Aula 002 - Visão Geral da Análise Léxica

Especificação da Linguagem de Trabalho

Prof. Rogério Aparecido Gonçalves
rogerioag@utfpr.edu.br

Universidade Tecnológica Federal do Paraná (UTFPR)
Departamento de Computação (DACOM)
Campo Mourão - Paraná - Brasil

Ciência da Computação

BCC36B - Compiladores

UTFPR

Agenda

Introdução

Fases/Estruturas do Compilador

1. Código fonte $\rightarrow$ `[Sistema de varredura (análise léxica)]`
2. Marcas (*tokens*) $\rightarrow$ `[Analisador sintático]`
3. Árvore sintática $\rightarrow$ `[Analisador semântico]`
4. Árvore anotada $\rightarrow$ `[Otimizador de código-fonte]`
5. Código intermediário $\rightarrow$ `[Gerador de código]`
6. Código-alvo $\rightarrow$ `[Otimizador de código-alvo]`
7. Código-alvo $\rightarrow$ `[Executado]`

UTFPR

Fases/Estruturas do Compilador

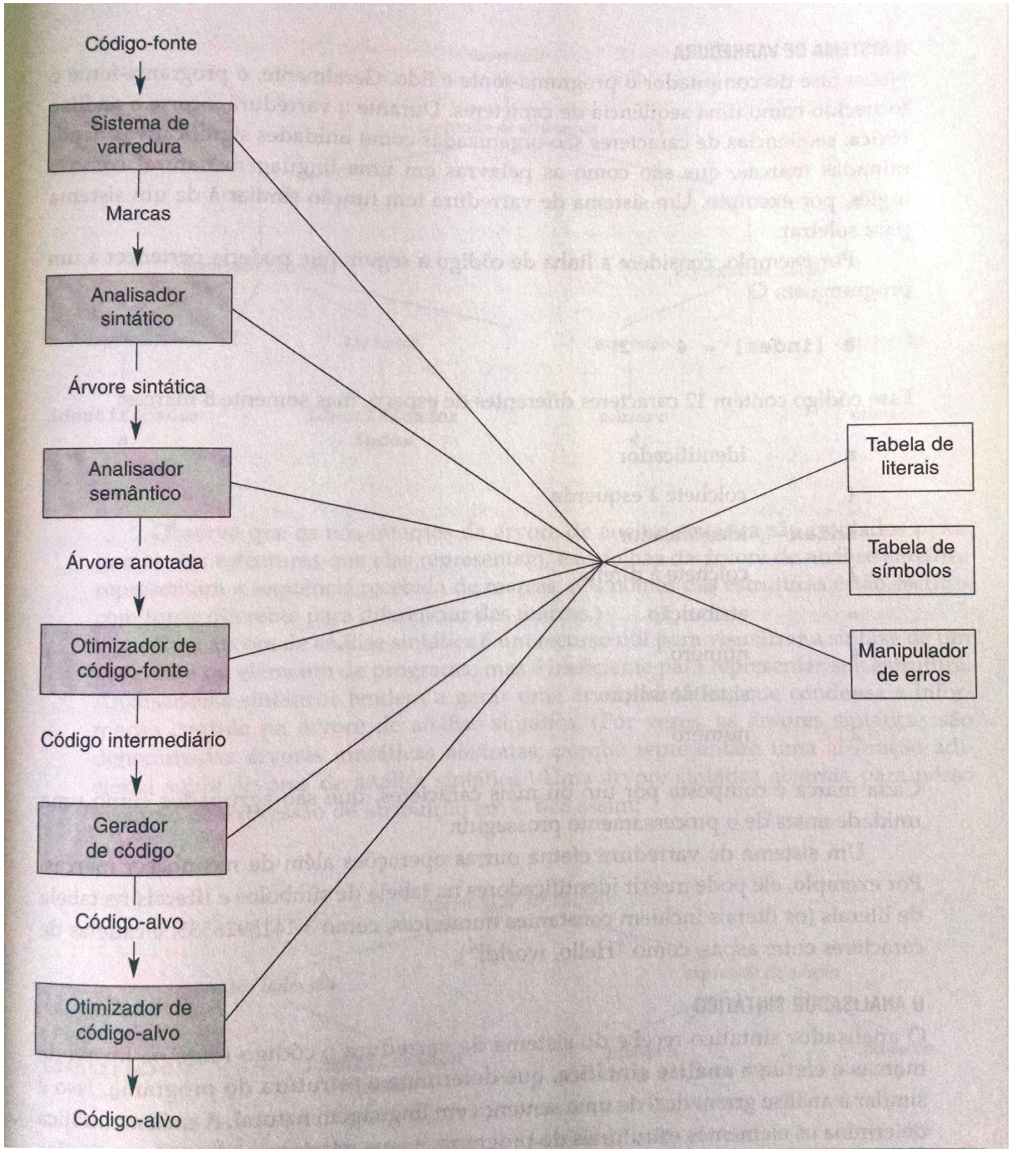

Figura 1: Estrutura/Fases do Compilador

Processo de Varredura

Introdução

- Sistema de Varredura $\equiv$ Analisar Léxico $\equiv$ Scanner
- **Análise léxica** é a fase do compilador que lê o programa-fonte como um arquivo de caracteres e o separa em um conjunto de marcas *(tokens)*
- Cada marca representa uma unidade:
  - `if`, `while`, `then`, etc (palavras-chave, palavras-reservadas)
- Ainda podemos reconhecer como marcas:
  - **identificadores**: geralmente composta por letras e números e *underscores*, mas começando com letra;
  - **símbolos especiais**: +, *, ==, <>, etc

Introdução

- Forma de reconhecimento das marcas: *identificação de padrões*
