Felipe de Paiva Miranda 7630486
Thiago Ryu Niwa Murakami 7626689

Questões

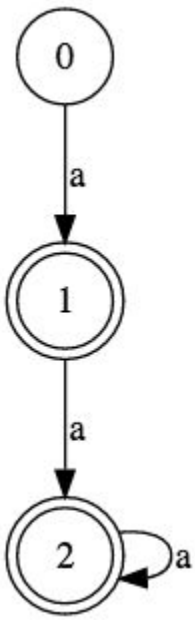
1) Um analisador léxico tem como entrada o código-fonte e deve traduzir essa cadeia de caracteres em símbolos léxicos, ou *tokens*, que são trechos elementares completos e com identidade própria. É função do analisador léxico também, classificar esses tokens segundo o tipo, como identificadores, palavras reservadas, números, cadeias de caracteres, sinais de pontuação, etc e extrair corretamente os valores associados a eles, como por exemplo, realizar conversões numéricas. Além de extrair e classificar, o analisador deve eliminar delimitadores, como espaços em branco, e comentários, pois estes não são utilizados na geração do código.

2) Ao implementar o analisador léxico como subrotina do analisador sintático, viabiliza-se uma comunicação entre esses dois módulos, elimina a necessidade de um arquivo de saída por parte do analisador léxico, já que os tokens são consumidos à medida que são gerados. Uma parte importante desta conexão é a tabela de símbolos que é populada pelo analisador léxico e consumida pelo analisador sintático. Uma desvantagem de se implementar o analisador léxico como uma sub-rotina é que a capacidade de paralelizar as etapas fica prejudicada, pois é criada uma dependência lógica entre a geração de tokens e o consumo dos mesmos.

3) <números>: 0-9
<alfabeto>: a-z|A-Z
<operador>: ‘=’ | ’>’ | ‘!’ | ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘^’ | ‘&’ | ‘|’
<delimitador>: ‘{’ | ‘}’ | ‘[’ | ‘]’ | ‘(’ | ‘)’ | ‘,’ | ‘;’
<inteiros>: <números><números>*
<decimais>: <inteiros>’.’<inteiros>
<string literal>: ‘“‘(<alfabeto>|<números>)*’”’
<identificadores>: <alfabeto>(<alfabeto>|<números>)*
<operadores>: <operador>(<operador>|ε)

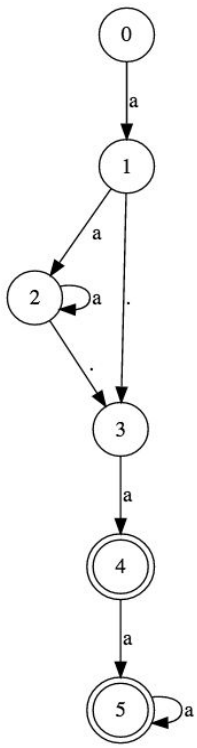
4)

Inteiros:



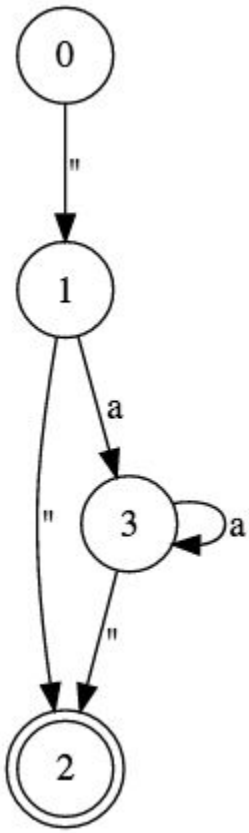
onde a = caracteres de 0 a 9.

Decimais:



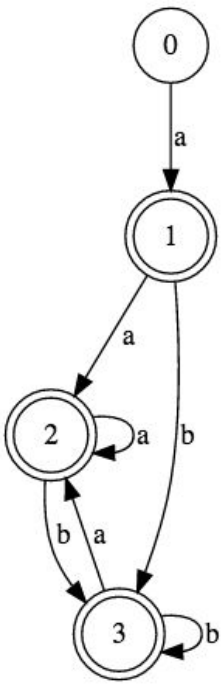
onde a = caracteres de 0 a 9.

