Universidade Federal Fluminense
Niterói - 2022.2

Compiladores - Relatório Final

Lucas Fuzato, Tiago Lacerda e Viviane Romero

Projeto disponível em:

https://replit.com/@TiagoMLB/c-minus-compiler#.replit
https://github.com/TiagoLacerda/c-minus-compiler

1. Ferramentas Utilizadas

Toolbox - https://cyberzhg.github.io/toolbox/nfa2dfa
RegExr - https://regexr.com/
Python - https://www.python.org/
Visual Studio Code - https://code.visualstudio.com/
Replit - https://replit.com/~

2. Dificuldades e Problemas Enfrentados

Inicialmente, tivemos dificuldade em identificar tokens de comentários, pois estávamos quebrando a cadeia lida assim que um símbolo lido levava a um estado de não-aceitação. Contornamos este problema lendo a cadeia até o final e mantendo um ponteiro para o símbolo que gerava a cadeia aceita mais longa.

Tivemos também dificuldade em gerar uma representação visual do autômato completo gerado, uma vez que possui mais de 100 estados. Portanto, optamos por gerar representações visuais dos autômatos mais simples que o compõem.

À princípio, tentamos implementar o analisador sintático de forma a receber a especificação de uma gramática e construir o analisador de forma genérica, porém, não conseguimos implementá-lo de forma a analisar toda a árvore sintática gerada durante a avaliação de uma sequência de tokens. Julgamos que seria inviável a correção da implementação anterior, visto que não era feita a análise sintática de fato e, portanto, optamos por elaborar o analisador sintático descendente recursivo da forma sugerida pela professora, atrelado à gramática da linguagem escolhida (C-minus).

3. Limitações

3.1. Limitações do Analisador Sintático

Uma das limitações da nossa implementação do analisador sintático é que a funcionalidade que faz a verificação de variáveis declaradas não guarda o escopo da variável declarada e portanto não é possível verificar se ela foi declarada no mesmo escopo onde ela foi referenciada.

3.2. Limitações da Linguagem Escolhida

Algumas das limitações da nossa linguagem escolhida são:

- Impossibilidade de atribuição de valor a uma variável na sua declaração (e.g. `int i = 0;`).
- Tipagem limitada (apenas inteiros).
- Impossibilidade de terem identificadores com números (e.g. int `variavel1`).

4. Alfabeto

{
  a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z,
  A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z,
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
  +, -, *, /, <, >, =, !,
  (,), [,], {, }, ;,
  ,
}

5. Expressões regulares

5.1. Keywords

else: else
if: if
int: int
return: return
void: void
while: while

Token	Expressão Regular	Valor do Atributo
else	else	-
if	if	-
int	int	-
return	return	-

void	void	-
while	while	-

5.2. Símbolos especiais

PLUS: +
MINUS: -
ASTERISK: *
SLASH: /
LT: <
LE: <=
GT: >
GE: >=
EQ: ==
NE: !=
ASSIGN: =
SEMICOLON: ;
COMMA: ,
OPENPARENTHESIS: (
CLOSEPARENTHESIS:)
OPENSQUAREBRACKETS: [
CLOSESQUAREBRACKETS:]
OPENBRACKETS: {
CLOSEBRACKETS: }
OPENCOMMENT: /*
CLOSECOMMENT: */

Token	Expressão Regular	Valor do Atributo
PLUS	+	-
MINUS	-	-
ASTERISK	*	-
SLASH	/	-

LT	<	-
LE	<=	-
GT	>	-
GE	>=	-
EQ	==	-
NE	!=	-
ASSIGN	=	-
SEMICOLON	;	-
COMMA	,	-
OPENPARENTHESIS	(	-
CLOSEPARENTHESIS	)	-
OPENSQUAREBRACKETS	[	-
CLOSESQUAREBRACKETS	]	-
OPENBRACKETS	{	-
CLOSEBRACKETS	}	-
OPENCOMMENT	/*	-
CLOSECOMMENT	*/	-

