ESCOLA POLITÉNICA □DA UNIVERSIDADE DE SÃO PAULO

Analisador Lexico

Alunos: Rafael Camargo Leite – 7629953
Vinicius Correa - 7631553

1. Quais são as funções do analisador léxico nos compiladores/interpretadores?

De maneira geral, o Analisador léxico é responsável pela leitura e extração de tokens do código fonte.

Tokens são definidos como:

```
typedef struct token_t {
	token_class class;
	union value {
		int i_value;
		float f_value;
	} value;
} token_t;
```

onde token_class é definido como:

```
typedef enum {
	CLASS_INT, // int number
	CLASS_FLOAT, // float number
	CLASS_RESERVED_WORD, // if, while, int, ...
	CLASS_IDENTIFIER, // variable name
	CLASS_SINGLE_OPERATOR, // '=', '>', '<', '!', '+', '-', '*', '/'
	CLASS_DOUBLE_OPERATOR, // "==", ">=", "<=", "!="
	CLASS_DELIMITER, //  '{', '}', '[', ']', ',', ';', ' ', '\t'
} token_class;
```

Durante sua execução algumas tarefas são executadas:

1. Remoção de delimitadores(ex: espaços em branco) e comentários
2. Expansão de Macros (#define) e pré-processamento(#ifndef etc..)
3. Armazenamento de linha em coluna dos tokens para prover mensagens de erro ao programador
4. Conversões numéricas diversas
5. Inserções na tabela de símbolos

Na implementação adotada, o único item que não foi desenvolvido foi a expansão de macros pois, inicialmente, foi feita a opção de remover essa feature da linguagem.

2. Quais as vantagens e desvantagens da implementação do analisador léxico como uma fase separada do processamento da linguagem de programação em relação à sua implementação como sub-rotina que vai extraindo um átomo a cada chamada?

Como fase separada, o léxico precisa apenas do arquivo de entrada com o código fonte e gera um arquivo de saída com os átomos. Sendo ele uma sub-rotina utilizada pelo analisador sintático, não há necessidade de arquivo de saída, átomos são consumidos conforme produzidos, o que permite adoção de esquema de compilação em um único passo.

Então temos como vantagem a abstração do analisador sintático para com o arquivo fonte. Caso suficientemente abstraído o analisador léxico pode ser simples, rápido e gerar linguagens livres de contexto.

Como desvantagem para a implementação do analisador léxico como um módulo separado podemos citar a relação intrínseca entre eles. Quanto maior a abstração do analisador léxico, maior a necessidade de incluir lógica no analisador sintático. O que poderia ser reduzido caso o analisador léxico a priori tivesse algum conhecimento sobre as necessidades do analisador sintático.

3. Defina formalmente, através de expressões regulares sobre o conjunto de caracteres ASCII, a sintaxe de cada um dos tipos de átomos a serem extraídos do texto-fonte pelo analisador léxico, bem como de cada um dos espaçadores e comentários.

