

PCS2056

Linguagens e Compiladores

Compilador - Primeira etapa:
Analizador Léxico

Professor: Ricardo Luis de Azevedo Rocha

Grupo:

Filipe Morgado Simões de Campos	5694101
---------------------------------	---------

Rafael Barbolo Lopes	5691262
----------------------	---------

Assunto: Análise léxica;
Enunciado da parte 1 do trabalho: Construção de um analisador léxico
Data de entrega: 28 de setembro de 2010

Palavras-chave:

Análise léxica	comentários
texto-fonte	token ou átomo = (classe, valor)
caracteres ASCII	expressões regulares
identificadores	autômatos reconhecedores
inteiros sem sinal	transdutores sintáticos
caracteres especiais	listagem do texto-fonte
espaçadores	expansão de macros
palavras reservadas	

Questões:

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

O analisador léxico é um módulo responsável pela leitura do(s) arquivo(s) onde o código fonte está localizado; ele realiza o *parsing* desse arquivo, extraíndo o que são chamados de *tokens* (átomos).

O analisador léxico lê cada um dos caracteres do código fonte e realiza agrupamento deles em *tokens*, que são classificados em tipos de informações importantes do programa que deve ser compilado/interpretado (um *token* pode ser, por exemplo, uma palavra reservada ou um identificador de variável). Durante esse processo, o analisador pode realizar conversões desses *tokens* (por exemplo, conversão para tipo numérico), e também elimina caracteres que não fazem parte do programa (espaços em branco, comentários, etc).

Também é papel do analisador léxico relatar erros durante sua execução, tratar macros, e criar e preencher estruturas de dado que guardam referência de localização de cada *token* no código fonte.

Vale comentar que o analisador léxico deve ser muito eficiente, pois tem grandes chances de ser o maior gargalo de um compilador/interpretador, pois é requisitado milhares de vezes por vários módulos do compilador e efetua operações de entrada/saída durante o processo de compilação.

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?

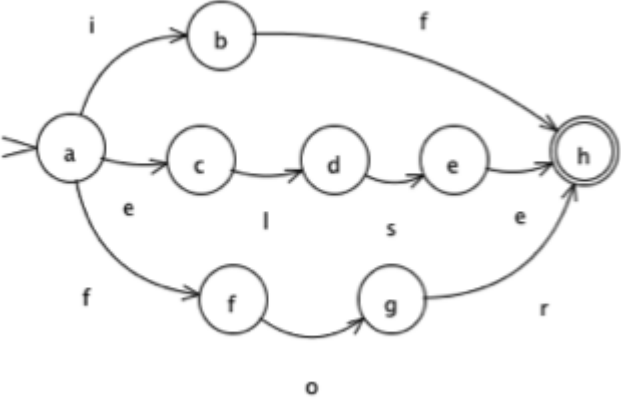
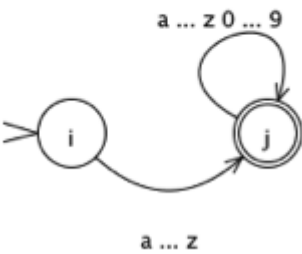
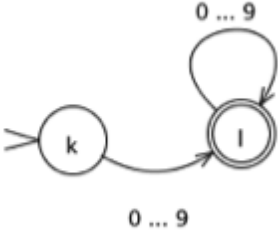

Quando o analisador léxico opera como um passo inicial do compilador, o código-fonte é o seu texto de entrada, e um arquivo com a cadeia de átomos extraídos forma a saída do programa (que será a entrada do analisador sintático). Nesta implementação, a vantagem é que o analisador léxico está bem isolado das outras fases de compilação, o que torna mais fácil seu entendimento e eventual manutenção. Porém, essa abordagem não tem o desempenho mais eficiente, já que o arquivo de *tokens* criado será navegado pelo menos duas vezes: durante sua criação pelo analisador léxico e durante sua leitura pelo analisador sintático. Para códigos-fonte muito grandes essa abordagem pode comprometer o tempo de compilação.

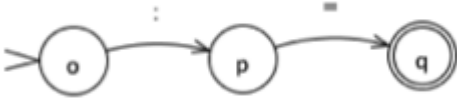
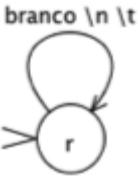
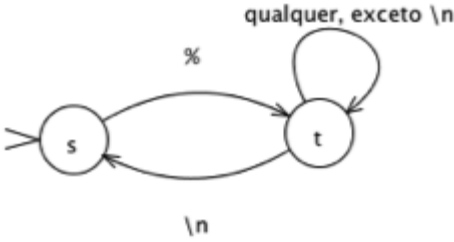

Quando o analisador léxico é implementado como subrotina, ele é requisitado, através de parâmetros e chamadas, pelo analisador sintático em cada vez que este necessita de um *token* adicional para continuar seu processamento. Para esta abordagem, não é necessária a criação de um arquivo de saída, já que o consumo dos *tokens* se dão concomitantemente com sua produção. Desta forma, é possível compilar o código fonte em um único passo, o que representa uma grande vantagem em desempenho computacional com relação à primeira abordagem. Uma desvantagem desta implementação é que se a arquitetura do compilador não estiver bem organizada, um programador pode se perder no código com facilidade.

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.

