Compiladores

Capítulo 2 - Análise Léxica

2.1 - Introdução

A base teórica para a construção de analisadores léxicos (*scanners*) é a teoria de linguagens regulares e autômatos finitos. Entretanto, não é necessário um conhecimento a fundo desse assunto para a construção manual de analisadores léxicos. Esse conhecimento é mais útil para a construção de geradores de analisadores léxicos, e em menor escala, para o uso desses geradores.

Este capítulo apresenta apenas uma breve exposição sobre a construção de analisadores léxicos, tomando como exemplo um analisador léxico construído para um subconjunto de Pascal. Este exemplo pode servir de base para a construção manual de um analisador léxico para linguagens cuja parte léxica é semelhante à da linguagem apresentada aqui. Entretanto, o leitor fica avisado de que para outras linguagens o analisador léxico pode ser significativamente mais complicado, e que nesse caso será necessário recorrer à literatura específica sobre o assunto. O exemplo de linguagem aqui utilizado recebeu o nome de *Pascal0*.

2.2 - Especificação do analisador sintático: a estrutura léxica da linguagem.

Para poder especificar um analisador léxico precisamos examinar os componentes léxicos (tokens) da linguagem. No caso de *Pascal0*, esses componentes são:

Palavras reservadas. As palavras reservadas de *Pascal0* são as seguintes:

```
AND ARRAY BEGIN CONST DIV DO ELSE END FOR FUNCTION GOTO
IF LABEL MOD NOT OF OR PROCEDURE PROGRAM RECORD REPEAT
THEN TO TYPE UNTIL VAR WHILE
```

Não se faz distinção entre letras maiúsculas e minúsculas em palavras reservadas.

Identificadores. Os identificadores de *Pascal0* são cadeias de caracteres contendo letras ('A', ..., 'Z', 'a', ..., 'z'), dígitos ('0', ... '9'), e o caracter sublinhado ('_'), devendo o primeiro caracter ser sempre uma letra. Não podem ser usadas como identificadores, entretanto, as cadeias correspondentes às palavras reservadas. Não se faz distinção entre letras maiúsculas e minúsculas em identificadores.

- A análise léxica/sintática de uma linguagem que tem palavras reservadas tende a ser mais simples que a de linguagens que tem apenas palavras-chave (*keywords*), que também podem ser usadas como identificadores. Por exemplo, se `PROCEDURE` pode ser um identificador ou uma palavra chave, não é possível decidir qual a classificação correta do token, sem um exame do contexto em que a cadeia foi encontrada. Por exemplo, em

```
PROCEDURE PROCEDURE(X:INTEGER); BEGIN ... END;
```

a primeira ocorrência seria da palavra chave, e a segunda seria de um identificador.

- Uma regra (raramente explicitada nos manuais das linguagens) diz que um token se estende até que seja encontrado um caracter que não faz parte dele. Essa regra,

(ou alguma regra semelhante) é necessária para que o analisador léxico possa reconhecer em

```
XYZ123A+1
```

uma ocorrência de um identificador `XYZ123A`, e não, por exemplo, ocorrências de três identificadores `XY`, `Z123` e `A`. Usando a regra mencionada, é a presença do caracter '+' que determina o fim do identificador. Essa regra nos obriga a separar palavras reservadas e identificadores por brancos ou outros caracteres. Por exemplo, em

```
IF XYZ=ABC+DEF THEN XYZ:=ABC DIV DEF;
```

os espaços não podem ser retirados sem alteração do significado.

• Pascal não faz distinção entre maiúsculas e minúsculas em identificadores e palavras reservadas, mas algumas linguagens (como C) tratam dois identificadores como `TabSimb` e `tabsimb` como distintos. A finalidade disso é permitir que identificadores relacionados possam ter formas semelhantes. (No mesmo caso, em Pascal, poderíamos usar `TabSimb` (equivalente a `tabsimb`) e `tab_simb`). Do ponto de vista da implementação, basta tomar cuidado de sempre converter todas as letras para maiúsculas (ou para minúsculas, se preferido) antes de qualquer comparação entre identificadores.

