	<p style="text-align: center;"> <Relatório Final Compiladores> por <Bruno Rodrigues > - 2020.1 Boa vista, 09/12/2020 </p>
--	---

Relatório Final Compiladores

Bruno Rodrigues Caputo (brunorcx@hotmail.com)

DCC605-Construção de compiladores 2020.1-Turma 01

Universidade Federal de Roraima

DCC -Departamento de Ciência da Computação - Bloco V

Campus Universitário do Paricarana - Aeroporto

69310-000 Boa vista, RR

1. INTRODUÇÃO

Este programa foi desenvolvido utilizando a *IDE Visual Studio 2019*. A linguagem C# fora adotada e toda a sua criação de interface se dá por *WindowsForms*, assim, somente será possível executá-lo em sistemas operacionais *windows*.

2. IMPLEMENTAÇÃO

2.1. Análise léxica

Inicialmente é feita a atribuição de todas as palavras reservadas, operadores e caracteres especiais em *strings*, sendo estes atributos da classe *Lexico.cs*. Em seguida, para facilitar a comparação, estas *strings* serão adicionadas a uma lista do tipo *List<string>*.

Tem-se a leitura do arquivo *Codigo.txt*, bem como, armazenamento do código em uma outra lista do tipo *List<string>*.

Finalmente a função principal *separarLexemas()* fará a separação dos lexemas de acordo com os seus devidos rótulos, cabe aqui ressaltar a utilização do tipo *DataTable*, estrutura adotada para representar a tabela final dos lexemas.

Duas importantes funções são *acharSimbolos()* e *identificarRotulos()*, responsáveis por comparar uma string com os símbolos que foram atribuídos anteriormente nas listas.

As 4 gramáticas regulares utilizadas no analisador léxico são:


Identificadores: *l* representa letras e *_*. Enquanto que *d* representa dígitos

$N \rightarrow lR$

$R \rightarrow lR$

$R \rightarrow dR$

$R \rightarrow \epsilon$

 <p>UFRR</p>	<p align="center"><Relatório Final Compiladores> por <Bruno Rodrigues > - 2020.1 Boa vista, 09/12/2020</p>
--	--

Números:

$S \rightarrow dR$

$R \rightarrow dR$

$R \rightarrow ;$

$R \rightarrow .T$

$T \rightarrow dP$

$P \rightarrow dP$

$P \rightarrow ;$

Função:

$S \rightarrow lR$

$R \rightarrow l \mid d R$

$R \rightarrow (T$

$T \rightarrow lP$

$P \rightarrow l \mid d P$

$P \rightarrow)M$

$T \rightarrow) M$

$M \rightarrow ;$

String:

$S \rightarrow \text{"} R$

$R \rightarrow l \mid d R$

$R \rightarrow \text{"} T$

$T \rightarrow \epsilon$

2.2. Análise Sintática

Gramáticas Utilizadas:


$G, V_t = \{+, x, (,), v\}, V_n = \{E, E', M, M', P\}$ e Símbolo sentencial E Representando a Soma e multiplicação:

$E \rightarrow M + E' \quad 1$

$E' \rightarrow +ME' \quad 2$

$E' \rightarrow \epsilon \quad 3$

$M \rightarrow PM' \quad 4$

	<p align="center"><Relatório Final Compiladores> por <Bruno Rodrigues > - 2020.1 Boa vista, 09/12/2020</p>
--	--

$M \rightarrow xPM'$	5
$M \rightarrow \epsilon$	6
$M \rightarrow (E)$	7
$M \rightarrow v$	8

V representa ID(identificadores), INT(inteiros) , FLOAT(floats)

Uma nova classe chamada *Sintaxe.cs* foi criada, a qual possui um construtor que deverá conter uma tabela já tratada pelo analisador léxico. Depois basta chamar o método principal, que se orienta da seguinte maneira `public List<TreeNode> AnalisadorPreditivo()`. Seu retorno é uma lista já configurada para exibição através da ferramenta de controle *TreeView*. Assim, basta duas linhas para realizar a abstração para o analisador sintático.

```
Sintaxe sintaxe = new Sintaxe(tabela);
arvores = sintaxe.AnalisadorPreditivo();
```

Finalmente, após pegar uma sentença por linha(até o “;”). Temos o método `private TreeNode<string> SomaMultiParen(string linhaLexica)`, ele possui o retorno da árvore sintática que será adicionado a uma lista com todas as sentenças reconhecidas do código fonte. Dentro deste método vale ressaltar, o uso de uma matriz para representar a tabela sintática da gramática, uma pilha e a adição de árvore. Dois tipos de estruturas de dados foram utilizados para a árvore, o *TreeNode* e o *TreeNode<string>* sendo este último uma classe adaptada e construída.

