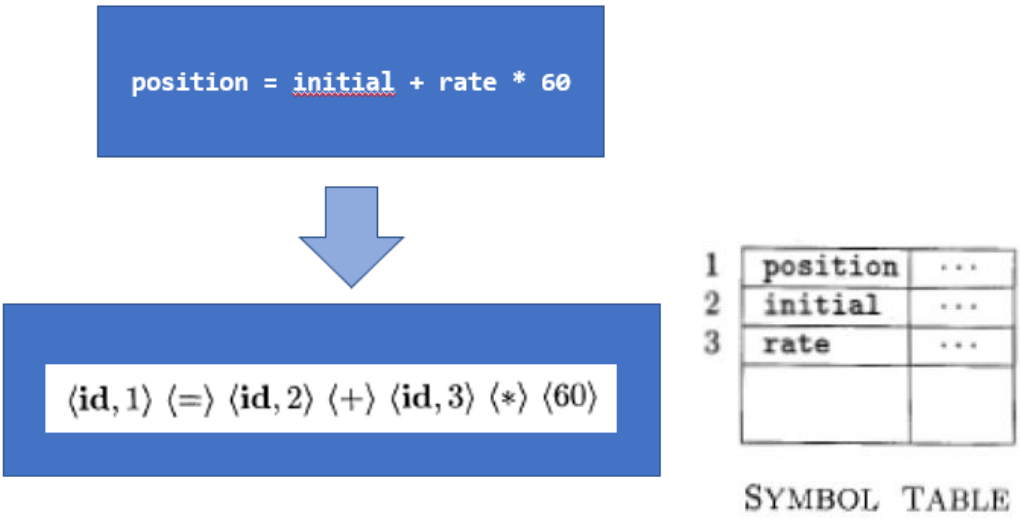
Análise Léxica

Introdução

Definições iniciais:

- *Lexeme*: seqüência de caracteres em um programa fonte que forma uma unidade léxica.
- Padrão: descrição do formato que *lexemes* podem apresentar.
- *Token*: objeto que contém um par de informações, que são um identificador do tipo de unidade léxica e um valor de atributo(s). A segunda informação é opcional.

Um analisador léxico deve ler os caracteres do programa fonte (entrada para o analisador), agrupá-los em *lexemes* e produzir como saída uma seqüência de *tokens*, um para cada *lexeme* do programa fonte.



Tokens

Um token é formado por um código que define seu tipo, e um possível valor de atributo(s) associado(s).

token = <CÓDIGO> ou <CÓDIGO, ATRIBUTO>

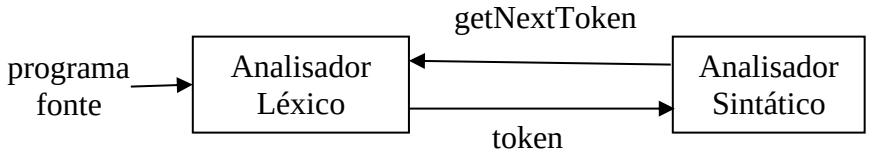
Os atributos podem assumir valores de tipos diferentes, como cadeias de caracteres, números inteiros ou reais, dependendo do tipo do token.

Tipos de tokens, usados em muitas linguagens de programação:

- Um código diferente para cada palavra-chave. Ex: <KW_IF> para palavra-chave “if”, <KW_CLASS> para palavra-chave “class” etc.
- Um código para representar os demais identificadores da linguagem (que não são palavras-chaves). Nesse caso, o atributo pode armazenar a representação textual. Exemplo: <IDENT, “tempo”> para um identificador “tempo”.
- Um código para cada operador. Exemplo: <OP_ADD> para “+”. Ou então, agrupar operadores em classes e usar atributos para diferenciá-los. Ex: <OP_ARIT,ADD> para “+”,<OP_COMP,LT> para “<”.
- Um ou mais códigos diferentes para constantes. Atributos podem ser usados para armazenar o valor. Ex: <CONST_INT, 123> para constante inteira “123”.
- Códigos para diferenciar cada sinal de pontuação. Ex: <LEFTPAR> para “(”, <COMMA> para “,”.

Interação com analisador sintático:

Em geral, o analisador léxico não processa toda a entrada em um único passo gerando uma sequência de tokens. Ele é acionado sob demanda pelo analisador sintático.



