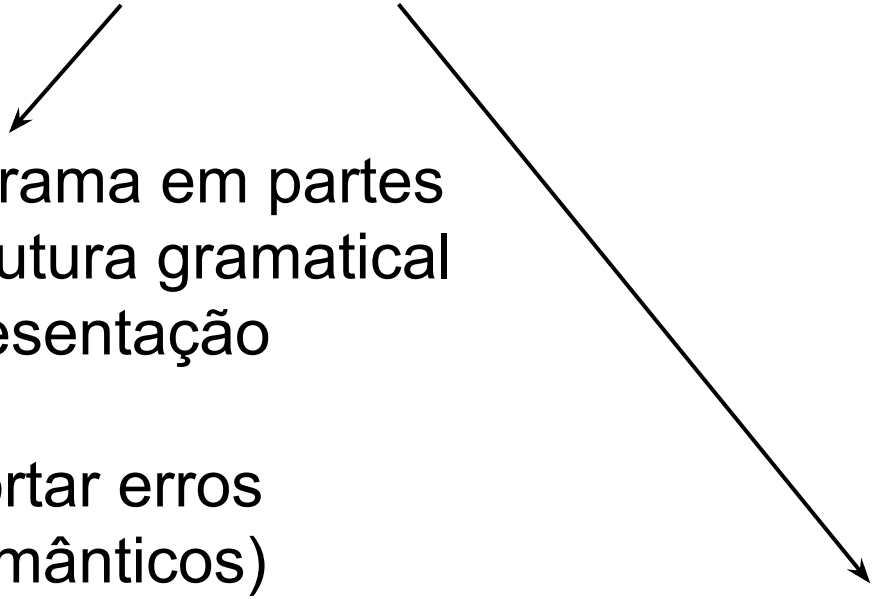
Construção de compiladores

Profs. Mário César San Felice (e Helena Caseli,
Murilo Naldi, Daniel Lucrédio)
Departamento de Computação - UFSCar
1º semestre / 2018
Tópico 7 - Análise Semântica