5.3. Comentários

/*(Σ)*/

5.4. Demais

LETTER: [a-zA-Z]
DIGIT: [0-9]
NUM: [0-9]+
ID: [a-zA-Z]+

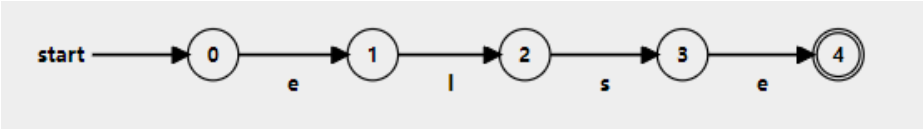
Token	Expressão Regular	Valor do Atributo
LETTER	[a-zA-Z]	Uma letra de a-z maiúscula ou minúscula
DIGIT	[0-9]	Um dígito numérico de 0-9
NUM	[0-9]+	Uma sequência de dígitos numéricos de 0-9
ID	[a-zA-Z]+	Uma sequência de letras de a-z maiúsculas ou minúsculas

6. Autômatos (NFA, DFA e DFA mínimo)

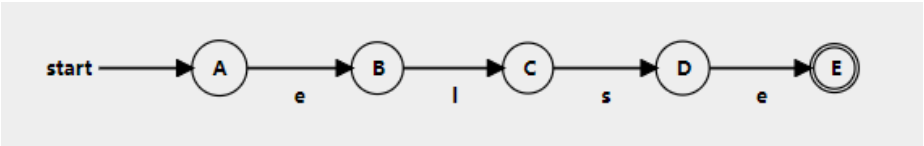
6.1. Keywords

- **else**

NFA:



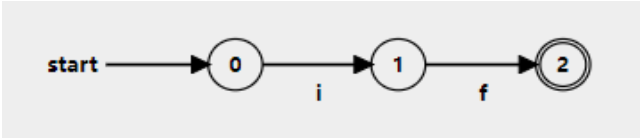
DFA:



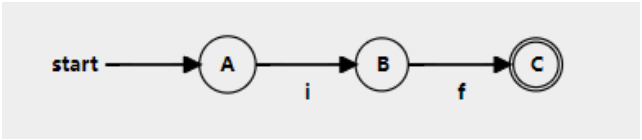
DFA mínimo:

- **if**

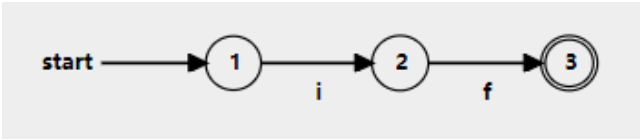
NFA:



DFA:



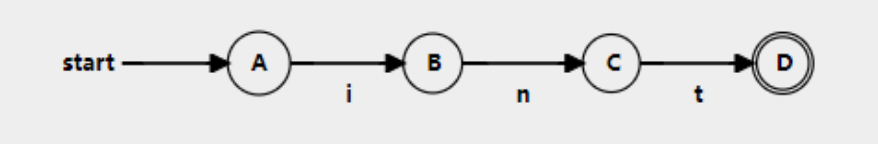
DFA mínimo:



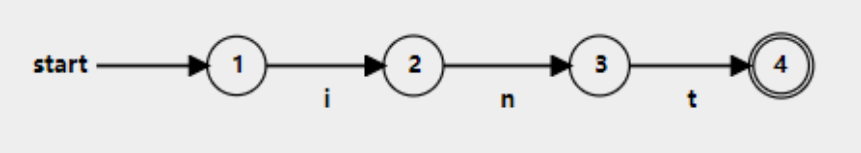
- **int**

NFA:

DFA:

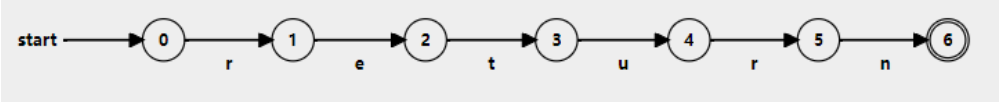


DFA mínimo:

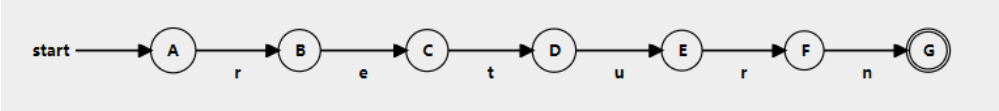


- **return**

NFA:



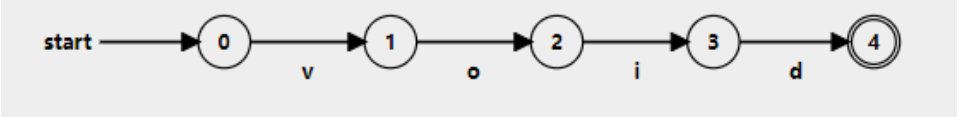
DFA:



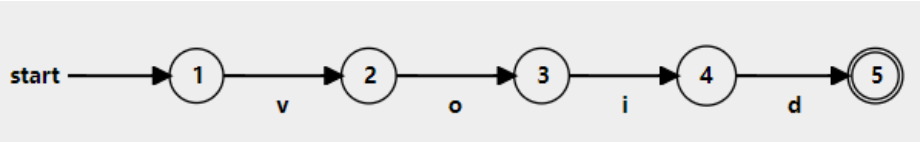
DFA mínimo:

- **void**

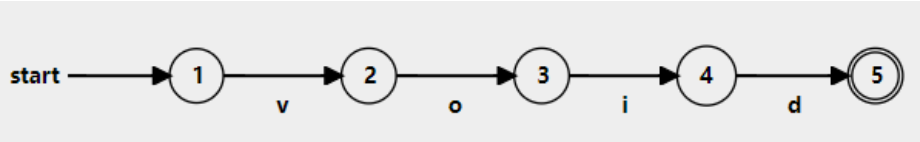
NFA:



DFA:

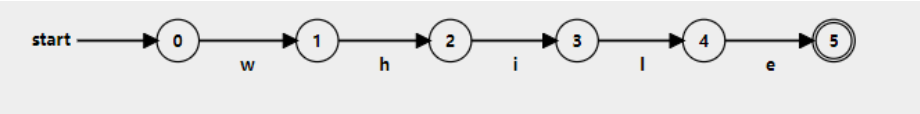


DFA mínimo:

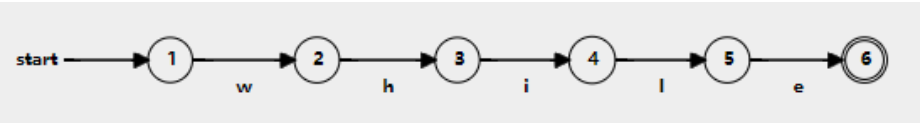


- **while**

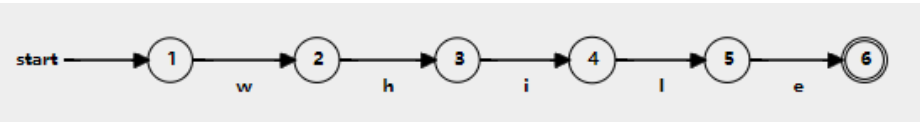
NFA:



DFA:

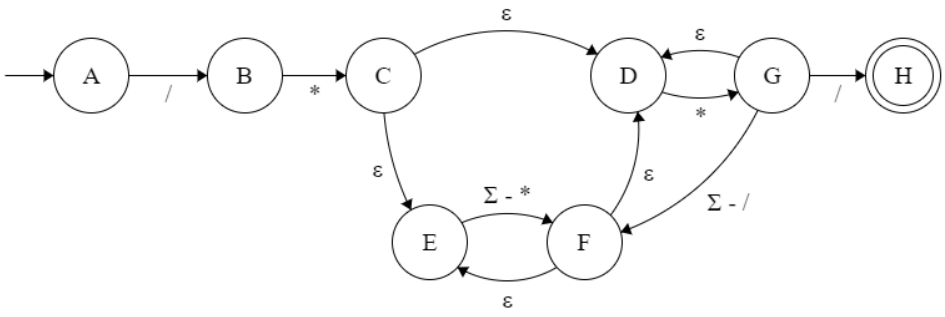


DFA mínimo:

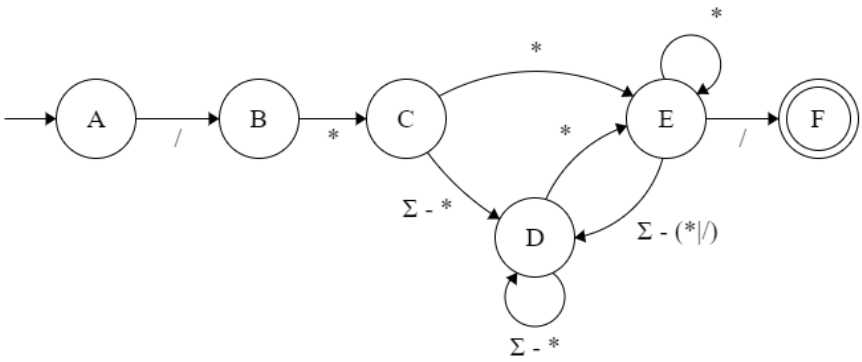


- **COMMENT**

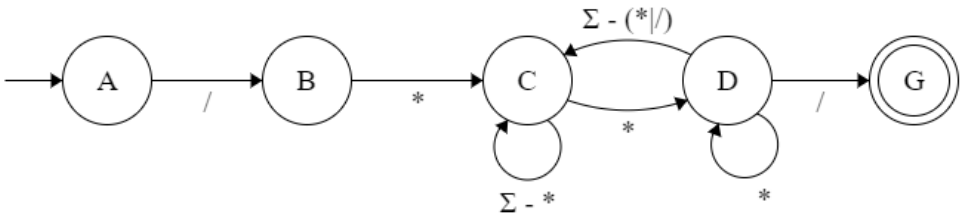
NFA:



DFA:



DFA mínimo:



- **LE:**

NFA:

DFA:

DFA mínimo:

- **GE:**

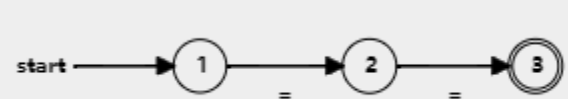
NFA:

DFA:

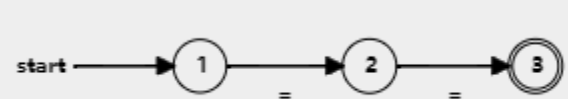
DFA mínimo:

- **EQ:**

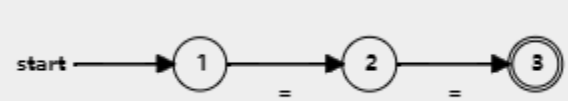
  NFA:



  DFA:

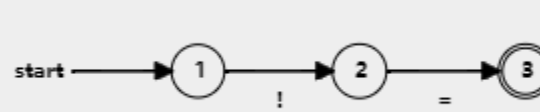


  DFA mínimo:



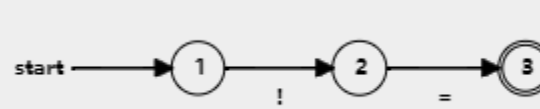
- **NE:**

  NFA:



  DFA:

  DFA mínimo:



- **LETTER**

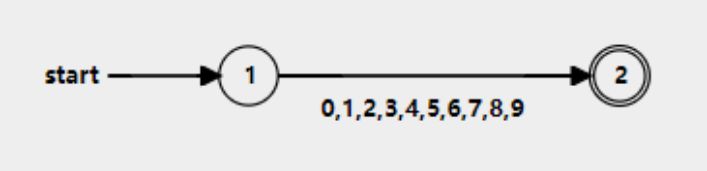
  NFA e DFA com muitos estados para demonstrar no desenho

DFA mínimo:

- **DIGIT**

NFA e DFA com muitos estados para demonstrar no desenho

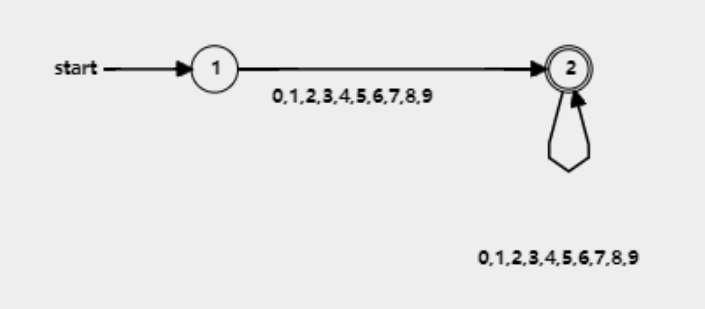
DFA mínimo:



- **NUM**

NFA e DFA com muitos estados para demonstrar no desenho

DFA mínimo:



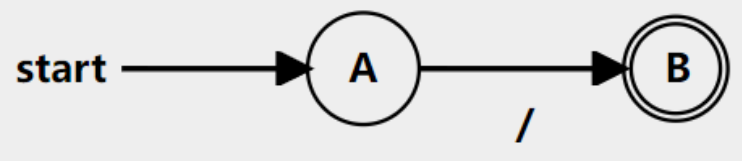
- **ID**

NFA e DFA com muitos estados para demonstrar no desenho

DFA mínimo:

- **PLUS**, **MINUS**, **ASTERISK**, **SLASH**, **LT**, **GT**, **ASSIGN**, **SEMICOLON**, **COMMA**, **OPENPARENTHESIS**, **CLOSEPARENTHESIS**, **OPENSQUAREBRACKETS**, **CLOSESQUAREBRACKETS**, **OPENBRACKETS**, **CLOSEBRACKETS**

  NFA, DFA e DFA mínimo:



Análogo aos demais tokens de apenas um símbolo...

7. Verificação se a Linguagem Escolhida é LL(1)
Enviada em arquivo anexo.

8. Analisador Sintático.

8.1 - Objeto SyNode

A classe SyNode (*Syntactic Tree Node*) representa um nó de uma árvore de análise sintática. Suas instâncias contém os seguintes atributos:

- symbol: A produção da gramática livre de contexto associada ao nó. Se o nó for um terminal, então necessariamente o nó é uma folha e terá o atributo “symbol” igual a nulo.

- token: O token produzido pelo analisador léxico associado ao nó. Se o nó for um não-terminal, então o nó terá o atributo “token” igual a nulo.

- level: Um inteiro representando a profundidade do nó, com o valor zero reservado para a raiz da árvore.

- parent: Referência ao nó pai. No contexto de análise sintática, representa a produção da gramática que levou à produção do nó atual.

- children: Lista de nós filhos.

8.2 - Objeto SyParser

A classe SyParser (*Syntactic Parser*) representa a estrutura que faz a análise sintática da sequência de tokens passada pelo analisador léxico e constrói sua árvore sintática. Suas instâncias contém os seguintes atributos:

- entrada: A lista de tokens passada pelo analisador léxico. Cada token contém as propriedades: “value”, “line”, “column” e “tags”, sendo “value” a string associada ao token, “line” e “column” referentes à posição do token no código fonte da linguagem, e “tags” referente ao token de fato.

- posicao_token_atual: Índice referente a posição na sequência de tokens passada na entrada. Essa propriedade serviu como estrutura auxiliar na construção da árvore sintática, visto que com esse índice conseguimos percorrer a sequência de tokens sem necessariamente adicionar o nó referente a cada produção explorada na árvore sintática, quando temos certeza que esta produção é a correta, adicionamos o nó referente a ela na arvore.

O parser implementado é um parser descendente recursivo que funciona da seguinte forma:

Primeiramente, é passado por parâmetro a sequência de tokens resultante do analisador léxico e definido inicialmente a posição token atual como 0, ao chamar a função parse, o parser chama a produção inicial “Program” que cria o nó raiz da árvore sintática e chama o método associado à derivação dessa produção na gramática.

Cada produção da gramática possui seu próprio método na classe “SyParser”, que recebe por parâmetro o nó pai referente a produção que levou a essa derivação chamada, e nesses métodos é onde são feitas as chamadas às outras produções presentes na derivação, caso existam, e/ou feito o “match” com do token na posição atual da sequência. Caso a derivação dessa produção esteja correta, o nó associado a essa produção é adicionado na árvore como filho do nó passado por parâmetro para o

método da produção. Caso a derivação esteja incorreta mas à mais derivações possíveis para determinada produção, voltamos a posição do token atual para a posição no início da produção e removemos os possíveis nós filhos gerados pela derivação incorreta. Ao final é retornado True à produção caso a derivação esteja correta ou False caso todas as derivações possíveis para aquela produção estejam incorretas, para esse caso também é retornada uma mensagem de erro apontando o token problemático e sua posição (linha e coluna) no código fonte da linguagem. Há também a verificação de declaração de identificadores, que guarda em uma lista os identificadores declarados ao longo do programa, e verifica antes de dar o “match” com o token “ID” se o identificador referenciado já foi declarado e, senão, retorna uma mensagem de erro apontando o identificador não declarado.

