UNISC
UNIVERSIDADE DE SANTA CRUZ DO SUL

Universidade de Santa Cruz do Sul – UNISC
Departamento de informática

COMPILADORES

Análise léxica
Parte 01

Geovane Griesang
geovanegriesang@unisc.br

Compilador

Compilador

"... é um programa de computador que lê um programa escrito em uma linguagem (linguagem fonte) e a traduz em um programa equivalente em outra linguagem (linguagem objeto)".
Aho, Sethi, Ullman.

Função: relatar quaisquer erros no programa fonte detectados durante este processo de tradução.

Portanto:
Cria representações intermediárias do programa
Verifica presença de certos tipos de erros

Compilador

Compilador

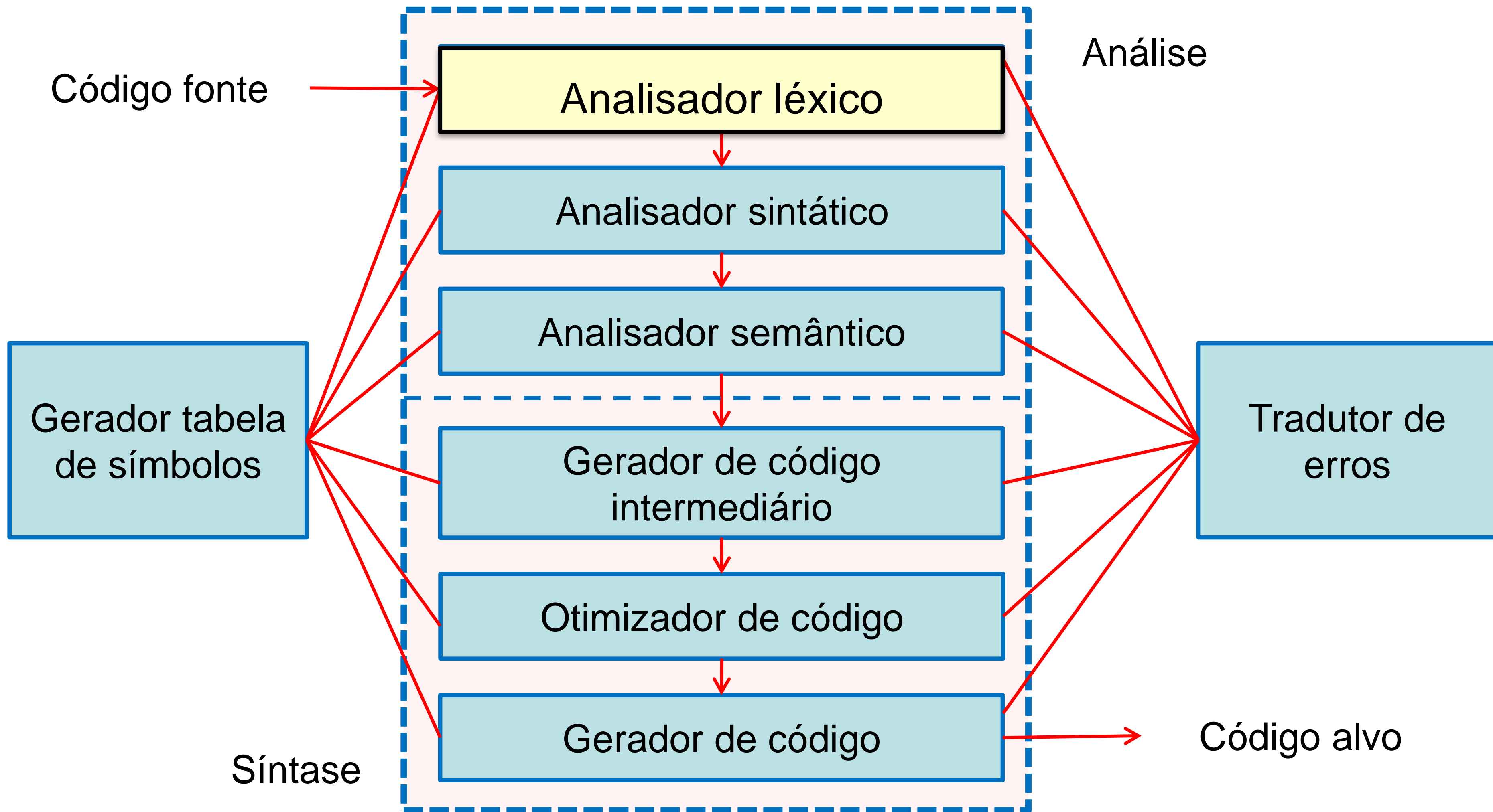
Nesse processo de tradução, há 2 tarefas básicas a serem executadas por um compilador:

Análise (*front-end*), em que o texto de entrada (na ling. fonte) é examinado, verificado e compreendido.

- Análise léxica, sintática e semântica.

Síntese (*back-end*), ou geração de código, em que o texto de saída (na linguagem objeto) é gerado, de forma a corresponder ao texto de entrada.

Compilador - fases



Análise léxica

2. Análise Léxica:

   2.1 especificação de analisadores léxicos;

   2.2 implementação de analisadores léxicos;

   2.3 geradores de analisadores léxicos;

   2.4 tabela de símbolos;

Análise léxica

Funcionalidade dos analisadores léxicos

O analisador léxico (*scanner*) é a parte do compilador responsável por ler caracteres do programa fonte e transformá-los em uma representação conveniente para o analisador sintático.

