Universidade Federal de Mato Grosso do Sul
Faculdade de Computação
Linguagens Formais e Autômatos - T1 - 2022/1
Profa. Edna A. Hoshino

# Trabalho Prático - autômatos finitos × analisadores léxicos + autômato de pilha × analisadores sintáticos

## 1 Analisador léxico e sintático para a linguagem C-

Um compilador é um programa que traduz um código de uma linguagem-fonte para um código em uma linguagem-alvo. Para isso, os compiladores basicamente realizam duas etapas: análise e síntese. Na fase de análise, o código na linguagem-fonte é analisada do ponto de vista léxico, sintático e semântico, enquanto na síntese, um código equivalente é gerado na linguagem-alvo.

O trabalho prático refere-se ao uso de autômatos finitos determinísticos e autômatos com pilha para realizar, respectivamente, a análise léxica e sintática de programas escritos em um subconjunto da linguagem C, o qual denominaremos linguagem C-.

A linguagem C- é aquela gerada pela seguinte gramática $G = (V, T, P, S)$ em que $P$ é constituída pelas seguintes produções:

```
S         -> Function S
           | Function
Function  -> Type main() { B }
           | Type id() { B }
Type      -> void
           | int
           | float
B         -> C B
           | epsilon
C         -> id = E ;
           | while (E) C
           | { B }
E         -> E + E
           | E * E
           | ( E )
           | id
           | num
```

Para que essa gramática seja usada para implementar um tipo específico de analisador sintático, chamado **Analisador Descendente Recursivo**, algumas alterações nas suas produções são necessárias (explicadas na Seção 1.2). Portanto, considere as seguintes produções em $P$ ao invés das elencadas anteriormente:

```
S         -> Function S_
S_        -> Function S_
           | epsilon
Function  -> Type Function_
Type      -> void
           | int
           | float
Function_ -> main() { B }
           | id() { B }
B         -> C B
```

```
            | epsilon
 C          -> id = E ;
            | while (E) C
            | { B }
 E          -> T E_
 E_         -> + T E_
            | epsilon
 T          -> F T_
 T_         -> * F T_
            | epsilon
 F          -> ( E )
            | id
            | num
```

Os conjuntos de variáveis $V$ e de terminais $T$ são dados abaixo:

```
V = {S, S_, Function, Function_, Type, C, B, E, T, E_, T_, F}

T= {ELSE, FLOAT, FOR, IF, INT, MAIN, VOID, WHILE, ID, NUM, OP_ATRIB, OP_ADIT, OP_MULT,
    OP_REL, ABRE_PARENT, FECHA_PARENT,PONTO_VIRG, VIRG, ABRE_CHAVES, FECHA_CHAVES, FIM,
    INVALIDO}
```

**Observação:** Note que o código do analisador léxico disponibilizado pelo professor, na verdade, aceita outros terminais além dos descritos no conjunto $T$. Cada terminal descrito no conjunto $T$ é uma constante inteira que representa uma palavra sobre os símbolos do alfabeto $\Sigma = \{=, +, *, >, <, >=, <=, ==, (, ), ;, , , ,, [a..z], [0..9], -, /, “, \}$, por exemplo, o terminal `OP_REL` representa “<”, “>”, “<=”, “>=”, “! =” e “==”. Para saber quais outros terminais estão implementados no código, basta usar o programa `avalia-lex`. Veja mais à frente as instruções sobre como compilar e usar os programas.

### 1.1 Análise Léxica

Um analisador léxico é o componente de um compilador responsável por reconhecer os terminais da gramática da linguagem-fonte.

O autômato finito determinístico $M$ é aquele que aceita o conjunto das palavras associadas aos terminais $T$. O autômato $M$ processa os caracteres de um arquivo de entrada, reconhecendo os terminais em $T$ ou identificando um terminal INVALIDO. Os espaços em branco, tabulações e fim de linha são ignorados por $M$.

