



Universidade Federal de Viçosa

*Campus Florestal*

Compiladores (CCF 441)

Pedro Oliveira, Matrícula: 2677

Estela Miranda Batista, Matrícula: 3305

Eduardo Vinícius, Matrícula: 3498

João Marcos Alves Modesto Ramos, Matrícula: 3506

Mariana Souza, Matrícula: 3898

# **amiibo**

Linguagem de Programação Baseada no Jogo Animal Crossing

Florestal, MG

2022

# SUMÁRIO

<b>INTRODUÇÃO</b>	<b>3</b>
<b>DEFINIÇÕES DA LINGUAGEM</b>	<b>4</b>
TIPOS DE DADOS	4
COMANDO	4
OPERADORES	6
PALAVRAS CHAVES E PALAVRAS RESERVADAS	7
GRAMÁTICA DA LINGUAGEM	7
TOKENS	8
PRODUÇÕES	8
Tipo	8
Declaração de Variáveis	8
Blocos	9
Produções de Operação Matemática	10
Produções de Comparação	10
Produções de Comandos	10
<b>ANALISADOR LÉXICO</b>	<b>12</b>
<b>ANALISADOR SINTÁTICO</b>	<b>18</b>
ALTERAÇÃO NA GRAMÁTICA	18
PROGRAMA FONTE COM LINHAS NUMERADAS	20
TABELA DE SÍMBOLOS	22
VERIFICAÇÃO SINTÁTICA	24
<b>ANÁLISE SEMÂNTICA E REPRESENTAÇÕES INTERMEDIÁRIAS</b>	<b>27</b>
ÁRVORE DE SINTAXE ABSTRATA	27
ANÁLISE SEMÂNTICA	31
Variáveis não Declaradas	31
Declaração de Múltiplas Variáveis	32
Verificação de Tipo	33
CÓDIGO DE TRÊS ENDEREÇOS	35
<b>CONSIDERAÇÕES FINAIS</b>	<b>39</b>
<b>REFERÊNCIAS BIBLIOGRÁFICAS</b>	<b>40</b>

## INTRODUÇÃO

Neste trabalho será apresentado a construção de um compilador para uma linguagem de programação. Para a construção do compilador então foi desenvolvida uma linguagem de programação baseada em um jogo da Nintendo, o Animal Crossing. O jogo se baseia na ideia de um novo morador irá se mudar para uma ilha e criar um paraíso para ele, e os demais moradores.

Para as definições da linguagem foram utilizados diversos elementos do jogo, como o nome de personagens que têm interação de construção e reforma da ilha, além das categorias de itens presentes no jogo. Durante a escolha do nome da linguagem, foi pensado em um dos elementos presentes em diversos jogos da produtora, e que tem grande presença no Animal Crossing, sendo esse o **amiibo**, que é uma forma de interagir com os personagens dos jogos, em que eles podem possuir uma forma de card, para colecionadores, ou então de pequenas estatuetas.

Após as definições da linguagem são apresentados os resultados obtidos no trabalho, sendo neste caso a análise sintática da linguagem, em que foi utilizado do gerador de analisador léxico FLEX, e o gerador de analisador sintático Yacc, ambos sendo utilizado em conjunto com a linguagem C.

## DEFINIÇÕES DA LINGUAGEM

### TIPOS DE DADOS

Para os tipos de dados primitivos utilizamos os nomes dos itens presentes no museu do Animal Crossing, assim temos: Bug, Fossil, SeaCreature, Fish, Butterfly, Flower e Tree, como nomes para os tipos de dados. Para os tipos **Float**, **String** e **Boolean**, foi escolhido o padrão para os nomes dos itens do Animal Crossing que contém a mesma letra inicial dos tipos, assim o **Fossil** irá representar o Float, o **SeaCreature** para a String e **Butterfly** para Boolean. Abaixo temos uma representação de como foi implementado os tipos de dados primitivos utilizando os dois itens. Os tipos **Int**, **Char** e **Date**, foi escolhido como padrão para o nome a mesma quantidade de letras, assim para os itens do Animal Crossing, temos: **Bug** para o Int, **Fish** para o Char e **Tree** para o Date. O vetor foi representado pelo último item selecionado que é o **Flower**, assim temos os tipos de dados primitivos com os nomes dos itens do Animal Crossing.

- **Fossil** (Float): Valores de ponto flutuante, no intervalo:  $-3.4 \cdot 10^{38}$  até  $+3.4 \cdot 10^{38}$ .
- **SeaCreature** (String): Valores que representam um conjunto de palavras.
- **Butterfly** (Boolean): Possui dois valores: False e True.
- **Bug** (Int): Valores inteiros, no intervalo: - 2.147.483.648 até 2.147.483.647.
- **Fish** (Char): Caracteres únicos, no intervalo: -128 até 127.
- **Tree** (Date): Valores de datas, no formato: dd-mm-aaaa.
- **Flower** (Vector): Uma lista de valores, que possuem um mesmo tipo, como por exemplo Bug.

### COMANDO



Pavé

Pensando em outros elementos presentes no Animal Crossing, temos que para os comandos foi utilizado o nome de personagens especiais, ou seja, são os personagens que durante o jogo temos interações, como por exemplo trabalhadores das lojas do jogo. Para o formato de



Saharah

Entrada e Saída da linguagem foi utilizado criado um padrão similar à linguagem Python, misturado à linguagem C, usando dos personagens Pavé e Saharah. Abaixo temos uma representação de como seria um código em Python com os três comandos, e como é aplicado na linguagem desenvolvida.

```
nome = input("")
print("Nome: "+nome)

scanf("%s", nome);
printf("Nome: %s", nome);
```

```
SeaCreature nome
Saharah(nome)
Pave("Nome: ", nome)
```



Mabel



Label



Sable

Para os comandos de **If**, **Else If** e **Else**, visto que ambos os três trabalham em conjunto, foi utilizada da referência das três irmãs que trabalham na loja de roupas, sendo elas Mabel, Sable e Label, abaixo temos uma representação de como seria um código em Python com os três comandos, e como é aplicado na linguagem desenvolvida.

```
if(x > 10):
    # código
elif(x > 5):
    # código
else:
    # código
```

```
Mabel(x > 10) { // código}
Sable(x > 5) { // código }
Label { // código }
```



Orville e Wilbur

Para as Estruturas de Repetição foi inserido os comandos **For** e **While**, em que como referência ao jogo foi utilizado dos personagens que trabalham no aeroporto, sendo eles Orville e Wilbur. Abaixo temos uma representação de como seria um código em Python com os três comandos, e como é aplicado na linguagem desenvolvida.

```
for i in range(0, n):
    # código

while True:
    # código
```

```
Orville(0, n) { // código }

Wilbur(True) { // código }
```



Brewster

Um detalhe importante sobre esses comandos é que caso queiramos sair dos mesmos antecipadamente é utilizado do comando break, e no caso da linguagem implementado foi utilizado do personagem **Brewster**, em que basta colocar o mesmo na identificação, que assim como a definição das variáveis basta dar '*n*' para o próximo comando.

Por fim, pensando nas Funções que as linguagens implementam, utilizando do formato apresentado na linguagem Python, em que temos **def nomeFuncao():**, a linguagem utilizou do personagem **Don** para definir o def apresentado. Além disso, o return apresentado nas linguagens também recebeu um nome especial, sendo aqui chamado de **Rover**, abaixo temos uma representação de como seria um código em Python com os três comandos, e como é aplicado na linguagem.



Don



Rover

```
def soma(x, y):
    return x + y
```

```
Don soma(Bug x, Bug y){
    Rover x + y
}
```

## OPERADORES

Para os operadores foi realizada uma implementação similar às linguagens existentes, mantendo assim seus valores lógicos, em que temos os **Operadores Lógicos** **and**, **or** e **not**, e como **Operadores de Comparação** **>**, **<**, **>=**, **<=**, **!=** e **==**, e como **Operador de Atribuição** o símbolo **=**.

Como **Operadores Aritméticos** foi pensado em uma forma básica, também apresentada em outras linguagens, sendo eles os operadores de soma (+), auto-incremento (++), subtração (-), auto-decremento (--), multiplicação (\*), divisão (/), potência (\*\*) e resto da divisão (%).