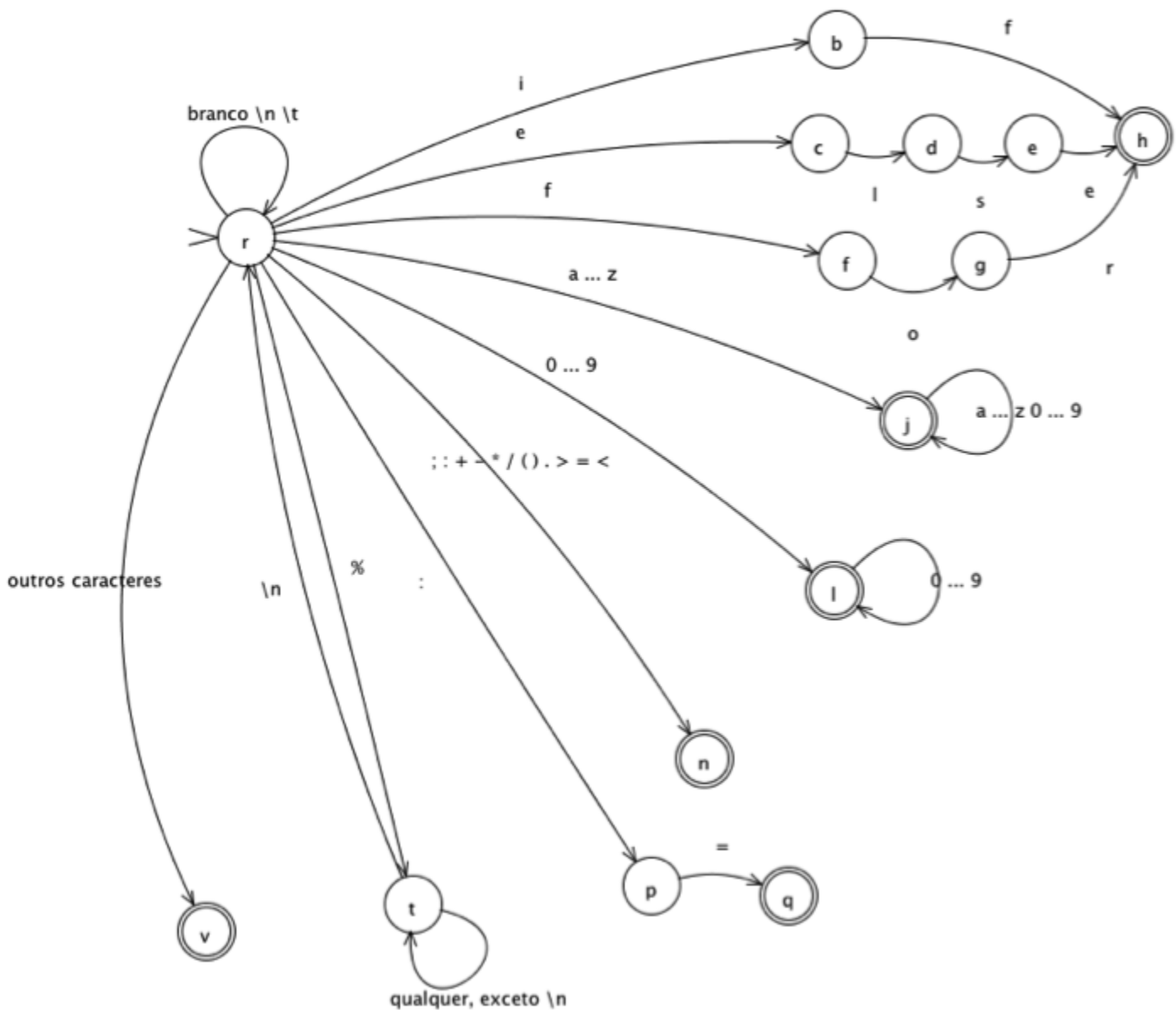
Tipo de Átomo	Expressão Regular
palavra reservada	if else for
identificador	[a-z][a-z0-9]*
número inteiro	[0-9][0-9]*
sinal de pontuação, operação ou separação	[;][:][+][-][*] V [(][)][\.\.][>][=][<]
sinal composto	\:\=
espaçador	(\n \t)*
comentário	%[^\n]*\n

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

Tipo de Átomo	Autômato Finito
palavra reservada	 <pre> graph LR a((a)) -- i --> b((b)) a -- e --> c((c)) a -- f --> f((f)) b -- f --> h(((h))) c -- l --> d((d)) d -- s --> e((e)) e -- e --> h f -- o --> g((g)) g -- r --> h style a fill:#fff,stroke:#000,stroke-width:1px style b fill:#fff,stroke:#000,stroke-width:1px style c fill:#fff,stroke:#000,stroke-width:1px style d fill:#fff,stroke:#000,stroke-width:1px style e fill:#fff,stroke:#000,stroke-width:1px style f fill:#fff,stroke:#000,stroke-width:1px style g fill:#fff,stroke:#000,stroke-width:1px style h fill:#fff,stroke:#000,stroke-width:1px,stroke-dasharray: 5 5 </pre>
identificador	 <pre> graph LR i((i)) -- "a ... z" --> j(((j))) j -- "a ... z 0 ... 9" --> j style i fill:#fff,stroke:#000,stroke-width:1px style j fill:#fff,stroke:#000,stroke-width:1px,stroke-dasharray: 5 5 </pre>
número inteiro	 <pre> graph LR k((k)) -- "0 ... 9" --> l(((l))) l -- "0 ... 9" --> l style k fill:#fff,stroke:#000,stroke-width:1px style l fill:#fff,stroke:#000,stroke-width:1px,stroke-dasharray: 5 5 </pre>
sinal de pontuação, operação ou separação	 <pre> graph LR m((m)) -- ";; + - * / () . > = <" --> n(((n))) style m fill:#fff,stroke:#000,stroke-width:1px style n fill:#fff,stroke:#000,stroke-width:1px,stroke-dasharray: 5 5 </pre>

sinal composto	 <pre> graph LR o((o)) -- ":" --> p((p)) p -- "=" --> q(((q))) </pre>
espaçador	 <pre> graph LR r(((r))) -- "branco \n \t" --> r </pre>
comentário	 <pre> graph LR s((s)) -- "%" --> t(((t))) t -- "\n" --> s t -- "qualquer, exceto \n" --> t </pre>
outros caracteres	 <pre> graph LR u((u)) -- "outros caracteres" --> v(((v))) </pre>