Estrutura de um compilador

- Duas partes: análise e síntese



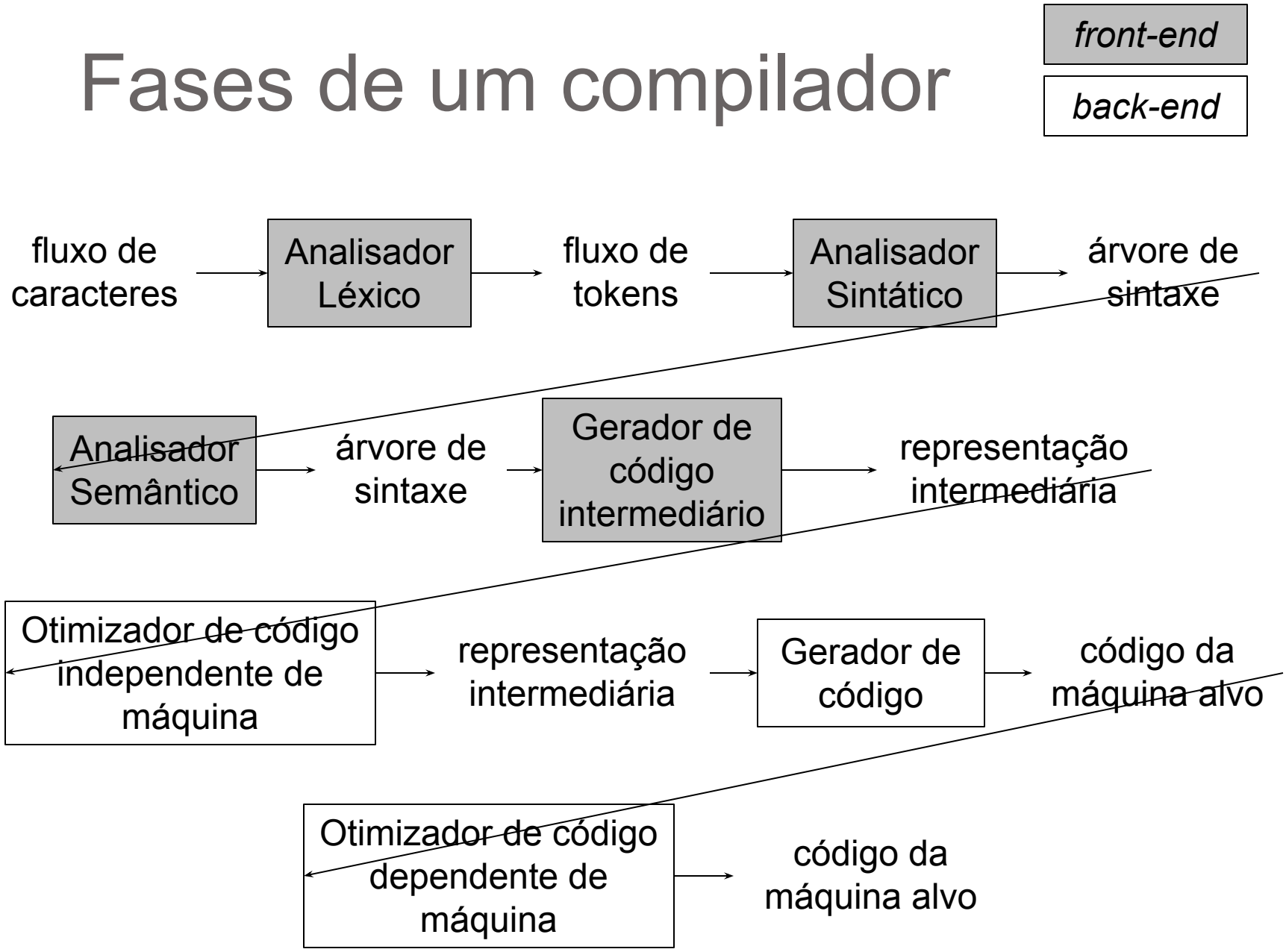
- Quebrar o programa em partes
- Impor uma estrutura gramatical
- Criar uma representação intermediária
- Detectar e reportar erros (sintáticos e semânticos)
- Criar a tabela de símbolos

(front-end)

(back-end)

- Construir o programa objeto, com base:
  - na representação intermediária
  - e na tabela de símbolos

Fases de um compilador



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 dos 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)

- É aqui que aparece a “sensibilidade ao contexto”

Manipulação de erros

```
int main()
{
  int i, a[1000000000000];
  float j@;

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

Manipulação de erros

```
int main()
{
  int i, a[1000000000000];
  float j@;