O método `private List<TreeNode> FormatarArvore()`, faz a adaptação final para visualização da árvore.


2.3. Análise Sintática Precedência Fraca

Gramáticas Utilizadas:

$G, V_t = \{+, x, (,), v\}$, $V_n = \{E, M, P\}$ e Símbolo sentencial E Representando a Soma e multiplicação:

$E \rightarrow E + M$	1
$E \rightarrow M$	2
$M \rightarrow M \times P$	3
$M \rightarrow P$	4
$P \rightarrow (E)$	5
$P \rightarrow v$	6

V representa ID(identificadores), INT(inteiros) , FLOAT(floats)

	<p align="center"><Relatório Final Compiladores> por <Bruno Rodrigues > - 2020.1 Boa vista, 09/12/2020</p>
--	--

De modo similar a análise sintática preditiva a análise de precedência fraca utiliza dois métodos principais, orientados da seguinte forma `public List<TreeNode> AnalisadorPrecedênciaFrac()`. Responsável por trabalhar com a tabela da análise léxica e enviar para o segundo método uma `string` contendo a linha que será validada quanto à gramática. Seu retorno é uma lista já configurada para exibição através da ferramenta de controle `TreeView`.

Como método principal tem-se `private TreeNode PrecedênciaSMParen(string LinhaLexica)` onde se é utilizado o tipo `TreeNode`, uma pilha auxiliar e uma lista de `TreeNodes` temporário para armazenar os filhos em uma árvore durante a execução do algoritmo. Com o retorno da árvore sintática que será adicionado a uma lista com todas as sentenças reconhecidas do código fonte. Dentro deste método vale ressaltar, o uso de uma matriz de deslocamento-redução, duas pilhas e a adição de árvore. Analogamente, basta duas linhas para realizar a abstração para o analisador sintático.

```
Sintaxe sintaxe = new Sintaxe(tabela);
arvoresFracas = sintaxe.AnalisadorPrecedênciaFrac();
```

2.4. Análise Semântica

Uma nova classe chamada *Semantico.cs* foi criada, ela contém 3 métodos principais:

- `private DataTable CriarEsquemaTabelaSimbolos();`
- `private DataRow ProcurarVariavelRepetida(DataTable tabela, int escopoAtual, string variavel)`
- `public DataTable PreencherTabela()`

Inicialmente é preciso criar um esquema da tabela de símbolos usando a estrutura de dados `DataTable` o que é feito pelo primeiro método. O segundo método é utilizado ao longo do restante da classe. Ele irá retornar a linha em que se encontrou uma variável que já foi criada. O método principal é *PreencherTabela()*. Nele com o auxílio da tabela de lexemas se faz a interpretação de expressões de atribuição e chamada de funções e também verificação de tipos. Sendo elas: Se uma variável já está sendo usada antes de ser declarada; Na atribuição verificar se o lado direito obedece ao tipo da variável (se ela for *INT*, por exemplo, só deve aceitar valores inteiros). E verificar escopos variáveis e funções no escopo global e *main*.

2.5. Geração de código intermediário

Uma nova classe chamada *CondInter.cs* foi criada. Ela é composta por quatro métodos principais:

- private DataTable CriarEsquemaCodInter()
- public DataTable PreencherTabela()
- private List<DataRow> GerarCodigoLinha(TreeNode arvore, DataTable tabelaCod)
- private void ArvoreSimplificada(TreeNode pai)

Similarmente com as outras classes principais inicialmente é preciso montar o esquema da tabela de código intermediário, em seguida, o principal método responde para o preenchimento da tabela. Auxiliado pelas árvores sintáticas de precedência fraca, tabela semântica e tabela de símbolos faz a varredura de atribuições. GerarCodigoLinha() por sua vez irá com a ajuda da ArvoreSimplificada() caminhar na árvore e pegar os valores para finalmente preenche-los.