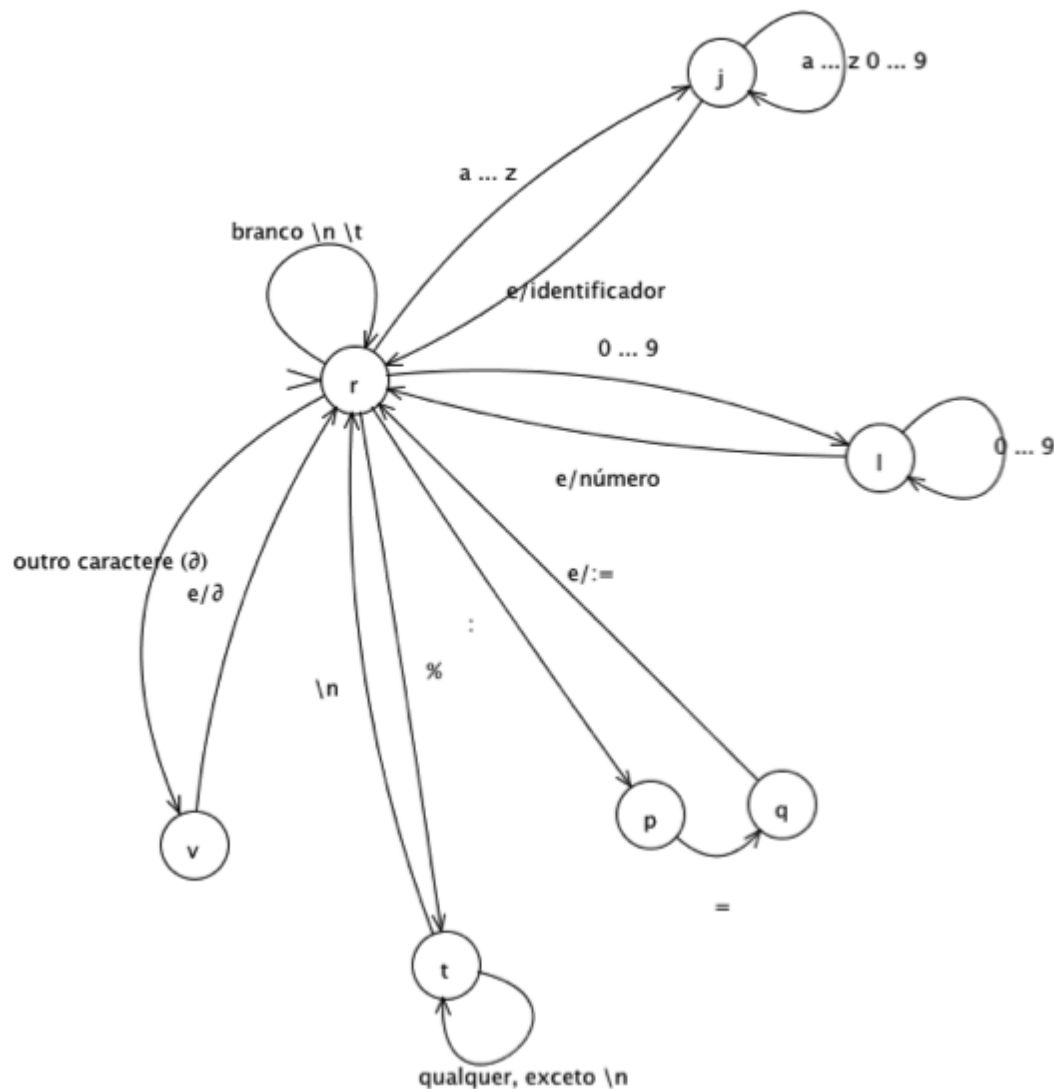
5. Crie um autômato único que aceite todas essas linguagens a partir de um mesmo estado inicial, mas que apresente um estado final diferenciado para cada uma delas.



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.

Para conseguir desenhar o autômato no software utilizado, foi necessário remover as transições que iam para o autômato que aceitava sinal de pontuação, operação ou separação e as que iam para o autômato que aceitava palavras reservadas. Apesar disso, pode-se entender

o raciocínio para criação do transdutor através do autômato final desenhado. O resultado pode ser visto abaixo:



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.

O transdutor foi abstraído como uma tabela de transições. Essa tabela possui 4 colunas: estado atual, entrada, próximo estado, saída. A implementação desta tabela foi feita através

de uma matriz de uma estrutura de dados que guarda próximo estado e saída. Desta forma, é possível acessar próximo estado e saída conhecendo apenas o estado atual e a entrada.

É importante deixar claro que a tabela de transições se apresentou como um meio para implementar o transdutor com flexibilidade. Como a linguagem de programação do código fonte ainda não foi definida, é importante que o código seja flexível para aceitar novas regras de transições do transdutor.

O transdutor é uma instância que tem um método que consome uma entrada e atualiza seu estado atual, retornando o tipo de token que está sendo lido pelo analisador léxico. Para que este método funcione, é necessário conhecer o caractere de entrada atual e o seguinte (*lookahead*), de maneira que transições com cadeia vazia sejam resolvidas e não sejam retornados resultados inválidos. Este método é chamado pelo analisador léxico enquanto está lendo um token e analisando seu tipo e valor.

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 as duas 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.

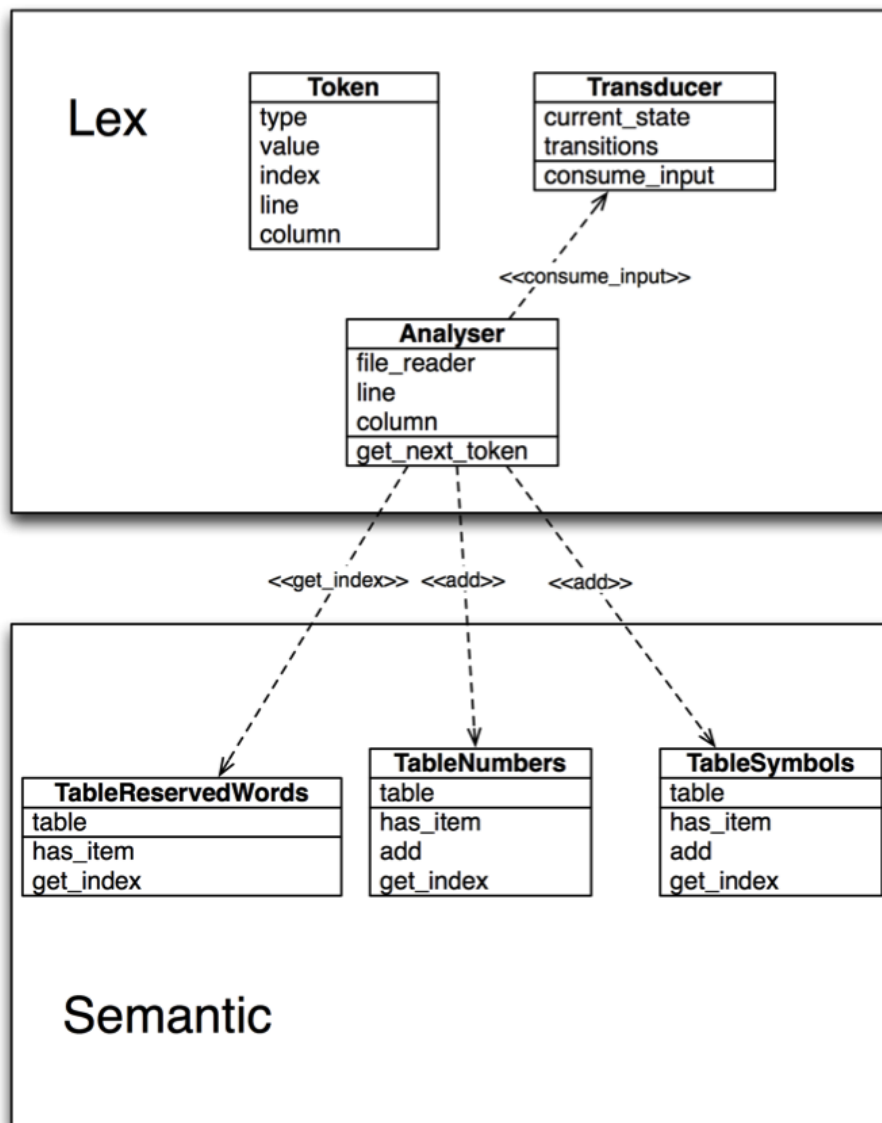
O programa analisador de tokens foi implementado. A essência dele é um laço que exclui caracteres espaçadores ou comentários e logo em seguida lê os caracteres até formar um token (quem garante que o token foi formado é o transdutor). Esse analisador deve guardar a linha e a coluna de cada caractere lido, de maneira que a posição de cada token no código fonte seja conhecida. O analisador também deve ser capaz de encontrar tokens inválidos para posterior depuração.

