Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Disciplina: Teoria dos Compiladores
Professor: Marcelo Bernardes Vieira

# PROJETO E IMPLEMENTAÇÃO DE UM COMPILADOR – 1ª PARTE

**Projete e implemente um analisador léxico para a linguagem C--, definida pela gramática livre de contexto listada no fim deste documento:**

1) Liste o conjunto de *tokens* da linguagem. Eles serão reconhecidos e retornados pelo seu analisador léxico;
2) Defina expressões regulares para este conjunto de *tokens*;
3) Derive **um único AFD** (autômato finito determinista) que reconheça todas as expressões regulares;
4) Implemente o AFD em **linguagem C**. O código deve ser claro e eficiente no reconhecimento dos *tokens*;
5) Implemente uma **classe de tabela de símbolos** em **C++** para gerenciar palavras reservadas, identificadores e literais encontrados em um programa;
6) **Atenda a todas as especificações mínimas a seguir**. Este trabalho faz parte de um projeto muito maior.

**Documentação mínima:**

A documentação, a ser entregue **em papel**, deve ser objetivo, impessoal e completo, explicando os detalhes do quê o programa faz, como faz, e ainda apresenta conclusões obtidas pelo trabalho. É um documento a parte que deve conter pelo menos as seguintes estruturas (seções):

1) Título:
   - Apresente o título do trabalho e/ou nome do compilador;
2) Visão geral:
   - Dê uma visão geral do funcionamento do programa;
   - Descrição sucinta sobre o desenvolvimento do trabalho. Explique as decisões de implementação tomadas.
   - Descrição dos módulos e suas interdependências. Apresente uma breve descrição de cada módulo bem como um diagrama, por exemplo, mostrando suas inter-relações;
3) *Tokens* e expressões regulares:
   - Apresente uma tabela de *tokens* com seus respectivos lexemas, expressões regulares e atributos. Mostre as palavras reservadas da linguagem;
   - *Tokens* são representados como valores inteiros internamente ao compilador. Liste os *tokens* com os valores que você escolheu;
   - Projete um autômato finito determinista que reconheça as expressões regulares definida;
4) Analisador léxico:
   - Descreva e justifique a forma com a qual vocês implementaram o analisador léxico;
   - Apresente as estruturas de dados, os métodos e os algoritmos utilizados;
5) Tabela de símbolos:
   - Descreva e justifique a forma com a qual vocês implementaram a classe de tabela de símbolos;
   - Apresente as estruturas de dados, os métodos e os algoritmos utilizados;
6) Gerenciador de erros léxicos:
   - Apresente claramente o método de gerenciamento de erros implementado;
7) Resultados experimentais
   - Listagem dos testes executados: deve conter os arquivos de entrada compilados pelo programa (arquivos texto **.cmm** com código C--) com os **respectivos resultados de saída**.
   - **COMENTE** os testes executados;
   - **ATENÇÃO**: procure fazer testes relevantes levando em conta as diferentes construções da linguagem a ser compilada. Prove que seu compilador é capaz de realizar as tarefas especificadas abaixo.
8) Conclusão
   - Esse é o lugar em que vocês podem me comunicar tudo o que vocês acham que é importante de ser dito;
   - Descreva todas as características especiais do seu compilador nessa seção.
   - Indique extensões e funcionalidades extras.
9) Referências bibliográficas
   - Mostre quais as fontes que vocês consultaram para realizar **cada etapa do trabalho**. Inclua quaisquer artigos, livros, citações, etc., consultados. No caso dos livros do curso, indique quais as seções lidas/usadas para cada tarefa.
10) Listagem do programa fonte
   - O código deve ser claro, comentado, indentado e **sem linhas quebradas**. Use uma fonte isométrica (todos os caracteres têm a mesma largura e altura). Ex. *Courier*, *Courier New*.
   - Forneçam uma listagem organizada e compacta. No Word, **use fonte com o menor tamanho possível**, em duas colunas e com orientação do tipo paisagem. Evite longos trechos de código dentro de um mesmo bloco.
   - Os programas fonte devem ser entregues prontos para compilar, sem erros nem avisos, com *makefile*, projeto CodeBlocks, projeto CodeLite, e descompactados. Na apresentação, leve um backup em pendrive. Não conte com o uso da rede. **ATENÇÃO**: o código deve rodar em ambiente Linux e Windows (MinGW).