3. EXECUÇÃO


3.1. Léxica

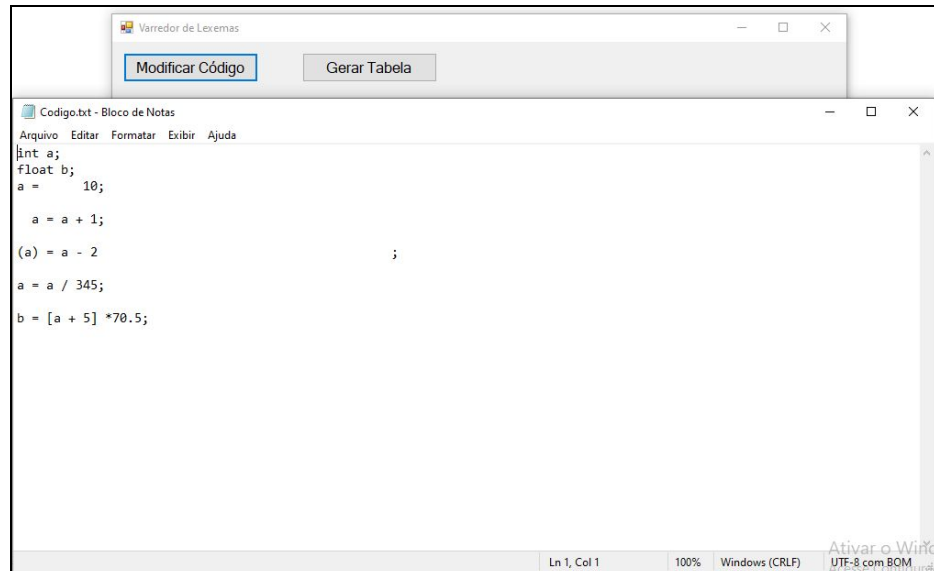
Basta executar o arquivo *CompiladorInterface.application*. Assim que o programa iniciar a primeira e única tela será mostrada.



Página inicial

Nela temos dois botões *Modificar Código* e *Gerar Tabela*. O primeiro irá abrir o notepad onde o código deve ser inserido(inicialmente já vem com um caso de teste).

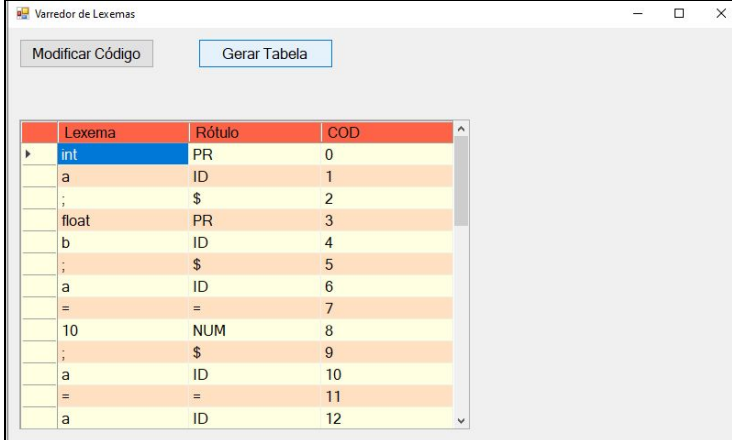
	<p style="text-align: center;"><Relatório Final Compiladores> por <Bruno Rodrigues > - 2020.1 Boa vista, 09/12/2020</p>
--	--



Bloco de Notas aberto após pressionar *Modificar Código*

- Neste caso de teste é possível observar vários exemplos interessantes;
- Na linha 1 temos *int a*; representados na tabela que será mostrado a seguir pelos códigos de [0-3].
 - Na linha 2 temos um espaço antes de *float*, que será ignorado.
 - Na linha 3 além do símbolo =, temos um espaço no meio da atribuição.
 - Na linha 4 temos uma linha em branco, que será ignorada.
 - Na linha 7 temos o parêntesis sem espaço, além do espaço até o ponto e vírgula.
 - Na linha 9 temos o símbolo / e um número maior 345.
 - Na linha 10 temos colchetes, um *float* e *.

