Análise Léxica

É a primeira fase de um compilador e juntamente com o módulo de leitura do programa são os únicos componentes de um compilador que chegam a ver o texto de programa inteiro.

Tarefa principal: ler os caracteres de entrada, organizá-los em unidades lógicas (tokens), e produzir uma seqüência de tokens que será utilizada pelo Parser na análise sintática.

Lexema – Corresponde a cadeia de caracteres representada por um token.

Qualquer valor associado a um token recebe o nome de atributo.

Outras tarefas do analisador léxico (scanner):

1. Remoção de espaços em branco e comentários;
2. Constantes – para uma constante pode-se criar um token.
   Ex.: Seja **num** a classe do token que representa um inteiro.
   31 + 28 + 59
   <**num**, 31> <+,> <**num**, 28> <+,> <**num**, 59>
3. Reconhecer identificadores e palavras-chave.
   Ex.: contador = contador + incremento;
   É convertida para <**id,1**> = <**id,1**> + <**id,2**>;

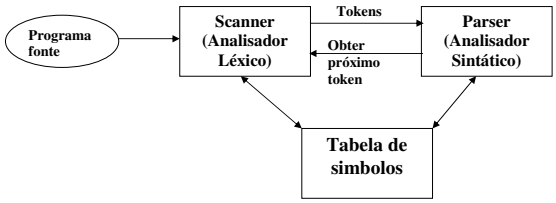
O lexema dos identificadores são armazenados na tabela de símbolos e um apontador para a entrada da tabela se torna um atributo do token.

É necessário algum mecanismo para distinguir as palavras-chave de uma linguagem dos identificadores, visto que os dois obedecem ao mesmo padrão.

Categorias de tokens:

- Palavras reservadas: IF - "if"
  ELSE – "else"
- Símbolos especiais: ADDOP - +, -
  MULTOP - *, /
- Cadeias múltiplas de caracteres: NUM e ID

Interface para um Analisador Léxico



Incorporando uma tabela de símbolo

Tabela de símbolos é uma estrutura de dados que armazena informações sobre várias construções da linguagem-fonte.

	token	**lexema**	Atributos
0			
1	**div**	div	
2	**mod**	mod	
3	**id**	contador	
4	**id**	i	

Interface com a tabela de símbolo

Inserir(s, t) – insere o token t para a cadeia s e retorna o índice da nova entrada.

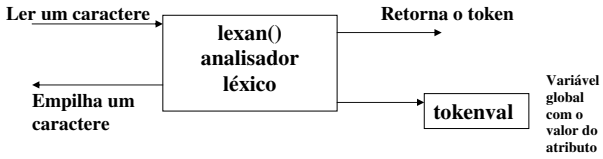
Buscar(s) – retorna o índice de uma entrada para a cadeia s ou 0 se s não for encontrado.

Palavras-chave reservadas

Podemos inicializar a tabela de símbolos com as palavras-chave reservadas.

Ex.: Inserir("div", **div**);
Inserir("mod", **mod**);

Um analisador léxico para expressões




```
Função lexan:inteiro;
Inicio
    Repetir inicio
        Ler um caractere em c;
        Se c for um espaço em branco ou um caractere de tabulação
            então
            não faz nada
        Senão se c for um caractere de avanço de linha então
            Incrementa o contador de linha
        Senão se c é um dígito então inicio
            Fazer tokenval igual ao volor deste dígito e dos dígitos
                seguintes
            Retornar num
        Senão se c é uma letra então início
            Coloca c e as letras e dígitos seguintes em lexema;
            P := buscar(lexema);
            Se p = 0 então
                P := inserir(lexema, id);
            Tokenval := p;
            Retorna o campo token da entrada p da tabela de
                símbolos;
        Fim
        Senão início
            Fazer tokenval igual a nulo;
            Retornar a codificação inteira do caractere c;
        Fim
    Fim
fim
```

Tokens, padrões e lexemas

Token – símbolos terminais na gramática para a linguagem-fonte: palavras-chave, operadores, identificadores, constantes, literal, símbolos de pontuação,...
Padrão – é uma regra que descreve o conjunto de cadeias de caracteres que pode representar um token.
Lexema – é a cadeia de caracteres de entrada de um determinado token.

Token	lexema	Padrão
const	const	Const
if	if	If
op_relacional	<, <=, =, <>, >, >=	< ou <= ou = ou <> ou > ou >=
id	Pi, contador, d2	Letra seguida por letras/dígitos
num	3.1416, 0, 6.02E23	Qualquer constante numérica
literal	"conteúdo da memória"	Quaisquer caracteres entre aspas

Atributos para os tokens

Do ponto de vista prático o token possui normalmente um único atributo.
Exemplo:

E = M * C ** 2

Especificação dos tokens

Cada padrão corresponde a um conjunto de cadeias e dessa forma, as **expressões regulares** servirão como nomes para os conjuntos de cadeias.

