Capítulo 3

Análise Lexical

A análise lexical é a primeira fase de um compilador. A sua tarefa é a de ler a sequência de caracteres do *canal de entrada* e produzir uma sequência de palavras (lexemas).

As expressões regulares e as gramáticas regulares são mecanismos bastante apropriados para descrever a sintaxe das linguagens de programação.

Trata-se agora de, dado uma linguagem, ser capaz de reconhecer de forma automática se uma dada frase pertence a essa linguagem. Esta tarefa é designada por *reconhecimento automático*, na terminologia inglesa temos o termo "parsing". Os programas que implementam essa tarefa são os reconhecedores ("parsers").

Definição 3.1 (Reconhecedor ("Parser")) *Seja L uma linguagem, tal que $L \subseteq V^*$, supondo R_L o reconhecedor de L e $l \in V^*$ a frase a reconhecer, então tem-se que:*

$$R_L(l) = \begin{cases} \text{aceita}, & l \in L \\ \text{erro}, & l \notin L \end{cases}$$

O reconhecedor além de fazer o reconhecimento da linguagem vai construir uma árvore sintáctica, a qual representa a estrutura sintáctica da frase. Nessa árvore cada nó representa um construtor da linguagem (não-terminal) e os seus filhos representam as componentes desse construtor. As folhas representam os símbolos básicos da linguagem.

3.1 Autómatos Finitos

A construção de um reconhecedor é feita normalmente através da construção de um *autómato finito*, isto deve-se a que:

- todo o autómato finito define uma linguagem regular;

[1](Versão 1.14)

- toda a linguagem regular pode ser definida por um autómato finito;
- existem métodos para converter gramáticas regulares e expressões regulares em autómatos finitos;
- existem técnicas bem definidas (ferramentas computacionais) para converter autómatos finitos (determinísticos) em programas de computador.

Definição 3.2 (Autómato Finito) *Um autómato finito A é um quíntuplo $A = (T, Q, I, F, \delta)$ em que:*

- *T é o abecedário, ou seja um conjunto de símbolos;*
- *Q é um conjunto finito, não vazio, de estados;*
- *$I \subseteq Q$ é um conjunto, não vazio, de estados iniciais;*
- *$F \subseteq Q$ é um conjunto, não vazio, de estados finais;*
- *$\delta \subseteq (Q \times (T \cup \{\epsilon\})) \times Q$ é um conjunto de transições de estados.*

Uma transição $((q, s), q\prime) \in \delta$, com $q, q\prime \in Q$ e $s \in (T \cup \{\epsilon\})$ deve ler-se: o autómato transita do estado q para o estado $q\prime$, através do reconhecimento do símbolo s. As transições pelo símbolo ϵ designam-se por transições-ϵ.

Exemplo 3.1 *Um exemplo:*

$$A = (T, Q, I, F, \delta)$$

com:

$$\begin{aligned} T &= \{+, -, \cdot, 1\} \\ Q &= \{A, B, C, D, E\} \\ I &= \{A\} \\ F &= \{E\} \\ \delta &= \left\{ \begin{array}{l} ((A,+),B),((A,-),B),((A,\epsilon),B), \\ ((B,1),C),((C,\epsilon),B),((C,\cdot),D), \\ ((C,\epsilon),E),((D,1),E),((E,\epsilon),D) \end{array} \right\} \end{aligned}$$

Esta forma de representar um autómato finito, nomeadamente o conjunto de transições, é muito pouco intuitiva, não sendo fácil perceber que frases é que são reconhecidas pelo autómato.Temos então duas alternativas.

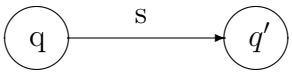
Tabela de Transições de Estados: a representação do conjunto de transições em forma de tabela de duas entradas, nas linhas os elementos de Q,

nas colunas os elementos de $T \cup \{\epsilon\}$, e nas células conjuntos de elementos de Q formados pelos estados de chegada das diferentes transições. Para o exemplo acima descrito têm-se:

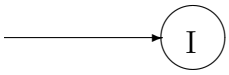
	$+$	$-$	$\cdot$	1	ϵ
A	B	B			B
B				C	
C			D		$\{E, B\}$
D				E	
E					D

Grafo Etiquetado: a representação do conjunto de transições em forma de grafo orientado e etiquetado, os nós são os elementos de Q, as etiquetas dos arcos orientados são os elementos de $T \cup \{\epsilon\}$.