O segundo botão irá mostrar a tabela com os lexemas separados, seus rótulos e códigos.

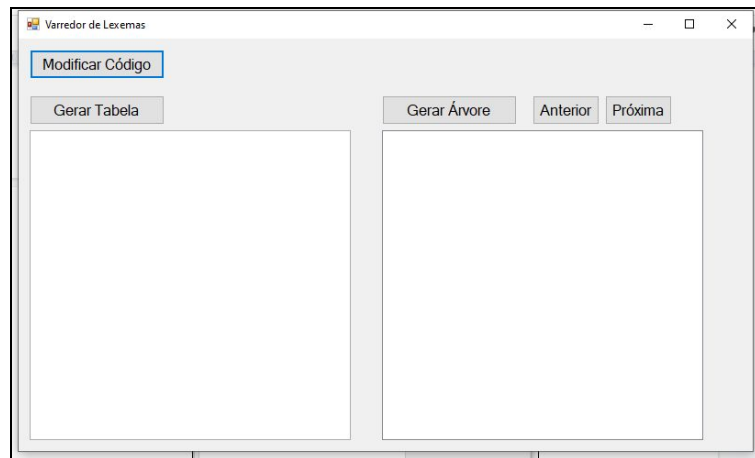


Lexema	Rótulo	COD
int	PR	0
a	ID	1
;	\$	2
float	PR	3
b	ID	4
;	\$	5
a	ID	6
=	=	7
10	NUM	8
;	\$	9
a	ID	10
=	=	11
a	ID	12

Exemplo de tabela de lexemas


3.2. Sintática

Basta executar o arquivo *CompiladorInterface.application*. Assim que o programa iniciar a primeira e única tela será mostrada. A tela foi modificada e agora contém 5 botões e duas caixas de texto



Página inicial

Os novos botões são *Gerar Árvore*, *Anterior* e *Próxima*. O primeiro irá gerar todas as árvores reconhecidas, o segundo e o terceiro ficarão responsáveis pela navegação destas. Para este caso de teste, duas sentenças que devem ser reconhecidas são colocadas nas linhas iniciais e na terceira uma em que a gramática não contempla.




UFRR

<Relatório Final Compiladores>

por <Bruno Rodrigues > - 2020.1

Boa vista, 09/12/2020



```

Codigo.txt - Bloco de Notas
Arquivo  Editar  Formatar  Exibir  Ajuda
a + 3 * iden2;
( a + 3 * x );
( a ) = a - 2;

int a;
float b;
a = 10;

func();
FunçãoComPar(Parâmentro);
FuncErrada(d;

    a = a + 1;

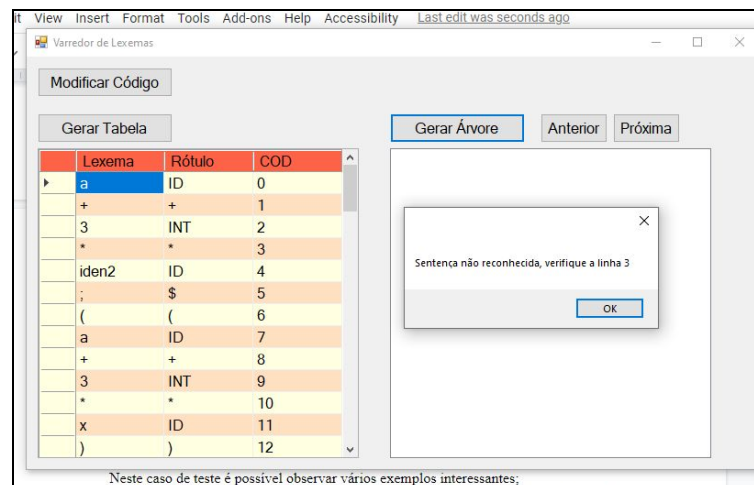
(a) = a - 2                                ;

variavel12 = _Delimitador / 345;

b = var * 70.5;
float real = 25.66;
"Mario"
"Mario"ç
"StringErrada ComEspaço"
  