1. ε é uma expressão regular que denota {ε};
2. Se a é um símbolo em Σ, então a é uma expressão regular que denota {a};
3. Sejam *r* e *s* expressões regulares denotando as linguagens *L(r)* e *L(s)*. Dessa forma,
   a) (r) | (s) é uma expressão regular denotando a linguagem L(r) U L(s).
   b) (r) (s) é uma expressão regular denotando a linguagem L(r) L(s).
   c) (r)* é uma expressão regular denotando a linguagem (L(r))*.
   d) (r) é uma expressão regular denotando a linguagem L(r).

Os parênteses desnecessários podem ser evitados com a convenção:

1. o operador unário * tem a maior precedência;
2. a concatenação tem a segunda maior precedência;
3. | tem a menor precedência.

Ex.: *(a) | ((b)*(c))* é equivalente a *a | b*c*

Ex.: Seja Σ = {a, b}

1. A expressão regular a | b denota
2. A expressão regular (a | b)(a | b) denota
3. A expressão regular a* denota
4. A expressão regular (a | b)* denota
5. A expressão regular a | a*b

Se duas expressões regulares, r e s, denotam a mesma linguagem dizemos que r e s são equivalentes e escrevemos r=s. Ex.: (a|b) = (b|a).

Definições regulares

Se Σ é um alfabeto de símbolos básicos, então uma **definição regular** é uma seqüência de definições da forma
$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
...
$d_n \rightarrow r_n$
onde cada d_i é um nome distinto e cada r_i é uma expressão regular definida sobre os símbolos em Σ U $\{d_1, d_2, ..., d_{i-1}\}$
É uma gramática livre de contexto em EBNF, com a restrição de que nenhum não terminal pode ser usado antes de ter sido definido completamente. Deve-se definir antes de usar.

Ex.:
1. Identificadores Pascal:
   letra → A | B | ... | Z | a | b | ... | z
   digito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
   id → **letra** (**letra** | **digito**)*
2. Números sem sinal

Simplificações de Notação

1. Uma ou mais ocorrências – r^+
2. Zero ou uma ocorrência – r?
3. Classe de caracteres – [abc] , [A-Za-z], [a-z]

Resumo

Padrões básicos	
x	**O caractere x**
.	**Qualquer caractere, exceto nova linha**
[abcA-Z]	**Qualquer um dos caracteres a, b, c e no intervalo de A até Z.**
Operadores de repetição	
R?	**Opcionalmente um R**
R*	**Zero ou mais ocorrências de R**
R+	**Uma ou mais ocorrências de R**
Operadores de composição	
R_1 R_2	**Um R_1 seguido por um R_2**
R_1 | R_2	**Um R_1 ou um R_2**
Agrupamento	
(R)	**O próprio R**

Caractere de escape em expressões regulares

Pode-se usar a barra: *, \\

Pode-se usar o caractere de aspas: "*", "*?", " " "

Exercícios

Escreva definições regulares:

1) Uma seqüência de dígitos opcionalmente seguida por um caractere que denota a base (b- binário, o- octal)

2) Um número de ponto fixo é uma opcional seqüência de dígitos seguida por um ponto e depois por uma seqüência de dígitos.

3) Um identificador é uma seqüência de letras e dígitos. O primeiro caractere deve ser uma letra. O sublinha (_) conta como letra, mas não pode ser usada como o primeiro ou o último caractere.

4) É mostrado a seguir uma gramática altamente simplificada para URLs, supondo-se definições apropriadas para letra e dígito.

URL → rótulo | URL "." rótulo

rótulo → letra (letdig_hyphen_string? letdig) ?

letdig_hyphen_string → letdig_hyphen | letdig_hyphen letdig_hyphen_string

letdig_hyphen → letdig | "-"

letdig → letra | digito

Reconhecimento de tokens

Ex.: Considere o seguinte fragmento de gramática

cmd → **if** *expr* **then** *cmd*
| **if** *expr* **then** *cmd* **else** *cmd*
| ε

expr → *termo* **relop** *termo*
| *termo*

termo → **id**
| **num**

definições regulares para os símbolos terminais:

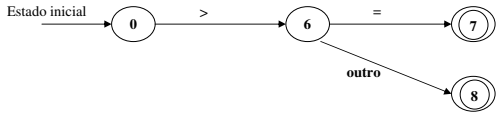
if → if
then → then
else → else
relop → < | <= | = | <> | > | >=
id → **letra** (**letra** | **digito**)*
num → **digito**$^+$ (. **digito**$^+$)? (E(+ | -)? **Digito**$^+$) ?

