Análise léxica

Ivan Ricarte

2008

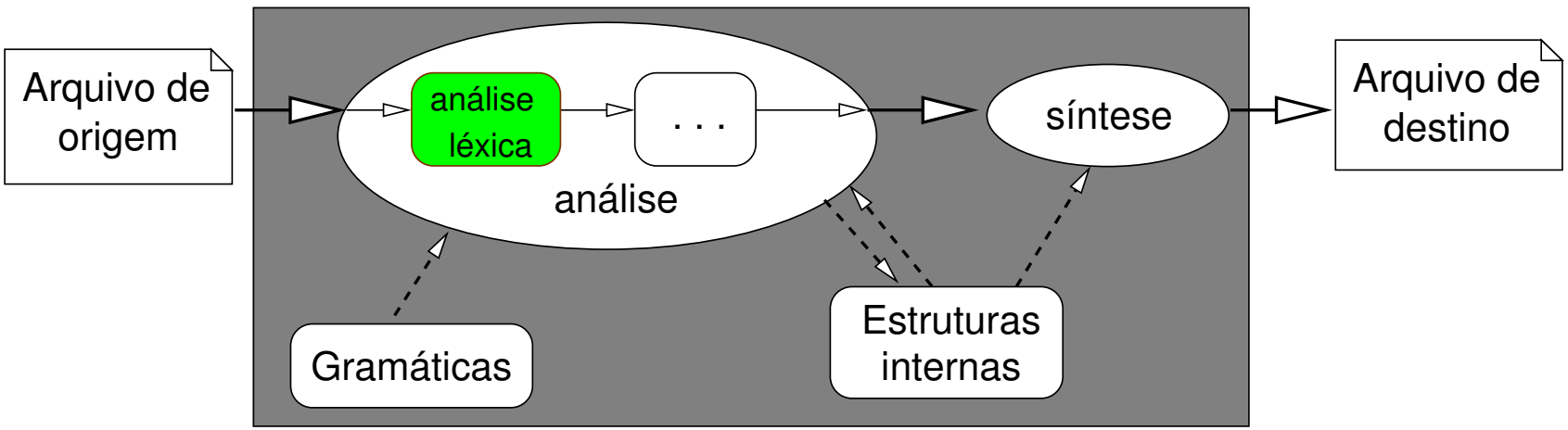
Sumário

Compiladores: análise léxica

Reconhecimento, a partir dos caracteres, das palavras básicas do “vocabulário” do programa

- palavras-chaves da linguagem, operadores, identificadores, constantes
- elementos que podem ser adequadamente descritos por meio de gramáticas regulares

Análise léxica



Gramática regular ou tipo 3

Revisão

- Apenas um símbolo não-terminal no lado esquerdo de cada produção
- Produções recursivas não apresentam auto-incorporação
  - Produção recursiva: mesmo símbolo aparece do lado esquerdo e do lado direito
  - Sem auto-incorporação: o símbolo do lado esquerdo aparece apenas no início ou no final da seqüência de símbolos do lado direito
- A linguagem regular descrita por uma gramática tipo 3 também pode ser descrita por meio de uma expressão regular

Analisadores léxicos

Programas que realizam a análise léxica

- Duas atividades básicas
  1. Obter grupos de caracteres, a partir do arquivo com conteúdo a ser analisado, que constituem strings válidas em uma dada linguagem
     - Cada grupo de caracteres é um token
  2. Identificar qual é o tipo de token

Varredura de tokens

- Leitura seqüencial dos caracteres da entrada
  - Agrupamento de caracteres reconhecido como um token da entrada
- Utiliza as facilidades oferecidas pelas linguagens de programação para leitura de arquivos

Varredura de tokens

Recursos da linguagem C++

- Definições no arquivo de cabeçalho `<fstream>`
- Leitura seqüencial de arquivos com a classe `ifstream`
  - Métodos `open`, `eof`
- Cada token é um objeto da classe `string` (arquivo de cabeçalho `<string>`)
- Leitura de tokens separados por brancos com o operador »

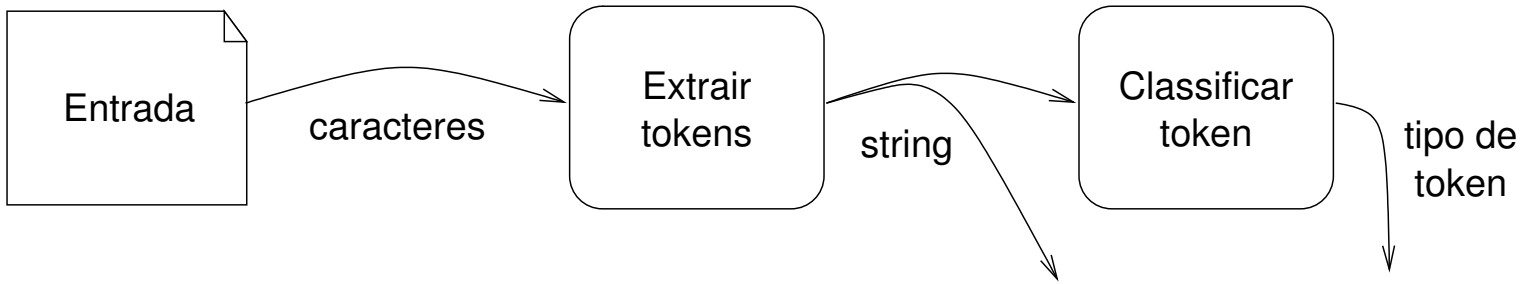
Varredura de tokens

Exemplo básico

```
#include <fstream>
#include <string>
...
ifstream arq;
string token;
...
arq.open("nome_arquivo");
...
while (! arq.eof() )
    arq >> token;
```

Classificação de tokens

- Token obtido precisa ser classificado
  - É um identificador, uma palavra-reservada, uma constante?