O analisador léxico lê o programa fonte caractere a caractere, agrupando os caracteres lidos p/ formar os símbolos básicos (*tokens*) da linguagem (identificadores, palavras-chaves, operadores, parêntesis e sinais de pontuação) e passar esta informação para o *parser* (analisador sintático).

Análise léxica

Resumo

Função simplificada do analisador léxico: organizar os caracteres e os agrupar em símbolos (*tokens*).

Entrada: Fluxo de caracteres.

Saída: Fluxo de símbolos.

Símbolos: Palavras reservadas, identificadores de variáveis e/ou procedimentos, operadores, pontuação, ...

Análise léxica

Resumo

Exemplo: montante := saldo + taxa_de_juros * 30;

São identificados os seguintes *tokens*:

Identificador montante

Símbolo de atribuição :=

Identificador saldo

Símbolo de adição +

Identificador taxa_de_juros

Símbolo de multiplicação *

Número 30

Análise léxica

Resumo

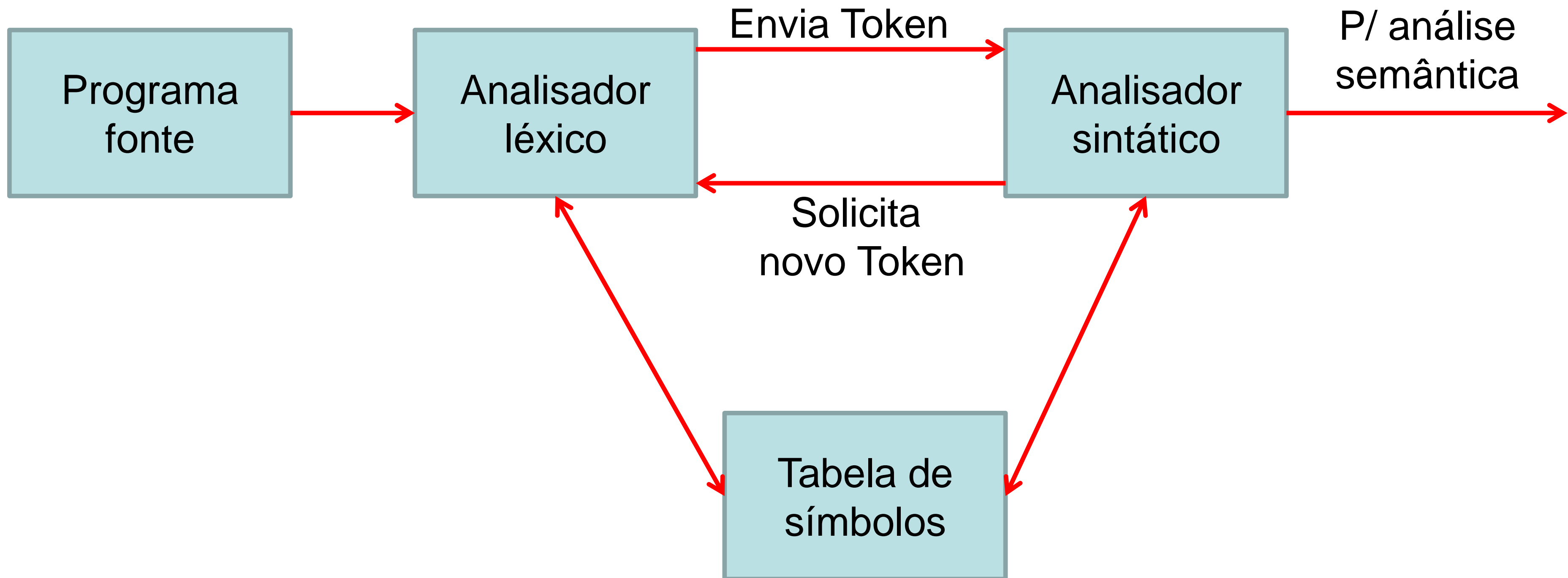
<identificador, 1>,
<:=>,
<identificador, 2>,
<+>,
<identificador, 3>,
<*>,
<número, 30>

Tabela de símbolos

	Nome	Tipo	
1	montante	-	...
2	Saldo	-	
3	Taxa_de_juros	-	
	...		

Análise léxica

Cenário – interações entre os analisadores léxico e sintático

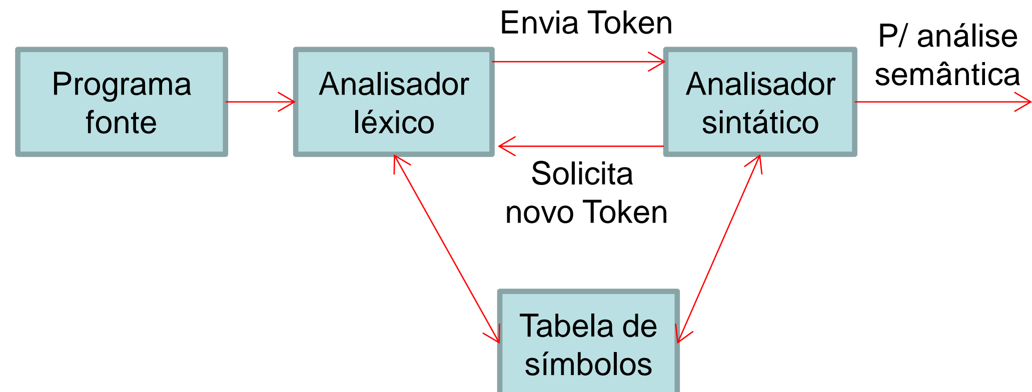


Análise léxica

Cenário – interações entre os analisadores léxico e sintático

No geral, a iteração é implementada fazendo-se com que o analisador sintático (AS) chame o analisador léxico (AL).

