Capítulo 4.

Análise léxica

1. Definição

Definição : O analisador léxico é uma função que transforma sequências de caracteres em sequências de símbolos (palavras ou "*tokens*").

"Desleitura" de um carácter

Passagem de um *token* e seus atributos

Exemplos de tokens :

Palavras reservadas do Pascal (begin, function, …), operadores (+, <=, …), identificadores, constantes, sinais de pontuação, … .

Por exemplo, a sequência : 45 * 10
será convertida em : < NUM, 45 > < *, > < NUM, 10 >
onde o atributo do *token* NUM é o seu valor numérico.

2. Métodos

Podem-se usar 2 métodos :

- Utilização de uma ***tabela de símbolos***, onde são registados os identificadores e as palavras-chave do texto fonte.
- Descrição dos símbolos através de ***expressões regulares***.

2.1. Tabela de símbolos

O atributo do *token* ID pode ser o endereço do elemento da tabela, onde está registada a cadeia de caracteres que descreve o identificador.

As operações de gestão de uma tabela de símbolos são as seguintes :

- Inicialização : com as palavras-chave e os identificadores-standard
- Pesquisa de um símbolo (fornece o endereço do registo que o contém)
- Inserção de um novo símbolo (e informação associada)

As tabelas de símbolos são exemplos típicos de utilização de endereçamento calculado (Hashing).

2.2. Descrição através de expressões regulares

Este método consiste em traduzir os tokens através de expressões regulares, e utilizar autómatos finitos para os reconhecer.

Construção do autómato (algoritmo) ***A*** = (Q, T, M, q_0, H) :

```
q ← estado_inicial
Repetir
   q ← delta (q, c)
   Se ( q ∉ H )          { H = conj. estados finais do autómato }
   Então
      guardar (c)
      ler (c)
Até ( q ∈ H)
Código_símbolo ← f (q)
```

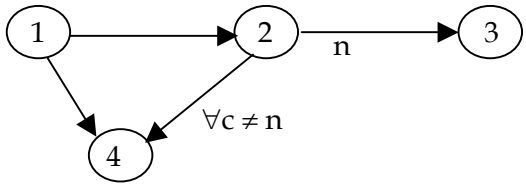
Na implementação do autómato, os estados q são inteiros, com estado_inicial = 0, reservando os negativos para os H. (se q < 0 então …)

Existem 2 formas de implementar a função de transição :

- Com um "array" bidimensional : delta [q, c]
- Como uma função : delta (q, c)
- Ou utilizar soluções intermédias

Na implementação da função de transição com um "array", a execução é mais rápida, mas ocupa mais memória.

Por exemplo, para o autómato seguinte :



a tabela de transição é a seguinte :

q \ c	…	a	…	n	…	z	…
. . .							
2	4	4	4	3	4	4	
. . .							

Possibilidade de utilização de "arrays" muito menores, por exemplo, agrupando conjuntos de caracteres.

Na implementação da função de transição que utiliza uma função, a execução é mais lenta, mas requer menos memória, pois não precisa guardar a informação que está na tabela.

```
função delta ( q : inteiro ; c : carácter ) : inteiro ;
   caso q seja
      1 : …
      2 : caso c seja
            'n' : q ← 3
             …
      …
```

3. Tratamento de erros

Quando o texto dado não cumpre as regras previstas, deve-se fazer o seguinte :

- Assinalar o erro — protocolo com o utilizador
- Tentar prosseguir — correcção/recuperação dos erros.

Categoria dos erros :

- Erros de programação estáticos
  - infracção das regras da linguagem
  - impedem o reconhecimento do texto-fonte
  - devem ser detectados e tratados pelo compilador durante a fase de análise
- Erros de programação dinâmicos
  - erros de execução (x/0, log(0), domínios de índices, …)
  - devem ser detectados e tratados pela máquina
  - o compilador deve prever (fase de geração de código)
- Erros lógicos ou do algoritmo
  - nada a fazer, pois o problema está no programador — estudar verificação formal.

Protocolo com o utilizador — a mensagem de erro deverá (pelo menos) indicar :

- Localização do erro (linha/coluna)
- Causa do erro
- Gravidade do erro.

Classes de erros (fase onde são detectados) :

- Erros léxicos
  - caracteres não previstos, sequências inválidas, EOF durante o reconhecimento de um símbolo, EOF quando se pede um símbolo, …
- Erros sintácticos
  - sequências inválidas de símbolos : omissão, inserção, substituição, troca, …
- Erros semânticos
  - identificadores não declarados ou redeclarados, utilização for a do seu domínio, incompatibilidade de tipos, …

Tipos de erros (a sua gravidade) :

- Erro fatal — impossível continuar a análise
- Erro grave — continua a análise, mas é impossível gerar código
- Aviso — a análise e geração continuam, mas foi feita uma "correcção".

