# Projeto de Compilador
# E1 de **Análise Léxica**

Prof. Lucas Mello Schnorr
schnorr@inf.ufrgs.br

## 1 Introdução

A primeira etapa consiste em fazer um analisador léxico utilizando a ferramenta de geração de reconhecedores `flex`. Tu deves manter o arquivo `tokens.h` (fornecido) sem modificações. A função principal deve estar em um arquivo `main.c` separado do arquivo `scanner.l` para facilitar testes automáticos que utilizam uma função principal parecida com a fornecida em anexo.

## 2 Funcionalidades Necessárias

### 2.1 Definir expressões regulares

Reconhecimento dos lexemas correspondentes aos tokens descritos na seção **Descrição dos Tokens** abaixo, unicamente através da definição de expressões regulares no arquivo da ferramenta `flex`. Cada expressão regular deve estar associada a pelo menos um tipo de token. Classificar os lexemas reconhecidos em tokens retornando as constantes definidas no arquivo `tokens.h` fornecido ou códigos ASCII para caracteres simples.

### 2.2 Contagem de linhas

Controlar o número de linha do arquivo de entrada. Uma função cujo protótipo é `int get_line_number(void)` deve ser implementada e deve retornar o número da linha atual no processo de reconhecimento de `tokens`. Ela é utilizada nos testes automáticos. Lembre-se que a primeira linha de qualquer arquivo dado como é entrada é a linha número um.

### 2.3 Ignorar comentários

Ignorar comentários no formato C99: tudo o que segue a partir de `//` e tudo que está compreendido entre `/*` e `*/`. As linhas devem ser contabilizadas mesmo dentro de comentários do segundo tipo. Espaços devem ser igualmente ignorados.

### 2.4 Lançar erros léxicos

Lançar erros léxicos ao encontrar caracteres inválidos na entrada, retornando o token de erro.

## 3 Descrição dos Tokens

Existem tokens que correspondem a caracteres particulares, como vírgula, ponto-e-vírgula, parênteses, para os quais é mais conveniente usar seu próprio código `ASCII`, convertido para inteiro, como valor de retorno que os identifica. Para os tokens compostos, como palavras reservadas e identificadores, utiliza-se uma constante, conforme o arquivo `tokens.h` fornecido (mais tarde utilizaremos recursos do `bison`) com um código maior do que 255 para representá-los. Os tokens se enquadram em diferentes categorias: (1) palavras reservadas da linguagem; (2) caracteres especiais; (3) operadores compostos; (4) identificadores; e (5) literais.

### 3.1 Palavras Reservadas da Linguagem

As palavras reservadas da linguagem são:

```
int float bool char string if then else
while do input output return const static
foreach for switch case break continue class
private public protected end default
```

### 3.2 Caracteres Especiais

Os caracteres simples especiais empregados pela linguagem são listados abaixo separados apenas por espaços, e devem ser retornados com o próprio código `ASCII` convertido para inteiro. São eles:

```
, ; : ( ) [ ] { } + - |
* / < > = ! & % # ^ . $
```

### 3.3 Operadores Compostos

A linguagem possui operadores compostos, além dos operadores representados por alguns dos caracteres da seção anterior. Os operadores compostos são:

```
<=  >=  == != && || >> <<
```

Onde o `>>` é o shift para a esquerda (`TOK_OC_SL`).

### 3.4 Identificadores

Os identificadores da linguagem são formados por um caractere alfabético seguido de zero ou mais caracteres alfanuméricos, onde considera-se caractere alfabético como letras maiúsculas ou minúsculas ou o caractere sublinhado e onde dígitos são `0`, `1`, `2`, ..., `9`.

### 3.5 Literais

Literais são formas de descrever constantes no código fonte. Literais do tipo `int` são representados como repetições de um ou mais dígitos precedidos opcionalmente pelo sinal de negativo ou positivo. Literais em `float` são formados como um inteiro seguido de ponto decimal e uma sequência de dígitos. A notação científica é possível para números ponto flutuantes utilizando um `E` ou `e` seguindo de um número positivo ou negativo inteiro. Literais do tipo `bool` podem ser `false` ou `true`. Literais do tipo `char` são representados por um único caractere entre entre aspas simples como por exemplo:

```
'a'
' '
'+'

"meu nome"
"x = 3;"
```

# A Arquivo `tokens.h`

```
/*
Lista dos tokens, com valores constantes associados.  Este arquivo
será posterioremente substituído, não acrescente nada.  Os valores das
constantes sao arbitrários, mas não podem ser alterados.  Cada valor
deve ser distinto e fora da escala ASCII.  Assim, não conflitam entre
si e com os tokens representados pelo próprio valor ASCII de
caracteres isolados.
*/

#define TK_PR_INT          256
#define TK_PR_FLOAT        257
#define TK_PR_BOOL         258
#define TK_PR_CHAR         259
#define TK_PR_STRING       260
#define TK_PR_IF           261
#define TK_PR_THEN         262
#define TK_PR_ELSE         263
#define TK_PR_WHILE        264
#define TK_PR_DO           265
#define TK_PR_INPUT        266
#define TK_PR_OUTPUT       267
#define TK_PR_RETURN       268
#define TK_PR_CONST        269
#define TK_PR_STATIC       270
#define TK_PR_FOREACH      271
#define TK_PR_FOR          272
#define TK_PR_SWITCH       273
#define TK_PR_CASE         274
#define TK_PR_BREAK        275
#define TK_PR_CONTINUE     276
#define TK_PR_CLASS        277
#define TK_PR_PRIVATE      278
#define TK_PR_PUBLIC       279
#define TK_PR_PROTECTED    280
#define TK_OC_LE           281
#define TK_OC_GE           282
#define TK_OC_EQ           283
#define TK_OC_NE           284
#define TK_OC_AND          285
#define TK_OC_OR           286
#define TK_OC_SL           287
#define TK_OC_SR           288
#define TK_LIT_INT         291
#define TK_LIT_FLOAT       292
#define TK_LIT_FALSE       293
#define TK_LIT_TRUE        294
#define TK_LIT_CHAR        295
#define TK_LIT_STRING      296
#define TK_IDENTIFICADOR   297
#define TOKEN_ERRO         298
#define TK_PR_END          299
#define TK_PR_DEFAULT      300
```