O analisador léxico deve oferecer um serviço *getNextToken* que inicia a análise do programa fonte no ponto após a finalização da construção do último *lexeme*; a entrada é percorrida até a identificação do próximo *lexeme*, que é usado para construir um *token*, que é finalmente retornado.

Razões para separar análise léxica de análise sintática:

1. **Simplicidade no projeto** - é a mais importante razão. Por exemplo, um analisador sintático não precisará lidar com elementos como comentários, espaços em branco etc. No projeto de uma nova linguagem, a separação das definições de elementos léxicos e sintáticos leva a um projeto mais claro e organizado.
2. A eficiência do compilador pode ser melhorada. Técnicas de otimização de código específicas para análise léxica podem ser aplicadas. Uso de *buffers* pode aumentar a velocidade da leitura dos caracteres da entrada.
3. A portabilidade é aumentada. Peculiaridades específicas de dispositivos de entrada podem ficar restritas ao analisador léxico.

Abordagens para construção de analisadores léxicos

Duas abordagens principais podem ser utilizadas para a construção de analisadores léxicos:

- construção à mão, usando linguagem de programação convencional;
- usando ferramentas para geração automática.

Em ambas as abordagens, os padrões são definidos, em geral, usando expressões regulares ou formalismos similares. Uma notação conveniente é o uso de nomes atribuídos a expressões regulares, que podem ser usadas em outras definições. Ex:

```
letter_ -> A | B | ... | Z | a | b | ... | z | _
digit -> 0 | 1 | ... | 9
id -> letter_ ( letter_ | digit ) *
digits -> digit digit*
optFrac -> (. digits) | λ
optExp -> (E (+ | - | λ) digits ) | λ
number -> digits optFrac optExp
```

Na notação usada acima, os símbolos | ( ) e * são meta-símbolos usados para definir expressões regulares. Os nomes usados como "variáveis" nas definições são apresentados em itálico.

O analisador deve definir quais nomes identificam o início de um lexeme. Na definição acima, por exemplo, o nome *id* pode ser associado a tokens de tipo "identificador" e *number* pode ser associado a tokens de tipo "constante inteira".

Extensões para essa notação muito utilizadas incluem:

- operador "+" como meta-símbolo para denotar 1 ou mais ocorrências;
- operador "?" para denotar 0 ou 1 ocorrência;
- "[ ]"como abreviatura para classes de caracteres - ex: [a-z] representa a | b | ... | z.

Usando essas extensões, as definições apresentadas podem ser simplificadas como a seguir.

```
letter_ -> [A-Za-z_]
digit -> [0-9]
id -> letter_ ( letter_ | digit ) *
digits -> digit+
number -> digits (. digits)? (E [+-]? digits )?
```

Simulando analisador léxico

Conforme vimos nas seções anteriores, os possíveis tipos de lexemas válidos em uma linguagem devem ser descritos por padrões. Uma forma comum de se especificar esses padrões é usar formalismos baseados em expressões regulares.

A tabela a seguir apresenta um exemplo com uma listagem de padrões e códigos associado a cada um deles, definindo um conjunto de possíveis lexemas. A tabela indica também o que fazer quando encontrar caracteres que devem ser desprezados, ou que não pertencem a nenhum lexema válido.

PADRÃO	código	ATRIBUTO
'('	1	
')'	2	
'{'	3	
'}'	4	
';'	5	
'.'	6	
‘=’	7	
'>='	8	
'if'	9	
letra (letra U digito)*	10	Nome do identificador
digito+	11	Valor da constante inteira
digits (. *digits*)? (E [+-]? *digits*)?	12	Valor da constante real
espaço U <EOL>	desprezar	
'//' char* <EOL>	desprezar	
se todos padrões falharem	255	

Procedimento:

- Analisar a entrada e tentar casamento da maior sequência de caracteres possível com algum padrão, **na ordem fornecida** (de 1 a 11).
- Quando um padrão for encontrado, um objeto na forma <CÓDIGO, ATRIBUTO> ou <CÓDIGO> deve ser construído.

Exemplo:

A entrada abaixo irá produzir a sequência de tokens apresentada, seguindo o procedimento definido e baseando na tabela de padrões e códigos fornecida.

```
if (x >= 1.234e+2) {
// ultrapassou limite permitido
    @
    c.y = 0;
}
```

Tokens produzidos:

```
<9>
<1>
<10, “x”>
<8>
<12, 123.4>
<2>
<3>
<255, ‘@’>
<10, “c”>
<6>
<10, “y”>
<7>
<11, 0>
<5>
<4>
```