Delimitadores e Operadores. As seguintes cadeias de caracteres são usadas como delimitadores (pontuação e organização do texto do programa) ou operadores (representando operações matemáticas comuns):

```
.     ;     =     [     ]     :     ..    ^     (     )
:=    ,     <     >     <>    >=    <=    +     -     *
```

Números inteiros. Números inteiros sem sinal são representados por uma cadeia de digitos.

Literais. Literais (*strings*) são cadeias de símbolos delimitadas por plicas (`'`). Se a cadeia deve incluir uma plica, ela é dobrada, como, por exemplo, em `'Bob''s'`.

Os comentários ficam entre chaves (`{`, `}`), e podem conter qualquer caracter exceto, naturalmente, fecha-chave (`}`).

2.3 - Interface e organização geral do analisador léxico

Este analisador léxico se comunica com o restante do programa através de uma interface constituída pelo tipo `tokens`, pelo procedimento `scan`, e por algumas variáveis.

O tipo `tokens` é um tipo de enumeração, e tem um valor para cada token da linguagem. Sua declaração é

```
type tokens=
    (t_eof,
     t_and, t_array, t_begin, t_const, t_div,
     t_do, t_else, t_end, t_for, t_function,
     t_goto, t_if, t_label, t_mod, t_not, t_of,
     t_or, t_procedure, t_program, t_record,
     t_repeat, t_then, t_to, t_type, t_until,
     t_var, t_while, t_pt, t_ptvg, t_eq,
     t_abrecol, t_fechacol, t_2pt, t_ptpt,
     t_pont, t_abrepar, t_fechapar, t_2pteq,
```

```
t_vg, t_lt, t_gt, t_ne, t_ge, t_le,
t_mais, t_menos, t_vezes,
t_id, t_int, t_lit,
t_erro);
```

Assim, `t_eof` é o token que indica o fim de arquivo, correspondendo ao símbolo `$`, usado como marcador de fim de cadeia em alguns métodos de análise sintática. Os tokens `t_and`, ... `t_while` correspondem às palavras reservadas `and`, ..., `while`. Os tokens identificador, inteiro e literal são representados por `t_id`, `t_int` e `t_lit`. Os demais tokens correspondem a delimitadores e operadores. O token `t_erro` é usado para situações não previstas.

Os tokens `t_id`, `t_int` e `t_lit` são chamados *tokens variáveis*, pelo fato de que se distinguem pelos seus nomes, ou seja, pelas cadeias associadas a eles, enquanto os demais tokens (tokens constantes) tem sempre um único valor. Em alguns raros casos, um token constante pode corresponder a mais de uma cadeia, mas não há nenhum significado especial atribuído a cada uma das variantes. Por exemplo, algumas implementações de Pascal aceitam `(.` e `.)` como equivalentes de `[` e `]`.

Em linguagens que não permitem a definição de tipos de enumeração, os elementos do tipo podem ser declarados como constantes do tipo inteiro, de forma que as referências dentro do programa aos tokens possam ser feitas através de seus nomes, o que conduz a menos erros do que o uso direto dos valores numéricos.

A cada chamada do procedimento `scan` um novo token é identificado. Entretanto, dois tokens diferentes são usados pelo analisador sintático e pelo analisador semântico. Podemos ter na gramática uma regra `iden`→`id`, cujo uso será identificado e sinalizado pelo analisador sintático em função do símbolo que segue o `id`, enquanto o analisador semântico está tratando o terminal `id` (identificador). Se tivermos

```
xyz := 102 ;
```

o analisador léxico indicará sucessivamente os tokens `t_id` (com nome "`xyz`"), `t_2pteq`, `t_int` (com nome "`102`"), e `t_ptvg`. O analisador sintático só avisa que a regra `iden`→`id` foi usada quando encontra o token `t_2pteq`. Esta regra, entretanto, é a regra que trata o identificador `xyz`. Esse tratamento pode ser, por exemplo, a consulta a uma tabela de símbolos para verificar qual o tipo com que o identificador foi declarado, para verificação da correção desse uso do identificador. Por essa razão o analisador léxico sempre mantém dois tokens disponíveis, um (o último) para a análise sintática (no caso, `t_2pteq`), e outro (o penúltimo) para a análise semântica (no caso `t_id`, com nome "`xyz`"). A situação seria semelhante para o tratamento do inteiro `102`: o analisador semântico estaria tratando o token `t_int` (com nome "`102`"), de acordo com uma regra sinalizada pelo analisador sintático após a chegada do token `t_ptvg`.