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

Violação de significado: Erro semântico

Manipulação de erros

```
int main()
{
  int i, a[1000000000000];
  float j@;

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

Violação de identificadores conhecidos: Erro contextual (“semântico”)

Antes

- Vamos fazer uma breve demonstração de um analisador semântico feito “à mão”

- Demonstração

Problemas

- E se eu precisar de outras análises semânticas?
  - Exs:
    - Detectar métodos com “return” faltando
    - Detectar código inalcançável
    - Considerar diferentes escopos
    - Etc...
- A implementação fica complicada

Problemas

- Além disso
  - Normalmente utilizamos geradores de analisadores
    - Yacc / ANTLR
  - Não temos controle direto sobre os procedimentos
    - Normalmente trabalhamos com a gramática
    - Em analisadores bottom-up é ainda pior o acesso ao código!!

Análise semântica guiada por sintaxe

- Surge a necessidade de um formalismo
  - Que nos permite expressar a análise semântica de forma acoplada à sintaxe
  - Assim como a (E)BNF permite gerar código de análise sintática
  - Esse formalismo permitiria gerar código de análise semântica

Análise semântica guiada por sintaxe

- No entanto, a análise semântica é muito diversificada
  - Temos que fazer coisas como:
    - Checar fluxo de controle em busca de código inalcançável
    - Calcular tipos de expressões (ex: ½ = real)
    - Verificar se variáveis foram declaradas ou não, seu escopo, etc...

- Em geral, a semântica de uma linguagem de programação não é formalmente especificada
  - O projetista do compilador tem que analisar a linguagem e extrair a semântica

Análise semântica guiada por sintaxe

- Não existe um modelo que cobre todos os casos
  - Assim, análise semântica é normalmente feita através de código comum
    - Ou seja, código que faz o que o projetista quiser
  - Porém, ainda é necessário algum controle
    - Considerando-se as principais ações semânticas
    - Principais tarefas feitas durante a análise semântica

Análise semântica guiada por sintaxe

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

- Conteúdo semântico é inserido na gramática
  - De forma que o analisador sintático (normalmente gerado) irá conter ações “extras”
  - Essas ações farão as verificações semânticas
    - Checagem de tipos
    - Declaração de variáveis, etc

Esquemas de Tradução Dirigida pela Sintaxe

Esquemas de TDS

- Um esquema de TDS é uma gramática livre de contexto com fragmentos de programa embutidos nos corpos das produções
  - Em qualquer lugar

- Vantagens: podem ser utilizados diretamente em geradores de analisadores
  - Yacc
  - ANTLR

Esquemas de TDS

Demonstração

Tabela de Símbolos

Tabela de símbolos

- Estrutura central na compilação
- Relacionada a todas as etapas da compilação
  - Mas é na análise semântica que melhor se ajusta
    - Captura a sensitividade ao contexto e as ações executadas no decorrer do programa
  - Fundamental na geração de código

Tabela de símbolos

- Permite saber, durante a compilação de um programa:
  - Tipo, valor, escopo de seus elementos (números e identificadores)

- Pode ser utilizada para armazenar as **palavras reservadas** e símbolos especiais da linguagem

Exemplo de Tabela de símbolos

- Cada token tem atributos/informações diferentes

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
  - **Inserir**
    - Armazena informações fornecidas pelas declarações
  - **Verificar**
    - Recupera informação associada a um elemento declarado quando esse elemento é utilizado
  - **Remover**
    - Remove (ou torna inacessível) a informação a respeito de um elemento declarado quando esse não é mais necessário

Tabela de símbolos

- Quando é acessada pelo compilador
  - Sempre que um elemento é mencionado no programa

- Principais objetivos do acesso
  - Verificar ou incluir sua declaração
  - Verificar seu tipo, escopo ou alguma outra informação
  - Atualizar alguma informação associada ao identificador (por exemplo, valor)
  - Remover um elemento quando este não se faz mais necessário ao programa

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 mais 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

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
      - Permite uma única passada
    - Aninhamento mais próximo para estrutura de blocos

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

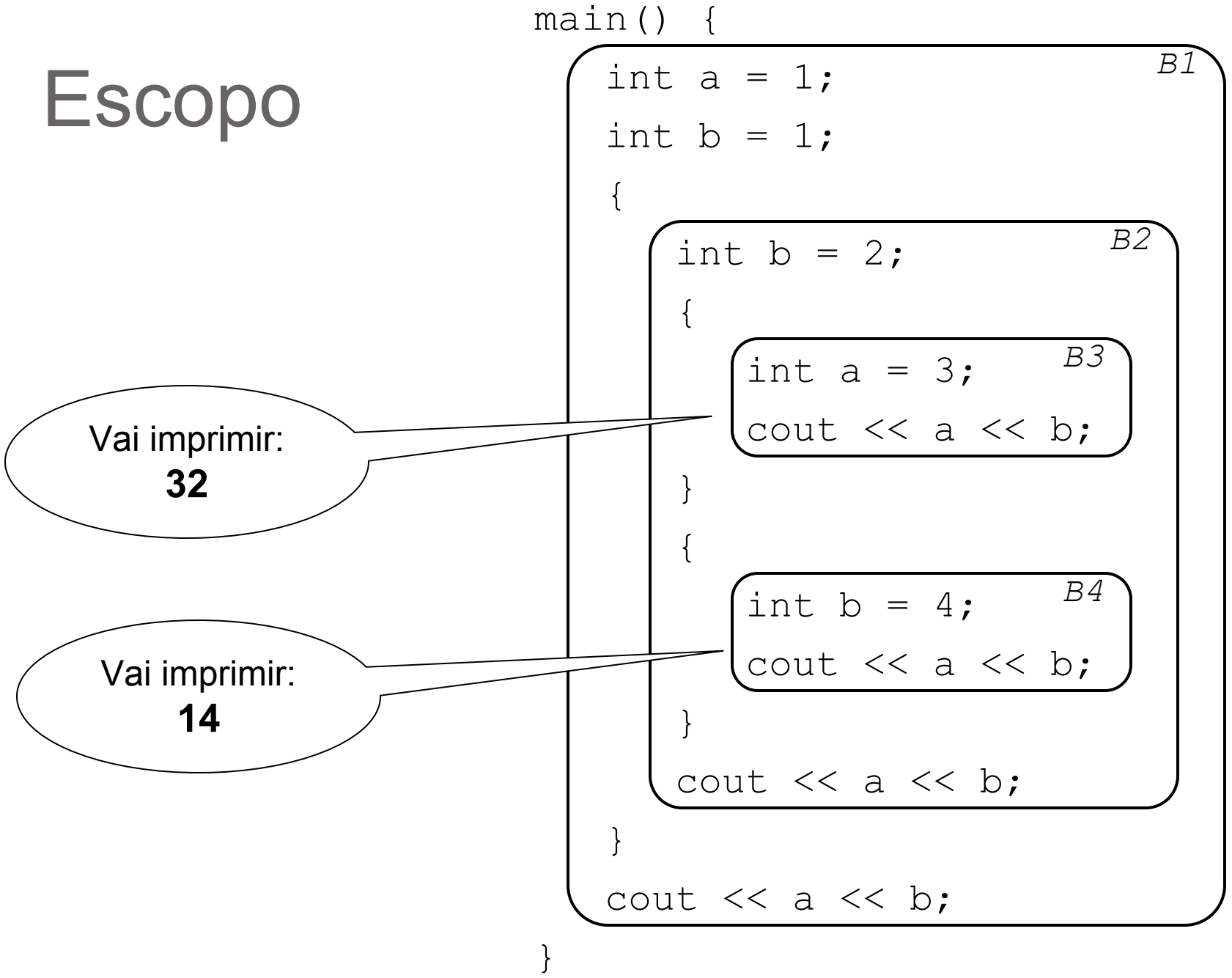
```
program Ex;
var i,j: integer;