  - **expressões regulares**
  - **autômatos finitos**
- Podemos dividir este estudo em:
  - visão geral de um sistema de varredura e os respectivos conceitos e estruturas
  - estudo sobre expressões regulares
  - [máquinas de estados finitos (ou autômatos finitos)]

UTFPR

Processo de varredura

- O primeiro passo então é o reconhecimento das marcas
- O analisador léxico transforma o programa de uma sequência de caracteres sem nenhuma estrutura para uma sequência de marcas:

```
if x == y then
    z = 1;
else
    z = 2;
```

- Sequência de marcas:

```
|if| |x| |==| |y| |then|\n|\t|z| |=| |1|;|\n|else|\n|\t|z|
|=| |2|;|<EOF>
```

Tipo do marcador

- Em português:
  - substantivo, verbo, adjetivo...
- Em uma linguagem de programação:
  - identificador, numeral
  - palavras reservadas: `if`, `while`...
  - símbolos: `(`, `;`, `[`, ...

Primeiros passos da Análise Léxica

- Fornecer essas marcas (par `<tipo,substring>`) ao analisador sintático
  - `foo = 42` $\rightarrow$ Análise Léxica
  - `(<id,"foo">, <=,"=">, <num,"42">)` $\rightarrow$ Análise Sintática
- As marcas, como entidades lógicas, precisam ser claramente diferenciada das cadeias de caracteres que elas representam.
- A marca de palavra reservada **IF** precisa ser diferenciada da cadeia de caracteres *"if"* que a representa.
- A cadeia de caracteres representada por uma marca é denominada de **valor** ou **lexema**
- **Identificadores** são todos representados pela marca **ID**, mas podem ter muitos valores que representam seus nomes individuais.