Ao fim do processamento do parser é retornada a árvore sintática da sequência de tokens de entrada.

9. Exemplos de Códigos-Fonte e Execução

9.1. Código sem Erros

```
int main(int i, int j)
{
    return i == j;
}
```

Saída do Analisador Léxico:

```
ln 1, cl 1 : int    ['int']
ln 1, cl 5 : main   ['id']
ln 1, cl 9 : (      ['open_parenthesis']
ln 1, cl 10: int    ['int']
ln 1, cl 14: i      ['id']
ln 1, cl 15: ,      ['comma']
ln 1, cl 17: int    ['int']
ln 1, cl 21: j      ['id']
ln 1, cl 22: )      ['close_parenthesis']
ln 2, cl 1 : {      ['open_brackets']
ln 3, cl 5 : return ['return']
ln 3, cl 12: i      ['id']
ln 3, cl 14: ==     ['eq']
ln 3, cl 17: j      ['id']
ln 3, cl 18: ;      ['semicolon']
ln 4, cl 1 : }      ['close_brackets']
```

Saída do Analisador Sintático:

```
PROGRAM
| DECLARATION_LIST
| | DECLARATION
| | | FUN_DECLARATION
| | | | TYPE_SPECIFIER
| | | | | int
| | | | id
| | | | open_parenthesis
| | | | PARAMS
| | | | | PARAM_LIST
| | | | | | PARAM
| | | | | | | TYPE_SPECIFIER
| | | | | | | | int
| | | | | | | id
| | | | | | PARAM_LIST_LINHA
| | | | | | | comma
| | | | | | | PARAM
| | | | | | | | TYPE_SPECIFIER
| | | | | | | | | int
| | | | | | | | id
| | | | close_parenthesis
```

```
| | | | COMPOUND_STMT
| | | | | open_brackets
| | | | | STATEMENT_LIST
| | | | | | STATEMENT_LIST_LINHA
| | | | | | | STATEMENT
| | | | | | | | RETURN_STMT
| | | | | | | | | return
| | | | | | | | | EXPRESSION_STMT
| | | | | | | | | | EXPRESSION
| | | | | | | | | | | SIMPLE_EXPRESSION
| | | | | | | | | | | | ADDITIVE_EXPRESSION
| | | | | | | | | | | | | TERM
| | | | | | | | | | | | | | FACTOR
| | | | | | | | | | | | | | | VAR
| | | | | | | | | | | | | | | | id
| | | | | | | | | | | | RELOP
| | | | | | | | | | | | | eq
| | | | | | | | | | | | ADDITIVE_EXPRESSION
| | | | | | | | | | | | | TERM
| | | | | | | | | | | | | | FACTOR
| | | | | | | | | | | | | | | VAR
| | | | | | | | | | | | | | | | id
| | | | | | | | | semicolon
| | | | | | STATEMENT_LIST
| | | | | close_brackets
```

Saída do Lex & Yacc:

```
ln 1: int - INT
ln 1: main - ID
ln 1: ( - OPENPARENTHESIS
ln 1: int - INT
ln 1: i - ID
ln 1: , - COMMA
ln 1: int - INT
ln 1: j - ID
ln 1: ) - CLOSEPARENTHESIS

ln 2: { - OPENBRACKETS

ln 3: return - RETURN
ln 3: i - ID
ln 3: == - EQ
ln 3: j - ID
ln 3: ; - SEMICOLON

ln 4: } - CLOSEBRACKETS



OK!
```