## PALAVRAS CHAVES E PALAVRAS RESERVADAS

Palavra	Funcionalidade	Chave/Reservada
Bug	Tipo de Dado	Chave e Reservada
Fossil	Tipo de Dado	Chave e Reservada
SeaCreature	Tipo de Dado	Chave e Reservada
Fish	Tipo de Dado	Chave e Reservada
Butterfly	Tipo de Dado	Chave e Reservada
Flower	Tipo de Dado	Chave e Reservada
Tree	Tipo de Dado	Chave e Reservada
Pave	Comando	Chave e Reservada
Saharah	Comando	Chave e Reservada
Mabel	Comando	Chave e Reservada
Label	Comando	Chave e Reservada
Sable	Comando	Chave e Reservada
Orville	Comando	Chave e Reservada
Wilbur	Comando	Chave e Reservada
Brewster	Comando	Chave e Reservada
Don	Comando	Chave e Reservada
Rover	Comando	Chave e Reservada

## GRAMÁTICA DA LINGUAGEM

Durante a definição de uma linguagem de programação um dos principais pontos é a formulação de sua gramática, que irá definir as variáveis, tokens e os padrões de lexemas e tokens, dessa forma, organizando assim questões como a ordem das palavras reservadas da linguagem que podem vir a ser combinadas nas operações.

## TOKENS

% Token Bug Fossil SeaCreature Fish Butterfly Flower Tree

% Token Mabel Sable Label Orville Wilbur Don Pave Saharah Rover Brewster

% Token digito [0-9]

% Token inteiro [-\+]?{digito}+

% Token decimal [-\+]?{digito}+\.{digito}+

% Token opComparacao ">" | "<" | ">=" | "<=" | "!=" | "=="

% Token opLogico "and" | "or" | "not"

% Token opMatematico [-\+\\\*\^\/\%]

% Token booleano "false" | "true"

% Token caractere [a-zA-Z0-9]

% Token letra [a-zA-Z]

% Token qualquerCoisa \. | [^"]

% Token string \"{qualquerCoisa}{qualquerCoisa}+\"

% Token char \"{qualquerCoisa}\"

% Token data {digito}{2}\/{digito}{2}\/{digito}{4}

% Token maisMais [+] {2}

% Token menosMenos [-] {2}

% Token variavel {letra}{caractere}\*

## PRODUÇÕES

### Tipo

tipo → Mabel | Sable | Label | Orville | Wilbur | Don | Pave | Saharah |  
Rover | Brewster

### Declaração de Variáveis

declaracoes → declaracoes declaracao  
| ε  
declaracao → tipo id



## Blocos

$\text{blocos} \rightarrow \{$   
     $\text{declaracoes stmts increment}$   
 $\}$

$\text{stmts} \rightarrow \text{stmts stmt}$   
     $|\epsilon$

$\text{stmt} \rightarrow \text{expressao}$   
     $|\text{stmtC}$   
     $|\text{bloco}$   
     $|\text{expressaoAritmetica}$

$\text{atributo} \rightarrow \text{id}$   
     $|\text{Bug}$   
     $|\text{Fossil}$   
     $|\text{True}$   
     $|\text{False}$   
     $|\text{Bug "-" Bug "-" Bug}$   
     $|\text{expressaoAritmetica}$   
     $|\text{expressaoComparacao}$   
     $|\epsilon$

$\text{listaAtributos} \rightarrow \text{atributo}$   
     $|\text{"[" listaAtributos atributo "]"}"$   
     $|\text{"[" listaAtributos "," atributo "]"}"$

$\text{expressao} \rightarrow \text{declaracao "\&" listaAtributos}$   
     $|\text{id "\&" listaAtributos}$   
     $|\text{chamadaFuncao}$

### **Produções de Operação Matemática**

$\text{expressaoAritmetica} \rightarrow \text{expressaoAritmetica} + \text{termo}$

|  $\text{expressaoAritmetica} - \text{termo}$

| termo

| termo++

| termo--

$\text{termo} \rightarrow \text{termo} * \text{fator}$

|  $\text{termo} / \text{fator}$

|  $\text{termo} ** \text{fator}$

| fator

$\text{fator} \rightarrow \text{Bug}$

| Fossil

|  $(\text{expressaoAritmetica})$

| id

### **Produções de Comparação**

$\text{expressaoComparacao} \rightarrow \text{atributo} > \text{atributo}$

|  $\text{atributo} < \text{atributo}$

|  $\text{atributo and atributo}$

|  $\text{atributo or atributo}$

|  $\text{atributo not atributo}$

|  $\text{atributo} \geq \text{atributo}$

|  $\text{atributo} \leq \text{atributo}$

|  $\text{atributo} == \text{atributo}$

|  $\text{expressaoComparacao}$

### **Produções de Comandos**

$\text{definicaoFuncao} \rightarrow \text{Don id "(" declaracoes ")" bloco}$

$\text{chamadaFuncao} \rightarrow \text{id "(" listaAtributos ")"}$

// Chamada do Break

iniciarBrewster → id

| Bug

| expressao

// Chamada do While

iniciarWilbur → id

| True

| expressaoComparacao

// Chamada do If

iniciarMabel → id

| True

| False

| expressaoComparacao

print → atributo

| print “,” atributo

stmtC → Orville(Bug, Bug) bloco

| Wilbur (iniciarWilbur) bloco

| Mabel (iniciarMabel) bloco (Label bloco)?

| Mabel (iniciarMabel) bloco (Sable (iniciarMabel) bloco Label bloco)?

| Pave (print)

| Saharah (atributo)

## ANALISADOR LÉXICO

Visto todos os comandos, e demais definições da linguagem de programação, a primeira etapa para a execução da mesma é a análise léxica, que neste trabalho foi utilizado do Flex para gerar o mesmo. Para compilar o mesmo, basta utilizar dos comandos abaixo:

```
flex lex.l  
gcc lex.yy.c  
./a.out < nomeArquivoEntrada.txt
```

Neste momento da compilação temos que como as demais etapas de compilação não estão formuladas o analisador léxico irá nos retornar os lexemas presentes na gramática, mostrando assim os mesmos como no exemplo abaixo. É importante citar que caso o lexema não seja reconhecido, nada acontecerá com o mesmo, de forma que também não será exibido o mesmo.

```
// ... código  
Brewster  
// ... código
```

```
// Mensagem Exibida  
“Foi encontrado um BREAK, Brewster”
```

Baseado nisso, foram realizados alguns testes com o analisador léxico, de forma que, foram implementados três códigos, um com comandos condicionais e definição de funções, e outro com comandos de repetição, e por fim, o terceiro código irá mostrar um pouco das variáveis dos tipos strings, e decimal, além de que, em ambos os casos temos a presença de entrada e saída de diversos tipos de dados. O código do primeiro exemplo pode ser visto abaixo, em que a saída para o mesmo está presente nas Figuras 1 e 2, em que na primeira podemos ver o retorno sobre a função para o cálculo de número par, em que foram identificados cada parte definida, como a abertura e fechamento de parênteses, os retorno declarados como Rover, além de que, como temos uma operação de comparação no primeiro if, ele identificou que x é uma variável, % é um operador matemático e == é um operador de comparação, de forma que nas etapas posteriores do trabalho isso possa ser utilizado para formular uma Expressão de Comparação como foi definido anteriormente na seção de produções da linguagem amiibo.

```

Don numeroPar(Bug x){
    Mabel(x%2==0){
        Rover true
    }

    Label{
        Rover false
    }
}

Bug numero
Pave("Digite um número: ")
Saharah(numero)

Mabel(numeroPar(numero) == true){
    Pave("O número é par!")
}

Label{
    Pave("O número é impar!")
}

```

```

Foi encontrado um DEF, Don
Foi encontrado uma variavel. LEXEMA: numeroPar
Foi encontrado uma abertura de parentese, (
Foi encontrado um INT, Bug
Foi encontrado uma variavel. LEXEMA: x
Foi encontrado um fechamento de parentese, )
Foi encontrado uma abertura de chave, {
Foi encontrado um IF, Mabel
Foi encontrado uma abertura de parentese, (
Foi encontrado uma variavel. LEXEMA: x
Foi encontrado um operador matematico. LEXEMA: %
Foi encontrado um valor inteiro, 2
Foi encontrado um operador de comparação. LEXEMA: ==
Foi encontrado um valor inteiro, 0
Foi encontrado um fechamento de parentese, )
Foi encontrado uma abertura de chave, {
Foi encontrado um RETURN, Rover
Foi encontrado um valor booleano, true
Foi encontrado um fechamento de chave, }
Foi encontrado um ELSE, Label
Foi encontrado uma abertura de chave, {
Foi encontrado um RETURN, Rover
Foi encontrado um valor booleano, false
Foi encontrado um fechamento de chave, }
Foi encontrado um fechamento de chave, }

