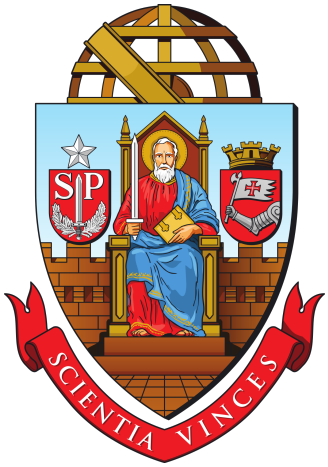
Escola de Engenharia de São Carlos
Instituto de Ciências Matemáticas e de Computação



SCC0605 - Teoria da Computação e Compiladores

2º Trabalho: Análise Sintática

Bruner Eduardo Augusto - nºUSP: 9435846
Carlos Santos Junior - nºUSP: 9435102
Clayton Miccas Junior - nºUSP: 9292441

Docente: Dr. Thiago A. S. Pardo

23 de abril de 2020

Conteúdo

Lista de Figuras

Lista de Códigos

"O sucesso não consiste em não errar,
mas não cometer os mesmos equívocos mais de uma vez."
(George Bernard Shaw)

1 Decisões e Discussões de Projeto

Com base na gramática da linguagem P–, disponível no repositório do Tídia e em anexo neste relatório, enriquecida com o comando “for” (como discutido em aula), e no analisador léxico desenvolvido no Trabalho 1, foi-se desenvolvido o analisador sintático para a linguagem P–, integrando-o com o analisador léxico mencionado anteriormente.

Através das especificações passadas para este trabalho, atentou-se à três diretrizes, sendo estas a correção dos eventuais erros na análise léxica feita no Trabalho 1, a implementação do analisador sintático descendente preditivo recursivo e por fim a implementação do tratamento de erros sintáticos pelo modo pânico.

A ”escolha”de ser um analisador sintático descendente (ASD) preditivo recursivo, deu-se pois caso fosse utilizado o método ”ASD com retrocessos”haveria demasiadas verificações que custariam tempo e em grande maioria seriam descartadas por não estarem corretas, dado que o mesmo funciona com uma metodologia parecida com ”força bruta”, onde este testaria todas as cadeias para todas as regras até encontrar a correta. Já o ASD preditivo recursivo, utilizando a gramática com alguns critérios, produz um resultado satisfatório com menor quantidade de verificações, implicando em um menor custo de tempo para a análise. Os critérios e implementações dessa escolha serão apresentados nos tópicos abaixo.

Conforme o exposto acima e com base no primeiro trabalho realizado, elaborou-se o projeto de um analisador sintático na linguagem python, o qual está em:

```
1 github.com/brunereduardo/Voa_Compila.git
```

2 Analisador Sintático Descendente Preditivo Recursivo

O Analisador Sintático Descendente Preditivo Recursivo é um analisador do tipo *top-down*, onde este necessita que a gramática seja do tipo LL(x), onde x é o número de *tokens* à frente do simbolo em análise. Gramáticas do tipo LL onde o acrônimo advém de *”**L**eftmost derivation”* e *”**L**eft to right”* como o próprio nome sugere, são as gramaticas que possuem sua entrada na direção da esquerda para a direita e tem sua derivação mais à esquerda. Para nossa implementação utilizaremos gramáticas do tipo LL(1) e para isso essa gramática deve seguir dois critérios principais:

Primeiro: Não ser recursiva à esquerda.

Segundo: Para um não terminal qualquer, não devem existir duas regras que comecem com um mesmo terminal, isso é, os conjuntos primeiros devem ser disjuntos.

A partir desses critérios e observando a Linguagem P– em anexo, se fez necessário adaptações para que a mesma seja caracterizada como uma gramática LL(1), se fez a aplicação das Regras de transformação, Regras de tradução e tratamento de erros.

