e com poucas restrições.

Como extra decidimos implementar a elaboração de **comentários** na nossa linguagem, pois a nosso ver uma linguagem deve permitir ao utilizador comentar o código de forma a uma melhor leitura de terceiros e até mesmo do próprio utilizador.

Por fim, tivemos um especial cuidado na elaboração da gramática, na qual se realizou **recursividade à esquerda**, uma vez que é mais eficiente e foi recomendado pelos docentes.

1.4 Estrutura do Relatório

Este relatório possui 9 capítulos.

O **capítulo 1** consiste na **Introdução**, na qual se faz um breve enquadramento do trabalho, refere-se os objectivos e problemas propostos no mesmo, as abordagens tomadas, e é referida a estrutura do relatório.

O **capítulo 2** aborda a parte da **análise e especificação**.

Os **capítulos 3, 4, 5 e 6** consistem na explicação detalhada do **GIC**, **analizador léxico**, **conversão da GIC para Python** e do **compilador yacc**, respectivamente. Explicam, portanto, a conceção da resolução.

O **capítulo 7** consiste na realização de **testes**, tanto com erros propositados como sem erros, nos quais se elabora programas capazes de responder aos exercícios propostos no enunciado do trabalho.

O **capítulo 8** consiste numa breve **conclusão**, onde é feita uma análise geral do trabalho.

Por fim, no **capítulo 9** está presente todo o **código** realizado pelo grupo.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

O problema inerente a este trabalho prático é conseguir definir uma linguagem de programação, possibilitando que os programas escritos nessa linguagem sejam traduzidos para código máquina, de modo a ser possível executá-los na máquina virtual VM.

2.2 Especificação de Requisitos

Os requisitos principais que tiveram de ser cumpridos para conseguir resolver o problema inerente a este trabalho prático foram: desenvolver uma **Gramática Independente do Contexto (GIC)** que permitisse satisfazer os requisitos descritos na secção 1.2, e que também fosse concordante com o aspeto e com as restantes características pretendidas para a nossa linguagem; especificar um **Analisador Léxico** que permitisse construir a linguagem pretendida; converter a GIC criada em funções de Python; a partir destas funções, definir como é realizada a tradução do código escrito em *Suricata* para código **Assembly**.

Capítulo 3

Gramática independente do contexto (GIC)

Neste capítulo, abordar-se-á a **gramática independente do contexto**. De notar que, na gramática da nossa linguagem, os símbolos terminais começam com letra maiúscula e os símbolos não terminais começam com letra minúscula.

Esta, inicia-se com o símbolo não terminal *programa* que "diverge em 2 caminhos", nas declarações, que corresponde ao símbolo não terminal *declaracoes*, e nas instruções, que tem o símbolo não terminal *statements*.

```
programa : declaracoes statements
```

Na nossa linguagem é permitido que não haja declaração de variáveis, porém, caso haja declaração de variáveis, estas têm de obdecer às seguintes regras:

- Apenas é permitido declarar dois tipos de variáveis, sendo elas variáveis do tipo inteiro (tipo *VarInt*) e variáveis do tipo array de inteiros (tipo *ArrayInt*). Nas variáveis do tipo *ArrayInt* é permitido criar arrays unidimensionais e bidimensionais;
- Para declarar uma variável é necessário começar sempre com o seu tipo (*VarInt* ou *ArrayInt*);
- Após o tipo da variável, é obrigatório atribuir-lhe um nome;
- No caso da variável ser do tipo *VarInt*, não é permitido utilizar parênteses retos a seguir ao nome tal como é feito nos arrays.
- No caso da variável ser do tipo *ArrayInt* é obrigatório indicar após o nome a sua dimensão. Para o caso dos arrays unidimensionais, a dimensão é indicada introduzindo um número inteiro entre parênteses retos, enquanto que para os arrays bidimensionais são criados dois pares de parênteses retos, na qual dentro de cada um se coloca um inteiro;
- No caso de um array bidimensional, o primeiro número inteiro entre parênteses retos representa o número de arrays unidimensionais contidos nesse array bidimensional, e o segundo número inteiro entre parênteses retos representa o número de elementos contidos em cada um desses arrays unidimensionais. Por outras palavras, o primeiro número inteiro entre parênteses retos é relativo ao número de linhas e o segundo número inteiro entre parênteses retos é relativo ao número de colunas do array bidimensional. Por exemplo, se um array bidimensional tiver [2][3] na sua declaração, então tem a estrutura [[a,b,c],[d,e,f]].

- No final de cada declaração é obrigatório colocar o caracter '|', que representa o **final de frase**;
- Não é permitido fazer atribuições às variáveis no bloco de declarações, devido a isso todas as variáveis são iniciadas a **zero por default**.
- É permitido escrever **comentários**.

Uma vez já referida as regras das declarações, segue-se o resto da gramática das declarações criada por nós:

```

declaracoes :
    | declaracoes declaracao

declaracao : declVar
    | declArray

declVar : VarInt listIds '|'

listIds : ID
    | listIds ',' ID

declArray : ArrayInt listIdArray '|'

listIdArray : array
    | listIdArray ',' array

array : ID '[' Num ']'
    | ID '[' Num ']' '[' Num ']'

```

ID é o símbolo terminal correspondente às palavras que podem identificar variáveis, e **Num** é o símbolo terminal correspondente aos números não negativos. De notar, tendo em conta as derivações possíveis dos símbolos não terminais **listIds** e **listIdArray**, que é possível fazer várias declarações de inteiros escrevendo "VarInt" uma única vez, e fazer várias declarações de arrays de inteiros escrevendo "ArrayInt" uma única vez. Por outro lado, também é possível escrever o tipo da variável em cada declaração de uma variável, tal como se pode verificar no seguinte exemplo escrito na linguagem Suricata:

```

VarInt x,y,z|
ArrayInt a[2],b[3][8],c[3]|
VarInt ola|
ArrayInt xau[3]|

```

Uma vez já explicado e mostrado como se realizou a gramática referente às declarações, será agora explicado como se elaborou a gramática referente às instruções.

```

statements : Inicio ':' instrucoes Fim

```

Após a declaração das variáveis, é necessário escrever a a palavra 'início:' que indica o **início das instruções**, de seguida escreve-se as instruções pretendidas pelo utilizador e no final escreve-se a palavra 'fim' que indica o **final das instruções**.

Para a elaboração das instruções, é necessário ter em conta o que se pode fazer e as regras para a sua criação:

- Não é permitido que não sejam realizadas instruções;
- Todas as instruções têm de terminar com o simbolo '|', à exceção dos **ciclos** e das instruções **condicionais**;
- É permitido realizar atribuições, ciclos **Repeat-Until**, instruções condicionais e escrever no **standart output**;
- Na nossa linguagem, o manuseio de arrays apenas permite a operação de **indexação**;
- No que toca a atribuições, pode-se atribuir valores inteiros às variáveis declaradas, também se pode atribuir valores provenientes do **standard input** através da expressão 'Ler()', atribuir valores provenientes das variáveis declaradas e atribuir o resultado de operações aritméticas, relacionais, lógicas, ou uma mistura de tudo isto. Portanto, às variáveis declaradas, pode-se atribuir ou valores provenientes do standard input, ou valores de expressões;
- Nas instruções **condicionais** existe dois tipos, o 'Se(condições){instruções}' e o 'Se(condições){instruções}Senao{instruções}', na qual as intruções podem ser qualquer uma das referidas anteriormente;
- Nos **ciclos** apenas é permitido o 'Repete{instruções}Ate(condições)', que é equivalente ao **Repeat-Until**;
- Para escrever no standart output utiliza-se 'Imprimir(valor ou string)', sendo que é possível escrever no standart output uma string ou um valor proveniente de uma expressão, como uma operação aritmética, relacional, lógica, um valor de uma variável declarada, ou simplesmente um inteiro;
- Nas condições é admitido utilizar expressões aritméticas, relacionais, lógicas, inteiros, variáveis declaradas e até mesmo uma mistura de tudo isto. Portanto, as condições podem correspondem a uma qualquer expressão da linguagem, isto é, a qualquer frase que derive do símbolo **expressao**;
- É permitido escrever **comentários**.

Uma vez já referido o que se pode fazer e quais a regras das instruções, aqui vai o que falta da gramática das instruções:

```
instrucoes : instrucao
           | instrucoes instrucao

instrucao : atribuicao
          | se
          | imprimir
          | ciclo
```

```

atribuicao : ID '=' expressao '|'
           | ID '=' Ler '(' ')' '|'
           | ID '=' '[' expressao ']' '=' expressao '|'
           | ID '=' '[' expressao ']' '=' Ler '(' ')' '|'
           | ID '=' '[' expressao ']' '[' expressao ']' '=' expressao '|'
           | ID '=' '[' expressao ']' '[' expressao ']' '=' Ler '(' ')' '|'

se : Se '(' condicoes ')' '{' conteudoSeRep '}'
    | Se '(' condicoes ')' '{' conteudoSeRep '}' Senao '{' conteudoSeRep '}'

ciclo : Repete '{' conteudoSeRep '}' Ate '(' condicoes ')'

```

Realça-se o facto de as instruções dentro dos ciclos e das instruções condicionais, correspondentes ao símbolo não terminal *conteudoSeRep*, poderem ser vazias:

```

conteudoSeRep :
    | instrucoes

imprimir : Imprimir '(' expressao ')' '|'
           | Imprimir '(' String ')' '|'

elemento : ID
           | ID '[' expressao ']'
           | ID '[' expressao ']' '[' expressao ']'

```

Realça-se também o facto de que nas operações aritméticas são válidas apenas a soma, subtração, divisão inteira, multiplicação e resto da divisão inteira. Nas operações relacionais são válidas apenas o maior, menor, maior ou igual, menor ou igual, igual e diferente. Por fim, nas operações lógicas apenas são válidas o And, Or e Not.

De notar que o símbolo não terminal *num_pos_neg* corresponde a números inteiros com qualquer valor.

```

expressao : expressao '+' termo
           | expressao '-' termo
           | termo
           | expressao And termo
           | expressao Or termo
           | expressao Igual termo
           | expressao Diferente termo
           | expressao MaiorIgual termo
           | expressao MenorIgual termo
           | Not '(' expressao ')'
           | expressao '<' termo
           | expressao '>' termo

termo : termo '*' fator
        | termo '/' fator
        | termo '%' fator
        | fator

```

Uma das derivações do símbolo não terminal *fator* é *'(expressao)'*, colocou-se desta forma para permitir que se coloquem parênteses para separar operações, como por exemplo $(2 * 4) / (2 - 5)$. Assim torna-se mais organizada e legível a escrita de expressões.

```
fator : num_pos_neg  
      | elemento  
      | '( expressao )'
```

```
num_pos_neg : Num_neg  
            | Num
```

```
condicoes : expressao
```

Capítulo 4

Analizador Léxico

Neste capítulo vamos proceder à descrição do **analizador léxico** que usamos no nosso trabalho.

O analisador léxico como vimos nas aulas serve de "fundamento" para a nossa linguagem convertendo deste modo uma sequência de caracteres em uma sequência de **tokens**.

Começamos então por definir os **states** (para podermos mais à frente realizar comentários), a lista de **tokens** (ou seja, a lista de símbolos terminais da GIC) e a lista de **literals** (são caracteres que ao serem encontrados pelo lexer, são retornados "como estão").

```
states=[('comentario','inclusive')]
```

```
tokens = ['OFF','IN','VarInt', 'ArrayInt', 'Inicio', 'Fim', 'Se', 'Senao', 'Repete', 'Ate',  
'Ler', 'Imprimir', 'String','Num', 'Num_neg','ID', 'Maiorouigual','Menorouigual','Igualigual',  
'Diferente','And','Or','Not']
```

```
literals = ['*', '+', '%', '/', '-', '=', '(', ')', '.', '<', '>', ',', '{', '}', '[', ']', '|',  
, ':']
```

Definido os states, tokens e literals vamos agora construir uma função para cada token e usando expressões regulares indentificaremos os caracteres que devem ser associados a cada token:

```
def t_And(t):  
    r'\\/'  
    return t
```

Neste caso o nosso analisador vai associar o **And** sempre que encontrar os caracteres `"/`.

```
def t_Or(t):  
    r'\\//'  
    return t
```

Agora para o token **Or** vai associar os caracteres `"/`.

As próximas 15 funções seguem uma estratégia semelhante às duas anteriores, devido a isso não serão explicadas com detalhe.

```

def t_Not(t):
    r'!'
    return t

def t_Maiorouigual(t):
    r'>='
    return t

def t_Igualigual(t):
    r'=='
    return t

def t_Diferente(t):
    r'!='
    return t

def t_Menorouigual(t):
    r'<='
    return t

def t_VarInt(t):
    r'VarInt'
    return t

def t_ArrayInt(t):
    r'ArrayInt'
    return t

def t_Inicio(t):
    r'inicio'
    return t

def t_Fim(t):
    r'fim'
    return t

def t_Senao(t):
    r'Senao'
    return t

def t_Se(t):
    r'Se'
    return t

def t_Repete(t):
    r'Repete'

```



```

        return t

def t_Ate(t):
    r'Ate'
    return t

def t_Ler(t):
    r'Ler'
    return t

def t_Imprimir(t):
    r'Imprimir'
    return t

```

As seguintes funções servem para identificar os comentários, nomeadamente quando se inicia um e quando acaba. Para se fazer um comentário basta colocar `/* ... */`.

```

def t_comentario(t):
    r'/*'
    t.lexer.begin('comentario')

def t_comentario_OFF(t):
    r'*/'
    t.lexer.begin('INITIAL')

def t_comentario_IN(t):
    r'(.|\n)'

```

Na função para o token ***String*** vamos usar a expressão regular abaixo que irá dar match a todos caracteres que estão entre aspas e além disso, iremos sempre trabalhar com strings não vazias, ou seja, strings com pelo menos um caracter.

```

def t_String(t):
    r'"[^"]+"'
    return t

def t_Num(t):
    r'\d+'
    return t

```

Agora para conseguirmos distinguir um número negativo vamos usar o seguinte código no qual usaremos a função `search` para identificar um conjunto de números antecidos pelo sinal `-`.

A função `search` faz parte da biblioteca `re` do python e vai buscar as ocorrências de uma certa expressão regular. Além disso também usamos a função `group` (que faz parte do objeto `search`) para que na procura também se englobe o sinal `-`.