Meta: construir um analisador léxico que irá isolar o lexema para o próximo token no buffer de entrada e produzir como saída um par consistindo no token e no valor de um atributo. Os valores de atributos para os operadores relacionais são dados pelas constantes simbólicas LT, LE, EQ, NE, GT e GE.

Diagrama de transição (Autômatos Finitos)

Outra forma de descrever os símbolos de uma linguagem seria com o uso de autômatos de estados finitos (que têm a vantagem de permitir uma tradução quase direta para código de programação).

Obter algoritmos a partir dos autômatos é uma tarefa bastante simples.



Ex.: Diagrama de transição para o token **relop**.

Ex.: Diagrama de transição para identificadores e palavras-chave.

Ex.: Diagrama de transição para número sem sinal.

Implementando um diagrama de Transição

Cada estado recebe um trecho de código:

- Se existirem arestas deixando um estado, então seu código lê um caractere e seleciona uma aresta possível para seguir, se possível.
- A função próximo_caractere() é utilizada para ler o próximo caractere a partir do buffer de entrada.
- Se existir uma aresta rotulada pelo caractere lido, ou por uma classe de caracteres que o contenha, o controle é transferido para o código do estado apontado por aquela aresta.
- Se não existir tal aresta, o estado corrente não é um daqueles que indica que um token foi encontrado e, nesse caso, a rotina falha() é chamada para retroceder o apontador adiante para a

posição inicial e começar nova busca por um token usando o próximo diagrama de transição.

- Se não existirem outros diagramas de transição a tentar, falhar() chama uma rotina de recuperação de erros.

```
int estado=0, partida =0;
int tokenval; /* para retornar o
segundo componente de um token */
int falhar() {
  apontador_adiante = inicio_do_lexema;
  switch (partida) {
    case 0: partida = 9; break;
    case 9: partida = 12; break;
    case 12: partida = 20; break;
    case 20: partida = 25; break;
    case 25: recuperar(); break;
    default: ; /* erro do compilador */
  }
  return (partida)
}
```



```
int proximo_token() {
   while(1) {
     switch(estado) {
      case 0: c=proximo_caractere();
         if (c==' '|| c=='\t' ||
            c=='\n') {
            estado = 0;
            inicio_do_lexema++;
         }
         else if (c=='<') estado=1;
         else if (c=='=') estado=5;
         else if (c=='>') estado=6;
         else estado = falhar();
         break;
      case 1:
         .
         .
         .
     }
   }
}
```


Opções para definir a classe de um token

As classes dos tokens podem ser definidas como:

- Tipos enumerados:

```
typedef enum {
    IF, THEN, ELSE, PLUS, NUM, ID,...
} Token;
```

- Constants:

```
#define IF 256
#define THEN 257
#define ELSE 258 ...
```

Ou

```
public class Token {
    public static final int IF=256;
    public static final int THEN=257;
    public static final int ELSE=258;...
}
```

- Classes e subclasses:

```
abstract public class Token { ... }
public class IF extends Token { ... }
public class THEN extends Token { ... }
public class ELSE extends Token { ... }
...
```

Exemplo de definição do tipo token (Token_Type)

```
/* Definição de classe 0 a 255 reservadas para caracteres ASCII. */
#define  EOF         256
#define  IDENTIFIER  257
#define  INTEGER     258
#DEFINE  ERRONEOUS   259

typedef struct {
   char *file_name;
   int line_number;
   int char_number;
} Position_in_File;

typedef struct {
   int class;
   char *repr;
   Position_in_File pos;
} Token_Type;
```


Leitura do texto de programa

O uso de rotinas de leitura caractere a caractere é desaconselhável.

Alguns compiladores utilizam técnicas de *bufferização*.

É recomendado ler o arquivo inteiro com uma única chamada do sistema e a quantidade de memória alocada para o arquivo deve ser apenas a necessária.

"Bufferização" da entrada

Pares de Buffer

É um buffer dividido em duas metades, com N caracteres cada uma.

N caracteres de entrada são lidos em cada metade do buffer em apenas um comando de leitura do sistema.

Se restarem menos do que N caracteres na entrada, um caractere especial eof é gravado no buffer após todos os caracteres.

São mantidos dois apontadores:

- inicio_do_lexema
- apontador_adiante

A cadeia entre os dois apontadores é o lexema corrente.

Se o apontador adiante estiver prestes a ser deslocado para além da marca do meio, a metade à direita é preenchida com N novos caracteres de entrada.

Se o apontador estiver prestes a se deslocar para fora do buffer a primeira metade é preenchida e o apontador adiante é deslocado para o início do buffer.

Uso de sentinelas

O esquema anterior requer que sejam feitos dois testes para cada avanço do apontador adiante, exceto ao final das metades.