2.1 Transformando a Linguagem P– em uma Gramática LL(1)

Observa-se que a linguagem P– possue 38 linhas de regras e para uma melhor visualização iremos descrever neste tópico apenas as linhas com modificações.

```
1
2 <corpo > ::= <dc> begin <comandos > end
3 <dc> ::= <dc_c > <dc_v > <dc_p >
4 <dc_c > ::= const ident = <numero > ; <dc_c > |

11 <lista_par > ::= <variaveis > :  <tipo_var > <mais_par >
7 <variaveis > ::= ident <mais_var >

13 <corpo_p > ::= <dc_loc > begin <comandos > end ;
14 <dc_loc > ::= <dc_v >
5 <dc_v > ::= var <variaveis > :  <tipo_var > ; <dc_v > |

19 <comandos > ::= <cmd > ; <comandos > |
20 <cmd > ::= read ( <variaveis > ) |
21 write ( <variaveis > ) |
22 while ( <condicao > ) do <cmd > |
23 for ident := <expressao> to <expressao> <cmd> |
24 if <condicao > then <cmd > <pfalsa > |
25 ident := <expressao > |
26 ident <lista_arg > |
27 begin <comandos > end

28 <condicao > ::= <expressao > <relacao > <expressao >
30 <expressao > ::= <termo > <outros_termos >
34 <termo > ::= <op_un > <fator > <mais_fatores >
31 <op_un > ::= + | - |

32 <outros_termos > ::= <op_ad > <termo > <outros_termos > |
33 <op_ad > ::= + | -

35 <mais_fatores > ::= <op_mul > <fator > <mais_fatores > |
36 <op_mul > ::= *| /

37 <fator > ::= ident | <numero > | ( <expressao > )
38 <numero > ::= numero_int | numero_real

15 <lista_arg> ::= ( <argumentos> ) | λ
```

Código 1: Linguagem P– Adaptação para Gramática LL(1)

Os agrupamentos, exemplo (2,3,4), (11,7), etc ..., fez-se necessário verificar se estes não são recursivos à esquerda, dado que seus primeiros são símbolos não terminais também, logo verifica-se que o próximo não terminal resulta em uma recursão a esquerda. Nos demais casos, nenhuma recursão à esquerda foi encontrada, logo não foi necessário nenhuma modificação. Já as regras das linhas 25 e 26 referentes ao não terminal $< cmd >$ necessitam de uma adaptação, dado que começam com o mesmo simbolo terminal.

Cria-se o simbolo não terminal com as seguintes regras: $< ident' >:=< expressao >$ $| < list\_arg >$ altera-se a linha 25 para $ident < ident' >$ e exclue-se a linha 26, isso resulta na solução do problema identificado. Neste momento deve-se verificar o não terminal $< list\_arg >$ que está na linha 15 para o caso de recursão a esquerda, o mesmo não apresenta nenhum problema.

2.2 Transformação da Linguagem em Grafos Sintáticos

Para a construção do analisador sintático pretendido, é necessário a passagem dos conjuntos de instruções como procedimentos, para isso se faz necessário o uso de duas ferramentas: as Regras de transformação e Regras de Tradução. Esta ultima é utilizada para transformar grafos sintáticos em procedimentos, isso é, nos códigos implementados, que em nosso caso é na linguagem Python. Para isso, foi utilizado pequenas estruturas passadas em aula e disponíveis nos slides da mesma, na qual relacionam o grafo sintático a uma estrutura de código. Esta transformação foi feita diretamente no algoritmo disponível no GitHub[1] informado.

Já as Regras de transformação são a ferramenta utilizada para o mapeamento das regras de um não terminal em grafos sintáticos, este mapeamento foi feito para a Linguagem P– abaixo. Ressalta-se que foram aglomerados alguns grafos para uma maior simplicidade e eficiência do código, vale dizer neste ponto que ainda seria possível unir mais grafos destes, mas para uma melhor visualização no relatório, e codificação, o grupo decidiu manter alguns destes, como por exemplo o da figura 2, 6, 8, ... etc. Segue abaixo os 13 grafos desenvolvidos para a linguagem:

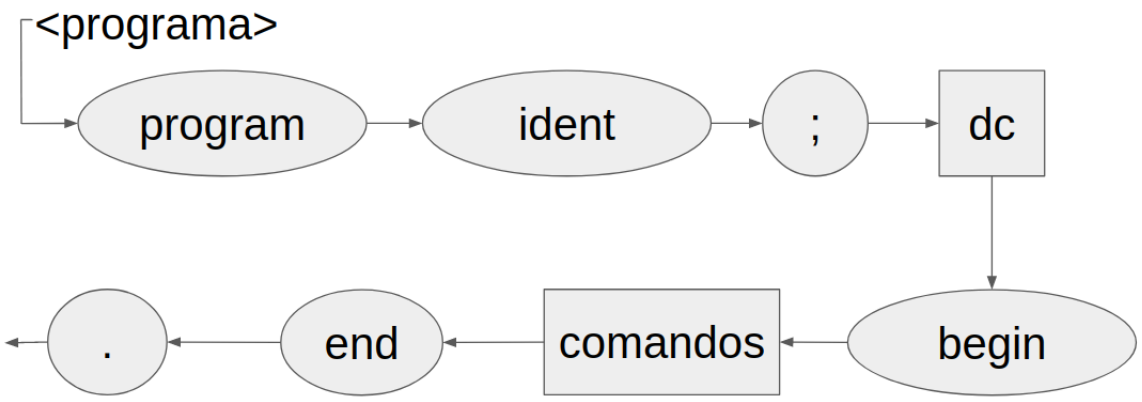


Figura 1: Grafo Sintático do Não Terminal - programa

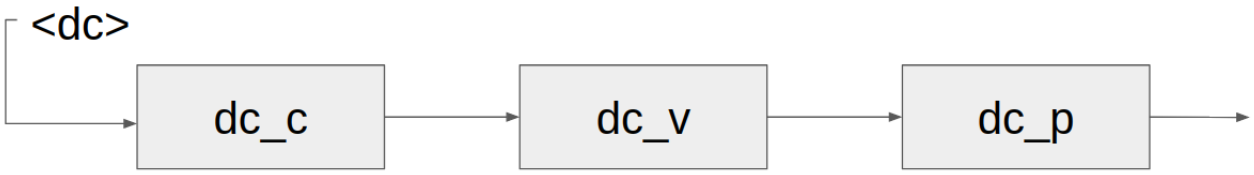


Figura 2: Grafo Sintático do Não Terminal - dc

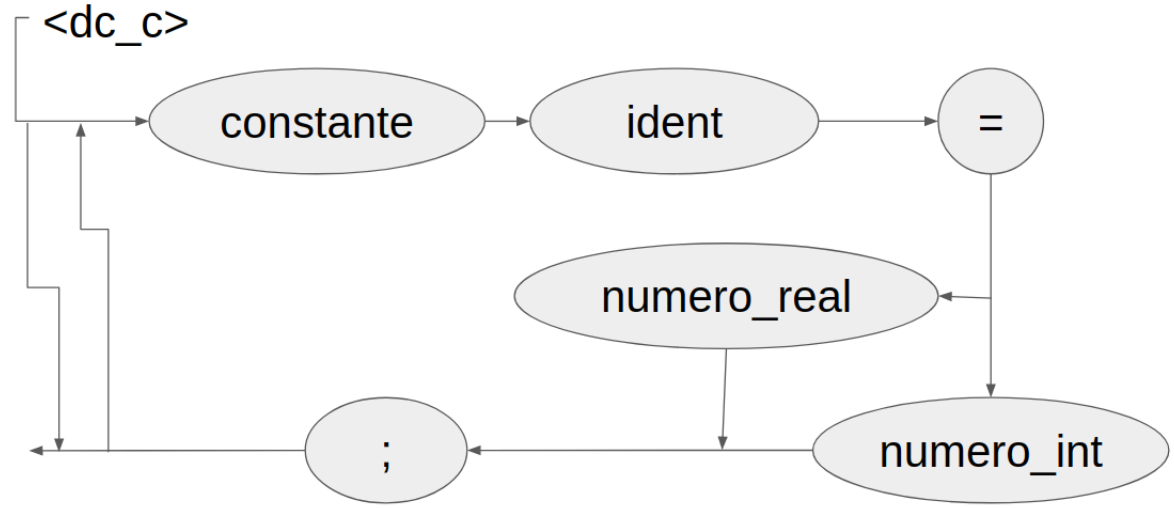


Figura 3: Grafo Sintático do Não Terminal - dc_c

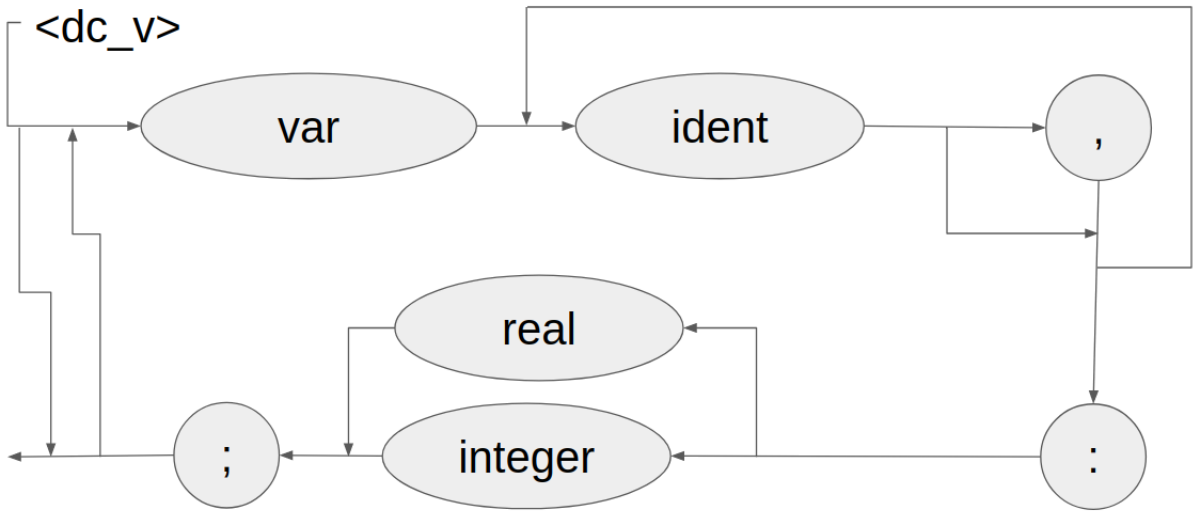


Figura 4: Grafo Sintático do Não Terminal - dc_v

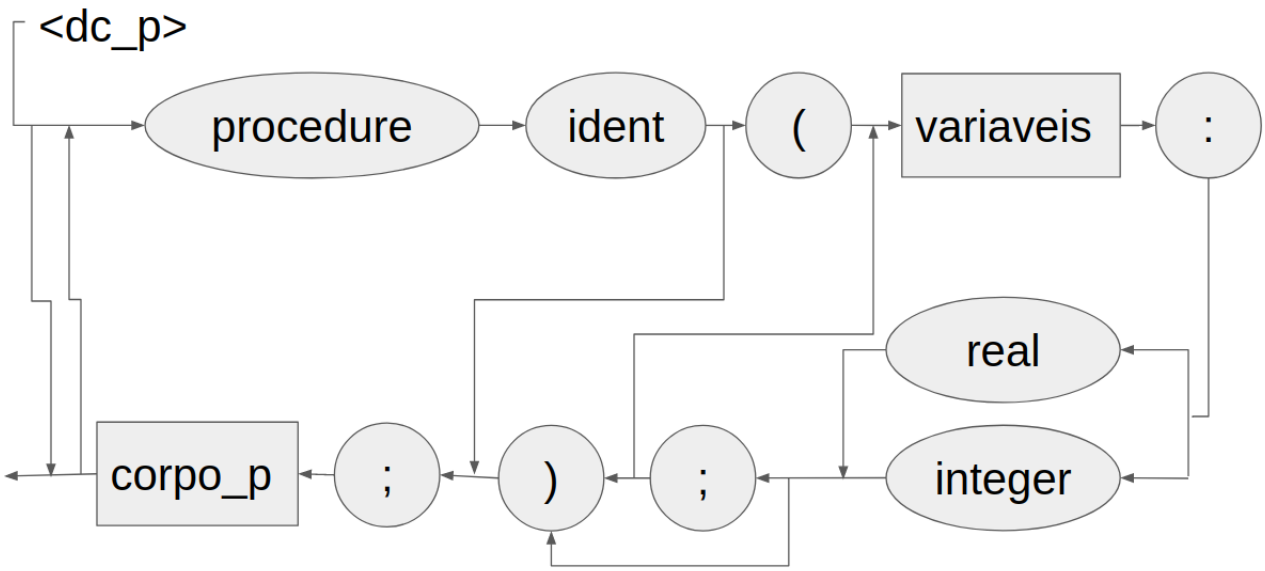


Figura 5: Grafo Sintático do Não Terminal - dc_p

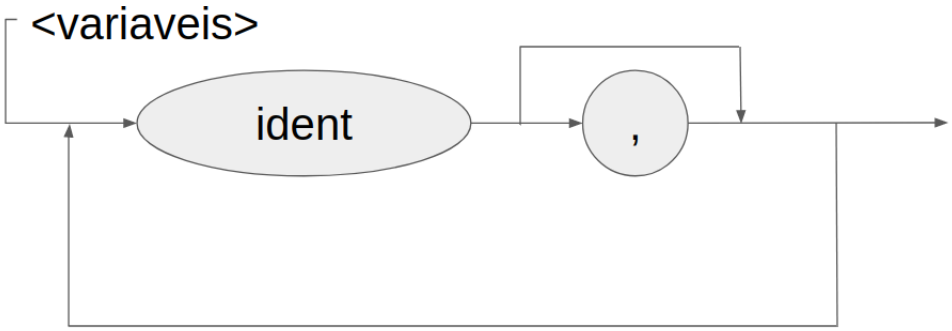


Figura 6: Grafo Sintático do Não Terminal - variáveis