```

Figura 1. Execução Exemplo 1, Definições da Função

Para a segunda parte do código do primeiro exemplo, apresentado na Figura 2, temos que uma importante observação é que durante a saída dos dados, no print, chamado aqui de Pave, visto que estamos na análise léxica do projeto, e não há identificação de dados posteriores ao visto naquele momento da análise, caso ele contenha “ ”, isso será identificado como uma string literal. No mais todas as outras identificações seguiram o previsto anteriormente.

```

Foi encontrado um INT, Bug
Foi encontrado uma variavel. LEXEMA: numero
Foi encontrado um PRINT, Pave
Foi encontrado uma abertura de parentese, (
Foi encontrado uma string literal, "Digite um número: "
Foi encontrado um fechamento de parentese, )
Foi encontrado um SCANF, Saharah
Foi encontrado uma abertura de parentese, (
Foi encontrado uma variavel. LEXEMA: numero
Foi encontrado um fechamento de parentese, )
Foi encontrado um IF, Mabel
Foi encontrado uma abertura de parentese, (
Foi encontrado uma variavel. LEXEMA: numeroPar
Foi encontrado uma abertura de parentese, (
Foi encontrado uma variavel. LEXEMA: numero
Foi encontrado um fechamento de parentese, )
Foi encontrado um operador de comparação. LEXEMA: ==
Foi encontrado um valor booleano, true
Foi encontrado um fechamento de parentese, )
Foi encontrado uma abertura de chave, {
Foi encontrado um PRINT, Pave
Foi encontrado uma abertura de parentese, (
Foi encontrado uma string literal, "O número é par!"
Foi encontrado um fechamento de parentese, )
Foi encontrado um fechamento de chave, }
Foi encontrado um ELSE, Label
Foi encontrado uma abertura de chave, {
Foi encontrado um PRINT, Pave
Foi encontrado uma abertura de parentese, (
Foi encontrado uma string literal, "O número é impar!"
Foi encontrado um fechamento de parentese, )
Foi encontrado um fechamento de chave, }

```

Figura 2. Execução Exemplo 1, Declarações, Entrada e Saída de Dados

O código do segundo exemplo pode ser visto abaixo, em que a saída para o mesmo está apresentado na Figura 3, em que aqui temos a inserção de novas estruturas previstas para a linguagem, começando pelas estruturas de repetição for e while, que aqui foram identificadas como Orville e Wilbur. Além disso, temos a presença de operadores matemáticos como soma e ++. Por fim, um importante identificador também pode ser visto na saída do exemplo, que é o comando break, chamado aqui de Brewster.

```

Bug numero
Pave("Digite um número: ")
Saharah(numero)

// Variavel para armazenar a soma da sequencia ate o numero
Bug somaSequencia = 0
Bug i = 1

Orville(1, numero){
    somaSequencia = somaSequencia + i
    i++
}

Pave("Soma da Sequência: ", somaSequencia)

// Variavel para armazenar a soma de dois em dois
Bug somaIntervalo = 0
Bug i = 1

Wilbur(true){

```

```

Mabel(i == (numero - 1)){
    Brewster
}

somaIntervalo = somaIntervalo + i
i = i + 2
}

Pave("Soma do Intervalo de Dois em Dois: ", somaIntervalo)

```

```

Foi encontrado um INT, Bug
Foi encontrado uma variavel. LEXEMA: numero
Foi encontrado um PRINT, Pave
Foi encontrado uma abertura de parentese, (
Foi encontrado uma string literal, "Digite um número: "
Foi encontrado um fechamento de parentese, )
Foi encontrado um SCANF, Saharah
Foi encontrado uma abertura de parentese, (
Foi encontrado uma variavel. LEXEMA: numero
Foi encontrado um fechamento de parentese, )
Foi encontrado um comentario. LEXEMA: // Variavel para armazenar a soma da sequencia ate o numero
Foi encontrado um INT, Bug
Foi encontrado uma variavel. LEXEMA: somaSequencia
Foi encontrado um operador de atribuição, =
Foi encontrado um valor inteiro, 0
Foi encontrado um INT, Bug
Foi encontrado uma variavel. LEXEMA: i
Foi encontrado um operador de atribuição, =
Foi encontrado um valor inteiro, 1
Foi encontrado um FOR, Orville
Foi encontrado uma abertura de parentese, (
Foi encontrado um valor inteiro, 1
Foi encontrado um separador, ,
Foi encontrado uma variavel. LEXEMA: numero
Foi encontrado um fechamento de parentese, )
Foi encontrado uma abertura de chave, {
Foi encontrado uma variavel. LEXEMA: somaSequencia
Foi encontrado um operador de atribuição, =
Foi encontrado uma variavel. LEXEMA: somaSequencia
Foi encontrado um operador matematico. LEXEMA: +
Foi encontrado uma variavel. LEXEMA: i
Foi encontrado uma variavel. LEXEMA: i
Foi encontrado um operador matematico. LEXEMA: ++
Foi encontrado um fechamento de chave, }
Foi encontrado um PRINT, Pave
Foi encontrado uma abertura de parentese, (
Foi encontrado uma string literal, "Soma da Sequência: "
Foi encontrado um separador, ,
Foi encontrado uma variavel. LEXEMA: somaSequencia

```

```

Foi encontrado um fechamento de parentese, )
Foi encontrado um comentario. LEXEMA: // Variavel para armazenar a soma de dois em dois
Foi encontrado um INT, Bug
Foi encontrado uma variavel. LEXEMA: somaIntervalo
Foi encontrado um operador de atribuição, =
Foi encontrado um valor inteiro, 0
Foi encontrado um INT, Bug
Foi encontrado uma variavel. LEXEMA: i
Foi encontrado um operador de atribuição, =
Foi encontrado um valor inteiro, 1
Foi encontrado um WHILE, Wilbur
Foi encontrado uma abertura de parentese, (
Foi encontrado um valor booleano, true
Foi encontrado um fechamento de parentese, )
Foi encontrado uma abertura de chave, {
Foi encontrado um IF, Mabel
Foi encontrado uma abertura de parentese, (
Foi encontrado uma variavel. LEXEMA: i
Foi encontrado um operador de comparação. LEXEMA: ==
Foi encontrado uma variavel. LEXEMA: numero
Foi encontrado um valor inteiro, -1
Foi encontrado um fechamento de parentese, )
Foi encontrado uma abertura de chave, {
Foi encontrado um BREAK, Brewster
Foi encontrado um fechamento de chave, }
Foi encontrado uma variavel. LEXEMA: somaIntervalo
Foi encontrado um operador de atribuição, =
Foi encontrado uma variavel. LEXEMA: somaIntervalo
Foi encontrado um operador matematico. LEXEMA: +
Foi encontrado uma variavel. LEXEMA: i
Foi encontrado uma variavel. LEXEMA: i
Foi encontrado um operador de atribuição, =
Foi encontrado uma variavel. LEXEMA: i
Foi encontrado um operador matematico. LEXEMA: +
Foi encontrado um valor inteiro, 2
Foi encontrado um fechamento de chave, }
Foi encontrado um PRINT, Pave
Foi encontrado uma abertura de parentese, (
Foi encontrado uma string literal, "Soma do Intervalo de Dois em Dois: "

```

```

Foi encontrado um separador, ,
Foi encontrado uma variavel. LEXEMA: somaIntervalo
Foi encontrado um fechamento de parentese, )