Considere o seguinte exemplo de programa escrito em C-:

```
void main()
{
   a 10 =;
   while (a!=0)
   {
     a = @a+1;
}
```

O processamento do autômato $M$ no exemplo acima resultará no reconhecimento da seguinte sequencia de terminais:

```
VOID
MAIN
ABRE_PARENT
FECHA_PARENT
ABRE_CHAVES
```

```
ID
NUM
OP_ATRIB
PONTO_VIRG
WHILE
ABRE_PARENT
ID
OP_REL
NUM
FECHA_PARENT
ABRE_CHAVES
ID
OP_ATRIB
INVALIDO
ID
OP_ADD
NUM
PONTO_VIRG
FECHA_CHAVES
```

Note que o analisador léxico não se preocupa com a ordem em que os terminais aparecem no programa nem na quantidade delas (o analisador léxico não verificou que o comando de atribuição está incorreto nem que falta '}'). O analisador léxico verifica somente a ordem em que os caracteres aparecem no arquivo de entrada, ou seja, no programa, para reconhecer um terminal.

O autômato $M$ pode ser implementado de duas formas diferentes.

Na primeira, as transições do autômato $M$ são implementadas através de uma tabela de transições armazenada em uma matriz de dimensão $|Q| \times |\Sigma|$, onde $Q$ é o conjunto de estados de $M$ e $\Sigma$ o alfabeto dos símbolos de entrada. O alfabeto é armazenado em um vetor de caracteres. No entanto, três símbolos adicionais também são considerados: LETRA, DIGITO e INVALIDO. O primeiro representa qualquer letra, o segundo um dígito qualquer e o último um símbolo não pertencente ao alfabeto. Este último é incluído para permitir ao autômato tratar os casos em que símbolos desconhecidos aparecem na entrada.

Na segunda forma, as transições do autômato $M$ são implementadas diretamente no código, usando comandos condicionais ou comandos `switch-case` aninhados, uma para cada estado do autômato e a outra interna para símbolos do alfabeto. A vantagem deste método é a rapidez em processar uma palavra da linguagem aceita pelo autômato, mas tem a desvantagem de exigir que o código seja alterado cada vez que a linguagem aceita pelo autômato muda.

No código disponibilizado pelo professor da disciplina, ambas as implementações estão disponíveis. A primeira no arquivo `lex.c` e a segunda no arquivo `lex2.c`. Instruções sobre como compilar o nosso "compilador de C-" usando um desses analisadores estão disponíveis em um arquivo chamado `README.txt`.

**Compilando e usando o avaliador do léxico:** Para compilar o avaliador do léxico (avaliaLex.c) usando o `lex.c` faça:

```
make -f makefile-lex
```

Para executar o avaliador, faça:

```
./avalia-lex exemplo1.c
```

aqui, `exemplo1.c` é um programa exemplo disponibilizado junto com o código. Você pode usar qualquer programa escrito na linguagem C-.

Caso prefira usar o `lex2.c` no lugar do `lex.c`, compile o avaliador da seguinte forma:

```
make -f makefile-lex2
```

Para executar o avaliador do `lex2.c`, faça:

```
./avalia-lex2 exemplo1.c
```

A saída gerada pelo programa será algo como:

```
Na linha   1 encontrado token:             int  (codigo 10):  [int]
Na linha   1 encontrado token:              id  (codigo 17):  [f]
Na linha   1 encontrado token:     ABRE_PARENT  (codigo 23):  [(]
Na linha   1 encontrado token:    FECHA_PARENT  (codigo 24):  [)]
Na linha   2 encontrado token:     ABRE_CHAVES  (codigo 27):  [{]
Na linha   3 encontrado token:              id  (codigo 17):  [a]
Na linha   3 encontrado token:        OP_ATRIB  (codigo 19):  [=]
Na linha   3 encontrado token:             num  (codigo 18):  [10]
Na linha   3 encontrado token:      PONTO_VIRG  (codigo 25):  [;]
...
```

significando que o léxico encontrou na linha 1 as seguintes palavras "int" (cujo terminal é `INT`), depois encontrou "f" (cujo terminal é `ID`), depois "(", terminal `ABRE_PARENT`, e assim sucessivamente.

