Linguagens de Domínio Específico

Fabio Mascarenhas – 2017.1

http://www.dcc.ufrj.br/~fabiom/dsl

Padrão *Visitor*

- A outra técnica para implementar um percurso consiste em usar o padrão *Visitor*

- A ideia é encapsular todo o percurso como uma implementação de um visitor para aquela AST, com cada tipo de nó sendo um método do visitor

- Os nós simplesmente implementam um método `visit(Visitor v)` que chama o método em v correspondente àquele nó, passando o próprio nó

```
/** A generic heterogeneous tree node used in our vector math trees */
public abstract class VecMathNode extends HeteroAST {
    public VecMathNode() {;}
    public VecMathNode(Token t) { this.token = t; }
    public abstract void visit(VecMathVisitor visitor); // dispatcher
}
```

Implementando o Visitor

- Note que o visitor usa sobrecarga, mas poderia usar nomes diferentes para cada método visit se a linguagem de implementação não tiver sobrecarga

```
public interface VecMathVisitor {
    void visit(AssignNode n);
    void visit(PrintNode n);
    void visit(StatListNode n);
    void visit(VarNode n);
    void visit(AddNode n);
    void visit(DotProductNode n);
    void visit(IntNode n);
    void visit(MultNode n);
    void visit(VectorNode n);
}
```

```
public class PrintVisitor implements VecMathVisitor {
    public void visit(AssignNode n) {
        n.id.visit(this);
        System.out.print("=");
        n.value.visit(this);
        System.out.println();
    }
```

Contexto e vantagens

- Para poder passar algum parâmetro de contexto para o visitor, assim como retornar um valor, podemos usar uma interface genérica

- Podemos até ter mais de uma, para visitors que não precisam de parâmetros, visitors com um parâmetro, visitors com dois parâmetros etc., à custa de precisar de mais métodos `visit` nas classes da AST

- Também podemos guardar informações de contexto na própria instância do visitor

- Apesar de mais complexidade e mais ruído nas chamadas recursivas, implementar um percurso com um visitor deixa o percurso mais fácil de escrever e manter

Amarração de nomes

- Depois da construção da AST, estamos prontos para fazer a *análise semântica* do programa, para poder detectar erros nele, e coletar informação para a construção do modelo semântico

- Uma informação essencial para qualquer análise e execução do programa é a *amarração* entre os usos dos identificadores do programa e suas declarações
  - Identificadores podem ser variáveis, funções, métodos, tipos, estados, ações...

- As regras da linguagem indicam qual o *visibilidade* de cada nome: em qual parte do programa aquele nome pode ser usado

Tabela de símbolos

- Fazemos a amarração dos nomes entrando e consultando nomes em estruturas de dados chamadas *tabelas de símbolos*

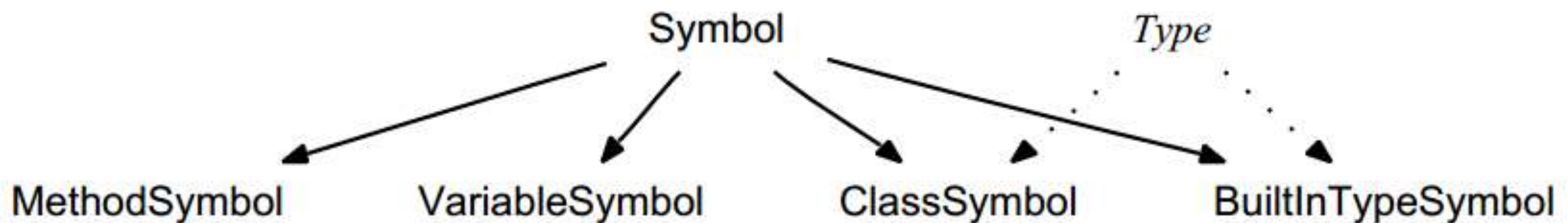
- A quantidade e organização das tabelas de símbolos do programa vai depender das suas regras de escopo

- Cada símbolo tem pelo menos um *nome* e uma *categoria* (função, variável etc.), e geralmente um *tipo*

- Em geral, entidades em categorias diferentes (variáveis vs. métodos, por exemplo) vão ter tabelas de símbolos distintas se o contexto do uso do nome já identifica a qual categoria ele se refere