Nota: Além de tudo isto, de forma a conseguirmos distinguir um número negativo de uma subtracção, foi necessário inserir os parênteses à volta do número negativo.

```
def t_Num_neg(t):
    r'\(-\d+\)'
    res = re.search(r'-\d+', t.value)
    t.value = res.group(0)
    return t

def t_ID(t):
    r'\w+'
    return t
```

Na seguinte função vamos criar um conjunto de caracteres que a nossa linguagem vai ignorar sempre que os encontra:

```
t_ignore=' \n\t'
```

Tendo já definido as regras para os tokens, literals e states partimos para a descrição dos erros quando o utilizador utiliza um caracter "proibido":

```
def t_error(t):
    print('Erro léxico, caracter inválido na linha:',t.lexer.lineno)
    t.lexer.skip(1)
```

Denotar que o *t.lexer.lineno* vai buscar a linha do erro e fazemos *t.lexer.skip(1)* de modo a saltar um caracter, ou seja, ignorar o erro e continuar.

Finalmente com as funções léxicas todas definidas anteriormente falta apenas construir o nosso analisador léxico, para isso usaremos a seguinte linha de código:

```
lexer=lex.lex()
```

Capítulo 5

Conversão da GIC para Python

Partindo da GIC descrita no capítulo 3, foi feita uma conversão desta para Python, ou seja, organizou-se as diferentes derivações possíveis da GIC em diferentes funções. Este passo é importante para que, de seguida, para cada uma das seguintes funções, correspondente a uma dada derivação, se possa definir a tradução do código em *Suricata* para código máquina.

A função seguinte diz respeito à derivação do axioma *programa*.

```
def p_programa(p):  
    "programa : declaracoes statements"
```

Seguem-se as funções referentes às duas possíveis derivações do símbolo não terminal *declaracoes* e às duas possíveis derivações do símbolo não terminal *declaracao*.

```
def p_declaracoes_vazio(p):  
    "declaracoes : "  
  
def p_declaracoes(p):  
    "declaracoes : declaracoes declaracao"  
  
def p_declaracao_declVar(p):  
    "declaracao : declVar"  
  
def p_declaracao_declArray(p):  
    "declaracao : declArray"
```

Vêm, de seguida, as funções relativas às derivações dos símbolos não terminais *declVar* e *declArray*, bem como as funções relativas às duas possíveis derivações do símbolo não terminal *listIds*, às duas possíveis derivações do símbolo não terminal *listIdArray* e às duas possíveis derivações do símbolo não terminal *array*.

```
def p_declVar(p):  
    "declVar : VarInt listIds '|' "  
  
def p_listIds_ID(p):  
    "listIds : ID"
```

```

def p_listIds_ID_listIds(p):
    "listIds : listIds ',' ID"

def p_declArray(p):
    "declArray : ArrayInt listIdArray '|' "

def p_listIdArray_list(p):
    "listIdArray : array"

def p_listIdArray_list_listIdArray(p):
    "listIdArray : listIdArray ',' array"

def p_list_ID1(p):
    "array : ID '[' Num ']' "

def p_list_ID2(p):
    "array : ID '[' Num ']' '[' Num ']' "

```

Seguem-se a função relativa à derivação do símbolo não terminal *statements*, bem como as funções relativas às duas possíveis derivações do símbolo não terminal *instrucoes* e às quatro possíveis derivações do símbolo não terminal *instrucao*.

```

def p_statements(p):
    "statements : Inicio ':' instrucoes Fim"

def p_instrucoes_instrucao(p):
    "instrucoes : instrucao"

def p_instrucoes_instrucao_instrucoes(p):
    "instrucoes : instrucoes instrucao"

def p_instrucao_atribuicao(p):
    "instrucao : atribuicao"

def p_instrucao_se(p):
    "instrucao : se"

def p_instrucao_imprimir(p):
    "instrucao : imprimir"

def p_instrucao_ciclo(p):
    "instrucao : ciclo"

```

Vêm, de seguida, as funções que dizem respeito às múltiplas possíveis derivações do símbolo não terminal *atribuicao*.

```

def p_atribuicao_ID_expressao(p):

```

```

    "atribuicao : ID '=' expressao '|' "

def p_atribuicao_ID_ler(p):
    "atribuicao : ID '=' Ler '(' )' '|' "

def p_atribuicao_ID_expressao1D(p):
    "atribuicao : ID '[' expressao ']' '=' expressao '|' "

def p_atribuicao_ID_ler1D(p):
    "atribuicao : ID '[' expressao ']' '=' Ler '(' )' '|' "

def p_atribuicao_ID_expressao2D(p):
    "atribuicao : ID '[' expressao ']' '[' expressao ']' '=' expressao '|' "

def p_atribuicao_ID_ler2D(p):
    "atribuicao : ID '[' expressao ']' '[' expressao ']' '=' Ler '(' )' '|' "

```

Seguem-se as funções relativas às duas possíveis derivações do símbolo não terminal *se* e às duas possíveis derivações do símbolo não terminal *conteudoSeRep*.

```

def p_se_Se(p):
    "se : Se '(' condicoes ')' '{' conteudoSeRep '}' "

def p_se_Se_Senao(p):
    "se : Se '(' condicoes ')' '{' conteudoSeRep '}' Senao '{' conteudoSeRep '}' "

def p_conteudoSeRep_vazio(p):
    "conteudoSeRep : "

def p_conteudoSeRep_instrucoes(p):
    "conteudoSeRep : instrucoes"

```

A seguinte função é relativa à derivação do símbolo não terminal *ciclo*.

```

def p_ciclo(p):
    "ciclo : Repete '{' conteudoSeRep '}' Ate '(' condicoes ')' "

```

Vêm, de seguida, as funções relativas às duas possíveis derivações do símbolo não terminal *imprimir*.

```

def p_imprimir_expressao(p):
    "imprimir : Imprimir '(' expressao ')' '|' "

def p_imprimir_string(p):
    "imprimir : Imprimir '(' String ')' '|' "

```

Seguem-se as funções relativas às três possíveis derivações do símbolo não terminal *elemento*.

```

def p_elemento_ID(p):

```

```

"elemento : ID"

def p_elemento_ID_expressao1D(p):
    "elemento : ID '[' expressao ']' "

def p_elemento_ID_expressao2D(p):
    "elemento : ID '[' expressao ']' '[' expressao ']' "

```

As funções seguintes são relativas às múltiplas derivações possíveis do símbolo não terminal *expressao*.

```

def p_expressao_mais(p):
    "expressao : expressao '+' termo"

def p_expressao_menos(p):
    "expressao : expressao '-' termo"

def p_expressao_termo(p):
    "expressao : termo"

def p_expressao_and(p):
    "expressao : expressao And termo"

def p_expressao_or(p):
    "expressao : expressao Or termo"

def p_expressao_igualigual(p):
    "expressao : expressao Igualigual termo"

def p_expressao_dif(p):
    "expressao : expressao Diferente termo"

def p_expressao_Maiorouigual(p):
    "expressao : expressao Maiorouigual termo"

def p_expressao_Menorouigual(p):
    "expressao : expressao Menorouigual termo"

def p_expressao_Not(p):
    "expressao : Not '(' expressao ')'"

def p_expressao_Menor_expressao(p):
    "expressao : expressao '<' termo"

def p_expressao_Maior_expressao(p):
    "expressao : expressao '>' termo"

```

Seguem-se as funções relativas às quatro derivações possíveis do símbolo não terminal *termo*.

```

def p_termo_vezes(p):

```

```

    "termo : termo '*' fator"

def p_termo_dividir(p):
    "termo : termo '/' fator"

def p_termo_divisao_inteira(p):
    "termo : termo '%' fator"

def p_termo_fator(p):
    "termo : fator"

```

Vêm, seguidamente, as funções relativas às três derivações possíveis do símbolo não terminal *fator*.

```

def p_fator_numposneg(p):
    "fator : num_pos_neg"

def p_fator_elemento(p):
    "fator : elemento"

def p_fator_expressao(p):
    "fator : '(' expressao ')'"

```

Seguem-se as funções relativas às duas derivações possíveis do símbolo não terminal *num_pos_neg*.

```

def p_numposneg_Num(p):
    "num_pos_neg : Num"

def p_numposneg_Num_neg(p):
    "num_pos_neg : Num_neg"

```

Por último, a função que diz respeito à derivação do símbolo não terminal *condicoes* é a seguinte.

```

def p_condicoes_expressao(p):
    "condicoes : expressao"

```

Capítulo 6

Compilador Yacc

Neste capítulo, irá ser descrito o modo como se definiu a tradução do código escrito na linguagem fonte para código Assembly. Essa tradução foi definida a partir de cada uma das funções apresentadas no Capítulo 5, e irá ser explicada para as diferentes componentes existentes na linguagem *Suricata*.

6.1 Declarações:

Começamos por realizar a parte das declarações:

```
def p_programa(p):
    "programa : declaracoes statements"
    p[0] = p[1] + p[2]
    parser.guardar = p[0]
    if(parser.success):
        if (len(parser.avisos1D_variaveis)):
            for key in parser.avisos1D_variaveis:
                print("AVISO! O array " + parser.avisos1D_variaveis[key][0] +
                    "só admite índices entre 0 e "+str(parser.avisos1D_variaveis[key][1]))+"!\n")
        if (len(parser.avisos2D_variaveis)):
            for key in parser.avisos2D_variaveis:
                print("AVISO! O array " + parser.avisos2D_variaveis[key][0] +
                    "admite índices:\nLinhas entre 0 e "+str(parser.avisos2D_variaveis[key][1][1]))+
                    "\nColunas entre 0 e "+str(parser.avisos2D_variaveis[key][2][1]))+"\n")
        print("Código máquina:\n"+ parser.guardar)
    else:

        print("Código máquina:\n" + parser.guardar)
        print("\n\nExiste erro/os, o código máquina não será gerado para um ficheiro .vm!
        \nConsulte o erro no código acima.\n")
```

A função acima é responsável pela produção inicial, ou seja, é a raiz de toda a nossa gramática.

Como podemos verificar, vamos guardando o código **Assembly** na variável *parser.guardar*, para futuramente, se não houver erros, gerar um ficheiro com código **Assembly** e corrê-lo na VM. Além disso, caso o código gerado em *Suricata* contenha arrays, é imprimida uma mensagem de aviso para o utilizador a

alertar acerca dos limites permitidos para indexar, quer para o caso do array de 1 dimensão quer para o de 2 dimensões (por isso temos 2 if no pedaço de código anterior).

Caso ocorra erro no parsing continuaremos a imprimir o código máquina (para o utilizador ter mais um instrumento para detetar erros e para poder corrigi-los) porém no final não será gerado um ficheiro .vm. Caso não haja erros, é gerado um ficheiro .vm com o código máquina correspondente ao programa gerado em *Suricata* pelo utilizador.

Continuamos o resto das declarações, as que vamos apresentar a seguir vão derivar em outras declarações ou em certos casos em apenas numa ou no vazio.

De notar que quando o símbolo *declaracoes* deriva em vazio, a instrução em código máquina correspondente será também uma string vazia.

```
def p_declaracoes_vazio(p):
    "declaracoes : "
    p[0] = ""

def p_declaracoes(p):
    "declaracoes : declaracoes declaracao"
    p[0]=p[1]+p[2]

def p_declaracao_declVar(p):
    "declaracao : declVar"
    p[0] = p[1]

def p_declaracao_declArray(p):
    "declaracao : declArray"
    p[0] = p[1]

def p_declVar(p):
    "declVar : VarInt listIds '|' '"
    p[0] = p[2]
```

Vamos agora verificar se as variáveis estão declaradas consultando desta forma um dicionário criado para o efeito onde sempre que uma variável é criada, esta lhe é adicionada.

Se já foi declarada é imprimida uma mensagem de erro, coloca-se o *parser.success=False* para a função da produção inicial saber que o código contém erros (isto acontece em qualquer erro) e gera-se código máquina correspondente à presença de um erro (através do *err ...* e do *stop*), caso contrário colocámo-la na pilha com a instrução *pushi 0* e aumentamos o *stack pointer*.

```
def p_listIds_ID(p):
    "listIds : ID"
    if (check_variable(p[1], parser.variaveis)):
        p[0] = ("err\"Variável repetida!\"\n") + "stop\n"
        print("A variável \""+p[1]+"\" já foi declarada.\n")
```

```

        parser.success = False
    else:
        parser.variaveis[p[1]]= [parser.sp,1,"VarInt"]
        p[0]=("pushi 0\n")
        parser.sp+=1

```

Podemos também declarar várias variáveis ao mesmo tempo sendo que o procedimento será parecido, porém no fim do if-else temos ainda de "derivar" para o outro ramo e fazer o mesmo até ficarmos sem variáveis por declarar.

```

def p_listIds_ID_listIds(p):
    "listIds : listIds ',' ID"
    if (check_variable(p[3], parser.variaveis)):
        x = ("err\"Variável repetida!\n\n")+ "stop\n"
        print("A variável \"" + p[3] + "\" já foi declarada.\n")
        parser.success = False
    else:
        parser.variaveis[p[3]]= [parser.sp,1,"VarInt"]
        x = ("pushi 0\n")
        parser.sp += 1
    p[0] = p[1] + x

```

Acabada a parte de declarar as variáveis partimos para a declaração dos arrays:

```

def p_declArray(p):
    "declArray : ArrayInt listIdArray '|' "
    p[0] = p[2]

```

```

def p_listIdArray_list_listIdArray(p):
    "listIdArray : listIdArray ',' array"
    p[0] = p[1] + p[3]

```

```

def p_listIdArray_list(p):
    "listIdArray : array"
    p[0] = p[1]

```

No caso de um array de 1 dimensão temos de novamente ver se o nome do array está declarado, caso esteja imprime-se uma mensagem de erro a alertar o utilizador, faz-se ***parser.success=False*** e coloca-se as instruções ***err...*** e ***stop*** na pilha, senão temos de adicionar a variável que define o array ao nosso dicionário de variáveis, gerar código máquina de forma a alocar na stack espaço para o array (usando o ***pushn***, que empilha n zeros na pilha) e de seguida incrementamos o ***stack pointer*** n vezes, com n a corresponder ao tamanho do array.

```

def p_list_ID1(p):
    "array : ID '[' Num ']' "
    if (check_variable(p[1], parser.variaveis)):

```

```

    p[0] = ("err\"Variável repetida!\n")+ "stop\n"
    print("A variável \"" + p[1] + "\" já foi declarada.\n")
    parser.success = False
else:
    parser.variaveis[p[1]]=int(p[3]),parser.sp,"ArrayInt"
    p[0] = (f"pushn {int(p[3])}\n")
    parser.sp += int(p[3])

```

Para o array de 2 dimensões vai ser a mesma coisa em termos de verificar se já está declarado ou não e como proceder caso já esteja declarado. Porém se não tiver, o espaço que vamos reservar na pilha passa a ser a multiplicação do número de linhas pelo número de colunas. Além disso este vai ser o valor que vamos incrementar ao *stack pointer*.

```

def p_list_ID2(p):
    "array : ID '[' Num ']' '[' Num ']' "
    if (check_variable(p[1], parser.variaveis)):
        p[0] = ("err\"Variável repetida!\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" já foi declarada.\n")
        parser.success = False
    else:
        dim_bi = int(p[3]) * int(p[6]) #dim_bi = numero de elementos
        parser.variaveis[p[1]]=((dim_bi,int(p[3]),int(p[6])),parser.sp,"DoubleArrayInt")
        p[0] = (f'pushn {dim_bi}\n')
        parser.sp += dim_bi

```

6.2 Funções auxiliares:

Uma vez já criado o dicionário "parser.variaveis", foram criadas 6 funções auxiliares com o objetivo de tornar o código mais legível e facilitar a sua escrita.