- cada transição $((q, s), q')$ são representadas por:

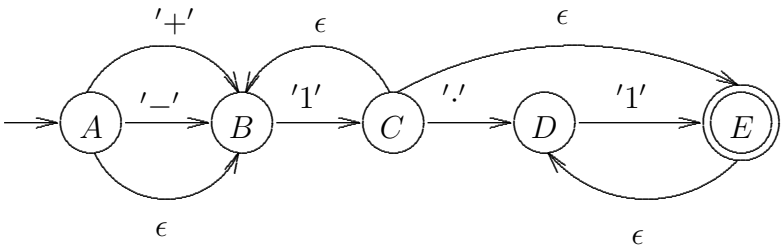


- os símbolos iniciais são representados por:



- os símbolos terminais são representados por:

O grafo final obtêm-se por composição de todos os sub-grafos referentes às diferentes transições. Para o exemplo que temos vindo a usar ter-se-ia:

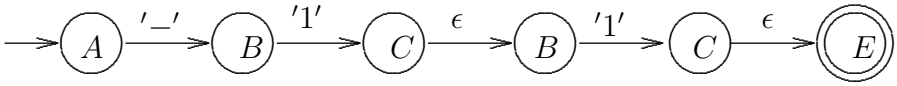


Para um autómato finito ser considerado como um reconhecedor de uma linguagem tem de considerar como *aceites* as frases válidas da linguagem e como *não aceites* todas as outras. No caso da representação do autómato finito como grafo orientado as frases aceites correspondem aos diferentes caminhos possíveis no grafo.

Definição 3.3 (Reconhecimento por um Autómato Finito) *Sendo $A = (T, Q, I, F, \delta)$ um autómato finito e $\gamma \in T^*$, diz-se que A aceita γ sse existir um caminho de um estado inicial para um estado final tal que a frase que se obtém concatenado as etiquetas do grafo é igual a γ.*

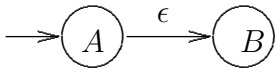
Vejamos se as frases $f_1 = -11$ e $f_2 = .11$ são ambas aceites pelo autómato A.

Para f_1 temos (entre outros):



a frase é reconhecida, partindo de um estado inicial, chegou-se a um estado final.

Para f_2 temos como único caminho:



como B não é um estado final temos uma situação de erro, isto é de não reconhecimento da frase em questão.

Definição 3.4 (Não reconhecimento) *Um autómato finito não aceita uma frase $f = a_1 \dots a_n$ sempre que num estado $q \in Q$ se está a tentar reconhecer um símbolo a_i e se verifica que $((q, a_i), q') \notin \delta$ e $((q, \epsilon), q'') \notin \delta$, para quaisquer $q, q', q'' \in Q$ e um qualquer $a_i, 1 \leq i \leq n$, ou sempre que se atinja o fim da frase num estado não terminal*

O conjunto de todas as frases aceites por um AF constitui a linguagem definida pelo AF.

Definição 3.5 (Linguagem definida por um AF) *A linguagem L_A definida por um AF A é definida como sendo o conjunto de todas as frases $a_1 \dots a_n \in T^*$, tal que existem $q_o, \dots, q_n \in Q$, com $q_o \in I$, $q_n \in F$, e $((q_{i-1}, a_i), q_i) \in \delta$ para todo o $1 \leq i \leq n$.*

Dois autómatos finitos dizem-se equivalentes se definirem a mesma linguagem.

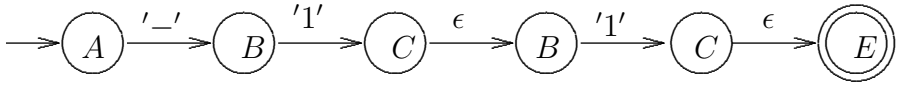
Definição 3.6 (AF equivalentes) *Dois autómatos finitos A_1 e A_2 dizem-se equivalentes, $A_1 \equiv A_2$, sse $L_{A_1} = L_{A_2}$.*

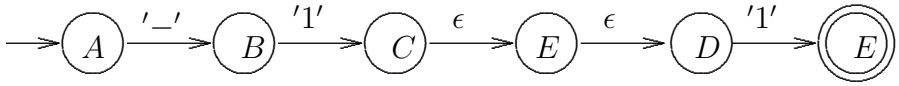
3.1.1 Autómatos Finitos Não Determinísticos

Os autómatos podem-se dividir em autómatos finitos não determinísticos (AFND), e autómatos finitos determinísticos (AFD).

Informalmente um AFND é um AF tal que existe pelo menos um estado com mais do que uma transição possível, ou em que existe pelo menos uma transição-ϵ.

Por exemplo, para o AF do exemplo 3.1 temos (pelo menos) duas derivações diferentes para a frase $f_1 = -11$





Um AFND define uma linguagem regular, no entanto a sua implementação computacional não é fácil (nem eficiente) dado que é necessário implementar um mecanismo de recuperação de situações de erro ("backtracking").

3.1.2 Autómatos Finitos Determinísticos

Vejamos o outro tipo de AF, os autómatos finitos determinísticos (AFD), como o seu nome indica para cada estado só pode haver uma transição possível (no máximo).

Definição 3.7 (Autómato Finito Determinístico) *Um AF* $A = (T, Q, I, F, \delta)$ *diz-se um autómato finito determinístico, AFD, se:*

- $|I| = 1$ *(um só estado inicial);*
- *para todo o* $q \in Q$ *e* $a \in T$*, existe no máximo um* $q' \in Q$ *tal que* $((q, a), q') \in \delta$ *(uma só escolha possível);*
- $\delta \subseteq (Q \times T) \times Q$ *(não existem transições-*ϵ*).*

Nos AFD a aceitação de uma frase f corresponde a um, e um só, caminho no grafo.

