

# Construção de Compiladores

Prof. Dr. Daniel Lucrédio

DC - Departamento de Computação

UFSCar - Universidade Federal de São Carlos

**Tópico 07 - Análise Semântica**

# Referências bibliográficas

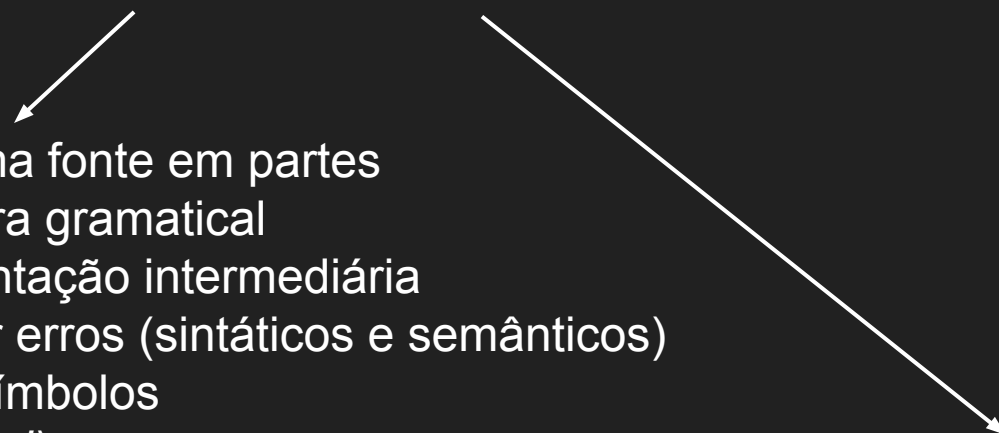
Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compiladores: Princípios, Técnicas e Ferramentas (2a. edição). Pearson, 2008.

Kenneth C. Louden. Compiladores: Princípios E Práticas (1a. edição). Cengage Learning, 2004.

Terence Parr. The Definitive Antlr 4 Reference (2a. edição). Pragmatic Bookshelf, 2013.

# Estrutura de um compilador

## Duas etapas: análise e síntese



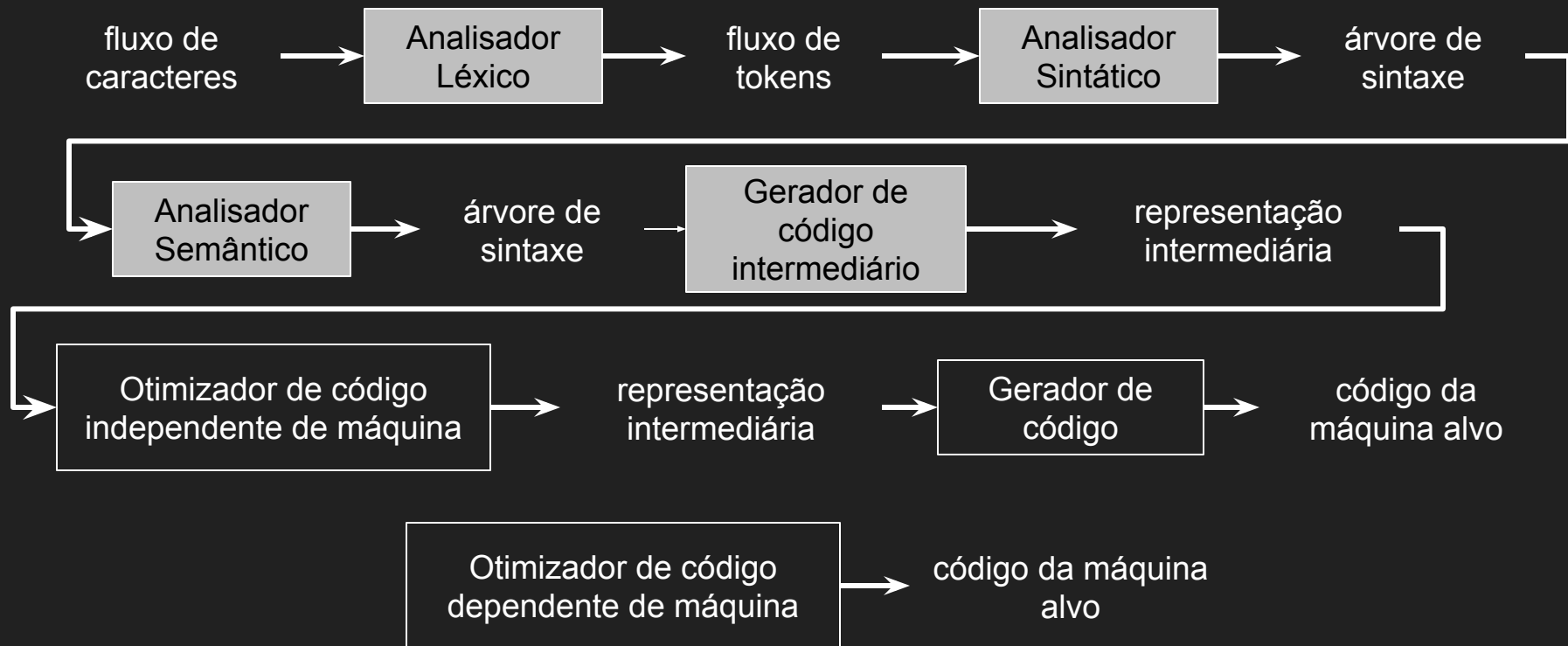
Quebrar o programa fonte em partes  
Impor uma estrutura gramatical  
Criar uma representação intermediária  
Detectar e reportar erros (sintáticos e semânticos)  
Criar a tabela de símbolos  
*(front-end)*

