Construção de compiladores

Prof. Daniel Lucrédio
Departamento de Computação - UFSCar
1º semestre / 2015
Aula 8

Análise semântica

- Tabela de símbolos
  - Estrutura principal da compilação
  - Está 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
- 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

Tabela de símbolos

- Exemplo
  - Cada token tem atributos/informações diferentes associadas

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 no programa quando esse elemento é utilizado
  - **Remover**
    - Remove (ou torna inacessível) a informação a respeito de um elemento declarado quando esse não é mais necessário
- O comportamento da tabela de símbolos depende fortemente das propriedades da linguagem sendo compilada

Tabela de símbolos

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

Tabela de símbolos

- 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)
  - 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
      - Necessidade de tratamento de colisões

Tabela de símbolos

- Exemplo de *hashing* com resolução de colisões para a inclusão dos identificadores **i**, **j**, **tamanho** e **temp**

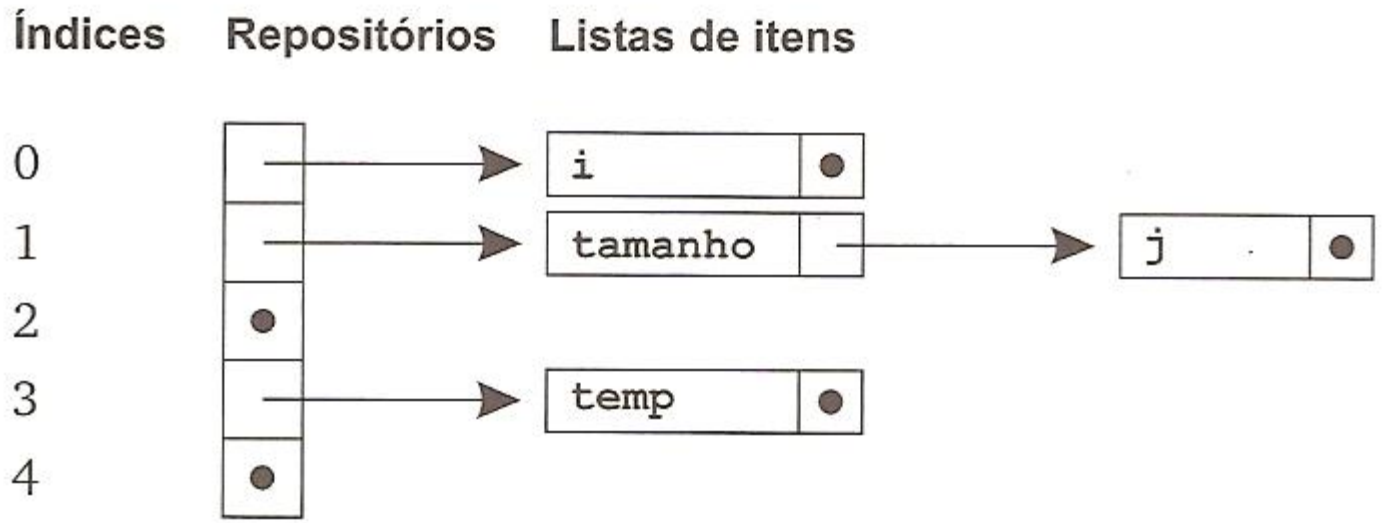


Tabela *hashing* com encadeamento separado

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 (como um atributo ou por meio de listas ligadas, por exemplo) 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