Aspectos positivos

- mais fáceis de implementar;
- mais eficientes.

Aspectos Negativos

- uma maior número (em geral) de transições.

3.1.3 Conversão de uma Gramática Regular num AFND

É possível converter uma gramática regular $G = (T, N, I, P)$ num autómato finito não determinístico $A = (T, Q, I, F, \delta)$ de tal forma que $L_G = L_A$.

Dado uma gramática $G = (T, N, I, P)$ regular à direita, isto é, as produções são do tipo $A \to a$, ou $A \to aB$, com o único símbolo não terminal do lado direito da produção o mais à direita possível, então é possível a sua conversão num autómato finito $A = (T, Q, I, F, \delta)$ não determinístico segundo o seguinte conjunto de regras:

1. $T = T$;
2. A cada não-terminal $X \in N$ corresponde um estado $X \in Q$. A I em G faz-se corresponder I em A;
3. A cada produção da forma $X \to a_1 \dots a_n Y$, com $X, Y \in N$ e $a_1, \dots, a_n \in T$ corresponde o seguinte conjunto de transições:

   $$((X, a_1), X_1), (X_1, a_2), X_2), \dots, (X_{n-1}, a_n), Y)) \in \delta$$

   em que $X_1, \dots, X_{n-1} \in Q$ são novos estados;
4. A cada produção da forma $X \to a_1 \dots a_n$ com $X \in N$, $a_1, \dots, a_n \in T$ corresponde o seguinte conjunto de transições:

   $$((X, a_1), X_1), (X_1, a_2), X_2), \dots, (X_{n-1}, a_n), Z_0)) \in \delta$$

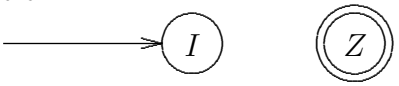
   em que $X_1, \dots, X_{n-1}, Z_0 \in Q$ são novos estados, com $Z_0 \in F$ um novo estado final;
5. A cada produção da forma $X \to Y$ corresponde a seguinte transição $((X, \epsilon), Y) \in \delta$.

A conversão de uma gramática regular à esquerda é feita de modo semelhante.

3.1.4 Conversão de uma Expressão Regular num AFND

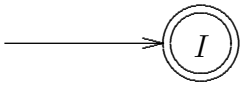
Dado uma expressão regular e geradora da linguagem L_e é possível obter um AFND $A = (T, Q, I, F, \delta)$ tal que $L_e = L_A$. A conversão é feita segundo o seguinte conjunto de regras, p e q designam expressões e AFND_p significa a inclusão nesse ponto do AFND referente à conversão da expressão regular p:

1. $e = \emptyset$ converte-se em

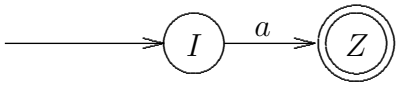
   

   com $Z \in F$;

2. $e = \epsilon$ converte-se em:

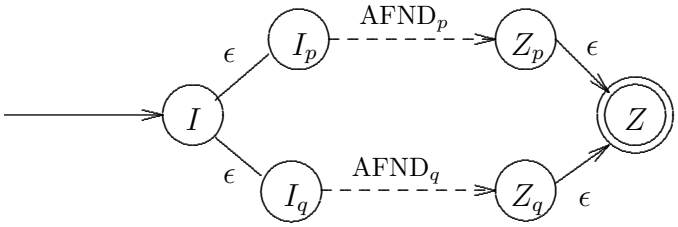


3. $e = a$ converte-se em:

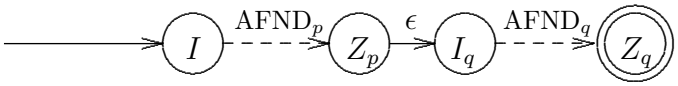


   com $a \in T$;

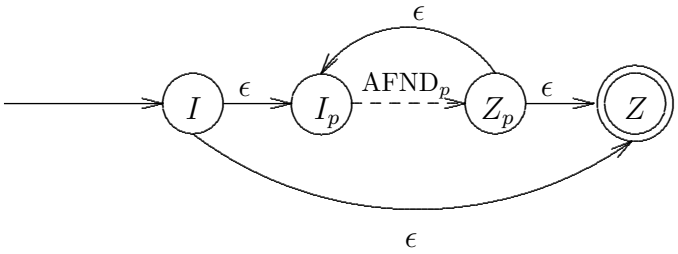
4. $e = p + q$ converte-se em:



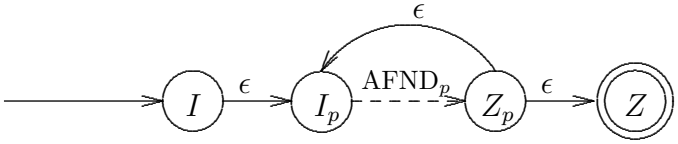
5. $e = pq$ converte-se em:



6. $e = p^*$ converte-se em:



7. $e = p^+$ converte-se em:



Exercício 3.1 *Construa o AFND correspondente à expressão regular* $e = a(bc^* + d)^+$.

