Análise Léxica

Análise Léxica

Papel do analisador léxico: ler o arquivo fonte em busca de unidades significativas (os *tokens*) instanciadas por lexemas ou átomos.

Também denominado de *scanner*, porque varre o arquivo de entrada eliminando comentários e caracteres indesejáveis ao agrupar caracteres com um papel bem definido.

Análise Léxica

A separação da **análise léxica** da **análise sintática** facilita o projeto e torna o compilador mais eficiente e portável.

Um **análisador léxico** executa tarefas como:

1. contar as linhas de um programa
2. eliminar comentários
3. contar a quantidade de caracteres de um arquivo
4. tratar espaços

Análise Léxica

Tokens – são padrões de caracteres com um significado específico em um código fonte. Definida por um alfabeto e um conjunto de ***definições regulares***

Lexemas – são ocorrências de um *token* em um código fonte, também são chamados de **átomos** por alguns autores

Análise Léxica

- Um mesmo *token* pode ser produzido por várias cadeias de entradas.

- Tal conjunto de cadeias é descrito por uma regra denominada *padrão*, associada a tais *tokens.*

- O padrão reconhece as cadeias de tal conjunto, ou seja, reconhece os *lexemas* que são padrão de um *token*.

Análise Léxica

- Usualmente os padrões são convenções determinadas pela linguagem para formação de classes de *tokens*.
  - **identificadores**: letra seguida por letras ou dígitos.
  - **literal**: cadeias de caracteres delimitadas por aspas.
  - **num**: qualquer constante numérica.

Análise Léxica

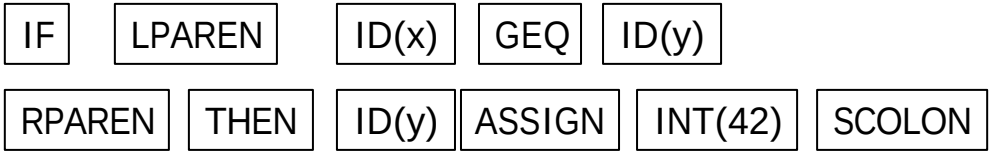
- Os *tokens* usualmente são conhecidos pelo seu *lexema*(seqüência de caracteres que compõe um único *token*) e atributos adicionais.
- Os *tokens* podem ser entregues ao *parser* como tuplas na forma <a, b, ..., n> assim a entrada:
  a = b + 3
- poderia gerar as tuplas:
  <**id**,a> <=, > <**id**, b> <**num**, 3>
- note que alguns *tokens* não necessitam atributos adicionais.

Análise Léxica

- Reconhece e classifica as palavras (tokens)
- Texto de entrada

```
if (x >= y) then
   y = 42;
```

- Cadeia de Tokens reconhecida:



- Elimina Tabulações, comentários, etc.

Análise Léxica

- A declaração C seguinte;
  int k = 123;
- Possui várias subcadeias:
- *int* é o lexema para um *token* tipo **palavra-reservada**.
- = é o lexema para um *token* tipo **operador**.
- *k* é o lexema para um *token* tipo **identificador**.
- *123* é o lexema para um *token* tipo **número literal** cujo atributo valor é 123.
- *;* é o lexema para um *token* tipo **pontuação**.

Análise Léxica

Quais os *tokens* que podem ser reconhecidos em uma linguagem de programação como C ?

palavras reservadas	if else while do
identificadores	
operadores relacionais	< > <= >= == !=
operadores aritméticos	+ * / -
operadores lógicos	&& || & | !
operador de atribuição	=
delimitadores	; ,
caracteres especiais	( ) [ ] { }

Análise Léxica

Quais os *tokens* que podem ser reconhecidos em uma linguagem de marcação como HTML ?

Tags	<html> <body> <table>...
Comentários	<!-- ... -->
Conteúdos	Página de teste…
Especiais	&eaccute;

Análise Léxica

Lexemas podem ter atributos como número da linha em que se encontra no código fonte e o valor de uma constante numérica ou um literal.

Normalmente utiliza-se um único atributo que é um apontador para a **Tabela de Símbolos** que armazena essas informações em registros.

Análise Léxica

O **analisador léxico** simplesmente varre a entrada (arquivo fonte) *em busca de padrões pertencentes a uma linguagem*. A única possibilidade de ocorrer **erro** é aparecer um caracter que não pertence ao alfabeto da linguagem.