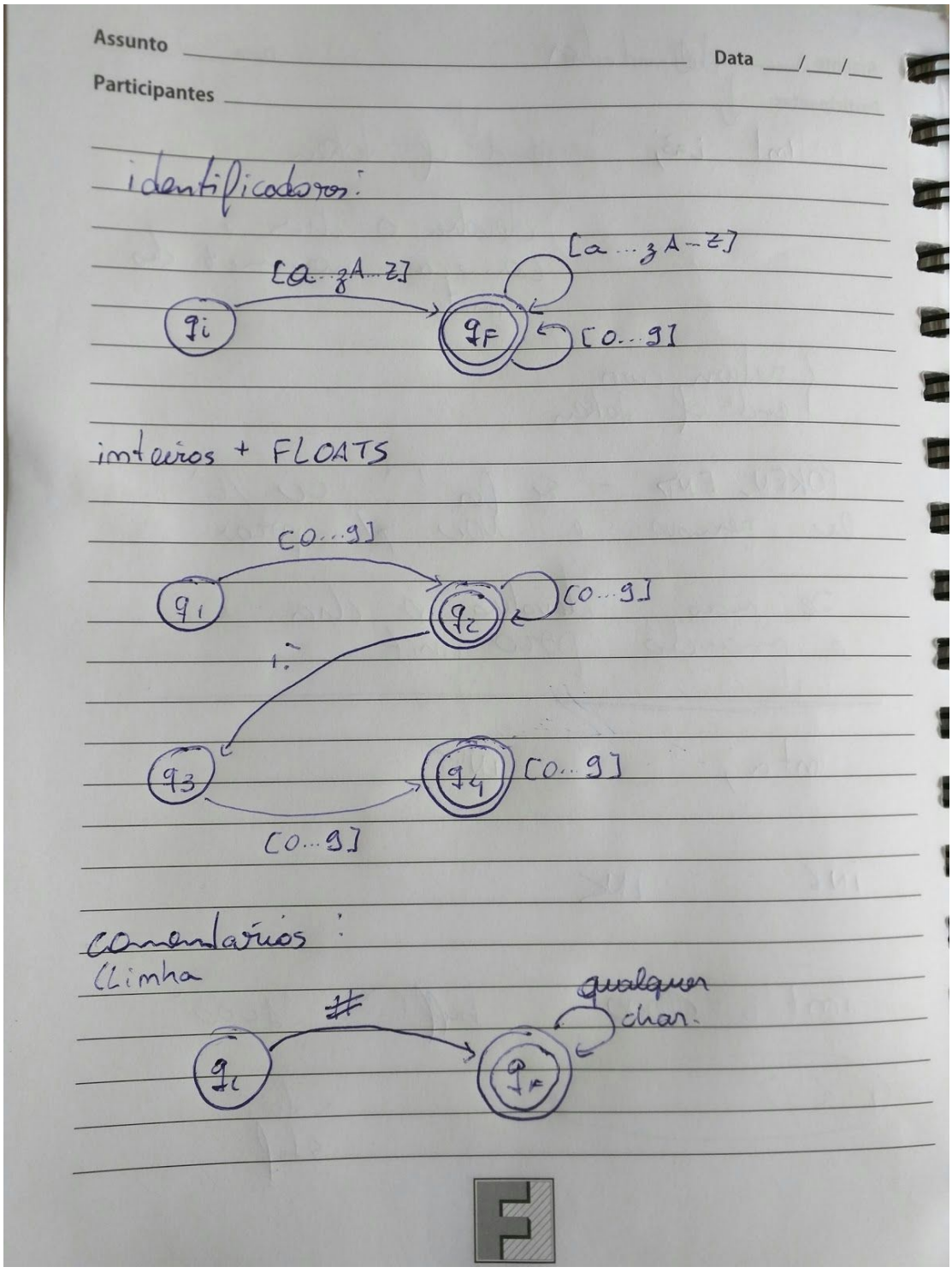
- Reserved words:
  - if
  - else
  - while
  - int
  - float
  - return
  - const
  - break
  - continue

- Single char operators:
  - =
  - >
  - <
  - !
  - +
  - -
  - *
  - ^
  - /
  - &
  - |