```

Bloco de Notas aberto após pressionar *Modificar Código*

Assim que o botão *Gerar Árvore* for acionado uma mensagem de erro será mostrada caso alguma sentença não tenha sido reconhecida



View Insert Format Tools Add-ons Help Accessibility Last edit was seconds ago

Varredor de Lexemas

Modificar Código

Gerar Tabela

Lexema	Rótulo	COD
a	ID	0
+	+	1
3	INT	2
*	*	3
iden2	ID	4
,	\$	5
((6
a	ID	7
+	+	8
3	INT	9
*	*	10
x	ID	11
))	12

Gerar Árvore Anterior Próxima

Sentença não reconhecida, verifique a linha 3

OK

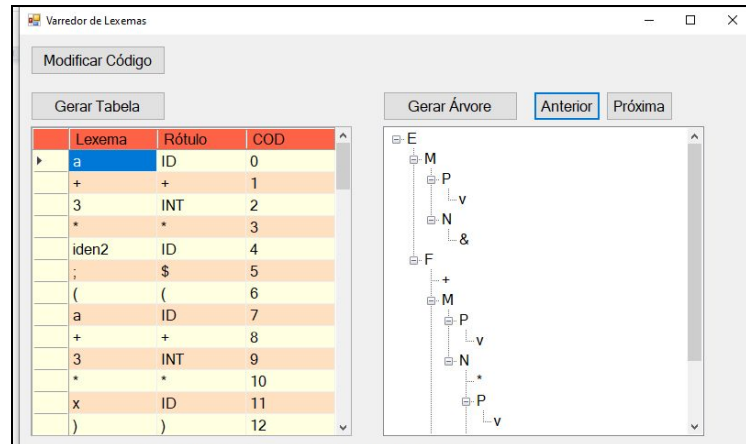
Neste caso de teste é possível observar vários exemplos interessantes;

Mensagem de erro com o número da linha com o erro

Em seguida a primeira árvore sintática será mostrada, agora basta navegar pelas árvores com os botões *Anterior* e *Próxima*.

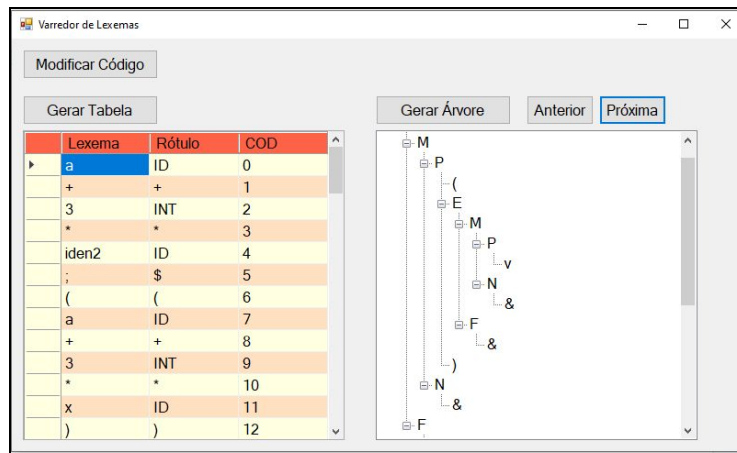


<Relatório Final Compiladores> por <Bruno Rodrigues > - 2020.1 Boa vista, 09/12/2020