Na ocorrência de um erro existem duas possibilidades, ou o projetista realmente esqueceu de incluir o caracter no alfabeto ou realmente o usuário utilizou algum caracter que não pertence ao alfabeto da linguagem.

Análise Léxica

Na ocorrência de erros o analisador léxico pode parar ou entrar em laço infinito. A **modalidade de pânico** pode ser usada para *recuperar erros léxicos* ignorando os caracteres inválidos até encontrar algum que pertença ao alfabeto ou o fim do arquivo.

Outras formas de recuperar erros:

1. remover caracteres estranhos
2. inserir caracteres que faltam
3. substituir caracteres incorretos por corretos
4. trocar dois caracteres adjacentes

Bufferização

Trata de **técnicas** para percorrer arquivos de entrada quando estes forem muito grandes e não houver memória suficiente.

Quando a memória for suficiente o arquivo pode ser aberto e percorrido diretamente.

Normalmente utiliza-se um **esquema de pares de *buffers***.

Bufferização

Em alguns momentos pode ser necessário que o **analisador léxico** precise examinar alguns caracteres a frente de um *token* antes de o reconhecer.

O movimento para frente e para trás no arquivo fonte pode consumir um certo tempo. Para tornar o **analisador léxico** *mais rápido* podem ser utilizados *buffers* em memória principal.

Bufferização

- É muito conveniente que a entrada seja buferizada, i.e., que a leitura da entrada seja fisicamente efetuada em blocos enquanto que logicamente o analisador léxico consome apenas um caractere por vez.

- A buferização facilita os procedimentos de devolução de caracteres.

Erros léxicos

- Poucos erros podem ser detectados durante a análise léxica dada a visão restrita desta fase, como mostra o exemplo:

fi(a== f(x)) *outro_and*;

  - *fi* é a palavra chave *if* grafada incorretamente?
  - *fi* é um identificador de função que não foi declarada faltando assim um separador (‘;’) entre a chamada da função *fi* e o comando seguinte (*outro_and*)?

Erros léxicos

- Ainda assim o analisador léxico pode não conseguir prosseguir dado que a cadeia encontrada não se enquadra em nenhum dos padrões conhecidos.
- Para permitir que o trabalho desta fase prossiga mesmo com a ocorrência de erros deve ser implementada uma estratégia de recuperação de erros.

Recuperação de erros léxicos

- Ações possíveis:

  (1) remoção de sucessivos caracteres até o reconhecimento de um *token* válido (modalidade pânico).

  (2) inserção de um caractere ausente.

  (3) substituição de um caractere incorreto por outro correto.

  (4) transposição de caracteres adjacentes.

Recuperação de erros léxicos

- Tais estratégias poderiam ser aplicadas dentro de um escopo limitado(denominado erros de distância mínima).

  While (a<100) {fi(a==b) break else a++;} ...

- Estas transformações seriam computadas na tentativa de obtenção de um programa sintaticamente correto (o que não significa um programa correto, daí o limitado número de compiladores experimentais que usam tais técnicas).

Especificação de Tokens

Tokens são padrões que podem ser **especificados** através de **expressões regulares**.

Um **alfabeto** determina o conjunto de caracteres válidos para a formação de cadeias, sentenças ou palavras.

Cadeias são seqüências finitas de caracteres. Algumas operações podem ser aplicadas a alfabetos para ajudar na definição de cadeias: concatenação, união e fechamento.

Especificação de Tokens

Concatenação

L.M = {st | s pertence a L e t pertence a M}

União

L U M= {s | s pertence a L ou a M}

Fecho

L*= U Li, i= 0...

Fecho Positivo

L+= U Li, i= 1...

Especificação de Tokens

As **regras** para *definir expressões regulares* sobre um alfabeto são:

1. ε é a expressão regular para a cadeia vazia
2. a é a expressão regular para um símbolo do alfabeto {a}
3. se a e b são expressões regulares, também são expressões regulares:

a) a|b
b) a.b
c) a*
d) a+

Especificação de Tokens

Expressões regulares podem receber um nome (**definição regular**), formando o *token* de um analisador léxico.

Algumas convenções podem facilitar a formação de **definições regulares**

1. Uma ou mais ocorrência (+)
2. Zero ou mais ocorrências (*)
3. Zero ou uma ocorrência (?)
4. Classe de caracteres [a-z]= a|b|...|z

Especificação de Tokens

São definições regulares

letra → [A-Z]|[a-z]
dígito → [0–9]
dígitos → dígito dígito*
identificador → letra[letra|dígito]*
fração_opc → .dígitos|ε
exp_opc → E[+|-|ε]dígitos|ε
num → dígitos fração_opc exp_opc
delim → branco|tabulação|avanço de linha

Reconhecimento de Tokens

Tokens podem ser reconhecidos através de autômatos finitos onde o estado final dispara o reconhecimento de um *token* específico e/ou um procedimento específico (inserir na tabela de símbolo, por exemplo).