Exercício

- Para o código abaixo, conte quantos tokens de cada tipo ele tem:

```
x = 0;\nwhile (x < 10) {\n\tx++;\n}
```

- Tipos: id, espaço, num, while, outros

Exercício

- Para o código abaixo, conte quantos tokens de cada tipo ele tem:

```
x = 0;\nwhile (x < 10) {\n\tx++;\n}
```

- Tipos: id (3), espaço (10), num (2), while (1), outros (9)

Ambiguidade

- A análise léxica de linguagens modernas é bem simples, mas historicamente esse não é o caso
- Em `Fortran`, espaços em branco dentro de uma marca também são ignorados:
  - `VAR1` e `VAR 1` são a mesma marca
  - (Em `Fortran`) `DO5I=1,25` são 7 marcas: 'DO', '5', 'I', '=', '1', ',', '25'
  - (Em outras linguagens) `DO5I=1.25` são 3 marcas: 'DO5I', '=', '1.25'

Ambiguidade

- As palavras-chave de PL/1 não são reservadas
  - `IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN`
- Mesmo em linguagens modernas existem ambiguidades léxicas
  - ==, ++, +=
  - Templates C++

Estruturas básicas de uma análise léxica:

- Definição das marcas (*hardcode*):

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

- Estrutura para armazenamento dos atributos de uma marca:

```
typedef struct {
    TokenType tokenval;
    char *stringval;
    ...
} TokenStruct;
```

Estruturas básicas de uma análise léxica:

- Definição das marcas (*hardcode*):

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

- Ou ainda utilizando uma **union**:

```
typedef struct {
    TokenType tokenval;
    union {
        char *stringval;
        ...
    } attribute;
} TokenStruct;
```

Função básica para a análise léxica:

```
TokenType getToken(void);
```

- Esta função retornará a próxima marca de um código fonte toda vez que for chamada;
- **Exercício**: Como deverá se comportar a função `getToken` para a seguinte sentença na linguagem de programação `C`:

```
a[index] = 4 + 2;
```

- Mostre todos os pares `marca:valor`

Sistema de Varredura (Analisador Léxico)