Autômatos finitos

- O núcleo do analisador léxico é uma implementação de um autômato finito
  - máquina de estados finitos que aceita símbolos de uma sentença
  - ao final da sentença, indica se ela é válida para a gramática ou não
  - autômato é definido para cada conjunto de símbolos que deve ser reconhecido
- Cada tipo de token é definido por uma lista ou por uma gramática regular

Autômato finito

Definição formal

Autômato é descrito por uma quíntupla

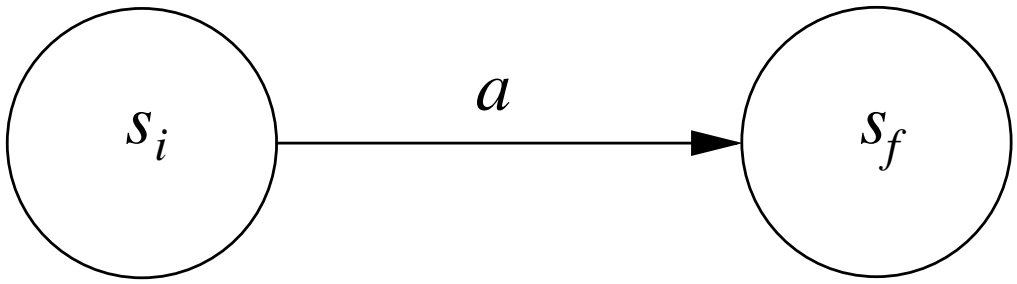
$$M = (K, \Sigma, \delta, s, F)$$

- K conjunto (finito) de estados
- Σ alfabeto (finito) de entrada
- δ conjunto de transições
- s estado inicial, $s \in K$
- F conjunto de estados finais, $F \subseteq K$

Autômato finito

Representação gráfica

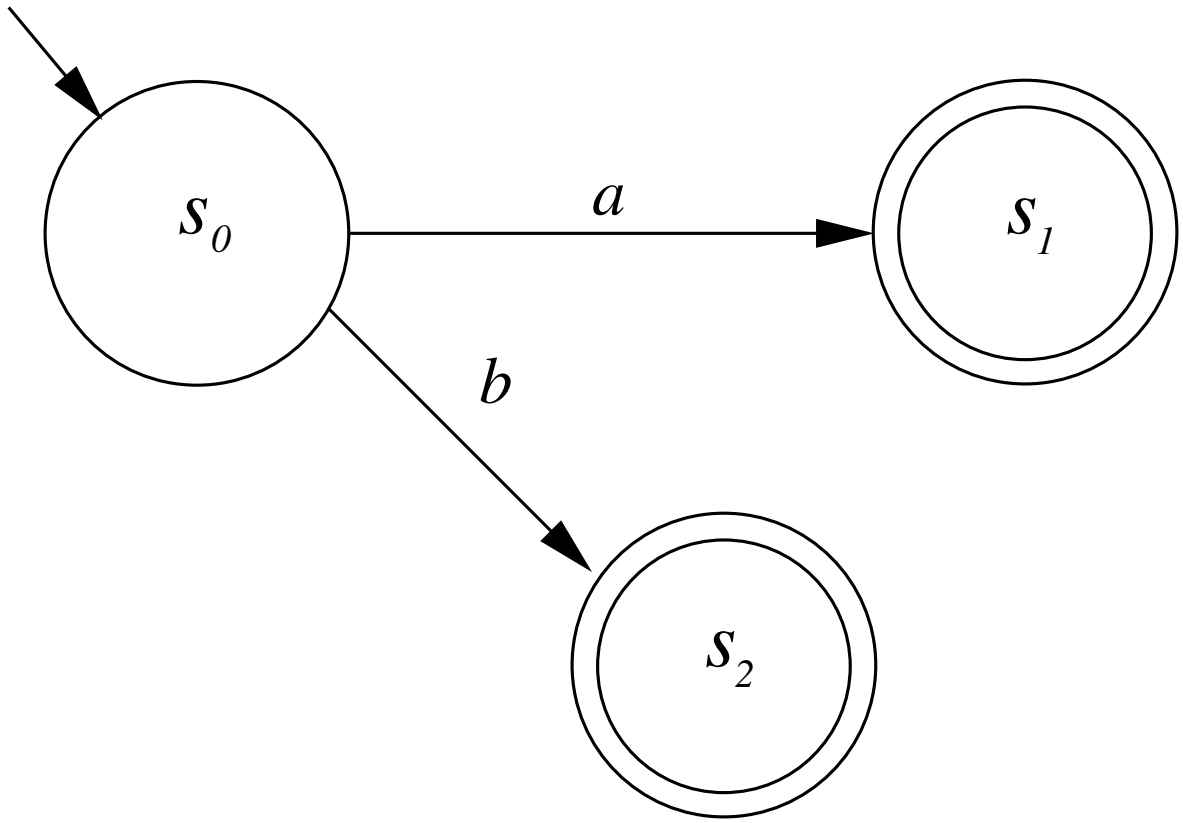
Estados e transição



Autômato finito

Representação gráfica

Estados inicial e finais



Autômato finito

Representação tabular

	s_0	s_1	s_2
a	s_1	—	—
b	s_2	—	—

Estado inicial: s_0

Estados finais: s_1 , s_2

Tipos de autômatos

Não determinísticos

A partir de um estado:

- pode haver transições para dois ou mais estados diferentes pela ocorrência de um mesmo símbolo na entrada
  - uma das transições é escolhida se ocorrer aquele símbolo
- pode haver transição para outro(s) estado(s) sem a ocorrência de nenhum símbolo
  - transição pela string vazia

Tipos de autômatos

Determinísticos

- Transição sempre ocorre pela ocorrência de um símbolo de entrada
  - Não há transições pela string vazia
- Não há alternativas distintas para a transição a partir de um dado estado e com um dado símbolo de entrada

Construção dos autômatos finitos

- Há um procedimento sistemático para construir um autômato que reconhece strings de uma linguagem regular:

  Algoritmo de Thompson construção de um autômato finito não determinístico a partir de uma expressão regular