Normalmente cria-se um diagrama de transição para representar o reconhecimento de tokens.

Reconhecimento de Tokens

Como são reconhecidos os identificadores e as palavras reservadas ?

Como um compilador sabe o que é uma palavra reservada ?

Há linguagens que permitem usar palavras reservadas como identificadores. Normalmente isto não acontece, mas o reconhecimento de identificadores e palavras reservadas é idêntico. É a tabela de símbolos que trata de identificar as palavras reservadas.

Reconhecimento de Tokens

Em geral a tabela de símbolos é *inicializada* com o registro as **palavras reservadas da linguagem**.

O compilador sempre insere identificadores na tabela de símbolos ? Isto é necessário ?

Não, os identificadores são armazenados apenas uma vez, mas seus atributos podem ser alterados ao longo da análise de um programa.

```
int a;
a= 10;
```

Projeto de Analisador Léxico

1. Definir o alfabeto

2. Listar os ***tokens*** necessários

3. Especificar os ***tokens*** por meio de definições regulares

4. Montar os autômatos para reconhecer os ***tokens***

5. Implementar o analisador léxico

Projeto de Analisador Léxico

EXERCÍCIO – Projetar um analisador léxico para uma calculadora simples com números naturais e reais e operações básicas (soma, subtração, multiplicação e divisão)

Enfoques de Implementação

- Existem 3 enfoques básicos para construção de um analisador léxico:
  - Utilizar um gerador automático de analisadores léxicos (tal como o compilador LEX, que gera um analisador a partir de uma especificação).
  - Escrever um analisador léxico usando uma linguagem de programação convencional que disponha de certas facilidades de E/S.
  - Escrever um analisador léxico usando linguagem de montagem.

Enfoques de Implementação

- As alternativas de enfoque estão listadas em ordem crescente de complexidade e (infelizmente).
- A construção via geradores é praticamente adequada quando o problema não esbarra em questões de eficiência e flexibilidade,
- A construção manual é uma alternativa atraente quando a linguagem a ser tratada não for por demais complexa.

Projeto de Analisador Léxico

Exemplo - seja a cadeia 3.2 + (2 * 12.01), o analisador léxico teria como saída:

3.2 => número real
+ => operador de soma
(=> abre parênteses
2 => número natural
* => operador de multiplicação
12.01 => número real

Projeto de Analisador Léxico

Que símbolo usar como separador de casas decimais?

A calculadora usa representação monetária?

A calculadora aceita espaços entre os operandos e operadores?

O projetista é quem decide sobre as características desejáveis do compilador ou interpretador. Para a maioria das linguagens de programação existem algumas **convenções** que devem ser respeitadas.

1. Definição do Alfabeto

Σ= {0,1,2,3,4,5,6,7,8,9,.,(,),+,-,*,/,\b}

OBS.: projetista deve considerar TODOS os símbolos que são necessários para formar os padrões

2. Listagem dos ***tokens***

OPSOMA: operador de soma
OPSUB: operador de subtração
OPMUL: operador de multiplicação
OPDIV: operador de divisão
AP: abre parênteses
FP: fecha parênteses
NUM: número natural/real

OBS.: projetista deve considerar ***tokens*** especiais e cuidar para que cada ***token*** seja uma unidade significativa para o problema

3. Especificação dos ***tokens*** com definições regulares

OPSOMA → +

OPSUB → -

OPMUL → *

OPDIV → /

AP → (

FP →)

NUM → [0-9]+.?[0-9]*

OBS.: cuidar para que as definições regulares reconheçam padrões claros, bem formados e definidos

4. Montar os autômatos para reconhecer cada ***token***

OBS.: os autômatos reconhecem ***tokens*** individuais, mas é o conjunto dos autômatos em um único autômato não-deterministico que determina o analisador léxico de um compilador, por isto, deve ser utilizada uma numeração crescente para os estados.