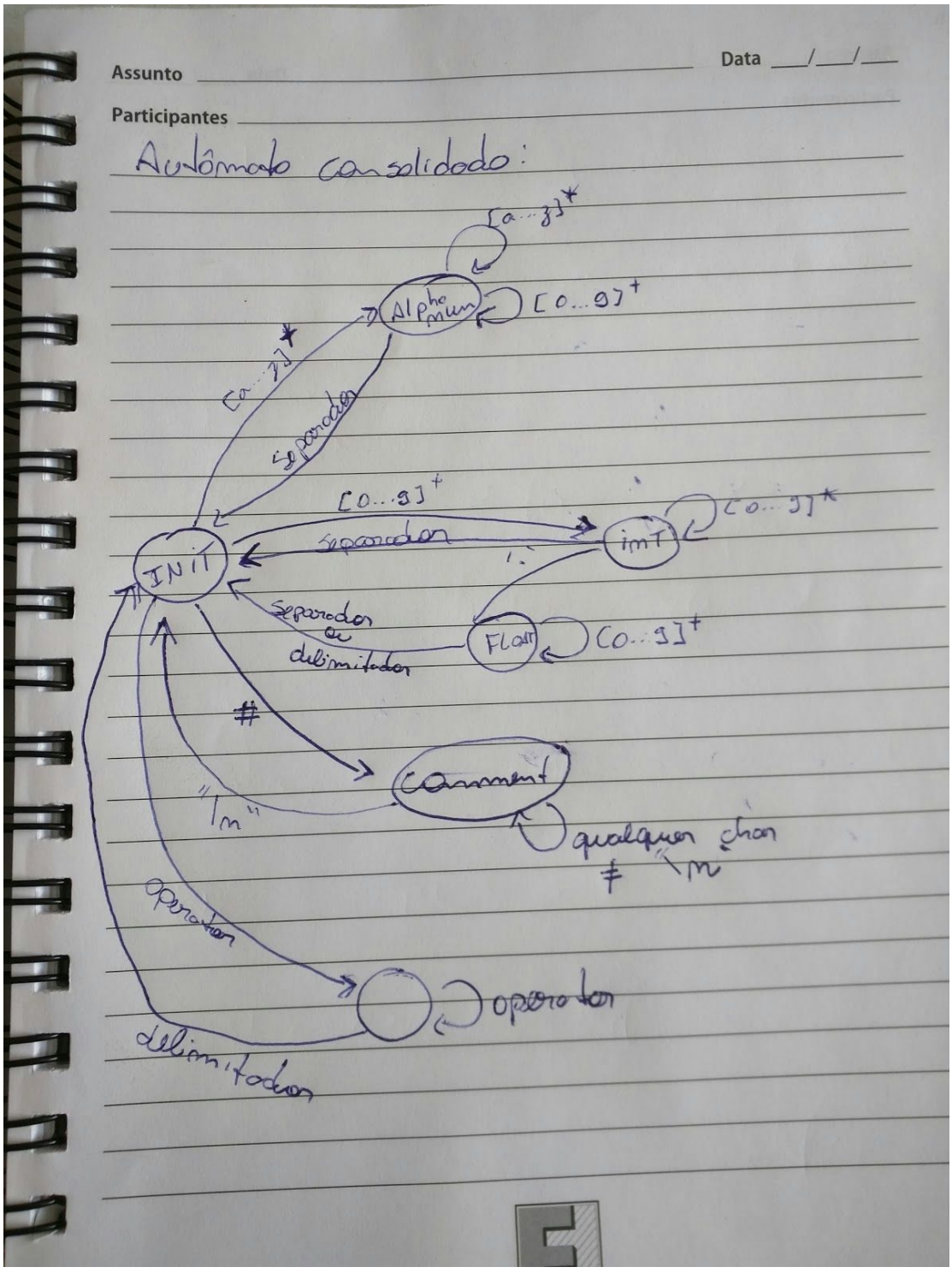
- Double char operators:
  - ==
  - >=
  - <=
  - !=
  - &&
  - ||

- Átomos Especiais:
  - Identificadores
    - [a...z]+ | [0...9]* / [a...z]*
  - Inteiros
    - [0...9]+
  - Flutuantes
    - [0...9]+ | [0...9]+ '.' [0...9]+
  - comentários
    - # [0...9]* / [a...z]+

4. Converta cada uma das expressões regulares, assim obtidas, em autômatos finitos equivalentes que reconheçam as correspondentes linguagens por elas definidas.



5. Crie um autômato único que aceite todas essas linguagens a partir de um mesmo estado inicial, mas que apresente um estado final diferenciado para cada uma delas.



6. Transforme o autômato assim obtido em um transdutor, que emita como saída o átomo encontrado ao abandonar cada um dos estados finais para iniciar o reconhecimento de mais um átomo do texto.

O transdutor é exatamente o mesmo da figura anterior. A única diferença é que no retorno para o estado INIT ocorre a devolução do Token encontrado.

7. Converta o transdutor assim obtido em uma sub-rotina, escrita na linguagem de programação de sua preferência. Não se esqueça que o final de cada átomo é determinado ao ser encontrado o primeiro símbolo do átomo ou do espaçador seguinte. Esse símbolo não pode ser perdido, devendo-se, portanto, tomar os cuidados de programação que forem necessários para reprocessá-los, apesar de já terem sido lidos pelo autômato.

Código nos arquivos em Anexo

8. Crie um programa principal que chame repetidamente a sub-rotina assim construída, e a aplique sobre um arquivo do tipo texto contendo o texto-fonte a ser analisado. Após cada chamada, esse programa principal deve imprimir os dois componentes do átomo extraído (o tipo e o valor do átomo encontrado). Faça o programa parar quando o programa principal receber do analisador léxico um átomo especial indicativo da ausência de novos
átomos no texto de entrada.

Código nos arquivos em Anexo

9. Relate detalhadamente o funcionamento do analisador léxico assim construído, incluindo no relatório: descrição teórica do programa; descrição da sua estrutura; descrição de seu funcionamento; descrição dos testes realizados e das saídas obtidas.