# B Arquivo `main.c`

```
/*
Função principal para impressão de tokens.

Este arquivo será posterioremente substituído, não acrescente nada.
*/
```

```
#include <stdio.h>
#include "tokens.h"
extern int yylex(void);
extern int yylex_destroy(void);

extern FILE *yyin;
extern char *yytext;
extern int get_line_number (void);
#define print_nome(TOKEN) \
    printf("%d " #TOKEN " [%s]\n", get_line_number(), yytext);
#define print_nome2(TOKEN) \
    printf("%d TK_ESPECIAL [%c]\n", get_line_number(), TOKEN);

int main (int argc, char **argv)
{
  int token = 0;
  while (token = yylex()) {
    switch (token){
    case ',':
    case ';':
    case ':':
    case '(':
    case ')':
    case '[':
    case ']':
    case '{':
    case '}':
    case '+':
    case '-':
    case '|':
    case '*':
    case '/':
    case '<':
    case '>':
    case '=':
    case '!':
    case '&':
    case '%':
    case '#':
    case '^':
    case '.':
    case '$': print_nome2 (token); break;
    case TK_PR_INT: print_nome(TK_PR_INT); break;
    case TK_PR_FLOAT: print_nome(TK_PR_FLOAT); break;
    case TK_PR_BOOL: print_nome (TK_PR_BOOL); break;
    case TK_PR_CHAR: print_nome (TK_PR_CHAR); break;
    case TK_PR_STRING: print_nome (TK_PR_STRING); break;
    case TK_PR_IF: print_nome (TK_PR_IF); break;
    case TK_PR_THEN: print_nome (TK_PR_THEN); break;
    case TK_PR_ELSE: print_nome (TK_PR_ELSE); break;
    case TK_PR_WHILE: print_nome (TK_PR_WHILE); break;
    case TK_PR_DO: print_nome (TK_PR_DO); break;
    case TK_PR_INPUT: print_nome (TK_PR_INPUT); break;
    case TK_PR_OUTPUT: print_nome (TK_PR_OUTPUT); break;
    case TK_PR_RETURN: print_nome (TK_PR_RETURN); break;
    case TK_PR_CONST: print_nome (TK_PR_CONST); break;
    case TK_PR_STATIC: print_nome (TK_PR_STATIC); break;
    case TK_PR_FOREACH: print_nome (TK_PR_FOREACH); break;
    case TK_PR_FOR: print_nome (TK_PR_FOR); break;
    case TK_PR_SWITCH: print_nome (TK_PR_SWITCH); break;
    case TK_PR_CASE: print_nome (TK_PR_CASE); break;
    case TK_PR_BREAK: print_nome (TK_PR_BREAK); break;
    case TK_PR_CONTINUE: print_nome (TK_PR_CONTINUE); break;
```

```
    case TK_PR_CLASS: print_nome (TK_PR_CLASS); break;
    case TK_PR_PRIVATE: print_nome (TK_PR_PRIVATE); break;
    case TK_PR_PUBLIC: print_nome (TK_PR_PUBLIC); break;
    case TK_PR_PROTECTED: print_nome (TK_PR_PROTECTED); break;
    case TK_PR_END: print_nome (TK_PR_END); break;
    case TK_PR_DEFAULT: print_nome (TK_PR_DEFAULT); break;
    case TK_OC_LE: print_nome (TK_OC_LE); break;
    case TK_OC_GE: print_nome (TK_OC_GE); break;
    case TK_OC_EQ: print_nome (TK_OC_EQ); break;
    case TK_OC_NE: print_nome (TK_OC_NE); break;
    case TK_OC_AND: print_nome (TK_OC_AND); break;
    case TK_OC_OR: print_nome (TK_OC_OR); break;
    case TK_OC_SL: print_nome (TK_OC_SL); break;
    case TK_OC_SR: print_nome (TK_OC_SR); break;
    case TK_LIT_INT: print_nome (TK_LIT_INT); break;
    case TK_LIT_FLOAT: print_nome (TK_LIT_FLOAT); break;
    case TK_LIT_FALSE: print_nome (TK_LIT_FALSE); break;
    case TK_LIT_TRUE: print_nome (TK_LIT_TRUE); break;
    case TK_LIT_CHAR: print_nome (TK_LIT_CHAR); break;
    case TK_LIT_STRING: print_nome (TK_LIT_STRING); break;
    case TK_IDENTIFICADOR: print_nome (TK_IDENTIFICADOR); break;
    case TOKEN_ERRO:  print_nome (TOKEN_ERRO); break;
    default: printf ("<Invalid Token with code %d>\n", token); return 1; break;
    }
  }
  yylex_destroy();
  return 0;
}
```