Os códigos e os nomes do penúltimo e do último token estão nas variáveis `tok`, `simb`, `nometok`, `nomesimb`. Normalmente, `nometok` e `simb` não são utilizados.

Além dessas, a interface do analisador léxico usa variáveis para identificar o arquivo fonte, a última linha lida do fonte, a posição (linha/coluna) do último caracter lido, etc. Os valores dessas variáveis podem ser usados em mensagens de erro.

2.4 - Alguns aspectos da implementação

Um ponto importante da implementação é a definição de classes de caracteres. Por exemplo, uma das classes (`c_letra`) corresponde às letras. Isso acontece porque o tratamento dado a todas as letras é semelhante, e é mais fácil testar se a classe do caracter é `c_letra` do que testar se o caracter é um dos caracteres 'A', ..., 'z'. As classes usadas são as seguintes:

classe	caracteres	tokens
c_eof	^Z	t_eof
c_ponto	. ..	t_pt, t_ptpt
c_opdel	; = [ ] ( ) , + - * ^	t_ptvg, t_eq, t_abrecol, t_fechacol, t_abrepar, t_fechapar, t_vg, t_mais, t_menos, t_vezes, t_pont
c_2pt	: :=	t_2pt, t_2pteq
c_menor	<	t_ne, t_lt, t_le
c_maior	>	t_gt, t_ge
c_letra	a ... z , A ... Z	t_id, t_and, ..., t_while
c_digito	0 ... 9	t_int
c_subl	_	
c_plica	'	t_lit
c_outros	....	t_erro

A terceira coluna mostra os tokens iniciados por caracteres de cada classe.

Cada vez que um caracter é processado, um novo caracter

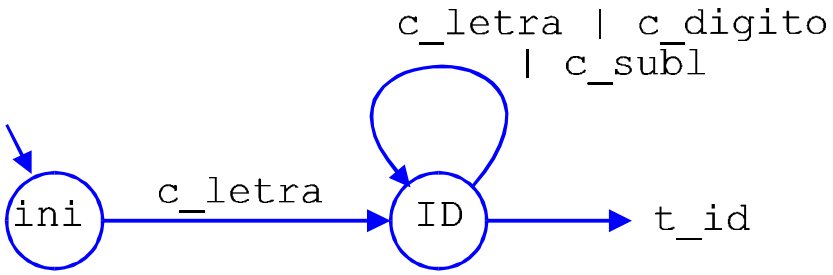

Fig.1- Reconhecimento de identificadores

Podemos ver na Fig. 4 os problemas associados com o reconhecimento direto de palavras reservadas. Primeiro, o esquema de divisão dos caracteres em classes é complicado pelo fato de que cada letra inicial de uma palavra reservada deve ser considerada em separado; segundo, podemos observar que o número de estados adicionais do automato finito pode ser avaliado somando os comprimentos de todas as palavras reservadas. Naturalmente, palavras reservadas curtas e com prefixos comuns levam a um número menor de estados, mas não se espera que isso seja levado em consideração quando se projeta uma linguagem. Ao contrário, palavras reservadas facilmente distinguíveis aumentam a legibilidade dos programas na linguagem.