Construir o programa objeto  
com base na representação intermediária  
e na tabela de símbolos  
*(back-end)*

# Fases de um compilador

*front-end*

*back-end*



# Fases de um compilador

- **Análise léxica** (scanning)

- Lê o fluxo de caracteres e os agrupa em sequências significativas
  - Chamadas lexemas
- Para cada lexema, produz um token

<nome-token, valor-atributo>

- 
- Identifica o tipo do token
  - Símbolo abstrato, usado durante a análise sintática
  - Aponta para a tabela de símbolos (quando o token tem valor)
  - Necessária para análise semântica e geração de código

# Fases de um compilador

- **Análise sintática (parsing)**
  - Usa os tokens produzidos pelo analisador léxico
    - Somente o primeiro “componente”
    - (ou seja, **despreza os aspectos não-livres-de-contexto**)
  - Produz uma árvore de análise sintática
    - Representa a estrutura gramatical do fluxo de tokens
  - As fases seguintes utilizam a estrutura gramatical para **realizar outras análises** e gerar o programa objeto

# Fases de um compilador

- **Análise semântica**

- Checa a consistência com a definição da linguagem
- Coleta informações sobre tipos e armazena na árvore de sintaxe ou na tabela de símbolos
- Checagem de tipos / coerção (adequação dos tipos) são tarefas típicas dessa fase

É aqui que aparece a “sensibilidade ao contexto”


```
int main()
{
    int i, a[1000000000000000];
    float j@;

    i = "1";
    while (i<3
        printf("%d\n", i);
    k = i;
    return (0);
}
```



```
int main()
{
    int i, a[1000000000000000];
    float j@;

    i = "1";
    while (i<3
        printf("%d\n", i);
    k = i;
    return (0);
}
```



**Violação de  
significado:  
Erro semântico**

```
int main()
{
    int i, a[1000000000000000];
    float j@;

    i = "1";
    while (i<3
        printf("%d\n", i);
    k = i;
    return (0);
}
```

**Violação de  
identificadores  
conhecidos:  
Erro contextual  
("semântico")**

# Análise semântica guiada por sintaxe

- Definições Dirigidas pela Sintaxe (DDS)
  - Pouco usado na prática
- Esquemas de Tradução Dirigida pela Sintaxe (TDS)
  - Uso com geradores

Ações semânticas são inseridas na gramática (pedaços de código)