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

Escopo x tabela de símbolos

- Duas opções principais para lidar com essa situação

1. Uma única tabela de espalhamento
   - Cada entrada é uma lista
   - Elementos encontrados antes nessa lista são aqueles que estão “valendo” num determinado momento

2. Uma lista de tabelas
   - A tabela no início representa o escopo mais próximo

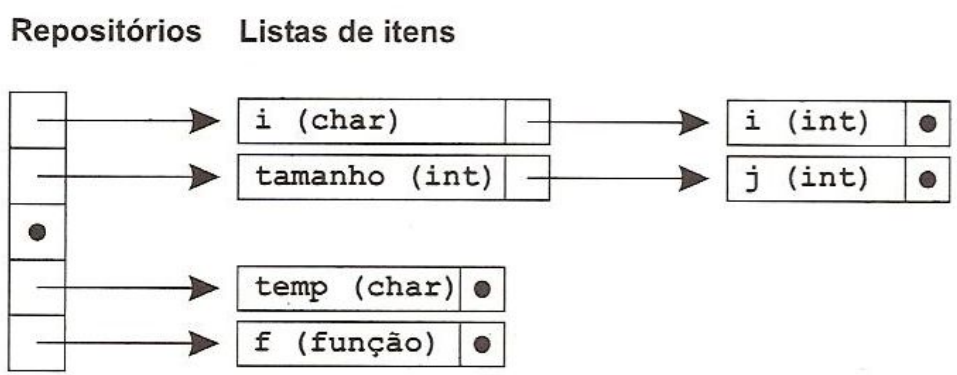
Escopo

- Exemplo

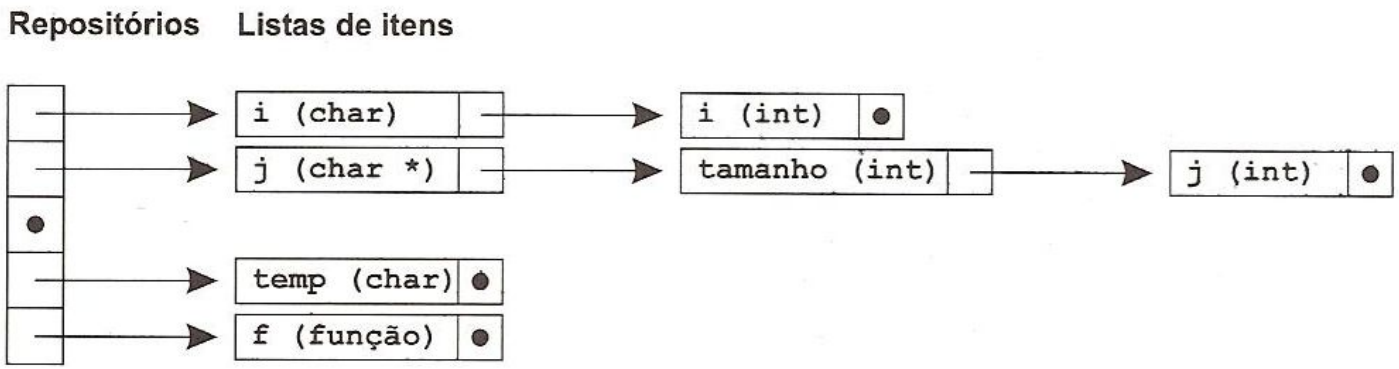
```
program Ex;
var i,j: integer;

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 x tabela de símbolos

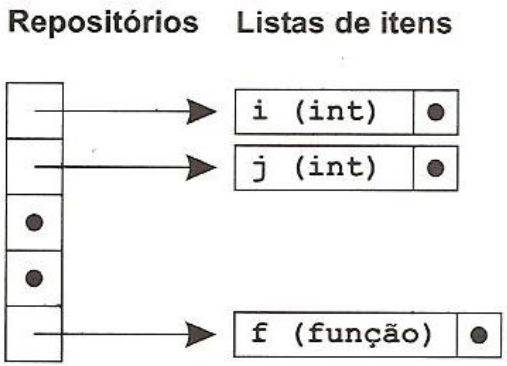
- Opção 1



(a) Após o processamento das declarações do corpo de f



(b) Após o processamento da declaração da segunda declaração composta aninhada dentro do corpo de f



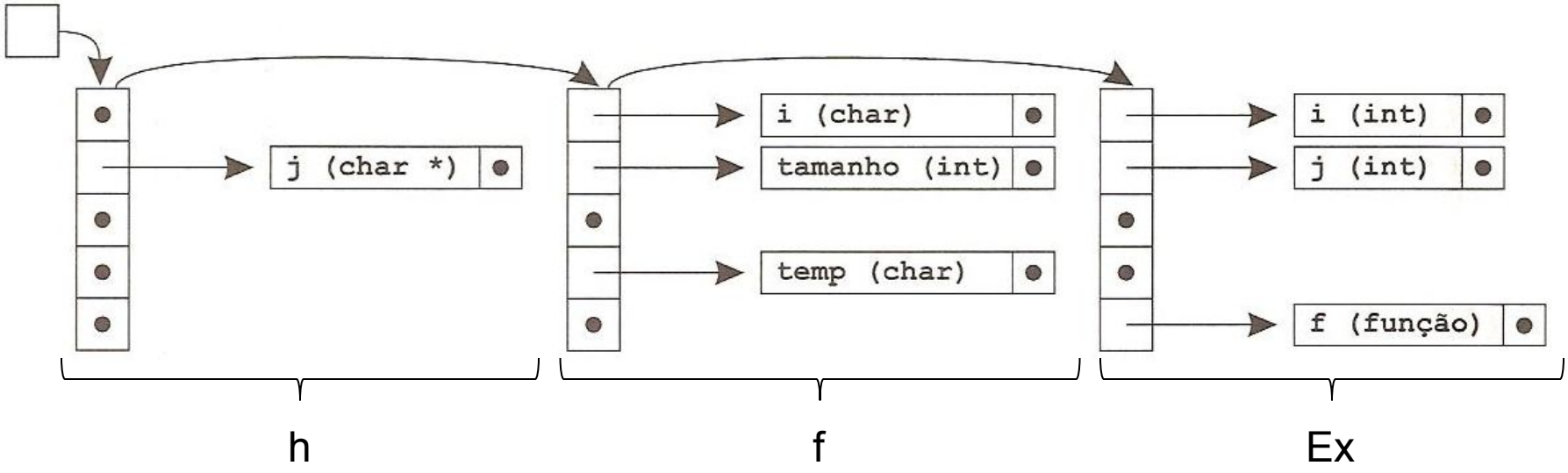
(c) Após abandonar o corpo de f (e apagar suas declarações)

Escopo x tabela de símbolos

- Na opção 1:
  - Função inserir modifica a lista em uma entrada específica, inserindo uma nova declaração no início
  - Função verificar percorre a lista de uma entrada
  - Função remover elimina o elemento de uma lista

Escopo x tabela de símbolos

- Opção 2



Escopo x tabela de símbolos

- Na opção 2:
  - Funções inserir/remover trabalham normalmente na tabela “atual” (no início da lista)
  - Função verificar varre as tabelas na lista, em busca de uma declaração válida
  - Para abandonar um escopo, basta eliminar toda a tabela no início da lista
    - Na opção 1 é necessário varrer as entradas em busca das declarações do escopo sendo abandonado