  Método da construção de subconjuntos conversão do autômato finito não determinístico para um autômato finito determinístico equivalente

  Minimização de estados combinação de estados redundantes do autômato para construir o menor autômato finito determinístico que reconhece as strings da linguagem regular especificada

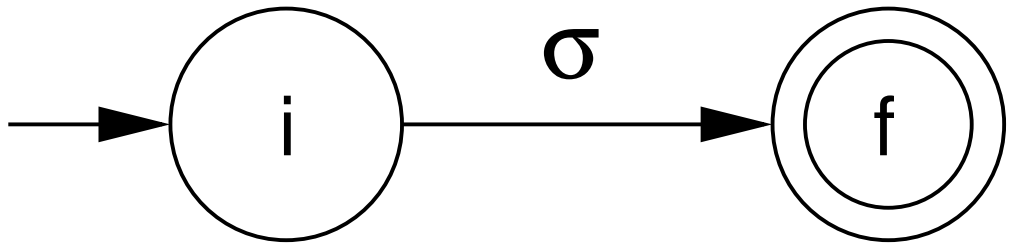
Algoritmo de Thompson

Compõe um autômato finito não determinístico pela combinação de pequenos autômatos que reconhecem os elementos primitivos de uma expressão regular:

- Um símbolo do alfabeto da linguagem
- Concatenação de duas expressões regulares
- Alternativa de duas expressões regulares
- Repetição (zero ou mais vezes) de uma expressão regular

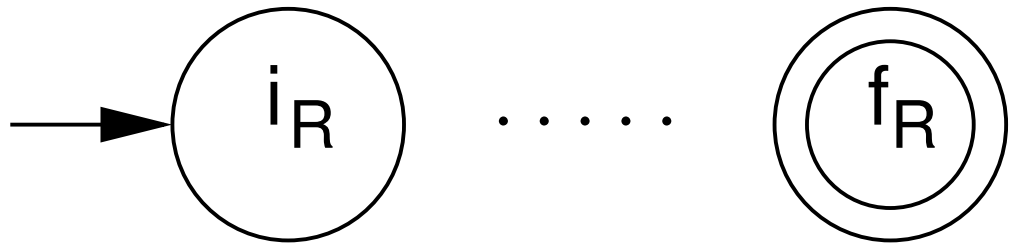
Algoritmo de Thompson

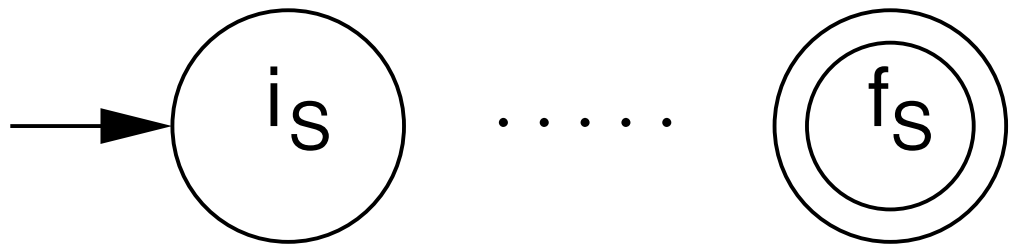
Autômato que reconhece um símbolo do alfabeto



Algoritmo de Thompson

- Para as demais construções, dois autômatos serão combinados, um para a expressão regular R e outro para a expressão regular S:

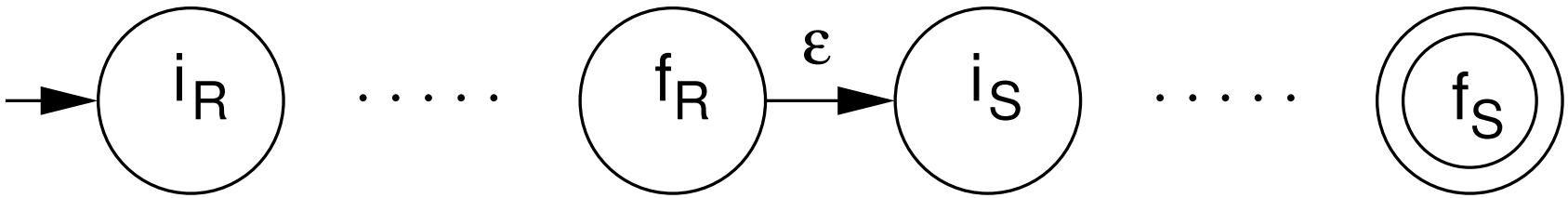




Algoritmo de Thompson

Concatenação de duas expressões regulares

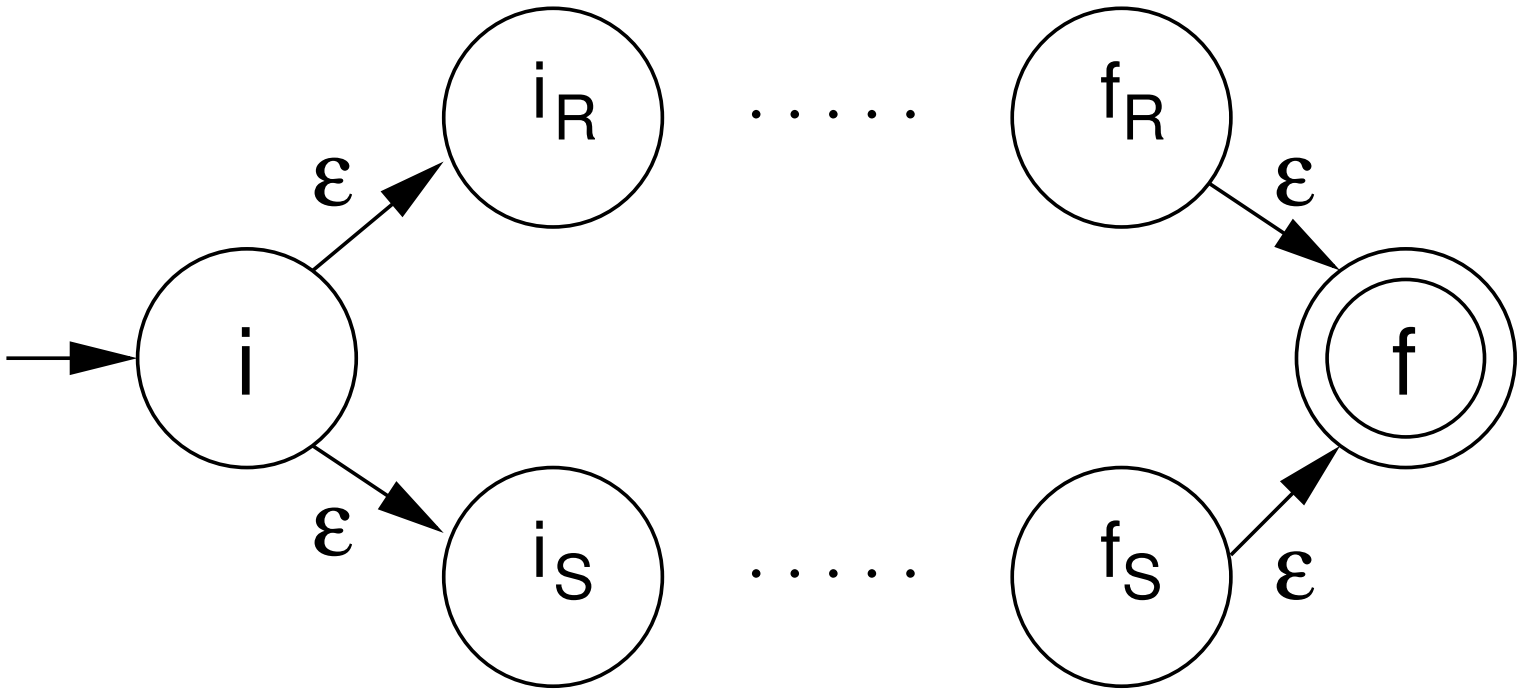
- Autômato que reconhece *RS*



Algoritmo de Thompson

Alternativa de duas expressões regulares

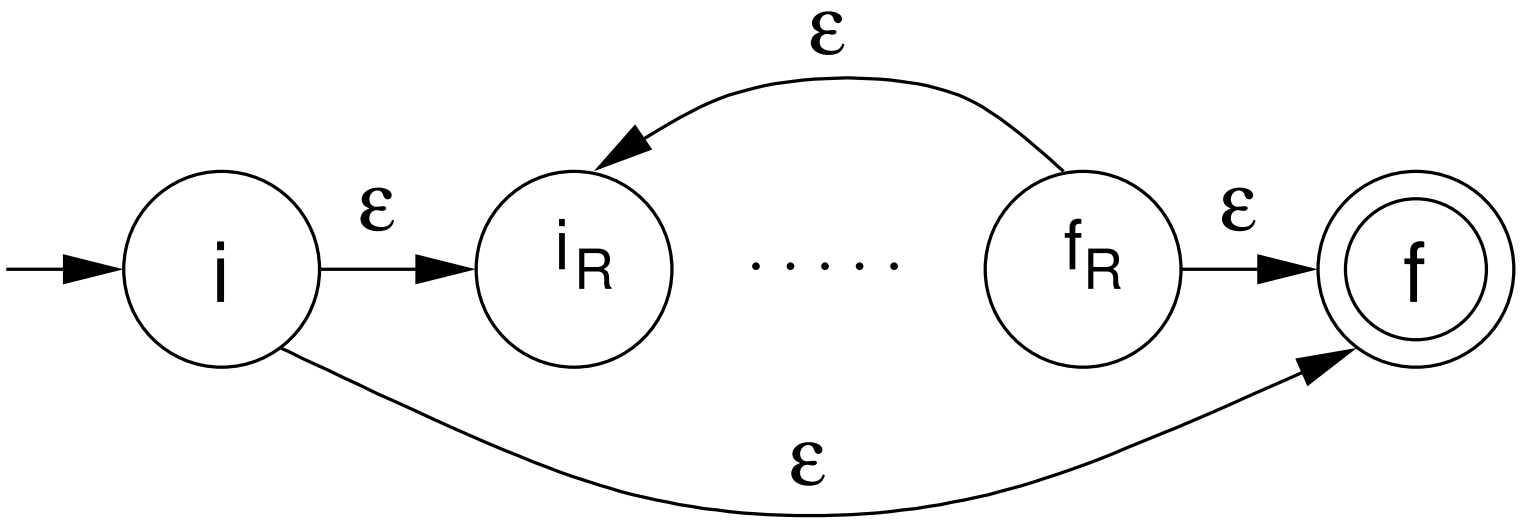
- Autômato que reconhece $R|S$



Algoritmo de Thompson

Repetição de uma expressão regular

- Autômato que reconhece $R*$



Algoritmo de Thompson

Exemplo

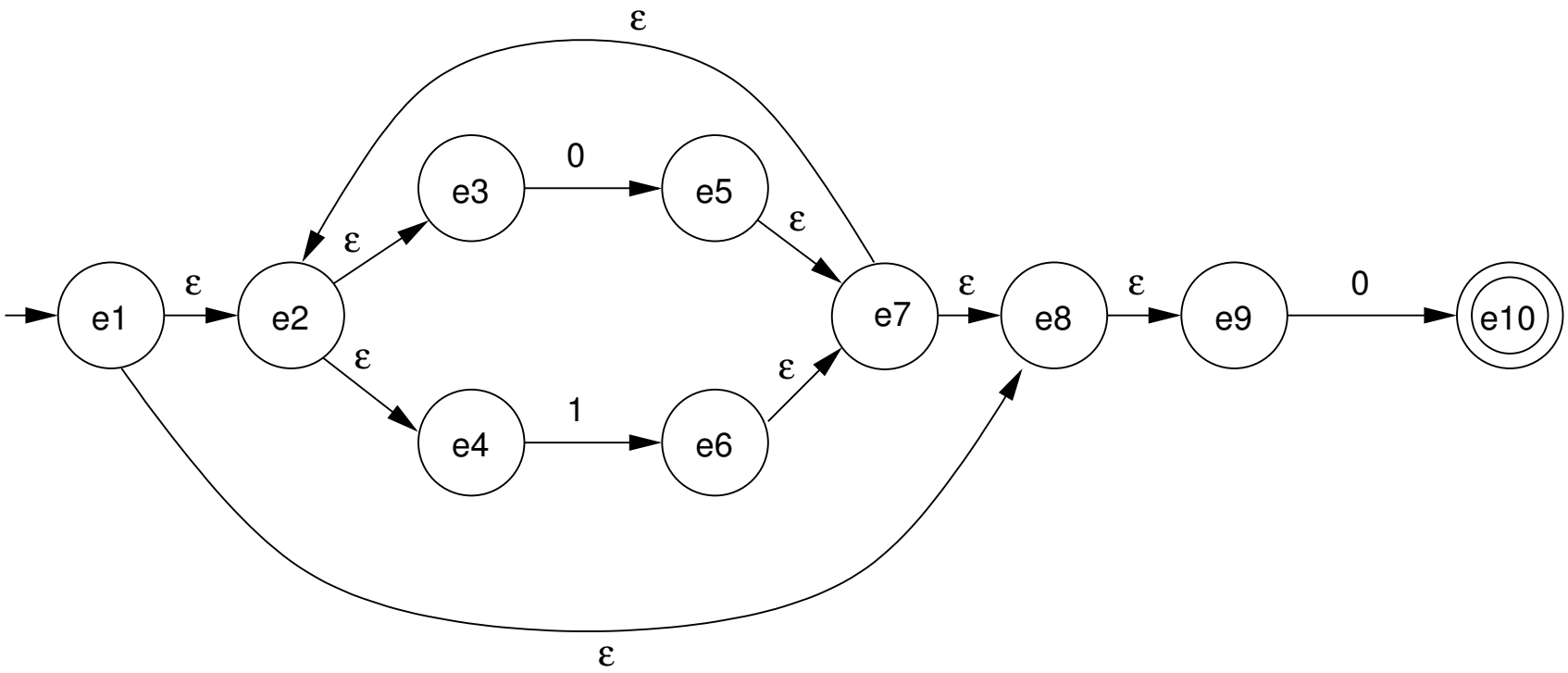
Autômato finito não-determinístico para reconhecer tokens da linguagem `(0|1)*0`:

1. Combinação de autômatos para reconhecer a concatenação de `(0|1)*` e `0`
2. Para reconhecer `(0|1)*`, um autômato para reconhecer a repetição de `0|1`
3. Para reconhecer `0|1`, um autômato para reconhecer a alternativa entre `0` e `1`

Algoritmo de Thompson

Exemplo

- Autômato para `(0|1)*0`



Conversão para autômato finito determinístico

O método da construção de subconjuntos

- Procedimento para conversão de autômato finito não-determinístico (AFND) para autômato finito determinístico (AFD) que reconhece strings da mesma expressão regular
- Estados que podem ser alcançados a partir de outro por meio de transições pela string vazia são combinados num único estado
  - Conceito de ε^* de um subconjunto de estados do AFND

O método da construção de subconjuntos

1. Obter estado inicial

- O estado inicial do AFD é definido como a ε^* do conjunto contendo apenas o estado inicial do AFND

2. Obter novos estados e transições

- Cada estado obtido para o AFD é analisado para descobrir, para cada símbolo do alfabeto, suas transições de saída e novos estados que são gerados