Primeira árvore

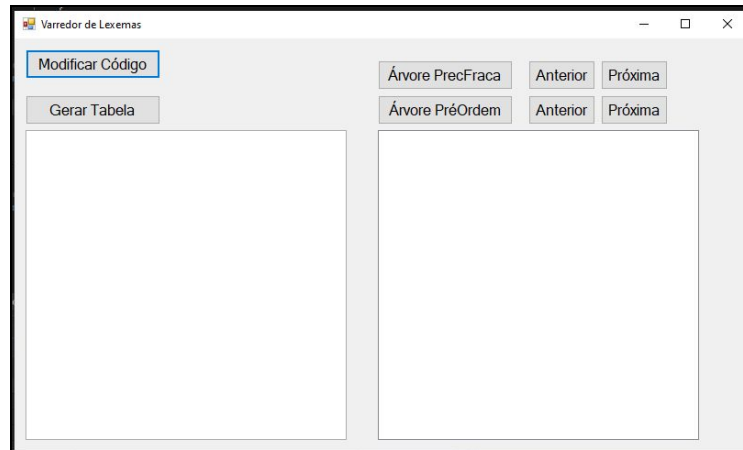
Vale lembrar que para a geração da árvore é estritamente necessário acionar o botão *Gerar Tabela*, uma vez que todo o processo parte da tabela léxica.



Segunda árvore

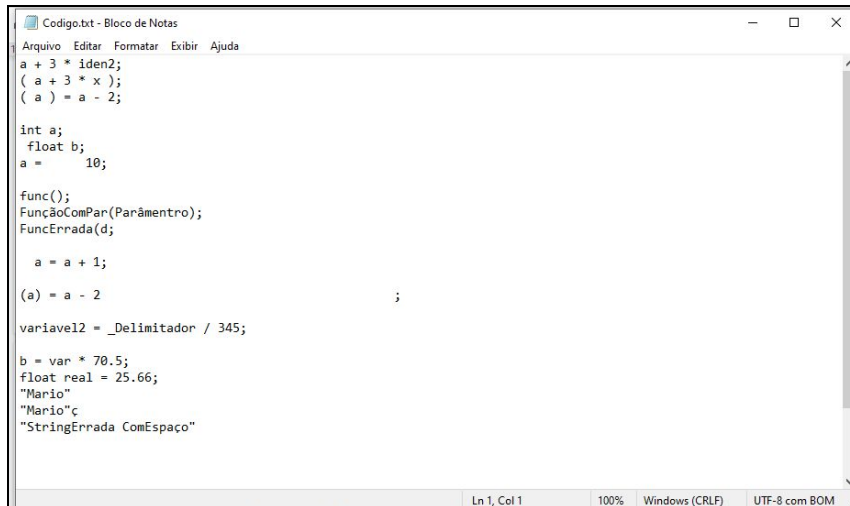
3.3. Sintática Precedência Fraca

Basta executar o arquivo *CompiladorInterface.application*. Assim que o programa iniciar a primeira e única tela será mostrada. A tela foi modificada e agora contém 8 botões.



Página inicial

Os novos botões são *Árvore PrecFrac*, *Anterior* e *Próxima*, com uma alteração no texto da árvore do analisador preditivo (*Árvore PréOrdem*). O primeiro irá gerar todas as árvores reconhecidas de precedência fraca, o segundo e o terceiro ficarão responsáveis pela navegação destas. Para este caso de teste, as mesmas duas sentenças para o reconhecimento são utilizadas, e na terceira uma em que a gramática não contempla.



```

a + 3 * iden2;
( a + 3 * x );
( a ) = a - 2;

int a;
float b;
a = 10;

func();
FunçãoComPar(Parâmento);
FuncErrada(d);

a = a + 1;

(a) = a - 2;

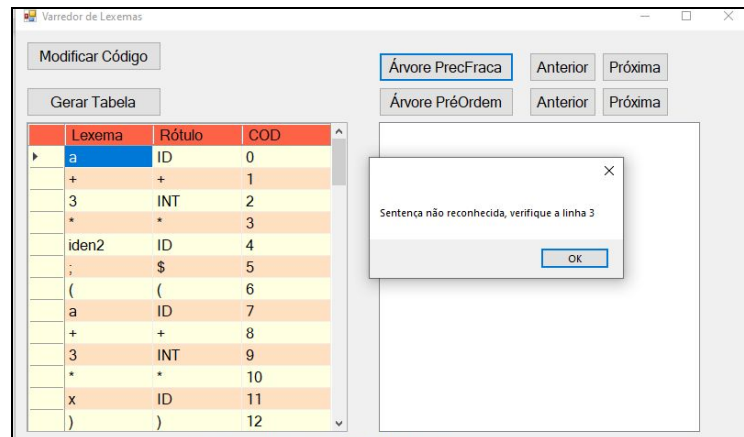
variavel12 = _Delimitador / 345;

b = var * 70.5;
float real = 25.66;
"\"Mario\"c
"\"StringErrada ComEspaço"