Escopo x tabela de símbolos

- Pode ser interessante armazenar o nome do escopo, para permitir o acesso identificado a ele
  - Exemplos: `Ex.f.g.j`, `Ex.f.h.j`, `Ex.j`
- Pode ser também necessário armazenar o nível ou profundidade de aninhamento de cada escopo
  - Para verificações semânticas como declarações de duas variáveis numa mesma profundidade
    - Só é necessário no caso da opção 1

Implementação

- As sub-rotinas de inserção, busca e remoção podem ser inseridas diretamente

- Associando-se regras semânticas às regras gramaticais

Implementação

- Inserção de elementos na tabela
  - Verificar se o elemento já não consta na tabela
  - Inserir o elemento no escopo correto

- Busca de informação na tabela
  - Realizada antes da inserção
  - Durante o uso de elementos na análise semântica

- Remoção de elementos da tabela
  - 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

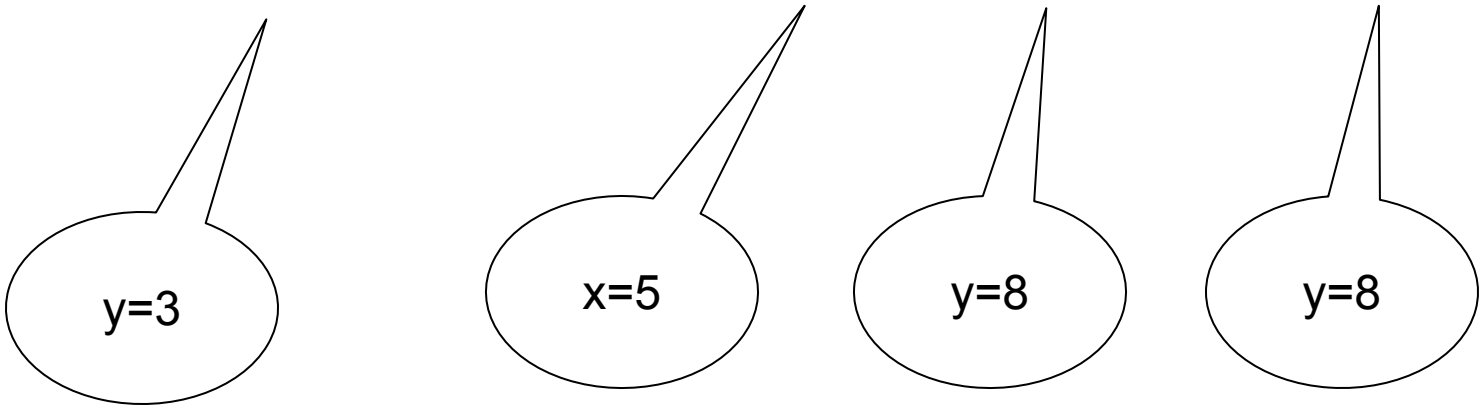
1. Não pode haver redeclaração do mesmo nome dentro da mesma expressão
   - Ex: `let x=2,x=3 in x+1` (erro)

2. Se um nome não estiver declarado previamente em uma expressão (antes do in), ocorre erro
   - Ex: `let x=2 in x+y` (erro)

3. 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)`
   - A expressão acima tem valor 3, e não 2

Regras

- Regras
  - Finalmente, 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)`



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

Outras verificações

- Verificação do uso adequado dos elementos do programa
  - Vimos a declaração de identificadores