Definindo símbolos

- Podemos modelar cada categoria de símbolo como uma classe




```
public class Symbol {
    public String name; // All symbols at least have a name
    public Type type;   // Symbols have types
}

public class VariableSymbol extends Symbol {
    public VariableSymbol(String name, Type type) { super(name, type); }
}

public class BuiltInTypeSymbol extends Symbol implements Type {
    public BuiltInTypeSymbol(String name) { super(name); }
}
```

Escopos

- Um escopo é uma região de código com uma fronteira bem definida que agrupa definições de símbolos

- Ex: o corpo de uma classe, o corpo de uma função, um bloco léxico, todo o programa

- Linguagens de programação geralmente possuem algum *aninhamento* de escopos, mas DSLs muito simples podem ter simplesmente um único escopo para o programa todo

- Geralmente a visibilidade de um nome está ligada ao escopo onde foi definido, mas nem sempre (campos públicos de uma classe, funções de um módulo)

Definindo escopos

- Cada escopo vai ser modelado pela sua tabela de símbolos, mais alguns metadados

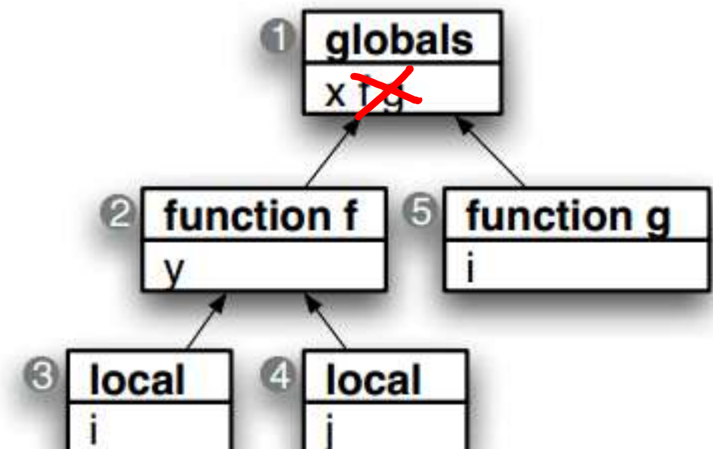
```
public interface Scope {
    public String getScopeName();         // do I have a name?
    public Scope getEnclosingScope();     // am I nested in another?
    public void define(Symbol sym);       // define sym in this scope
    public Symbol resolve(String name);   // look up name in scope
}
```

- Essa mesma interface vale para escopos de diferentes categorias, e escopos aninhados ou não

Árvore de escopos

- Em uma linguagem com escopos aninhados, todos os escopos do programa formam uma *árvore de escopos*

```
❶  // start of global scope
   int x;         // define variable x in global scope
❷  void f() {     // define function f in global scope
     int y;       // define variable y in local scope of f
❸    { int i; }   // define variable i in nested local scope
❹    { int j; }   // define variable j in another nested local scope
   }
❺  void g() {     // define function g in global scope
     int i;       // define variable i in local scope of g
   }
```



Resolvendo um símbolo

- Com escopos aninhados, a procura de um símbolo é uma função recursiva simples:

```
public Symbol resolve(String name) {
    Symbol s = members.get(name);   // look in this scope
    if ( s!=null ) return s;        // return it if in this scope
    if ( enclosingScope != null ) { // have an enclosing scope?
        return enclosingScope.resolve(name); // check enclosing scope
    }
    return null; // not found in this scope or there's no scope above
}
```

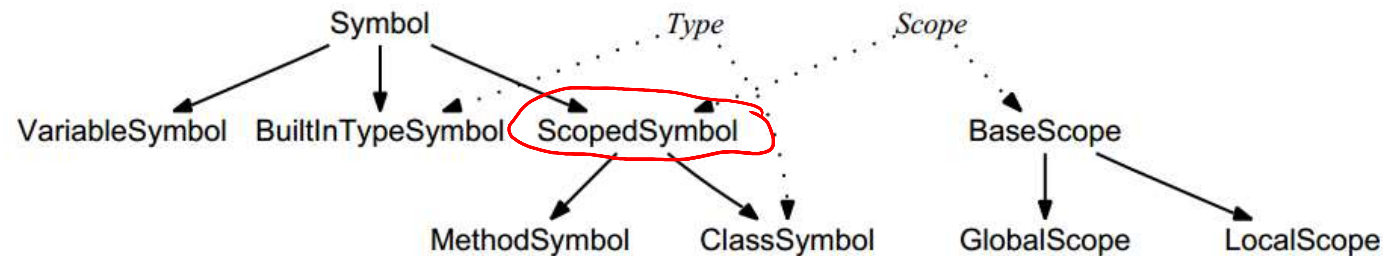
- A depender da linguagem, não achar um símbolo pode ser um erro ou não