O projeto foi desenvolvido basicamente em 2 módulos:

1. token.c / token.h
2. lex.c / lex.h

Em token.c / token.h foi feita a definição da estrutura do token e suas funcões associadas. Um token é representado da seguinte maneira:

```
typedef struct token_t {
	token_class class;
	union value {
		int i_value;
		float f_value;
	} value;
} token_t;
```

Seu valor foi codificado como uma Union. Sendo assim, pode assumir um dos dois valores mostrados.
Caso o token seja um inteiro literal, seu valor é colocado em i_value.
Caso o token seja um float literal, seu valor é colocado em f_value.
Nos demais casos, i_value é preenchido com o índice da tabela específica que guarda o valor do token.

token_class é definido da seguinte maneira:

```
typedef enum {
```

```
        CLASS_INT, // int number
        CLASS_FLOAT, // float number
        CLASS_RESERVED_WORD, // if, while, int, ...
        CLASS_IDENTIFIER, // variable name
        CLASS_SINGLE_OPERATOR, // '=', '>', '<', '!', '+', '-', '*', '/'
        CLASS_DOUBLE_OPERATOR, // "==", ">=", "<=", "!="
        CLASS_DELIMITER, //  '{', '}', '[', ']', ',', ';', ' ', '\t'
} token_class;
```

Em lex.c foram definidas as tabelas necessárias ao analisador.
Elas são:

```
/*
 *      Symbol table
 */
unsigned symbol_table_ptr = 0;
char symbol_table[BUFFER_SIZE][SYMBOL_TABLE_SIZE];

void print_symbol_table() {
        unsigned i;
        printf("Printing Symbol table: \n");
        for(i = 0; i < symbol_table_ptr; i++) {
                printf("index: %u -> %s\n", i, symbol_table[i]);
        }
}

/*
 *      Reserved words
 */
#define RESERVED_WORDS_SIZE 9
const char* const reserved_words[] = { "if", "else", "while", "int", "float",
                "return", "const" "break", "continue" };

/*
 *      Single operators
 */
#define SINGLE_OPERATORS_SIZE 11
const char* const single_operators[] = {"=", ">", "<", "!", "+", "-", "*", "/",
"^", "&", "|" };

/*
 *      Double operators
 */
#define DOUBLE_OPERATORS_SIZE 6
const char* const double_operators[] = {"==", ">=", "<=", "!=", "&&", "||" };

/*
 *      delimiters
 */
#define DELIMITERS_SIZE 11
const char delimiters[DELIMITERS_SIZE] = { '{', '}', '[', ']', '(', ')', ',', ';',
' ', '\n', '\t' };

/*
 *      comment begin
 */
const char commentary = '#';
```

Essas são as tabelas referenciadas pelo ‘value’ do token.

Finalmente, de maneira geral, o códido em lex.c representa um automato por meio de tabelas de transição de estado.
O Screenshot abaixo representa a tabela de transição de estados:

```
 *
 *          digit   alpha   operator    DELIMITER   COMM_INIT   DOT
 *  INIT
 *  COMMENT
 *  N_INT
 *  FLOAT
 *  ALPHAN
 *  OP
 *  DELIM
 *  LEX_ERROR
 */
const state_t next_state[STATES_SIZE][IN_CLASS_SIZE] = {
        { ST_NUM_INT,   ST_APLHANUM,    ST_OPERATOR,    ST_DELIMITER,   ST_COMMENT, ST_LEX_ERROR },
        { ST_COMMENT,   ST_COMMENT, ST_COMMENT, ST_COMMENT, ST_COMMENT, ST_COMMENT },
        { ST_NUM_INT,   ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END,   ST_NUM_FLOAT },
        { ST_NUM_FLOAT, ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END,   ST_LEX_ERROR },
        { ST_APLHANUM,  ST_APLHANUM,    ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END,   ST_LEX_ERROR },
        { ST_TOKEN_END, ST_TOKEN_END,   ST_OPERATOR,    ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END },
        { ST_TOKEN_END, ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END },
        { ST_LEX_ERROR, ST_LEX_ERROR,   ST_LEX_ERROR,   ST_LEX_ERROR,   ST_LEX_ERROR,   ST_LEX_ERROR }
};
```