**Especificações mínimas para o analisador léxico:**

- Este trabalho é realizável somente através da leitura dos livros. Não reinvente a roda. Seja objetivo e desenvolva como está na literatura;
- Abuse ostensivamente de comentários esclarecedores sobre e em **todo** o código do compilador;
- Os programas fonte da linguagem possuem extensão **.cmm**;
- O programa principal deve ser do tipo "linha de comando" cujo primeiro parâmetro é **opcional** e define o arquivo texto fonte. Se o arquivo não for especificado, a entrada padrão **`stdin`** deve ser usada. Ex.: para compilar um arquivo "teste.cmm", pode-se executar: `C:> compila teste.cmm` ou `C:> compila teste`. Note que a extensão é adicionada se não for informada. A lista de parâmetros será ampliada nas próximas fases do projeto;
- A saída do programa **deve** ser enviada **exclusivamente para `stdout`**. Não crie nenhum arquivo de saída;
- As mensagens de erro **devem** ser enviadas **exclusivamente para `stderr`**. Não crie nenhum arquivo de mensagens de erro.
- As **strings numéricas** e **alfanuméricas** não podem ter restrição de comprimento. Assim, vocês devem gerenciar um arranjo dinâmico de caracteres (com tamanho monótono) dentro do analisador léxico para agregar cada caractere lido do buffer de entrada;
- O analisador léxico deve desconsiderar comentários (não geram *tokens*), que têm a seguinte forma em C--:

```
/* este comentário pode
   se estender por várias linhas */
```

- O gerenciador de erros deve relatar a linha na qual ocorreu o erro, seguida de uma mensagem com significado. Os erros deverão ser obrigatoriamente enviados para o fluxo padrão de erros **`stderr`**. Quando ocorrer um erro **não pare** a compilação. Ao contrário, **tente encontrar o próximo *token* válido**;
- Ao encontrar o fim do arquivo texto de entrada, o analisador léxico deverá retornar o *token* especial **EOF**.
- A saída do compilador deve ser a seqüência de *tokens* encontrada pela análise do arquivo texto de entrada. Para facilitar a correção, não imprima a representação numérica dos *tokens*. Ao invés disso, mostre seus nomes. Por exemplo, se o *token* 56 indica a palavra reservada **`if`**, então imprima `IF` ao invés de 56 quando ocorrer uma string '`if`' no texto fonte. *Tokens* com atributo devem ser seguidos de um ponto '.' e o valor do atributo. Dado o seguinte texto fonte, por exemplo:

```
if(x1 <= 32) b = 10;
```

a saída do compilador deve ser parecida com

```
IF
LPARENT
ID.x1
RELATIONALLT
NUMINT.32
RPARENT
ID.b
ASSIGN
NUMINT.10
SEMICOLON
EOF
```

- **Não imprima nada dentro do módulo do analisador léxico**: ele deve ser mantido o mais limpo possível. Toda a saída de texto para **`stdout`** deve estar no programa principal.
- A **classe genérica para tabela de símbolos** deve ser eficiente na busca e inserção de *strings*. Ela deve ser composta por **registros** cuja chave de busca e inserção são *strings*. **IMPORTANTE**: a política de acesso deve ser, no mínimo, do tipo *hash*.
  - Não é permitida nenhuma restrição ao tamanho de quaisquer *strings*. Crie um arranjo dinâmico (atributo privado da classe) para guardar **todas** as *strings* referenciadas pela instância de tabela.
  - A tabela de símbolos simplifica a identificação de palavras reservadas, a manipulação de informações sobre identificadores e o armazenamento de literais. Portanto, utilizando a classe desenvolvida, **instancie três tabelas nessa fase do compilador:** palavras reservadas, identificadores, literais.
    - Todas as **strings** (alfanuméricas ou literais) reconhecidas pelo autômato devem ser pesquisadas nas tabelas de símbolos para se descobrir seu *token*.
  - Na tabela de palavras reservadas, insira um campo do **tipo inteiro** para indicar o *token* da *string*.
- Assim que o analisador léxico retornar EOF, a função `main` do seu compilador deve imprimir todas as entradas de cada uma das três tabelas de símbolos. Exemplo:

```
TABELA DE SÍMBOLOS: PALAVRAS RESERVADAS
---------------------------------------
LEXEMA                  Token numérico
---------------------------------------
IF                        56
WHILE                     27

TABELA DE SÍMBOLOS: IDENTIFICADORES
-----------------------------------
x1
b

TABELA DE SÍMBOLOS: LITERAIS
----------------------------
```

```
"Digite seu nome: "
```

**Desenvolvendo o trabalho:**

- Não escreva nenhum código antes de obter um autômato finito determinista **definitivo**.
  - Primeiramente, encontre os *tokens* da linguagem a partir da GLC C-- no final desta especificação.
  - Espaços (branco, tabulação, fim de linha, etc.) e comentários não geram *tokens*. Eles devem ser simplesmente descartados;
  - Procure determinar *tokens* individuais para cada símbolo de pontuação e para operadores (COMMA para ‘, ‘, SEMICOLON para ‘; ‘ e RELATIONALGT para “>=”, por exemplo). A outra opção, **não recomendada**, seria definir apenas um *token* para todos os símbolos e diferenciá-los através de um atributo.
  - Uma vez com a lista de *tokens*, crie expressões regulares para as diversas classes. As palavras reservadas e os identificadores são inseridos e diferenciados na tabela de símbolos. Ela tem a função de simplificar o autômato. **Atenção: não use a tabela de símbolos para gerenciar operadores e pontuação**. Isso torna o analisador léxico muito menos eficiente. Eles devem ser reconhecidos pelo autômato.
  - Defina o autômato finito determinista que reconheça todas as expressões regulares;
  - Não deixe de consultar o livro a respeito de como implementar a tabela de símbolos.
- Mais uma vez, consulte os livros do curso para desenvolver cada etapa. Nele vocês encontrarão vários atalhos e exemplos.

