



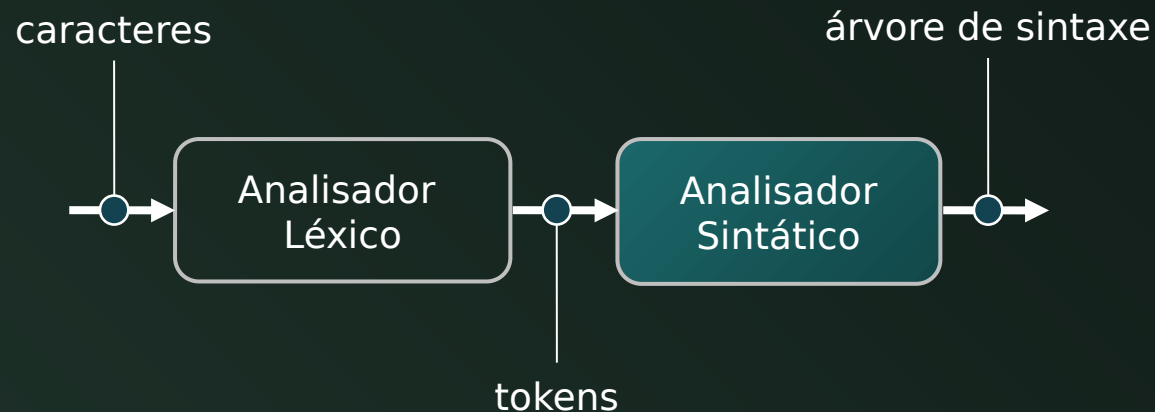
Judson Santos Santiago

Análise Léxica

Compiladores

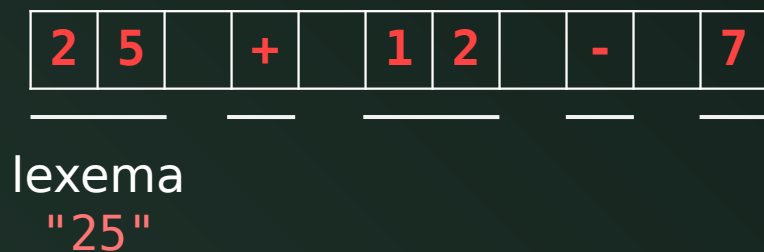
Introdução

- O tradutor implementado até agora está incompleto
 - Trabalha diretamente com os caracteres individuais da entrada:
 - Caracteres estranhos (ex.: espaços, tabulações, etc.) provocam erros
 - Números com mais de um dígito (ex.: 25, 84, etc.) também



Análise Léxica

- Um **analisador léxico** deve realizar as seguintes **tarefas**:
 - Ler caracteres da entrada e os agrupar em **lexemas**
 - Formar objetos, chamados **tokens**, que armazenam os lexemas
 - **Ignorar espaços**, tabulações e saltos de linha



<num, 25> <+> <num, 12> <->
<num, 7>
token

Análise Léxica

- Um **lexema** é uma **sequência de caracteres da entrada**
 - O analisador léxico encontra os lexemas e constrói os tokens
 - O analisador sintático trabalha com tokens

"25"

- Um **token** é um **símbolo terminal** da gramática
 - Possui atributos com informações adicionais sobre o símbolo
Ex.: valor, tipo, escopo, etc.

<num, 25


>


Tradutor

- O **tradutor** trabalhava com:
 - Soma e Subtração de dígitos individuais
- O **analisador léxico** permitirá processar:
 - Números e identificadores
 - Espaços em branco (espaços, tabulações e saltos de linha)
- Ele será **estendido** para considerar também:
 - Multiplicação/Divisão
 - Expressões parentizadas

Gramática

- O esquema de tradução estendido é mostrado abaixo:

```
expr  expr + term { print('+') }  
    | expr - term { print('-') }  
    | term
```

```
term  term * fact { print('*') }  
    | term / fact { print('/') }  
    | fact
```

```
fact  (expr)  
    | num { print(num.valor) }  
    | id { print(id.nome) }
```

num.valor e id.nome
são atributos dos tokens
(símbolos terminais)

Leitura da Entrada

- O analisador léxico precisa sempre ler um caractere à frente para decidir sobre o token reconhecido:
 - Se ' $>$ ' for encontrado, é preciso ler o próximo caractere:
 - Se for achado um ' $=$ ' temos o token "maior ou igual a"
 - Se for achado algo diferente temos o token "maior"

```
if (x >= 5)
{
}
```

- Nesse processo, o analisador muitas vezes lerá um caractere além do token

Leitura da Entrada

- Uma técnica para implementar a leitura da entrada é usar um **buffer de caracteres** que permita
 - O consumo de caracteres do buffer
 - A **devolução** de caracteres ao buffer



```
char ch;  
cin.get(ch);  
  
cin.unget();  
cin.putback('1');
```

Em C++ o método **unget()** coloca de volta o último caractere lido e **putback(char)** coloca de volta o caractere passado como parâmetro