O programa principal usa o analisador para retornar todos os tokens do código fonte e imprimi-los um a um. Esse programa posteriormente poderá estar presente, por exemplo, no analisador sintático para implementação de um compilador de um único passo.

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 analisador léxico, bem como todo o compilador que será desenvolvido, foi implementado utilizando-se a linguagem de programação C. Após uma etapa inicial de estudos teóricos, que foram apresentados nas primeiras questões e portanto não serão reintroduzidos aqui, prosseguimos com o seu desenvolvimento e testes para validar seu funcionamento. Logo, cabe apresentar a estrutura do compilador até então, descrevendo seu funcionamento e os testes realizados.

O programa foi estruturado em dois grupos para essa fase do compilador. O primeiro dele, representa o analisador léxico e suas estruturas auxiliares que são o foco dessa atividade. Já o segundo, reserva espaço para a etapa futura, o analisador semântico, que pôde ser contemplado com algumas estruturas que também são utilizadas pelo analisador léxico. A interação entre esses módulos ficará mais clara no decorrer da descrição do funcionamento do sistema.



O primeiro grupo é composto por uma dupla de arquivos que representa um token (token.h e token.c), uma outra dupla para implementar o transdutor (transducer.h e transducer.c) e, por último, a dupla de arquivos que coordena o processo do analisador léxico (analyser.h e analyser.c). Seguem explicações detalhadas:

No arquivo `token.h`, Apêndice I, foi definida a estrutura de dados de um token e todos os tipos possíveis a que este poderia pertencer. Além dessa estrutura e dos tipos de tokens que são utilizados por todo o analisador léxico, foi declarada a variável global que representa o token no compilador.

Para o arquivo `token.c`, Apêndice II, restou a implementação da função `token_type_name` que é responsável por retornar o nome do tipo do token que está sendo processado pelo analisador léxico. Ela foi utilizada na impressão desse nome na saída do programa.

O arquivo `transducer.h`, Apêndice III, e o arquivo `transducer.c`, Apêndice IV, já tiveram seu funcionamento explicado na questão 8. Vale apenas acrescentar que seu funcionamento é solicitado pelo `analyser` através da função `transducer_consume_input` em que são passados os valores do caractere atual e do próximo (*lookahead*) e a função retorna o código do tipo do token com base na consulta da matriz `transducer_transitions` que representa o transdutor da questão 6.

O arquivo `analyser.h`, Apêndice V, contém apenas a declaração da função `get_next_token` que foi implementada no arquivo `analyser.c`, Apêndice VI. Essa função é responsável pelo funcionamento da parte do compilador responsável pela análise léxica como já foi explicado na questão 8. Durante esse procedimento, é feito o uso da função `read_next_char`, que retorna os caracteres do arquivo que contém o código fonte, e das estruturas anteriormente indicadas do grupo semântico para a consulta e adição de valores nas tabelas semânticas.

O segundo grupo, contém as tabelas semânticas que estão implementadas nos arquivos `tables.h`, Apêndice VII, e `tables.c`, Apêndice VIII. Elas são: tabela de símbolos, tabela de números, tabela de palavra reservadas e a tabela de caracteres especiais.

Além desses dois grupos, foram desenvolvidas algumas bibliotecas para o projeto. Dentre elas, encontra-se uma para a representação das tabelas semânticas, `hashtable.h`, Apêndice IX, e `hashtable.c`, Apêndice X.

Outra biblioteca foi criada para representar uma string, `strings.h`, Apêndice XI, e `strings.c`, Apêndice XII. Até agora, a utilizamos apenas para comparar se uma string é igual a outra através da função `strcmp`.

Por fim, a última biblioteca realiza a leitura dos caracteres dos arquivos, `reader.h`, Apêndice XIII, e `reader.c`, Apêndice XIV. Vale comentar que essa possui uma estrutura representando uma cabeça de leitura que possui os caracteres atual, anterior e futuro (*lookahead*), a posição (linha e coluna) do caractere atual e o ponteiro para o arquivo sendo lido.

Em resumo, o `main.c`, Apêndice XV, (já explicado na questão 8) por meio do `get_next_token` faz uso no `analyser` para identificar os tokens contidos nele. Esse, por sua vez, retira os caracteres do arquivo do código fonte e utiliza as informações presentes no `transducer`, através do `transducer_consume_input`, para decidir o tipo do token e ainda atualiza as tabelas de símbolos presentes no grupo semântico. O processo se repete até o fim do arquivo ser

encontrado.