A chamada, sugerida pelo comando *getNextToken* (solicita novo *token*), faz com que o analisador leia caracteres de uma entrada até que ele possa identificar o próximo *lexema* e produza para ele o próximo *token*, que retorna ao analisador sintático.



Análise léxica

Vantagens da divisão em análise léxica (AL) e sintática (AS):

Projeto mais simples: diminui a complexidade do analisador sintático que não precisa mais lidar com estruturas foras de seu escopo, como por exemplo, tratamento de caracteres vazios. Portanto, um AS que tivesse que lidar com comentários e espaço em branco como unidades sintáticas seria muito mais complexo (talvez, a mais importante).

Melhorar a eficiência do compilador: técnicas de otimização específicas para o analisador léxico.

Melhor portabilidade: particularidades da linguagem fonte podem ser tratadas diretamente pelo analisador léxico.

Análise léxica

Tokens, padrões e lexemas

Tokens: são padrões de caracteres com um significado específico em um código fonte.

Um *token* é um par consistindo em um valor e atributo opcional.

O nome do *token* é um símbolo abstrato que representa um tipo de unidade léxica, por exemplo, uma palavra chave em particular, ou uma sequencia de caracteres da entrada denotando um identificador.

Nomes de *tokens* são os símbolos da entrada que o analisador sintático processa.

Análise léxica

Tokens, padrões e lexemas

Lexemas: são ocorrências de um *token* em um código fonte, também são chamados de átomos.

Um *lexema* é uma sequencia de caracteres no programa fonte que casa com o padrão para um *token* e é identificado pelo analisador léxico como uma instância desse *token*.

> *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

Tokens, padrões e lexemas

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

O conjunto de cadeias é descrito por uma regra denominada padrão, que está associada a tais *tokens*.

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

> No caso de uma palavra-chave, o padrão é só uma sequencia de caracteres que formam uma palavra-chave. Para identificadores e outros *tokens*, o padrão é uma estrutura complexa, que é "*casada*" por muitas sequencias de caracteres.

Análise léxica

Tokens, padrões e lexemas

Como o analisador léxico é a parte do compilador que lê o texto fonte, ele pode realizar outras tarefas além da identificação de *lexemas*, como:

- Remover comentários
- Tratar espaços em branco (espaço, quebra de linha, tabulação, ...)
- Contar as linhas de um programa
- Contar a quantidade de caracteres de um arquivo

Portanto:

Na leitura do arquivo de entrada varre este eliminando comentários e caracteres indesejáveis

Análise léxica

Tokens, padrões e lexemas

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

Tokens, padrões e lexemas

Token	Descrição informal	Ex. de Lexemas
if	caracteres i, f	If
else	caracteres e, l, s, e	else
comparison	< or > ou <= ou >= ou == ou !=	<=, !=
id	letra seguida por letras e dígitos	pi, score, D2
number	qualquer constante numérica	3.14159, 0, 6.02e23
literal	qualquer caractere diferente de “, cercado por “s	“core dumped”

printf(“Total = %d\n”, score);

Tanto *printf* quanto *score* são lexemas casando com o padrão p/ o *token* id, e “Total = %d\n”, é um lexema casando com o literal.

Análise léxica

Tokens, padrões e lexemas

Os *tokens* usualmente são conhecidos pelo seu *lexema* e atributos adicionais e podem ser entregues ao *parser* (AS) como tuplas na forma:

<a, b, ..., n>

Assim, a entrada a = b + 3, poderia gerar as tuplas:

<id, a> <=, > <id, b> <+, > <num, 3>

Observação: alguns *tokens* não necessitam atributos adicionais.

Análise léxica

Tokens, padrões e lexemas – Exemplo

Texto de entrada:

if (x >= y) then

y = 42;

Cadeia de *tokens* reconhecida:

IF / LPAREN / ID(x) / GEQ / ID(y) / RPAREN / THEN

ID(y) / ASSIGN / INT(42) / SCOLON

Análise léxica

Tokens, padrões e lexemas

Outra tarefa do analisador léxico é correlacionar as mensagens de erro geradas pelo compilador com o programa fonte.

Exemplo: o analisador léxico pode registrar o número de caracteres de quebra de linha, de modo que possa associar um dado número a cada mensagem de erro.

Em alguns compiladores, o analisador léxico faz uma cópia do programa fonte com as mensagens de erro inseridas nas posições apropriadas.

Com isso, pode-se saber qual linha está com erro.

Análise léxica

Erros léxicos

É difícil um analisador léxico saber, sem auxílio de outros componentes, que existe um erro no código fonte.