Princípios gerais do tratamento de erros :

- Poupar esforço ao programador
  - enviar mensagens claras e completas
  - tentar "corrigir o erro", ou pelo menos tentar isolá-lo "recuperando" o resto do texto
- Detectar o erro, o mais cedo possível
  - não propagar o erro à fase seguinte
- Assegurar uma perda mínima de texto
  - confiança máxima na "correcção" efectuada
- Evitar "mensagens em cascata"
  - erros assinalados em partes correctas do programa
- Não degradar a eficiência
  - tempo gasto na análise de texto correcto
- Procurar soluções genéricas
  - soluções que possam ser descritas formalmente de modo a serem geradas automaticamente.

Uma abordagem modular do tratamento de erros :

- Centralizar o tratamento de erros num só bloco
- Independente da fase onde é detectado
- Tentar garantir uniformidade nas mensagens.

```
mensagem_erro (tipo, classe, código, símbolo, posição)
    caso tipo seja
        1 : escrever ('ERRO FATAL ');
        2 : escrever ('ERRO GRAVE ');
        3 : escrever ('AVISO ');
    caso classe seja
        1 : escrever ('NA ANÁLISE LÉXICA ');
        2 : escrever ('NA ANÁLISE SINTÁCTICA ');
        3 : escrever ('NA ANÁLISE SEMÂNTICA ');
    escrever ('NO SÍMBOLO ', símbolo, 'NA POSIÇÃO ', posição.linha, posição.coluna);
    caso código seja
        1 : …
        2 : …
        …
    fim
```

Tratamento de erros léxicos :

```
E = { estados de erro }
    q ← estado_inicial
    Repetir
       q ← delta (q, c)
       Se (q ∉ H) e (q ∉ E)
       Então
          guardar (c)
          ler (c)
       Se (q ∈ E)
       Então
          mensagem_erro ( … )
          ler (c)
    Até (q ∈ H)
```

Implementação de E : estabelecer um inteiro BASE (> q_{max}) e definir o estado de erro de q_i como (q_i + BASE).

```
    q ← 0
    Repetir
       q ← delta (q, c)
       Se (q > 0) e (q < BASE)
       Então
          guardar (c)
          ler (c)
       Se (q > BASE)
       Então
          mensagem_erro ( … )
          ler (c)
          q ← q – BASE          { recuperar o erro – volta ao estado antes do erro }
    Até (q < 0)
```

Casos típicos de erros léxicos :

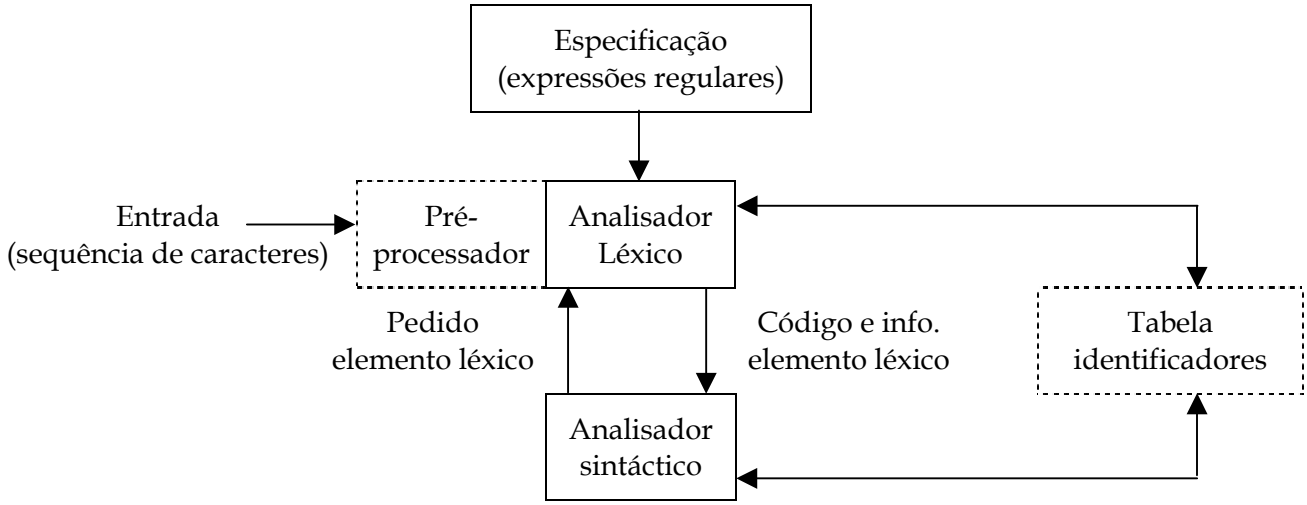
- Carácter inválido ($\notin \Sigma$)
  - avançar até encontrar um carácter conhecido
- Identificador demasiado longo
  - desprezar os caracteres que excedem o valor máximo
- Carácter não aceitável no contexto do símbolo
  - desprezá-lo e tentar prosseguir até ao próximo terminador
  - { terminadores } = { separadores } $\cup$ { inicio dos sinais }
- Aparecimento de EOF num estado intermédio
  - considerar EOF como terminador. AVISO !
- Aparecimento de EOF no estado inicial
  - erro provocado por outra fase da compilação. ERRO FATAL !