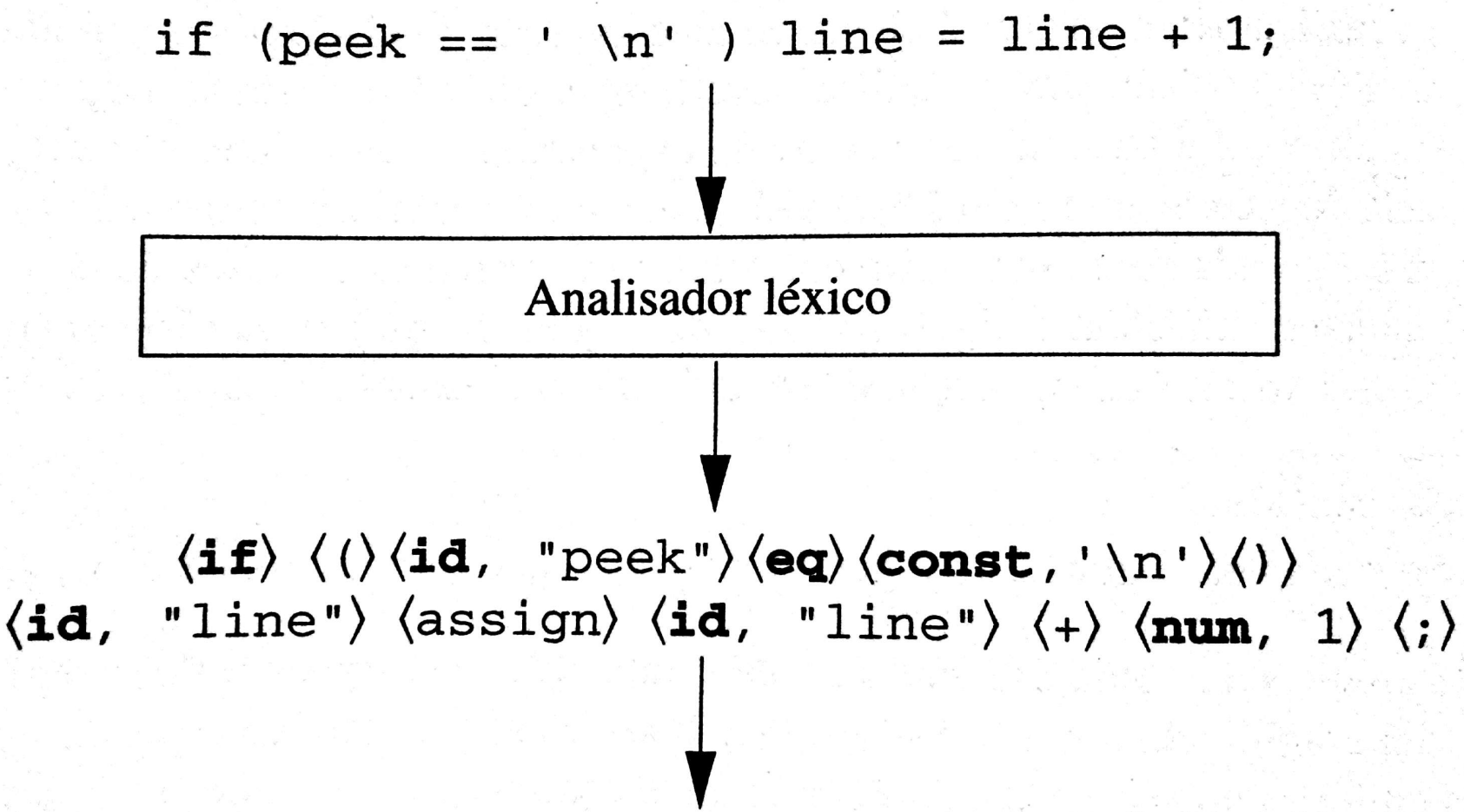

Figura 2: Tokens

Linguagem de Estudo nesta disciplina: T++

Esquemas principais da linguagem: T++

- Tipos básicos de dados suportado: **inteiro** e **flutuante**
- Suporte a arranjos uni e bidimensionais (**arrays**)
- Exemplos:
  - tipo: identificador[dim]
  - tipo: identificador[dim][dim]
- Variáveis locais e globais devem ter um dos tipos especificados
- Tipos de funções podem ser omitidos (quando omitidos viram um **procedimento** e um tipo **void** é devolvido explicitamente
- Linguagem quase fortemente tipificada: nem todos os erros são especificados mas sempre deve ocorrer avisos
- ...

UTFPR

Exemplos na linguagem T++

```
inteiro: n

inteiro fatorial(inteiro: n)
    inteiro: fat
    se n > 0 então {não calcula se n > 0}
        fat := 1
        repita
            fat := fat * n
            n := n - 1
        até n = 0
        retorna(fat) {retorna o valor do fatorial de n}
    senão
        retorna(0)
    fim
fim

inteiro principal()
    leia(n)
    escreva(fatorial(n))
    retorna(0)
fim
```

Exemplos na linguagem T++

```
flutuante: A[1024]
flutuante: B[1024]
flutuante: C[1024]

somaVetores(inteiro: n)
    inteiro: i
    i := 0
    repita
        C[i] := A[i] + B[i]
        i := i + 1
    até i = n
fim

inteiro principal()
    inteiro: i
    i := 0
    repita
        A[i] := 1
        B[i] := 1
        i := i + 1
    até i = 1024

    somaVetores(1024)
```

Exercício

- Escreva um código de algum algoritmo que você conheça na linguagem `T++`
- Esses códigos `.tpp` serão utilizados como um conjunto de testes para a *Análise Léxica*

Leitura Recomendada

Capítulo 2: Varredura

- LOUDEN, Kenneth C. Compiladores: princípios e práticas. São Paulo, SP: Thomson, c2004. xiv, 569 p. ISBN 8522104220.

Próximas Aulas

Próximas Aulas: Análise Léxica

- Expressões regulares
- Autômatos finitos
- Implementação da análise léxica
- Utilização de ferramentas