```

Figura 3. Execução Exemplo 2

O código do segundo exemplo pode ser visto abaixo, em que a saída para o mesmo está apresentado na Figura 3, em que visto que nesse exemplo temos uma formulação mais simples, com algumas declarações de variáveis, e entrada e saída desses dados, de forma que temos a presença de operadores matemáticos de divisão e potência.

```
SeaCreature nome
Pave("Digite seu nome: ")
Saharah(SeaCreature)

Tree nascimento
Pave("Digite sua data de nascimento: ")
Saharah(nascimento)

Fossil peso
Pave("Digite seu peso: ")
Saharah(peso)

Fossil altura
Pave("Digite sua altura: ")
Saharah(altura)

Fossil imc = peso / (altura**2)

Pave("A pessoa ", nome, " nasceu em ", nascimento, " e um IMC de ", imc)
```

```
Foi encontrado um STRING, SeaCreature
Foi encontrado uma variavel. LEXEMA: nome
Foi encontrado um PRINT, Pave
Foi encontrado uma abertura de parentese, (
Foi encontrado uma string literal, "Digite seu nome: "
Foi encontrado um fechamento de parentese, )
Foi encontrado um SCANF, Saharah
Foi encontrado uma abertura de parentese, (
Foi encontrado um STRING, SeaCreature
Foi encontrado um fechamento de parentese, )
Foi encontrado um DATETIME, Tree
Foi encontrado uma variavel. LEXEMA: nascimento
Foi encontrado um PRINT, Pave
Foi encontrado uma abertura de parentese, (
Foi encontrado uma string literal, "Digite sua data de nascimento: "
Foi encontrado um fechamento de parentese, )
Foi encontrado um SCANF, Saharah
Foi encontrado uma abertura de parentese, (
Foi encontrado uma variavel. LEXEMA: nascimento
Foi encontrado um fechamento de parentese, )
Foi encontrado um FLOAT, Fossil
Foi encontrado uma variavel. LEXEMA: peso
Foi encontrado um PRINT, Pave
Foi encontrado uma abertura de parentese, (
Foi encontrado uma string literal, "Digite seu peso: "
Foi encontrado um fechamento de parentese, )
Foi encontrado um SCANF, Saharah
Foi encontrado uma abertura de parentese, (
Foi encontrado uma variavel. LEXEMA: peso
Foi encontrado um fechamento de parentese, )
Foi encontrado um FLOAT, Fossil
Foi encontrado uma variavel. LEXEMA: altura
Foi encontrado um PRINT, Pave
Foi encontrado uma abertura de parentese, (
Foi encontrado uma string literal, "Digite sua altura: "
Foi encontrado um fechamento de parentese, )
Foi encontrado um SCANF, Saharah
Foi encontrado uma abertura de parentese, (
Foi encontrado uma variavel. LEXEMA: altura
```



```
Foi encontrado um fechamento de parentese, )
Foi encontrado um FLOAT, Fossil
Foi encontrado uma variavel. LEXEMA: imc
Foi encontrado um operador de atribuição, =
Foi encontrado uma variavel. LEXEMA: peso
Foi encontrado um operador matematico. LEXEMA: /
Foi encontrado uma abertura de parentese, (
Foi encontrado uma variavel. LEXEMA: altura
Foi encontrado um operador matematico. LEXEMA: **
Foi encontrado um valor inteiro, 2
Foi encontrado um fechamento de parentese, )
Foi encontrado um PRINT, Pave
Foi encontrado uma abertura de parentese, (
Foi encontrado uma string literal, "A pessoa "
Foi encontrado um separador, ,
Foi encontrado uma variavel. LEXEMA: nome
Foi encontrado um separador, ,
Foi encontrado uma string literal, " nasceu em "
Foi encontrado um separador, ,
Foi encontrado uma variavel. LEXEMA: nascimento
Foi encontrado um separador, ,
Foi encontrado uma string literal, " e um IMC de "
Foi encontrado um separador, ,
Foi encontrado uma variavel. LEXEMA: imc
Foi encontrado um fechamento de parentese, )
```

Figura 4. Execução Exemplo 3

## ANALISADOR SINTÁTICO

Seguindo na implementação da linguagem apresentada, para a segunda parte do trabalho foi definido que deveria ser criado o analisador sintático, usando da ferramenta Yacc em conjunto o LEX definido anteriormente no analisador léxico, além do uso da linguagem C em ambos. As tarefas para o analisador sintático então são que ao usar o executável do compilador para a linguagem definida deve ser impresso na saída o **Programa Fonte com as Linhas Numeradas**, o **Conteúdo da Tabela de Símbolos** e se o **Programa está Sintaticamente Correto ou Sintaticamente Incorreto**.

É importante citar que algumas alterações foram feitas na formatação textual, e na gramática, em que a mesma será apresentada nas subseções a seguir. Já sobre a formatação textual do código, temos que a forma de entrada de dados passa a ser um valor recebido na variável, para se tornar mais próximo à linguagem Python. Abaixo podemos ver como era feita a entrada dos dados, e como passou a ser feita agora.

```
SeaCreature nome  
Saharah(nome)
```

```
SeaCreature nome  
nome = Saharah()
```

Para executar o que será apresentado do trabalho nas próximas seções, basta acessar a pasta **gramatica** no arquivo fonte, em que utilizando do **make** da linguagem C, e com as extensões do Yacc instaladas no sistema operacional Linux, basta utilizar os comandos abaixo.

```
make all  
make run < nomeArquivo.txt
```

### ALTERAÇÃO NA GRAMÁTICA

Como citado anteriormente, foram realizadas alterações na gramática utilizada pela linguagem amiibo, de forma a melhor se adequarem ao Yacc, e as definições de produções, visto que muitas delas vieram a serem executáveis dentro do programa Yacc. Primeiramente, como adição, foi feita a implementação da precedência dos operadores matemáticos, como soma e subtração, sendo isso apresentado no código abaixo, em que a precedência dos operadores ficou definida como *left* para soma, subtração, divisão e multiplicação, *right* para potência, incremento e decremento na declaração das variáveis.

```
%left MATEMATICO_SOMA MATEMATICO_SUBTRACAO
%left MATEMATICO_DIVISAO MATEMATICO_MULTIPLICACAO
%right MATEMATICO_POW MATEMATICO_INCREMENTO MATEMATICO_DECREMENTO
```

Além disso, os nomes apresentados na seção de produções da primeira parte do trabalho sofreu alterações em relação ao nome de seus tokens, fazendo com muitas das produções fossem alteradas, como é o caso do **stmC**, em que ele passou a se chamar **comando**, e além disso, ao invés de definir em seu próprio corpo como cada comando funciona, foram criadas produções para cada um dos comandos, que são chamadas dentro da produção de **comando**, isso pode ser observado no código abaixo, em que foi exemplificado a chamada de **comando**, e do comando **condicional**.

```
comando:
    condicional
    | repeticao
    | imprimir
    | input
    | retornar
    | BREAK_TOKEN

//...

condicional:
    if
    | else
    | elseIF

if:
    IF_TOKEN ABRIR_PARENTESES_TOKEN relacional
    FECHAR_PARENTESES_TOKEN corpo

else:
    ELSE_TOKEN corpo

elseIF:
    ELSE_IF_TOKEN ABRIR_PARENTESES_TOKEN relacional
    FECHAR_PARENTESES_TOKEN corpo
```

Outras produções que passaram por mudanças de formatação ou mudança de nomes, podem ser verificadas no arquivo **translate.y** na pasta **gramatica**, enviado em conjunto no trabalho, além de serem apresentados os tokens gerados, e algumas outras definições da gramática nas próximas subseções.

## PROGRAMA FONTE COM LINHAS NUMERADAS

Como foi previsto na documentação do trabalho, foi gerado uma forma de imprimir o código fonte analisado, constando o número de cada linha, sendo que isso foi feito no arquivo FLEX, em que foi utilizado a definição do LEX para a contagem de linhas, utilizando a variável no trecho de código da linguagem C chamada **yylineno**, em que cada definição de tokens da linguagem amiibo é somado mais um, como exemplificado abaixo para a declaração de uma variável do tipo **Bug**, em que foi deixado em negrito a parte referente à soma da linha do código.

```
"Bug" {if(yylineno==1) printf("%d\t",yylineno++); printf(RED"%s\n",yytext); return INT_TYPE;}
```

Além disso, foi feita também uma implementação de cores no código, em que as palavras chaves da linguagem, como tokens de declarações de tipos e comandos receberam cores de acordo com seus respectivos personagens, que foram apresentados na seção anterior. Para a declaração das cores, elas foram definidas inicialmente no trecho de código da linguagem C no arquivo FLEX, como pode ser visto abaixo.

```
%{  
    void yyerror(char *c);  
    #include "y.tab.h" //Contém as definições do token, o YACC gera  
  
    #define RED          "\x1b[31m"  
    #define BLACK        "\033[38;5;0m"  
    #define HIGHPINK     "\033[38;5;13m"  
    #define ROSERED      "\033[38;5;160m"  
    #define GREEN        "\x1b[32m"  
    #define YELLOW       "\x1b[33m"  
    #define BLUE         "\x1b[34m"  
    #define DEEPBLUE     "\033[38;5;21m"  
    #define AQUABLU     "\033[38;5;45m"  
    #define LEAFGREEN    "\033[38;5;40m"  
    #define HIGHBLUE     "\033[38;5;12m"  
    #define GRAY         "\033[38;5;8m"  
    #define ORANGE       "\033[38;5;94m"  
    #define PINK         "\033[38;5;206m"  
    #define MAGENTA      "\x1b[35m"  
    #define CYAN         "\x1b[36m"  
    #define RESET        "\x1b[0m"  
}%}
```

A partir disso, em cada local que necessitava da chamada de uma cor específica elas eram inseridas no print, como mostrado em negrito no exemplo abaixo.

```
"Bug" {if(yylineno==1) printf("%d\t",yylineno++); printf(RED"%s\n",yytext); return INT_TYPE;}
```