3.1.5 Conversão de um AFND num AFD

Por razões de eficiência os AFND devem ser convertidos em AFD. É possível converter um AFND num AFD de uma forma sistemática, assim como é possível converter uma gramática regulares e expressões regulares em AFD. Vamos ver de seguida como converter um AFND num AFD.

Trata-se de converter um AFND num AFD que defina a mesma linguagem.

- Cada estado do AFD corresponde a um conjunto de estados do AFND, isto é, cada estado de um AFD contém a informação que se pode alcançar a partir de um AFND transitando por um dado símbolo.
- Cada estado do AFD incluí ainda todos os estados que se alcançam no AFND por transições-ϵ.

O número de estados de um AFD pode ser (em geral é) muito superior ao do AFND, na pior das situações pode ser exponencialmente superior.

Pretende-se então de um dado AFND $A_{nd} = (T, Q, I, F, \delta)$ obter um AFD $A_d = (T, Q, I, F, \delta)$ tal que $A_{nd} \equiv A_d$. Antes de mais é útil definir uma função auxiliar, a função *fecho-*ϵ, informalmente esta função determina o conjunto de estados que se atingem a partir de um dado conjunto de estados a partir de transições-ϵ (incluindo o próprio estado).

Definição 3.8 (Função *fecho-*ϵ)

$$\begin{array}{rcccl} \text{fecho} - \epsilon & : & 2^Q & \longrightarrow & 2^Q \\ & & X & \longmapsto & X \cup \bigcup_{q \in X} \text{fecho} - \epsilon(\delta(q, \epsilon)) \end{array}$$

Considerando dois autómatos finitos A (AFND) e A' (AFD) temos:

- $T' = T$;
- $I' = \text{fecho} - \epsilon(\{I\})$;
- $\delta'(X, t) = \text{fecho} - \epsilon(\bigcup_{q \in X} \delta(q, t))$;
- Q' define-se recursivamente do seguinte modo:
  - $I' \in Q'$
  - $X \in Q' \wedge \delta'(X, t) \neq \emptyset \Rightarrow \delta'(X, t) \in Q'$;
- $F' = \{X \in Q' \mid X \cap F \neq \emptyset\}$.

Dito de outra forma.

- O vocabulário do AFD T' é o mesmo do que o AFND.

- O estado inicial do AFD S' é o conjunto formado pelos estados que se alcançam a partir do estado inicial do AFND por transições-ϵ.
- O conjunto das transições δ' obtém-se do seguinte modo:
  - para cada estado X do AFD e para cada símbolo do vocabulário t obtém-se, caso exista, uma transição para um novo estado. Este novo estado é obtido por aplicação da função *fecho-*ϵ à união dos estados que se obtém no AFND a partir dos estados que constituem X, transitando por t.
- O conjunto de estados Q' do AFD é constituído pelo seu estado inicial e ainda por todos os estados que se alcançam em transições de um estado de Q' por um símbolo do vocabulário.
- O conjunto de estados finais Z' é formado por todos os estados do AFD que incluem algum estado final do AFND.

Uma forma de sistematizar o processo de conversão é dado pela construção de uma tabela de conversão.

Tabela de Conversão de um AFND num AFD Vai-se construir uma tabela de duas entradas em que:

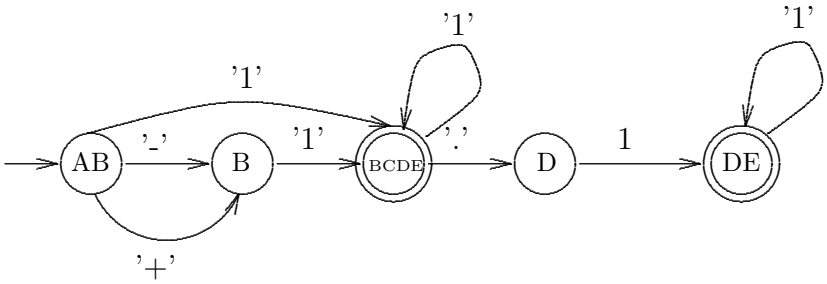
- como índices das linhas temos os estados do AFD;
- como índices das colunas temos os símbolos do abecedário;
- nas diferentes entradas vão ficar as transições associadas aos estados do AFD e aos símbolos do abecedário.

O preenchimento vai ser feito de forma indutiva:

- O primeiro índice para a primeira linha é dado pelo fecho-ϵ do estado inicial do AFND, o qual passará a ser o estado inicial (único) do AFD;
- De seguida preenchem-se os elementos da primeira linha por cálculo do fecho-ϵ dos estados que se obtêm por transições no AFND correspondentes aos diferentes estados iniciais e aos símbolos correspondentes a cada uma das colunas. Neste processo são (eventualmente) criados novos estados para o AFD;
- Os novos estados do AFD criados no processo anterior dão origem a novas linhas da tabela, repetindo-se o processo de cálculo das transições respectivas;
- O processo para quando num dos passos descritos anteriormente não é criado mais nenhum novo estado para o AFD.