**Prazo e avaliação:**

Lembre-se que este projeto é o principal elemento de avaliação do curso. Uma documentação de alta qualidade técnica, formal e de escrita é importante. O código fonte deve estar correto, muito bem documentado (mas muito mesmo) e deve ser eficiente.

**Todos os grupos não têm permissão de possuir ou manter consigo o código ou os resultados de outro grupo, incluindo trabalhos antigos**. É aconselhável que os grupos mantenham sigilo sobre suas soluções. Qualquer indício de cópia será considerado fato grave contra todos os grupos envolvidos, e a pena será a anulação de TODAS AS NOTAS DE TRABALHO.

A **avaliação do grupo** depende de:

- Atendimento ao que foi solicitado;
- Resultados práticos;
- Qualidade da documentação;
- Qualidade do código;
- Pontualidade na entrega;
- **Apresentação de no mínimo 30 minutos** demonstrando as funcionalidades do compilador;
- Criatividade na resolução dos problemas;
- Criatividade em ir além do mínimo (**mas somente após alcançar o mínimo**).

A **avaliação individual** depende de:

- **Presença e participação em todas as aulas e durante toda a aula**;
- Destaque no grupo. Ter participação maior do que os outros participantes (só se não forçar por interesse);
- Destaque na apresentação do compilador.

A **avaliação da classe** depende de:

- Diferença entre a maior e a menor nota dos grupos;
- Diferença entre a maior e a menor nota individual;
- Apresentações dos grupos.

A apresentação deverá ser feita no momento da entrega da documentação.

**PRAZOS:**

**Este trabalho tem a carga esperada de 60 Homem Hora: em um grupo de três pessoas, cada um individualmente deve, em média, se dedicar 5 horas semanais ao trabalho.**

O prazo máximo para a entrega e a apresentação é dia **18 de outubro de 2022** fora do horário da aula, se houver.
**Penalidade por dia de atraso (corridos)**: 3 pontos.

**Funcionalidades opcionais:**

**Não adicione funcionalidades opcionais antes que a solução básica esteja completa e correta**. Uma implementação incompleta das funcionalidades obrigatórias mais adicionais resultará em uma nota **menor** do que uma implementação completa sem nenhuma característica extra.

**Gramática C--:**

```
Program      -->   FunctionDecl Program      |
                   TypeDecl Program          |
                   VarDecl Program           |
                   FunctionDecl

TypeDecl     -->   typedef struct { Type IdList ; VarDecl } ID ; TypeDecl | epsilon

VarDecl      -->   Type IdList ; VarDecl     |     epsilon

IdList       -->   Pointer ID Array          |
                   IdList , Pointer ID Array |

Pointer      -->   *      |     epsilon

Array        -->   [  NUM  ] Array   |    epsilon

FunctionDecl -->   Type Pointer ID ( FormalList ) { VarDecl StmtList }

FormalList   -->   Type Pointer ID Array FormalRest | epsilon

FormalRest   -->   , Type Pointer ID Array FormalRest | epsilon

Type         -->   long  |  int  |  float |  bool  |  ID  |  char  |  double

StmtList     -->   Stmt          |
                   Stmt StmtList

Stmt         -->   if ( Expr ) Stmt else Stmt       |
                   while ( Expr ) Stmt              |
                   switch( Expr ) { CaseBlock }     |
                   break ;                          |
                   print ( ExprList ) ;             |
                   readln ( Expr ) ;                |
                   return Expr ;                    |
                   throw ;                          |
                   { StmtList }                     |
                   ID ( ExprList ) ;                |
                   try Stmt catch ( "…" ) Stmt      |
                   Expr  ;                          |

CaseBlock    -->   case  NUM ":" StmtList CaseBlock|
                   case  NUM ":" CaseBlock

ExprList     -->   epsilon      |
                   ExprListTail

ExprListTail -->   Expr         |
                   Expr , ExprListTail

Expr         -->   Primary            |
                   UnaryOp Expr       |
                   Expr BinOp Expr    |
                   Expr = Expr


Primary      -->   ID | NUM | LITERAL  |
                   ´ ASCII ´          |
                   ( Expr )           |
                   Expr "."   ID      |
                   Expr "->"  ID      |
                   ID ( ExprList )    |
                   Expr [ Expr ]      |
                   "&"  Expr          |
                   "*"  Expr          |
                   true | false

UnaryOp      -->   "-"  |  "!" | "+"

BinOp        -->   "==" | "<" | "<=" | ">=" | ">" | "!=" | "+"  |
                   "-"  | "|" | "*"  | "/"  | "%" | "&"  | "&&" | "||"
```