Para exemplificar o uso da impressão do programa fonte com as linhas numeradas, foi utilizado o seguinte programa abaixo, em que na Figura 5 podemos ver os resultados obtidos após a sua execução.

```
Don numeroPar(Bug x){  
  Mabel(x%2==0){  
    Rover true  
  }  
  
  Label{  
    Rover false  
  }  
}  
  
Bug numero  
Pave("Digite um número: ")  
Saharah(numero)  
  
Mabel(numeroPar(numero) == true){  
  Pave("O número é par!")  
}  
  
Label{  
  Pave("O número é impar!")  
}
```



```
stardotwav@stardotwav: /mnt/c/Users/teLal/OneDrive/Documents/github/TPCompiladores/TP2/gramaticas  
./TP2  
1  Don numeroPar ( Bug x ) {  
3  Mabel ( x % 2 == 0 ) {  
4  Rover true  
5  }  
6  
7  Label {  
8  Rover false  
9  }  
10 }  
11  
12 Bug numero  
13 Pave ( "Digite um número: " )  
14 numero = Saharah ( )  
15  
16 Mabel ( numeroPar ( numero ) == true ) {  
17 Pave ( "O número é par!" )  
18 }  
19  
20 Label {  
21 Pave ( "O número é impar!" )  
22 }  
  
Código compilado com sucesso!
```

Figura 5. Exemplo de Amostragem do Programa Fonte com Linhas Numeradas

Outro exemplo possível, é caso na linha 14 por exemplo, inserirmos a antiga formatação de entrada de dados, em que nosso compilador apontará o erro, como mostrado na Figura 6.

```
stardotwav@stardotwav:/mnt/c/Users/telal/OneDrive/Documents/github/TPCompiladores/TP2/gramatica$ ./TP2
1      Don numeroPar ( Bug x ) {
3      Mabel ( x % 2 == 0 ) {
4      Rover true
5      }
6
7      Label {
8      Rover false
9      }
10     }
11
12     Bug numero
13     Pave ( "Digite um número: " )
14     Saharah ( numero
Erro ao compilar o código!
Erro na linha 14
Tipo do erro: syntax error
```

Figura 6. Exemplo de Amostragem de Programa Fonte com Linhas Numeradas com Erro

## TABELA DE SÍMBOLOS

Para a implementação da tabela de símbolos foi utilizado o conceito de listas encadeadas, em que podemos dividir em duas partes, sendo a primeira delas a tabela de um determinado escopo, e a formação dos escopos. Olhando inicialmente para a tabela em si, temos que a **Tabela de Símbolos** é uma estrutura que contém **Símbolos** apontando para outros símbolos, além de um contador de quantos símbolos estão presentes naquela tabela. Cada símbolo possui um Endereço de Memória, um Lexema, Tipo, Tipo de Token, a linha correspondente à aquele símbolo, além do ponteiro para o próximo símbolo, que pode vir a ser um dos tipos definidos pela linguagem, como Bug ou Fish, ou ser um Token, como o If, ou até mesmo ser um Operador, como os operadores relacionais e matemáticos. É importante citar que para esses dois conceitos foram implementados seis métodos, sendo eles o de **criaSimbolo**, **imprimeSimbolo**, **criaTabelaDeSimbolo**, **adicionaSimboloNaTabela**, **pegarSimbolo**, **simboloExiste** e **imprimirTabeladeSimbolos**. Abaixo podemos ver o código na linguagem C que implementa a estrutura dos conceitos apresentados da tabela de símbolos e símbolos.

```
// Ponteiro para um símbolo
typedef struct Simbolo* SimboloPonteiro;
typedef struct Simbolo {
    // Endereço do símbolo
    unsigned int enderecoMemoria; //0 - 255
    // Lexema do símbolo
    char* lexema;
```

```

// Tipo do símbolo
TipoPrimitivo tipo;
TipoToken tipoToken;
int linha;
// Ponteiro para o próximo símbolo da lista
SimboloPonteiro prox;
} Simbolo;

// Ponteiro para a tabela de símbolos
typedef struct TabelaDeSimbolos {
    // Ponteiro para a célula cabeça
    SimboloPonteiro prox;
    int contador;
} TabelaDeSimbolos;

```

Olhando agora para um **Escopo**, temos que ele também foi implementado utilizando do conceito de listas encadeadas, porém, aqui ele não irá apontar para o próximo escopo, ele irá apontar para o escopo acima do que está sendo criado, ou seja, o escopo pai. Para esse conceito então foi implementado três funções, sendo elas **criarEscopo**, **procuraTokenPeloLexema** e **procuraTokenPeloLexemaEscopoAtual**. Abaixo podemos ver o código na linguagem C que implementa a estrutura dos conceitos apresentados da tabela de símbolos e símbolos.

```

// Ponteiro para um escopo
typedef struct Escopo* EscopoPonteiro;

// Representação de um escopo do compilador
typedef struct Escopo {
    // Escopo anterior
    EscopoPonteiro prev;
    // Tabela do escopo
    TabelaDeSimbolos* tabela;
} Escopo;

```

É importante citar que, durante a execução do arquivo Yacc, visto que todos os comandos que geram novos escopos da linguagem lidam com abertura e fechamento de chaves, ao executar o trecho de corpo de abertura das chaves é gerado um novo escopo, que fica referenciado como **escopoAtual**. Abaixo podemos ver o código que mostra a utilização do mesmo, visto que há uma exemplificação do uso da tabela de símbolos, da Figura 7, que mostra o código citado na seção anterior, com a divisão de variáveis por escopo, em um dos trechos do código.

### corpo:

```
ABRIR_CHAVE_TOKEN {escopoAtual = aprofundarEscopo(escopoAtual);} expr  
FECHAR_CHAVE_TOKEN {escopoAtual = voltarEscopo(escopoAtual);}  
| ABRIR_CHAVE_TOKEN FECHAR_CHAVE_TOKEN
```

```
stardotwav@stardotwav:/mnt/c/Users/telal/OneDrive/Documents/github/TPCompiladores/TP2/gramatica  
1 Don numeroPar ( Bug x ) {  
3 Mabel ( x % 2 == 0 ) {  
4 Rover true  
5 }  
-----  
Tabela de Símbolos  
-----  
Num. Símbolos = 1  
-----  
Endereco: 0, Lexema: true, Tipo: Butterfly, Tipo token: Constante, Linha: 4  
-----  
6  
7 Label {  
8 Rover false  
9 }  
-----  
Tabela de Símbolos  
-----  
Num. Símbolos = 1  
-----  
Endereco: 0, Lexema: false, Tipo: Butterfly, Tipo token: Constante, Linha: 8  
-----  
10 }
```

Figura 7. Exemplo de Tabela de Símbolos por Escopo

Para exemplificar o uso da tabela de símbolos, usando do exemplo apresentado anteriormente na etapa do programa fonte com linhas numeradas, temos que foi exibido na saída o resultado apresentado na Figura 8.

```
Tabela de Símbolos Global:  
-----  
Tabela de Símbolos  
-----  
Num. Símbolos = 8  
-----  
Endereco: 0, Lexema: numeroPar, Tipo: None, Tipo token: Funcao, Linha: 2  
Endereco: 0, Lexema: x, Tipo: Bug, Tipo token: Variavel, Linha: 2  
Endereco: 0, Lexema: true, Tipo: Butterfly, Tipo token: Constante, Linha: 4  
Endereco: 0, Lexema: false, Tipo: Butterfly, Tipo token: Constante, Linha: 8  
Endereco: 0, Lexema: numero, Tipo: Bug, Tipo token: Variavel, Linha: 12  
Endereco: 0, Lexema: "Digite um número: ", Tipo: SeaCreature, Tipo token: Constante, Linha: 13  
Endereco: 0, Lexema: "O número é par!", Tipo: SeaCreature, Tipo token: Constante, Linha: 17  
Endereco: 0, Lexema: "O número é impar!", Tipo: SeaCreature, Tipo token: Constante, Linha: 21  
-----
```