A função abaixo recebe como argumento uma variável e um dicionário de variáveis, que neste trabalho é o "parser.variaveis", retornando 1 se a variável estiver no dicionário ou 0 se não estiver.

Esta função será útil futuramente para verificar se a variável já foi declarada ou não.

```

def check_variable(variavel,variaveis_dic):
    if variavel in variaveis_dic:
        return 1
    else:
        return 0

```

As próximas duas funções têm como argumentos uma variável e um dicionário de variáveis, sendo que a primeira retorna a posição na stack de uma variável do tipo **VarInt** e a segunda retorna a posição de uma variável do tipo **ArrayInt**.

```

def return_sp_VarInt(variavel,variaveis_dic):
    sp=variaveis_dic[variavel][0]
    return sp

```

```
def return_sp_ArrayInt(variavel,variaveis_dic):
    sp=variaveis_dic[variavel][1]
    return sp
```

As próximas duas funções recebem como argumento o mesmo que as anteriores, sendo que a primeira retorna uma lista com a dimensão de uma variável do tipo ***ArrayInt*** uni-dimensional e a segunda retorna uma lista com a dimensão, número de linhas e número de colunas de uma variável do tipo ***ArrayInt*** bidimensional.

```
def return_dimensions_ArraInt1D(variavel,variaveis_dic):
    dim=variaveis_dic[variavel][0]
    return [dim]
```

```
def return_dimensions_ArraInt2D(variavel,variaveis_dic):
    dim=variaveis_dic[variavel][0][0]
    n_linhas=variaveis_dic[variavel][0][1]
    n_colunas=variaveis_dic[variavel][0][2]
    return [dim,n_linhas,n_colunas]
```

A próxima função auxiliar recebe os mesmos argumentos que as anteriores, retornando o tipo da variável em questão.

```
def return_tipo(variavel,variaveis_dic):
    tipo=variaveis_dic[variavel][2]
    return tipo
```

6.3 Instruções:

Depois de já estarem realizadas as declarações, iremos entrar na parte das instruções. Usamos as operações ***start*** e ***stop*** da VM de forma a dar início e fim às instruções. O símbolo não terminal ***instrucoes*** pode derivar no símbolo não terminal ***instrucao***, sendo que este último pode, por exemplo, ser apenas uma atribuição, ou nos símbolos não terminais ***instrucoes instrucao***, uma vez que podemos ter, por exemplo, mais do que uma atribuição. Além das atribuições, a instrução pode derivar também num ***ciclo(Repete-Ate)***, numa ***instrução condicional(Se ou Se-Senao)*** ou até ***imprimir*** algo, como por exemplo, uma string ou uma variável.

```
def p_statements(p):
    "statements : Inicio ':' instrucoes Fim"
    p[0] = "start\n" + p[3] + "stop\n"
```

```
def p_instrucoes_instrucao(p):
    "instrucoes : instrucao"
    p[0] = p[1]
```

```
def p_instrucoes_instrucao_instrucoes(p):
    "instrucoes : instrucoes instrucao"
    p[0] = p[1] + p[2]
```

```
def p_instrucao_atribuicao(p):
    "instrucao : atribuicao"
    p[0] = p[1]
```

```
def p_instrucao_se(p):
    "instrucao : se"
    p[0] = p[1]
```

```
def p_instrucao_imprimir(p):
    "instrucao : imprimir"
    p[0] = p[1]
```

```
def p_instrucao_ciclo(p):
    "instrucao : ciclo"
    p[0] = p[1]
```

6.3.1 Atribuições:

Tal como referido anteriormente, o símbolo não terminal **instrucao** pode derivar num outro símbolo não terminal **atribuicao**. Dentro deste, vamos ter várias possibilidades de atribuição:

- atribuição de uma expressão a uma variável do tipo **VarInt**;
- atribuição a uma variável do tipo **VarInt** a partir da leitura do teclado (input);
- atribuição de uma expressão a uma posição de um **array** 1D, 2D;
- atribuição a uma posição de um **array** 1D, 2D a partir da leitura do teclado.

Atribuição a uma variável do tipo VarInt:

Há dois possíveis erros que podem acontecer:

- a variável não está declarada
- o utilizador fazer uma atribuição a um array, sendo que se esqueceu dos índices (neste caso, embora a atribuição seja feita a uma variável do tipo **ArrayInt**, esta atribuição toma a forma de uma atribuição feita corretamente a uma variável do tipo **VarInt**).

Tal como referido, um dos erros corresponde à não declaração de uma variável. Se a variável não é declarada, emitimos de imediato um aviso a dizer que a variável não está declarada, indicamos a variável em questão e ainda colocamos **parser.success=False** porque estamos na presença de um erro.

O outro erro possível é o utilizador atribuir um valor a um array sem indicar o/os índice/es. Por exemplo, consideremos a seguinte declaração: *ArrayInt a[2][2]*. O erro em questão corresponde à possibilidade de o utilizador fazer *a=3*.

Sempre que um erro acontece, é gerado código máquina que sinaliza a presença de um erro. Essa sinalização é realizada com base na instrução "err ... + stop", de forma a imprimir o aviso do erro em código máquina. Caso a variável seja declarada e seja um VarInt, então não há problema e podemos prosseguir. A partir da função auxiliar, é-nos retornado a posição onde a variável está na stack. Seguidamente, a partir da utilização do *storeg*, vamos ao topo do stack, pegamos no valor que se encontra lá e vamos colocá-lo em gp[posição]. Por exemplo, consideremos que "p[3]=5" (o 5 estará no topo da stack) e que a posição da variável "x" é 1. O que vai acontecer é que vamos colocar o valor 5 em gp[1], ou seja, equivale a *x=5*.

```
def p_atribuicao_ID_expressao(p):
    "atribuicao : ID '=' expressao '|' '"
    # ver se a variavel está declarada. Se nao estiver enviamos um erro

    # Se estiver declarada entao não há erro
    if (check_variable(p[1],parser.variaveis)):
        if (return_tipo(p[1],parser.variaveis)=="VarInt"):
            stackpointer=return_sp_VarInt(p[1],parser.variaveis)
            p[0] = p[3] + f"storeg {stackpointer}\n"

        else:
            p[0] = ("err\"Indexação em falta!\n")+ "stop\n"
            print("A variável \"" + p[1] + "\" precisa de indexação.\n")
            parser.success = False

    else:
        # a variavel nao esta declarada
        p[0] = ("err\"Variável não existe!\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" não foi declarada.\n")
        parser.success = False
```

Para o caso de uma leitura, a estrutura é quase igual, apenas temos de inserir agora o *read* e o *atoi*. O *read* irá colocar o endereço da string inserida pelo teclado na topo da stack. Seguidamente o *atoi* irá "pegar" nesse endereço e converte a string num inteiro. Após a conversão, empilha o inteiro na stack.

```
def p_atribuicao_ID_ler(p):
    "atribuicao : ID '=' Ler '(' ')' '|' '"

    # ver se a variavel está declarada. Se nao estiver enviamos um erro
    # Se estiver declarada entao não há erro
    if (check_variable(p[1],parser.variaveis)):
        if(return_tipo(p[1],parser.variaveis)=="VarInt"):
            stackpointer = return_sp_VarInt(p[1], parser.variaveis)
```

```

        p[0] = f"read\natoi\n" + f"storeg {stackpointer}\n"
    else:
        p[0] = ("err\"Indexação em falta!\n\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" precisa de indexação.\n")
        parser.success = False
    else:
        # a variavel nao esta declarada
        p[0] = ("err\"Variável não existe!\n\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" não foi declarada.\n")
        parser.success = False

```

Atribuição a um array 1D:

Outra possibilidade é realizar uma atribuição sobre um **array de uma dimensão**.

Mais uma vez, verificamos se a variável está declarada. Se estiver declarada, ainda temos de verificar se é do tipo `VarInt` porque o utilizador podia ter inserido uma variável deste tipo com índices. O problema em questão, está no caso do utilizador inserir por exemplo: `a[1]=3`, na qual a variável "a" não é um array. Nesse caso, teremos de emitir um erro.

Por sua vez, caso esteja tudo "ok", podemos prosseguir. Uma vez, estarmos a tratar de um array, primeiro necessitamos de saber onde se encontra a primeira posição do array. Essa posição é-nos dada a partir da variável "*stack pointer*" que se encontra nas funções abaixo. Desse modo, foi necessário utilizar a função auxiliar *return_sp_ArrayInt* que nos retorna a posição do array correspondente na stack.

Primeiro é necessário colocar o *pushgp*. Este irá empilhar o valor do registo *gp*. Seguidamente realizamos o *pushi stack pointer*, ou seja, empilhamos o inteiro que representa a distância entre o global pointer e a primeira posição do array. Uma vez que o *padd* retira da pilha um inteiro *a*, um endereço *x* e empilha o endereço *x+a*, então vamos empilhar o endereço do primeiro elemento do array. Após sabermos "o local do array na stack" falta-nos saber o valor que queremos atribuir e ainda o índice do array onde pretendemos inserir o valor. Depois de estar na stack, o valor(`p[6]`), o índice(`p[3]`) e o endereço do primeiro elemento do array, podemos utilizar o *storen* uma vez que ele vai pegar no valor e vai colocá-lo em *endereço do primeiro elemento do array[índice]*.

```

def p_atribuicao_ID_expressao1D(p):
    "atribuicao : ID '[' expressao ']' '=' expressao '|'"
    if (check_variable(p[1],parser.variaveis)):
        if (return_tipo(p[1],parser.variaveis)=="ArrayInt"):
            dim=return_dimensions_ArraInt1D(p[1],parser.variaveis)
            parser.avisos1D_variaveis[p[1]]=p[1],dim[0]-1
            stackpointer=return_sp_ArrayInt(p[1],parser.variaveis)
            p[0]= f"pushgp\npushi {stackpointer}\npadd\n" + p[3] + p[6] + "storen\n"
        else:
            p[0] = ("err\"A variável em questão não admite indexação,
                    ou é um array de dimensão diferente!\n\n")+ "stop\n"

            print("A variável \"" + p[1] + "\" não admite indexação,
                    ou é um array de dimensão diferente.\n")

            parser.success = False

```

```

else:
    # a variavel nao esta declarada
    p[0] = ("err\"Variável não declarada!\n")+ "stop\n"
    print("A variável \"" + p[1] + "\" não está declarada.\n")
    parser.success = False

```

Como visto anteriormente, sempre que o utilizador aceder a uma posição de um array, será emitido um aviso para o mesmo ter o cuidado de estar a aceder a uma posição válida do array. Nesse aviso, será referido nome do array que "se está a utilizar" e ainda as posições válidas para aceder ao mesmo. Desse modo, todos os arrays 1D utilizados são adicionados ao dicionário *parser.avisos1D_variaveis*. Ao serem adicionados, tanto o "nome" como a dimensão são especificados.

Uma vez que já visualizamos como funciona o *read* e o *atoi*, o código relativo a uma atribuição de um array 1D a partir da leitura, será o seguinte:

```

def p_atribuicao_ID_1D(p):
    "atribuicao : ID '[' expressao ']' '=' Ler '(' ')' '|' '"
    if (check_variable(p[1], parser.variaveis)):
        if (return_tipo(p[1], parser.variaveis) == "ArrayInt"):
            dim = return_dimensions_ArrayInt1D(p[1], parser.variaveis)
            parser.avisos1D_variaveis[p[1]] = [p[1], dim[0] - 1]
            stackpointer = return_sp_ArrayInt(p[1], parser.variaveis)
            p[0] = f"pushgp\npushi {stackpointer}\npadd\n" + p[3] + f"read\natoi\n" + f"store\n"
        else:
            p[0] = ("err\"A variável em questão não admite indexação,
                    ou é um array de dimensão diferente!\n")+ "stop\n"

            print("A variável \"" + p[1] + "\" não admite indexação,
                    ou é um array de dimensão diferente.\n")

            parser.success = False

    else:
        # a variavel nao esta declarada
        p[0] = ("err\"Variável não declarada!\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" não está declarada.\n")
        parser.success = False

```

Atribuição a um array 2D:

Para o array de duas dimensões esta atribuição torna-se mais complexa uma vez que vamos ter dois índices. Inicialmente visualizamos se já foi declarado e seguidamente verificamos se a variável em questão é um array de duas dimensões, ou seja, **DoubleArrayInt**.

A parte mais difícil foca-se em conseguir determinar o "local correto da stack" onde queremos colocar o valor. Uma vez que sabemos a posição do primeiro elemento do array, então apenas necessitamos de saber "quantos blocos temos de subir" para chegar ao índice correto. Para determinar quanto temos de subir relativamente à primeira posição do array, faz-se o seguinte raciocínio:

- multiplicamos a linha que estamos a tentar aceder pelo número de colunas total do array

- o resultado do ponto anterior, é somado à coluna que estamos a tentar aceder

Após realizar os cálculos anteriores já temos o resultado pretendido.

A primeira parte é exactamente igual ao array de 1D visto que pretendemos obter a primeira posição do array, ou seja, *pushgp\ n pushi stackpointer\ n padd\ n*.

No entanto, na parte seguinte temos de fazer *p[3] + f"pushi colunas\ n" + "mul\ n"* de forma a obedecer ao primeiro tópico e o resultado anterior é empilhado na stack. Uma vez que vamos querer somar esse valor à coluna que estamos a tentar aceder, será necessário colocar *resultadoAnterior + p[6] + "add\ n"* e o resultado final é empilhado. Desta forma *p[3] + f"pushi colunas\ n" + "mul\ n" + p[6] + "add\ n"* indicar-nos-á quantos "blocos" terei de subir, a partir da primeira posição do array, para chegar ao índice do array pretendido (por exemplo: *array[1][0]*). Por fim, pegamos no valor (*p[9]*), no número de blocos a subir (*n*) e o endereço do primeiro elemento do array(*x*) e vamos inserir o valor no endereço *x[n]*

```
def p_atribuicao_ID_expressao2D(p):
    "atribuicao : ID '[' expressao ']' '[' expressao ']' '=' expressao '|'"
    if (check_variable(p[1],parser.variaveis)):
        if(return_tipo(p[1],parser.variaveis)=="DoubleArrayInt"):
            stackpointer=return_sp_ArrayInt(p[1],parser.variaveis)
            dimensoes=return_dimensions_ArraInt2D(p[1],parser.variaveis)
            colunas=dimensoes[2]
            linhas=dimensoes[1]
            parser.aviso2D_variaveis[p[1]]=[p[1],(0,linhas-1),(0,colunas-1)]
            p[0]= f"pushgp\npushi {stackpointer}\npadd\n" + p[3] + f"pushi {colunas}\n" +
                "mul\n" + p[6] + "add\n" + p[9] + "store\n"
        else:
            p[0] = ("err\ "A variável em questão não admite indexação,
                ou é um array de dimensão diferente!\n\n")+ "stop\n"

            print("A variável \"" + p[1] + "\" não admite indexação,
                ou é um array de dimensão diferente.\n")

            parser.success = False

    else:
        # a variavel nao esta declarada
        p[0] = ("err\ "Variável não declarada!\n\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" não está declarada.\n")
        parser.success = False
```

Tal como no array 1D, sempre que estamos a aceder a uma posição de um array 2D é emitido um aviso para o utilizador ter o cuidado de estar a aceder a uma posição válida do array. Nesse aviso é referida o nome da variável do array 2D, as linhas e as colunas válidas. Nesse sentido, todos os arrays 2D utilizados são adicionados ao dicionário *parser.aviso2D_variaveis* e ao serem adicionados, tanto o nome, tanto o número de linhas como o número de colunas são especificados.