A função principal possui um loop que começa no estado inicial e é executado até atingit o estado ST_TOKEN_END
O código abaixo é o desta função:

```
token_t* get_token(FILE *fp) {
        state_struct_t state_struct;
        create_state_struct(&state_struct, fp);

        /*
         *      Main loop
         */
        do {
                /*
                 *      Get next input
                 */
                state_struct.curr_input = getc(fp);
                if(state_struct.curr_input == EOF) {
                        break;
                }

                /*
                 *      update state_struct
                 */
                state_struct.input_class =
classify_input_class(state_struct.curr_input);


state_function[*state_struct.curr_state][state_struct.input_class](&state_struct);
        } while(*state_struct.curr_state != ST_TOKEN_END);

        /*
         *      Build token to be returned
         */
        if(*state_struct.buffer_ptr != 0){
                printf(">> buffer: %s\n", state_struct.buffer);
                build_token(&state_struct);
        }

        /*
         *      Free dynamic allocated memory and return
         */
        destroy_state_struct(&state_struct);

        return state_struct.token;
}
```

Sendo assim, o programa funcionou como o esperado.
O Arquivo de teste utilizado foi:

```
#comentario bla bla
int main() {
        int a= 1.2;
        if (a>= 3) {
                int b = 4;
        }
        #outro comentario bla bla
        while(1) {
                const float c =4.3
        }
}
```

O output de execução foi o seguinte:

```
>> buffer: int
Printing Symbol table:
token_class: CLASS_RESERVED_WORD
token value: 3

>> buffer: main
Printing Symbol table:
index: 0 -> main
token_class: CLASS_IDENTIFIER
token value: 0

>> buffer: (
Printing Symbol table:
index: 0 -> main
token_class: CLASS_DELIMITER
token value: 4

>> buffer: )
Printing Symbol table:
index: 0 -> main
token_class: CLASS_DELIMITER
token value: 5

>> buffer: {
Printing Symbol table:
index: 0 -> main
token_class: CLASS_DELIMITER
token value: 0

>> buffer: int
Printing Symbol table:
index: 0 -> main
token_class: CLASS_RESERVED_WORD
token value: 3

>> buffer: a
Printing Symbol table:
index: 0 -> main
index: 1 -> a
token_class: CLASS_IDENTIFIER
token value: 1
```

```
>> buffer: =
Printing Symbol table:
index: 0 -> main
index: 1 -> a
token_class: CLASS_SINGLE_OPERATOR
token value: 0

>> buffer: 1.2
Printing Symbol table:
index: 0 -> main
index: 1 -> a
token_class: CLASS_FLOAT
token_value: 1.200000

>> buffer: ;
Printing Symbol table:
index: 0 -> main
index: 1 -> a
token_class: CLASS_DELIMITER
token value: 7

>> buffer: if
Printing Symbol table:
index: 0 -> main
index: 1 -> a
token_class: CLASS_RESERVED_WORD
token value: 0

>> buffer: (
Printing Symbol table:
index: 0 -> main
index: 1 -> a
token_class: CLASS_DELIMITER
token value: 4

>> buffer: a
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
token_class: CLASS_IDENTIFIER
token value: 2

>> buffer: >=
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
token_class: CLASS_DOUBLE_OPERATOR
token value: 1

>> buffer: 3
Printing Symbol table:
index: 0 -> main
```

index: 1 -> a
index: 2 -> a
token_class: CLASS_INT
token value: 3

>> buffer:)
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
token_class: CLASS_DELIMITER
token value: 5

>> buffer: {
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
token_class: CLASS_DELIMITER
token value: 0

>> buffer: int
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
token_class: CLASS_RESERVED_WORD
token value: 3

>> buffer: b
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
token_class: CLASS_IDENTIFIER
token value: 3

>> buffer: =
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
token_class: CLASS_SINGLE_OPERATOR
token value: 0

>> buffer: 4
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
token_class: CLASS_INT

token value: 4

>> buffer: ;
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
token_class: CLASS_DELIMITER
token value: 7

>> buffer: }
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
token_class: CLASS_DELIMITER
token value: 1

>> buffer: while
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
token_class: CLASS_RESERVED_WORD
token value: 2

>> buffer: (
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
token_class: CLASS_DELIMITER
token value: 4

>> buffer: 1
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
token_class: CLASS_INT
token value: 1

>> buffer:)
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
token_class: CLASS_DELIMITER

token value: 5

>> buffer: {
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
token_class: CLASS_DELIMITER
token value: 0

>> buffer: const
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
token_class: CLASS_RESERVED_WORD
token value: 6