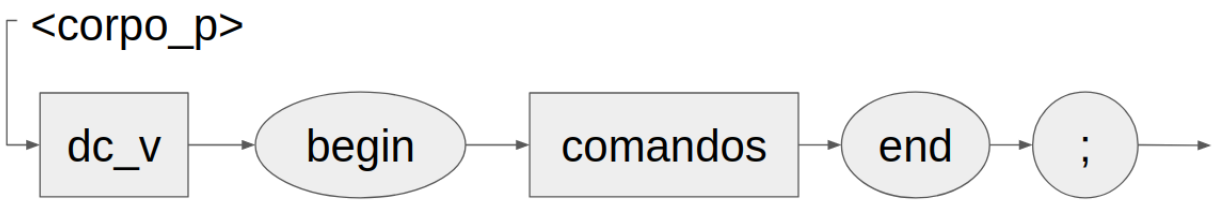


Figura 7: Grafo Sintático do Não Terminal - corpo_p

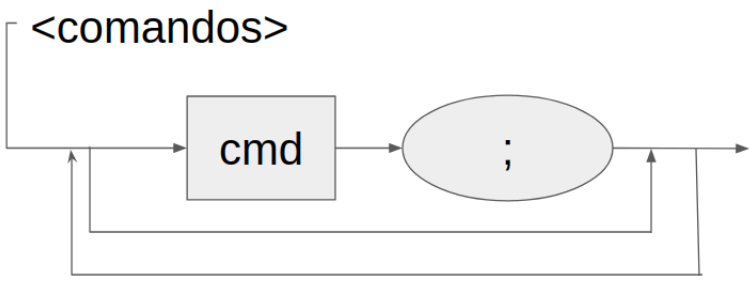


Figura 8: Grafo Sintático do Não Terminal - comandos

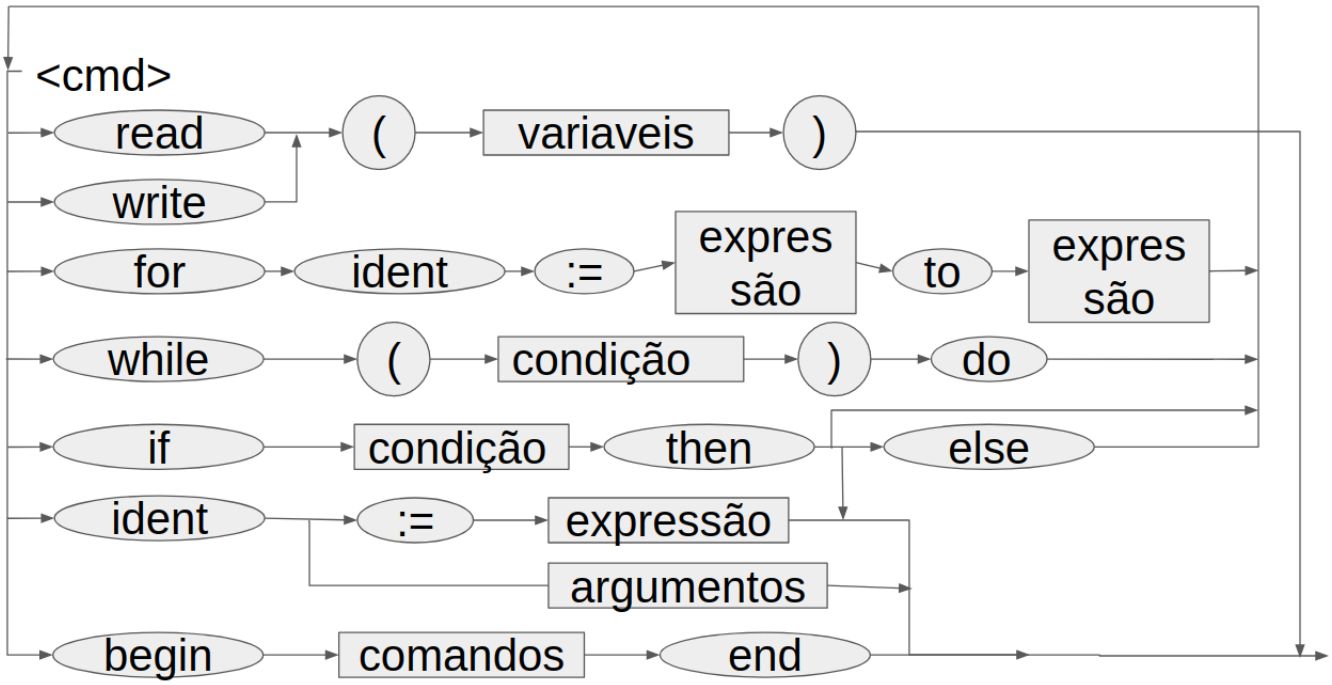


Figura 9: Grafo Sintático do Não Terminal - cmd

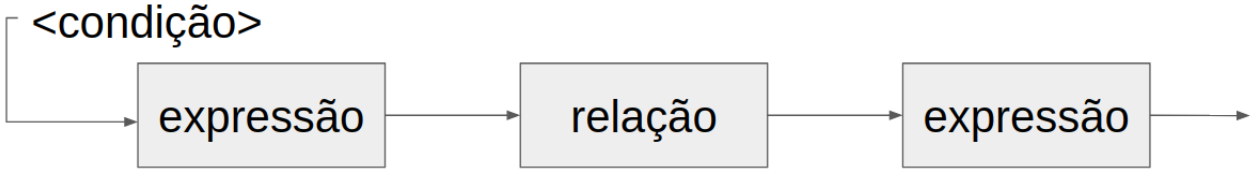


Figura 10: Grafo Sintático do Não Terminal - condição

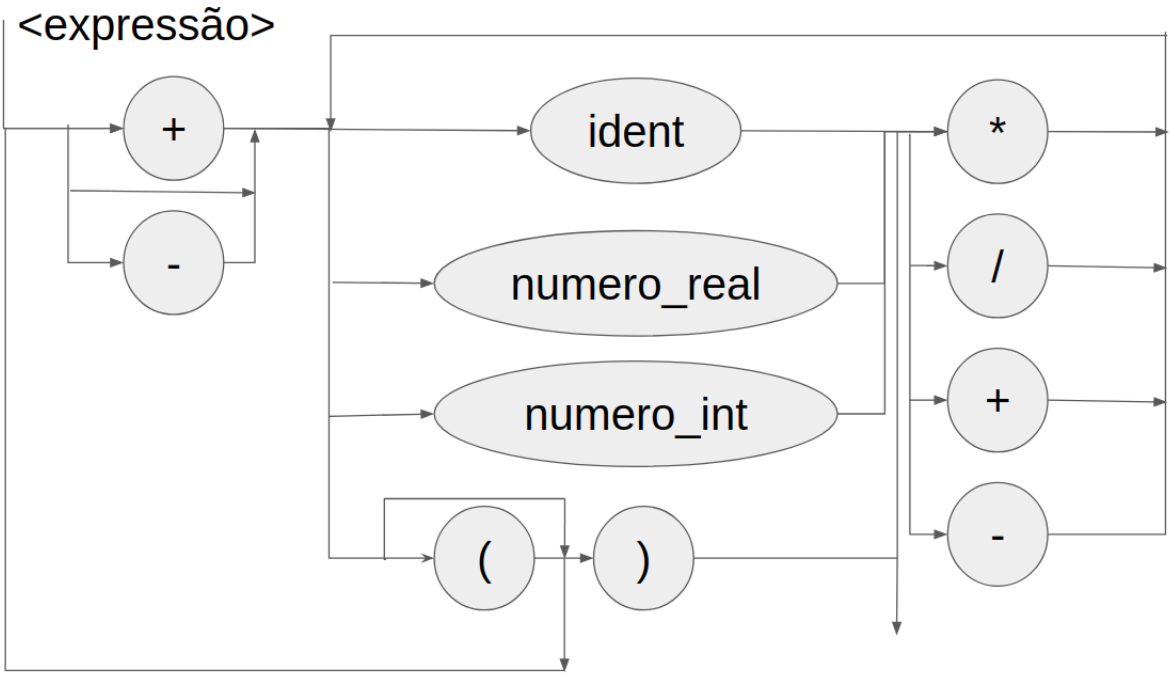


Figura 11: Grafo Sintático do Não Terminal - expressão

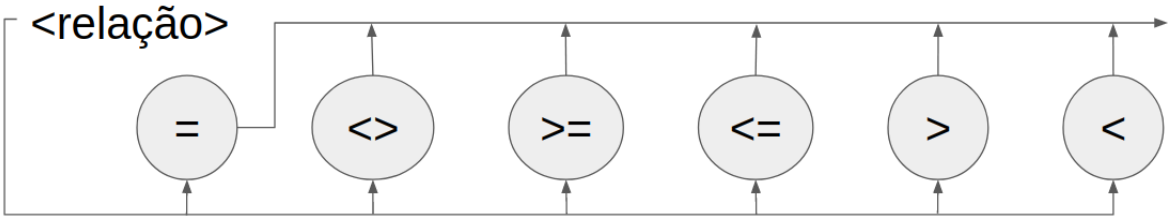


Figura 12: Grafo Sintático do Não Terminal - relação

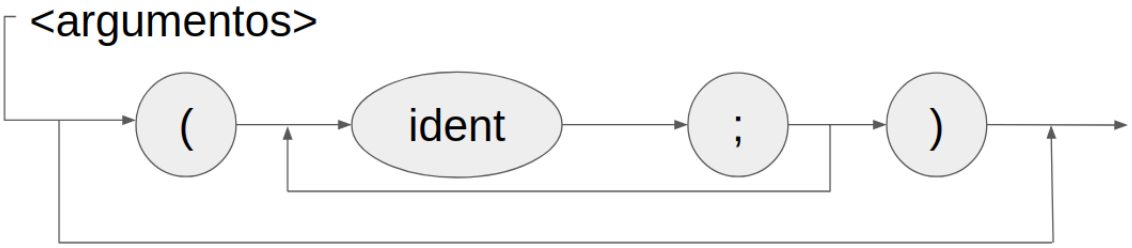