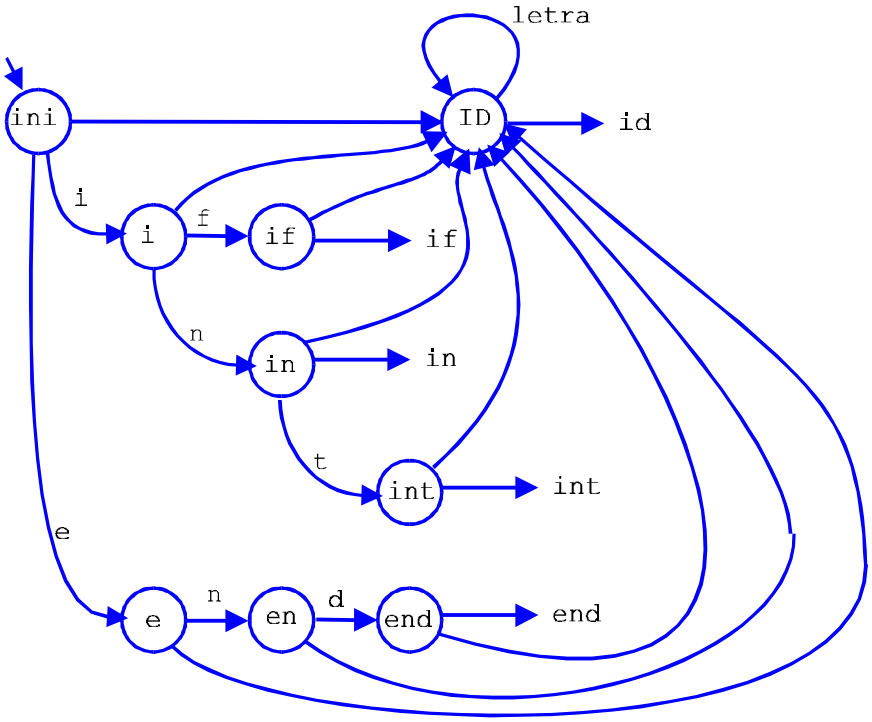

Fig. 4 - Reconhecimento das palavras reservadas `if`, `in`, `int` e `end`.

Um esquema mais simples de tratamento é o do nosso exemplo: no tratamento inicial as palavras reservadas são ignoradas. Encontrado um suposto identificador, a cadeia correspondente é então procurada (procedimento `lookup`) em uma tabela de palavras reservadas indexada pelos valores do tipo `tokens` correspondentes às palavras reservadas. Caso desejado, algoritmos e estruturas de dados mais sofisticados podem ser utilizados.

2.5 - Listagem do analisador léxico exemplo

O analisador léxico para a linguagem *Pascal0* foi construído com as características mencionadas anteriormente, e deve ser entendido apenas como um exemplo. Com algumas pequenas alterações, tem sido usado para demonstrar o funcionamento de analisadores sintáticos construídos pelo gerador R*S simples[1].