Figura 8. Exemplo de Tabela de Símbolos Geral

## VERIFICAÇÃO SINTÁTICA

Para finalizar essa parte do trabalho, temos que seguindo a gramática da linguagem apresentada anteriormente, foi gerado um arquivo no formato Yacc que segue tais definições, de forma que inicialmente foi definido a tabela de símbolos apresentada anteriormente, sendo isso feito no trecho de código da linguagem C. Após isso foi definido os



tokens, como começando pelos tipos de variáveis, seguido de operadores relacionais, operadores matemáticos, operador desconhecido, if, else if, else, for, while, return, break, abrir e fechar parênteses, abrir e fechar chaves, abrir e fechar colchetes, atribuição de valores, vírgula, End of File, e os tipos da linguagem, além da precedência dos operadores matemáticos, como apresentado no código abaixo.

```
%token INT_TYPE FLOAT_TYPE STRING_TYPE BOOLEAN_TYPE VECTOR_TYPE CHAR_TYPE
DATE_TYPE

%token RELACIONAL_IGUALDADE RELACIONAL_NEGACAO RELACIONAL_MAIORQUE
RELACIONAL_MENORQUE RELACIONAL_MAIORIGUAL RELACIONAL_MENORIGUAL
RELACIONAL_OR RELACIONAL_AND

%token MATEMATICO_SOMA MATEMATICO_SUBTRACAO MATEMATICO_MULTIPLICACAO
MATEMATICO_DIVISAO MATEMATICO_MOD MATEMATICO_POW MATEMATICO_INCREMENTO
MATEMATICO_DECREMENTO

%token DESCONHECIDO
%token IF_TOKEN ELSE_IF_TOKEN ELSE_TOKEN
%token FOR_TOKEN WHILE_TOKEN RETURN_TOKEN BREAK_TOKEN
%token CREATE_FUNC_TOKEN PRINT_TOKEN SCANF_TOKEN
%token ABRIR_PARENTESES_TOKEN FECHAR_PARENTESES_TOKEN
%token ABRIR_CHAVE_TOKEN FECHAR_CHAVE_TOKEN
%token ABRIR_COLCHETE_TOKEN FECHAR_COLCHETE_TOKEN
%token ATRIB_TOKEN VIRGULA_TOKEN VAR_TOKEN COMMENT_TOKEN
%token EOL_TOKEN
%token INT FLOAT DATE STRING CHAR BOOLEAN VECTOR

%left MATEMATICO_SOMA MATEMATICO_SUBTRACAO
%left MATEMATICO_DIVISAO MATEMATICO_MULTIPLICACAO
%right MATEMATICO_POW MATEMATICO_INCREMENTO MATEMATICO_DECREMENTO
```

A partir dessas definições então foi gerado a gramática na linguagem do Yacc, usando assim como anteriormente, da gramática, e de suas alterações apresentadas anteriormente. Abaixo podemos ver um exemplo de definição do Yacc para a gramática, em que será apresentado a definição do que é um comando, e logo em seguida, o que é um definido como sendo um comando de repetição.

```
comando:
    condicional
    | repeticao
    | imprimir
    | input
    | retornar
    | BREAK_TOKEN
```

//...

**repeticao:**

```
    FOR_TOKEN ABRIR_PARENTESES_TOKEN aritmetica FECHAR_PARENTESES_TOKEN  
corpo  
    | FOR_TOKEN ABRIR_PARENTESES_TOKEN aritmetica VIRGULA_TOKEN  
aritmetica FECHAR_PARENTESES_TOKEN corpo  
    | WHILE_TOKEN ABRIR_PARENTESES_TOKEN relacional  
FECHAR_PARENTESES_TOKEN corpo
```

## ANÁLISE SEMÂNTICA E REPRESENTAÇÕES INTERMEDIÁRIAS

Como última parte integrante do trabalho prático, foi realizada a implementação da análise semântica da linguagem, em paralelo com a análise sintática, além da inserção das representações intermediárias. Inicialmente é importante citar que como o grupo optou por realizar a implementação da **Árvore de Sintaxe Abstrata**, ela foi utilizada durante a **Análise Semântica**, dessa forma, nas seções posteriores, iremos apresentar a implementação inicial da árvore de sintaxe abstrata, depois, a utilização e as alterações feitas na mesma para a análise semântica, e por fim, a apresentação da segunda representações intermediária adotada: o **Código de Três Endereços**.

Porém, antes de falarmos sobre essas implementações, é importante citar que foram feitas alterações no corpo dos códigos da linguagem, em que, anteriormente, a linguagem tinha seu corpo de código similar à linguagem Python, que não necessitava de uma função declarada como **main**, e com suas funções externas ao main, e nesse caso, agora passaram a serem internas ao corpo do main. É importante citar que tal alteração foi feita de acordo com a necessidade de se ter um local inicial do código, para assim fazer a declaração da árvore de sintaxe abstrata. Abaixo podemos ver como o código era formulado, e como passou a ser com essa alteração.

```
Don soma(Bug x, Bug y){
    Rover x + y
}

Bug a
a = Saharah()
b = Saharah()

Pave(soma(a,b))
```

```
main{
    Don soma(Bug x, Bug y){
        Rover x + y
    }

    Bug a, b
    a = Saharah()
    b = Saharah()

    Pave(soma(a,b))
}
```

### ÁRVORE DE SINTAXE ABSTRATA

Para a árvore de sintaxe abstrata foi criado um novo arquivo, contendo sua estrutura, de forma que, essa árvore, assim como outras implementadas na linguagem C possuem um nó de filho à esquerda, e um nó de filho à direita, além de claro, um char que identifica nessa caso específico o token presente naquele nó. No código abaixo podemos então verificar

como foi definido o tipo de cada nó, além do cabeçalho das funções que essa árvore possui, de forma a criar e a imprimir a árvore no terminal para o usuário.

```
// Representação de um nó de árvore de sintaxe abstrata
struct No {
    // Escopo anterior
    struct No *filhoEsquerdo;
    // Tabela do escopo
    struct No *filhoDireito;
    char *token;
};

struct No* criaNo(struct No *filhoEsquerdo, struct No *filhoDireito, char
*token);
void imprimirArvore(struct No* arvore);
void imprimirEmOrdem(struct No *arvore);

program:
    MAIN_TOKEN corpo { $$ .np = criaNo(NULL, $2.np, "main"); raiz = $$ .np; }
    ;
```

No arquivo de análise sintática, o translate.y, foi criado baseado nisso, uma estrutura do tipo union, para que ao decorrer da análise os tipos fossem retornados e inseridos na árvore. No código abaixo podemos verificar um exemplo de declaração dos tokens recebendo esse novo tipo, além da estrutura union criada.

```
%union{
    struct nomeVariavel{
        char nome[100];
        struct No *np;
    } NoObjeto;
}

%token <NoObjeto> MAIN_TOKEN
```

Além disso, no arquivo léxico, cada um dos retornos de tokens que temos passaram a receber o tipo desse nó, para que ao retornar para o código de análise sintática não houvessem erros, sendo isso destacado no código abaixo.

```
"main" {if(yylineno==1) printf("%d\t",yylineno++); printf(GRAY"%s
"RESET,yytext); strcpy(yylval.NoObjeto.nome,(yytext)); return
MAIN_TOKEN;}
```

Por fim, para exemplificar então o uso da nossa árvore de sintaxe abstrata, abaixo temos um código, que utiliza da declaração de função, e de outros comandos da linguagem, e logo em seguida uma imagem de como foi feita a saída no terminal.

```
main{
  // declaracao variavel
  Bug numero
  Pave("Digite um número: ")
  numero = Saharah()

  Mabel(numeroPar(numero) == true){
    Pave("O número é par!")
  }

  Sable(numeroPar(numero) == false){
    Pave("O número é impar!")
  }

  Label{
    Pave("O número é impar!")
  }

  Don numeroPar(Bug x, Bug y){
    Mabel(x%2==0){
      Rover true
    }

    Label{
      Rover false
    }
  }
}
```



Figura 9. Exemplo de Árvore Abstrata

Na Figura 9 podemos notar que ao apresentar sobre os comandos condicionais é repetido o nome do comando else (Label), sendo que isso ocorre devido a formulação do if, em que como apresentado no código abaixo, temos que primeiro passamos por uma etapa de verificação do formato do if, ou seja, se ele está sozinho, ou acompanhado de else if, ou else, e após isso é chamado o comando específico que necessitamos.

```
if:
    | IF_TOKEN ABRIR_PARENTESES_TOKEN relacional
    FECHAR_PARENTESES_TOKEN corpo tabulacao else {struct No *temp =
    criaNo($3.np, $5.np, $1.nome); $$np = criaNo(temp, $7.np,"Mabel
    Label");}
    | IF_TOKEN ABRIR_PARENTESES_TOKEN relacional
    FECHAR_PARENTESES_TOKEN corpo tabulacao elseIF tabulacao else { struct
    No *temp = criaNo($3.np, $5.np, $1.nome); struct No *temp2 =
    criaNo(temp, $7.np,$7.nome);$$np = criaNo(temp2,$9.np,"Sable Label");}

else:
    ELSE_TOKEN corpo {$$.np = criaNo(NULL, $2.np, $1.nome);}
    | {$$.np = NULL;}

elseIF:
    ELSE_IF_TOKEN ABRIR_PARENTESES_TOKEN relacional
    FECHAR_PARENTESES_TOKEN corpo {$$.np = criaNo($3.np, $5.np, $1.nome);}
    | elseIF tabulacao ELSE_IF_TOKEN ABRIR_PARENTESES_TOKEN relacional
    FECHAR_PARENTESES_TOKEN corpo tabulacao {$3.np = criaNo($5.np, $7.np,
    $3.nome); $$np = criaNo($1.np, $3.np, $1.nome);}
```