Por exemplo, se a cadeia de caracteres *fi* for encontrada pela primeira vez em um programa C no contexto: *fi (a == f(x))* ...

Um analisador léxico não tem como saber se fi é a palavra-chave if escrita errada ou um identificador de função não declarada.

Como fi é um *lexema* válido para o *token* id, o analisador léxico precisa retornar o *token id* ao analisador sintático e deixar que alguma fase do compilador trate o erro – provavelmente, o analisador sintático.

Análise léxica

Erros léxicos

Existem estratégias de recuperação para tentar corrigir erros:

Remover um caractere da entrada restante.	aif → if
Inserir um caractere que falta na entrada restante.	f → if
Substituir um caractere por outro caractere.	of → if
Transpor dois caracteres adjacentes.	fi → if

Transformações como essas podem ser experimentadas em uma tentativa de reparar a entrada. Inclusive, essas estratégias fazem sentido, pois a maior parte dos erros léxicos envolve um caractere.

Deve-se analisar se tais estratégias compensam o esforço da correção.

Análise léxica

Buffers de entrada

Buffers: A simples tarefa de ler o programa fonte pode ser acelerada.

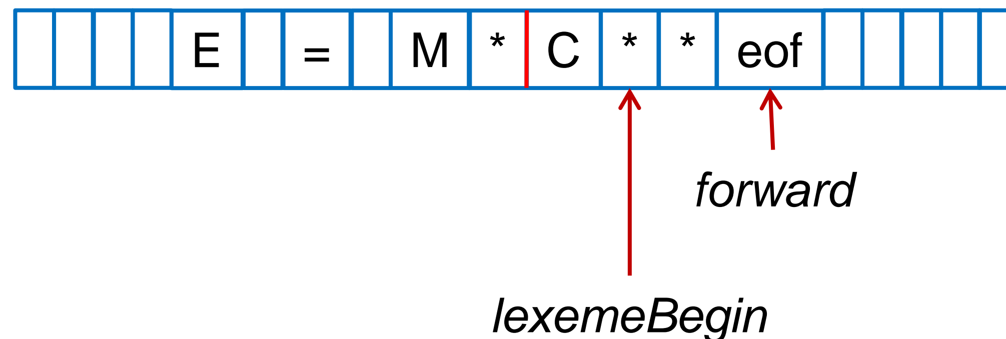
Leitura: essa tarefa se torna difícil pelo fato de c/ frequência precisarmos examinar um ou mais caracteres além do próximo *lexema* para nos certificarmos de ter o *lexema* correto.

Análise léxica

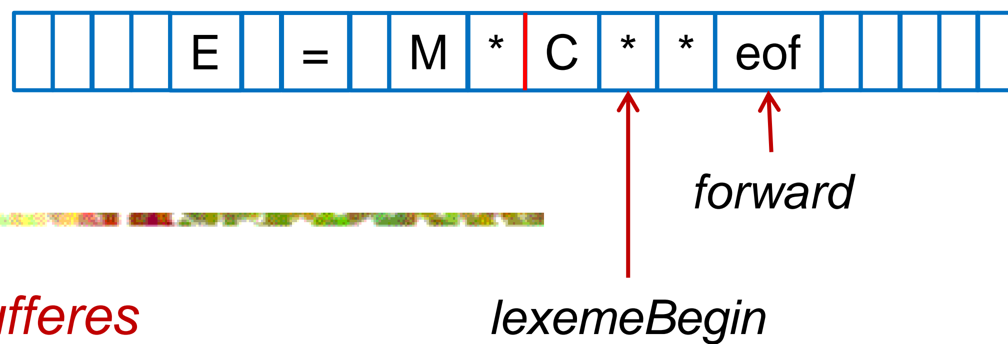
Buffers de entrada – Pares de *bufferes*

Devido a quantidade de tempo necessária para processar caracteres e o grande número de caracteres que precisam ser processados durante a compilação de um programa fonte grande, foram desenvolvidas técnicas especializadas de *buffering* p/ reduzir o custo exigido no processamento de um único caractere de entrada.

Um esquema envolve dois *bufferes* que são recarregados alternadamente.



Análise léxica



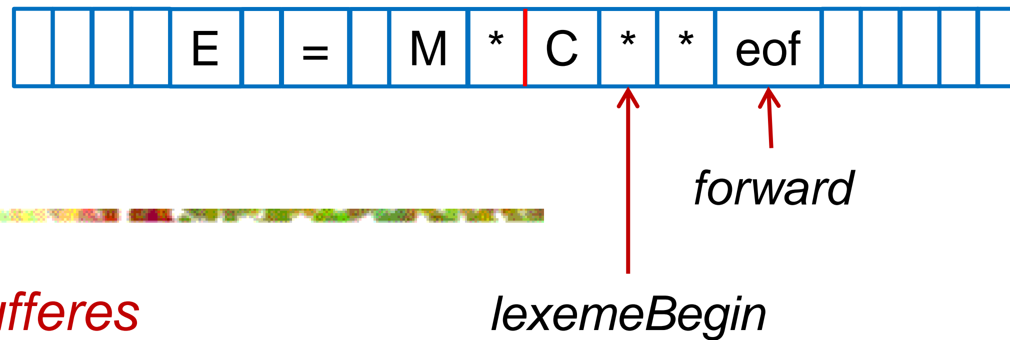
Buffers de entrada – Pares de *bufferes*

Cada *buffer* possui o mesmo tamanho N, e N normalmente corresponde ao tamanho de um bloco de disco, por exemplo, *4096 bytes*.