Leitura da Entrada

- Uma **alternativa mais simples** é usar uma variável ***peek***
 - Ler um caractere à frente normalmente é suficiente
 - A variável contém o próximo caractere da entrada
- O analisador léxico **lê à frente apenas quando necessário**
 - O operador '*****' pode ser identificado sem leituras adicionais
 - Nesse caso a **variável *peek*** receberá um espaço em branco
 - Depois de identificarmos um token, ***peek*** conterá:
 - O caractere seguinte ao lexema
 - Um espaço em branco

Remoção de Espaços

- O **tradutor** enxergava cada caractere da entrada
 - Caracteres estranhos, como **espaços em branco**, **causavam erros**
 - $9-5+2$ 
 - $9 - 5 + 2$ 
- A maioria das linguagens permite:
 - Espaços em branco, tabulações e saltos de linha
 - Comentários – funcionam como espaços em branco
- Se o **analisador léxico eliminar os espaços**, o analisador sintático nunca precisará considerá-lo

Remoção de Espaços

- O **pseudocódigo** abaixo:
 - Ignora espaços em branco, tabulações e saltos de linha
 - Mantém uma contagem do número da linha

```
for (;; peek = próximo caractere da entrada)
{
    if (peek é espaço ou tabulação) não faz nada;
    else if (peek é quebra de linha) line = line + 1;
    else break;
}
```

Reconhecendo Constantes

- Existem duas formas de definir **constantes inteiras**:
 - Criar um símbolo terminal, por exemplo **num**, para as constantes
 - Incorporar a sintaxe das constantes na gramática
- O **tratamento no analisador léxico** é mais simples
 - Ele também vai transformar os dígitos em números inteiros
 - Os números são tratados como uma única unidade:
 - No analisador sintático
 - Na tradução

Reconhecendo Constantes

- Uma **sequência de dígitos** se transformará em um **token**
 - Um símbolo terminal **num** junto com um atributo de valor inteiro

31 + 28 - 59
<num,31> <+> <num,28> <-> <num,59>

```
if (peek é um dígito)
{
    v = 0
    do
    {
        v = v * 10 + valor inteiro do dígito peek;
        peek = próximo caractere da entrada;
    }
    while (peek é um dígito);
    return token <num,v>;
}
```

Palavras-Chave e Identificadores

- As sequências de caracteres da entrada podem formar:
 - **Palavras-chaves**: construções das linguagens de programação
Ex.: **for**, **while**, **do-while**, **if**, etc.
 - **Identificadores**: nomes utilizados no código fonte
Ex.: nomes de variáveis, funções, classes, etc.
- As gramáticas tratam **identificadores** como símbolos terminais

```
cont = cont + 1;
```

```
<id, "cont"> <=> <id, "cont"> <+> <num, 1> <;>
```

Palavras-Chave e Identificadores

- As palavras-chave e os identificadores normalmente seguem as mesmas regras de formação

- Inicia com uma letra ou sublinhado seguido de um ou mais caracteres

abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ

0123456789

—

- É mais fácil distinguir usando palavras-chave reservadas
 - Um lexema é um identificador apenas se não for uma palavra-chave

Palavras-Chave e Identificadores

- O analisador léxico pode usar uma **tabela** para armazenar **identificadores e palavras-chave**
 - A tabela resolve vários problemas:
 - Isola a representação em "sequência de caracteres"
 - As sequências podem ser acessadas por **referências** ou **apontadores** para a tabela
 - As palavras reservadas podem ser implementadas **inicializando a tabela** com as **sequências de caracteres reservadas** e seus tokens
 - Essa tabela pode ser uma *hash table*

string	token
"if"	<if>
"while"	<while>
"for"	<for>
"cont"	<id>
"val"	<id>

```
unordered_map  
<string, token>  
id_table;
```


Palavras-Chave e Identificadores

- O **pseudocódigo** transforma sequências de caracteres em **tokens**:

```
if (peek contém uma letra)
{
    junta letras ou dígitos em um buffer b;
    s = sequência de caracteres formada pelos caracteres em b;
    t = token retornado por id_table.find(s);
    if (t não é nulo)
        return t;
    else
    {
        Entra com o par chave-valor (s, <id,s>) em id_table;
        return token <id,s>;
    }
}
```

Analizador Léxico

- Os fragmentos de pseudocódigo apresentados até agora podem ser reunidos para formar **nosso analisador léxico**:

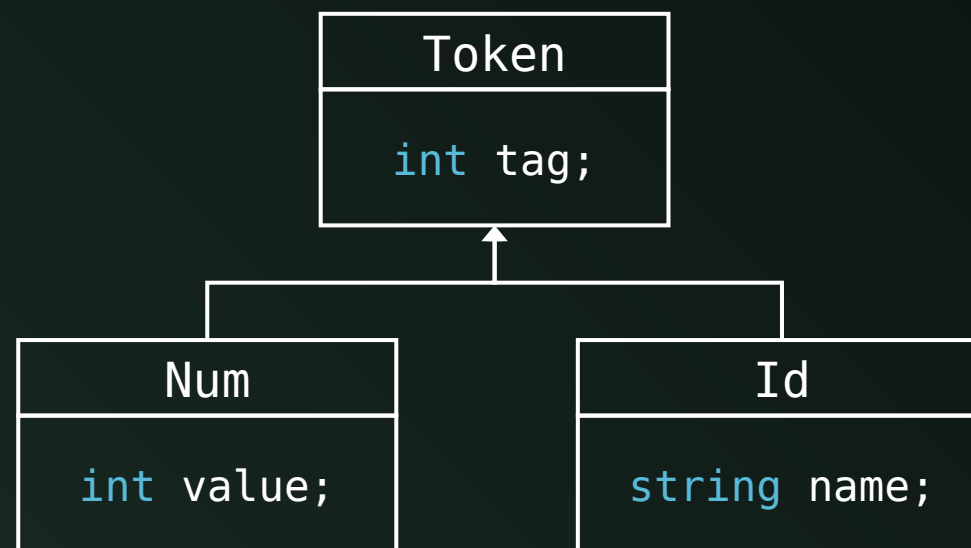
```
token scan()
{
    ignora os espaços em branco;
    trata os números;
    trata as palavras reservadas e identificadores;

    /* se chegar aqui trata o caractere lido como token */
    token t = token(peek);
    peek = espaço;
    return t;
}
```

Analizador Léxico

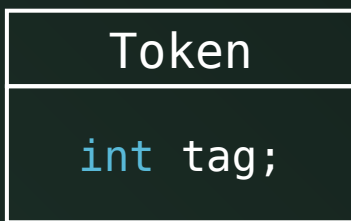
- A implementação do analisador léxico **representa um token** pelas seguintes classes:

- **Token**: operadores
Ex.: '+', '-', '*', '/'
- **Num**: números
Ex.: 25, 428, 0
- **Id**: identificadores e palavras-chave
Ex.: cont, val, if, while, true



Analizador Léxico

- A classe `Token` será usada para representar os símbolos que não necessitam de atributos, como os operadores aritméticos



O valor da tag será
o código ASCII do
caractere

```
struct Token
{
    int tag;
    Token(int t) : tag(t) {}
};
```

```
Token t = Token('+');
```

Analizador Léxico

- A tag das **classes Num e Id** serão constantes inteiras

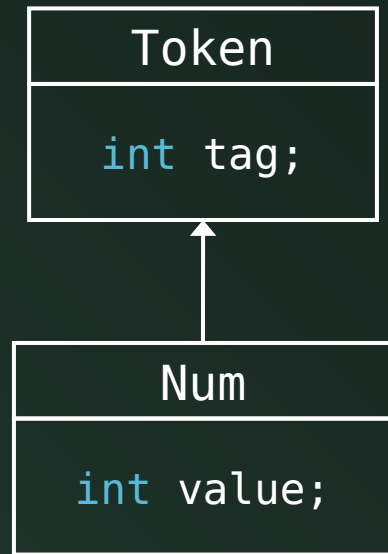
```
enum Tag { NUM = 256, ID = 257, TRUE = 258, FALSE = 259 };
```

- TRUE e FALSE estão ai para ilustrar o uso de palavras-chave reservadas
- Os caracteres ASCII tem código entre 0 e 255, por isso as tags para os demais tokens recebem valores a partir de 256
- Uso das tags:

```
Tag::NUM  
Tag::ID  
Tag::TRUE  
Tag::FALSE
```

Analizador Léxico

- A classe `Num` será usada para representar dígitos numéricos, e possui um atributo para guardar o seu valor inteiro

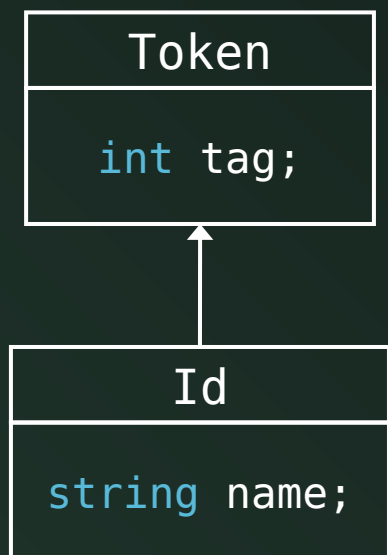


```
struct Num : public Token
{
    int value;
    Num(int v) : Token(Tag::NUM), value(v) {}
};

Num n = Num(21);
```

Analizador Léxico

- A classe `Id` será usada para representar identificadores e palavras-chave, e possui um atributo para guardar uma string



```
struct Id : public Token
{
    string name;
    Id(int t, string s) : Token(t), name(s) {}
};
```

```
Id var = Id(Tag::ID, "cont");
Id res = Id(Tag::TRUE, "true");
```

Resumo

- O analisador léxico precisa tratar os caracteres da entrada de forma a convertê-los em unidades mais simples para as próximas fases da compilação
- Constituem tarefas do analisador léxico:
 - Remover espaços em branco e comentários
 - Atualizar o número da linha sendo processada
 - Ler os caracteres da entrada e os agrupar em lexemas
 - Construir tokens para cada lexema