De forma que o analisador sintático (normalmente gerado) irá conter ações “extras”

Executadas DURANTE a análise sintática

# Esquemas de TDS no ANTLR

Demonstração 1

# Tabela de símbolos

- Captura a sensibilidade ao contexto
- Permite ao compilador “lembrar” detalhes associados aos nomes
  - Tipos de variáveis
  - Tipo de retorno de uma função
  - Argumentos de um procedimento
  - etc.
- Fundamental na detecção de erros semânticos
- Fundamental na geração de código

# Exemplo de Tabela de símbolos

Cadeia	Token	Categoria	Tipo	Valor	...
i	ident	var	inteiro	1	...
fat	ident	proc	-	-	...
2	num	-	inteiro	2	...
...					

- Exemplo de atributos para uma variável
- Tipo (inteira, real etc.), nome, endereço na memória, escopo (programa principal, função etc.) entre outros
- Para vetor, ainda seriam necessários atributos de tamanho do vetor, o valor de seus limites etc.

# Tabela de símbolos

## Principais operações

### 1. Inserir

- a. Armazena informações fornecidas pelas declarações

### 2. Verificar

- a. Recupera informação associada a um elemento declarado quando esse elemento é utilizado

### 3. Remover

- a. Remove (ou torna inacessível) a informação a respeito de um elemento declarado quando esse não é mais necessário

# Questões de projeto

- Como é frequentemente acessada, o acesso tem de ser **eficiente**
- Implementação
  - Estática
  - **Dinâmica**: melhor opção
- Estrutura de dados
  - Listas, matrizes
  - Árvores de busca (por exemplo, B e AVL)
  - Tabelas de espalhamento
- Acesso
  - Sequencial, busca binária, etc.
  - Hashing: opção **eficiente**
    - O elemento do programa é a chave e a função hash indica sua posição na tabela de símbolos



# Questões de projeto

- Tamanho da tabela

- Tipicamente, de algumas centenas a mil “linhas”
- Dependente da forma de implementação
- Na implementação dinâmica, não é necessário se preocupar tanto com isso

- Uma única tabela X várias tabelas

- Diferentes declarações têm diferentes informações e atributos
- Por exemplo, variáveis não têm número de argumentos, enquanto procedimentos têm
- Diferentes escopos

# Questões de projeto

- Escopo
  - Representação
    - Várias tabelas ou uma única tabela com a identificação do escopo para cada identificador
- Tratamento
  - Inserção de identificadores de mesmo nome, mas em níveis diferentes
  - Remoção de identificadores cujos escopos deixaram de existir
- Regras gerais
  - Declaração antes do uso
  - Aninhamento mais próximo

# Escopo

## Exemplo

```
program Ex;
var i,j: integer;

function f(tamanho: integer): integer;
var i,temp: char;

    procedure g;
    var j: real;
    begin
        ...
    end;

    procedure h;
    var j: ^char;
    begin
        ...
    end;

begin (* f *)
    ...
end;

begin (* programa principal *)
    ...
end.
```

# Escopo

## Exemplo

Variáveis locais  
e globais com  
mesmo nome

Subrotinas  
aninhadas

```
program Ex;  
var i,j: integer;
```

```
function f(tamanho: integer): integer;  
var i,temp: char;
```

```
    procedure g;  
    var j: real;  
    begin  
        ...  
    end;
```

```
    procedure h;  
    var j: ^char;  
    begin  
        ...  
    end;
```

```
begin (* f *)  
    ...  
end;
```

```
begin (* programa principal *)  
    ...  
end.
```

# Escopo

Declaração	Escopo
<code>int a = 1;</code>	B1 – B3
<code>int b = 1;</code>	B1 – B2
<code>int b = 2;</code>	B2 – B4
<code>int a = 3;</code>	B3
<code>int b = 4;</code>	B4

```
main() {  
    int a = 1; B1  
    int b = 1;  
    {  
        int b = 2; B2  
        {  
            int a = 3; B3  
            cout << a << b;  
        }  
        {  
            int b = 4; B4  
            cout << a << b;  
        }  
        cout << a << b;  
    }  
    cout << a << b;  
}
```