Usando um comando de leitura do sistema, podemos ler N caracteres p/ um *buffer*, em vez de fazer uma chamada do sistema para cada caractere.

Se restarem menos de N caracteres no arquivo de entrada, então um caractere especial, representado por **eof**, marca o fim do arquivo fonte e é diferente de qualquer caractere possível do programa fonte.

Análise léxica



Buffers de entrada – Pares de *bufferes*

São mantidos dois apontadores para a entrada:

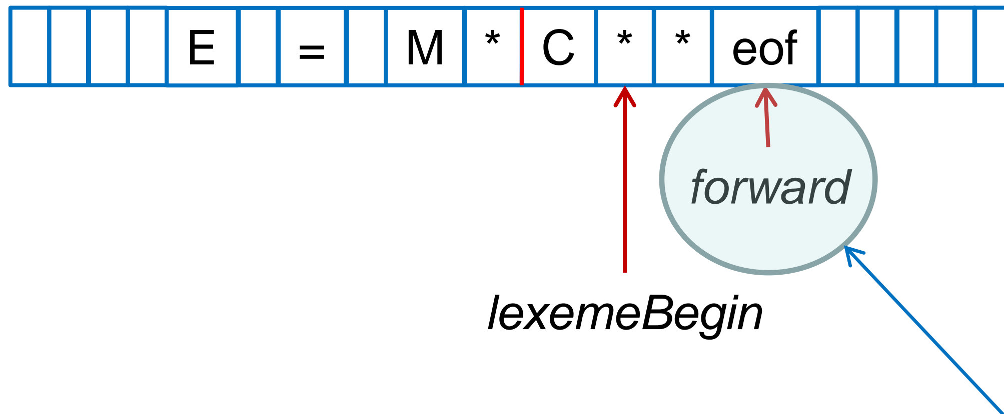
1. O apontador *lexemeBegin* marca o início do lexema corrente, cuja extensão estamos tentando determinar.
2. O *forward* lê adiante, até que haja um casamento de padrão.

Uma vez que o próximo *lexema* é determinado, *forward* é configurado p/ apontar para o último caractere à direita.

Em seguida, após o *lexema* ser registrado como um valor de atributo de um *token* retornado ao analisador sintático, *lexemeBegin* é configurado p/ apontar para o caractere imediatamente após o lexema recém encontrado.

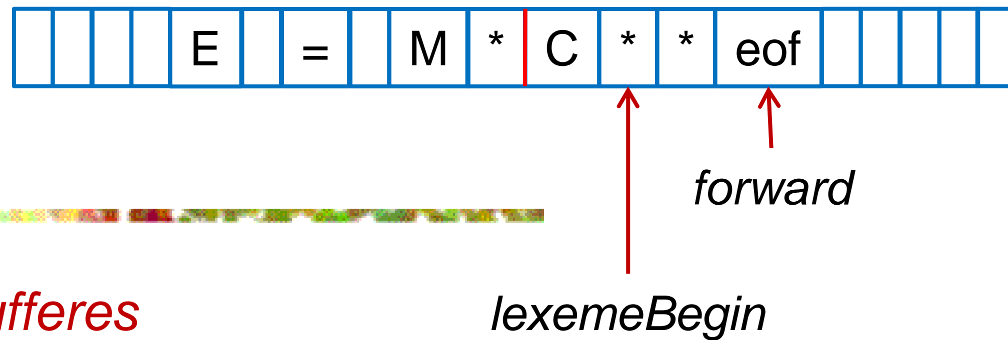
Análise léxica

Buffers de entrada – Pares de *bufferes*



Vemos que o apontador *forward* passou do fim do próximo lexema, * * (o operador de exponenciação - Fortran), e precisa ser rotulado em uma posição à esquerda.

Análise léxica

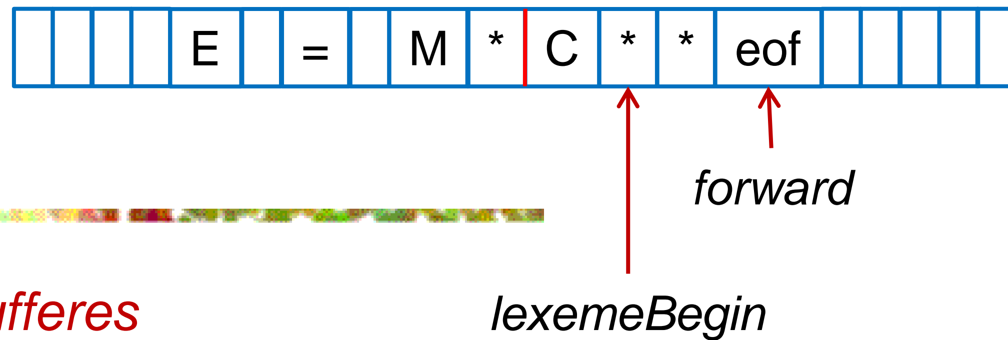


Buffers de entrada – Pares de *bufferes*

Avançar o apontador *forward* exige que primeiro testemos se chegamos ao fim de um dos *bufferes* e, neste caso, precisamos recarregar o outro *buffer* da entrada e mover o apontador *forward* para o início do *buffer* recém carregado.