Vejamos a tabela que se obtêm para o exemplo 3.1

	'+'	'-'	'1'	'.'
I'=fecho-ϵ(A) ={A,B}	fecho-ϵ({B}) ={B}	fecho-ϵ({B}) ={B}	fecho-ϵ({C}) ={B,C,D,E}	—
{B}	—	—	fecho-ϵ({C}) ={B,C,D,E}	—
{B,C,D,E}	—	—	fecho-ϵ({C,E}) ={B,C,D,E}	fecho-ϵ({D}) ={D}
{D}	—	—	fecho-ϵ({E}) ={D,E}	—
{D,E}	—	—	fecho-ϵ({E}) ={D,E}	—



3.2 Codificação em *C* de um *AFD*

A codificação de um AFD numa dada linguagem de programação passa pela implementação da tabela de transições de estados e de um algoritmo de reconhecimento muito simples baseado na referida tabela.

A tabela pode ser implementada muito simplesmente através de uma tabela bidimensional com elementos e índices apropriados.

$$TT = M_{Q\times T}(Q \cup \{\text{erro}\}) = \texttt{array}[Q \times T[\ \texttt{of}\ Q \cup \{\text{erro}\}$$

O elemento "erro" corresponde às entradas da tabela de transições para as quais não há nenhum estado definido, isto é, correspondem a situações de erro no reconhecimento.

O algoritmo de reconhecimento para um AFD é dado então pelo seguinte algoritmo:

```
função R_L(δ : TT; γ : T*):{aceita,erro}
alg
  α ← I;
  enquanto (γ ≠ ε) ∧ (α ≠ erro) faz
    α ← δ(α, cabeça(γ));
    γ ← cauda(γ))
  fimenquanto;
  se (α ∈ Z) ∧ (γ = ε) então r ← aceita
```

```
    senão r ← erro
    fimse;
    R_L ← r
  fimalg
```

A linguagem de programação *C* não dá-nos a liberdade de escolher o tipo (nem os limites) dos índices da tabela, sendo assim ter-se-ia de implementar a tabela de transições do seguinte modo:

```
int tt[n_estados][n_simb] = {{1,-1,2,-1},
  {-1,3,-1,-1},
  {1,-1,2,-1},
  {4,5,6,-1},
  {4,5,6,-1},
  {4,5,6,-1},
  {4,5,6,-1}};
```

sendo que os valores iniciais vão depender do problema em questão (ver apêndice A).

As etiquetas dos estados do AFD são implementadas através de valores inteiros sendo que 0 nos dá o estado inicial, isso resolve-nos o problema dos índices das linhas assim como dos elementos da tabela, para resolver o problema dos índices das colunas basta implementar uma função, `int simb2int(char s)`, que nos faz a conversão entre os elementos do vocabulário do AFD e os índices das colunas da tabela.

O conjunto Z dos estados terminais é facilmente implementada como uma tabela de inteiros, `int Z[] = {3,4,5,6}`, novamente a inicialização é específica para o AFD em questão.

Posto isto a função de reconhecimento, para sequências dadas a partir do teclado, tomaria a forma:

```
char *reconhece(){
  int estado=INICIAL;
  char *r;
  char simbolo;
  int simb2int(char);
  int e_terminais(int,int[]);

  simbolo=getchar();
  while ((simbolo!='\n') && (estado!=ERRO)) {
    estado=tt[estado][simb2int(simbolo)];
    simbolo=getchar();
  }
  if (e_terminais(estado,Z) && (simbolo=='\n'))
    r="aceita";
```

```
  else
    r="erro";
  return r;
}
```

em que a função `int e_terminais(int,int[])` nos permite verificar se um dado estado é, ou não, um estado terminal.

3.3 Autómatos Reactivos

Em alguns casos torna-se útil extender um dado reconhecedor de forma a que este não se limite a dar a informação "aceita/não aceita", mas possa incorporar acções semânticas que vão sendo executadas à medida que se processa o reconhecimento.

Em (Crespo(1998)) podemos ver duas metodologias nas quais as acções semânticas se limitam à escrita de mensagens de saída, são elas as *Máquinas de Moore* e as *Máquinas de Mealy*, para o nosso estudo interessa-nos o caso mais geral dado pelos *Autómatos Finitos Determinísticos Reactivos*, isto dado ser esse o tipo de autómato implementado pelo programa *Flex* como veremos mais adiante.