Váriáveis locais e globais com mesmo nome

Subrotinas aninhadas

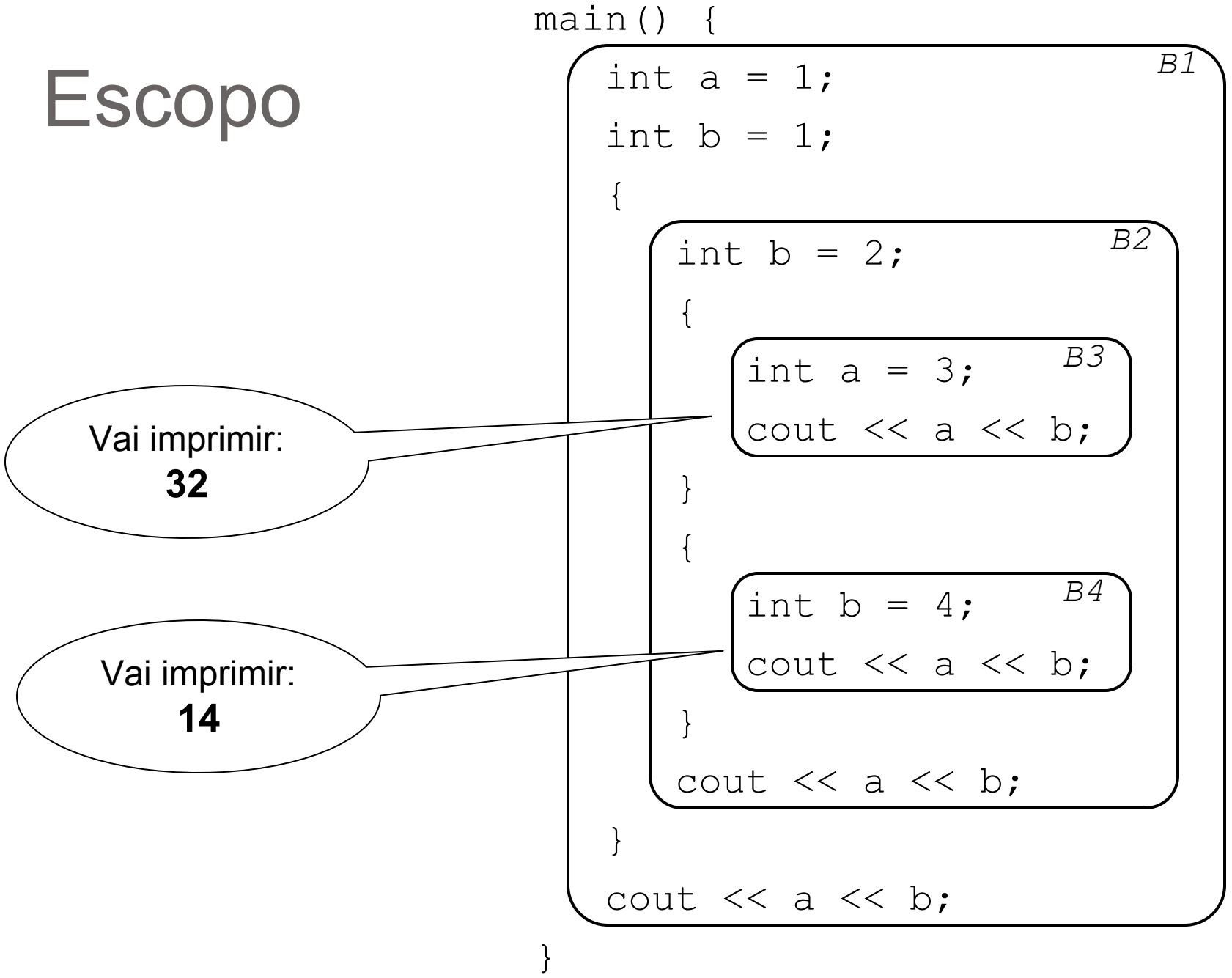
```
program Ex;
var i,j: integer;

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

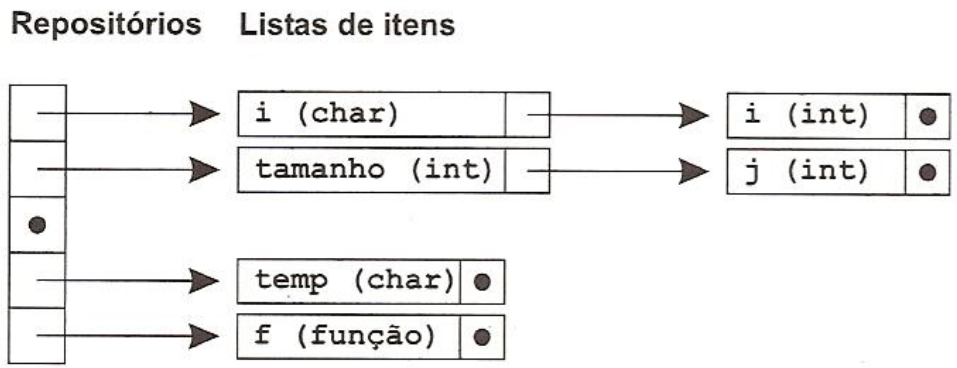
- Operação inserir:
  - Não pode escrever por cima de declarações anteriores
  - Mas deve ocultá-las temporariamente
- Operação verificar:
  - Deve sempre acessar o escopo mais próximo (regra do aninhamento)
- Operação remover:
  - Deve remover apenas declarações no escopo mais próximo
  - Deve restaurar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

Escopo x tabela de símbolos

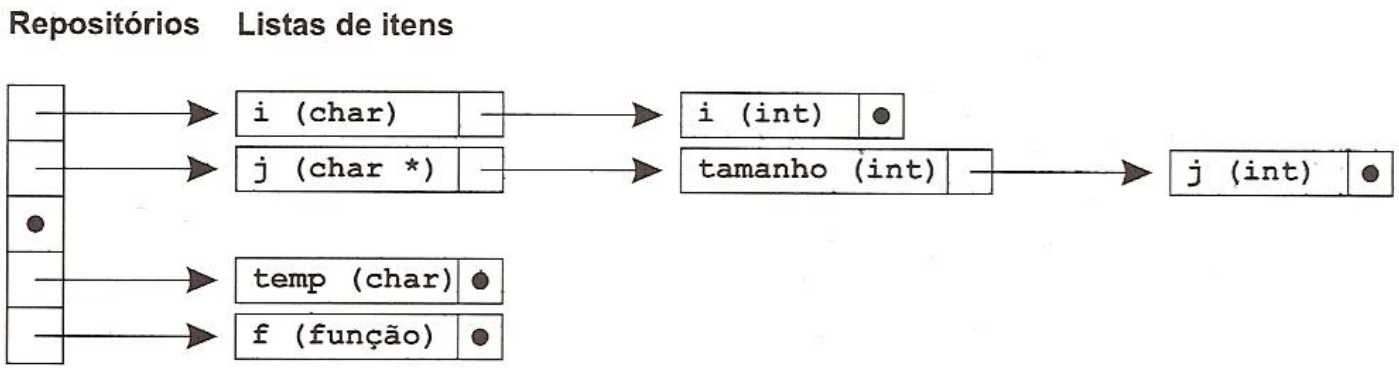
- Duas opções principais para lidar com essa situação
  1. Uma única tabela, cada entrada é uma pilha, elementos no topo dessa pilha são aqueles que estão “valendo” num determinado momento
  2. Uma pilha de tabelas, a tabela no topo da pilha representa o escopo mais próximo

Escopo x tabela de símbolos

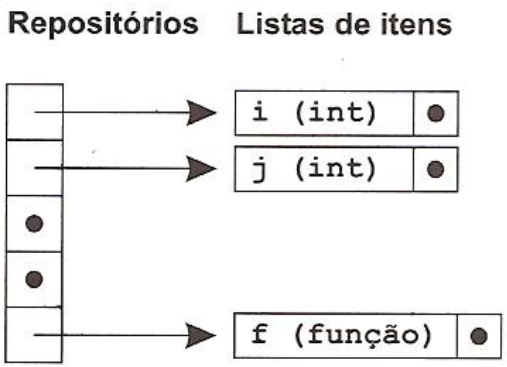
- Opção 1



(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



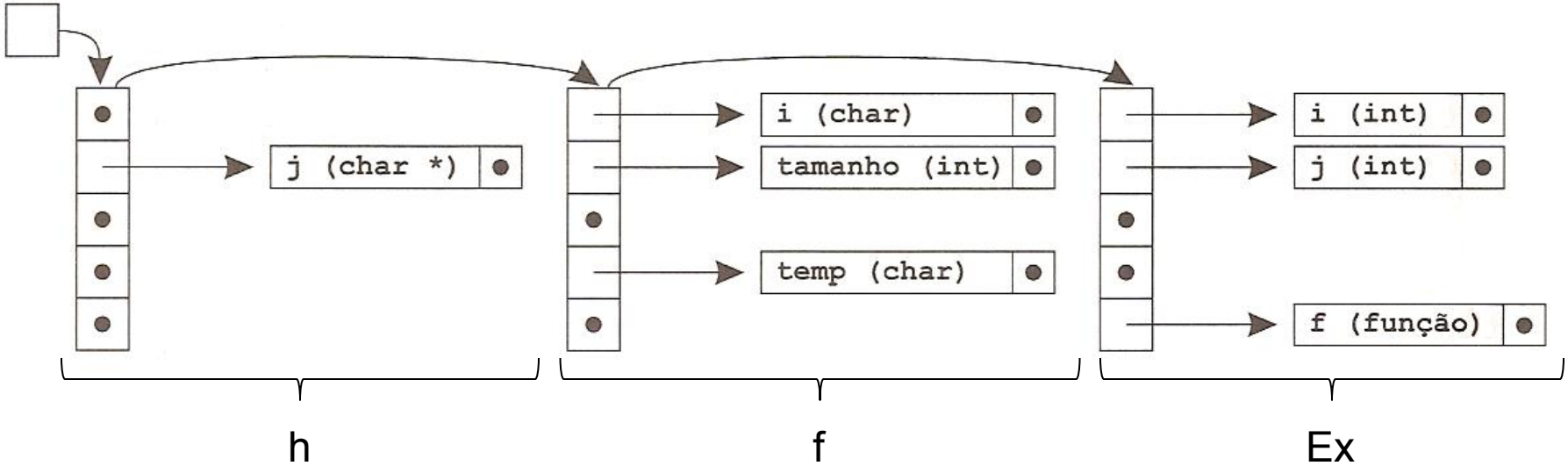
(c) Após abandonar o corpo de f (e apagar suas declarações)

Escopo x tabela de símbolos

- Na opção 1:
  - Função inserir modifica a pilha em uma entrada específica, inserindo uma nova declaração no topo
  - Função verificar apenas olha o topo das pilhas, ignorando o resto
  - Função remover elimina o topo de uma pilha, transformando a declaração imediatamente abaixo no novo topo

Escopo x tabela de símbolos

- Opção 2



Escopo x tabela de símbolos

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

Escopo x tabela de símbolos

- Outras considerações
  - Pode ser interessante armazenar o nome do escopo, para permitir o acesso identificado a determinado escopo
    - 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 outras verificações semânticas, como por exemplo declarações de duas variáveis em um mesmo nível de profundidade
      - Obviamente, só faz sentido utilizando a opção 1 descrita anteriormente
  - Essas regras valem para escopo estático apenas (mais comum)
    - Existe também o escopo dinâmico, que é pouco utilizado

Implementação

- As sub-rotinas de inserção, busca e remoção podem ser inseridas diretamente em uma DDS
  - Associando-se regras semânticas às regras gramaticais
- Principais operações
  - Inserção de elementos na tabela
    - Verificar se o elemento já não consta na tabela
    - Inserir o elemento considerando escopo correto
  - Busca de informação na tabela
    - Realizada antes da inserção
    - Busca informações para análise semântica durante o uso de elementos
  - Remoção de elementos da tabela
    - Torna inacessíveis dados que não são mais necessários (por exemplo, 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
```