## ANÁLISE SEMÂNTICA

Na análise semântica é realizada a verificação de conversão de tipos, variáveis não declaradas, múltiplas declarações de variável, entre outras questões, de forma que caso haja erros, os mesmos são imprimidos juntamente com a linha onde o erro se encontra. É importante citar que a estrutura union apresentada anteriormente na seção sobre a árvore de sintaxe abstrata foi melhorada, para que agora identifique o tipo das variáveis. Abaixo está representado o novo formato da estrutura.

```
%union{
    struct nomeVariavel{
        char nome[100];
        struct No *np;
    } NoObjeto;

    struct NoObjeto2 {
        char nome[100];
        struct No *np;
        int tipo;
    } NoObjetoTipado;
}
```

Nessa nova estrutura temos que o tipo em **NoObjetoTipado** recebeu o tipo int para o **tipo** devido à declaração feita anteriormente dos tipos da linguagem, em que os mesmos foram declarados como um enum chamado **TipoToken**, apresentado na seção da tabela de símbolos durante a análise sintática. Unido a isso, foram feitas então nessa implementação 3 verificações possíveis da análise semântica, que são apresentadas a seguir.

### 1. Variáveis não Declaradas

Para essa verificação, foi feita uma função simples, em que se verifica se o identificador passado como parâmetro da função está presente ou não na tabela de símbolos, em que, se o identificador não estiver presente na tabela de símbolos, será exibida uma mensagem de erro dizendo que aquela variável não foi instanciada. Abaixo podemos ver o corpo da função mencionada, e em seguida um trecho de código que gera o erro de variável não declarada, em conjunto com a Figura 10, que ilustra o erro no terminal.

```
void checar_declaracao(char* c){
```

```

int simbl = simboloExiste(escopoAtual->tabela, strdup(c));
if (simbl == 0){
    sprintf(errors[errosemantrico], "Linha %d: Variável \"%s\" sendo
utilizada antes de ser declarada!\n", yylineno, c);
    errosemantrico++;
}
}

```


```

main{
    Bug numero
    Pave("Digite um número: ")
    numero = Saharah()

    Bug numero
    Pave("Digite um número: ")
    numero = Saharah()

    Pave(numero)
}

```



```

stardotwav@stardotwav: /mnt/c/Users/telal/OneDrive/Documents/github/TPCompiladores/TP3/gramatica$ make run < exemplo
./TP2
1      main {
3      Pave ( "Digite um número: " )
4      numero = Saharah ( )
5      Pave ( numero )
6  } Linha 4: Variável "numero" sendo utilizada antes de ser declarada!

Linha 4: inconsistência de tipos! Entre: None e SeaCreature

Código compilado com sucesso!

-----
Tabela de Símbolos
Num. Símbolos = 1
-----

Endereco: 0, Lexema: "Digite um número: ", Tipo: SeaCreature, Tipo token: Constante, Linha: 3

```

Figura 10. Exemplo de Erro Semântico em Variável não Declarada

## 2. Declaração de Múltiplas Variáveis

Para a verificação da declaração de múltiplas variáveis, visto que uma variável não pode ser redeclarada no mesmo escopo, a solução encontrada para fazer essa verificação foi feita uma modificação na função de inserção de símbolos na tabela de símbolos. De forma que, se o símbolo já estiver presente na tabela de símbolos, e a variável for do mesmo tipo, é emitido uma mensagem de erro informando que o usuário que houve múltiplas declarações de uma mesma variável. Abaixo podemos verificar um trecho de código que exemplifica tal situação, e logo após, na Figura 11 é mostrado a mensagem emitida ao usuário.

```

main{

```



```

Pave("Digite um número: ")
numero = Saharah()
Pave(numero)
}

```

```

stardotwav@stardotwav:/mnt/c/Users/telal/OneDrive/Documents/github/TPCompiladores/TP3/gramatica$ make run < exempl
./TP2
1   main {
3   Bug numero
4   Pave ( "Digite um número: " )
5   numero = Saharah ( )
6
7   Bug numero
8   Pave ( "Digite um número: " )
9   numero = Saharah ( )
10
11  Pave ( numero )
12  } Linha 5: inconsistência de tipos! Entre: Bug e SeaCreature

Linha 8: Múltiplas declarações da variável "numero"!

Linha 9: inconsistência de tipos! Entre: Bug e SeaCreature

Código compilado com sucesso!

-----
Tabela de Símbolos
Num. Símbolos = 2
-----
Endereco: 0, Lexema: numero, Tipo: Bug, Tipo token: Variavel, Linha: 4
Endereco: 0, Lexema: "Digite um número: ", Tipo: SeaCreature, Tipo token: Constante, Linha: 4

```

Figura 11. Exemplo de Erro Semântico de Múltiplas Declarações de Variável

Observando ainda a Figura 11, temos que é apresentado uma inconsistência de tipos durante a entrada de uma variável do tipo inteiro, em que é dito que a mesma também pode ser do tipo string, isso ocorre devido ao fato de que durante o trabalho foi usado do formato de entrada da linguagem Python, que trata situações assim com a conversão explícita de variáveis, usando por exemplo de `numero = int(input())`, porém tal conversão não foi implementada em nossa linguagem, dessa forma, sendo mantido o warning de tipos durante a entrada que não se apresenta no formato string.

### 3. Verificação de Tipo

Para a última implementação, usando da saída apresentada pela representação intermediária da árvore de sintaxe abstrata, foi criada uma função que indica como um aviso ao usuário quando verifica que durante a declaração de uma variável ela está recebendo o tipo correto, ou não. Abaixo podemos ver um trecho de código que ilustra tal situação, seguido da Figura 12 que apresenta os avisos apresentados ao usuário.

```

main{
  Bug idade
  idade = "22"
}

```

```
SeaCreature nome
nome = 22
}
```

```
stardotwav@stardotwav:/mnt/c/Users/teLa1/OneDrive/Documents/github/TPCompiladores/TP3/gramatica$
./TP2
1      main {
3      Bug idade
4      idade = "22"
5
6      SeaCreature nome
7      nome = 22
8      } Linha 4: inconsistência de tipos! Entre: Bug e SeaCreature

Linha 8: inconsistência de tipos! Entre: SeaCreature e Bug

Código compilado com sucesso!

-----Tabela de Símbolos-----
----- Num. Símbolos = 3 -----
Endereco: 0, Lexema: idade, Tipo: Bug, Tipo token: Variavel, Linha: 4
Endereco: 0, Lexema: "22", Tipo: SeaCreature, Tipo token: Constante, Linha: 4
Endereco: 0, Lexema: nome, Tipo: SeaCreature, Tipo token: Variavel, Linha: 7
```

Figura 12. Exemplo de Erros Desconhecidos e Tipo Incorreto

Além disso, é importante citar que como durante a chamada de uma função, como as mesmas na linguagem desenvolvida seguem o padrão da linguagem Python, ou seja, que não possuem identificação de tipo, a exemplo da Figura 11, imprimem uma mensagem de aviso ao usuário, indicando que a operação feita com as funções pode possuir tipos desconhecidos.

```

stardotwav@stardotwav:/mnt/c/Users/telal/OneDrive/Documents/github/TPCompiladores/TP3/gramatica
./TP2
1      main {
3      Bug num1
4      num1 = Saharah ( )
5
6      Fossil num2
7      num2 = Saharah ( )
8
9      Bug a = soma ( num1 , num2 )
10     Pave ( a )
11
12     Don soma ( Bug x , Bug y ) {
13     Rover x + y
14     }
15 } Linha 4: inconsistência de tipos! Entre: Bug e SeaCreature

Linha 7: inconsistência de tipos! Entre: Fossil e SeaCreature

Linha 10: Aviso! Operações entre tipo(s) desconhecidos!: Bug e None

Código compilado com sucesso!

-----
Tabela de Símbolos
-----
Num. Símbolos = 6 -----

Endereco: 0, Lexema: num1, Tipo: Bug, Tipo token: Variavel, Linha: 4
Endereco: 0, Lexema: num2, Tipo: Fossil, Tipo token: Variavel, Linha: 7
Endereco: 0, Lexema: a, Tipo: Bug, Tipo token: Variavel, Linha: 9
Endereco: 0, Lexema: soma, Tipo: None, Tipo token: Funcao, Linha: 12
Endereco: 0, Lexema: Bug, Tipo: Bug, Tipo token: Variavel, Linha: 12
Endereco: 0, Lexema: y, Tipo: Bug, Tipo token: Variavel, Linha: 12
-----