Para consolidar o desenvolvimento dessa etapa do compilador, testamos a saída do programa para o seguinte código de entrada:

```
b
b
b == a = 10120 + 15;
> >=
< <=

3

&

bc = 10

d == ; 29123 7

10

10

15

bc

bc

=

for

else
```

A saída encontrada, que também era a esperada, foi:

Padrão da saída para cada token:

valor (índice na tabela semântica) :: nome do tipo (identificador do tipo) :: linha ,
coluna

Token:	b	(0)	::	identifier	(2)	::	line	1, column	1
Token:	b	(0)	::	identifier	(2)	::	line	2, column	1
Token:	b	(0)	::	identifier	(2)	::	line	3, column	1
Token:	==	(0)	::	special char	(4)	::	line	3, column	3
Token:	a	(1)	::	identifier	(2)	::	line	3, column	6

Token:	=	(1)	::	special char	(4)	::	line	3, column	8
Token:	10120	(0)	::	integer number	(3)	::	line	3, column	10
Token:	+	(2)	::	special char	(4)	::	line	3, column	16
Token:	15	(1)	::	integer number	(3)	::	line	3, column	18
Token:	;	(3)	::	special char	(4)	::	line	3, column	20
Token:	>	(4)	::	special char	(4)	::	line	4, column	1
Token:	>=	(5)	::	special char	(4)	::	line	4, column	3
Token:	<	(6)	::	special char	(4)	::	line	5, column	1
Token:	<=	(7)	::	special char	(4)	::	line	5, column	3
Token:	3	(2)	::	integer number	(3)	::	line	7, column	1
Token:	&	(-1)	::	invalid	(-1)	::	line	9, column	1
Token:	bc	(2)	::	identifier	(2)	::	line	11, column	1
Token:	=	(1)	::	special char	(4)	::	line	11, column	4
Token:	10	(3)	::	integer number	(3)	::	line	11, column	6
Token:	d	(3)	::	identifier	(2)	::	line	13, column	1
Token:	==	(0)	::	special char	(4)	::	line	13, column	3
Token:	;	(3)	::	special char	(4)	::	line	13, column	6
Token:	29123	(4)	::	integer number	(3)	::	line	13, column	8
Token:	7	(5)	::	integer number	(3)	::	line	13, column	14
Token:	10	(3)	::	integer number	(3)	::	line	15, column	1
Token:	10	(3)	::	integer number	(3)	::	line	17, column	1
Token:	15	(1)	::	integer number	(3)	::	line	19, column	1
Token:	bc	(2)	::	identifier	(2)	::	line	21, column	1
Token:	bc	(2)	::	identifier	(2)	::	line	23, column	1
Token:	=	(1)	::	special char	(4)	::	line	25, column	1
Token:	for	(1)	::	reserved word	(1)	::	line	27, column	1
Token:	else	(2)	::	reserved word	(1)	::	line	29, column	1

10.Explique como enriquecer esse analisador léxico com um expensor de macros do tipo #DEFINE, não paramétrico nem recursivo, mas que permita a qualquer macro chamar outras macros, de forma não cíclica. (O expensor de macros não precisa ser implementado).

Pode-se criar uma etapa de pré-processamento que, quando encontrado, no código fonte, um identificador igual ao que foi definido com o uso do *#define*, esse será substituído pelo valor especificado na macro.

Para que isso se torne realidade em nosso projeto, deve-se criar uma tabela no grupo semântico para que sejam guardados ao menos os identificadores e o valor a ser substituído especificados no *#define*. Assim, quando se estiver lendo o trecho de código em que esse identificador aparecer, o token que está sendo criado será identificado como correspondente a essa macro e então deve-se fazer uso de uma rotina, ainda a ser implementada, que testará se

essa macro chama outra, ou se já é possível fazer a substituição do valor, e então realizar essa troca.

Apêndice I - token.h

```
#ifndef TOKEN_H
#define TOKEN_H

/*
 * token.h
 * compiler
 *
 * Created by Filipe Morgado Simões de Campos e Rafael Barbolo Lopes on 24/09/10.
 *
 */

// ---- Token Types - start ----

#define TOKEN_TYPE_INVALID          -1
#define TOKEN_TYPE_INCOMPLETE      0
#define TOKEN_TYPE_RESERVED_WORD   1
#define TOKEN_TYPE_IDENTIFIER      2
#define TOKEN_TYPE_INT_NUMBER      3
#define TOKEN_TYPE_SPECIAL          4
#define TOKEN_TYPE_IGNORED          5    // blank char, \n, \t, comments
#define TOKEN_TYPE_END_OF_FILE      6    // this is a special token type, that
// belongs to the tokens returned when the file content has been all read

typedef int type_of_token;

// ---- Token Types - end ----

typedef struct token_type {
    type_of_token type;
    char *value;
    int index;
    int line;
    int column;
} Token;

/* global variable token */
Token token;
```

```
/******  
name: token_type_name  
purpose: Get the name of the token type.  
args:  
returns: The name of the token type.  
*****/  
char * token_type_name();
```

```
#endif
```

Apêndice II - token.c

```
/*  
 * token.c  
 * compiler  
 *  
 * Created by Filipe Morgado Simões de Campos e Rafael Barbolo Lopes on 24/09/10.  
 *  
 */
```

```
#include "token.h"
```

```
char * token_type_name() {  
  
    char * type_name;  
  
    switch ( token.type ) {  
        case -1:  
            type_name = "invalid";  
            break;  
        case 0:  
            type_name = "incomplete";  
            break;  
        case 1:  
            type_name = "reserved word";  
            break;  
        case 2:  
            type_name = "identifier";  
            break;  
        case 3:  
            type_name = "integer number";  
            break;  
        case 4:  
            type_name = "special char";  
            break;  
        case 5:  
            type_name = "ignored char";  
            break;  
        case 6:  
            type_name = "end of file";  
            break;  
        default:  
            type_name = "Wrong token type identifier.";
```



```
                break;
            }
    return type_name;
}
```

Apêndice III - transducer.h

```
#ifndef TRANSDUCER_H
#define TRANSDUCER_H

/*
 * transducer.h
 * compiler
 *
 * Created by Filipe Morgado Simões de Campos e Rafael Barbolo Lopes on 24/09/10.
 *
 * The transducer is an implementation of a finite automata.
 * It offers a function that receives the actual STATE of the automata and
 * the INPUT char, and it returns the NEXT_STATE and the OUTPUT.
 *
 * The OUTPUT may be a token type or an empty string.
 */

#include "token.h"

#define INVALID_STATE      -1
#define EMPTY_STRING      256

#define INITIAL_STATE      0
#define AVAILABLE_STATES   100

typedef int transducer_state;

typedef struct {
    transducer_state next_state;
    type_of_token type;
} automata_output;

transducer_state transducer_current_state = INITIAL_STATE;
int transducer_created_states = 1; // the initial state is always created

/* automata transitions */
automata_output transducer_transitions[AVAILABLE_STATES][257];
```

```
/******  
name: initialize_transitions  
purpose: initialize table of automata transitions;  
args:  
returns:  
*****/
```

```
void initialize_transitions();
```

```
/******  
name: transducer_new_state  
purpose: creates a new state for the automata and return it;  
args:  
returns: transducer_state (int).  
*****/
```

```
transducer_state transducer_new_state();
```

```
/******  
name: transducer_consume_input  
purpose: execute a transition in transducer automata.  
args: current and nex chars.  
returns: token type (int).  
*****/
```

```
type_of_token transducer_consume_input(char current, char lookahead);
```

```
#endif
```