Por sua vez, no caso da leitura o nosso valor (no anterior era o *p[9]*) vai ser substituído pelo o uso adequado do *read* e *atoi*, ou seja,

```

def p_atribuicao_ID_ler2D(p):
    "atribuicao : ID '[' expressao ']' '[' expressao ']' '=' Ler '(' ')' ' '|'"
    if (check_variable(p[1],parser.variaveis)):
        if(return_tipo(p[1],parser.variaveis)=="DoubleArrayInt"):
            parser.avisos2D=1
            stackpointer = return_sp_ArrayInt(p[1], parser.variaveis)
            dimensoes = return_dimensions_ArraInt2D(p[1], parser.variaveis)
            colunas = dimensoes[2]
            linhas = dimensoes[1]
            parser.avisos2D_variaveis[p[1]] = [p[1], (0, linhas - 1), (0, colunas - 1)]
            p[0]= f"pushgp\npushi {stackpointer}\npadd\n" + p[3] + f"pushi {colunas}\n"+
                "mul\n" + p[6]+ "add\n" + f"read\natoi\n" + "storen\n"
        else:
            p[0] = ("err\A variável em questão não admite indexação,
                ou é um array de dimensão diferente!\n\n")+ "stop\n"

            print("A variável \"" + p[1] + "\" não admite indexação,
                ou é um array de dimensão diferente.\n")

            parser.success = False
    else:
        # a variavel nao esta declarada
        p[0] = ("err\Variável não declarada!\n\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" não está declarada.\n")
        parser.success = False

```

6.3.2 Estruturas condicionais:

- **Se(condição) { instruções }**

Relativamente a esta estrutura condicional, para definir a sua tradução para código **Assembly**, utilizou-se a instrução de controlo **jz**. Esta instrução retira um valor da pilha e, caso este seja nulo, é executada a instrução onde se encontra a label e, caso contrário, é executada a instrução seguinte. A cada label utilizada como argumento desta instrução no contexto desta estrutura condicional deu-se o nome "fimSen", em que *n* é um inteiro cujo valor é igual ao do contador **parser.fimSeCount**. O motivo pelo qual se acrescenta este *n* no final desta label é evitar que existam várias labels com nomes iguais, mas correspondentes a diferentes estruturas condicionais. A label "fimSen" está presente na instrução que sucede o código máquina correspondente às instruções desta estrutura condicional, caso estas existam.

Tal como se pode observar no pedaço de código seguinte, através da instrução **jz**, caso o valor colocado na stack correspondente ao valor da condição desta estrutura seja diferente de 0, isto é, a condição seja verdadeira, são executadas as instruções da estrutura condicional (caso existam). Caso o valor colocado na stack correspondente ao valor da condição seja igual a 0, isto é, a condição seja falsa, é executada a instrução onde se encontra a label "fimSen", ou seja, as instruções da estrutura condicional não são realizadas.

```

def p_se_Se(p):
    "se : Se '(' condicoes ')' '{' conteudoSeRep '}'"
    p[0] = p[3] + f"jz fimSe{parser.fimSeCount}\n" + p[6] + f"fimSe{parser.fimSeCount}: nop\n"
    parser.fimSeCount +=1

```

- **Se(condição) { instruçõesSe } Senão { instruçõesSenão }**

Aqui, *instruçõesSe* correspondem às instruções executadas se a condição for verdadeira, e *instruçõesSenão* às que se executam se a condição for falsa. Relativamente a esta estrutura condicional, para definir a sua tradução para código **Assembly**, utiliza-se novamente a instrução *jz*, como também a instrução de controlo *jump*. Esta última instrução faz com que seja executada a instrução onde se encontra a label correspondente. Nesta estrutura, são utilizadas duas labels: a label "fimSen", em que *n* é um inteiro cujo valor é igual ao do contador *parser.fimSeCount*, e a label "senaox", sendo *x* um inteiro cujo valor é igual ao do contador *parser.senaoCount*. O motivo pelo qual se acrescenta este *x* no final desta label é o mesmo pelo qual se acrescenta o *n* no final da label "fimSen", e que foi explicado anteriormente. A label "fimSen" está presente na instrução que sucede o código máquina correspondente às *instruçõesSenão*, caso estas existam. Por outro lado, a label "senaox" está presente na instrução que antecede esse código máquina.

Tal como se pode observar no pedaço de código seguinte, através da instrução "jz senaox", caso o valor colocado na stack correspondente ao valor da condição desta estrutura seja diferente de 0, isto é, a condição seja verdadeira, são executadas as *instruçõesSe* (caso existam). De seguida, através da instrução "jump fimSen", é executada a instrução onde se encontra a label "fimSen", e as *instruçõesSenão* não são executadas.

Caso o valor colocado na stack correspondente ao valor da condição seja igual a 0, isto é, a condição seja falsa, é executada a instrução onde se encontra a label "senaox", ou seja, as *instruçõesSe* não são realizadas, sendo executadas as *instruçõesSenão* em vez dessas.

```
def p_se_Se_Senao(p):
    "se : Se '(' condicoes ')' '{' conteudoSeRep '}' Senao '{' conteudoSeRep '}'"
    p[0] = p[3] + f"jz senao{parser.senaoCount}\n" + p[6] + f"jump fimSe{parser.fimSeCount}\n"
        + f"senao{parser.senaoCount}: nop\n" + p[10] + f"fimSe{parser.fimSeCount}: nop\n"
    parser.senaoCount +=1
    parser.fimSeCount +=1
```

6.3.3 Instruções das estruturas condicionais e cíclicas:

Tal como se observa no seguinte pedaço de código, quando o símbolo *conteudoSeRep* deriva em vazio, a instrução em código máquina correspondente será também uma string vazia, e caso este símbolo derive em instruções, as instruções em código máquina correspondentes serão as que correspondem a essas instruções.

```
def p_conteudoSeRep_vazio(p):
    "conteudoSeRep : "
    p[0]=""
```

```
def p_conteudoSeRep_instrucoes(p):
    "conteudoSeRep : instrucoes"
    p[0]=p[1]
```

6.3.4 Ciclo:

Relativamente ao ciclo repeat-until, para definir a sua tradução para código **Assembly**, utilizou-se novamente a instrução *jz*. À label utilizada como argumento desta instrução no contexto de um ciclo deu-se

o nome "ciclon", em que n é um inteiro cujo valor é igual ao do contador *parser.cicloCount*. O motivo pelo qual se acrescenta este n no final desta label é evitar que existam várias labels com nomes iguais, mas correspondentes a diferentes ciclos. A label "ciclon" está presente na instrução que antecede o código máquina correspondente às instruções do ciclo, caso existam.

Tal como se observa no seguinte pedaço de código, as instruções do ciclo (caso existam) são executadas pelo menos uma vez. Após serem executadas, o valor da condição do ciclo irá ser colocado na stack. Caso este valor seja diferente de 0 (se a condição for verdadeira), a instrução "jz ciclon" faz com que seja executada a instrução seguinte, e o ciclo termina. Caso o valor da condição seja igual a 0 (se a condição for falsa), a instrução "jz ciclon" faz com que seja executada a instrução onde se encontra a label "ciclon", pelo que as instruções do ciclo tornam a ser executadas, e mais tarde a instrução "jz ciclon" volta a ser executada. Isto dá origem à dinâmica correspondente ao ciclo repeat-until.

```
def p_ciclo(p):
    "ciclo : Repete '{' conteudoSeRep '}' Ate '(' condicoes ')'"
    p[0] = f"ciclo{parser.cicloCount}: nop\n" + p[3] + p[7] + f"jz ciclo{parser.cicloCount}\n"
    parser.cicloCount +=1
```

6.3.5 Imprimir:

Relativamente à instrução de imprimir o valor de qualquer expressão (que no caso da nossa linguagem é sempre um inteiro), para definir a sua tradução para código **Assembly**, utilizou-se a instrução *writei*. Tal como se observa no seguinte pedaço de código, após ser colocado na stack o inteiro correspondente ao valor da expressão a imprimir, a instrução *writei* retira esse inteiro da stack e imprime o seu valor na saída standard.

```
def p_imprimir_expressao(p):
    "imprimir : Imprimir '(' expressao ')'"
    p[0] = p[3] + "writei\n"
```

Relativamente à instrução de imprimir uma string, para definir a sua tradução para código **Assembly**, utilizou-se as instruções *pushs* e *writes*. Tal como se observa no seguinte pedaço de código, a instrução *pushs* é usada para empilhar o endereço da string que se pretende imprimir, e a instrução *writes* é usada de seguida para retirar esse endereço da stack e imprimir a string correspondente (a string que se pretende imprimir) na saída standard.

```
def p_imprimir_string(p):
    "imprimir : Imprimir '(' String ')'"
    p[0] = f"pushs {p[3]}\nwrites\n"
```

6.3.6 Expressões:

No caso das expressões tem-se como objetivo criar código máquina de forma a realizar as operações aritméticas, relacionais, lógicas, atribuir inteiros, variáveis declaradas ou uma combinação de tudo isto, colocando o valor da expressão no topo da stack.

```
def p_expressao_mais(p):
    "expressao : expressao '+' termo"
```

```

p[0] = p[1] + p[3] + "add\n"

def p_expressao_menos(p):
    "expressao : expressao '-' termo"
    p[0] = p[1] + p[3] + "sub\n"

def p_expressao_termo(p):
    "expressao : termo"
    p[0]=p[1]

def p_expressao_Igualigual(p):
    "expressao : expressao Igualigual termo"
    p[0]=p[1]+p[3]+"equal\n"

def p_expressao_dif(p):
    "expressao : expressao Diferente termo"
    p[0]=p[1]+p[3]+"equal\n"+"not\n"

def p_expressao_Maiorouigual(p):
    "expressao : expressao Maiorouigual termo"
    p[0]=p[1]+p[3]+"supeq\n"

def p_expressao_Menorouigual(p):
    "expressao : expressao Menorouigual termo"
    p[0] = p[1] + p[3] + "ineq\n"

def p_expressao_Not(p):
    "expressao : Not '(' expressao ')'"
    p[0]=p[3]+"not\n"

def p_expressao_Menor_expressao(p):
    "expressao : expressao '<' termo"
    p[0]=p[1]+p[3]+"inf\n"

def p_expressao_Maior_expressao(p):
    "expressao : expressao '>' termo"
    p[0] = p[1] + p[3] + "sup\n"

```

Em cima temos a parte dos códigos capazes de produzir operações aritméticas e relacionais.

Quanto às operações lógicas foi necessário ter um pouco de cuidado, uma vez que o resultado só pode ser 0 ou 1.

Como tal, imaginemos o caso da seguinte expressão: $50 \wedge 30$.

Por definição o 50 e o 30 representam o valor 1 quando o assunto se refere a operações lógicas, e portanto o resultado desta expressão teria de ser igual a 1. Para solucionar este tipo de problemas implementamos as seguintes estratégias:

- **Condição OR:** Coloca-se ambos os valores das expressões no topo da stack e utiliza-se o código *add* que soma os valores e coloca o resultado em cima da stack, de seguida coloca-se de novo os valores das expressões no topo da stack e utiliza-se o código *mul* que multiplica os valores e coloca o resultado em cima da stack. Uma vez tendo o resultado da soma e da multiplicação no topo da stack, basta realizar o código *sub* que subtrai o resultado da soma com o resultado da multiplicação. Caso ambos os valores da expressão sejam diferentes de 0 ou só haja um 0, então esta subtração irá dar um número diferente de 0 e caso ambos sejam zero o resultado da subtração será 0. Uma vez feito isto basta operar duas vezes com o código *not*, pois se o valor da subtração for diferente de zero, o resultado de aplicar duas vezes o *not* será 1 e se o valor da subtração for igual a 0 então o resultado de aplicar duas vezes o *not* será 0. Esta técnica satisfaz na perfeição aquilo que a condição **OR** pretende, ou seja só retorna 0 se ambos forem 0.
- **Condição AND:** Coloca-se ambos os valores das expressões no topo da stack e utiliza-se o código *mul*, operando de seguida com dois *not*. Caso ambos os valores ou um valor da expressão seja 0, o resultado da multiplicação será 0 e aplicando duas vezes com o *not* temos o valor 0 em cima da stack. Se ambos os valores forem diferentes de 0 o resultado de aplicar o *mul* e os dois *not* é um 1 em cima da stack. Esta técnica também satisfaz na perfeição aquilo que a condição **AND** pretende, ou seja só retorna 1 se ambos forem 1.

```
def p_expressao_and(p):  
    "expressao : expressao And termo"  
    p[0] = p[1]+p[3]+"mul\n" + "not\n" + "not\n"
```

```
def p_expressao_or(p):  
    "expressao : expressao Or termo"  
    p[0] = p[1] + p[3] + "add\n" + p[1] + p[3] + "mul\n" + "sub\n" + "not\n" + "not\n"
```

Realça-se o facto da existência do símbolo não terminal *termo*, que tem como objetivo vir a priorizar as operações de multiplicação, divisão inteira e resto da divisão face às outras operações.

```
def p_termo_vezes(p):  
    "termo : termo '*' fator"  
    p[0] = p[1] + p[3] + "mul\n"
```

```
def p_termo_dividir(p):
    "termo : termo '/' fator"
    p[0] = p[1] + p[3] + "div\n"
```

```
def p_termo_divisao_inteira(p):
    "termo : termo '%' fator"
    p[0] = p[1] + p[3] + "mod\n"
```

```
def p_termo_fator(p):
    "termo : fator"
    p[0]=p[1]
```

```
def p_fator_numposneg(p):
    "fator : num_pos_neg"
    p[0]=p[1]
```

```
def p_fator_elemento(p):
    "fator : elemento"
    p[0]=p[1]
```

```
def p_fator_expressao(p):
    "fator : '(' expressao ')'"
    p[0]=p[2]
```

```
def p_numposneg_Num(p):
    "num_pos_neg : Num"
    p[0]= f"pushi {p[1]}\n"
```

```
def p_numposneg_Num_neg(p):
    "num_pos_neg : Num_neg"
    p[0] = f"pushi {p[1]}\n"
```

O símbolo não terminal *elemento* tem como objectivo permitir ao utilizador aplicar variáveis declaradas nas expressões.