[1]José Lucas Rangel, Manual de Aplicação do Sistema de Geração de Analisadores Sintáticos R*S simples, Monografias em Ciência da Computação, no. 11/88, Dep. Informática, PUC-Rio.

```
{ ---------------------------------- } unit lex0;

   { +------------------------------------------------+
     | exemplo de analisador lexico                   |
     | Pascal Turbo 6.0                               |
     | "as is"                     J.L.Rangel, set'94 |
     +------------------------------------------------+ }

{ ---------------------------------- } interface

     uses crt; { para escrever na tela }

type tokens= ( t_eof,
   t_and,     t_array,    t_begin,     t_const,   t_div,
   t_do,      t_else,     t_end,       t_for,     t_function,
   t_goto,    t_if,       t_label,     t_mod,     t_not,
   t_of,      t_or,       t_procedure, t_program, t_record,
   t_repeat,  t_then,     t_to,        t_type,    t_until,
   t_var,     t_while,    t_pt,        t_ptvg,    t_eq,
   t_abrecol, t_fechacol, t_2pt,       t_ptpt,    t_pont,
   t_abrepar, t_fechapar, t_2pteq,     t_vg,      t_lt,
   t_gt,      t_ne,       t_ge,        t_le,      t_mais,
   t_menos,   t_vezes,    t_id,        t_int,     t_lit,
   t_erro);

var
   fonte:text;     { arquivo fonte }
   tok,            { codigo do ultimo token encontrado }
   simb:tokens;    { codigo do penultimo token encontrado }
   nometok,        { nome do ultimo token encontrado }
   nomesimb,       { nome do penultimo token encontrado,
                     para a semantica }
   linha:string;   { ultima linha lida do fonte }
   poslinha:byte;  { posicao do proximo caracter a ser lido
                     em linha }
   numlinha:integer; { numero de linha no arquivo fonte }

procedure scan;      { analisador léxico }

{ ---------------------------------- } implementation
procedure idfonte;  { identifica o arquivo fonte }
   var nome:string;
begin {idfonte}
   clrscr;
   {$I-}     { erro de entrada/saida não interrompe execução }
      repeat
        write('fonte > '^g);
        readln(nome);
        if pos('.',nome)=0 then
          nome:=nome+'.0';
        writeln('fonte : ',nome);
        assign(fonte,nome);
        reset(fonte);
      until IOresult=0;  { IOResult <> 0 indica erro,
                           no caso a inexistencia do arquivo fonte }
   {$I+}
end; {idfonte}
```

```
type classes = (
   c_eof,    { ^Z }
   c_ponto,  { . .. }
   c_opdel,  { ; = [ ] ( ) , + - * ^ }
   c_2pt,    { : := }
   c_menor,  { <> < <= }
   c_maior,  { > >= }
   c_letra,  { a ... z , A ... Z }
   c_digito, { 0 ... 9 }
   c_subl,   { _ }
   c_plica,  { ' }
   c_outros );

var
   tabclasses:array[char] of classes;
   tabopdel:array['('..'^'] of tokens;

procedure inittabs;
   { inicia tabclasses e tabopdel }
   var c:char;
begin
   for c:=#0 to #255 do
      tabclasses[c]:=c_outros;
   tabclasses[^Z]:=c_eof;      tabclasses['.']:=c_ponto;
   tabclasses[';']:=c_opdel;   tabclasses['=']:=c_opdel;
   tabclasses['[']:=c_opdel;   tabclasses[']']:=c_opdel;
   tabclasses['(']:=c_opdel;   tabclasses[')']:=c_opdel;
   tabclasses[',']:=c_opdel;   tabclasses['+']:=c_opdel;
   tabclasses['-']:=c_opdel;   tabclasses['*']:=c_opdel;
   tabclasses['^']:=c_opdel;   tabclasses[':']:=c_2pt;
   tabclasses['<']:=c_menor;   tabclasses['>']:=c_maior;
   for c:='A' to 'Z' do
      tabclasses[c]:=c_letra;
   for c:='a' to 'z' do
      tabclasses[c]:=c_letra;
   for c:='0' to '9' do
      tabclasses[c]:=c_digito;
   tabclasses['_']:=c_subl;    tabclasses['''']:=c_plica;
   for c:='(' to '^' do
      tabopdel[c]:=t_erro;
   tabopdel[';']:=t_ptvg;      tabopdel['=']:=t_eq;
   tabopdel['[']:=t_abrecol;   tabopdel[']']:=t_fechacol;
   tabopdel['(']:=t_abrepar;   tabopdel[')']:=t_fechapar;
   tabopdel[',']:=t_vg;        tabopdel['*']:=t_vezes;
   tabopdel['+']:=t_mais;      tabopdel['-']:=t_menos;
   tabopdel['^']:=t_pont;
end; { inittabs }

procedure erroLex(i:byte);
                    { tratamento radical de erros lexicos }
   const msg:array[1..3] of string[25] = (
      'comentario nao fechado',
      'caracter inesperado',
      'string sem plica final');
begin
   writeln('---> erroLex <---':35);
   writeln(linha);
   writeln('Erro Lexico: ',msg[i]);
   writeln('Linha:',numlinha,' /',poslinha);
```

```
   write('>');
   readln;
   halt;                        { encerra execucao }
end; {erroLex}

var
   ch:char;
   cl:classes;
procedure add;
begin
   nometok:=nometok + ch;
end; {add}

procedure getchar;
begin