String literal:



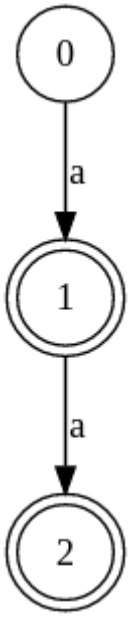
onde a é qualquer caractere alfanumérico.

Identificador:



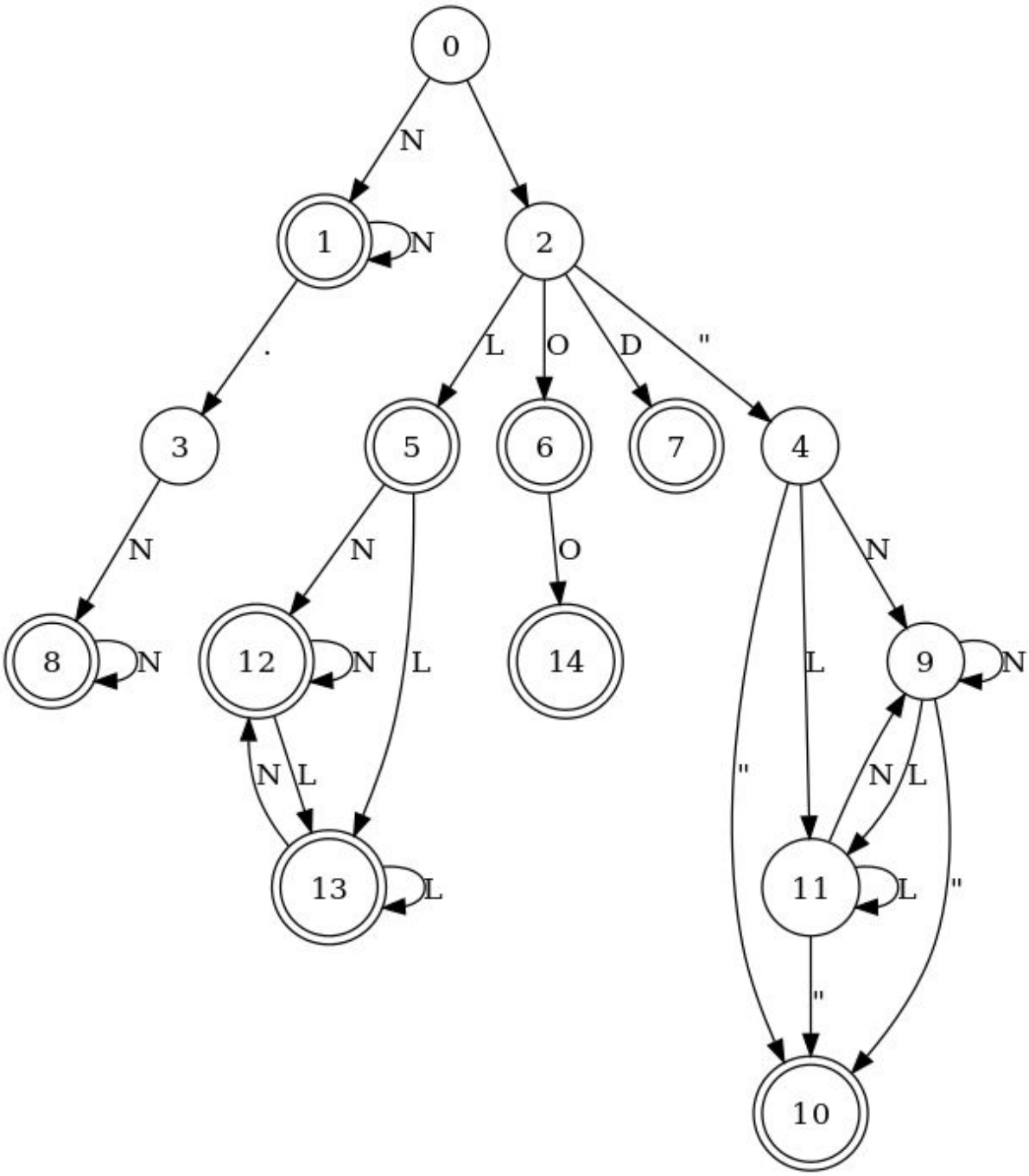
onde a = a-z|A-Z e b = 0-9

Operadores:



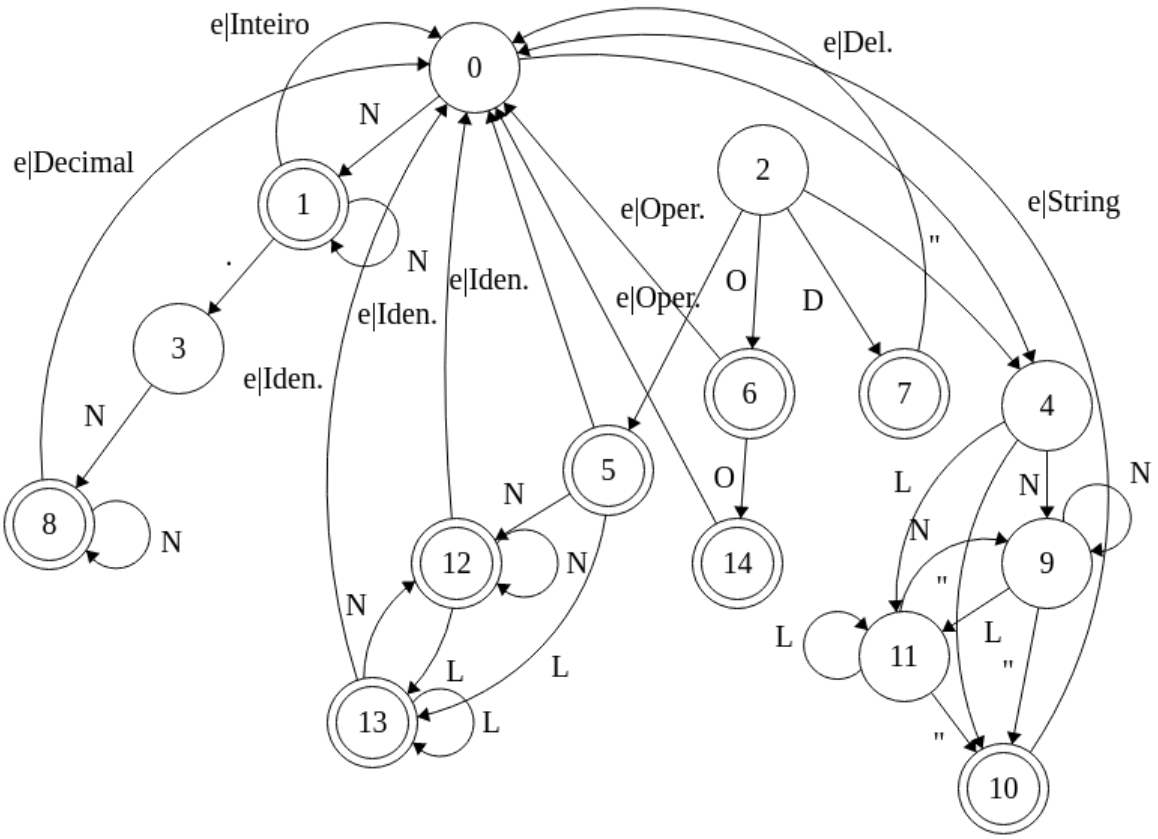
onde a = ‘=’ | ’>’ | ‘!’ | ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘^’ | ‘&’ | ‘|’

5)



onde N = 0 a 9.
onde L = a-z|A-Z
onde O = ‘=’ | ’>’ | ‘!’ | ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘^’ | ‘&’ | ‘|’
onde D = ‘{’ | ‘}’ | ‘[’ | ‘]’ | ‘(’ | ‘)’ | ‘,’ | ‘;’

6)



onde N = 0 a 9.
onde L = a-z|A-Z
onde O = ‘=’ | ’>’ | ‘!’ | ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘^’ | ‘&’ | ‘|’
onde D = ‘{’ | ‘}’ | ‘[’ | ‘]’ | ‘(’ | ‘)’ | ‘,’ | ‘;’

7) Vide Função nextToken() do arquivo *lex.c* anexo.
8) Vide Função main() do *main.c* anexo.
9)

O Programa foi escrito na linguagem de programação C e consiste de 4 partes:

- O arquivo principal *main.c* (que contém a função main())
- O arquivo *lex.c* e seu header *lex.h* (que contém a função nextToken())
- O arquivo *token.c* e seu header *token.h* (que contém as informações referentes a um token)
- O arquivo *util.c* e seu header *util.h* (que contém funções auxiliares)

A maior parte do analisador léxico encontra-se no arquivo *lex.c*, e sua única função exposta é a função nextToken(). Esta função recebe um ponteiro para um arquivo que contém o código fonte, e retorna um inteiro (com informações de erro) e um Token.