Como tal, teve de se ter algum cuidado, uma vez que não se pode invocar variáveis que nunca foram declaradas, nem utilizar variáveis do tipo *VarArray* sem colocar a indexação, como também não se pode colocar uma variável do tipo *VarArray* com estrutura bidimensional e só se colocar um índice ou nenhum, nem mesmo utilizar variáveis do tipo *VarInt* e colocar indexações. No fundo teve de se ter cuidado com o

tipo da variável e se ela já foi previamente declarada no bloco de declarações, como tal utilizou-se a função auxiliar *check_variable(variavel,variaveis_dic)* que verificava se a variável utilizada estava presente no dicionário *parser.variaveis*, ou seja, se ela já tinha sido declarada, caso tenha sido avança-se para a próxima condição, caso contrário, coloca-se a variável *parser.success=False* de forma a não ser gerado um ficheiro .vm com o código **Assembly** referente ao programa criado em *Suricata*. Também é colocado o código *err Variável não declarada* na stack seguida do código *stop* de forma a ser impresso o código **Assembly** no terminal aquando do axioma, para o utilizador poder utilizar isso como mais um instrumento para verificar onde ocorreu o erro e como deve corrigir.

A próxima condição a verificar é se o tipo da variável é concordante com a gramática aplicada sobre ela, tal como explicado há pouco, utilizando a função auxiliar *return_tipo(variável,variaveis_dic)* que retorna o tipo da variável. Se o tipo corresponder então coloca-se o código na stack adequado de forma a colocar no topo da mesma o valor da variável em questão, se não corresponder então coloca-se a variável *parser.success=False* de forma a não ser gerado nenhum ficheiro .vm com o código **Assembly** referente ao programa criado em *Suricata*. Também se coloca o código *err Indexação em falta!* ou *err A variável em questão não admite indexação, ou é um array de dimensão diferente!* na stack , dependendo da situação, de forma ao utilizador ter mais um instrumento para verificar onde ocorreu o erro e como o deve corrigir.

```
def p_elemento_ID(p):
    "elemento : ID"
    if (check_variable(p[1],parser.variaveis)):
        if(return_tipo(p[1],parser.variaveis)=="VarInt"):
            stackpointer=return_sp_VarInt(p[1],parser.variaveis)
            p[0] = f"pushg {stackpointer}\n"
        else:
            p[0] = ("err\"Indexação em falta!\n")+ "stop\n"
            print("A variável \"" + p[1] + "\" precisa de indexação.\n")
            parser.success = False

    else:
        p[0] = ("err\"Variável não declarada!\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" não está declarada.\n")
        parser.success = False

def p_elemento_ID_expressao1D(p):
    "elemento : ID '[' expressao ']' "
    if (check_variable(p[1],parser.variaveis)):
        if(return_tipo(p[1],parser.variaveis)=="ArrayInt"):
            dim = return_dimensions_ArraInt1D(p[1], parser.variaveis)
            parser.avisos1D_variaveis[p[1]] = [p[1], dim[0] - 1]
            stackpointer = return_sp_ArrayInt(p[1], parser.variaveis)
            p[0] = f"pushgp\npushi {stackpointer}\npadd\n"+p[3]+"loadn\n"
```



```

else:
    p[0] = ("err\"A variável em questão não admite indexação,
            ou é um array de dimensão diferente!\n")+ "stop\n"

    print("A variável \"" + p[1] + "\" não admite indexação,
            ou é um array de dimensão diferente.\n")

    parser.success = False

else:
    p[0] = ("err\"Variável não declarada!\n")+ "stop\n"
    print("A variável \"" + p[1] + "\" não está declarada.\n")
    parser.success = False

def p_elemento_ID_expressao2D(p):
    "elemento : ID '[' expressao ']' '[' expressao ']' "
    if (check_variable(p[1],parser.variaveis)):
        if (return_tipo(p[1], parser.variaveis) == "DoubleArrayInt"):
            parser.aviso2D=1
            stackpointer = return_sp_ArrayInt(p[1], parser.variaveis)
            dimensoes = return_dimensions_ArraInt2D(p[1], parser.variaveis)
            colunas = dimensoes[2]
            linhas = dimensoes[1]
            parser.aviso2D_variaveis[p[1]] = [p[1], (0, linhas - 1), (0, colunas - 1)]
            p[0] = f"pushgp\npushi {stackpointer}\npadd\n" + p[3] + f"pushi {colunas}\n" +
                    "mul\n" + p[6] + "add\n" + "loadn\n"

        else:
            p[0] = ("err\"A variável em questão não admite indexação,
                    ou é um array de dimensão diferente!\n")+ "stop\n"

            print("A variável \"" + p[1] + "\" não admite indexação,
                    ou é um array de dimensão diferente.\n")

            parser.success = False

    else:
        p[0] = ("err\"Variável não declarada!\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" não está declarada.\n")
        parser.success = False

```

De notar que tal como em situações anteriores se preencheu o dicionário ***parser.avisos1D_variaveis*** e ***parser.avisos2D_variaveis*** de forma a utilizar essa informação para imprimir um aviso na produção inicial, associada ao axioma inicial, que serve para o utilizador saber que tem de respeitar os limites de indexação referentes aos arrays.

6.3.7 Condições:

Para o caso das condições, optamos por fazer uma linguagem que pode ter como condição qualquer expressão que se derivou na secção "expressões", podendo assim ter como condição uma expressão aritmética, relacional, lógica, um inteiro, uma variável declarada ou uma combinação de tudo isso.

```
def p_condicoes_expressao(p):  
    "condicoes : expressao"  
    p[0]=p[1]
```

Capítulo 7

Testes

Neste capítulo serão realizados testes à linguagem desenvolvida neste projeto.

Esses testes vão consistir na resolução dos 5 exemplos propostos no enunciado do trabalho, bem como a realização de outros 7 exemplos que contenham erros de forma a analisar a sua detecção.

7.1 Exemplos propostos no enunciado do trabalho

7.1.1 Exemplo 1:

"Ler 4 números e dizer se podem ser os lados de um quadrado."

Programa escrito em Suricata:

```
/* declaracoes */

VarInt a|
ArrayInt b[4]|

/* inicio das instrucoes */

inicio:
a=0|
Imprimir("Agora serao preenchidos os valores dos lados do quadrado:\n")|

/* este ciclo vai servir para preencher o array b com os valore do quadrado */

Repete{
  Imprimir("Introduz um valor:")|
  b[a] = Ler()|
  a=a+1|
}Ate(a==4)

/* agora verifica-se se sao ou nao lados de um quadrado */
```

```

Se((b[0]==b[1]) /\ (b[0]==b[2]) /\ (b[0]==b[3])){
    Imprimir("Sao lados de um quadrado\n")|
} Senao{
    Imprimir("Nao sao lados de um quadrado\n")|
}

fim

```

Avisos do Terminal:

AVISO! O array b só admite índices entre 0 e 3!

Figura 7.1: *Aviso no terminal (Exemplo 1).*

Código Assembly gerado para um ficheiro:

```

pushi 0
pushn 4
start
pushi 0
storeg 0
pushs "Agora serao preenchidos os valores dos lados do quadrado:\n"
writes
ciclo0: nop
pushs "Introduz um valor:"
writes
pushgp
pushi 1
padd
pushg 0
read
atoi
storen
pushg 0
pushi 1
add
storeg 0
pushg 0
pushi 4
equal
jz ciclo0
pushgp

```

```

pushi 1
padd
pushi 0
loadn
pushgp
pushi 1
padd
pushi 1
loadn
equal
pushgp
pushi 1
padd
pushi 0
loadn
pushgp
pushi 1
padd
pushi 2
loadn
equal
mul
not
not
pushgp
pushi 1
padd
pushi 0
loadn
pushgp
pushi 1
padd
pushi 3
loadn
equal
mul
not
not
jz senao0
pushs "Sao lados de um quadrado\n"
writes
jump fimSe0
senao0: nop
pushs "Nao sao lados de um quadrado\n"
writes
fimSe0: nop
stop

```

Testes na VM:

```
Agora serao preenchidos os valores dos lados do c
Introduz um valor:2
Introduz um valor:2
Introduz um valor:2
Introduz um valor:2
Sao lados de um quadrado
```

Figura 7.2: Exemplo de um caso em que são lados de um quadrado

```
Agora serao preenchidos os valores dos lados do c
Introduz um valor:2
Introduz um valor:5
Introduz um valor:2
Introduz um valor:2
Nao sao lados de um quadrado
```

Figura 7.3: Exemplo de um caso em que não são lados de um quadrado

7.1.2 Exemplo 2:

"Ler n números e retornar qual o menor número."

Programa escrito em Suricata:

```
VarInt a|
VarInt aux|
VarInt menor|

inicio:

Imprimir("Digite um numero N:")|
a = Ler()|

Se(a==0){

    Imprimir("Nao ha minimo")|

}Senao{

    Se(a==1){

        menor = Ler()|
        Imprimir("O menor numero e:")|
        Imprimir(menor)|
        Imprimir("\n")|

    }Senao{

        Imprimir("Insira um elemento: ")|
        menor = Ler()|
        a = a - 1|
```

```

Repete{
    Imprimir("Insira um elemento: ")|
    aux = Ler()|
    Se(aux<menor){

        menor = aux|

    }Senao{

        a = a - 1|

    }Ate(a==0)

    Imprimir("O menor numero e:")|
    Imprimir(menor)|
    Imprimir("\n")|

}

}

fim

```

Código Assembly gerado para um ficheiro

```

pushi 0
pushi 0
pushi 0
start
pushs "Digite um numero N:"
writes
read
atoi
storeg 0
pushg 0
pushi 0
equal
jz senao2
pushs "Nao ha minimo"
writes
jump fimSe2
senao2: nop
pushg 0
pushi 1

```

```

equal
jz senao1
read
atoi
storeg 2
pushs "0 menor numero e:"
writes
pushg 2
writei
pushs "\n"
writes
jump fimSe1
senao1: nop
pushs "Insira um elemento: "
writes
read
atoi
storeg 2
pushg 0
pushi 1
sub
storeg 0
ciclo0: nop
pushs "Insira um elemento: "
writes
read
atoi
storeg 1
pushg 1
pushg 2
inf
jz senao0
pushg 1
storeg 2
jump fimSe0
senao0: nop
fimSe0: nop
pushg 0
pushi 1
sub
storeg 0
pushg 0
pushi 0
equal
jz ciclo0
pushs "0 menor numero e:"
writes
pushg 2

```



```

writei
pushs "\n"
writes
fimSe1: nop
fimSe2: nop
stop

```

Testes na VM:

```

Digite um numero N:2
Insira um elemento: 10
Insira um elemento: 5
O menor numero e:5

```

Figura 7.4: *Exemplo onde comparamos apenas 2 números*

```

Digite um numero N:4
Insira um elemento: 10
Insira um elemento: 5
Insira um elemento: 4
Insira um elemento: 6
O menor numero e:4

```

Figura 7.5: *Exemplo onde comparamos 4 números*

7.1.3 Exemplo 3:

"Ler N (fornecido pelo utilizador) números e calcular e imprimir o seu produtório"

Programa escrito em Suricata:

```

VarInt n|
VarInt prod|
VarInt x|

inicio:

prod = 1|
n = Ler()|

Repete{

    x = Ler()|
    prod = prod * x|
    n = n - 1|

```

```
}Ate(n==0)
```

```
Imprimir(prod)|
```

```
fim
```

Código Assembly gerado para um ficheiro:

```
pushi 0
pushi 0
pushi 0
start
pushi 1
storeg 1
read
atoi
storeg 0
ciclo0: nop
read
atoi
storeg 2
pushg 1
pushg 2
mul
storeg 1
pushg 0
pushi 1
sub
storeg 0
pushg 0
pushi 0
equal
jz ciclo0
pushg 1
writei
stop
```

Testes na VM:

```

O numero de elementos a multiplicar e: 3
O elemento a multiplicar e: 1
O elemento a multiplicar e: 2
O elemento a multiplicar e: 3
O resultado do produtorio e: 6

```

Figura 7.6: *Multiplicação de 3 números(1,2,3)*

Como verificamos o último número é o 6 que como esperamos é o nosso resultado.

7.1.4 Exemplo 4:

"Contar e imprimir os números ímpares de uma sequência de números naturais."

Programa escrito em Suricata:

```

VarInt aux|
VarInt aux2|
ArrayInt lista[5]|
VarInt count|
VarInt i|
VarInt j|

inicio:

Imprimir("Introduz um valor entre 0 e 5:")|
aux=Ler()|
aux2=aux|
Imprimir("\n")|

Se((aux==0)){
    Imprimir("Nao ha numeros impares para 0 elementos!\n")|
}Senao{
    Repete{

        Imprimir("Coloca um elemento na sequencia: ")|
        lista[i]=Ler()|
        Imprimir("\n")|
        Se(lista[i]>=0){
            i=i+1|
            aux=aux-1|
        }Senao{

```

```

        Imprimir("Nao e um numero natural!\n")|
    }

}Ate(aux==0)

Repete{
    Se((lista[j] % 2)!=0){
        count=count+1|
        Imprimir("O seguinte numero e impar:")|
        Imprimir(lista[j])|
        Imprimir("\n")|

    }

    j=j+1|

    aux2=aux2-1|
}Ate(aux2==0)

Imprimir("Numero de elementos impares:")|
Imprimir(count)|
Imprimir("\n")|

}

fim

```

Avisos do Terminal:

|AVISO! O array lista só admite índices entre 0 e 4!

Figura 7.7: *Aviso no terminal (Exemplo 4).*

Tal como é possível observar na Figura 7.7, no terminal, surge um aviso de que o array *lista*, declarado com 5 elementos, só admite índices entre 0 e 4.