Se nunca precisarmos examinar tão adiante do *lexema* corrente que a soma do tamanho do *lexema* com a distância que examinamos adiante é maior que N, nunca sobrescreveremos o *lexema* no *buffer* antes de determiná-lo.

Análise léxica



Buffers de entrada – Pares de *bufferes*

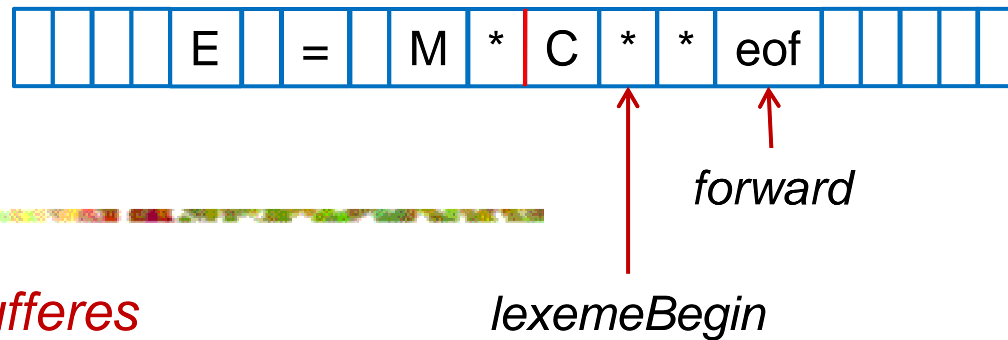
Podemos esgotar o espaço de um buffer?

Na maioria das linguagens são modernas, os *lexemas* são pequenos, e um ou dois caracteres de *lookahead* são suficientes.

Um *buffer* de tamanho N na casa de milhões é suficiente, e o esquema duplo visto anteriormente funciona sem problemas.

Existem, porém, alguns riscos. Por exemplo, se as cadeias de caracteres puderem ser muito grandes, estendendo-se por muitas linhas, talvez nos deparemos com a possibilidade de um *lexema* ser maior que N.

Análise léxica



Buffers de entrada – Pares de *bufferes*

Podemos esgotar o espaço de um buffer?

Para evitar problemas com cadeias de caracteres longas, podemos tratá-las com uma concatenação de componentes, um de cada linha na qual a cadeira de caracteres é escrita.

Exemplo, Java adota a convenção de representar cadeias de caracteres longas escrevendo uma parte em cada linha e concatenando-as com um operador + ao final de cada parte.

Análise léxica

Especificação de *tokens*

Tokens podem ser especificados através de Expressões Regulares (ER)

Um alfabeto determina o conjunto de caracteres válidos para a formação de cadeias, sentenças ou palavras, ou seja, é qualquer conjunto finito de símbolos.

Cadeias são sequências finitas de caracteres.

Linguagens: qualquer conjunto de algum alfabeto fixo. Definição ampla, incluem linguagens abstratas como $\varnothing$, o conjunto vazio $\{\}$, ou $\{\varepsilon\}$.

Análise léxica

Especificação de *tokens* - Operações sobre linguagens

Algumas operações podem ser aplicadas a alfabetos para auxiliar na definição de cadeias. As mais importantes são: união, concatenação e fechamento.

Ex. de concatenação: se *x = mal* e *y = tratar*, *então xy = maltratar*

Fechamento: O fecho (*Kleene*) de uma linguagem L, indicado por L*, é o conjunto de cadeias obtidas concatenando L zero ou mais vezes.

L^0, a “concatenação de L zero vezes” é definida como {ε}, e, L^i é $L^{i-1}L$

L+ é o fechamento positivo, mesmo que L^0, mas sem o ε, ao menos que seja o próprio L.

Análise léxica

Especificação de *tokens* - Operações sobre linguagens

Operação	Cadeia
União de L e M	L U M = {s | s está em L ou s está em M}
Concatenação de L e M	LM = {st | s está em L e t está em M}
Fecho *Kleene* de L	$L^* = \bigcup_{i=0}^{\infty} L^i$

Exemplo: L é o conjunto de letras {A, B, ..., Z, a, b, ..., z} e D o conjunto de dígitos {0, 1, ..., 9}.

Análise léxica

União de L e M
Concatenação de L e M
Fecho *Kleene* de L

Especificação de *tokens* - Operações sobre linguagens

Algumas outras linguagens que podem ser constituídas das linguagens L e D, usando os operadores da tabela apresentada anteriormente.

L = {A, B, ..., Z, a, b, ..., z} e D = {0, 1, ..., 9}.

1. L U D: é o conjunto de letras e dígitos.
2. LD: Conj. de cadeias de tamanho 2, letra seguida de dígito.
3. L^4: Conj. de todas as cadeias de quatro letras.
4. L^*: Conj. de todas as cadeias, inclusive a cadeia vazia ε.
5. $L(L \cup D)^*$: Conj. das cadeias de letras e dígitos, iniciando com letras.
6. D^+: Conj. De todas as cadeias com um ou mais dígitos.