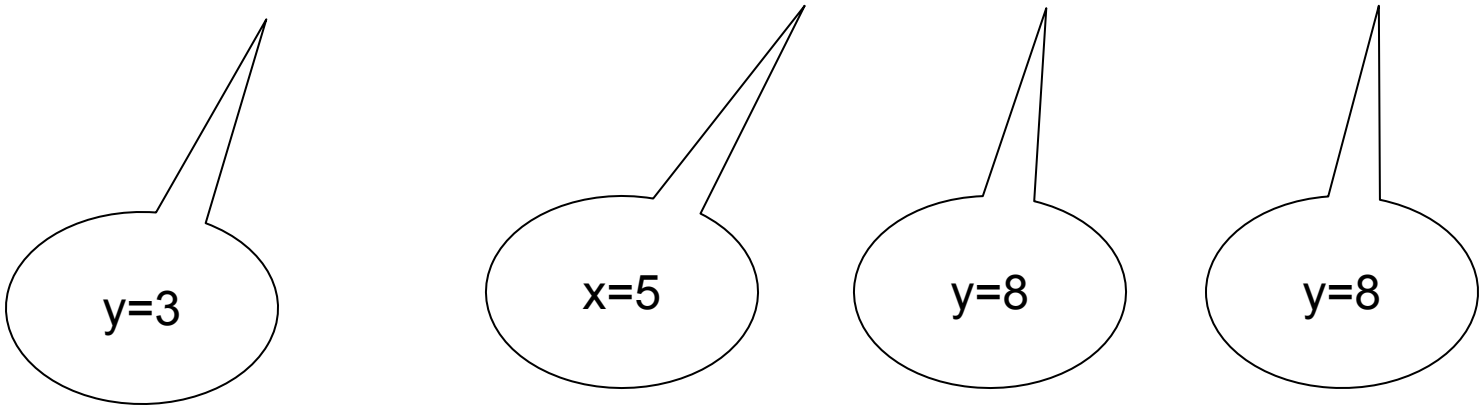
```
let x=2, y=3 in
    (let x=x+1, y=(let z=3, x=4 in x+y+z)
     in (x+y)
    )
```