Código Assembly gerado para um ficheiro:

```

pushi 0
pushi 0
pushn 5
pushi 0
pushi 0
pushi 0
start

```

```

pushs "Introduz um valor entre 0 e 5:"
writes
read
atoi
storeg 0
pushg 0
storeg 1
pushs "\n"
writes
pushg 0
pushi 0
equal
jz senao1
pushs "Nao ha numeros impares para 0 elementos!\n"
writes
jump fimSe2
senao1: nop
ciclo0: nop
pushs "Coloca um elemento na sequencia: "
writes
pushgp
pushi 2
padd
pushg 8
read
atoi
storen
pushs "\n"
writes
pushgp
pushi 2
padd
pushg 8
loadn
pushi 0
supeq
jz senao0
pushg 8
pushi 1
add
storeg 8
pushg 0
pushi 1
sub
storeg 0
jump fimSe0
senao0: nop
pushs "Nao e um numero natural!\n"

```

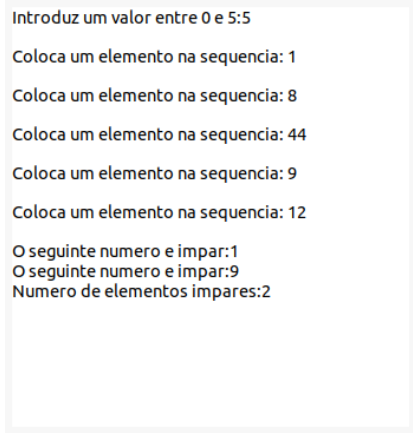
```

writes
fimSe0: nop
pushg 0
pushi 0
equal
jz ciclo0
ciclo1: nop
pushgp
pushi 2
padd
pushg 9
loadn
pushi 2
mod
pushi 0
equal
not
jz fimSe1
pushg 7
pushi 1
add
storeg 7
pushs "0 seguinte numero e impar:"
writes
pushgp
pushi 2
padd
pushg 9
loadn
writei
pushs "\n"
writes
fimSe1: nop
pushg 9
pushi 1
add
storeg 9
pushg 1
pushi 1
sub
storeg 1
pushg 1
pushi 0
equal
jz ciclo1
pushs "Numero de elementos impares:"
writes
pushg 7

```

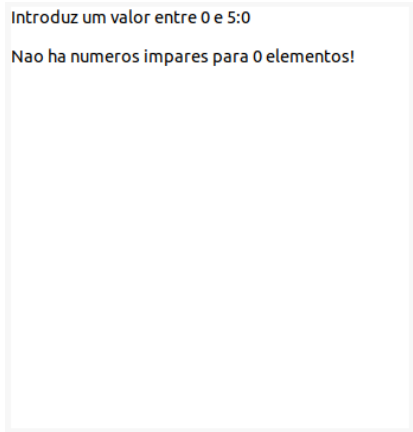
```
writei  
pushs "\n"  
writes  
fimSe2: nop  
stop
```

Testes na VM:



Introduz um valor entre 0 e 5:5
Coloca um elemento na sequencia: 1
Coloca um elemento na sequencia: 8
Coloca um elemento na sequencia: 44
Coloca um elemento na sequencia: 9
Coloca um elemento na sequencia: 12
O seguinte numero e impar:1
O seguinte numero e impar:9
Numero de elementos impares:2

Figura 7.8: *Resultado obtido na VM para uma sequência de 5 números.*



Introduz um valor entre 0 e 5:0
Nao ha numeros impares para 0 elementos!

Figura 7.9: *Resultado obtido na VM para uma sequência de 0 números.*

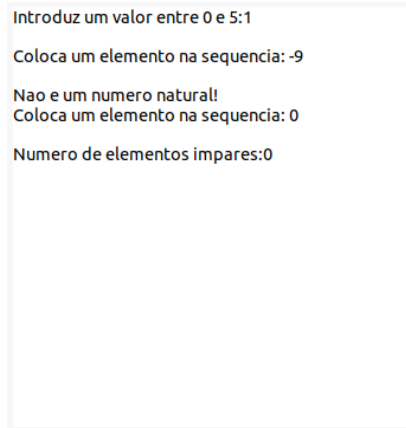


Figura 7.10: Resultado obtido na VM para uma sequência de 1 número.

De notar que, no caso representado na Figura 7.9, caso o número de elementos da sequência introduzido seja 0, o programa avisa que não há números ímpares para 0 elementos. Por outro lado, no caso representado na Figura 7.10, caso se coloque um número não natural (como é o caso do -9) na sequência, o programa avisa que não é um número natural, e volta a pedir para introduzir um elemento na sequência.

7.1.5 Exemplo 5:

"Ler e armazenar N números num array; imprimir os valores por ordem inversa."

Programa escrito em Suricata:

```
VarInt i, N|
ArrayInt numeros[100]|

inicio:

i=0|
Repete{
    Imprimir("Insira o tamanho do array (numero entre 1 e 100): ")|
    N= Ler()|
}
Ate((N>0) /\ (N<=100))

Repete{
    Imprimir("Insira um elemento do array: ")|
    numeros[i]= Ler()|
    i= i+1|
}
Ate(i==N)

Imprimir("Elementos do array por ordem inversa:\n")|
Repete{
    i= i-1|
    Imprimir(numeros[i])|
```



```

        Imprimir("\n")|
    }
    Ate(i==0)

fim

```

Avisos do Terminal:

AVISO! O array *numeros* só admite índices entre 0 e 99!

Figura 7.11: *Aviso no terminal (Exemplo 5).*

Tal como é possível verificar na Figura 7.11, no terminal, surge um aviso de que o array *numeros*, declarado com 100 elementos, só admite índices entre 0 e 99.

Código Assembly gerado para um ficheiro:

```

pushi 0
pushi 0
pushn 100
start
pushi 0
storeg 0
ciclo0: nop
pushs "Insira o tamanho do array (numero entre 1 e 100): "
writes
read
atoi
storeg 1
pushg 1
pushi 0
sup
pushg 1
pushi 100
infeq
mul
not
not
jz ciclo0
ciclo1: nop
pushs "Insira um elemento do array: "
writes
pushgp
pushi 2

```

```

padd
pushg 0
read
atoi
storen
pushg 0
pushi 1
add
storeg 0
pushg 0
pushg 1
equal
jz ciclo1
pushs "Elementos do array por ordem inversa:\n"
writes
ciclo2: nop
pushg 0
pushi 1
sub
storeg 0
pushgp
pushi 2
padd
pushg 0
loadn
writei
pushs "\n"
writes
pushg 0
pushi 0
equal
jz ciclo2
stop

```

Testes na VM:

<pre> Insira o tamanho do array (numero entre 1 e 100): 4 Insira um elemento do array: 8 Insira um elemento do array: 5 Insira um elemento do array: 4 Insira um elemento do array: 11 Elementos do array por ordem inversa: 11 4 5 8 </pre>	<pre> nsira o tamanho do array (numero entre 1 e 100): 4 nsira um elemento do array: 8 nsira um elemento do array: 5 nsira um elemento do array: 4 nsira um elemento do array: 11 Elementos do array por ordem inversa: 11 4 5 3 </pre>
--	---

(a)

(b)

Figura 7.12: Resultado obtido na VM para o caso de $N = 4$ (a figura 7.12b é a continuação da figura 7.12a).

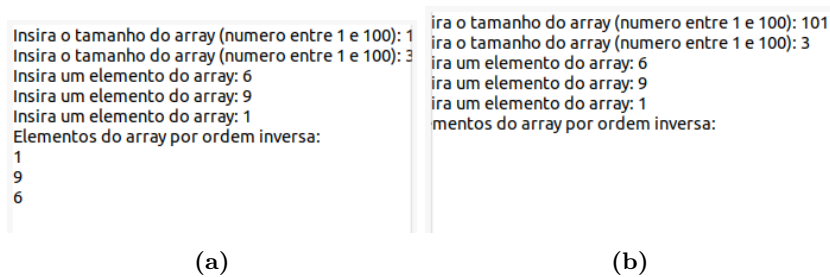


Figura 7.13: Resultado obtido na VM para o caso de $N = 3$ (a figura 7.13b é a continuação da figura 7.13a).

De notar que, no caso representado na Figura 7.13, como o tamanho do array inicialmente inserido pelo utilizador (101) excede o tamanho máximo do array (100), então o programa voltou a pedir ao utilizador para inserir o tamanho do array como sendo um número entre 1 e 100.

7.2 Exemplos de casos de erro

7.2.1 Variável não declarada

No seguinte exemplo vamos mostrar um pequeno código onde não iremos declarar a variável e veremos o erro mostrado pela VM.

Programa escrito em Suricata:

```

inicio:
a=3|

fim

```

Resultado obtido no terminal:

Como podemos verificar surge um aviso de que a variável (neste exemplo 'a') não está declarada.

Embora não obtemos um ficheiro .vm conseguimos visualizar o código que entretanto foi transcrito de modo a podermos alterar e visualizar o erro.

7.2.2 Redecaração de uma variável

O seguinte programa redeclara uma variável do tipo *VarInt*, no entanto, o mesmo se podia aplicar para uma redeclaração de um array.

Programa escrito em Suricata:

```

VarInt a,b|
VarInt a|

inicio:
b=3|
a=2|

fim

```

```
início:
a=3|

fim
A variável "a" não foi declarada.
```

```
Código máquina:
start
err"Variável não existe!"
stop
stop
```

Existe erro/os, o código máquina não será gerado para um ficheiro .vm!
Consulte o erro no código acima.

Figura 7.14: *Resultado apresentado no terminal*

Resultados no terminal:

```
Nome do ficheiro a abrir: teste.txt
VarInt a,b|
VarInt a|
```

```
início:

b=3|
a=2|

fim
A variável "a" já foi declarada.
```

```
Código máquina:
pushi 0
pushi 0
err"Variável repetida!"
stop
start
pushi 3
storeg 1
pushi 2
storeg 0
stop
```

Existe erro/os, o código máquina não será gerado para um ficheiro .vm!
Consulte o erro no código acima.

Figura 7.15: *Erro obtido no terminal, proveniente da redeclaração de uma variável VarInt.*

Tal como é possível observar na figura anterior, uma vez redeclarada uma variável, é emitida uma mensagem a indicar a variável em questão. Além disso, como estamos na presença de um erro, não há criação de um ficheiro .vm com o código máquina gerado.

7.2.3 Variável do tipo VarInt com indexação

O seguinte programa declara uma variável do tipo *VarInt*, ou seja, do tipo inteiro, denominada *i*. No entanto, na parte das instruções do programa, a variável *i* encontra-se indexada com o índice 0, e a indexação de uma variável do tipo inteiro corresponde a um erro semântico.

Programa escrito em Suricata:

```
VarInt i|  
  
inicio:  
  
i[0]=3*4|  
Imprimir("...")|  
  
fim
```

Erros e avisos mostrados no terminal:

```
VarInt i|  
  
inicio:  
  
i[0]=3*4|  
Imprimir("...")|  
  
fim  
A variável "i" não admite indexação, ou é um array de dimensão diferente.  
  
Código máquina:  
pushi 0  
start  
err"A variável em questão não admite indexação, ou é um array de dimensão diferente!"  
stop  
pushs "..."  
writes  
stop  
  
Existe erro/os, o código máquina não será gerado para um ficheiro .vm!  
Consulte o erro no código acima.
```

Figura 7.16: Erro obtido no terminal, proveniente da indexação de uma variável *VarInt*.

Tal como é possível observar na Figura 7.16, no terminal aparece uma mensagem de erro, que avisa que a variável *i* não admite indexação.

Uma vez que se detetou erros, não foi gerado código Assembly para um ficheiro .vm, tal como é indicado no aviso visível na parte inferior da Figura 7.16, impresso no terminal.

O código que este aviso pede para consultar é o código Assembly gerado a partir deste programa com erros, servindo assim como mais um instrumento que o utilizador pode utilizar de forma a identificar e corrigir o(s) erro(s) presente(s).

Código Assembly gerado para este programa:

```
pushi 0
start
err"A variável em questão não admite indexação, ou é um array de dimensão diferente!"
stop
pushs "... "
writes
stop
```

7.2.4 Variável do tipo *ArrayInt* sem indexação

O seguinte programa declara uma variável do tipo *ArrayInt*, ou seja, um array de inteiros, denominada *a*. No entanto, na parte das instruções do programa, existe uma atribuição à variável *a*, sem que esta esteja indexada, o que corresponde a um erro semântico.

Programa escrito em Suricata:

```
ArrayInt a[3] |

inicio:

a= 21%4 |
Imprimir("...") |

fim
```

Erros e avisos mostrados no terminal:

```

ArrayInt a[3]|

inicio:

a= 21%4|
Imprimir("...")|

fim

```

A variável "a" precisa de indexação.

```

Código máquina:
pushn 3
start
err"Indexação em falta!"
stop
pushs "..."
writes
stop

```

Existe erro/os, o código máquina não será gerado para um ficheiro .vm!
Consulte o erro no código acima.

Figura 7.17: *Erro obtido no terminal, proveniente de não indexar uma variável ArrayInt.*

Tal como é possível observar na Figura 7.17, no terminal aparece uma mensagem de erro, que avisa que a variável *a* necessita de indexação.

Uma vez que se detetou erros, não foi gerado código Assembly para um ficheiro .vm, tal como é indicado no aviso visível na parte inferior da Figura 7.17, impresso no terminal.

Tal como foi explicado anteriormente, o código que este aviso pede para consultar é o código Assembly gerado a partir deste programa com erros, servindo assim como mais um instrumento que o utilizador pode utilizar de forma a identificar e corrigir o(s) erro(s) presente(s).

7.2.5 ArrayInt unidimensional com 2 índices

O seguinte programa declara um array unidimensional com dimensão 3 porém vamos tentar definir um valor para a posição usando 2 índices, o que gerará um erro visto não termos declarado acima um array bidimensional.

Programa escrito em Suricata:

```

ArrayInt a[3]|

inicio:
a[1][2]=6|

```

```
fim
```

Erros e avisos mostrados no terminal:

```
ArrayInt a[3]|  
  
inicio:  
a[1][2]=6|  
  
fim  
A variável "a" não admite indexação, ou é um array de dimensão diferente.  
  
Código máquina:  
pushn 3  
start  
err"A variável em questão não admite indexação, ou é um array de dimensão diferente!"  
stop  
stop  
  
Existe erro/os, o código máquina não será gerado para um ficheiro .vm!  
Consulte o erro no código acima.
```

Figura 7.18: Erro obtido no terminal, proveniente de usar dois índices para indexar um array unidimensional.

Tal como podemos verificar, obtemos um erro associado ao fato de indexarmos um array unidimensional com 2 índices. Além disto, o código máquina não é guardado num ficheiro .vm, em vez disso, é nos apresentado no terminal de modo que o utilizador possa descobrir onde o erro se encontra.

7.2.6 Erro léxico:

Este exemplo simples retrata o que acontece se forem utilizados caracteres que, no contexto em que se encontram, não pertencem à nossa linguagem.

Para tal decidimos introduzir o '?' nas declarações, de forma a visualizar o efeito. O '?' pode ser utilizado na nossa linguagem dentro de aspas, mas não no contexto em que se encontra.