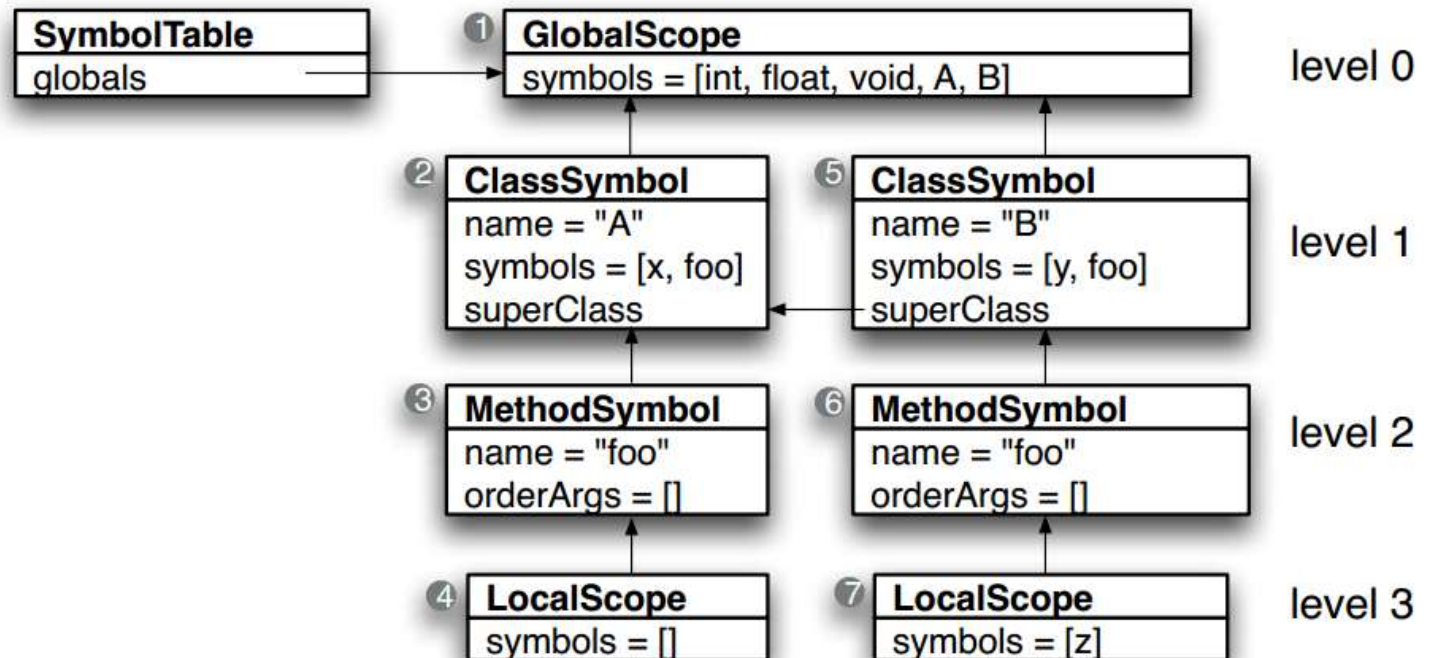
- Da mesma forma, definir um símbolo que já existe pode ser um erro ou não

Dados estruturados e escopo

- Várias linguagens possuem declarações de dados estruturados, como structs e classes

```
❶ // start of global scope
❷ class A {
  public:
   int x;
❸  void foo()
❹  { ; }
  };
❺ class B : public A {
   int y;
❻  void foo()
❼  {
     int z = x + y;
   }
  };
```