Figura 13: Grafo Sintático do Não Terminal - argumentos

3 Modo Pânico

Modo de pânico , dentre os métodos para se resolver problemas sintáticos, chama a atenção pela sua facilidade de implementação, haja visto que a maioria dos analisadores sintáticos acabam usando-o. Sua capacidade de encontrar um erro ,dá-se pela fato de ao se deparar com um, o analisador sintático apenas relata o erro ao compilador e desvia do problema sintático. Para tal, pulam-se tokens até um, no qual, seja o ponto de sincronização.

Tokens denominados de sincronizadores, são palavras-chave, e ou delimitadores do código, onde é seguro fazer novamente a análise sintática. Portanto é necessário que cada projeto de compilador determine quais serão os tokens de sincronização.

O modo pânico, devido ao seu alto grau de relacionamento com o sintático, e com os Primeiros e Seguidores, foi acoplado ao código SyntacticAnalyzer.pye os possíveis erros são armazenados na função Erros.py. Assim, o modo pânico é definida pela seguinte função:

```
def panic_mode(self, seguidores_imediatos = [], seguidores_pai = [],
    tokens_extras = []) -> bool:
```

3.1 Elementos Primeiros e Seguidores

Todo analisador sintático descendente preditivo recursivo com tratamento de erros sintáticos pelo modo pânico, irá basear sua busca por erros nos tokens de sincronização. Portanto a busca por tais é feita sempre que um token esperando não é encontrado.

Símbolos de sincronização, são inicialmente utilizados nos seguidores de algum procedimento. Assim, é necessário acrescentar os seguidores do procedimento pai, ou seja, os seguidores de quem acionaram o procedimento filho, pois em caso de total erro dentro do procedimento filho, podemos voltar aos seguidores do pai.