    - Erro: identificador não declarado ou declarado duas vezes
    - Verificado durante a construção da tabela de símbolos
    - Tratamento de escopo
  - Mas existem outras verificações comuns

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

Outras verificações

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

Verificação de tipos

- Expressão de tipo
  - Tipos básicos
    - Booleano, caractere, real, etc.
  - Formada por meio da aplicação de um construtor de tipos a outras expressões de tipo
    - Construtor de tipos: arrays, registros, ponteiros, funções etc.

- Sistema de tipos
  - Coleção de regras para as expressões de tipos

Verificação de tipos

- Verificador de tipos
  - Implementa um sistema de tipos, utilizando
    - Informações sobre a sintaxe da linguagem
    - A noção de tipos
    - As regras de compatibilidade de tipos

- Equivalência de expressões de tipo
  - `function tipoIgual (t1, t2: TipoExp): booleano;`
  - Retorna verdadeiro se t1 e t2 representam o mesmo tipo segundo as regras de equivalência de tipos da linguagem

Verificação de tipos

- 2 tipos principais de equivalências
  - **Equivalência de nomes** – tipos compatíveis se
    - Têm o mesmo nome do tipo, definido pelo usuário ou primitivo
    - Ou aparecem na mesma declaração
  - **Equivalência estrutural** – tipos compatíveis se
    - Possuem a mesma estrutura (p. ex. representada por árvores sintáticas)
    - Única disponível na ausência de nomes de tipos
- A maioria das linguagens implementa as duas estratégias de compatibilidade de tipos

Exemplo de Equivalência

- Para as declarações abaixo