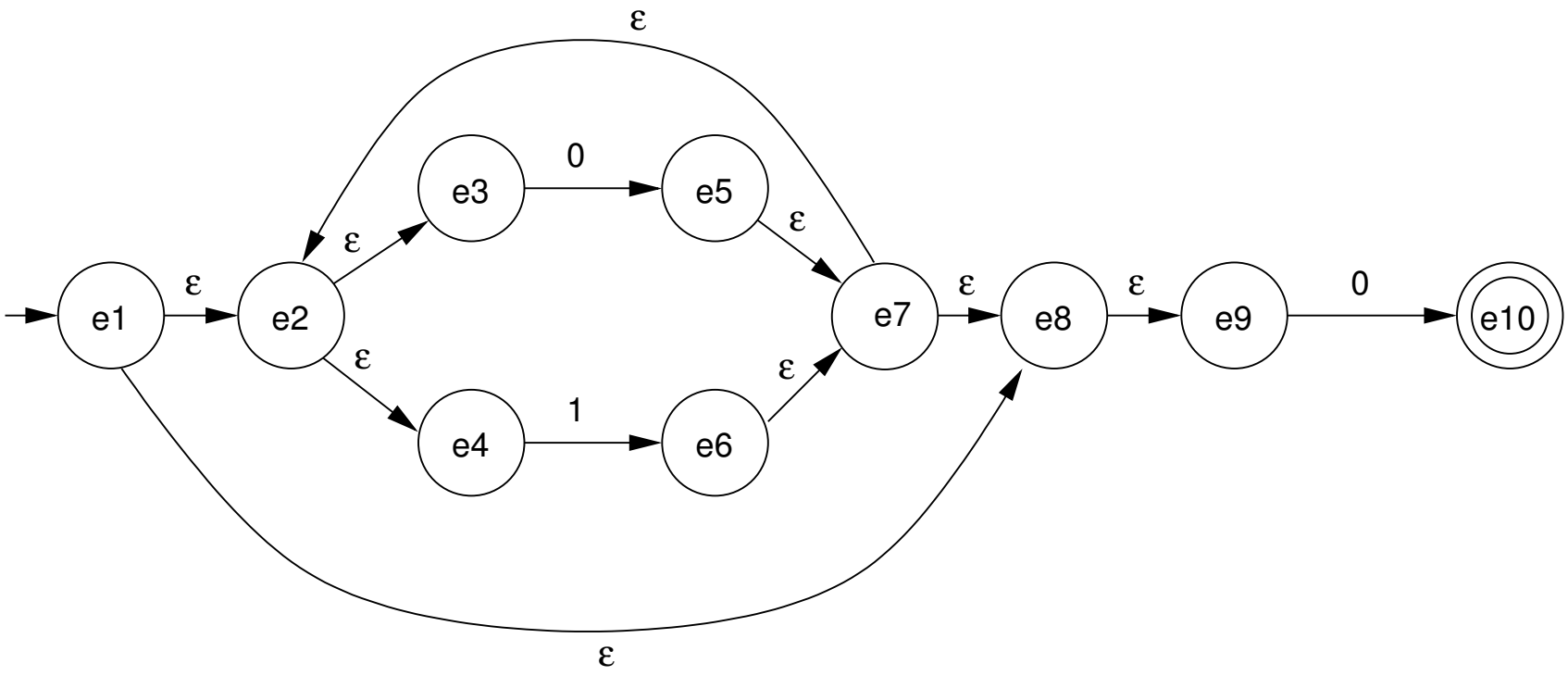
3. Marcar estados finais

- Cada estado do AFD que contenha em seu subconjunto um estado final do AFND será um estado final do AFD

Conversão para autômato finito determinístico

Exemplo

- Conversão do autômato para `(0|1)*0`



Conversão para autômato finito determinístico

Exemplo

- Estado inicial $\varepsilon^*\{e1\}$
  - $\{e1, e2, e3, e4, e8, e9\} : s0$
- Transições a partir do estado $s0$

$s0$	$e1$	$e2$	$e3$	$e4$	$e8$	$e9$
0	—	—	$e5$	—	—	$e10$
1	—	—	—	$e6$	—	—

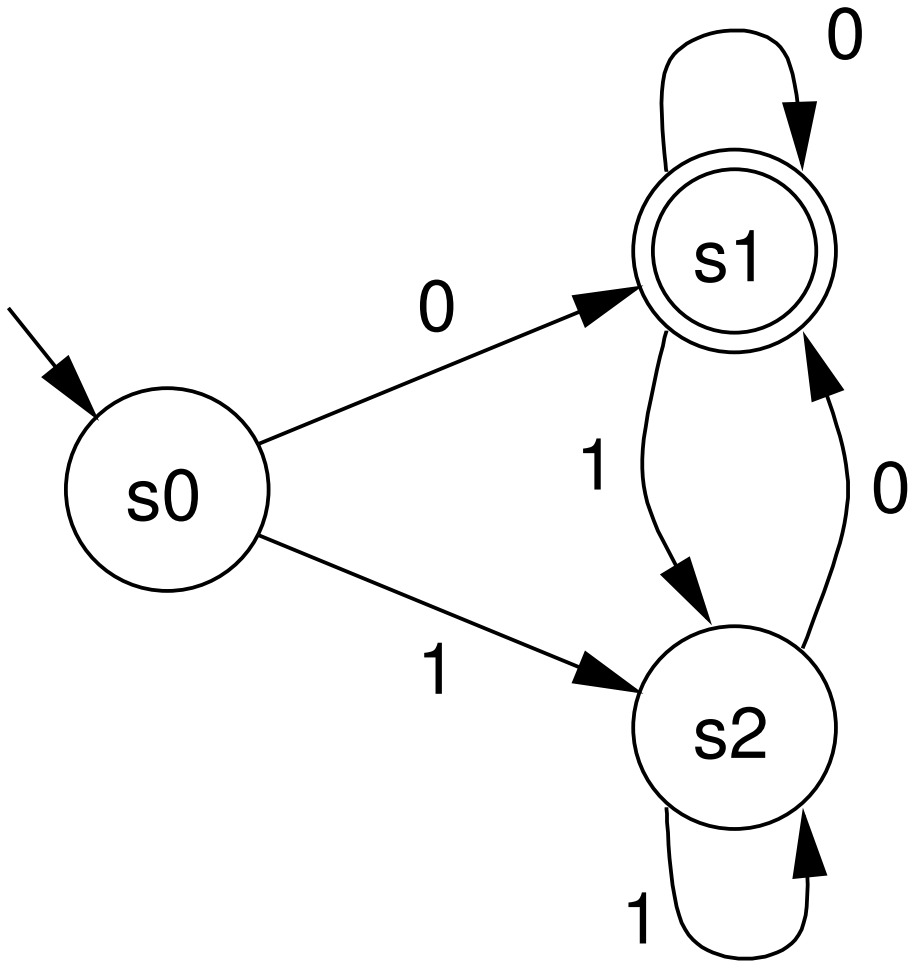
$s0/0$: $\varepsilon^*\{e5, e10\} =$
$\{e2, e3, e4, e5, e7, e8, e9, e10\}$ $(s1)$ (final)

$s0/1$: $\varepsilon^*\{e6\} =$
$\{e2, e3, e4, e6, e7, e8, e9\}$ $(s2)$

- Estados $s1, s2$ devem ser analisados da mesma forma, assim como novos estados que surjam desta análise

Conversão para autômato finito determinístico

Exemplo: resultado



Minimização de estados

- Método da construção de subconjuntos gera autômato finito determinístico
  - Possivelmente, com estados redundantes
- Procedimento de minimização permite obter autômato equivalente com menor número de estados
  - Baseado no particionamento sucessivo do conjunto de estados

Procedimento de minimização de estados

1. Particione o conjunto de estados do autômato

- Em um grupo, estados que são finais
- No outro grupo, todos os demais estados

2. Avalie transições para os estados de cada grupo

- Refinar particionamento: estados que têm transições para grupos diferentes têm que ser colocados em (novos) grupos diferentes

3. Combinar estados redundantes

- Quando todos os estados do grupo não mudam os grupos de destino quando transições são avaliadas, estados são redundantes — um único estado é suficiente para descrever esse comportamento