# Escopo

```
main() {  
    int a = 1; B1  
    int b = 1;  
    {  
        int b = 2; B2  
        {  
            int a = 3; B3  
            cout << a << b;  
        }  
        {  
            int b = 4; B4  
            cout << a << b;  
        }  
        cout << a << b;  
    }  
    cout << a << b;  
}
```

Vai imprimir:  
**32**

Vai imprimir:  
**14**

# Escopo x tabela de símbolos

- Operação **inserir**:
  - Não pode escrever por cima de declarações anteriores
  - Mas deve ocultá-las temporariamente
- Operação **verificar**:
  - Deve sempre acessar o escopo mais próximo (regra do aninhamento)
- Operação **remover**:
  - Deve remover apenas declarações no escopo mais próximo
  - Deve restaurar as declarações anteriormente ocultadas

# Que ações realizar?

- Operação **inserir**:
  - Verificar se o elemento já não consta na tabela
  - Inserir o elemento no escopo correto
- Operação **verificar**:
  - Realizada antes da inserção
  - Durante o uso de elementos na análise semântica
- Operação **remove**:
  - Torna inacessíveis dados que não são mais necessários (Ex.: após o escopo ter terminado)
  - Linguagens que permitem estruturação em blocos



# Exemplo

- Faremos um exemplo de análise semântica usando tabela de símbolos
- Iremos implementar as duas regras anteriores:
  - Declaração antes do uso
  - Aninhamento mais próximo

# Exemplo

- Teremos uma linguagem para cálculo de expressões aritméticas
  - Declarações de variáveis e expressões

■ Exs:

```
let x=2+1, y=3+4 in x+y
```

```
let x=2, y=3 in
```

```
  (let x=x+1, y=(let z=3, x=4 in x+y+z)
```

```
    in (x+y)
```

```
)
```

# Regras

Não pode haver redeclaração do mesmo nome dentro da mesma expressão

Ex: `let x=2, x=3 in x+1` (erro)

Se um nome não estiver declarado previamente em uma expressão (antes do `in`), ocorre erro

Ex: `let x=2 in x+y` (erro)

O escopo de cada declaração se estende pelo corpo segundo a regra do aninhamento mais próximo

Ex: `let x=2 in (let x=3 in x)` (valor da expressão=3)

# Regras

- A interação das declarações em uma lista no mesmo let é sequencial
  - Ou seja, cada declaração fica imediatamente disponível para a próxima da lista
  - **Ex:** `let x=2, y=x+1 in (let x=x+y, y=x+y in y)`

y=3

x=5

y=8

y=8

# Exercício

- Calcule o valor das seguintes expressões

```
let x=2+1, y=3+4 in x+y
```

Resp: 10

```
let x=2, y=3 in  
  (let x=x+1, y=(let z=3, x=4 in x+y+z)  
   in (x+y)  
  )
```

Resp: 13

## Demonstração 2

# Outras verificações

- Compatibilidade de tipos em comandos
  - Checagem de tipos é dependente do contexto
- **Atribuição**: normalmente, tem-se erro quando inteiro:=real
- Comandos de **repetição**: while booleano do, if booleano then
- Expressões e tipos esperados pelos **operadores**
  - Erro: inteiro+booleano

## Demonstração 3



# Outras verificações

- Concordância entre parâmetros formais e atuais, em termos de número, ordem e tipo
- Declaração: procedimento `p(var x: inteiro; var y: real)`
  - `p(10,2.3)` - OK
  - `p(10,2)` - OK
  - `p(2.3,10)` - Erro
  - `p(10)` - Erro

# Considerações finais

- Devido às variações de especificação semântica das linguagens de programação, a análise semântica **não é tão bem formalizada**
  - Não existe um método ou modelo padrão de representação do conhecimento (como BNF)
  - Não há uniformidade na quantidade e nos tipos de análise semântica entre linguagens
  - Não existe um mapeamento claro da representação para o algoritmo correspondente

**Análise é artesanal, dependente da linguagem de programação**

Fim