Definição 3.9 (AFDR) *Um Autómato Finito Determinístico Reactivo é um sêxtuplo* $A = (T, Q, I, F, R, \delta)$ *em que:*

- T *é o abecedário, ou seja um conjunto de símbolos;*
- $Q \neq \emptyset$ *é um conjunto finito de estados;*
- $I \subseteq Q, I \neq \emptyset$ *é um conjunto de estados iniciais;*
- $F \subseteq Q, F \neq \emptyset$ *é um conjunto de estados finais;*
- R *é um conjunto de acções semânticas;*
- $\delta \subseteq (Q \times T) \times (Q \times R)$ *é um conjunto de transições de estados.*

sendo que:

- $|I| = 1$ *(um só estado inicial);*
- *para todo o* $q \in Q$ *e* $a \in T$*, existe no máximo um* $q' \in Q$ *e* $r \in R$ *tal que* $((q, a), (q', r)) \in \delta$ *(uma só escolha possível);*
- *não existem transições-*ϵ*.*

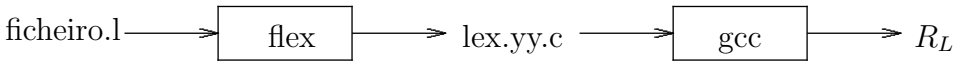
Há semelhança de um AFD pode-se representar graficamente um AFDR através de um grafo etiquetado, neste caso cada seta terá uma etiqueta dada por um par $< s, r >$ em que s é o símbolo a reconhecer, e r é a reacção (acção semântica). Sempre que, para um dado autómato D, se tenha uma transição $< q, s > \mapsto < q', r >$ diz-se, "o autómato D transita do estado q para o estado q' executando a acção semântica r, através do reconhecimento do símbolo s.

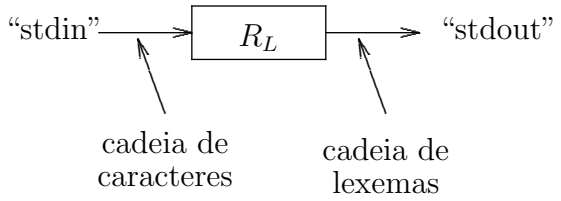
O *Flex* é uma ferramenta computacional que a partir de uma especificação para um AFDR gera automaticamente um analisador léxico capaz de reconhecer as frases da linguagem regular definida pelo AFDR.

3.4 *Flex*

O *Flex* (Mason and Brown(1991); Paxson(2000)) é uma ferramenta que a partir de uma especificação em termos de expressões regulares gera um autómato finito capaz de reconhecer as expressões especificadas.

O seu modo de utilização/funcionamento pode ser descrito pelos seguintes diagramas.





Um ficheiro de especificação em "flexês" é dividido em três secções, tendo como separadores uma linha com os caracteres "%%". Algo como:

```
⋮     ⟵ Secção das Definições
%%
⋮     ⟵ Secção das Regras de Emparelhamento
%%
⋮     ⟵ Secção das Rotinas do Utilizador
```

É de notar que os caracteres "%%" têm de ser escritos exactamente na primeira coluna da linha em que são escritos.

3.4.1 Regras de escrita

Nas duas primeiras secções o *flex* interpreta todas as linhas que começam na primeira coluna e ignora todas as linhas que começam após a primeira

coluna (i.e. o primeiro caracter que não o espaço está após a primeira coluna), temos então:

- toda a linha que começa por um espaço ou um "tab" (espaço tabular) não é interpretada pelo *flex*, essas linhas são copiadas integralmente, como código *C* para o ficheiro `lex.yy.c`;
  - se o código aparece na primeira secção é considerado externo a qualquer função;
  - se aparece na segunda secção é considerado local a `yylex`.
- todo o texto (código *C*) colocado entre um par de linhas que contenham `%{` e `%}` (e só isso) respectivamente é copiado integralmente para `lex.yy.c`. Este aspecto permite a inclusão de instruções para pré-processamento através do pré-processador do *gcc*, por exemplo a inclusão das directivas `#include`, as quais têm de ser colocadas exactamente na primeira coluna das linhas em que são escritas (o que entrava em conflito com as restantes regras de escrita do *flex*.
- comentários são definidos como para os programas em *C*, isto é, entre `/*` e `*/`. Na secção de definições os comentários podem ocupar mais do que uma linha.

3.4.2 Definição de Nomes

A primeira secção vai servir para incluir código global, directivas para pré-processamento, e definições de nomes. A definições de nomes não acrescenta nada de novo tendo como única função facilitar a escrita das regras de emparelhamento a definir na segunda secção.

Temos a seguinte sintaxe para a definição de nomes:

$< nome >$ $< expressão\ regular >$

em que *nome* é dado por uma sequência de caracteres começada por uma letra ou por uma barra horizontal "_", seguida de letras, barras ("_") e sinais menos ("-") entre o nome e a expressão regular tem de existir um (ou mais) espaço(s).

Posteriormente vamos poder referir os nomes definidos nesta secção através da sintaxe $< nome >$.