Apêndice IV - transducer.c

```
/*
 * transducer.c
 * compiler
 *
 * Created by Filipe Morgado Simões de Campos e Rafael Barbolo Lopes on 24/09/10.
 *
 */

#include <stdio.h>
#include "transducer.h"
#include "token.h"

void initialize_transitions() {
    transducer_state s;
    int aux, aux2;

    /* default transition goes to an invalid state and returns an invalid token type */
    for (aux = 0; aux < 100; aux++) {
        for(aux2 = 0; aux2 < 256; aux2++) {
            transducer_transitions[aux][aux2].next_state = INVALID_STATE;
            transducer_transitions[aux][aux2].type = TOKEN_TYPE_INVALID;
        }
        transducer_transitions[aux][EMPTY_STRING].next_state = INVALID_STATE;
        transducer_transitions[aux][EMPTY_STRING].type = TOKEN_TYPE_INVALID;
    }

    /* transitions with spaces */
    transducer_transitions[0][(int)' '].next_state = 0;
    transducer_transitions[0][(int)' '].type = TOKEN_TYPE_IGNORED;

    transducer_transitions[0][(int)'\n'].next_state = 0;
    transducer_transitions[0][(int)'\n'].type = TOKEN_TYPE_IGNORED;

    transducer_transitions[0][(int)'\t'].next_state = 0;
    transducer_transitions[0][(int)'\t'].type = TOKEN_TYPE_IGNORED;

    /* transitions with comments */
    s = transducer_new_state();
    transducer_transitions[0][(int)'%'].next_state = s;
    transducer_transitions[0][(int)'%'].type = TOKEN_TYPE_IGNORED;
    for (aux = 0; aux < 256; aux++) {
```

```

        transducer_transitions[s][aux].next_state = s;
        transducer_transitions[s][aux].type = TOKEN_TYPE_IGNORED;
    }
    transducer_transitions[s][(int)'\n'].next_state = 0;
    transducer_transitions[s][(int)'\n'].type = TOKEN_TYPE_IGNORED;

/* transitions with identifiers */
s = transducer_new_state();
for (aux = (int)'a'; aux <= (int)'z'; aux++) {
    transducer_transitions[0][aux].next_state = s;
    transducer_transitions[0][aux].type = TOKEN_TYPE_INCOMPLETE;

    transducer_transitions[s][aux].next_state = s;
    transducer_transitions[s][aux].type = TOKEN_TYPE_INCOMPLETE;
}
for (aux = (int)'0'; aux <= (int)'9'; aux++) {
    transducer_transitions[s][aux].next_state = s;
    transducer_transitions[s][aux].type = TOKEN_TYPE_INCOMPLETE;
}
transducer_transitions[s][EMPTY_STRING].next_state = 0;
transducer_transitions[s][EMPTY_STRING].type = TOKEN_TYPE_IDENTIFIER;

/* transitions with integer numbers */
s = transducer_new_state();
for (aux = (int)'0'; aux <= (int)'9'; aux++) {
    transducer_transitions[0][aux].next_state = s;
    transducer_transitions[0][aux].type = TOKEN_TYPE_INCOMPLETE;

    transducer_transitions[s][aux].next_state = s;
    transducer_transitions[s][aux].type = TOKEN_TYPE_INCOMPLETE;
}
transducer_transitions[s][EMPTY_STRING].next_state = 0;
transducer_transitions[s][EMPTY_STRING].type = TOKEN_TYPE_INT_NUMBER;

/* transitions with special chars */
transducer_transitions[0][(int)':'].next_state = 0;
transducer_transitions[0][(int)':'].type = TOKEN_TYPE_SPECIAL;

transducer_transitions[0][(int)';'].next_state = 0;
transducer_transitions[0][(int)';'].type = TOKEN_TYPE_SPECIAL;

transducer_transitions[0][(int)'+'].next_state = 0;
transducer_transitions[0][(int)'+'].type = TOKEN_TYPE_SPECIAL;

```

```

transducer_transitions[0][(int)'-'].next_state = 0;
transducer_transitions[0][(int)'-'].type = TOKEN_TYPE_SPECIAL;

transducer_transitions[0][(int)'*'].next_state = 0;
transducer_transitions[0][(int)'*'].type = TOKEN_TYPE_SPECIAL;

transducer_transitions[0][(int)'/'].next_state = 0;
transducer_transitions[0][(int)'/'].type = TOKEN_TYPE_SPECIAL;

transducer_transitions[0][(int)'('].next_state = 0;
transducer_transitions[0][(int)'('].type = TOKEN_TYPE_SPECIAL;

transducer_transitions[0][(int)')'].next_state = 0;
transducer_transitions[0][(int)')'].type = TOKEN_TYPE_SPECIAL;

transducer_transitions[0][(int)'.'].next_state = 0;
transducer_transitions[0][(int)'.'].type = TOKEN_TYPE_SPECIAL;

/* nondeterministic special chars with dept <= 2 */
// ==
s = transducer_new_state();
transducer_transitions[0][(int)'='].next_state = s;
transducer_transitions[0][(int)'='].type = TOKEN_TYPE_INCOMPLETE;

transducer_transitions[s][(int)'='].next_state = 0;
transducer_transitions[s][(int)'='].type = TOKEN_TYPE_SPECIAL;

transducer_transitions[s][EMPTY_STRING].next_state = 0;
transducer_transitions[s][EMPTY_STRING].type = TOKEN_TYPE_SPECIAL;

// > >=
s = transducer_new_state();
transducer_transitions[0][(int) '>'].next_state = s;
transducer_transitions[0][(int) '>'].type = TOKEN_TYPE_INCOMPLETE;

transducer_transitions[s][(int)'>'].next_state = 0;
transducer_transitions[s][(int)'>'].type = TOKEN_TYPE_SPECIAL;

transducer_transitions[s][EMPTY_STRING].next_state = 0;
transducer_transitions[s][EMPTY_STRING].type = TOKEN_TYPE_SPECIAL;

// < <=
s = transducer_new_state();
transducer_transitions[0][(int)'<'].next_state = s;