Esta função realiza os seguintes passos:

1. Lê um caracter do arquivo
2. Classifica esse caracter de acordo com seu tipo (Número, letra, operador, etc)

3. Muda de estado de acordo com a tabela gerada a partir do autômato da questão 6.
4. Acumula o valor lido em um buffer quando necessário (ex: string literal)
5. Checa se é estado final, e repete a partir de 1 caso não.
6. Volta um caracter na leitura do arquivo.
7. Gera um Token a partir com o valor do buffer.

O programa principal, contido em main.c, recebe o endereço de um arquivo e lê e imprime todos os tokens deste até o seu final.

Os arquivos token.* contêm o código para alocação e desalocação de um Token, assim como suas estruturas e enumerados.

Os arquivos util.* contêm códigos comuns, como verificação de números, delimitadores etc.

O arquivo de teste utilizado foi o lex_tex.txt, que é reproduzido aqui:

```
int func(int integer) {
        string z;
        z = "literal example";
        int a;
        a = 10;
        float b;
        b = 0.5;
        if(a == b){
    return a;
        }
        return integer;
}
```

E a saída esperada e obtida pelo analisador léxico foi:

```
Classe do Token: Palavra Reservada:     Valor do Token: int
Classe do Token: Identificador          Valor do Token: func
Classe do Token: Delimitador            Valor do Token: (
Classe do Token: Palavra Reservada:     Valor do Token: int
Classe do Token: Identificador          Valor do Token: integer
Classe do Token: Delimitador            Valor do Token: )
Classe do Token: Delimitador            Valor do Token: {
Classe do Token: Palavra Reservada:     Valor do Token: string
Classe do Token: Identificador          Valor do Token: z
Classe do Token: Delimitador            Valor do Token: ;
Classe do Token: Identificador          Valor do Token: z
Classe do Token: Operador               Valor do Token: =
Classe do Token: String Literal         Valor do Token: literal example
Classe do Token: Delimitador            Valor do Token: ;
Classe do Token: Palavra Reservada:     Valor do Token: int
Classe do Token: Identificador          Valor do Token: a
Classe do Token: Delimitador            Valor do Token: ;
Classe do Token: Identificador          Valor do Token: a
```

Classe do Token: Operador	*Valor do Token: =*
Classe do Token: Inteiro	*Valor do Token: 10*
Classe do Token: Delimitador	*Valor do Token: ;*
Classe do Token: Palavra Reservada:	*Valor do Token: float*
Classe do Token: Identificador	*Valor do Token: b*
Classe do Token: Delimitador	*Valor do Token: ;*
Classe do Token: Identificador	*Valor do Token: b*
Classe do Token: Operador	*Valor do Token: =*
Classe do Token: Decimal	*Valor do Token: 0.500000*
Classe do Token: Delimitador	*Valor do Token: ;*
Classe do Token: Palavra Reservada:	*Valor do Token: if*
Classe do Token: Delimitador	*Valor do Token: (*
Classe do Token: Identificador	*Valor do Token: a*
Classe do Token: Operador	*Valor do Token: ==*
Classe do Token: Identificador	*Valor do Token: b*
Classe do Token: Delimitador	*Valor do Token:)*
Classe do Token: Delimitador	*Valor do Token: {*
Classe do Token: Palavra Reservada:	*Valor do Token: return*
Classe do Token: Identificador	*Valor do Token: a*
Classe do Token: Delimitador	*Valor do Token: ;*
Classe do Token: Delimitador	*Valor do Token: }*
Classe do Token: Palavra Reservada:	*Valor do Token: return*
Classe do Token: Identificador	*Valor do Token: integer*
Classe do Token: Delimitador	*Valor do Token: ;*
Classe do Token: Delimitador	*Valor do Token: }*