Por exemplo:

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

definem dois nomes que posteriormente podem ser referenciados, por exemplo: escrever

`{DIGIT}+"."{DIGIT}*` é equivalente a escrever `([0-9])+"."([0-9])*`

3.4.3 Expressões Regulares

Os padrões que definem as expressões regulares em *flex* são dadas na tabela seguinte. Antes de mais é necessário ter em conta que os seguintes caracteres são meta-caracteres, isto é, têm um significado próprio:

`", \, [, ], (, ), <, >, {, }, ^, -, ?, ., *, +, |, $, /, %`

`x`	o caracter 'x'
`"x"`	o caracter 'x', mesmo que este seja um meta-caracter
`.`	todos os caracteres excepto '\n' ("newline")
`[xyz]`	'x', ou 'y', ou 'z'
`[x-z]`	todos os caracteres entre 'x' e 'z' (classe)
`[^x-z]`	todos os caracteres excepto os entre 'x' e 'z'
`r*`	a expressão regular r, zero ou mais vezes
`r+`	r, uma ou mais vezes
`r?`	r opcional
`r{n}`	r n vezes
`r{n,}`	r n ou mais vezes
`r{n,m}`	r n a m vezes
`{name}`	a expansão da definição referente a *nome*.
$\backslash x$	Se x é igual a 'a', 'b', 'f', 'n', 'r', 't', ou 'v', então tem-se a interpretação ANSI-C, caso contrário dá-nos o caracter 'x' (necessário para se poder aceder aos meta-caracteres).
$\backslash 0$	o caracter nulo (NUL, código ASCII 0).
$\backslash 123$	o caracter com valor octal 123.
$\backslash x2a$	o caracter com valor hexadecimal 2a
`(r)`	a expressão regular r; necessário para contornar as regras de precedência, ver mais a baixo.
`rs`	concatenação das expressões regulares r e s
`r|s`	r ou s.
`r/s`	r mas só se for seguindo de um s.
`^r`	r no princípio de uma linha.
`r$`	r no fim de uma linha.
`<s>r`	r, mas só numa condição inicial.
`<s1,s2>r`	r, mas só nas condições iniciais s1, ou s2.
`<*>r`	r, numa qualquer condição inicial.
`<<EOF>>`	um fim-de-linha.
`<s1,s2><<EOF>>`	um fim-de-linha nas condições iniciais s1 ou s2.

Notas:

- dentro de uma classe de caracteres todos os meta-caracteres perdem os seus significados especiais, excepção feita a "\", "`-`", "`]`", e no começo da classe "`^`";
- a tabela anterior está organizada segundo uma ordem decrescente de prioridades. Os parêntesis têm a sua função normal de se sobrepor às precedências.

Além dos caracteres e das sequências de caracteres as classes de caracteres podem também conter *expressões de classes de caracteres*, as quais só podem ocorrer dentro das classes de caracteres, isto é, entre `[` e `]`. As expressões existentes são:

```
[:alnum:] [:alpha:] [:blank:]
[:cntrl:] [:digit:] [:graph:]
[:lower:] [:print:] [:punct:]
[:space:] [:upper:] [:xdigit:]
```

com o significado óbvio.

Exemplo 3.2 *Vejamos um pequeno exemplo: quer-se identificar como "número" as sequências do tipo*

```
100    1.1   1    11.10
```

temos o seguinte ficheiro/especificação flex*,* `numero.l`*.*

```
%%
([0-9]+\.)?[0-9]+     printf("%s","Reconheci um numero");
.                     ;
```

A segunda regra tem como efeito o parar do efeito de eco para todas as expressões não reconhecidas que é o procedimento normal do flex*.*

O *flex* pode ser usado de duas formas distintas: como um processo isolado, de forma a produzir um reconhecedor lexical; ou como um primeiro passo na construção de um reconhecedor sintáctico através da utilização de um outro programa, nomeadamente o *bison*.

O primeiro passo (para ambos os casos) é dado pelo processamento do ficheiro `numero.l` através do programa *flex*.

```
> flex numero.l
```

O qual, se não houver erros na especificação, dá origem a um ficheiro `lex.yy.c`, este ficheiro não contém (no caso do exemplo) um programa em *C* completo, contém a rotina que constitui o reconhecedor `yylex()`, não contém no entanto a rotina principal `main`. Podemos proceder das seguintes formas:

- completar o programa escrevendo (na terceira secção) a rotina `main`, a qual tem de chamar a rotina `yylex`;
- compilar o programa recorrendo à biblioteca `fl`, a qual providência as rotinas necessárias para criar um programa isolado;
- emparelhar o programa *flex* com o programa *bison*.

Vejamos para já o criar de um programa isolado através da utilização da biblioteca `fl`. Teríamos então os seguintes passos:

```
> flex numero.l
```

e

```
> gcc lex.yy.c -lfl -o numero
```

Após este dois passo obteríamos um programa isolado que recebendo os seus dados do "stdin" envia os resultados para o "stdout".

3.4.4 Resolução de Ambiguidades

Pode acontecer que a mesma cadeia de caracteres possa ser emparelhadas por mais do que uma expressão regular. Por exemplo, `ab12cd`, é um só identificador ou é um identificador, seguido de um número, seguido de outro identificador. Temos então de saber como resolver as ambiguidades que podem ocorrer no processamento da sequência de entrada por parte dos reconhecedores criados através da utilização do programa *flex*, esse reconhecimento é feito de acordo com as seguintes regras:

1. o analisador léxico tenta reconhecer a cadeia de caracteres de maior comprimento possível;
2. quando a mesma cadeia de caracteres é descrita por duas expressões regulares distintas, o analisador léxico selecciona a que aparece em primeiro lugar.