Minimização de estados

Exemplo

- Para o autômato obtido para a expressão `(0|1)*0`
  1. Partição inicial $P_1 = \{C_1, C_2\}$, com

$$C_1 = \{s1\} \qquad C_2 = \{s0, s2\}$$

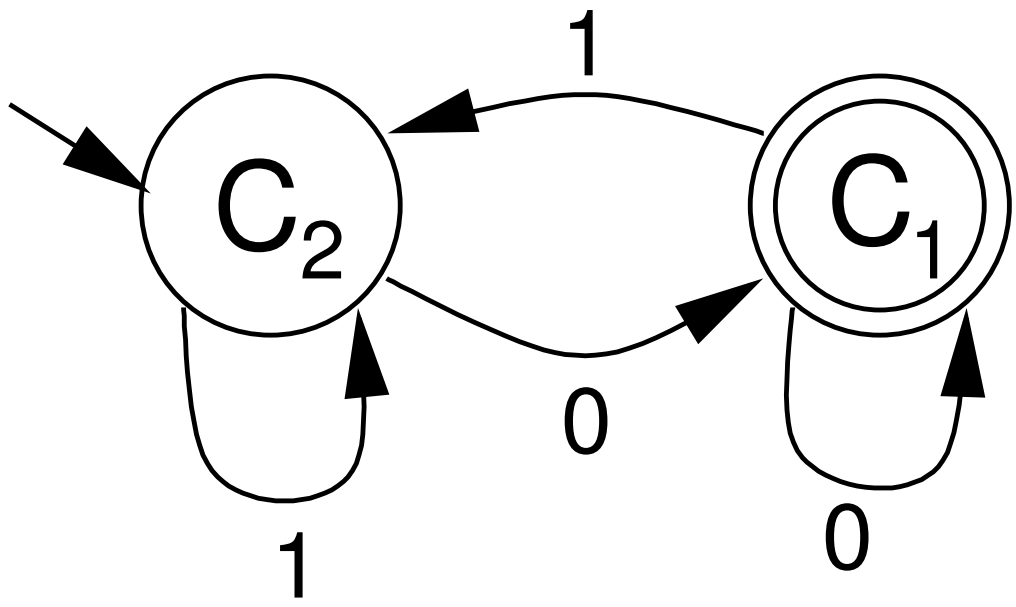
  2. Para a partição C_2:

	s0	s2
0	C_1	C_1
1	C_2	C_2

  $\Rightarrow$ Estados $s0, s2$ são redundantes

Minimização de estados

Exemplo: resultado



Analisadores léxicos

- Autômato de reconhecimento de strings é o núcleo do programa que faz a análise léxica

Analisador léxico

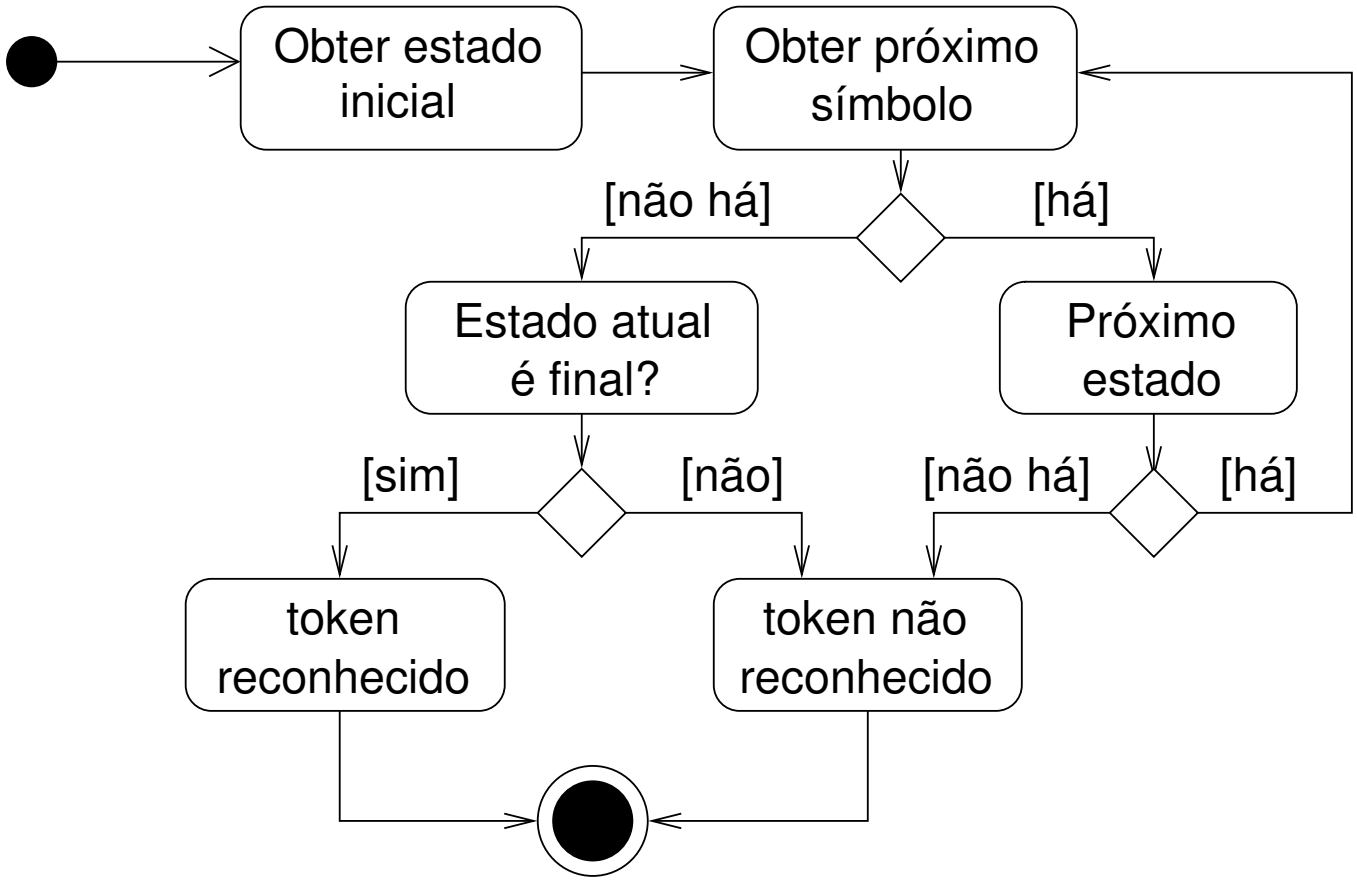
O analisador léxico recebe o nome de um arquivo que contém uma seqüência de caracteres.
Para cada token que é extraído desse arquivo, retorna a indicação de se o token é válido e qual é a sua classe.
O analisador encerra a execução normalmente após a análise do último token.
Caso algum token não seja reconhecido, o analisador indica a situação e passa ao próximo token, se possível; caso contrário, encerra a execução.