```
type t = array[1..20] of integer;
var a, b: array[1..20] of integer;
c: array[1..20] of integer;
d: t;
e, f: record
        a: integer;
        b: t
End
```

- Pode-se observar que...

Exemplo de Equivalência

- Para as declarações abaixo

```
type t = array[1..20] of integer;
var a, b: array[1..20] of integer;
c: array[1..20] of integer;
d: t;
e, f: record
        a: integer;
        b: t
End
```

- Pode-se observar que (**a** e **b**), (**e** e **f**) e (**d**, **e.b** e **f.b**) têm equivalência de nomes, enquanto **a**, **b**, **c**, **d**, **e.b** e **f.b** têm tipos compatíveis estruturalmente

Verificação de tipos

- Pontos importantes
  - Polimorfismo – construções com mais de um tipo
    - Uma função que troca o valor de duas variáveis de tipos iguais independentemente de quais tipos são
    - Uma função que conta os elementos de uma lista sem levar em consideração os tipos dos elementos da mesma

Verificação de tipos

- Pontos importantes
  - Sobrecarga – diversas declarações separadas que se aplicam a um mesmo nome
    - Mesmo operador, significados distintos dependendo do contexto
      - Ex.: + soma e + concatenação
  - Amarração estática X dinâmica
    - Estática: declaração explícita do tipo, boa para compilação
    - Dinâmica: tipo inferido na execução, boa para interpretação

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