```

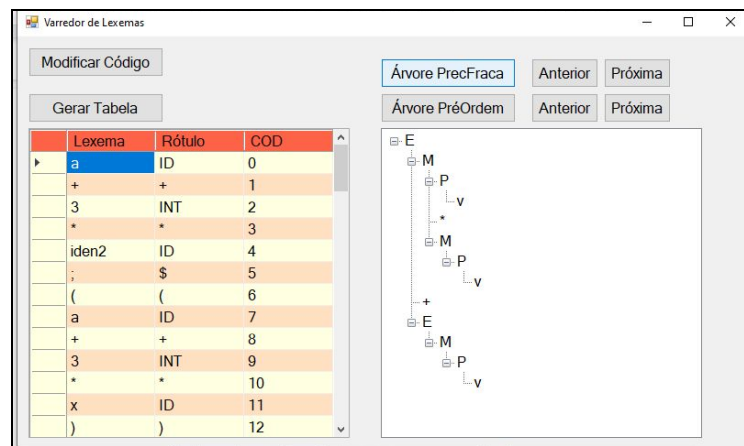
Bloco de Notas aberto após pressionar *Modificar Código*

Assim que o botão *Árvore PrecFrac* for acionado uma mensagem de erro será mostrada caso alguma sentença não tenha sido reconhecida



Mensagem de erro com o número da linha

Em seguida a primeira árvore sintática será mostrada, agora basta navegar pelas árvores com os botões *Anterior* e *Próxima*.

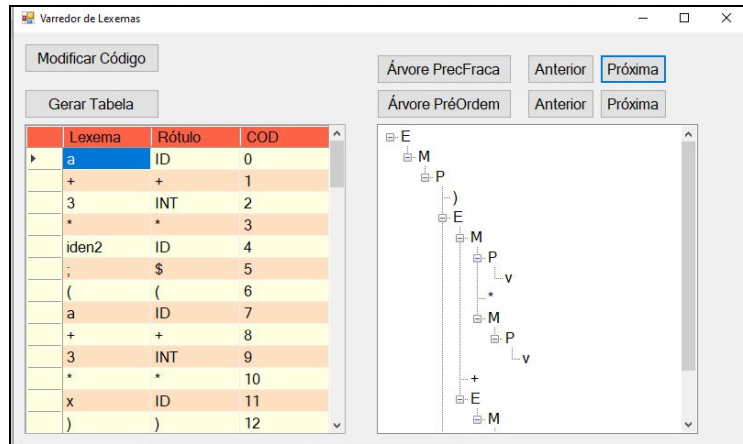


Primeira árvore

Vale lembrar que para a geração da árvore é estritamente necessário acionar o botão *Gerar Tabela*, uma vez que todo o processo parte da tabela léxica.



<Relatório Final Compiladores> por <Bruno Rodrigues > - 2020.1 Boa vista, 09/12/2020



Segunda árvore

3.4. Semântica

Um novo botão, uma nova tabela e uma caixa com a lista de erros foram adicionados, seu funcionamento é bem intuitivo basta selecionar o botão *Tabela Símbolos* para gerar a tabela e os erros encontrados também serão listados logo abaixo.



<Relatório Final Compiladores> por <Bruno Rodrigues > - 2020.1 Boa vista, 09/12/2020

Compilador

Modificar Código

Gerar Tabela

Lexema	Rótulo	COD
int	PR	0
x	ID	1
=	=	2
2	INT	3
;	\$	4
c	ID	5
=	=	6
3	INT	7
+	+	8
5	INT	9
*	*	10
9	INT	11
;	\$	12