>> buffer: float
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
token_class: CLASS_RESERVED_WORD
token value: 4

>> buffer: c
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
token_class: CLASS_IDENTIFIER
token value: 4

>> buffer: =
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
token_class: CLASS_SINGLE_OPERATOR
token value: 0

>> buffer: 4.3
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a

index: 3 -> b
index: 4 -> c
token_class: CLASS_FLOAT
token_value: 4.300000

>> buffer: }
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
token_class: CLASS_DELIMITER
token value: 1

>> buffer: if
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
token_class: CLASS_RESERVED_WORD
token value: 0

>> buffer: (
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
token_class: CLASS_DELIMITER
token value: 4

>> buffer: (
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
token_class: CLASS_DELIMITER
token value: 4

>> buffer: a
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
index: 5 -> a
token_class: CLASS_IDENTIFIER

token value: 5

>> buffer: ||
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
index: 5 -> a
token_class: CLASS_DOUBLE_OPERATOR
token value: 5

>> buffer: b
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
index: 5 -> a
index: 6 -> b
token_class: CLASS_IDENTIFIER
token value: 6

>> buffer:)
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
index: 5 -> a
index: 6 -> b
token_class: CLASS_DELIMITER
token value: 5

>> buffer: ^
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
index: 5 -> a
index: 6 -> b
token_class: CLASS_SINGLE_OPERATOR
token value: 8

>> buffer: (
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a

index: 3 -> b
index: 4 -> c
index: 5 -> a
index: 6 -> b
token_class: CLASS_DELIMITER
token value: 4

>> buffer: a
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
index: 5 -> a
index: 6 -> b
index: 7 -> a
token_class: CLASS_IDENTIFIER
token value: 7

>> buffer: &&
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
index: 5 -> a
index: 6 -> b
index: 7 -> a
token_class: CLASS_DOUBLE_OPERATOR
token value: 4

>> buffer: b
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
index: 5 -> a
index: 6 -> b
index: 7 -> a
index: 8 -> b
token_class: CLASS_IDENTIFIER
token value: 8

>> buffer:)
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c

index: 5 -> a
index: 6 -> b
index: 7 -> a
index: 8 -> b
token_class: CLASS_DELIMITER
token value: 5

>> buffer:)
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
index: 5 -> a
index: 6 -> b
index: 7 -> a
index: 8 -> b
token_class: CLASS_DELIMITER
token value: 5

>> buffer: {
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
index: 5 -> a
index: 6 -> b
index: 7 -> a
index: 8 -> b
token_class: CLASS_DELIMITER
token value: 0

>> buffer: return
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
index: 5 -> a
index: 6 -> b
index: 7 -> a
index: 8 -> b
token_class: CLASS_RESERVED_WORD
token value: 5

>> buffer: a
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a

index: 3 -> b
index: 4 -> c
index: 5 -> a
index: 6 -> b
index: 7 -> a
index: 8 -> b
index: 9 -> a
token_class: CLASS_IDENTIFIER
token value: 9

>> buffer: ;
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
index: 5 -> a
index: 6 -> b
index: 7 -> a
index: 8 -> b
index: 9 -> a
token_class: CLASS_DELIMITER
token value: 7

>> buffer: }
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
index: 5 -> a
index: 6 -> b
index: 7 -> a
index: 8 -> b
index: 9 -> a
token_class: CLASS_DELIMITER
token value: 1

>> buffer: }
Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
index: 5 -> a
index: 6 -> b
index: 7 -> a
index: 8 -> b
index: 9 -> a
token_class: CLASS_DELIMITER
token value: 1

Printing Symbol table:
index: 0 -> main
index: 1 -> a
index: 2 -> a
index: 3 -> b
index: 4 -> c
index: 5 -> a
index: 6 -> b
index: 7 -> a
index: 8 -> b
index: 9 -> a

>> FINISHED!

O Código completo está na pasta junto com este documento.
Caso necessário, o código também está disponível no github em:
https://github.com/rcmgleite/lexical_analyzer

Fontes:
http://disciplinas.stoa.usp.br/pluginfile.php/366697/mod_resource/content/1/Introducao_a_Compilacao_Unicode_Encoding_Conflict_.pdf
http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/03-Lexical-Analysis.pdf
http://www.cs.nyu.edu/courses/spring11/G22.2130-001/lecture4.pdf
http://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm
https://www.youtube.com/watch?v=IQWfPVwJLF8