Pode-se reduzir o número de testes para um colocando-se sentinelas ao final das duas metades. O sentinela deve ser um caractere que não faça parte do programa-fonte.

Construção de um AFN a partir de uma expressão regular

Construção de Thompson

Entrada: Uma expressão regular r sobre um alfabeto Σ.

Saída: Um AFN N que aceita L(r)

Método: Divide-se r gramaticalmente em suas expressões constituintes.

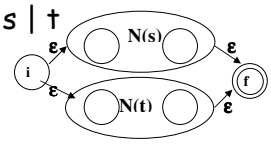
Usa-se as regras (1) e (2) para construir AFN para os símbolos básicos em r.

Depois usa-se a regra 3 para combinar esses AFN recursivamente.

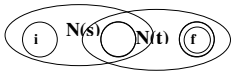
1. Para ε, construímos um estado inicial i, um estado final f e um lado rotulado por ε indo de i a f.
2. Para *a* pertencente Σ, construímos um estado inicial i, um estado final f e um lado rotulado por ε indo de i a f.

3. Sejam N(s) e N(t) AFN para as expressões regulares s e t.

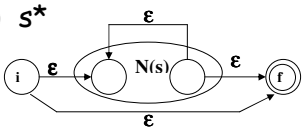
a) s | t



b) st



c) s*



Exercício: Construa AFN para:

a) (a|b)*abb
b) (a|b)*
c) ((ε|a)b*)*
d) (a|b)*abb(a|b)*
e) letra (letra|digito)

Conversão de um AFN para um AFD

Algoritmo: Construção de subconjuntos

Entrada: Um AFN N

Saída: Um AFD D aceitando a mesma linguagem

Método: Construir uma tabela de transição Dtran para D.

Usa-se as operações: fecho-ε(s) , fecho-ε(T) e movimento(T,a)

Antes de ler, o primeiro símbolo da entrada N pode estar em qualquer estado do conjunto fecho-ε(s_0). Seja T o conjunto de estados atingíveis a partir de s_0 a uma dada seqüência de símbolos de entrada e seja *a* o próximo símbolo da entrada. Ao ler *a* , N pode ir para qualquer um dos estados do conjunto movimento(T,a). Permitindo transições-ε, N pode estar em qualquer estado do fecho_ε(movimento(T,*a*)), após ler *a*.

Um estado em D é de aceitação se for um conjunto de estados do AFN contendo pelo menos um estado de aceitação de N.

Exemplo: (a | b)* abb

Exercício para AFN do exercício anterior construa AFD.

Uso do Lex para gerar automaticamente um analisador léxico

A versão mais comum: flex (Fast Lex), distribuída como parte do pacote de compilação GNU.

Recebe como entrada um arquivo texto contendo definições regulares, juntamente com ações associadas a cada expressão e produz um arquivo de saída que define um procedimento yylex() que é responsável por formar os tokens.

O arquivo de saída é denominado lexyy.c ou lex.yy.c

Convenções lex para definições regulares

- Escape: " - "(*"; \ - \(*
- \n – fim de linha
- \t – tablação
- - zero ou mais concatenações
- + - uma ou mais concatenações
- (,) – define prioridades
- | - ou

- ? – opcional
- [abxz] – classe de caracteres (a|b|x|z)
- [0-9] – intervalo de caracteres.
- [^0-9abc] – conjunto complementar. Qualquer caractere exceto dígito, *a*, *b* ou *c*. Normalmente os metacaracteres perdem o seu efeito quando estão entre colchetes[].
- . qualquer caractere, exceto mudança de linha.
- {xxx} é a expressão regular representada pelo nome xxx.

Formato do arquivo de entrada Lex

{definições}
%%
{regras}
%%
{rotinas auxiliares}

Exemplo:

Arquivo lex para numerar linhas de um texto.

```
%{
/* programa Lex para adicionar números de linhas a linhas de um texto e imprimir o novo texto para a saída padrão */
#include <stdio.h>
```



```
int numlinha=1;
%}
line .*\n
%%
{line}      {printf("%5d %s", numlinha++, yytext);}
%%
main() {
     yylex();
     return 0; }
```

Alguns nomes internos Lex

yylex	função que realiza análise léxica.
yytext	cadeia que casou com a definição corrente.
yyin	arquivo de entrada lex (padrão stdin)
yyout	arquivo de saída lex (padrão stdout)
input	rotina de entrada lex
ECHO	imprime yytext em yyout

Exercícios

1) Faça um programa em lex para modificar os números em notação decimal para hexadecimal e imprimir a quantidade de erros ocorridos.
2) Apresentar apenas as linhas que iniciam ou terminam com *a*.
3) Converter caixa alta em caixa baixa, exceto dentro de comentários.