Árvore PrecFraca

Árvore PréOrdem

Tabela Símbolos

Lexema	Rótulo	Tipo	Valor	Escopo	Declar	Linha
int	PR		0			1
x	VAR	INT	0	S		1
int	PR		0			3
a	VAR	INT	0	S		3
int	PR		0			5
b	VAR	INT	0	S		5
int	PR		1			10
main	IDFU...	int	1			10
float	PR		1			11
b	VAR	FLOAT	1	S		11
double	PR		1			14
d	VAR	DOU...	1	S		14
}	}		0			15

Cód. Intermediário

Erros Semânticos

- ERRO Variável c não foi declarada próximo a linha 2
- ERRO Variável iden2 não foi declarada próximo a linha 5
- ERRO Variável d não foi declarada próximo a linha 6
- Somente as expressões de atribuição e chamada podem ser usadas como instrução linha 7
- Somente as expressões de atribuição e chamada podem ser usadas como instrução linha 8
- ERRO variável x é do tipo INT, portanto só pode receber valores do mesmo tipo linha 9
- ERRO Variável m não foi declarada próximo a linha 13
- ERRO Variável d não foi declarada próximo a linha 16

Tela Atual mostrando erros e tabela de símbolos

3.5. Geração de código intermediário

Um novo botão chamado *Cod. Intermediário* foi criado, após a execução de todos os componentes anteriores é possível visualizar a tabela seguindo o esperado de acordo com as árvores sintáticas.

Compilador

Modificar Código

Gerar Tabela

Lexema	Rótulo	COD
int	PR	0
x	ID	1
=	=	2
2	INT	3
;	\$	4
c	ID	5
=	=	6
3	INT	7
+	+	8
5	INT	9
*	*	10
9	INT	11
*	*	12

Árvore PrecFraca

Árvore PréOrdem

Tabela Símbolos

Lexema	Rótulo	Tipo	Valor	Escopo	Declar	Linha
int	PR		0			1
x	VAR	INT	0	S		1
int	PR		0			3
a	VAR	INT	0	S		3
int	PR		0			5
b	VAR	INT	0	S		5
int	PR		1			10
main	IDFU...	int	1			10
float	PR		1			11
b	VAR	FLOAT	1	S		11
double	PR		1			14
d	VAR	DOU...	1	S		14
}	}		0			15


Cód. Intermediário

Operador	Arg1	Arg2	Resultado	Linha
=	2		2	1
*	p	9	t0	2
*	t0	5	t1	2

Erros Semânticos

- ERRO Variável c não foi declarada próximo a linha 2
- ERRO Variável iden2 não foi declarada próximo a linha 5
- ERRO Variável d não foi declarada próximo a linha 6
- Somente as expressões de atribuição e chamada podem ser usadas como instrução linha 7
- Somente as expressões de atribuição e chamada podem ser usadas como instrução linha 8
- ERRO variável x é do tipo INT, portanto só pode receber valores do mesmo tipo linha 9
- ERRO Variável m não foi declarada próximo a linha 13
- ERRO Variável d não foi declarada próximo a linha 16


Cód intermediário apresentado na tabela

	<p><Relatório Final Compiladores> por <Bruno Rodrigues > - 2020.1 Boa vista, 09/12/2020</p>
--	---

A coluna *Linha* nesta nova tabela representa a linha do código fonte referente ao código gerado, por isso é possível observar o número 2 se repetindo na imagem acima.

Por fim, basta fechar o programa no ícone *x*. Para desinstalar o programa basta utilizar o menu padrão do windows. Caso queira abrir o menu rapidamente pressione Windows+R e digite: *appwiz.cpl* e clique em *OK*.

O código fonte localizado no sítio: <https://github.com/brunorex/CompiladorInterface.git>.

	<p><Relatório Final Compiladores> por <Bruno Rodrigues > - 2020.1 Boa vista, 09/12/2020</p>
--	---

4. BIBLIOGRAFIA

Conceitos baseado nos *slides* do professor Luciano Ferreira Silva, Dr.

Embasamento teórico “Introduction to Compiler Design, Springer-Verlag London Limited 2011, Torben Ægidius Mogensen University of Copenhagen, Denmark”

<https://www.facom.ufms.br/~ricardo/Courses/CompilerI-2009/Materials/>

<http://www.decom.ufop.br/romildo/2018-2/bcc328/>