Exemplo

- Regras:
  - Primeiro, não pode haver redeclaração do mesmo nome dentro da mesma expressão
    - Ex: `let x=2,x=3 in x+1` (erro)
  - Segundo, se um nome não estiver declarado previamente em uma expressão (antes do in), ocorre erro
    - Ex: `let x=2 in x+y` (erro)
  - Terceiro, 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

Exemplo

- 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

Exemplo

- Gramática livre de contexto

```
programa → exp
exp → '(' exp ')' | exp '+' exp | ID
    | INT | 'let' listaDecl 'in' exp
listaDecl → listaDecl ',' decl | decl
decl → id '=' exp
```

Exemplo

- Para implementar no ANTLR (que é LL)

```
programa → exp
exp → ( '(' exp ')'
      | ID
      | INT
      | 'let' listaDecl 'in' exp
      )
      ('+' exp)*
listaDecl → decl (',' decl)*
decl → ID '=' exp
```

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

- Verificação do uso adequado dos elementos do programa
  - 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

Implementação

- Verificação do uso adequado dos elementos do programa
  - 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
  - Tipo básico
    - 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
- Verificador de tipos
  - Implementa um sistema de tipos, utilizando informações sobre a sintaxe da linguagem, a noção de tipos e as regras de compatibilidade de tipos

Verificação de tipos

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

Verificação de tipos

- Exemplo
  - 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
  - **a**, **b**, **c**, **d**, **e.b** e **f.b** têm tipos compatíveis estruturalmente