Portanto, para elaboração do analisador proposto, devemos observar as relações entre os elementos dessa gramatica, estas relações são as de primeiro *(first)* e seguidor *(follow)*, onde o primeiro é referente ao elemento inicial gerado pela regra e o seguidor é referente ao terminal que segue este determinado token, sendo que este ultimo pode ser o seguidor de um outro não terminal. Há outras relações, como cabeça e ultimo que não abordaremos em nossa implementação, por não se fazer necessário. Estes elementos primeiros e seguidores, foram identificados e utilizados como pode ser visto nos procedimentos de tratamento de erros disponíveis na codificação do ASD, a qual também está disponível no repositório GitHub[1].

4 Instruções para Compilar o Código-Fonte

Para preparar o terreno para a compilação, respeitando a linguagem utilizada, python, os passos essenciais do código fonte do ASD- ScriptySintatico.py -, são basicamente ter a versão básica, python 3 ou superior, para que se possa ter acesso a um interpretador da linguagem utilizada e, assumindo que se tenha acesso a um terminal, basta rodar o código do compilador - compiler.py -para que o mesmo utilize o analisador léxico, o analisador sintático e o armazenamento de erros - Errors.py -na entrada do arquivo txt a ser analisado. Por fim, gera a saida - saida.txt -com as análises feitas pelo léxico e sintático, com o intuito de ser utilizado em futuras implementações. Abaixo temos um exemplo de execução:

```
python3 compiler.py <caminho/nome_do_arquivo_de_entrada.txt>
```

Código 2: Exemplo de Execução Via Terminal

5 Exemplo de Execução

5.1 Exemplo 1

```
1 program teste3;{teste 3 - testando erro de comentario nao fechado,
     checando comportamento com variavel nao declarada}
2 var x: integer;
3 begin
4 read(a);
5 {lendo a variavel e imprimindo
6 .
```

Código 3: Exemplo Um

```
1 Erro lexico na linha 5: Comentario nao fechado
2 Erro sintatico na linha 5: end esperado
```

Código 4: Saída1.tex

6 Anexos e Apêndices

6.1 Linguagem P– Enriquecida com o Comando For

```
<programa> ::= program ident ; <corpo> .
<corpo> ::= <dc> begin <comandos> end
<dc> ::= <dc_c> <dc_v> <dc_p>
<dc_c> ::= const ident = <numero> ; <dc_c> | λ
<dc_v> ::= var <variaveis> : <tipo_var> ; <dc_v> | λ
<tipo_var> ::= real | integer
<variaveis> ::= ident <mais_var>
<mais_var> ::= , <variaveis> | λ
<dc_p> ::= procedure ident <parametros> ; <corpo_p> <dc_p> | λ
<parametros> ::= ( <lista_par> ) | λ
<lista_par> ::= <variaveis> : <tipo_var> <mais_par>
<mais_par> ::= ; <lista_par> | λ
<corpo_p> ::= <dc_loc> begin <comandos> end ;
<dc_loc> ::= <dc_v>
<lista_arg> ::= ( <argumentos> ) | λ
<argumentos> ::= ident <mais_ident>
<mais_ident> ::= ; <argumentos> | λ
<pfalsa> ::= else <cmd> | λ
<comandos> ::= <cmd> ; <comandos> | λ
<cmd> ::= read ( <variaveis> ) |
          write ( <variaveis> ) |
          while ( <condicao> ) do <cmd> |
          for ident := <expressao> to <expressao> <cmd> |
          if <condicao> then <cmd> <pfalsa> |
          ident := <expressao> |
          ident <lista_arg> |
          begin <comandos> end
<condicao> ::= <expressao> <relacao> <expressao>
<relacao> ::= = | <> | >= | <= | > | <
<expressao> ::= <termo> <outros_termos>
<op_un> ::= + | - | λ
<outros_termos> ::= <op_ad> <termo> <outros_termos> | λ
<op_ad> ::= + | -
<termo> ::= <op_un> <fator> <mais_fatores>
<mais_fatores> ::= <op_mul> <fator> <mais_fatores> | λ
<op_mul> ::= * | /
<fator> ::= ident | <numero> | ( <expressao> )
<numero> ::= numero_int | numero_real
```

Código 5: Linguagem P– Enriquecida com o Comando For[23]