9.2. Código com Caractere Inesperado

```
int main(int ç, int j)
{
    return i == j;
}
```

Saída do Analisador Léxico:

```
Traceback (most recent call last):
   File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\compiler.py", line 84, in
<module>
    tokens = scanner.scan(code)
   File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\scanner.py", line 68, in
scan
    raise ValueError(
ValueError: Unexpected symbol Ã at line 1, column 14!
```

Saída do Analisador Sintático:

N/A

Saída do Lex & Yacc:

```
ln 1: int - INT
ln 1: main - ID
ln 1: ( - OPENPARENTHESIS
ln 1: int - INT
ln 1: ├ - ERROR
ln 1: º - ERROR
ln 1: , - COMMA
ln 1: int - INT
ln 1: j - ID
ln 1: ) - CLOSEPARENTHESIS

ln 2: { - OPENBRACKETS

ln 3: return - RETURN
ln 3: i - ID
ln 3: == - EQ
ln 3: j - ID
ln 3: ; - SEMICOLON

ln 4: } - CLOSEBRACKETS
syntax error
```

9.3. Código com Variável Referenciada Não Declarada

```
int main(int i, int j)
{
    return i == k;
}
```

Saída do Analisador Léxico:

```
ln 1, cl 1 : int    ['int']
ln 1, cl 5 : main   ['id']
ln 1, cl 9 : (      ['open_parenthesis']
ln 1, cl 10: int    ['int']
ln 1, cl 14: i      ['id']
ln 1, cl 15: ,      ['comma']
ln 1, cl 17: int    ['int']
ln 1, cl 21: j      ['id']
ln 1, cl 22: )      ['close_parenthesis']
ln 2, cl 1 : {      ['open_brackets']
ln 3, cl 5 : return ['return']
ln 3, cl 12: i      ['id']
ln 3, cl 14: ==     ['eq']
ln 3, cl 17: k      ['id']
ln 3, cl 18: ;      ['semicolon']
ln 4, cl 1 : }      ['close_brackets']
```

Saída do Analisador Sintático:

```
Traceback (most recent call last):
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\compiler.py", line 103, in
<module>
    tree = parser.parse()
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 66, in
parse
    return self.program()
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 71, in
program
    self.declaration_list(raiz)
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 84, in
declaration_list
    if (self.declaration(novo_no)):
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 105, in
declaration
    if (self.fun_declaration(novo_no)):
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 176, in
fun_declaration
    if (self.compound_statement(novo_no)):
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 257, in
compound_statement
    if (self.statement_list(novo_no)):
```

```
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 296, in
statement_list
    if (self.statement_list_linha(novo_no)):
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 308, in
statement_list_linha
    if (self.statement(novo_no)):
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 351, in
statement
    if (self.return_statement(novo_no)):
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 421, in
return_statement
    if (self.expression_statement(novo_no)):
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 363, in
expression_statement
    if (self.expression(novo_no)):
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 442, in
expression
    if (self.simple_expression(novo_no)):
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 475, in
simple_expression
    if (self.additive_expression(novo_no)):
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 521, in
additive_expression
    if (self.term(novo_no)):
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 562, in
term
    if (self.factor(novo_no)):
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 613, in
factor
    if (self.var(novo_no)):
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 453, in
var
     if (self.match_identificador_declarado() and self.match_terminal(parent=novo_no,
expectedTokenTag="id")):
   File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 59, in
match_identificador_declarado
    raise SyParserDeclarationException(self.entrada[self.posicao_token_atual])
syParser.SyParserDeclarationException: Identificador não declarado k - linha: 3 ,
coluna: 17
```

Saída do Lex & Yacc:

```
ln 1: int - INT
ln 1: main - ID
ln 1: ( - OPENPARENTHESIS
ln 1: int - INT
ln 1: i - ID
ln 1: , - COMMA
ln 1: int - INT
```

```
ln 1: j - ID
ln 1: ) - CLOSEPARENTHESIS

ln 2: { - OPENBRACKETS

ln 3: return - RETURN
ln 3: i - ID
ln 3: == - EQ
ln 3: k - ID
ln 3: ; - SEMICOLON

ln 4: } - CLOSEBRACKETS



OK!
```