Verificação de tipos

- Exemplo de uma DDS para verificação de tipos

Regra gramatical	Regras semânticas
var-decl → **id** : *tipo-exp*	*inserir*(**id**.*nome*, *tipo-exp.tipo*)
tipo-exp → **int**	*tipo-exp.tipo* := *inteiro*
tipo-exp → **bool**	*tipo-exp.tipo* := *booleano*
$tipo\text{-}exp_1$ → **array** [**num**] **of** $tipo\text{-}exp_2$	$tipo\text{-}exp_1.tipo$:= *criaTipoNó*(*matriz*, **num**.*tamanho*, $tipo\text{-}exp_2.tipo$)
decl → **if** *exp* **then** *decl*	**if not** *tipoIgual*(*exp.tipo*, *booleano*) **then** *tipo-erro*(*decl*)
decl → **id** := *exp*	**if not** *tipoIgual*(*verificar*(**id**.*nome*), *exp.tipo*) **then** *tipo-erro*(*decl*)
exp_1 → exp_2 + exp_3	**if not** (*tipoIgual*($exp_2.tipo$, *inteiro*) **and** *tipoIgual*($exp_3.tipo$, *inteiro*)) **then** *tipo-erro*(exp_1) ; $exp_1.tipo$:= *inteiro*
exp_1 → exp_2 **or** exp_3	**if not** (*tipoIgual*($exp_2.tipo$, *booleano*) **and** *tipoIgual*($exp_3.tipo$, *booleano*)) **then** *tipo-erro*(exp_1) ; $exp_1.tipo$:= *booleano*
exp_1 → exp_2 [exp_3]	**if** *éTipoMatriz*($exp_2.tipo$) **and** *tipoIgual*($exp_3.tipo$, *inteiro*) **then** $exp_1.tipo$:= $exp_2.tipo.filho1$ **else** *tipo-erro*(exp_1)
exp → **num**	*exp.tipo* := *inteiro*
exp → **true**	*exp.tipo* := *booleano*
exp → **false**	*exp.tipo* := *booleano*
exp → **id**	*exp.tipo* := *verificar*(**id**.*nome*)

Verificação de tipos

- Exemplo de uma DDS para ve… t…

Regra gramatical	Regras semânticas
var-decl → **id** : *tipo-exp*	*inserir*(**id**.*nome*, *tipo-exp.tipo*)
tipo-exp → **int**	*tipo-exp.tipo* := *inteiro*
tipo-exp → **bool**	*tipo-exp.tipo* := *booleano*
$tipo\text{-}exp_1$ → **array** [**num**] **of** $tipo\text{-}exp_2$	$tipo\text{-}exp_1.tipo$:= *criaTipoNó*(*matriz*, **num**.*tamanho*, $tipo\text{-}exp_2.tipo$)
decl → **if** *exp* **then** *decl*	**if not** *tipoIgual*(*exp.tipo*, *booleano*) **then** *tipo-erro*(*decl*)
decl → **id**	**if not** *tipoIgual*(*verificar*(**id**.*nome*), *exp.tipo*) **then** *tipo-erro*(*decl*)
exp_1 → exp_2 + exp_3	**if not** (*tipoIgual*($exp_2.tipo$, *inteiro*) **and** *tipoIgual*($exp_3.tipo$, *inteiro*)) **then** *tipo-erro*(exp_1) ; $exp_1.tipo$:= *inteiro*
exp_1 → exp_2 **or** exp_3	**if not** (*tipoIgual*($exp_2.tipo$, *booleano*) **and** *tipoIgual*($exp_3.tipo$, *booleano*)) **then** *tipo-erro*(exp_1) ; $exp_1.tipo$:= *booleano*
exp_1 → exp_2 [exp_3]	**if** *éTipoMatriz*($exp_2.tipo$) **and** *tipoIgual*($exp_3.tipo$, *inteiro*) **then** $exp_1.tipo$:= $exp_2.tipo.filho1$ **else** *tipo-erro*(exp_1)
…**num**	*exp.tipo* := *inteiro*
…→ **true**	*exp.tipo* := *booleano*
…→ **false**	*exp.tipo* := *booleano*
exp → **id**	*exp.tipo* := *verificar*(**id**.*nome*)

Função tipoIgual faz a verificação estrutural / por nomes

Função éTipoMatriz checa se é uma matriz

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

Análise semântica

- 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 estática semântica entre linguagens
    - Não existe um mapeamento claro da representação para o algoritmo correspondente
  - Análise é artesanal, dependente da linguagem de programação

Fim