Programa escrito em Suricata:

```
/* introduzimos o '?', mas neste contexto nao pertence a linguagem */  
  
VarInt ?|  
  
inicio:  
  
a=1|  
Imprimir("...")|
```



```
fim
```

Erros e avisos mostrados no terminal:

```
/* introduzimos o '?', mas neste contexto nao pertence a linguagem */
```

```
VarInt ?|
```

```
inicio:
```

```
a=1|
```

```
Imprimir("...")|
```

```
fim
```

```
Erro léxico, character inválido na linha: 1
```

```
Ocorreu um erro sintático: LexToken(|,'|',1,79)
```

```
A variável "a" não foi declarada.
```

```
Código máquina:
```

```
start
```

```
err"Variável não existe!"
```

```
stop
```

```
pushs "..."
```

```
writes
```

```
stop
```

```
Existe erro/os, o código máquina não será gerado para um ficheiro .vm!
```

```
Consulte o erro no código acima.
```

Figura 7.19: Erros e informações obtidas no terminal, proveniente de um programa com erro léxico.

Verifica-se que foi identificado um erro léxico na linha 1, tal como era pretendido.

Podemos ver também que esse erro léxico desencadeou um erro sintático, pois o '?' nem sequer existe de forma a poder corresponder à gramática, e ainda desencadeou um erro semântico, pois a variável 'a' nem sequer foi declarada (ela era a que estava presente no lugar do '?'). Visto isto verificamos que tudo corresponde ao esperado.

Uma vez que se detectou erros, não foi gerado código Assembly para um ficheiro .vm, tal como indica esta informação impressa no terminal:

Existe erro/os, o código máquina não será gerado para um ficheiro .vm!
Consulte o erro no código acima.

Este código que o aviso pede para consultar é o código Assembly gerado por este programa com erros, servindo assim como mais um instrumento que o utilizador pode utilizar de forma a identificar e corrigir o/os erros presentes.

7.2.7 Erro sintático:

Este exemplo simples retrata o que acontece se a gramática for violada.

Para tal decidimos colocar indexação na declaração de uma variável do tipo VarInt, o que sintaticamente a nossa linguagem não permite, de forma a visualizar o efeito.

Programa escrito em Suricata:

```
/* introduziu-se indexacao numa variavel VarInt, de forma a introduzir um erro sintatico */  
  
VarInt a[1]|  
  
inicio:  
  
a=1|  
Imprimir("...")|  
  
fim
```

Erros e avisos mostrados no terminal:

```
/* introduziu-se indexacao numa variavel VarInt, de forma a introduzir um erro sintatico */
```

```
VarInt a[1]|
```

```
inicio:
```

```
a=1|
```

```
Imprimir("...")|
```

```
fim
```

```
Ocorreu um erro sintático: LexToken(['',1,103)
```

```
A variável "a" não foi declarada.
```

```
Código máquina:
```

```
start
```

```
err"Variável não existe!"
```

```
stop
```

```
pushs "..."
```

```
writes
```

```
stop
```

```
Existe erro/os, o código máquina não será gerado para um ficheiro .vm!
```

```
Consulte o erro no código acima.
```

Figura 7.20: *Erros e informações obtidas no terminal, proveniente de um programa com erro sintático.*

Verifica-se, tal como era esperado, que ocorreu um erro sintático. Este erro levou à criação de outro erro, tal como no exemplo anterior, pois a variável 'a' é tida como não declarada.

Uma vez que se detectou erros, não foi gerado código Assembly para um ficheiro .vm, tal como indica esta informação impressa no terminal:

```
Existe erro/os, o código máquina não será gerado para um ficheiro .vm!
```

```
Consulte o erro no código acima.
```

O código que a informação pede para consultar é o código Assembly gerado por este programa com erros, de forma ao utilizador ter mais um instrumento para poder detetar e corrigir erros.

Capítulo 8

Conclusão

Este documento aborda a realização do trabalho prático nº 2, relativo a Gramáticas e Compiladores. São discutidos o enquadramento e o objetivo do trabalho, bem como o problema inerente ao mesmo, a abordagem utilizada para resolvê-lo, e quais os requisitos a cumprir. Este documento explica em detalhe as componentes principais da resolução do problema: a Gramática Independente do Contexto (GIC), o Analisador Léxico, a conversão da GIC para Python e o Compilador Yacc. Inclui também os testes realizados, que mostram não só exemplos de programas escritos na linguagem *Suricata*, como também a respetiva transformação em código máquina e execução na máquina virtual VM. Além disso, os testes realizados incluem casos de erro. Tendo em conta os requisitos mencionados para este trabalho prático, consideramos que todos estes foram cumpridos com sucesso. Para além dos requisitos obrigatórios do trabalho prático, foi também possível implementar a possibilidade de escrever comentários na linguagem *Suricata*. Além disso, consideramos que a linguagem *Suricata* foi construída de tal modo que é fácil de utilizar, sendo relativamente simples para o utilizador compreender a causa de alguns erros que podem surgir ao executar programas escritos nessa linguagem.

Por outro lado, possibilidades de melhoria deste trabalho incluem: conseguir implementar não só ciclos repeat-until, como também ciclos while-do e for-do; permitir definir e invocar subprogramas; permitir identificar *runtime errors*; permitir atribuir às variáveis nomes começados por palavras reservadas (tais como *Senhora* ou *inícioVar*).

Outra melhoria que se poderia implementar é relativa aos dicionários das variáveis. Quando estamos na presença de uma variável, esta é guardada em *parser.variaveis*. Ao guardar, é fornecido um array com o tipo da variável (tipo), com a "distância entre a variável e o global pointer"(*d_gp*) e com as características da mesma (*caract*). Essas características variam de acordo com o tipo da variável. Se for do tipo *VarInt*, é adicionado ao array o 1, enquanto se for um array 1D é adicionado a dimensão do mesmo. Por sua vez, no caso de um array 2D, é indicado tanto a dimensão (*linhas*colunas*), como o número de linhas e colunas. Assim, para as variáveis do tipo *VarInt* guardávamos [*d_gp* , *caract* , tipo], já para as do tipo *ArrayInt* guardávamos no formato [*caract*, *d_gp*, tipo]. O facto de não terem a mesma estrutura obrigou-nos posteriormente a ter de criar mais funções auxiliares. Portanto, a melhoria a fazer consiste em fazer com que os arrays correspondentes às 3 variáveis distintas possuam a mesma estrutura e, deste modo, possibilitar uma redução no número de funções auxiliares.

Para concluir, consideramos que este trabalho prático permitiu-nos aprofundar bastante os nossos conhecimentos acerca de Gramáticas e Compiladores, bem como acerca das análises léxica, sintática e semântica, já que nos obrigou a aplicar estes conhecimentos de modo a construir uma linguagem de programação cujos programas pudessem ser executados.

Capítulo 9

Código

9.1 Analisador Léxico

```
import sys

import ply.lex as lex
import re

states=[('comentario','inclusive')]
tokens = ['OFF','IN','VarInt', 'ArrayInt', 'Inicio', 'Fim', 'Se', 'Senao',
          'Repete', 'Ate', 'Ler', 'Imprimir', 'String','Num', 'Num_neg','ID',
          'Maiorouigual','Menorouigual','Igualigual','Diferente','And','Or','
          Not']

literals = ['*', '+', '%', '/', '-', '=', '(', ')', '.', '<', '>', ',', '{',
            '}', '[', ']', '|', ':']

def t_And(t):
    r'\/\\'
    return t

def t_Or(t):
    r'\\/\\/'
    return t

def t_Not(t):
    r'!'
    return t
```

```

def t_Maiorouigual(t):
    r'>='
    return t

def t_Igualigual(t):
    r'=='
    return t

def t_Diferente(t):
    r'!='
    return t

def t_Menorouigual(t):
    r'<='
    return t

def t_VarInt(t):
    r'VarInt'
    return t

def t_ArrayInt(t):
    r'ArrayInt'
    return t

def t_Inicio(t):
    r'inicio'
    return t

def t_Fim(t):
    r'fim'
    return t

def t_Senao(t):
    r'Senao'
    return t

def t_Se(t):
    r'Se'
    return t

def t_Repete(t):
    r'Repete'
    return t

def t_Ate(t):
    r'Ate'

```

```

        return t

def t_Ler(t):
    r'Ler'
    return t

def t_Imprimir(t):
    r'Imprimir'
    return t

def t_comentario(t):
    r'/*'
    t.lexer.begin('comentario')

def t_comentario_OFF(t):
    r'*/'
    t.lexer.begin('INITIAL')

def t_comentario_IN(t):
    r'(\n)'

def t_String(t):
    r'"[^"]+"'

    return t

def t_Num(t):
    r'\d+'
    return t

def t_Num_neg(t):
    r'\(-\d+\)'
    res = re.search(r'\d+', t.value)
    t.value = res.group(0)
    return t

def t_ID(t):
    r'\w+'
    return t

t_ignore=' \n\t'

def t_error(t):
    print('Erro léxico, caracter inválido na linha:',t.lexer.lineno)

```

```

        t.lexer.skip(1) ## ingnora o erro e continua

lexer=lex.lex()

```

9.2 Compilador Yacc

```

import re
import ply.yacc as yacc
import sys
from projeto_analisador_lexico14jan import tokens

## funções auxiliares:

# se a função retornar 1 é porque a variável já foi declarada,
# ...se retornar 0 é porque não foi
def check_variable(variavel,variaveis_dic):
    if variavel in variaveis_dic:
        return 1
    else:
        return 0

# esta função retorna o valor do stackPointer de uma variável VarInt
def return_sp_VarInt(variavel,variaveis_dic):
    sp=variaveis_dic[variavel][0]
    return sp

# esta função retorna o valor do stackPointer de uma variável ArrayInt
def return_sp_ArrayInt(variavel,variaveis_dic):
    sp=variaveis_dic[variavel][1]
    return sp

# esta função retorna a dimensão de um ArrayInt 1D
def return_dimensions_ArraInt1D(variavel,variaveis_dic):
    dim=variaveis_dic[variavel][0]
    return [dim]

# esta função retorna uma lista:[dimensão,n_linhas,n_colunas] de um ArrayInt 2D
def return_dimensions_ArraInt2D(variavel,variaveis_dic):
    dim=variaveis_dic[variavel][0][0]
    n_linhas=variaveis_dic[variavel][0][1]

```



```

n_colunas=variaveis_dic[variavel][0][2]
return [dim,n_linhas,n_colunas]

# retorna o tipo
def return_tipo(variavel,variaveis_dic):
    tipo=variaveis_dic[variavel][2]
    return tipo

def p_programa(p):
    "programa : declaracoes statements"

    p[0] = p[1] + p[2]
    parser.guardar = p[0]
    if(parser.success):
        if (len(parser.avisos1D_variaveis)):
            for key in parser.avisos1D_variaveis:
                print("AVISO! O array " + parser.avisos1D_variaveis[key][0] +
                    " só admite índices entre 0"+
                    str(parser.avisos1D_variaveis[key][1]))+"!\n")

        if (len(parser.avisos2D_variaveis)):
            for key in parser.avisos2D_variaveis:
                print("AVISO! O array " + parser.avisos2D_variaveis[key][0] +
                    " admite índices:\nLinhas entre 0 e "+
                    str(parser.avisos2D_variaveis[key][1][1]))+"\nColunas entre 0 e "+
                    str(parser.avisos2D_variaveis[key][2][1]))+"\n")

        print("Código máquina:\n"+ parser.guardar)
    else:

        print("Código máquina:\n" + parser.guardar)
        print("\n\nExiste erro/os, o código máquina não será gerado para um
            ficheiro .vm!\nConsulte o erro no código acima.\n")

```

```
# -----declarações-----
```

```
def p_declaracoes_vazio(p):  
    "declaracoes : "  
    p[0] = ""
```

```
def p_declaracoes(p):  
    "declaracoes : declaracoes declaracao"  
    p[0]=p[1]+p[2]
```

```
def p_declaracao_declVar(p):  
    "declaracao : declVar"  
    p[0] = p[1]
```

```
def p_declaracao_declArray(p):  
    "declaracao : declArray"  
    p[0] = p[1]
```

```
def p_declVar(p):  
    "declVar : VarInt listIds ','"  
    p[0] = p[2]
```

```
def p_listIds_ID(p):  
    "listIds : ID"  
    if (check_variable(p[1], parser.variaveis)):  
        p[0] = ("err\"Variável repetida!\"\n") + "stop\n"  
        print("A variável \""+p[1]+"\" já foi declarada.\n")  
        parser.success = False  
    else:  
        parser.variaveis[p[1]]=["parser.sp,1,\"VarInt\"]  
        p[0]=("pushi 0\n")  
        parser.sp+=1
```

```
def p_listIds_ID_listIds(p):  
    "listIds : listIds ',' ID"  
    if (check_variable(p[3], parser.variaveis)):  
        x = ("err\"Variável repetida!\"\n")+ "stop\n"  
        print("A variável \"" + p[3] + "\" já foi declarada.\n")  
        parser.success = False  
    else:
```

```

        parser.variaveis[p[3]]=[parser.sp,1,"VarInt"]
        x = ("pushi 0\n")
        parser.sp += 1
    p[0] = p[1] + x

def p_declArray(p):
    "declArray : ArrayInt listIdArray '|' "
    p[0] = p[2]

def p_listIdArray_list_listIdArray(p):
    "listIdArray : listIdArray ',' array"
    p[0] = p[1] + p[3]

def p_listIdArray_list(p):
    "listIdArray : array"
    p[0] = p[1]

def p_list_ID1(p):
    "array : ID '[' Num ']' "
    if (check_variable(p[1], parser.variaveis)):
        p[0] = ("err\"Variável repetida!\n\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" já foi declarada.\n")
        parser.success = False
    else:
        parser.variaveis[p[1]]=[(int(p[3]),parser.sp,"ArrayInt")]
        p[0] = (f"pushn {int(p[3])}\n")
        parser.sp += int(p[3])

def p_list_ID2(p):
    "array : ID '[' Num ']' '[' Num ']' "
    if (check_variable(p[1], parser.variaveis)):
        p[0] = ("err\"Variável repetida!\n\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" já foi declarada.\n")
        parser.success = False
    else:
        dim_bi = int(p[3]) * int(p[6]) #dim_bi = numero de elementos
        parser.variaveis[p[1]]=[(dim_bi,int(p[3]),int(p[6])),parser.sp,"DoubleArrayInt"]
        p[0] = (f'pushn {dim_bi}\n')
        parser.sp += dim_bi