Elementos do analisador léxico

Estrutura que implementa o autômato deve contemplar

1. O estado inicial para o autômato;
2. Dado um estado qualquer, indicar se este é um estado final (condição de aceitação); e
3. Dado um estado qualquer e um símbolo, a indicação de qual é o próximo estado.

Algoritmo do analisador léxico



Aspectos de implementação

Como representar os elementos do analisador em C++

Símbolos variáveis do tipo `char`

Estados variáveis do tipo `int`

Estado inicial uma variável do tipo `int`

Estados finais um conjunto de variáveis do tipo `int`

- `set<int>`

Transições uma seqüência de mapeamentos (um elemento por estado) de `char` (símbolo) para `int` (estado)

- `vector< map<char,int> >`

Geradores automáticos

- É possível implementar analisadores léxicos por esta abordagem
  - Obter autômato a partir da expressão regular
  - Implementar programa para cada autômato
- Este procedimento é sistemático — pode ser automatizado
  - Papel dos geradores de analisadores léxicos

Gerador de analisadores léxicos

lex

Programa do sistema Unix, foi o primeiro gerador de analisadores léxicos em C

- Há atualmente diversos programas similares que geram analisadores léxicos para diferentes linguagens
- Forma de operação similar para todos
  - Arquivo de especificação indica padrões (expressões regulares) que devem ser reconhecidos e ações correspondentes
  - O gerador de analisador léxico traduz essa especificação num programa que, ao reconhecer uma string no dado padrão, executa a ação
- Não tem por objetivo gerar uma aplicação completa, mas sim um programa a ser integrado numa aplicação maior

Arquivo de especificação

Primeira seção

```
Definições e declarações para descrição de
padrões ou para o programa gerado
```

```
%%
```

Segunda seção

```
Padrões que devem ser reconhecidos e ações
correspondentes
```

```
%%
```

Terceira seção

```
Código complementar do usuário
```

Especificação dos padrões

- Utiliza uma extensão da forma de especificação de expressões regulares
- Além das construções primitivas para descrever uma expressão regular, pode utilizar

`.`	qualquer caráter (exceto nova linha)
`+`	uma ou mais ocorrências do padrão anterior
`?`	zero ou uma ocorrência do padrão anterior
$\{n\}$	n ocorrências do padrão anterior
$[\cdots]$	classe de caracteres
$[\hat{}\cdots]$	classe negada
`^...`	padrão no início da linha
`...$`	padrão no fim da linha
`<<EOF>>`	fim do arquivo de entrada

Definições e declarações

Para a descrição de regras

- É possível associar um nome a um fragmento de padrão que aparece muitas vezes na seção de regras

```
nome    fragmento_padrão
```

- O nome pode ser usado na especificação dos padrões na seção de regras, entre chaves

```
...{nome}...
```

Para o código gerado

- Fragmentos de declarações e definições em C podem ser incluídos no programa gerado

```
%{
...
%}
```

Ações

- Fragmentos de código C que serão executados quando o padrão for reconhecido no arquivo de entrada
- Especificados após o padrão
  - Se usam mais de um comando C, ação deve estar entre chaves

Integração com código

- Especificação é transformada numa função C (`yylex()`)
  - Varre os caracteres de um arquivo de entrada, `yyin`
  - Retorna um valor inteiro, tipicamente associado ao reconhecimento de um padrão
  - String reconhecida por um padrão é apontada pela variável `yytext`
  - String não associada a padrão algum é ecoada para um arquivo de saída, `yyout`
  - Valores iniciais de `yyin` e `yyout`: entrada e saída padrão, respectivamente
- Programador precisa fornecer o programa que invoca `yylex`
  - Implementação básica é fornecida numa biblioteca do gerador de analisador léxico

Exemplo de aplicação

Objetivo Obter valores inteiros e apresentar o valor acumulado ao final

Papel do analisador léxico Reconhecer as strings que representam valores inteiros e converter esses valores para inteiros

O que o analisador não faz Acumular os valores

Exemplo de código para a aplicação

Seções 1 e 2

```
%{
#include <iostream>
using namespace std;
#define END -1
%}
%%
[0-9]+ return atoi(yytext);
<<EOF>> return END;
%%
```

Exemplo de código para a aplicação

Seção 3

```
int main(int argc, char *argv[]) {
   int total=0, valor=0;

   do {
      valor = yylex();
      if (valor != END)
         total += valor;
   } while (valor != END);

   cout << "Total: " << total << endl;
}
```

Geração e execução da aplicação

- Gerar o analisador léxico a partir do arquivo de especificação `acumul.l`

  ```
  > flex acumul.l
  ```

- Compilar o arquivo C gerado (`lex.yy.c`) e fazer a ligação com as rotinas auxiliares da biblioteca `libfl.so`; criar o arquivo executável `acumul`

  ```
  > g++ -o acumul lex.yy.c -lfl
  ```

- Executar a aplicação

  ```
  > ./acumul
  ```

Sugestão de leitura (Web)

- Flex - a scanner generator (manual)
  `http://www.gnu.org/software/flex/manual/`
- flex: The Fast Lexical Analyzer (software)
  `http://flex.sourceforge.net/`