Análise léxica

Especificação de *tokens* - Expressões Regulares (ER)

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

Especificação de *tokens* - Expressões Regulares (ER)

Exercício/exemplo: Descrever o conjunto de identificadores válidos em C.

letra_ = {a, ..., z, A, ..., Z, _ }
dígito = {0, ..., 9}

Um identificador em C deve iniciar com uma letra ou o *underline* (_). Em seguida, pode ter números, *underline* e/ou letras, ou ainda, encerrar com um identificador de tamanho 1.

letra_	(Letra_ | dígito)	*

Análise léxica

Especificação de *tokens* - Expressões Regulares (ER)

ERs são construídas recursivamente a partir de ERs menores, usando as regras de definição de ER.

As regras para definir expressões regulares sobre um alfabeto (∑) e as linguagens que essas expressões denotam são:

1. $\boldsymbol{\varepsilon}$ é a expressão regular para a cadeia vazia, ou seja, $L(\varepsilon)$ é $\{\varepsilon\}$.

2. Se a é um símbolo pertencente a ∑, então a é uma expressão regular e, L(a) = {a}.

Análise léxica

Especificação de *tokens* - Expressões Regulares (ER)

Existem quatro partes na indução, por meio das quais ERs maiores são construídas a partir de ERs menores. Suponhamos que r e s sejam ERs denotando as linguagens L(r) e L(s), respectivamente.

1. (r)|(s) é uma ER denotando a linguagem L(r) U L(s).

2. (r)(s) é uma ER denotando a linguagem L(r)L(s).

3. (r)* é uma ER denotando (L(r))*.

4. (r) é uma ER denotando L(r) ou (L(r)).

Análise léxica

Especificação de *tokens* - Expressões Regulares (ER)

As ERs normalmente contêm pares de parênteses desnecessários. Pode-se retirar pares de parênteses se algumas convenções forem adotadas.

a) operador unário * possui precedência mais alta e é associativo à esquerda.

b) A concatenação possui a 2ª maior precedência, e é associativa à esquerda.

c) | possui a precedência mais baixa, e é associativa à esquerda.

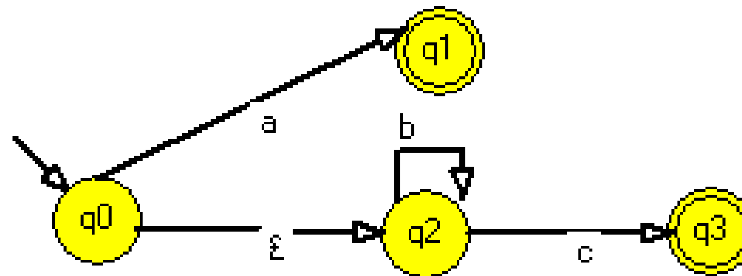
Análise léxica

Especificação de *tokens* - Expressões Regulares (ER)

As ERs normalmente contêm pares de parênteses desnecessários. Pode-se retirar pares de parênteses se algumas convenções forem adotadas.

Exemplo, subtituir (a)|((b)*(c)) por a|b*c

Conjunto de cadeias que são um único a ou são zero ou mais b seguido por um c.



Análise léxica

Especificação de *tokens* - Expressões Regulares (ER)

Exemplo: Considere ∑ = {a, b}.

1. a ER a|b denota a linguagem {a, b}

2. (a|b)(a|b) denota {aa, ab, ba, bb}, a linguagem de todas as cadeias de tamanho dois sob o alfabeto ∑.

   Outra ER para mesma linguagem poderia ser: aa|ab|ba|bb

3. a* denota todas as cadeias de zero ou mais as: {ε, a, aa, ...}

Análise léxica

Especificação de *tokens* - Expressões Regulares (ER)

Exemplo: Considere ∑ = {a, b}.

4. (a|b)* denota todas as cadeias de zero ou mais instâncias de as e bs: {ε, a, b, aa, ab, ba, bb, aaa, ...}

   Outra ER para mesma linguagem é (a*b*)*

5. a|a*b denota a linguagem {a, b, ab, aab, aaab}

Análise léxica

Especificação de *tokens* - Expressões Regulares (ER)

Uma linguagem que pode ser definida por uma ER é um conjunto regular.

Se duas ER r e s denotarem o mesmo conjunto regular, pode-se dizer que são equivalentes e pode ser escrito assim: r = s.

Exemplo: (a | b) = (b | a)

Análise léxica

Especificação de *tokens* - Expressões Regulares (ER)

Leis algébricas para expressar ER:

<table>
<tr><th>Lei</th><th>Descrição</th></tr>
<tr><td>r|s = s|r</td><td>| é comutativo</td></tr>
<tr><td>r|(s|t) = (r|s)|t</td><td>| é associativo</td></tr>
<tr><td>r(st) = (rs)t</td><td>A concatenação é associativa</td></tr>
<tr><td>r(s|t) = rs|st; (s|t)r = sr|tr</td><td>A concatenação distribui entre |</td></tr>
<tr><td>εr = rε = r</td><td>ε é o elemento identidade para concatenação</td></tr>
<tr><td>r* = (r|ε)*</td><td>ε é garantido em um fechamento</td></tr>
<tr><td>r** = r*</td><td>* É igual a potência</td></tr>
</table>

Análise léxica

Especificação de *tokens* – Definições regulares

Por conveniência de notação, podemos dar nome a certas ERs e usar esses nomes em expressões subsequentes, como se os nomes fossem os próprios símbolos.