## 1.2 Análise Sintática

Um analisador sintático é o responsável por reconhecer as palavras geradas pela gramática da linguagem-fonte. Note que as palavras geradas por $G$ equivalem aos programas que respeitam as regras (ou produções) da gramática C-.

Para explicar o funcionamento de um analisador sintático, considere o seguinte exemplo de programa:

```
void main()
{
   a = 10;
   b = 3;
   while (a){
     c = (a+b*2;
     a = a + 1;
     b = b*b;
   }
}
```

Para o exemplo acima, um analisador sintático reportaria uma mensagem de erro como apresentado abaixo para indicar que um ")" era esperado e não foi encontrado:

```
Total de linhas processadas: 6
Resultado: Esperado token 15 (FECHA_PARENT)
```

Nesse trabalho, é fornecido o código de um analisador sintático (chamado **analisador descendente recursivo**) que simula a execução de um autômato com pilha determinístico $M'$ e aceita a linguagem gerada por uma gramática $G$.

A construção do analisador sintático descendente recursivo exige que a gramática $G$ seja do tipo LL(1). Gramáticas do tipo **LL(1)** são aquelas que podem gerar todas as palavras de uma linguagem usando apenas 1 símbolo por vez da palavra, começando pelo símbolo mais à esquerda (**L**eft) e fazendo derivações mais à esquerda (**L**eft) de forma determinística.

Formalmente, uma gramática é LL(1) se e somente se, sempre que $A \rightarrow \alpha|\beta$ forem duas produções distintas da gramática as seguintes condições devem ser satisfeitas:

- (i) $\alpha$ e $\beta$ não derivam em palavras começando pelo mesmo terminal;

- (ii) ambos não podem derivar $\varepsilon$;
- (iii) Se $\beta$ derivar em $\varepsilon$ então $\alpha$ não pode derivar palavras que começam com um terminal que pertença ao conjunto $\{x \in T : S \Rightarrow^* \gamma A x \eta$, com $\gamma$, $\eta$ em $(T \cup V)^*\}$, ou seja, o terminal não pode aparecer imediatamente à frente de $A$ em alguma forma sentencial derivada a partir de $S$.

Observe que gramáticas ambíguas e gramáticas com recursão mais à esquerda não são LL(1).

Quando a gramática $G$ é LL(1), o analisador sintático descendente recursivo pode ser construído da forma descrita na próxima seção.

## 2 Implementando um analisador sintático descendente recursivo

Considere que a gramática $G$ é LL(1). Observe que a gramática descrita na Seção 1, que é a gramática da linguagem C-, é LL(1).

A ideia é criar uma função para cada variável da gramática. Essa função simula as derivações com respeito às produções da variável. Uma vez que a condição (i) a (iii) são respeitadas pela gramática, é possível decidir qual produção usar a cada passo de derivação olhando-se apenas o próximo símbolo da palavra (denominado `lookahead`).

Seja $A$ uma variável e sejam as seguintes produções de $A$:

$$A \to aA \quad | \quad bbB \quad | \quad c$$

A função associada à variável $A$ é

```
void A()
{
  if (lookahead==a){ /* usa a producao A-> aA */
      match(a);
      A();
  }
  else if (lookahead==b){   /* usa a producao A-> bbB */
      match(b);
      match(b);
      B();
  }
  else{  /* usa a producao A-> c */
      match(c);
  }
}
```

Note que a decisão por qual produção derivar é tomada com base no `lookahead`. A função `match(t)` verifica se o próximo símbolo é igual ao símbolo esperado `t` e avança o ponteiro da fita de entrada para atualizar o lookahead. Note que em caso de erro, a função mostra uma mensagem de erro e aborta o programa.

```
void match(int t)
{
  if (lookahead==t){
      lookahead = lex();
  }
  else{
      printf("\nErro: simbolo %d esperado.", t);
      exit(1);
  }
}
```

Apenas para complementar, suponha que a variável $B$ tenha uma produção vazia, por exemplo,

$$B \to bBb \quad | \quad \varepsilon.$$

Logo, a função associada a $B$, neste caso, seria:

```
void B()
{
  if (lookahead==b){ /* usa a producao B-> bBb */
      match(b);
      B();
      match(b);
  }
}
```

Note que se o lookahead for diferente de $b$, a função B não faz nada, o que equivale a realizar a derivação com a produção vazia $B \to \varepsilon$.