```

```

transducer_transitions[0][(int)'<'].type = TOKEN_TYPE_INCOMPLETE;

transducer_transitions[s][(int)'='].next_state = 0;
transducer_transitions[s][(int)'='].type = TOKEN_TYPE_SPECIAL;

transducer_transitions[s][EMPTY_STRING].next_state = 0;
transducer_transitions[s][EMPTY_STRING].type = TOKEN_TYPE_SPECIAL;

}

transducer_state transducer_new_state() {
    transducer_created_states += 1;
    return transducer_created_states - 1;
}

type_of_token transducer_consume_input(char current, char lookahead) {

    if (lookahead == EOF) {
        lookahead = ' ';
    }

    transducer_state next_state = transducer_transitions[transducer_current_state][(int)
current].next_state;
    type_of_token type = transducer_transitions[transducer_current_state][(int) current].type;

    if (next_state == INVALID_STATE) {
        type = TOKEN_TYPE_INVALID;
        next_state = 0;
    } else if (type == TOKEN_TYPE_INCOMPLETE && transducer_transitions[next_state]
[(int) lookahead].next_state == INVALID_STATE) {
        type = transducer_transitions[next_state][EMPTY_STRING].type;
        next_state = 0;
    }

    transducer_current_state = next_state;

    return type;
}

```

Apêndice V - analyser.h

```
#ifndef ANALYSER_H
#define ANALYSER_H

/*
 * analyser.h
 * compiler
 *
 * Created by Filipe Morgado Simões de Campos e Rafael Barbolo Lopes on 24/09/10.
 *
 */

/*****
name: get_next_token
purpose: get the next token from the source code.
args:
returns:
*****/

void get_next_token();

#endif
```


Apêndice VI - analyser.c

```
/*
 * analyser.c
 * compiler
 *
 * Created by Filipe Morgado Simões de Campos e Rafael Barbolo Lopes on 24/09/10.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include "token.h"
#include "reader.h"
#include "analyser.h"
#include "tables.h"

void get_next_token() {

    char temp_string[500];
    char stra[3];
    int counter;

    token.type = TOKEN_TYPE_IGNORED;
    token.value = NULL;

    if (reading_head.current == EOF) {
        /* return a END_OF_FILE token if the file was completely read */

        token.type = TOKEN_TYPE_END_OF_FILE;

    } else {
        /* extract a token from file */

        while (reading_head.current != EOF && token.type == TOKEN_TYPE_IGNORED)
        {
            /* chars being discarded */

            /* the transducer automata will execute a transition to consume the
current char */
            token.type = transducer_consume_input(reading_head.current,
reading_head.next);

            read_next_char();

        }
    }
}
```

```

/*
 * the previous char will not be dicarted because it created a transition to a
 * valid state with transducer automata
 */
temp_string[0] = reading_head.previous;
counter = 1;

token.line = reading_head.line;
token.column = reading_head.column-1;

while (reading_head.current != EOF && token.type ==
TOKEN_TYPE_INCOMPLETE) {
    /* building token */

    /* the transducer automata will execute a transition to consume the
current char */
    token.type = transducer_consume_input(reading_head.current,
reading_head.next);

    temp_string[counter] = reading_head.current;
    counter += 1;

    read_next_char();
}

/* allocate memory to token value */
token.value = (char*) malloc(counter*sizeof(char));
while (counter > 0) {
    /* fill each char of the token value */
    token.value[counter-1] = temp_string[counter-1];
    counter -= 1;
}

if (token.type == TOKEN_TYPE_IGNORED) {
    free(token.value);
    token.value = NULL;
} else if (token.type == TOKEN_TYPE_INCOMPLETE) {
    token.type = TOKEN_TYPE_INVALID;
} else if (token.type == TOKEN_TYPE_IDENTIFIER &&
(find_by_key(&table_reserved_words, token.value) >= 0)) {
    token.type = TOKEN_TYPE_RESERVED_WORD;
}

```

```
token.index = update_semantic_tables();
```

```
}
```

```
}
```

Apêndice VII - tables.h

```
#ifndef TABLES_H
#define TABLES_H
/*
 * tables.h
 * compiler
 *
 * Created by Filipe Morgado Simões de Campos e Rafael Barbolo Lopes on 24/09/10.
 */

#include "hashtable.h"

hash_table table_symbols;
hash_table table_numbers;
hash_table table_reserved_words;
hash_table table_specials;

/*****
name: initialize_semantic_tables
purpose: initialize the tables user by the semantic analyzer
args:
returns:
*****/

void initialize_semantic_tables();

/*****
name: update_semantic_tables
purpose: insert/update token into a table related to its type.
args:
returns: index in table.
*****/

int update_semantic_tables();

#endif
```

Apêndice VIII - tables.c

```
/*
 * tablesymbols.c
 * compiler
 *
 * Created by Filipe Morgado Simões de Campos e Rafael Barbolo Lopes on 24/09/10.
 *
 */

#include "tables.h"
#include "token.h"

void initialize_semantic_tables() {
    table_symbols = init_table(table_symbols);
    table_numbers = init_table(table_numbers);
    table_reserved_words = init_table(table_reserved_words);
    table_specials = init_table(table_specials);

    /* add reserved words */
    add(&table_reserved_words, "if");
    add(&table_reserved_words, "for");
    add(&table_reserved_words, "else");
}

int update_semantic_tables() {

    int index = -1;

    if (token.type == TOKEN_TYPE_INT_NUMBER)
        index = add(&table_numbers, token.value);
    else if (token.type == TOKEN_TYPE_RESERVED_WORD)
        index = add(&table_reserved_words, token.value);
    else if (token.type == TOKEN_TYPE_IDENTIFIER)
        index = add(&table_symbols, token.value);
    else if (token.type == TOKEN_TYPE_SPECIAL)
        index = add(&table_specials, token.value);

    return index;
}
```