Por exemplo, a seguinte especificação:

```
%%
zip         printf("Zip");
zip-zip     printf("ZIP-ZIP");
```

produz o seguinte resultado:

```
zip         ← entrada
Zip
zip-zip     ← entrada
ZIP-ZIP
```

resolvendo a ambiguidade entre as duas expressões regulares através da regra 1.

Como já foi referido atrás o procedimento usual dos reconhecedores produzidos com auxílio do *flex*, é o de fazerem o eco de tudo o que não é reconhecido. Se quisermos que tal não aconteça é necessário acrescentar a seguinte regra como sendo a última regra da especificação:

```
.       ;
```

em que a expressão regular "`.`" associa a qualquer caracter (excepto a mudança de linha) e por outro lado a instrução "`;`" dá-nos a instrução nula.

Se quisermos que nem as mudanças de linha sejam ecoadas teremos de acrescentar a seguinte regra:

```
\n      ;
```

ou juntando ambas as regras:

```
(.|\n)      ;
```

3.4.5 Variáveis, "Macros" e Funções pré-definidas

Ao utilizar-se o *flex* vamos ter acesso a um conjunto de variáveis, "macros", e funções as quais podemos, e devemos, usar de forma a construir os nossos reconhecedores.

Variáveis Globais

`char yytext[]`	Tabela que contém a cadeia de caracteres reconhecida
`int yyleng`	Comprimento da cadeia de caracteres reconhecida
`int yylineno`	Número da linha do ficheiro de entrada
`extern yyval`	variável de "transporte do valor léxico", a ser usada pelo *bison*
`FILE *yyin`	definição do ficheiro de leitura
`FILE *yyout`	definição do ficheiro de escrita

Acções pré-definidas ("macros")

`ECHO`	`printf("%s",yytext)`
`REJECT`	devolve os caracteres lidos ao canal de entrada e analisa as regras seguintes.

Funções Auxiliares

`char input()`	recolhe o próximo caracter
`unput(char c)`	devolve o caracter 'c'
`output(char c)`	envia para o canal de saída o caracter 'c'
`yymore()`	a expressão seguinte é concatenada à expressão corrente
`int yywrap()`	função invocada quando é detectado o fim de ficheiro

3.4.6 Utilização

Como já foi dito a utilização do programa *flex* é feita de dois modos diferentes, ou como processo isolado tendo em vista a criação de um reconhecedor lexical, ou como um primeiro passo na criação de um reconhecedor sintáctico através do programa *bison*.

Como processo isolado a forma de o fazer é aquela que foi descrito logo no inicio desta secção, isto é, escreve-se uma especificação da linguagem que se quer tratar em termos de expressões regulares, através do programa *flex* converte-se esta especificação numa rotina em *C*, a rotina `yylex()`, compila-se esta rotina adicionado-lhe a biblioteca `fl`, produzindo-se deste modo um programa que implementa o reconhecedor para a linguagem expressa pela expressão regular definida no ficheiro *flex*. Podemos como variante decidir escrever nós próprios a rotina `main`, vejamos um exemplo:

```
        int num_lines = 0, num_chars = 0;

%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;

%%
main()
        {
        yylex();
        printf( "# de linhas = %d, # de caracteres = %d\n",
                num_lines, num_chars );
        }
```

Esta especificação *flex* implementa um reconhecedor que conta o número de caracteres e de linhas de um ficheiro. É de notar que a invocação do reconhecedor dentro da rotina `main` é feito através da chamada da rotina `yylex()`, e que ao escrevermos a nossa própria rotina `main` tal facto é detectado pelo *flex* deixando este de gerar automaticamente a sua versão da rotina `main`.

O outro modo de utilização do programa *flex* é nos dado pela incorporação da rotina `lex.yy.c` num programa escrito para o programa *bison*, nesse caso é necessário estabelecer a comunicação entre os dois processos (analisadores lexical e sintáctico) incorporando no ficheiro *flex* instruções de `return` as quais vão ter, ou um símbolo que identifica a palavra, ou o valor da palavra através da variável `yyval`. O tratamento deste caso será feito mais à frente quando falarmos do programa *bison*, só como ilustração deste modo de funcionamento apresenta-se de seguida um pequeno exemplo.

- ficheiro exemplo.l (*flex*)

```
%{
#include "exemplo.tab.h"
%}

number [0-9]+
%%
[ ]          { /* Salta espacos em branco */ }
{number}     { sscanf(yytext,"%d",&yylval);
               return NUMBER;}
\n|.         {return yytext[0];}
```

- ficheiro exemplo.y (*bison*)

```
%token NUMBER

%%
line  : expr '\n'       { printf("%d\n",$1); }
      ;
expr  : expr '+' term   { $$ = $1+$3;}
      | term
      ;
term  : term '*' factor { $$ = $1*$3;}
      | factor
      ;
factor : '(' expr ')'   { $$ = $2; }
       | NUMBER
       ;
%%
#include "lex.yy.c"
```

Como se pode ver os dois ficheiro partilham informação, a forma de obter um programa que implemente o reconhecedor sintáctico passa pela utilização dos programas *bison*, *flex*, e de um compilador de *C*, veremos como o fazer mais à frente.