## 2.1 Objetivo

O objetivo deste trabalho é alterar o analisador sintático e/ou léxico para aumentar o conjunto de palavras geradas pela gramática da linguagem C-.

Cada grupo deve selecionar **apenas uma** das propriedades listadas abaixo e que ainda está inativa no compilador da linguagem C- e, então, deve alterar o programa `parser.c` e/ou o programa programa `lex.c` ou `lex2.c`[1] fornecido pelo professor para que o analisador sintático e léxico reconheça programas sintaticamente corretos satisfazendo a propriedade escolhida:

1. comando `for` + uso de constante caracter (como 'a');
2. declaração de variáveis globais de tipos básicos + uso de constante numérica real;
3. declaracao de variáveis locais de tipos básicos e do tipo ponteiro + uso de indireção com ponteiros (i.e., x = *a + 10);
4. declaração de parâmetros do tipo básico em função + chamada de função com passagem de argumentos por valor e por referência;
5. chamada de função com e sem argumentos + uso de constantes literais (strings como "teste");
6. comando `if` simples e composto + ignorar comentários de linha (esta alteração deve ser feita no analisador léxico e não no sintático);
7. declaração e uso de estruturas (por exemplo, `struct` x `int` a; `double` b);
8. comando `switch` - `case` + comando `break`;
9. uso de rótulos (`labels`) e de desvio incondicional (`goto`) + expressão com operador condicional ternário (por exemplo, $a > b?c : d$);
10. expressões aritméticas envolvendo operadores binários ($-$ e $/$) + operadores de incremento e decremento ($++$ e $--$) + operadores de atribuição ($+=$, $-=$, $*=$, $/=$);
11. uso de vetores com uma ou mais dimensões (i.é., indexação como V[i][j] = cont + A[i*10+1];) + ignorar comentários de bloco (esta alteração deve ser feita no analisador léxico e não no sintático);
12. expressões lógicas e relacionais.

[1] Eventualmente, também pode ser necessário alterar os arquivos de cabeçalhos lex.h e parser.h.

Para realizar estas atividades, cada grupo potencialmente apenas incluirá uma nova produção para uma variável já existente na gramática $G$ e alterará a função associada a esta variável (conforme explicado na Seção 2).

Note que o programa está dividido em 2 arquivos: `lex.c` e `parser.c`. As alterações que devem ser feitas por cada grupo envolvem o código do `parser.c` e potencialmente do `lex.c` (ou `lex2.c`), mas para ver o nome da constante associada a um terminal pode ser necessário olhar o código `lex.c` ou usar o avaliador do léxico, o qual exibe os nomes das constantes dos terminais.

### 2.2 Observações importantes

Para entregar o seu trabalho corretamente, observe os itens listados abaixo:

- o trabalho deve ser feito por grupos de **três a quatro integrantes** e o resultado do trabalho deve ser apresentado em sala de aula **com a participação de todos os integrantes do grupo**.
- qualquer forma de plágio não será aceita. Se identificado plágio, a nota de cada grupo envolvido será ZERO!
- a programação deve ser feita em linguagem `C` e compilável em uma instalação Linux padrão ANSI (com `gcc`.
- um relatório no formato `PDF` de **no máximo 1 página** (limite rígido) deve ser entregue junto com o código, para detalhar as alterações realizadas na gramática $G$. Descreva no relatório, as novas produções que foram adicionadas na gramática.
- o trabalho deve ser entregue em um arquivo formato `tgz` ou `zip` enviado via *EAD* até as 23:59 horas da data fixada para a entrega na página da disciplina. Deixe todo o código, ou seja os programas-fontes "lex.c" (ou o "lex2.c"), "lex.h", "parser.c", "makefile" (ou o "makefile2"), que foi alterado pelo grupo em uma pasta chamada `code` e o relatório em outra pasta chamada `texto`.