```

```

# -----statements-----

def p_statements(p):
    "statements : Inicio ':' instrucoes Fim"
    p[0] = "start\n" + p[3] + "stop\n"

def p_instrucoes_instrucao(p):
    "instrucoes : instrucao"
    p[0] = p[1]

def p_instrucoes_instrucao_instrucoes(p):
    "instrucoes : instrucoes instrucao"
    p[0] = p[1] + p[2]

def p_instrucao_atribuicao(p):
    "instrucao : atribuicao"
    p[0] = p[1]

def p_instrucao_se(p):
    "instrucao : se"
    p[0] = p[1]

def p_instrucao_imprimir(p):
    "instrucao : imprimir"
    p[0] = p[1]

def p_instrucao_ciclo(p):
    "instrucao : ciclo"
    p[0] = p[1]

def p_atribuicao_ID_expressao(p):
    "atribuicao : ID '=' expressao '|' "
    # ver se a variavel está declarada. Se nao estiver enviamos um erro

    # Se estiver declarada entao não há erro
    if (check_variable(p[1],parser.variaveis)):
        if (return_tipo(p[1],parser.variaveis)=="VarInt"):

```

```

        stackpointer=return_sp_VarInt(p[1],parser.variaveis)
        p[0] = p[3] + f"storeg {stackpointer}\n"

    else:
        p[0] = ("err\"Indexação em falta!\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" precisa de indexação.\n")
        parser.success = False

else:
    # a variavel nao esta declarada
    p[0] = ("err\"Variável não existe!\n")+ "stop\n"
    print("A variável \"" + p[1] + "\" não foi declarada.\n")
    parser.success = False

def p_atribuicao_ID_ler(p):
    "atribuicao : ID '=' Ler '(' ')' ','"

    # ver se a variavel está declarada. Se nao estiver enviamos um erro
    # Se estiver declarada entao não há erro
    if (check_variable(p[1],parser.variaveis)):
        if(return_tipo(p[1],parser.variaveis)=="VarInt"):
            stackpointer = return_sp_VarInt(p[1], parser.variaveis)
            p[0] = f"read\natoi\n" + f"storeg {stackpointer}\n"
        else:
            p[0] = ("err\"Indexação em falta!\n")+ "stop\n"
            print("A variável \"" + p[1] + "\" precisa de indexação.\n")
            parser.success = False
    else:
        # a variavel nao esta declarada
        p[0] = ("err\"Variável não existe!\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" não foi declarada.\n")
        parser.success = False

def p_atribuicao_ID_expressao1D(p):
    "atribuicao : ID '[' expressao ']' '=' expressao '|'"
    if (check_variable(p[1],parser.variaveis)):
        if (return_tipo(p[1],parser.variaveis)=="ArrayInt"):
            dim=return_dimensions_ArraInt1D(p[1],parser.variaveis)
            parser.aviso1D_variaveis[p[1]]=p[1],dim[0]-1

```

```

        stackpointer=return_sp_ArrayInt(p[1],parser.variaveis)
        p[0]= f"pushgp\npushi {stackpointer}\npadd\n" + p[3] + p[6] + "storen\n"
    else:
        p[0] = ("err\"A variável em questão não admite indexação,
                ou é um array de dimensão diferente!\n\n")+ "stop\n"

        print("A variável \"" + p[1] + "\" não admite indexação,
                ou é um array de dimensão diferente.\n")

        parser.success = False

```

```

else:
    # a variavel nao esta declarada
    p[0] = ("err\"Variável não declarada!\n\n")+ "stop\n"
    print("A variável \"" + p[1] + "\" não está declarada.\n")
    parser.success = False

```

```

def p_atribuicao_ID_ler1D(p):
    "atribuicao : ID '[' expressao ']' '=' Ler '(' ')' '|' '"
    if (check_variable(p[1],parser.variaveis)):
        if (return_tipo(p[1], parser.variaveis) == "ArrayInt"):
            dim = return_dimensions_ArraInt1D(p[1], parser.variaveis)
            parser.avisolD_variaveis[p[1]] = [p[1], dim[0] - 1]
            stackpointer = return_sp_ArrayInt(p[1], parser.variaveis)
            p[0] =f"pushgp\npushi {stackpointer}\npadd\n" + p[3]+ f"read\natoi\n" + f"storen\n"

        else:
            p[0] = ("err\"A variável em questão não admite indexação,
                    ou é um array de dimensão diferente!\n\n")+ "stop\n"

            print("A variável \"" + p[1] + "\" não admite indexação,
                    ou é um array de dimensão diferente.\n")

            parser.success = False

    else:
        # a variavel nao esta declarada
        p[0] = ("err\"Variável não declarada!\n\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" não está declarada.\n")
        parser.success = False

```

```

def p_atribuicao_ID_expressao2D(p):
    "atribuicao : ID '[' expressao ']' '[' expressao ']' '=' expressao '|' "
    if (check_variable(p[1],parser.variaveis)):
        if(return_tipo(p[1],parser.variaveis)=="DoubleArrayInt"):
            stackpointer=return_sp_ArrayInt(p[1],parser.variaveis)
            dimensoes=return_dimensions_ArraInt2D(p[1],parser.variaveis)
            colunas=dimensoes[2]
            linhas=dimensoes[1]
            parser.avisos2D_variaveis[p[1]]=p[1],(0,linhas-1),(0,colunas-1)]
            p[0]= f"pushgp\npushi {stackpointer}\npadd\n" + p[3] + f"pushi {colunas}\n" +
                "mul\n" + p[6] + "add\n" + p[9] + "store\n"
        else:
            p[0] = ("err\A variável em questão não admite indexação,
                ou é um array de dimensão diferente!\n\n")+ "stop\n"

            print("A variável \"" + p[1] + "\" não admite indexação,
                ou é um array de dimensão diferente.\n")

            parser.success = False

    else:
        # a variavel nao esta declarada
        p[0] = ("err\Variável não declarada!\n\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" não está declarada.\n")
        parser.success = False

def p_atribuicao_ID_ler2D(p):
    "atribuicao : ID '[' expressao ']' '[' expressao ']' '=' Ler '(' ')' '|' "
    if (check_variable(p[1],parser.variaveis)):
        if(return_tipo(p[1],parser.variaveis)=="DoubleArrayInt"):
            parser.avisos2D=1
            stackpointer = return_sp_ArrayInt(p[1], parser.variaveis)
            dimensoes = return_dimensions_ArraInt2D(p[1], parser.variaveis)
            colunas = dimensoes[2]
            linhas = dimensoes[1]
            parser.avisos2D_variaveis[p[1]] = p[1], (0, linhas - 1), (0, colunas - 1)]
            p[0]= f"pushgp\npushi {stackpointer}\npadd\n" + p[3] + f"pushi {colunas}\n"+
                "mul\n" + p[6]+ "add\n" + f"read\natoi\n" + "store\n"
        else:
            p[0] = ("err\A variável em questão não admite indexação,
                ou é um array de dimensão diferente!\n\n")+ "stop\n"

```

```

        print("A variável \"" + p[1] + "\"" não admite indexação,
              ou é um array de dimensão diferente.\n")

        parser.success = False
    else:
        # a variavel nao esta declarada
        p[0] = ("err\"Variável não declarada!\n")+ "stop\n"
        print("A variável \"" + p[1] + "\"" não está declarada.\n")
        parser.success = False

def p_se_Se(p):
    "se : Se '(' condicoes ')' '{' conteudoSeRep '}'"
    p[0] = p[3] + f"jz fimSe{parser.fimSeCount}\n" + p[6] +
        f"fimSe{parser.fimSeCount}: nop\n"

    parser.fimSeCount +=1

def p_se_Se_Senao(p):
    "se : Se '(' condicoes ')' '{' conteudoSeRep '}' Senao '{' conteudoSeRep '}'"
    p[0] = p[3] + f"jz senao{parser.senaoCount}\n" + p[6] + f"jump fimSe{parser.fimSeCount}\n" +
        f"senao{parser.senaoCount}: nop\n" + p[10] + f"fimSe{parser.fimSeCount}: nop\n"

    parser.senaoCount +=1
    parser.fimSeCount +=1

def p_conteudoSeRep_vazio(p):
    "conteudoSeRep : "
    p[0]=" "

def p_conteudoSeRep_instrucoes(p):
    "conteudoSeRep : instrucoes"
    p[0]=p[1]

def p_ciclo(p):
    "ciclo : Repete '{' conteudoSeRep '}' Ate '(' condicoes ')'"
    p[0] = f"ciclo{parser.cicloCount}: nop\n" + p[3] + p[7] + f"jz ciclo{parser.cicloCount}\n"
    parser.cicloCount +=1

```



```

def p_imprimir_expressao(p):
    "imprimir : Imprimir '(' expressao ')', '|'"
    p[0] = p[3] + "writei\n"

def p_imprimir_string(p):
    "imprimir : Imprimir '(' String ')', '|'"
    p[0] = f"pushs {p[3]}\nwrites\n"

def p_elemento_ID(p):
    "elemento : ID"
    if (check_variable(p[1],parser.variaveis)):
        if(return_tipo(p[1],parser.variaveis)=="VarInt"):
            stackpointer=return_sp_VarInt(p[1],parser.variaveis)
            p[0] = f"pushg {stackpointer}\n"
        else:
            p[0] = ("err\Indexação em falta!\n\n")+ "stop\n"
            print("A variável \"" + p[1] + "\" precisa de indexação.\n")
            parser.success = False

    else:
        p[0] = ("err\Variável não declarada!\n\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" não está declarada.\n")
        parser.success = False

def p_elemento_ID_expressao1D(p):
    "elemento : ID '[' expressao ']' "
    if (check_variable(p[1],parser.variaveis)):
        if(return_tipo(p[1],parser.variaveis)=="ArrayInt"):
            dim = return_dimensions_ArraInt1D(p[1], parser.variaveis)
            parser.aviso1D_variaveis[p[1]] = [p[1], dim[0] - 1]
            stackpointer = return_sp_ArrayInt(p[1], parser.variaveis)
            p[0] = f"pushgp\npushi {stackpointer}\npadd\n"+p[3]+"loadn\n"

        else:
            p[0] = ("err\A variável em questão não admite indexação,

```

```

        ou é um array de dimensão diferente!\n")+ "stop\n"

print("A variável \"" + p[1] + "\" não admite indexação,
      ou é um array de dimensão diferente.\n")

parser.success = False

else:
    p[0] = ("err\"Variável não declarada!\n")+ "stop\n"
    print("A variável \"" + p[1] + "\" não está declarada.\n")
    parser.success = False

def p_elemento_ID_expressao2D(p):
    "elemento : ID '[' expressao ']' '[' expressao ']' "
    if (check_variable(p[1],parser.variaveis)):
        if (return_tipo(p[1], parser.variaveis) == "DoubleArrayInt"):
            parser.avisos2D=1
            stackpointer = return_sp_ArrayInt(p[1], parser.variaveis)
            dimensoes = return_dimensions_ArraInt2D(p[1], parser.variaveis)
            colunas = dimensoes[2]
            linhas = dimensoes[1]
            parser.avisos2D_variaveis[p[1]] = [p[1], (0, linhas - 1), (0, colunas - 1)]
            p[0] = f"pushgp\npushi {stackpointer}\npadd\n" + p[3] + f"pushi {colunas}\n" +
                "mul\n" + p[6] + "add\n" + "loadn\n"

        else:
            p[0] = ("err\"A variável em questão não admite indexação,
                    ou é um array de dimensão diferente!\n")+ "stop\n"

            print("A variável \"" + p[1] + "\" não admite indexação,
                  ou é um array de dimensão diferente.\n")

            parser.success = False

    else:
        p[0] = ("err\"Variável não declarada!\n")+ "stop\n"
        print("A variável \"" + p[1] + "\" não está declarada.\n")
        parser.success = False

def p_expressao_mais(p):

```

```

    "expressao : expressao '+' termo"
    p[0] = p[1] + p[3] + "add\n"

def p_expressao_menos(p):
    "expressao : expressao '-' termo"
    p[0] = p[1] + p[3] + "sub\n"

def p_expressao_termo(p):
    "expressao : termo"
    p[0]=p[1]

def p_expressao_and(p):
    "expressao : expressao And termo"
    p[0] = p[1]+p[3]+"mul\n" + "not\n" +"not\n" # retorna 1 ou 0

def p_expressao_or(p):
    "expressao : expressao Or termo"
    p[0] = p[1] + p[3] + "add\n" + p[1] + p[3] + "mul\n" +
        "sub\n" + "not\n" + "not\n" # retorna 1 ou 0

def p_expressao_Igualigual(p):
    "expressao : expressao Igualigual termo"
    p[0]=p[1]+p[3]+"equal\n"

def p_expressao_dif(p):
    "expressao : expressao Diferente termo"
    p[0]=p[1]+p[3]+"equal\n"+"not\n"

def p_expressao_Maiorouigual(p):
    "expressao : expressao Maiorouigual termo"
    p[0]=p[1]+p[3]+"supeq\n"

def p_expressao_Menorouigual(p):
    "expressao : expressao Menorouigual termo"
    p[0] = p[1] + p[3] + "infeq\n"

```

```

def p_expressao_Not(p):
    "expressao : Not '(' expressao ')'"
    p[0]=p[3]+"not\n"

def p_expressao_Menor_expressao(p):
    "expressao : expressao '<' termo"
    p[0]=p[1]+p[3]+"inf\n"

def p_expressao_Maior_expressao(p):
    "expressao : expressao '>' termo"
    p[0] = p[1] + p[3] + "sup\n"

def p_termo_vezes(p):
    "termo : termo '*' fator"
    p[0] = p[1] + p[3] + "mul\n"

def p_termo_dividir(p):
    "termo : termo '/' fator"
    p[0] = p[1] + p[3] + "div\n"

def p_termo_divisao_inteira(p):
    "termo : termo '%' fator"
    p[0] = p[1] + p[3] + "mod\n"

def p_termo_fator(p):
    "termo : fator"
    p[0]=p[1]

def p_fator_numposneg(p):
    "fator : num_pos_neg"
    p[0]=p[1]

def p_fator_elemento(p):
    "fator : elemento"
    p[0]=p[1]

def p_fator_expressao(p):
    "fator : '(' expressao ')'"

```

```

p[0]=p[2]

def p_numposneg_Num(p):
    "num_pos_neg : Num"
    p[0]= f"pushi {p[1]}\n"

def p_numposneg_Num_neg(p):
    "num_pos_neg : Num_neg"
    p[0] = f"pushi {p[1]}\n"

def p_condicoes_expressao(p):
    "condicoes : expressao"
    p[0]=p[1]

def p_error(p):
    print("Ocorreu um erro sintático: ",p)
    parser.success = False


parser = yacc.yacc()
parser.variaveis = {}
parser.sp = 0
parser.senaoCount= 0
parser.fimSeCount= 0
parser.cicloCount= 0
parser.guardar=""
parser.avisos1D_variaveis={}
parser.avisos2D_variaveis={}


nome1=input("Nome do ficheiro a abrir: ")
file = open(nome1,'r')
contents = file.read()
print(contents)


parser.success = True
res = parser.parse(contents)

if (parser.success):
    nome2=input("Introduz o nome do ficheiro: ")

```

```
ficheiro=open(nome2,'w')  
ficheiro.write(parser.guardar)
```