9.4. Código com Erro Sintático

```
int main(int i, int j)
{
    return i + i = j;
}
```

Saída do Analisador Léxico:

```
ln 1, cl 1 : int    ['int']
ln 1, cl 5 : main   ['id']
ln 1, cl 9 : (      ['open_parenthesis']
ln 1, cl 10: int    ['int']
ln 1, cl 14: i      ['id']
ln 1, cl 15: ,      ['comma']
ln 1, cl 17: int    ['int']
ln 1, cl 21: j      ['id']
ln 1, cl 22: )      ['close_parenthesis']
ln 2, cl 1 : {      ['open_brackets']
ln 3, cl 5 : return ['return']
ln 3, cl 12: i      ['id']
ln 3, cl 14: +      ['plus']
ln 3, cl 16: i      ['id']
ln 3, cl 18: =      ['assign']
ln 3, cl 20: j      ['id']
ln 3, cl 21: ;      ['semicolon']
ln 4, cl 1 : }      ['close_brackets']
```

Saída do Analisador Sintático:

```
Traceback (most recent call last):
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\compiler.py", line 103, in
<module>
    tree = parser.parse()
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 66, in
parse
    return self.program()
  File "C:\Users\tiago\Documents\Git\c-minus-compiler\src\syParser.py", line 74, in
program
    raise SyParserTokenException(
syParser.SyParserTokenException: Erro sintatico - linha: 3 , coluna: 18
 Token problemático: =
```

Saída do Lex & Yacc:

```
ln 1: int - INT
ln 1: main - ID
ln 1: ( - OPENPARENTHESIS
ln 1: int - INT
ln 1: i - ID
```

```
ln 1: , - COMMA
ln 1: int - INT
ln 1: j - ID
ln 1: ) - CLOSEPARENTHESIS

ln 2: { - OPENBRACKETS

ln 3: return - RETURN
ln 3: i - ID
ln 3: + - PLUS
ln 3: i - ID
ln 3: = - ASSIGN
ln 3: j - ID
ln 3: ; - SEMICOLON

ln 4: } - CLOSEBRACKETS


syntax error
```

10. Análise Comparativa com Lex & Yacc

Na nossa implementação do analisador léxico nós construímos autômatos finitos para cada tipo de token e em seguida fizemos a união e minimização destes. Através do autômato gerado, realizamos a análise léxica de uma string de entrada, correspondente ao código fonte na linguagem escolhida (C-minus). A saída do analisador léxico é uma lista de tokens, contendo seu tipo, valor, linha e coluna correspondentes ao seu local no código-fonte (string) original. Também é feito tratamento de erros para diferentes tipos de erros léxicos.

Já no scanner construído com o Lex, são especificadas expressões regulares correspondentes aos tokens da linguagem, e ações a serem executadas quando o Lex encontra uma cadeia de caracteres correspondente a esta E.R. Internamente, através destas especificações, é gerado um autômato finito correspondente, responsável pela tokenização da cadeia de caracteres de entrada. Nesta, o único tratamento de erros realizado é a identificação de caracteres inesperados (caracteres fora do alfabeto da linguagem), enquanto na implementação em python, além deste, é possível encontrar o tratamento de erros de ambiguidade e de impossibilidade de tokenização de uma cadeia.

O parser construído pelo Yacc é um parser ascendente (*bottom-up*) LALR(1), ou seja, ele usa a estratégia de derivação mais à direita com “lookahead” de 1 token. Ele recebe como entrada os tokens retornados pela função yylex() do scanner e sua saída é uma mensagem indicando se a sequência de tokens é válida na gramática. No entanto,

a gramática correspondente à linguagem C-minus não é LALR(1), o que significa que há casos onde o analisador sintático não é capaz de decidir entre avançar ou reduzir.

Já na nossa implementação do analisador sintático, a estratégia de parsing é descendente (*top-down*) recursiva, de forma que, após a eliminação de recursões à esquerda, é realizada uma busca em profundidade, a partir da produção inicial da gramática, de forma a encontrar as derivações que resultem numa árvore sintática cujas folhas, da esquerda para a direita, sejam equivalentes à lista de tokens de entrada. Nesta implementação é possível identificar os tokens problemáticos e identificadores não declarados, enquanto na construção do parser pelo Yacc não foi possível fazer a identificação dos tokens que causam erros sintáticos.