   if poslinha>length(linha) then
      if eof(fonte) then
         ch:=^Z                    { fim do fonte }
      else begin                   { outra linha }
         readln(fonte,linha);
         linha[succ(length(linha))]:=#0;
         inc(numlinha);
       { write(^m,numlinha); contador indicador de progresso }
         writeln('[',numlinha,']',linha);
         poslinha:=1;
         ch:=^M;
      end else begin               { mesma linha }
         ch:=linha[poslinha];
         inc(poslinha);
      end;
   cl:=tabclasses[ch];
end; {getchar}

procedure getnonblank;            { pula brancos,comentarios }
begin
   while cl=c_outros do begin
      if ch='{' then begin           { inicio de comentario }
         getchar;
         repeat
            getchar
         until (ch='}') or (ch=^Z);
         if ch=^Z then
            erroLex(1);
      end else if (ch<>' ')            { branco }
                  and (ch<>^M)         { mudanca de linha }
                  and (ch<>^I) then    { tab }
         erroLex(2);
      getchar;
   end;
end; {getnonblank}

procedure lookup;
   const tab: array[t_and..t_while] of string[9]= (
      'AND', 'ARRAY', 'BEGIN', 'CONST', 'DIV', 'DO', 'ELSE',
      'END', 'FOR', 'FUNCTION', 'GOTO', 'IF', 'LABEL', 'MOD',
      'NOT', 'OF', 'OR', 'PROCEDURE', 'PROGRAM', 'RECORD',
      'REPEAT', 'THEN', 'TO', 'TYPE', 'UNTIL', 'VAR',
      'WHILE' );
   var t1:tokens; c1:char;
```

```
   { UpCase e UpCaseStr convertem simbolos e cadeias
     para maiusculas }
   { supõe semântica de curto-circuito:
     ---avaliação incompleta de expressões booleanas ---
     só testa a cadeia toda se o primeiro caracter confere }
begin { lookup }
   tok:=t_ID;
   c1:=nometok[1];
   for t1 := t_and to t_while do
      if tab[t1,1]>c1 then
         exit
      else if (tab[t1,1]=UpCase(c1))
              and (tab[t1]=UpCaseStr(nometok)) then begin
         tok:=t1;
         exit;
      end;
end; {lookup}
procedure scan;
begin
   nomesimb:=nometok;
   getnonblank;
   nometok:='';
   case cl of
      c_letra:
         begin
            repeat
               add;
               getchar;
            until (cl<>c_letra) and (cl<>c_digito)
                                and (cl<>c_subl);
            lookup;
         end;
      c_digito:
         begin
            repeat
               add;
               getchar;
            until cl<>c_digito;
            tok:=t_Int;
         end;
      c_ponto:
         begin { . .. }
            getchar;
            if ch='.' then begin
               getchar;
               tok:=t_ptpt;
            end else
               tok:=t_pt;
            end;
      c_2pt:
         begin { : := }
            getchar;
            if ch='=' then begin
               getchar;
               tok:=t_2pteq;
            end else
               tok:=t_2pt;
         end;
```

```
      c_maior:
         begin { > >= }
            getchar;
            if ch='=' then begin
               tok:=t_ge;
               getchar;
            end else
               tok:=t_gt;
         end;
      c_menor:
         begin { <> < <= }
            getchar;
            if ch='=' then begin
               tok:=t_le;
               getchar;
            end else if ch='>' then begin
               tok:=t_ne;
               getchar;
            end else
               tok:=t_lt;
         end;
      c_plica:
         begin
            getchar;                { a plica não é incluída }
            fim:= false;
            repeat
               if ch= '''' then begin
                  getchar;
                  if ch= '''' then begin
                                       { inclui segunda plica }
                     add;
                     getchar;
                  end else
                     fim:=true
               end else if ch=^M then
                  erroLex(3);      { literal sem plica final }
               else
                  add
            until fim;
            tok:=t_lit;
         end;
      c_eof: { $ }
         tok:=t_eof;
      c_opdel:
         begin
            tok:=tabopdel[ch];
            getchar;
         end;
   end; {case}
end {scan};
```

```
{ ---------------------------------- } begin { lex0 }
   { iniciação das tabelas e variáveis;
     identificação do arquivo fonte }
inittabs;  { inicia tabclasses e tabopdel }
numlinha:=0;
linha:='';
poslinha:=1;
ch:=' ';
cl:=c_outros;
idfonte; { identifica o arquivo fonte }
{ ---------------------------------- } end. { lex0 }
```

(rev. mar 99)