Algoritmo de um analisador léxico com tratamento de erros :

```
q ← estado_inicial
Repetir
   Se (não EOF)
   Então  car ← ficheiro↑
   Senão  Se (q em estados_intermédios)
          Então  car ← SEPARADOR
          Senão  mensagem_erro ( … )
                 ABORTA
   q ← delta (q, car)
   Se (não q em estados_intermédios)
   Então  Se (q em estados_de_erro)
          Então  mensagem_erro
          Senão  mensagem_erro ( … )
                 q ← q mod BASE        { q ← q – BASE }
                 Avança (ficheiro)
   Senão  Se (q em estados_intermédios)
          Então  Se (Comp < CompMax)
                 Então  Comp ← Comp + 1
                        Simb.nome[Comp] ← car
                 Senão  mensagem_erro ( … )
          Avança (ficheiro)
Até (q em estados_finais)
```

4. Interligação entre os analisadores léxico e sintáctico

O esquema de interacção entre o analisador léxico e o analisador sintáctico pode ser visualizado na figura seguinte.



O analisador léxico funciona com sub-rotina do analisador sintáctico e comunica com este através de um interface simples.

O analisador sintáctico chama o analisador léxico sempre que necessita de obter um elemento léxico. O analisador léxico devolve o código do elemento léxico e informação extra. Por exemplo, se o analisador léxico detectar uma sequência de caracteres descrita pela expressão regular [0–9]+, devolve ao analisador sintáctico o código do número inteiro. Uma vez que o valor do número inteiro pode ser importante, por exemplo, se for o literal de uma expressão aritmética, o analisador léxico também deve devolver ao analisador sintáctico o valor do número.

Os identificadores de elementos sintácticos como, por exemplo variáveis, devem ser armazenados na tabela de identificadores. A inserção pode ser feita pelo analisador léxico ou pelo analisador sintáctico, conforme a linguagem processada e opção do compilador.

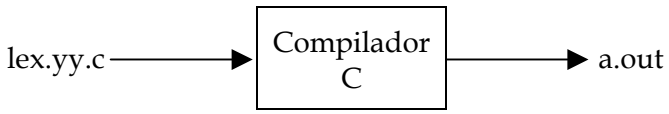
Nas linguagens de programação imperativa (Pascal e C), os identificadores podem referir elementos distintos, tais como variáveis e funções. Neste caso, é preferível o analisador léxico passar ao analisador sintáctico a referência à zona de memória para onde é copiado o identificador. Será o analisador sintáctico quem determina o significado do identificador e efectua as acções apropriadas de inserção/pesquisa do símbolo na tabela de identificadores.

5. LEX : um gerador automático de analisadores léxicos

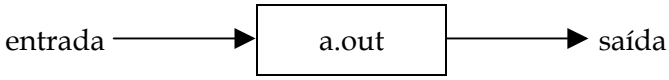
Dado um programa escrito na linguagem de programação denominada por LEX (prog.lex), as três fases da compilação deste programa são as seguintes :

1. lex prog.lex { contém uma função em C chamada yylex() }

2. cc lex.yy.c -ll { compilar, usando a biblioteca do LEX }



3. a.out < entrada > saída



O LEX gera um autómato determinístico finito, capaz de reconhecer padrões (de acordo com as expressões regulares especificadas), permitindo ainda associar acções.

Forma geral de um programa em LEX :

```
[ definições ]
%%                                          { begin }
( expressão_regular     [ acção ] ) *       { regras }
[%%
   funções  C  auxiliares ]
```

Especificação de expressões regulares em LEX — cada carácter tem o seu próprio valor, excepto os "operadores caracteres" : " \ [ ] ^– ? . * + | ( ) $ / { } % < >

- * zero ou mais ocorrências de …
- \+ uma ou mais …
- ? zero ou uma ocorrência de …
- | alternativa
- ( ) quando necessário …
- [ ] classe da caracteres; um só de entre …
- – sub-domínio

  Exemplo : Identificador :

  [A–Za–z] [a–zA–Z0–9]*

  { 1 letra }+{ sequência de 0 ou mais letras e/ou dígitos }

- { } número de ocorrências; utilização de definições

  Exemplo : de 4 a 6 ocorrências do carácter x

  x { 4, 6 }

  uma ou mais ocorrências de *digito* (definição)

  { digito }+

- "* " o próprio carácter *
- * o próprio carácter *
- . um carácter qualquer, excepto *newline*

Exemplos de definições :

branco	[\t\n]
brancos	{ branco }+
letra	[A–Za–z]
digito	[0–9]
identificador	{ letra } ({ letra } | { digito })*
numero	{ digito }+ (\. { digito }+)? (E [+\–]? { digito }+)?