```

Figura 13. Exemplo de Erros Desconhecidos em Chamadas de Funções

## CÓDIGO DE TRÊS ENDEREÇOS

Para finalizar, como última implementação, temos uma representação intermediária linear, o Código de Três Endereços. Para a implementação dessa representação foi necessário a criação de algumas variáveis, que armazenam um contador de quantas variáveis temporárias temos, o número de labels usadas, e o código de três endereços em si. Além disso, foi acrescido ao union da estrutura da análise semântica mais um bloco, que agora contém o corpo de if, else e else if de estruturas condicionais, e de repetição, de forma que tal estrutura foi atribuída aos comandos que se relacionam com tais estruturas. Nos recortes do código apresentado podemos ver as variáveis adicionadas, e as alterações de estrutura atribuídas.

```

%{
    // ...
    int varTemporaria = 0;
    int label = 0;

    // ...
    char codigoEndereco[50][100];
    int contadorLinhasEndereco = 0;

```

```
}%

%union{
    struct nomeVariavel{
        char nome[100];
        struct No *np;
    } NoObjeto;

    struct NoObjeto2 {
        char nome[100];
        struct No *np;
        int tipo;
    } NoObjetoTipado;

    struct NoObjeto3 {
        char nome[100];
        struct No *np;
        int tipo;
        char corpoIf[5];
        char corpoElse[5];
        char corpoElseIf[5];
    } NoObjetoCorpo;
}
```

Após isso, foram adicionados aos comandos de atribuição, e operações aritméticas trechos de código que adicionavam à variável que armazena o código de três endereços suas operações, abaixo podemos verificar isso, em que temos a produção de atribuição, que utiliza de init, que armazena assim as formas possíveis de atribuição, como uma operação aritmética, por exemplo, e usando como exemplo a produção de soma, podemos verificar os passos que se leva para a produção desse código de três endereços.

```
atribuicao:
    VAR_TOKEN {checar_declaracao($1.nome);} ATRIB_TOKEN init {
        $1.np = criaNo(NULL, NULL, $1.nome);
        $$np = criaNo($1.np, $4.np, "=");
        checar_tipos(get_tipo_Tabela($1.nome), $4.tipo);
        sprintf(codigoEndereco[contadorLinhasEndereco++], "%s = %s\n",
$1.nome, $4.nome);
    }

init:
    aritmetica {$$.np = $1.np; $.tipo = $1.tipo; strcpy($$.nome,
$1.nome);}
    | anyType {$$.np = $1.np; $.tipo = $1.tipo; strcpy($$.nome,
$1.nome);}
    | relacional {$$.np = $1.np; $.tipo = $1.tipo; strcpy($$.nome,
$1.nome);}
    | input {$$.np = $1.np; $.tipo = T_STRING; strcpy($$.nome, $1.nome);}
```

#### aritmetica:

```
    numero {$$$.np = $1.np; $$$.tipo = $1.tipo; strcpy($$.nome, $1.nome);}
    | VAR_TOKEN {$$$.np = criaNo(NULL, NULL, $1.nome); $$$.tipo =
get_tipo_Tabela($1.nome); strcpy($$.nome, $1.nome);}
    | funcao {$$$.np = $1.np; $$$.tipo = T_DESCONHECIDO; strcpy($$.nome,
$1.nome);}
    | aritmetica MATEMATICO_SOMA aritmetica {
    $$$np = criaNo($1.np, $3.np, "+");
    $$$tipo = $1.tipo;
    sprintf($$.nome, "t%d", varTemporaria++);
    printf(codigoEndereco[contadorLinhasEndereco++], "%s = %s %s
%s\n", $$.nome, $1.nome, $2.nome, $3.nome);
    }
```

Como última inserção então do código de três endereços foi definido os comandos das estruturas condicionais e de repetição, isso porque eles necessitam da implementação de Labels e do Goto. Abaixo temos um exemplo da implementação desses comandos, em que no comando while, ele possui uma condição, sendo essa a condição do IF, que gera o goto, caso seja verdadeira, ela executa uma determinada label, caso contrário, é enviado para a próxima label após o comando de repetição.

#### repeticao:

```
// ...
| WHILE_TOKEN ABRIR_PARENTESES_TOKEN relacional {
    sprintf($3.corpoIf, "%d", label++);
    sprintf($3.corpoElse, "%d", label++);
    printf(codigoEndereco[contadorLinhasEndereco++], "t%d = %s\n",
varTemporaria, $3.nome);
    printf(codigoEndereco[contadorLinhasEndereco++], "LABEL %s\n",
$3.corpoIf);
    printf(codigoEndereco[contadorLinhasEndereco++], "IF FALSE (t%d)
GOTO %s\n", varTemporaria, $3.corpoElse);
    printf(codigoEndereco[contadorLinhasEndereco++], "t%d = t%d +
1\n", varTemporaria, varTemporaria);
    varTemporaria++;
}
| FECHAR_PARENTESES_TOKEN corpo {
    $$$np = criaNo($3.np, $6.np, $1.nome);
    printf(codigoEndereco[contadorLinhasEndereco++], "GOTO %s\n",
$3.corpoIf);
    printf(codigoEndereco[contadorLinhasEndereco++], "LABEL %s\n",
$3.corpoElse);
}
```

É importante ressaltar que para as definições de gramática da linguagem, apenas a declaração e chamada de funções não foi implementada por possuir um alto nível de complexidade. Abaixo podemos ver um exemplo de código que possui comandos condicionais, de repetição, além de atribuições simples, e compostas por operações aritméticas para exemplificar a geração do código de três endereços, na Figura 14 temos então a execução do mesmo.

```
main{
  Bug numero
  Pave("Digite um número: ")
  numero = Saharah()

  // Variavel para armazenar a soma de dois em dois
  Bug somaIntervalo = 0
  Bug i = 1
  numero = numero-1

  Wilbur(true){
    Mabel(i == numero){
      Brewster
    }

    somaIntervalo = somaIntervalo + i
    i = i + 2
  }

  Pave(somaIntervalo)
}
```

```
-----
CÓDIGO DE TRÊS ENDEREÇOS
-----
Pave("Digite um número: ")
numero = Saharah()
somaIntervalo = 0
i = 1
numero = numero
t0 = true
LABEL 0
IF FALSE (t0) GOTO 1
t0 = t0 + 1
t1 = i == numero
IF FALSE (t1) GOTO 3
GOTO 2
LABEL 3
LABEL 2
t2 = somaIntervalo + i
somaIntervalo = t2
t3 = i + 2
i = t3
GOTO 0
LABEL 1
Pave(somaIntervalo)
-----
```

Figura 14. Exemplo de Código de Três Endereços

## CONSIDERAÇÕES FINAIS

Para a primeira parte do trabalho já é possível notar uma alta complexidade na formulação tanto de uma linguagem, quanto a sua compilação, isso dado o fato de que muita coisa pode ser refatorada ao longo do projeto por questões de como será implementada a compilação da mesma. É importante frisar também como a definição dos conceitos de tokens e da gramática trás uma ampla visão sobre o que será implementado, ajudando assim a ver algumas questões que podem gerar problemas ao longo das demais etapas de compilação.

Para a segunda parte do trabalho foi possível notar que a implementação da gramática utilizando o Yacc foi complexa de ser realizada, demandando uma curva de aprendizado maior comparado à primeira parte. Foi possível concluir o trabalho com êxito implementando as todas as funções que foram definidas na primeira parte do presente trabalho prático.

Na última parte do trabalho, foi possível identificar a necessidade de passar por todas as etapas de compilação da linguagem para encontrar questões que poderiam ser implementadas de melhor ou pior forma de acordo com a forma que as análises léxicas, sintáticas e semânticas eram feitas, além de observar as estruturas que essas análises usam, para assim facilitar a utilização das mesmas. O trabalho como um todo foi de extremo valor de aprendizado, de forma que, com ele foi possível entender melhor os conceitos que envolvem o projeto de um compilador, e as etapas de análise feita em cada uma delas.

## REFERÊNCIAS BIBLIOGRÁFICAS

[1] Aho, Alfred V., et al. Compilers: principles, techniques and tools. 2020

[2] Animal Crossing Wiki. Disponível em: <[https://animalcrossing.fandom.com/wiki/Animal\\_Crossing\\_Wiki](https://animalcrossing.fandom.com/wiki/Animal_Crossing_Wiki)>. Acesso em 13 de Junho de 2022.