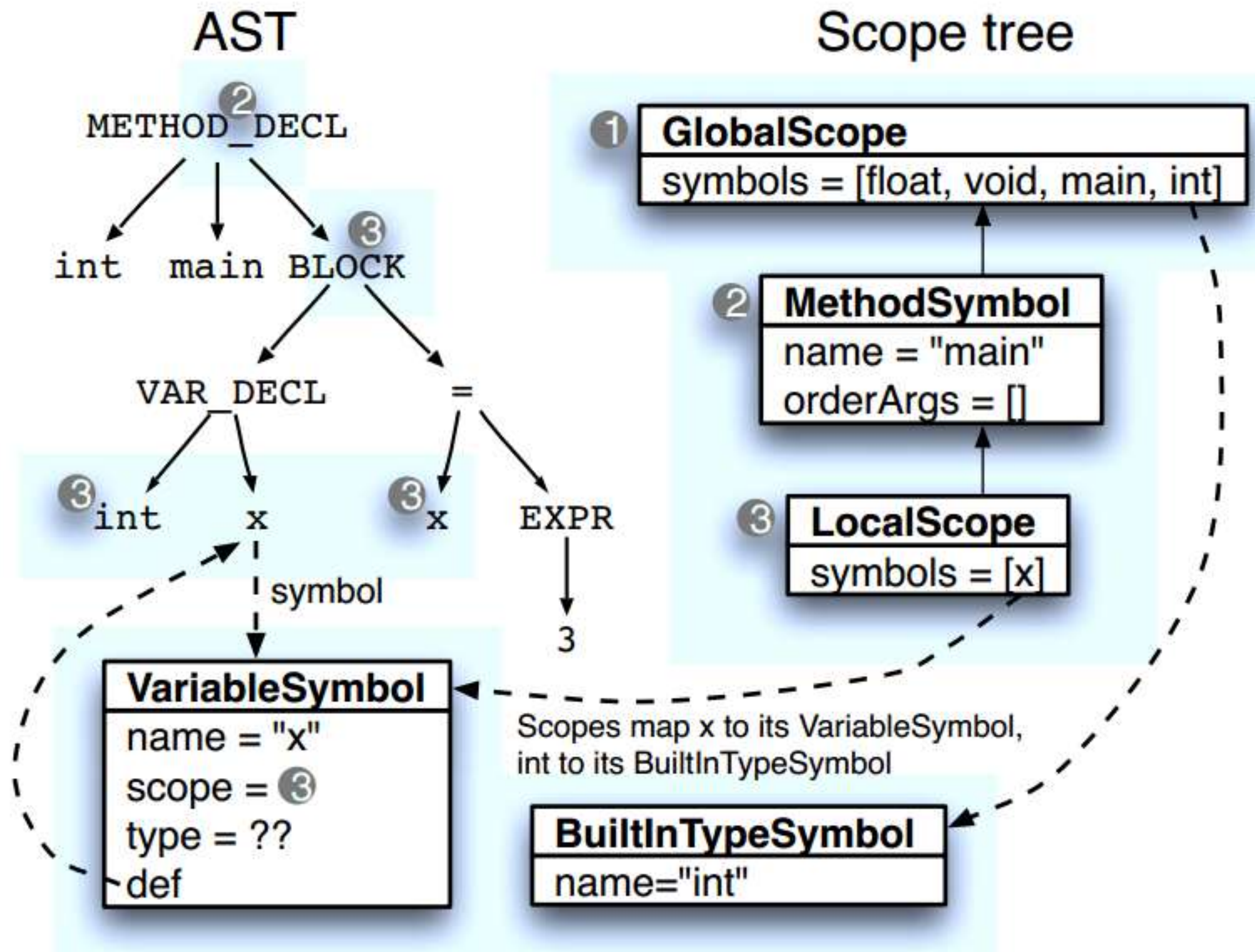




Visibilidade retroativa

- Também é comum que nomes sejam visíveis mesmo antes do ponto em que são declarados, como as classes e métodos em Java/C#
- Nesse caso, antes da análise semântica precisamos percorrer a árvore sintática fazendo apenas a construção dos escopos e coleta de todos os nomes definidos que têm visibilidade retroativa
- A árvore é anotada com esses escopos, assim fases de análise futuras já encontrarão todos os nomes com visibilidade retroativa já declarados
- Um truque para poder coletar todos os nomes nessa primeira fase é guardar a posição da declaração, e depois comparar essa posição com a posição do uso para nomes que não têm visibilidade retroativa

AST anotada – coleta



AST anotada – resolução