5. Implementar o analisador léxico

Existem duas formas básicas para implementar os autômatos: usando a **tabela de transição de estados** ou **diretamente em código**

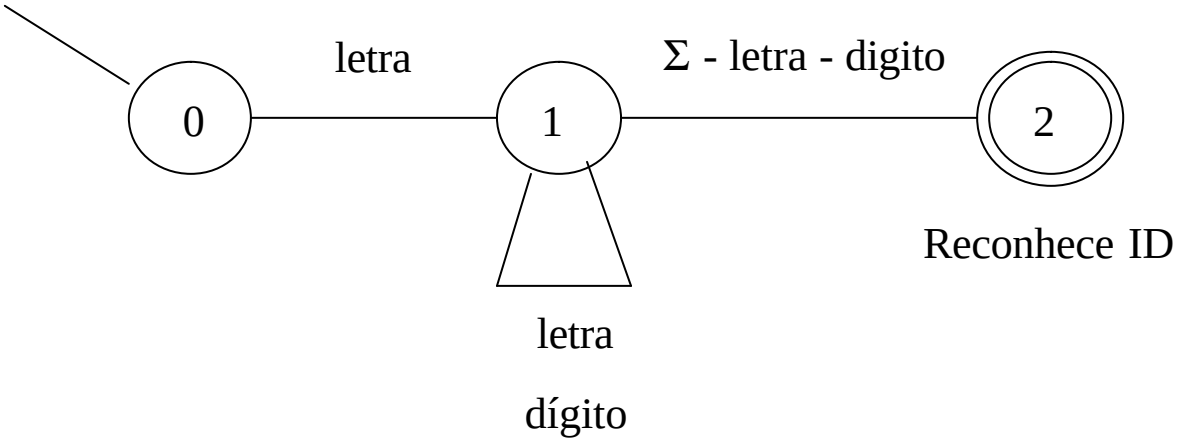
Qual é a mais eficiente? Por que ?

ESTILO DE IMPLEMENTAÇÃO DO LÉXICO

- Cada ***token*** listado é codificado em um número natural
- Deve haver uma variável para controlar o **estado** corrente do autômato e outro para indicar o estado de **partida** do autômato em uso
- Uma função **falhar** é usada para desviar o estado corrente para um outro autômato no caso de um estado não reconhecer uma letra

ESTILO DE IMPLEMENTAÇÃO DO LÉXICO

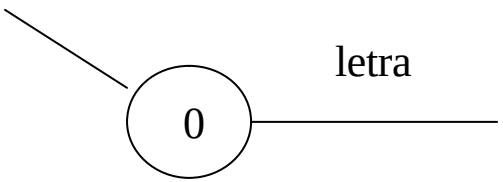
Cada estado é analisado individualmente em uma estrutura do tipo **switch…case**



ESTILO DE IMPLEMENTAÇÃO DO LÉXICO

Cada estado é analisado individualmente em uma estrutura do tipo **switch…case**

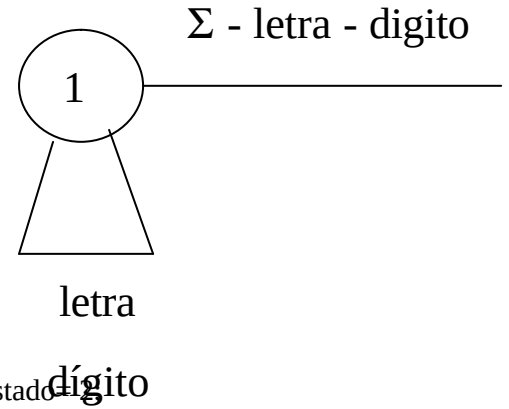
```
int lexico()
{
  while (1)
  {
    switch (estado)
    {
      case 0: c= proximo_caracter();
              if (isalpha(c))
              {
                estado= 1;
                adiante++;
              }
              else
              {
                falhar();
              }
              break;
      …
    }
}
```



ESTILO DE IMPLEMENTAÇÃO DO LÉXICO

Estado 1: iteração sobre letras ou dígitos

```
        …
        case 1: c= proximo_caracter();
                if (isalpha(c) || isdigit(c))
                {
                  estado= 1;
                  adiante++;
                }
                else
                {
                   if ((c == '\n') || (c == '\t') || (c == '\b')) estado= 2;
                   else falhar();
                }
                break;
        …
    }
}
```



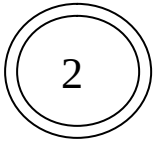
ESTILO DE IMPLEMENTAÇÃO DO LÉXICO

Estado de aceitação:

reconhecimento de um identificador

```
        …
        case 2: estado= 0;
                partida= 0;
                return ID;
                break;

    }
}
```



Reconhece ID