Os caracteres-operadores podem valer por si próprios, quando isso não cause ambiguidade. Ex.:

```
[+ –]
[– + 0–9]
```

^ complementar

Exemplo : todos os caracteres ASCII, excepto dígitos :

```
[^0–9]
```

Condições de contexto :

^ no início de uma linha

$ imediatamente antes de *newline*

Exemplo :

```
^ola
ola$        ou        ola/\n
```

/ contexto direito

Exemplo : a só quando imediatamente seguido de b

```
a/b
```

<> definição de condições de partida, para pesquisa em contexto esquerdo

Definições e mais definições :

```
[ %  { definições externas
  %  } ]
[ definições internas ]
 %%
[ regras ]
[ %%
      funções em C ]
```

Exemplos de definições externas :

```
% {
#define    max          100
#define    eoln         while (getchar() != '\n')
#define    escrever(x)  printf("%d", x)
#include   "ficheiro"
% }
```

que funcionam de modo habitual em C.

Exemplos de definições internas ao LEX (antes de %%) :

```
dig         [0–9]
expo        [DEde] [– +]? { dig }+
%%
{ dig }+        printf("Um número inteiro ") ;
{ dig }+ "." { dig }* ( { expo } )?        |
{ dig }*  "." { dig }+ ( {expo } )?        |
{ dig }+ { dig }            printf("Um número real ") ;
```

para reconhecer os diferentes tipos de números em Fortran : inteiros, float e double.

Quando duas ou mais expressões regulares estão associadas à mesma acção, elas devem estar separadas por |.

Se a acção for formada por mais do que uma instrução, colocar entre chavetas : { }

Sensibilidade ao contexto esquerdo :

Problema : Copiar um texto integralmente, excepto a palavra magica que será substituída por :

primeira	se ocorre numa linha começada por a
segunda	se ocorre numa linha começada por b
terceira	se ocorre numa linha começada por c

```
int    flag ;
%%
^a     { flag = 'a' ;  ECHO ; }
^b     { flag = 'b' ; ECHO ; }
^c     { flag = 'c' ; ECHO ; }
\n     { flag = 0 ;   ECHO ; }
magica  {
            switch  (flag)
            {
                  case  'a' : printf ("primeira"); break;
                  case  'b' : printf ("segunda"); break;
                  case  'c' : printf ("terceira"); break;
                  default : ECHO ; break ;
            }
```

As condições de partida são definidas por :

%Start uma duas (ou %S ou %s)

e referidas por **<uma>**.

Exemplo :

```
% START   uma  duas  tres
%%
^a     { ECHO ; BEGIN  uma ; }
^b     { ECHO ; BEGIN  duas ; }
^c     { ECHO ; BEGIN  tres ; }
\n     { ECHO ; BEGIN  0 ; }
<uma>magica     printf ("primeira");
<duas>magica    printf ("segunda");
<tres>magica    printf ("terceira");
```

O LEX como analisador léxico :

yylex() é uma função que fornece o valor zero no fim do ficheiro, e pode fornecer os códigos dos símbolos reconhecidos.

Exemplo :

```
letra          [a–zA–Z]
dig            [0-9]
%%
BEGIN          { return (1) ; }
END            { return (2) ; }
:=             { return (3) ; }
{ letra }   ( { letra } | { dig } ) *          { return (4) ; }
.
.
.
```

Para escrever todos os valores dos códigos dos símbolos que ocorrem nem texto :

```
While  (c = yylex () )
      printf ("%d", c) ;
```

yytext é um "array" externo onde ficam armazenados os caracteres reconhecidos.

Para escrever a palavra lida :

```
[a–z] +        printf ("%s", yytext) ;
```

ou, abreviadamente

```
[a–z] +        ECHO ;
```

yyleng dá o comprimento do padrão reconhecido

```
[a–zA–Z] +      { palavras ++ ;  letras += yyleng ; }
```

Exemplo :

Somar 3 a todo o inteiro não negativo que seja divisível por 7.

```
%%
        int     k ;
[0-9] +         {       k = atoi (yytext) ;
                        if  (k % 7 == 0)
                                printf (“%d”, k+3) ;
                        else
                                printf (“%d”, k) ;
                }
```