Ex. 1: Os identificadores de C são cadeias de letras, dígitos e sublinhados.

*letter_* → A | B | ... | Z | a | b | ... | z | _
digit → 0 | 1 | ... | 9
id → *letter_* (*letter_* | *_digit*) *

Ex. 2: *digits* → *digit digit**

Análise léxica

Especificação de *tokens* – Definições regulares

Ex. 3: *optionalFraction* → **.** *digits* | ε

optionalExponent → (E (+ | - | ε) *digits*) | E

number → *digits optionalFraction optionalExponent*

Uma *optionalFraction* é um ponto decimal seguido por um ou mais dígitos ou não existe (a cadeia vazia).

Um *optionalExponent*, se não estiver faltando, é a letra E seguida por um sinal de + ou – opcional, seguido por um ou mais dígitos.

Análise léxica

Especificação de *tokens* – Extensões de expressões regulares

1. Uma ou mais instâncias. O operador unário pós-fixado + representa o fechamento positivo de um ER e sua linguagem.

Ou seja, se r é uma ER, então $(r)^+$ denota a linguagem $(L(r))^+$. O operador + tem a mesma precedência e associatividade do operador *.

Duas leis algébricas úteis: $r^* = r^+|\varepsilon$ e $r^+ = rr^* = r^*r$, se relacionado ao fecho e ao fecho positivo de *Kleene*, respectivamente.

Análise léxica

Especificação de *tokens* – Extensões de expressões regulares

2. Zero ou uma instância. O operador unário pós-fixado ?.

Ou seja, se $r?$ é equivalente a $r|\varepsilon$, de outra maneira, $L(r?)=L(r)U\{\varepsilon\}$. O operador ? tem a mesma precedência e associatividade dos operadores * e +.

3. Classes de caracteres: uma ER $a_1|a_2|...|n_i$, onde os a_i's são cada um dos símbolos do alfabeto, pode ser substituída pela abreviação $[a_1a_2...a_n]$.

Importante: quando $a_1,a_2,...,a_n$ for uma sequencia lógica, pode-se substituir por a_1-a_n. Assim, [abc] é abreviatura de a|b|c, e [a-z] é abreviatura de a|b|...|z.

Análise léxica

Especificação de *tokens* – Extensões de expressões regulares

Exemplo de classes de caracteres

*letter_* → [A-Za-z_]

digit → [0-9]

id → letter_ (letter_ | digit)*

optionalFraction → **.** *digits* | ε
optionalExponent → (E (+ | - | ε) *digits*) | E
number → *digits optionalFraction optionalExponent*

digit → [0-9]

digits → digit$^+$

number → digits (. digits)? (E [+-]? Digits)?

Análise léxica

x	o caractere 'x'
"x"	o caractere 'x', mesmo que este seja um meta-caractere
.	todos os caracteres exceto '/n' ("*newline*")
[xyz]	'x', 'y' ou 'z'
[x-z]	todos os caracteres entre 'x' e 'z' (classe)
[^x-z]	todos os caracteres, exceto 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 expressão da definição referente a nome

Análise léxica

<table>
<tr><td>\x</td><td>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 caractere ‘x’ (necessário para se poder aceder aos meta-caracteres)</td></tr>
<tr><td>\0</td><td>o caractere nulo (NUL, código ASCII 0)</td></tr>
<tr><td>\123</td><td>o caractere com valor octal 123</td></tr>
<tr><td>\x2a</td><td>o caractere com valor hexadecimal 2ª</td></tr>
<tr><td>(r)</td><td>a expressão regular r, necessário para contornar as regras de precedência, ver mais abaixo</td></tr>
<tr><td>rs</td><td>concatenação de expressões regulares r e s</td></tr>
<tr><td>r|s</td><td>r ou s</td></tr>
<tr><td>r/s</td><td>r, mas só se for seguido de um s</td></tr>
<tr><td>^r</td><td>r no princípio de uma linha</td></tr>
<tr><td>r$</td><td>r no fim de uma linha</td></tr>
<tr><td><s>r</td><td>r, mas só num condição inicial</td></tr>
</table>

Análise léxica

Exercícios:

Descreva as linguagens denotadas pelas seguintes ER:

a) a(a|b)*a. L = {aa, aaa, aba, aaaa, aaba, abaa, abba, ...}

b) ((ε|a)b*)* L = {ε, a, b, aa, bb, ba, ab, aab, abb, ...}

c) (a|b)*a(a|b)(a|b) L = {}

d) (aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*

L={}

Análise léxica

Exercícios:

Definir as expressões regulares capazes de definir:

Exemplo: Número de matrículas da UNISC

m numInicial $(num)^*$

numInicial → [1-9];

num → [0-9]

a) Números de telefones no Brasil

b) Placas de carro no Brasil

Análise léxica

Avaliação:

Disponível no EAD, a partir das 20:30h.

10 questões para serem respondidas.

3 tentativas.

Nota é igual a média tirada nas três tentativas (ou menos)

Intervalo entre as tentativas: 24h.

Boa sorte!!

COMPILADORES

Obrigado!!

Prof. Geovane Griesang
geovanegriesang@unisc.br