Apêndice IX - hashtable.h

```
#ifndef HASHTABLE_H
#define HASHTABLE_H
/*
 * hashtable.h
 * compiler
 *
 * Created by Filipe Morgado Simões de Campos e Rafael Barbolo Lopes on 24/09/10.
 */

#include <stdio.h>

typedef struct _No {

    char * key;

    //Value from the last No plus 1.
    int value;

    struct _No * next;

} hash_table;

/*****
name:
purpose: .
args: .
returns: .
*****/
hash_table init_table(hash_table new_hash_table);

/*****
name: add
purpose: Add values to the table.
args: The key and the table.
```

```

returns: The hash_table value added.
*****/

int add(hash_table * table, char * key);

/*****
name:
purpose: .
args: .
returns: .
*****/

hash_table * find_last_cell(hash_table * table);

/*****
name:
purpose: .
args: .
returns: .
*****/

int find_by_key(hash_table * table, char * key);

#endif

```

Apêndice X - hashtable.c

```
/*
 * hashtable.c
 * compiler
 *
 * Created by Filipe Morgado Simões de Campos e Rafael Barbolo Lopes on 24/09/10
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include "strings.h"
#include "hashtable.h"

hash_table init_table(hash_table new_hash_table){

    new_hash_table.key = "no_key";
    new_hash_table.value = -1;
    new_hash_table.next = NULL;

    return new_hash_table;
}

int add(hash_table * table, char * key){

    int value;

    //Check if there is an entry with the key given.
    value = find_by_key(table, key);

    if ( value != -1 ){
        //The key already exists.
        return value;
    }
    else {
        //Add a new cell

        //If the first cell has its key == no_key. So, to add a cell we only overwrite its values.
        if (table->key == "no_key") {
```



```

        table->key = key;
        table->value = 0;
        table->next = NULL;
    }
    else {

        //Allocate memory to the new cell
        hash_table * new_hash_cell = (hash_table *) malloc(sizeof(hash_table));

        //Find the last cell and store a pointer to it.
        hash_table * last_cell_pointer = find_last_cell(table);

        //
        new_hash_cell->key = key;
        new_hash_cell->value = last_cell_pointer->value + 1;
        new_hash_cell->next = NULL;

        last_cell_pointer->next = new_hash_cell;

        return new_hash_cell->value;
    }
}

return table->value;
}

hash_table * find_last_cell(hash_table * table){

    hash_table * current_table_cell;

    current_table_cell = table;

    while (1) {

        if (current_table_cell->next == NULL) {
            return current_table_cell;
        }

        current_table_cell = current_table_cell->next;
    }

}

int find_by_key(hash_table * table, char * key){

```

```
hash_table * current_table_cell;

current_table_cell = table;

while (1) {

    if (strcmp(current_table_cell->key, key) == 0) {
        return current_table_cell->value;
    }

    if (current_table_cell->next != NULL) {
        current_table_cell = current_table_cell->next;
    }
    else {
        return -1;
    }
}

return -1;
}
```

Apêndice XI - strings.h

```
#ifndef STRING_H
#define STRING_H

/*
 * strings.h
 * compiler
 *
 * Created by Filipe Morgado Simões de Campos e Rafael Barbolo Lopes on 27/09/10
 *
 */

/*****
name: strcmp
purpose: compares two string and return 1 if they have the same value and 0 if not.
args: string s1 and string s2
returns: true or false (1 or 0)
*****/

int strcmp(char *s1, char *s2);

#endif
```

Apêndice XII - strings.c

```
/*
 * strings.c
 * compiler
 *
 * Created by Filipe Morgado Simões de Campos e Rafael Barbolo Lopes on 27/09/10
 *
 */

#include "strings.h"

int strcmp(char *s1, char *s2) {

    while (*s1 == *s2) {

        if (*s1 == 0) {
            /* finish the comparison and return false */
            return 0;
        }

        s1++;
        s2++;
    }

    /* return true */
    return 1;
}
```

Apêndice XIII - reader.h

```
#ifndef READER_H
#define READER_H
/*
 * reader.h
 * compilador
 *
 * Created by Filipe Morgado Simões de Campos e Rafael Barbolo Lopes on 24/09/10.
 *
 *
 * This file is responsible for reading the characters from the source code and returning it
 * to the compiler. It is specially used by the lexical analysis step.
 *
 *
 */

/* struct used to read a file */
typedef struct reader_type {
    FILE * file_pointer; // pointer to the file being read

    /* last, current and next chars */
    char previous; // last char
    char current;
    char next;

    /* column and line of the cursor in the file */
    int line;
    int column;
} Reader;

/* global variable: reading head */
Reader reading_head;

/*****
name: read_next_char
purpose: read the next char from the file being scanned.
args:
returns:
*****/
void read_next_char();
```

```
/******  
name: init_reader  
purpose: initialize the file reader.  
args: filename  
returns:  
*****/
```

```
void init_reader(char *filename);
```

```
#endif
```

Apêndice XIV - reader.c

```
/*
 * reader.c
 * compilador
 *
 * Created by Filipe Morgado Simões de Campos e Rafael Barbolo Lopes on 24/09/10.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include "reader.h"

void init_reader(char *filename) {

    /* open file */
    reading_head.file_pointer = fopen(filename, "r");

    /* raise an error and terminate execution if the file could not be opened */
    if( reading_head.file_pointer == NULL ) {
        printf("ERROR: \n\tFile \"%s\" could not be opened. \n\tCheck its path, name and\n\textension.\n", filename);
        exit(1);
    }

    /* update the reading head */
    reading_head.previous = 0;
    reading_head.current = fgetc(reading_head.file_pointer);
    reading_head.next = fgetc(reading_head.file_pointer);
    reading_head.line = 1;
    reading_head.column = 1;

}

void read_next_char() {
    /* update the heading read */
    reading_head.previous = reading_head.current;
    reading_head.current = reading_head.next;
    reading_head.next = fgetc(reading_head.file_pointer);

    if (reading_head.previous == '\n') {
        reading_head.line += 1;
        reading_head.column = 0;
    }
}
```

```
    reading_head.column += 1;  
}
```


Apêndice XV - main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "reader.h"
#include "token.h"
#include "analyser.h"
#include "tables.h"

int main (int argc, const char * argv[]) {

    /* initialize the file reader */
    init_reader("../resources/test_lex.poli");

    /* initialize transducer automata transitions */
    initialize_transitions();

    /* initialize semantic tables */
    initialize_semantic_tables();

    /* loop that reads all tokens from source code */
    while (token.type != TOKEN_TYPE_END_OF_FILE) {
        get_next_token();

        if (token.type != TOKEN_TYPE_END_OF_FILE) {
            printf("Token: %s    (%d)  ::    %s    (%d)  ::    line    %i,
column%i\n",
                                token.value, token.index, token_type_name(), token.type,
token.line, token.column);
        }
    }

    return 0;
}
```